IMPLEMENTING FORTH

by

Christopher T. Carson

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
the Department of Computer Science

Winnipeg, Manitoba

ISBN 0-315-33978-0

IMPLEMENTING FORTH

BY

CHRISTOPHER T. CARSON

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Christopher T. Carson

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Christopher T. Carson

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

ABSTRACT

Forth is an interactive, extensible, self-contained programming language system. Its unorthodox syntax and high degree of freedom have made it a rebel amongst more traditional languages. Forth has nonetheless proven to be valuable for use in real-time process control applications.

This thesis discusses the nature of Forth and concentrates on the language implementation. A machine-independent model is presented in terms of Forth and a generic assembly language. Machine specific details are provided where necessary. The later chapters explore the possibility of direct-execution Forth processors and other enhancements to the Forth language and development environment.

ACKNOWLEDGEMENTS

Many thanks to my supervisor, Mike Doyle and to Peter Somers, Howard Ferch and Don Marcynuk for their helpful comments and criticisms.

In addition, a debt of gratitude is owed to the Natural Science and Engineering Research Council for their financial support during my research.

CONTENTS

LIST OF FIGURES

Chapter I

INTRODUCTION

The main goal behind the development of Forth was to increase programmer productivity [Moo80]. This goal has become increasingly important due to the rising cost of software [Sho83]. The need for proper software development tools is reflected by the growing interest in syntax directed editors, source level debuggers, and language systems such as Smalltalk-80 [GoR83] and the R1000 Ada Development System [Car85].

Another goal of Forth was to minimize costs in terms of hardware - specifically memory requirements. In the past, memory was considered a precious commodity - much time was spent trying to minimize program size and make efficient use of memory through overlay techniques. Forth was able to drastically reduce memory requirements by using a threaded interpretive language (TIL) implementation and a small RPN (reverse polish notation or postfix) compiler.

The purpose of this thesis is to discuss the implementation of Forth. Attention is given to the language and various methods of implementing its intermediary language interpreter.

1.1 OVERVIEW

Forth was developed by Charles H. Moore over a period of several years. An early version having most of the essential features was implemented in 1968 on an IBM 1130. It was the five-character limitation of this machine which resulted in the name FORTH - an abbreviation of FOURTH, meaning fourth generation computer language. It wasn't until 1971 that a true Forth was installed on a Honeywell H316 at the National Radio Astronomy Observatory in Charlottesville Virginia.

Moore has said that "At no point in time did I sit down to design a programming language. I solved the problems as they arose." [Moo80] This approach, together with Forth's early use in real-time telescope control, resulted in its development as an interactive, extensible programming language. Forth provides an interactive environment in which the language (operators, data types and control structures) can be extended to suit the application - unlike traditional languages which provide a fixed set of features.

Forth is much more than a programming language. It is a language system - an amalgamation of a high level language, job control language, and operating system [Pay84]. This integrated software environment is characterized by its dictionary, stacks, interpreters, assembler, and virtual memory, all of which interact to produce a synergistic effect

that creates a powerful and flexible programming system [RaM76].

The Forth language is based on a dictionary of words[1]. Each word gives the name of a Forth procedure whose definition is kept in the dictionary. Originally the dictionary contains a set of kernel words (basic operators and control structures). User defined words are added to the dictionary and become a part of the language as well as an extension to it.

The dictionary is organized into several vocabularies each containing an ordered list of word definitions. Vocabularies such as FORTH (containing kernel words), ASSEMBLER (assembler words) and EDITOR (editor words) are available to the user. Additional vocabularies may be defined as required. These are useful for grouping related words together. The user can switch from one vocabulary to another in order to select a specific portion of the dictionary (such as words needed only for editing).

Words and word names are synonymous. A word refers to a Forth procedure. A word is any string of non-blank characters (there are no other restrictions). Source code consists of words delimited by white-space. A number is a string of digits that form a word not found in the dictionary.

---

[1] Refer to the FORTH-83 Standard [FST83] for a summary of standard word definitions.

Arguments are passed between words via the data stack. This results in the natural use of postfix expressions such as 2 2 + which calls the word '+' (meaning "add"), preceded by its two arguments. This expression will push the numbers onto the stack[2] and execute '+' which pops the two numbers, adds them, and pushes the result (4) onto the stack.

All words adhere to this linkage convention (i.e. they pop their arguments and push their results). Intermediate results are usually kept on the stack and manipulated using words like SWAP (reverse order of top two stack elements), DROP (discard top stack element), DUP (push copy of top of stack onto stack), and OVER (push copy of second stack element onto top of stack). Another stack, known as the return stack may also be used to store intermediate results although its main function is to save return addresses for nested words.

Since Forth is interactive, like APL, an expression can be executed simply by typing the expression followed by a carriage return. Forth's text interpreter will execute the expression from left to right by invoking a specified word or pushing a given number onto the stack. A word is invoked by locating and executing its definition (as found in the dictionary). For example:

---

[2] When the name "stack" is used alone, it implies the data stack.

```
11 17 * OK[3]                    ( calculate 11 x 17 )[4]
DUP * 2 + . 34971 OK             ( square this and add 2 )
```

The word '.' (pronounced "dot") is used to pop and print the top stack entry. Note that the expression could be entered as one or many lines. Forth, like Pascal, is format free.

New word definitions are added to the dictionary by using one of several defining words. For instance, the word ':' (pronounced "colon") is used to define new words in terms of existing ones. Thus the statement

```
: PHOTOGRAPH SHUTTER OPEN TIME DELAY SHUTTER CLOSE ; OK
```

defines a new word named PHOTOGRAPH. The semicolon marks the end of the definition. The text interpreter recognizes ':' and compiles the definition into the dictionary. PHOTOGRAPH can now be used in subsequent definitions or can be executed by itself. The words SHUTTER, OPEN, TIME, DELAY, SHUTTER and CLOSE are invoked in that order whenever PHOTOGRAPH is executed.

A colon definition locates its component words and compiles a list of their dictionary addresses (this list of addresses is known as threaded code). The text interpreter is able to interpret a word definition by calling the address interpreter. The address interpreter then executes the

---

[3] Underlined text indicates output generated upon entering a carriage return. This convention is used to separate user input from system output. The message 'OK' is printed by the text interpreter to indicate that the expression was complete and was successful.

[4] Comments are placed within brackets as in (...)

threaded code by invoking each component word as given by its address. The effect of this is shown in the following example:

```
: SQUARED                          ( n1 -- n2 )[5]
   DUP * ; OK                      ( define the word SQUARED )
                                   ( which calls DUP, then * )
12 SQUARED . 144 OK

: CUBED ( n1 -- n2 )
   DUP SQUARED * ; OK              ( define CUBED in terms )
                                   ( of SQUARED and * )

3 CUBED . 27 OK
```

Thus words can be built into a hierarchy of subroutine calls. Note the use of DUP to make a copy of the input parameter to CUBED.

Two other defining words are CONSTANT and VARIABLE. CONSTANT is used to define a word which returns a constant value:

```
5 CONSTANT MAX OK                  ( define MAX as a constant )
MAX . 5 OK
```

VARIABLE is used to allocate a storage location which can then be accessed using the words '!' (pronounced "store") and '@' (pronounced "fetch"):

```
VARIABLE APPLES OK                 ( define APPLES as a variable )
10 APPLES ! OK                     ( store value of 10 )
APPLES @ . 10 OK                   ( fetch and print value )
APPLES @                           ( add MAX to APPLES )
MAX + APPLES ! OK                  ( this sets APPLES to 15 )
```

---

[5] The notation (before -- after) is used to specify stack parameters input to and output from a word at execution time. Before is a list of stack parameters before the word is executed. After shows the stack after execution. The top stack entry is the rightmost parameter. For example, OVER (n1 n2 -- n1 n2 n1). All words expect and return 16-bit values unless otherwise noted.

It should be noted that many word definitions do not require the use of VARIABLEs. The data and return stacks are usually sufficient for storage and manipulation of parameters and temporary results.

In addition to CONSTANT and VARIABLE, Forth provides the ability to create new defining words. This enables the programmer to define whatever data types the application requires (e.g. matrices, queues, linked lists). The language can also be extended to include application dependent control structures such as case statements or multiple exit loops [Har80]. IF-THEN-ELSE, WHILE, UNTIL and DO-LOOP constructs are already provided.

Forth also provides an assembler language specific to the host machine. Words coded in Forth or assembler can be intermixed. This feature allows selected high level Forth definitions to be redefined in assembler if increased execution speed is necessary.

The last component of Forth is its virtual memory. FORTH implements a block storage system. Fixed sized blocks are stored on disk and are automatically swapped between disk and memory (block buffers) as needed. A block is referenced by its block number, which is directly related to its physical address on disk. This standard block interface allows high-level Forth applications to be easily ported from one machine to another [RaM76].

Blocks are used to store source code and data. Source can then be edited, saved to disk, or compiled into memory from disk. It is important that source be kept on disk since definitions entered at the keyboard are immediately compiled and the source is lost. This is not a problem since definitions can be edited within block buffers and then compiled almost instantaneously. As an alternative, some systems provide a decompiler which allows source code to be regenerated from compiled code.

The input stream to the text interpreter can be switched between disk and keyboard to allow definitions or expressions to be entered at the keyboard or input from disk.

Data may be stored as several records per block, or may span several blocks per record. The programmer is free to allocate and use blocks as he wishes.

1.2 ADVANTAGES

Forth has many advantages - some are a result of its implementation, while others are inherent to the language. As a threaded interpretive language (TIL), Forth makes efficient use of memory. The entire system (interpreters, assembler and operating system) typically occupies less than 8K bytes and can therefore be kept in memory at all times. Source code is translated into a compact form known as threaded code (TC) in which each subroutine call requires

only two bytes. This can drastically reduce code size - especially since Forth definitions consist mostly of words which are, in effect, subroutine calls. A traditional compiler would generate a 3-6 byte subroutine call instruction versus two bytes of TC. Thus TC provides a 33-67% reduction in code size.

As a TIL, Forth is also fast. Benchmarks have shown TC, versus assembly code, to run in the neighborhood of twice as slow on eight-bit microprocessors, and 20% slower on minicomputers [Kog82]. This is an order of magnitude improvement over conventional interpreters (such as those for BASIC), and compares favorably with other high level language compilers which generate code that runs in the range of 10-100% slower [Loe81].

Forth's use of stacks offers several advantages. The stack simplifies subroutine linkage and reduces subroutine call overhead. This affords the use of many subroutines. The stack also provides dynamic storage for local variables. This results in the natural ability to create recursive programs (a program may call itself) and re-entrant programs (a single copy can be simultaneously shared by several users). The compiler and interpreter are simplified by using the return and data stacks for compile-time calculations, temporary iterative loop variables, and evaluation of postfix expressions. The use of postfix notation itself eliminates a significant amount of syntactic analysis (which is required by other interpreters like BASIC and APL) [Kog82].

Forth promotes structured and highly modular programming. Short, easily understood procedures (less than twenty words) are built into a hierarchy of subroutine calls which communicate via a well defined interface - the stack. This results in modules with the desirable qualities of high cohesion and low connectivity. Short definitions reduce the number of possible execution paths. Individual words can be quickly tested by placing arguments onto the stack, invoking the word, and checking the stack for proper results. By building and testing word upon word in an interactive environment, side effects are minimized to produce remarkably bug-free programs [Jam80]. Top-down design and bottom-up testing are encouraged.

Forth is an interactive programming environment. Source code is incrementally compiled or assembled as entered and is immediately ready to run (translation from source to object code[6] is transparent to the user). This speeds development by reducing the compile time and eliminating the link phase required by many other languages. Rapid prototyping is facilitated.

Forth provides an extensible high level language. This allows the user to define new data and control structures. These abstractions become a part of the language. There is no distinction between levels of abstraction because all words operate using the same protocol. This contributes

[6] Object code refers to both threaded code and machine code.

greatly towards the integration of system components.

Forth is portable since it is largely written in itself. It has been implemented on almost every micro/mini computer ever made[7] [Moo80]. Forth's portability helps protect the investment in software and development tools by allowing them to be migrated to new machines, with a minimum of effort.

Forth provides an integrated virtual memory mechanism and assembler language. Virtual strings and other "virtual" data types can be easily defined. High level language definitions can be directly replaced with assembler routines if additional speed or machine-dependent code is required.

Forth can be run as a stand-alone operating system or it can be run as a task under another operating system. It can also be target-compiled to produce a minimal system containing only the routines needed by a particular program. This facilitates the easy development of ROM-based application code [Pay84].

---

[7] See appendix "Target Machines".

1.3 DISADVANTAGES

Forth leaves itself wide open to criticism in several areas. These include poor readability, lack of type or range checking, and a primitive file system. The use of postfix notation has always been cause for dislike. This, together with the contorted use of stack operations, can produce what has been referred to as a "write-only language" - even more so than APL [Wil80]. The use of meaningful names and proper documentation are a must.

Forth can be especially hard for the beginner. It takes a while to learn to "think in Forth" - to be able to keep track of parameters and temporary values as they are manipulated on the stack. This source of frustration does disappear with experience and through the use of short definitions which minimize visible stack depth.

The potential for unreadable code and the long learning curve pose the danger of non-maintainability [Jon86]. Proficient Forth programmers are few (but growing) in number. There is a general disinterest in Forth and other "non-standard" languages which are "not worth learning" (as opposed to C and Pascal). These factors combine to increase the risk that the author of a Forth program will be the only person easily capable of maintaining that program. In some cases the author may find his own code unreadable after a period of time.

Forth does not provide any type or range checking [Jam80]. It is up to the programmer to add whatever forms of error checking are deemed necessary (validating arguments, subscript ranges, etc). The programmer must also ensure that each word maintains stack balance (in order to preserve the integrity of the stacks). Forth gives complete access to the machine through fundamental operators like '!' ("store"), which allows any RAM location to be modified. These "freedoms" place total responsibility on the programmer.

> "Forth allows the programmer to make all sorts of terrible mistakes and poor choices. I like to say that Forth is like a high-performance Jet Fighter: if you know what you're doing you can do some amazing things... If you don't, it's a good way to get yourself killed" [Lin84].

Put another way,

> "Forth is an amplifier. A good programmer can do a fantastic job with Forth; a bad programmer can do a disastrous one" [Moo80].

Forth programs are dialectic. Programmers often tailor the language to their own way of thinking. This improves individual productivity but may pose problems in a multi-programmer environment. The solution is to enforce the use of standard words and coding practices in order to produce consistent and readable code.

Some versions of Forth, such as indirect token threaded code implementations (see section 4.6), suffer from slower execution speed. These may be in the range of 5 to 10 times slower than assembler or native code language compilers.

Forth cannot easily be used in mixed language programming (other than its own assembly language). This is a result of Forth's dependency on its environment. A Forth program requires most or all of the Forth system components. At the very least, this means that dictionary entries and the address interpreter must be included (often at a fixed address) within the load module. Interlanguage calls are convoluted at best; restricting the user to a Forth-only development environment.

Another disadvantage is the primitive file system. Forth lumps its file system and virtual memory into a block storage system. Source is loaded by specifying a block number. There is no concept of a directory with named files. A block of source code contains 16 lines of 64 characters each. This fixed format requires unused areas to be blank filled - often wasting a large percentage of disk space.

No floating point, string, array, record or local variables are provided. All word definitions are more or less globally accessible. These, along with most other disadvantages, can be overcome by extending the language. Enhancements such as named parameters, infix notation, and local variables will be discussed later.

1.4 USES

Forth has been called a meta-application language. Its extensibility allows it to be adapted to a wide variety of areas. Forth is particularly well suited to interactive real-time applications such as process control or data acquisition and analysis. Examples of Forth in the real-world include:

Bell Canada's Plant Administration System II (PAMS II) which was developed using polyFORTH.[8] PAMS II is used to locate and analyze problems in Bells' telephone switching network. PolyFORTH II was chosen for its ability to support 32 terminals and a 200MB database on a single 68000-based system, with response times averaging under one second. No other operating system was able to match these requirements.

PolyFORTH II is currently being used in the development and installation of a large-scale Monitoring and Control System. This distributed system utilizes 8 PDP-11/44's, 320 custom 8085-based security panels and 378 custom 8086-based machines with up to 36,000 sensors (17,000 are in use). The system incorporates fire and safety monitoring, security and access control, heating, ventilation and air conditioning, runway lights plus water, fuel and power distribution [Rat85].

---

[8] PolyFORTH II is a professional multitasking multiprogrammed FORTH system available from FORTH, Inc.

Network communications are based on a variation of the clusterFORTH communications protocol [BSR84]. The polyForth Data Base Support System, an extensive graphics package and memory management techniques are also in use.

The Elicon Special Effects Camera Robot is a multi-axis robot used to create high quality special effects shots [Sla83]. The robot "flies" the camera past miniature scale models of spacecraft, starfields and other scene elements. These elements are superimposed to create the effect image.

Forth was extended to produce a special film language. This language allows the user to define "flight paths" according to the relative position and angle of view as seen from a moving object. The control program performs the necessary geometry transformations and real-time control of the robot servo motion and camera. This system has been used for special effects in dozens of movies and commercials.

A similar application is that of Skycam [Con85]. Skycam is an aerial robotic camera system. Forth was again chosen as the development language.

Other Process Control Applications include a peach sorter [Wil80], baggage handler [Bro81], thyristor laser beveling system [Ant84], gyro rotor cleaning assembly [Dum83] and telescope control [Cha83].

GameForth is one of several versions of Forth which have been customized for a specific application. Atari Inc. developed gameForth for creating arcade games [Wil80]. GraForth is another graphics-oriented Forth that runs on the Apple II series. GraForth provides an image editor and functions for manipulating 3-D objects, text, and graphics.

DELTA is an expert system for diesel electric locomotive repair. This rule-based system is designed to aid maintenance personnel in problem diagnosis and repair procedures. A prototype system has been implemented in Forth by creating a special representation language. This language contains predicate functions and verbs used to express the rules. The inference engine then uses these rules to evaluate facts.

A feasibility prototype was implemented in LISP and then re-written in Forth to achieve "efficiency and portability to other microprocessor-based systems" [JoB83].

Data Acquisition and Analysis for Scientific Instruments such as a real-time interactive spectroscope which uses Forth for acquisition, reduction and manipulation of data from 500 channels [BoF83].

Control and Image Processing for a "passive ultrasonic microscope" based on a mechanically scanned ultrasonic imaging system [LyJ85].

Portable Intelligent Devices like the Craig M100 Language Translator and a portable Cardiac monitoring device, both of which run Forth internally [Wil80].

1.5 FORTH VS OTHER LANGUAGES

Forth exhibits some of the properties of a functional programming language. Words can be considered functions as long as each word adheres to passing arguments and results through the stack only. In fact FORJR, a version of Backus' FP, has been implemented using Forth [DEF84].

An object oriented system called NEON [DuI84] (a subset of Smalltalk-80) is also been written in Forth. Similar object oriented extensions to Forth are described in [Pou86]. Other languages implemented in Forth include Lisp, Prolog and OPS5.

As a language, Forth has more disadvantages than advantages when compared against its two most popular contenders: C and Pascal. Forth has the advantage of being extremely flexible. It gives the programmer complete control of the machine and is extensible at all levels. Other languages are more protective and provide extensibility through subroutines, arrays and record data structures. Language features cannot be extended as may be done in Forth.

Pascal provides strong type checking and is able to detect many subtle errors. C has less checking and Forth has

none. However, C and Forth are more flexible than Pascal. C and Forth are "closer to the machine". This allows them to be used in a wider range of applications.

Pascal has a richer set of data types than C and Forth. Pascal includes boolean, character, integer, real, pointer, enumerated and set data types. C provides character, integer, pointer and real while Forth provides only character and integer. Forth and C use integers to represent boolean data. Integers can be 16 or 32 bits (this may vary in some C versions). Forth also uses integers as pointers. Forth's disadvantage of fewer data types is offset by its ability to create new ones.

C and Forth provide a large number of extraordinary operators or expressions. C has eleven levels of operator precedence, Pascal has four and Forth has only one. A single precedence level leaves no doubt as to the result of an expression, however, it restricts the order of operations.

The Pascal syntax is very readable, C is less readable (address/pointer expressions can be difficult) and Forth has been called unreadable. Forth's poor readability can be attributed to its postfix syntax and lack of named parameters and local variables.

All three languages have reasonably good control structures. Forth allows new control structures to be defined if necessary. Pascal has the disadvantage of not being able to

break as easily from within a loop. This forces the programmer to create boolean flags that control loop termination.

Forth has virtually no method of scope control. This disadvantage forces the use of global variables and procedures. C uses source files to define scope but cannot nest procedures as can Pascal. However, Pascal enforces a strict ordering amongst program sections. This forces containing procedures to be subdivided by their nested procedures.

Forth procedures have an advantage in that there are no restrictions on the number or type of arguments being passed or returned. C and Forth procedures can accept a variable number of parameters; Pascal cannot.

Each language makes tradeoffs between flexibility and strong typing; syntax and clarity versus compactness and expressive power. Forth has chosen flexibility and power; a different style and method of abstraction which has its own strengths and weaknesses.

Forth is a high-level language that allows the programmer to extend the machine to fit the application. The Forth language can be grouped amongst other languages as shown in figure 1.1 [Bra85b].

Forth provides a set of operators, system variables, control structures and defining words that form the basis of

Non-extensible machine language or micro-code
   Non-stack oriented programming
   Many programmers operate at this level even if the computer supports stacks

Extensible machine language -- Forth
   Two stacks, micro-codes higher level facilities

Traditional Algebraic -- Fortran, Pascal, C, ADA
   Both data structures and control must be planned
   Usually single stack implementation
Dynamic memory allocation -- Lisp
   Data structuring automatic, control must be planned

Dynamic control -- Prolog, Expert Systems languages
   Both data structures and control are automatic
   Forward versus backward chaining
   Probabilistic control often used

Figure 1.1: Programming Language Hierarchy

the language. The last three items are discussed in subsequent chapters. Operators are discussed here since they are required throughout the remainder of this thesis.

Each operator is a word which performs a simple function. Most operators expect and return 16-bit values. 32-bit operators can be recognized by their 'D' or '2' prefixes as in D+, D- and 2DUP. 8-bit operators act on the least significant byte of a 16-bit entry. These operators are prefixed by 'C'.

Binary operators take the top two stack entries and push their result. For example, AND performs a logical bitwise and; >= performs a relational test which returns zero if false and minus one if true.

Unary operators act only on the top stack entry. 1+ adds one to the top stack entry. 2+, 1- and 2- perform other increment and decrement operations.

Stack operators (DUP, SWAP, etc) are used to manipulate the data stack. Interstack operators such as R> and >R are also provided. R> pops the top return stack entry to the data stack. >R performs the reverse function.

Memory reference operators allow access to 16-bit (@, !) and 8-bit (C@, C!) data. The remaining operators perform system, I/O and utility functions.

1.6 THESIS OUTLINE

The first chapter provides an overview of Forth together with its advantages, disadvantages and uses. This is intended to give the reader a sense of the language and its capabilities.

Chapters two and three together discuss the implementation of a FORTH-83 standard system. Chapter two deals with system components: the dictionary, stacks, interpreters and block storage system. Chapter three presents language elements: literals, defining words, control structures and the assembly language.

The fourth and fifth chapters describe various methods of implementing Forth's underlying virtual machine or address

interpreter. Chapter four discusses software implementations of the virtual machine. This includes threaded code and other techniques. The fifth chapter continues by discussing implementations of the virtual machine in hardware.

The concluding chapter looks at possible enhancements to Forth and the trend toward more powerful Forth development systems.

Chapter II

SYSTEM COMPONENTS

This chapter takes a closer look at Forth system components. It discusses the implementation of the dictionary, interpreters, block storage system and related words which integrate these components into the language.

A subset of the FORTH-83 Standard words are used to illustrate implementation details while also providing a language tutorial. This material is sufficient, together with the next chapter ("language components") and the FORTH-83 Standard [FST83], to allow the reader to implement a complete single-user FORTH-83 system. It may also be used as the basis of a customized Forth-like language.

2.1 FORTH-83 STANDARD

Forth's extensibility promotes "customization" of the language to the requirements of a particular application. This has led to the existence of many versions of Forth, and the need to standardize the language in the interests of software portability. The FORTH Standards Team has most recently published the FORTH-83 Standard (August 1983) which is adhered to in this thesis.

The FORTH-83 Standard specifies a set of required words which must be available in a standard system. Standard programs can then be defined using these words. Each required word is specified by name, name pronunciation, and function. The function describes input and output parameters plus other effects the word may have upon system variables. Possible error conditions are also noted.

Several extension word sets are provided to facilitate options such as 32-bit arithmetic and assembly language programming. A standard system need not provide these features. In fact, a standard program can not include assembly code since this would be machine dependent. Controlled reference words (being considered for standardization) and uncontrolled reference words (which are presently in use on some systems) are also included. The final section describes experimental proposals which may be adopted in future standards.

2.2 THREADED CODE

Forth has traditionally been implemented as a threaded interpretive language (TIL). TILs translate source code to an intermediary form known as threaded code (TC). There are several types of TC, all of which are functionally the same. Indirect threaded code (ITC) will be used for illustrative purposes since it is probably the most common type used to implement Forth. Other types of TC will be discussed later.

A change in the type of TC mainly effects the address interpreter (also known as the inner interpreter). The address interpreter acts as a virtual machine which interprets or "executes" TC "instructions". Each "instruction" corresponds to a word definition. In ITC, an "instruction" is represented as the address of a subroutine which performs the function of its associated word. Other types of TC may represent subroutine calls in a different manner. In any case, the address interpreter steps through the TC - executing each subroutine it encounters.

If you think of a hierarchy of subroutine calls, you can imagine an execution path which is "threaded" throughout the structure (see figure 2.1 which illustrates the basic structure of word definitions). The thread follows the path of execution - hence the term threaded code.

If you view this structure as a tree then the leaf nodes are called primitives and the others are called secondaries. Primitive words are defined in assembly language. Secondary words are defined in terms of primitives, secondaries, or both. The address interpreter must distinguish between primitives (which are directly executed) and secondaries (which require further interpretation).

Note that the primitives do all the real work. In referring to figure 2.1, the thread of execution follows a preorder traversal of the tree, executing DUP DUP * * which

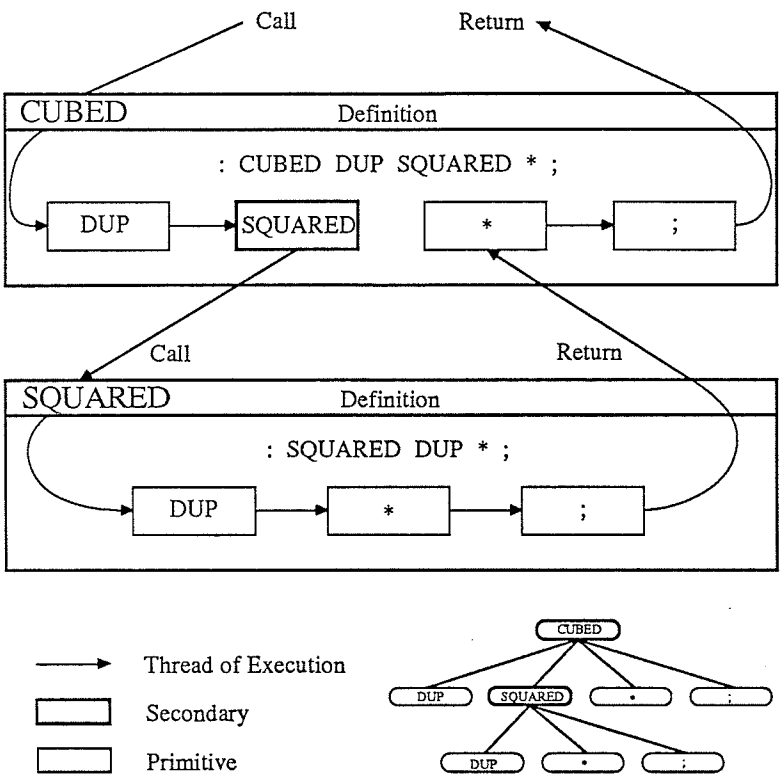


Figure 2.1: Thread of Execution in a Hierarchy of Words

is the function of CUBED. Thus primitives form the basic "instruction" set of the virtual machine. Secondaries then allow the implementation of high level, extended "instructions".

2.3 DICTIONARY

The dictionary is system-wide data structure used to maintain word definitions. It is usually stored as a linked list with each node (dictionary entry) representing the definition of a single word. A dictionary entry contains the

name of the word, its body (threaded code, machine code or data) and a pointer to the previous entry (see figure 2.2).

The dictionary is used by the text interpreter for locating existing word definitions and compiling new ones. The function of a word is determined by the code in the body of

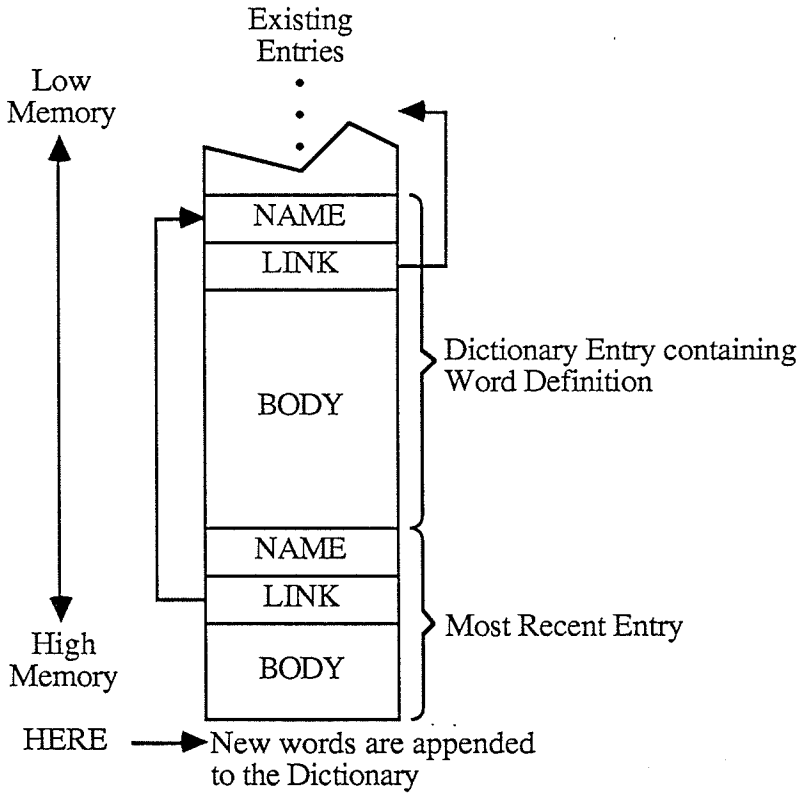


Figure 2.2: Basic Dictionary Structure

its definition.

The name field begins with a byte containing the length of the word name. The most significant bit of this byte is used as a flag whose purpose will be discussed later. In

some systems, a fixed length name field is used to store only the length and the first three characters of a word name. This saves space but increases the probability of a conflict in names. For this reason, FORTH-83 specifies that word names may be up to 31 characters in length - this necessitates the use of a variable length name field. The typical structure of a dictionary entry is shown in figure 2.3.

Fixed length name fields have traditionally been placed at the start of each dictionary entry, followed by the link field. With a variable length name the order is often reversed by placing the link field at the start. A dictionary search can then proceed faster because the link field is always at a fixed location within each entry.

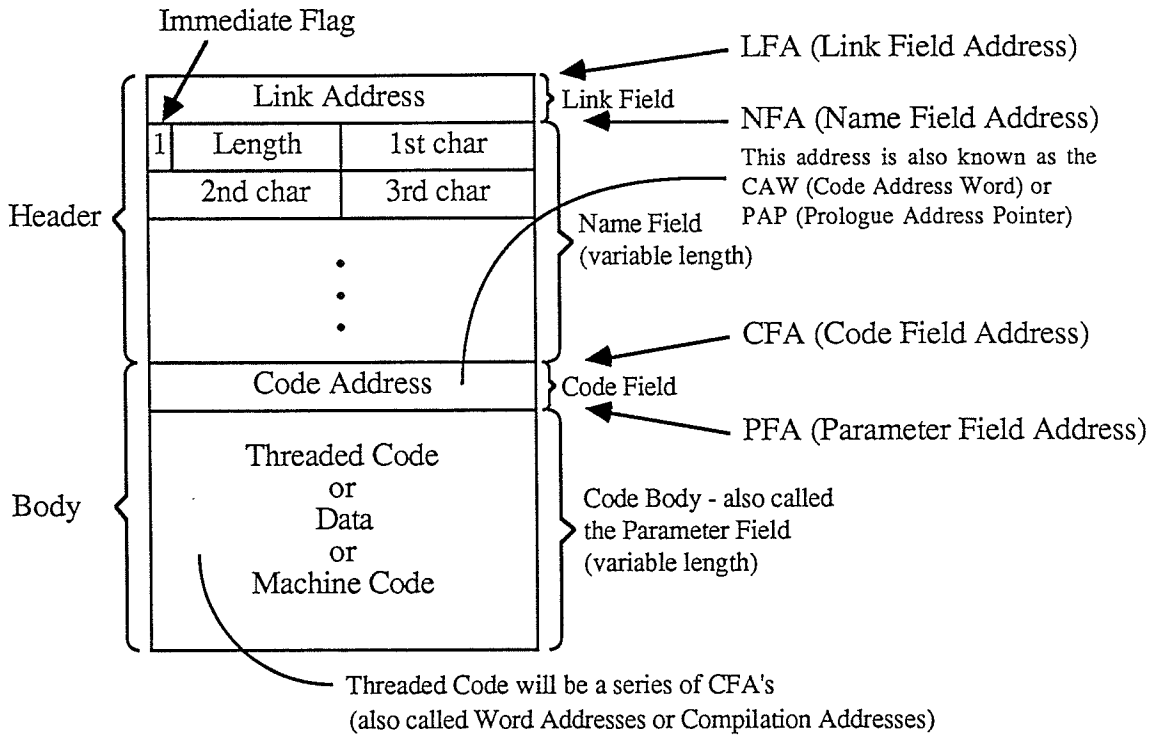
The FORTH-83 Standard does not describe the internal structure of the dictionary. It merely defines the dictionary as:

> "A structure of word definitions in computer memory which is extensible and grows toward higher memory addresses. Entries are organized in vocabularies to aid location by name" [FST83].

The details of dictionary implementation are irrelevant provided it functions properly in conjunction with standard words which access or alter it.

The dictionary grows toward higher memory addresses as word definitions are added. Forth maintains a dictionary pointer (DP) which points at the next available dictionary

Figure 2.3: Dictionary Entry Structure



DICTIONARY HEADERS

Variable Length

Link Address	
0A	P
H	O
T	O
G	R
A	P
H	

Fixed Length

0A	P
H	O
Link Address	

Does not uniquely identify words such as:

phoenician
photometry
phosphoric

location. DP is usually stored in a system variable but may be kept in a dedicated register. Several words directly access or manipulate this dictionary pointer. Examples include:

Word	Function
HERE	returns the address of the next available dictionary location. (The value of DP.)
n ALLOT	allocates n bytes from the dictionary. (Increments DP by n.)
n ,	encloses n in the dictionary (allocates 2 bytes and stores n at HERE - 2). Defined as : , HERE 2 ALLOT ! ;
n C,	encloses the least significant byte of n in the dictionary (allocates 1 byte and stores n at HERE - 1). Defined as : C, HERE 1 ALLOT C! ;
FORGET <name>	searches the compilation vocabulary (see next section) for <name> and sets DP to point at the entry for <name>.

FORGET provides a simple and effective way of managing the dictionary. FORGET <name> will delete <name> from the dictionary along with all words added after <name>. It is common practice to place the statements:

```
FORGET FENCE
VARIABLE FENCE
```

at the start of an application. This automatically releases the previous application and marks the beginning of the new one being compiled. FORGET issues a warning message if FENCE is undefined.

2.3.1 Vocabularies

A vocabulary is "an ordered list of word definitions" [FST83]. Vocabularies are useful in grouping together related words and in controlling the order in which the dictionary is searched. Word definitions with the same name can be placed in separate vocabularies and still be uniquely identified by the search order. An existing word may be defined more than once within the same vocabulary. In this case, only the most recent definition will be found when the vocabulary is searched.

The search order is:

> "A specification of the order in which selected vocabularies in the dictionary are searched. Execution of a vocabulary makes it the first vocabulary in the search order. The dictionary is searched whenever a word is to be located by its name. This order applies to all dictionary searches unless otherwise noted. The search order begins with the last vocabulary executed and ends with FORTH,[9] unless altered in a system dependent manner" [FST83].

A mechanism for clearly specifying the search order has not been standardized. The most common approach is the use of chained vocabularies as shown in figure 2.4. Each vocabulary is defined as in VOCABULARY <name>, which gives the name of the vocabulary and a defines a word <name> that contains a pointer to the most recent definition in the vocabulary. When <name> is executed, it updates the system variable CONTEXT. This causes the search order to begin with

[9] FORTH is the name of the primary vocabulary which contains kernel words.

<name>. The search order proceeds from the last vocabulary executed and continues until "end-of-dictionary". In referring to figure 2.4, if the CONTEXT vocabulary was CAD, the

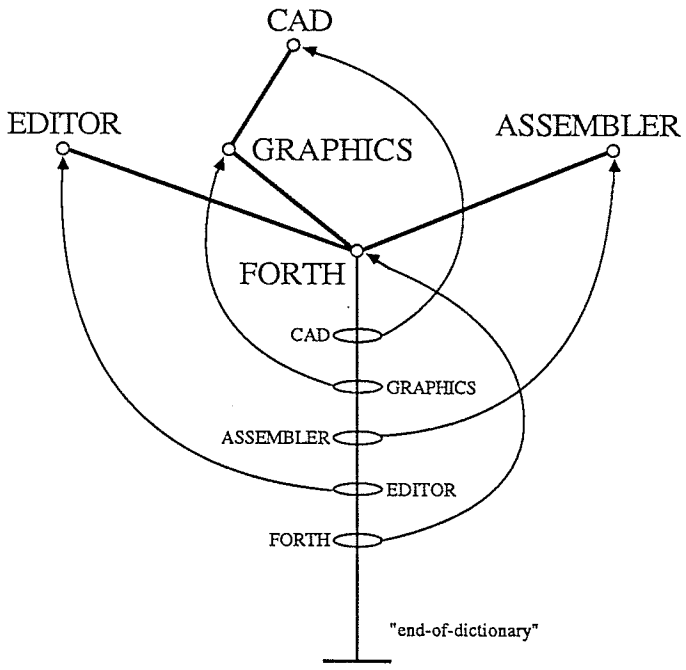


Figure 2.4: Chained Vocabularies

search order would be CAD-GRAPHICS-FORTH.

Dictionary entries are back-linked[10] so that the first word in a vocabulary will be the most recent definition of that word. New words are appended to the dictionary and are thus maintained in chronological order from newest to oldest.

---

[10] Structures other than a linear linked list may also be used as discussed in section 6.1.1.

Redefining a word has no effect on definitions which were previously compiled. Prior definitions will continue to use the most recent version of each word as was available at compile-time. In practice, the entire application is re-compiled whenever necessary. This "catch-all" procedure takes only seconds to ensure that all definitions are up-to-date.

The system extension word set specifies two system variables which control use of the dictionary. CONTEXT determines the dictionary search order. CURRENT specifies the vocabulary in which new word definitions are appended (this is referred to as the compilation vocabulary). With chained vocabularies, CONTEXT specifies the first vocabulary to be searched.

CONTEXT and CURRENT allow words in one vocabulary to be defined using words from another vocabulary. The word DEFINITIONS sets CURRENT to CONTEXT so that new definitions are added to the first vocabulary in the search order. For example, suppose a new word is to be added to the GRAPHICS vocabulary and must use the ASSEMBLER vocabulary to define the word in assembly language. Before defining the word you might say:

```
GRAPHICS OK                ( set CONTEXT vocabulary )
                           ( to GRAPHICS )

DEFINITIONS OK             ( set CURRENT vocabulary )
                           ( to CONTEXT vocabulary )

ASSEMBLER OK               ( set CONTEXT vocabulary )
                           ( to ASSEMBLER )
```

This would normally be entered on one line as:

GRAPHICS DEFINITIONS ASSEMBLER_OK

which clearly states the intention to define new graphics words using assembly language.

A new vocabulary <name> is created by saying:

VOCABULARY <name>

This creates a special vocabulary variable <name> and sets this variable to point at the entry for <name>. As words are added to this vocabulary, the variable <name> is updated to point at the most recent dictionary entry. The execution of <name> causes CONTEXT to be set to <name>'s vocabulary pointer. The implementation of VOCABULARY is discussed in section 3.10.

Since CONTEXT and CURRENT are system variables, DEFINITIONS can be defined as:

```
: DEFINITIONS
   CONTEXT @              ( fetch CONTEXT vocabulary )
   CURRENT ! ;            ( store as CURRENT vocabulary )
```

Both CONTEXT and CURRENT contain pointers to vocabulary variables. The statement CURRENT @ @ would return a pointer to the most recent entry added to the compilation vocabulary. CONTEXT @ specifies the first vocabulary in the search order.

2.3.2 Primitives and Secondaries

In an ITC system, the body of each dictionary entry begins with a code address word (CAW - also known as a prologue address pointer) which distinguishes between primitives and secondaries (see figure 2.5). The CAW contains a pointer to machine code which is executed as the first step in the execution of a word definition. This prologue code

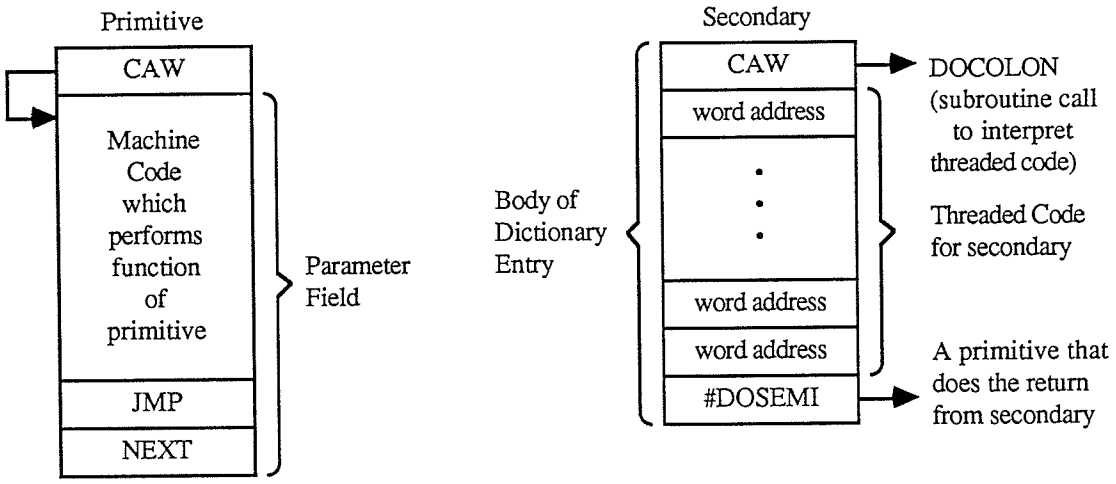


Figure 2.5: Indirect Threaded Code Bodies

determines the type of definition being executed.

In primitives, the CAW points to the parameter field (also known as the code body) which contains machine code to carry out the function of the word. The last machine instruction does a jump back to the address interpreter which continues executing the threaded code (TC).

In secondaries, the CAW points to a machine code routine commonly called DOCOLON (since a colon definition is used to define a secondary). DOCOLON acts as a subroutine call/entry procedure which pushes a return address onto the return stack and calls the address interpreter to execute the TC that follows in the parameter field. The last TC address is a pointer to DOSEMI - a primitive which performs the subroutine return/exit from the secondary. DOSEMI pops the return address from the return stack and calls the address interpreter to continue execution at this address (the TC following the word address of the secondary). This sequence of

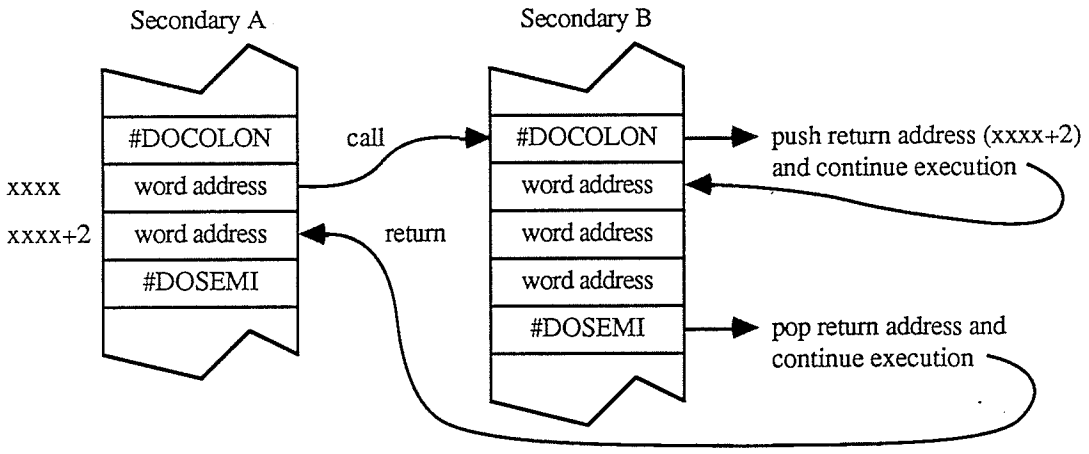


Figure 2.6: A Subroutine Call to a Secondary

events is shown in figure 2.6.

2.3.3 Pre-compiled Dictionary

The basic elements of Forth including the required word set, text interpreter and address interpreter are contained in the pre-compiled portion of the dictionary (see figure 2.7). This pre-compiled code is required to start-up a Forth system. The code is written is assembly language and is comprised mostly of dictionary entries.

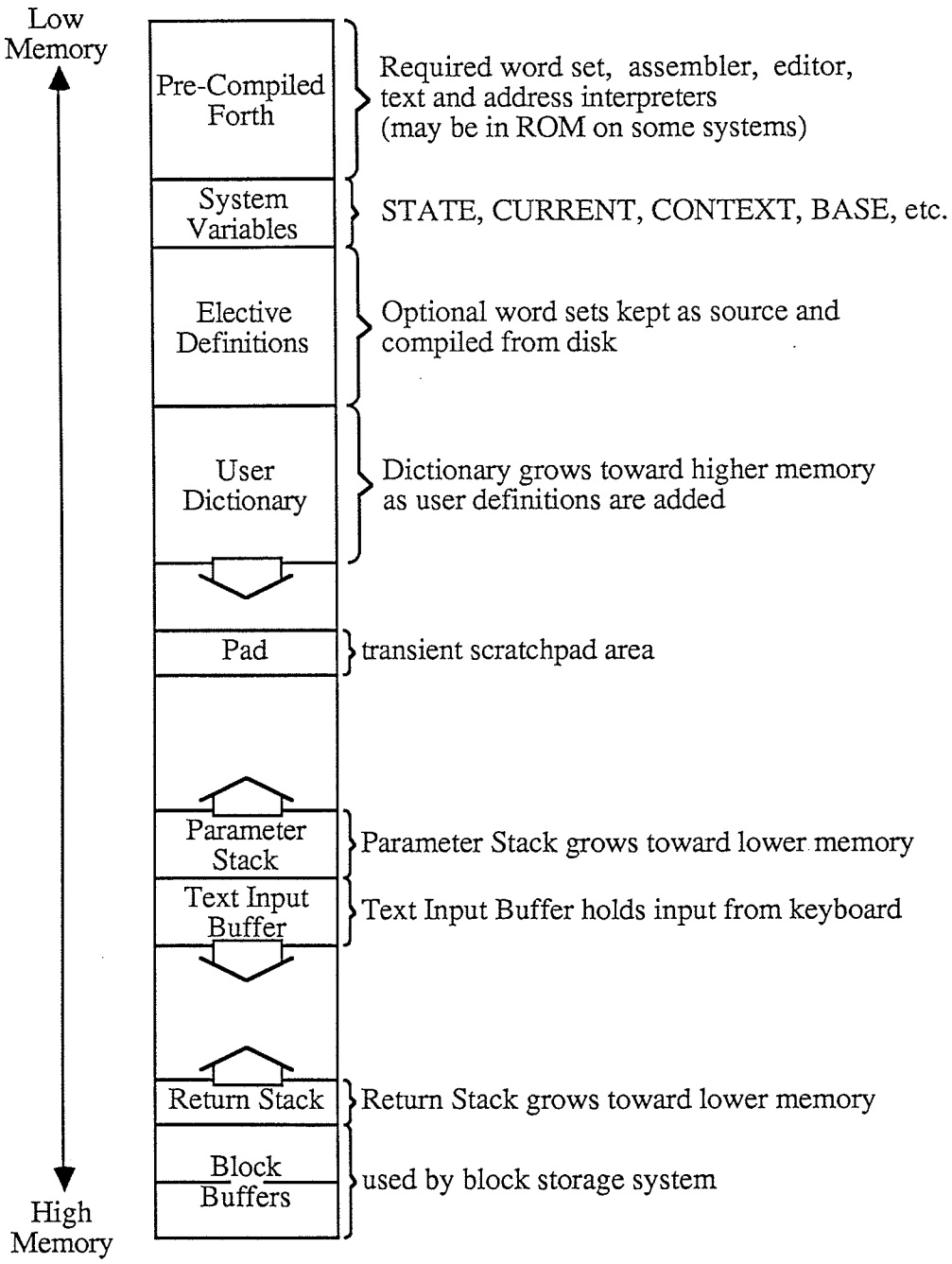
The headers for dictionary entries are coded using assembler directives to generate name, link and CAW fields. Primitive code bodies consist of native assembly code mnemonics. The TC for secondaries is "hand compiled" into assembler directives which generate the address list data. This source code is then assembled under the host operating system or cross-assembled on a different machine. The resulting object code contains the pre-compiled dictionary entries for the target machine.

The first step in this code is to call the address interpreter to begin executing the text interpreter. On some systems the pre-compiled code contains only the bare necessities: a few primitives and a simple text interpreter which does no error checking and can only read from disk. This code is then used to bootstrap a complete system by compiling the remaining Forth source from disk. This implementation of Forth is even easier to transport since assembly language code is kept to a minimum. Most of the system is

defined in Forth itself. This technique does produce a slower system because there are fewer primitives.

Usually most of the standard words are coded as primitives for efficiency reasons. The text interpreter is almost always written as a secondary. Once Forth is running, additional primitives may be defined using Forth's assembler language facility; secondaries are created via colon definitions.

Figure 2.7: A Forth Memory Map



2.3.4 Pad

The pad is a transient area in memory, located just above the dictionary. The pad acts as a scratch pad for storing intermediate results. One of its uses is to hold the ASCII character string generated by number-formatting words (see section 3.9). Once a number has been formatted into the pad, its ascii representation can be output to the terminal or placed elsewhere.

The word PAD returns the lower address of the pad. PAD is often defined as : PAD HERE n + ; where n is some fixed offset (say 100 bytes). The FORTH-83 Standard stipulates that the pad must have a minimum capacity of 84 bytes and that "the address or contents of PAD may change and the data lost if the address of the next available dictionary location is changed." This rarely causes problems since the two events do not normally take place at the same time (ie. they are mutually exclusive).

2.4 STACKS

The data stack and return stack are extensible structures which grow toward lower memory addresses (refer back to figure 2.7). The data stack is used for passing parameters between words. The return stack is used to save return addresses during execution of nested words. Both stacks provide storage for temporary variables and intermediate results.

FORTH-83 specifies that both stacks contain 16-bit entries. 32-bit numbers are stored on the stack as two entries (the most-significant 16-bits are pushed last). Byte, character and boolean data are zero extended and stored as 16-bit entries.

Some machines implement stacks directly in hardware. Others may require that stacks be implemented in software. In the later case, each stack can be represented by a stack pointer (SP) kept in memory (a variable) or, preferably, a dedicated register. A stack can then be implemented as

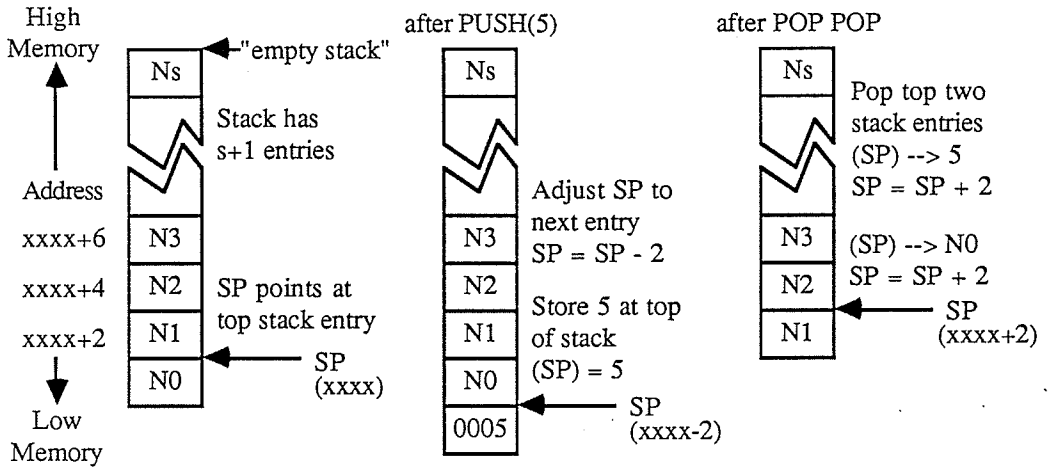


Figure 2.8: Implementing a Stack

shown in figure 2.8.

SP points at the top stack entry. The PUSH X operation decrements SP to point to the next entry and stores X at this location. POP X performs the reverse procedure by as-

signing X the value of the top stack entry and incrementing SP to point at the previous entry. The stack is empty if SP equals the "empty stack" pointer (the value of SP before any PUSH/POP operations). If SP is less than "empty stack" then the stack has underflowed (more POPs than PUSHes). Stack overflow occurs if the stack grows beyond its allocated area in memory (too many PUSHes).

Some machines require that stack entries be word aligned. Even if this is not a requirement it is often advantageous to word align the stacks since this generally allows faster word access (depending on the hardware). For example, a 16-bit wide path to memory can provide single access to any word located at an even address. Words located at odd addresses require two memory access cycles to obtain their two separate bytes.

2.4.1 Overflow and Underflow

Stack overflow or underflow can be disastrous since these conditions are rarely guarded against. Run-time error checking can be very costly in terms of execution time. Therefore the text interpreter checks the stacks after, but not during, the execution of a word.

Underflow occurs when a word consumes parameters beyond the "empty stack". Data stack underflow returns invalid data and will cause incorrect results at best. Return stack

underflow or any corruption of return addresses will cause the address interpreter to "run wild" - executing at some unknown address. Pushing values onto an underflowed stack will corrupt memory below the stack.

Overflow results in the destruction of memory immediately above the stack. Overflows are usually caused by lengthy recursive calls or stack-dictionary "collisions". The effects of stack corruption can easily "crash" the system.

As always, it is up to the programmer to test each word separately so that stack integrity can be assured. This process of incremental testing is widely accepted.

2.5 TEXT INTERPRETER

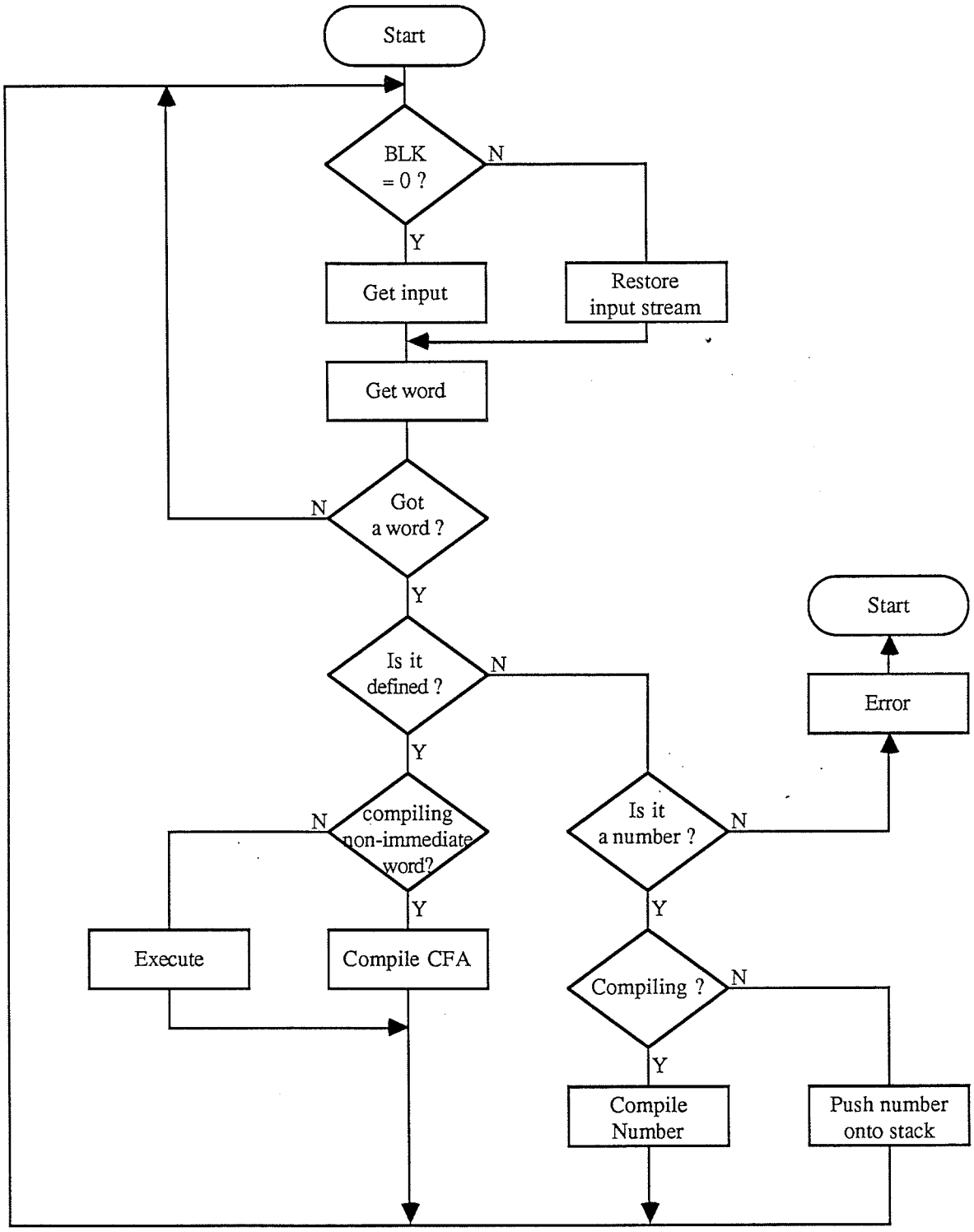
The text interpreter acts as a system executive. It provides an operator interface much like the UNIX[11] shell. The text interpreter is an infinite loop which repeatedly accepts word names and numbers from the input stream (see figure 2.9). The input stream comes from the current input device (usually the keyboard) or from backing store.

When Forth is booted up, the pre-compiled dictionary of kernel words is loaded into memory and the text interpreter (also known as the outer interpreter) is invoked. The text interpreter acts either as a compiler or an interpreter de-

---

[11] UNIX is a multiuser time-shared multitasking operating system developed by Bell Labs.

Figure 2.9: A Simple Text Interpreter



pending on the value of a system variable named STATE. STATE is a boolean flag; an integer whose value is either false (zero) or true (non-zero). If STATE equals true then the text interpreter is compiling, otherwise it is interpreting. In either case, the text interpreter scans the input stream and picks off one word at a time. Each word is located in the dictionary and the address of its body is returned. If STATE equals compiling as in the case of a colon definition, the address is appended to the body of the new word being defined (thus producing threaded code - a list of the addresses of its component words). On the other hand, if STATE equals interpreting, the address is passed to the word EXECUTE which calls the address interpreter to "execute" the code body of the word.

Immediate words are an exception to the above rule. An immediate word is always executed - even if STATE equals compiling. Immediate words are used to perform compile time actions. An example of such a word is ';', one of whose functions is to set STATE to interpreting upon completing a colon definition.

If a word name cannot be located it is assumed to be a number. The text interpreter attempts to convert the name to a number. Conversion is done using the system variable BASE, which contains an integer value in the range 2 to 72 (default is base 10). If the number is valid, it is pushed onto the stack (if STATE equals interpreting) or compiled

into the dictionary as a literal (if STATE equals compiling). A literal will push its value to the stack when it is encountered later during execution of a word (see section 3.1). Numbers are converted to 16-bit values unless immediately followed by a period as in 25. The period causes conversion to a 32-bit value.

If the text interpreter determines the name is an invalid number then an error condition exists. Error handling is system dependent but the usual response is to print the offending name along with an appropriate message, clear the stacks and set STATE to interpreting. Other possible actions and error conditions are detailed in the FORTH-83 Standard [FST83].

Aside from word name and number validation, the text interpreter does little or no syntax checking. This is largely due to the use of postfix notation. Error checking is often reduced to verifying that "stack empty" did not occur after executing a word.

2.5.1 Input Stream

The input stream is defined by the system variables BLK, TIB, #TIB and >IN. BLK determines the source of the input stream. If BLK is zero then the input stream is taken from the text input buffer depicted in figure 2.10. TIB returns the address of the first byte in the text input buffer.

#TIB gives the number of bytes which have been input to the text input buffer. >IN returns an offset which, when added to TIB, will point at the next character in the input

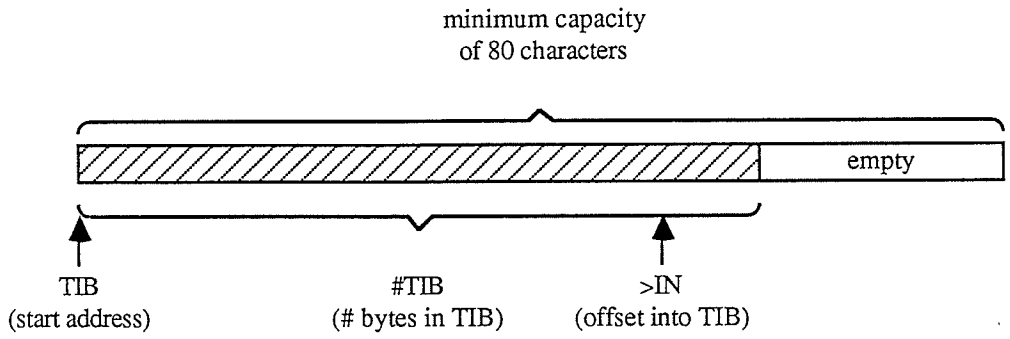


Figure 2.10: The Text Input Buffer

stream.

The text input buffer can be initially filled by a word such as:

```
: GET_INPUT   ( -- )
   TIB 80 EXPECT          ( input into buffer at TIB )
   SPAN @ #TIB !          ( set #TIB to #chars input )
   0 >IN ! ;              ( set >IN to zero )
```

EXPECT requires a buffer address and length. It then accepts input (from the keyboard) into the buffer. This continues until the buffer is full or a carriage return is entered. The carriage return is not stored in the buffer but is displayed as a space. EXPECT sets the system variable SPAN equal to the number of characters which were input to the buffer.

If BLK is non-zero, it specifies the block number of the block buffer containing the input stream. In this case, the input stream contains exactly 1024 bytes (the block size) and >IN contains an offset into the block buffer.

The word LOAD can be used to alter BLK and cause the input stream to be taken from backing store. This allows source code to be compiled or interpreted from disk. The statement n LOAD requests block n to be LOADed (refer to section 2.7). LOAD begins by saving the current input stream using a word like:

```
: SAVE_INPUT_STREAM   ( -- )
   R>                      ( get return address )
   BLK @ >R                ( push BLK onto return stack )
   >IN @ >R                ( push >IN onto return stack )
   >R ;                    ( restore return address )
```

which saves BLK and >IN onto the return stack. LOAD then sets up a block buffer for block n, sets BLK to n and >IN to zero. The input stream is now taken from the block buffer. Once this buffer is exhausted, a word such as:

```
: RESTORE_INPUT_STREAM   ( -- )
   R>                      ( get return address )
   R> >IN !                ( pop return stack into >IN )
   R> BLK !                ( pop return stack into BLK )
   >R ;                    ( restore return address )
```

is used to restore the input stream to its previous position. This allows nested LOADs. For example, block five may contain the sequence 6 LOAD 7 LOAD 8 LOAD. The statement 5 LOAD will now load blocks five through eight.

The input stream is normally accessed using WORD. WORD expects a delimiting character and returns the address of a

counted string. The sequence BL WORD is used to acquire the next word name from the input stream. BL pushes an ASCII blank (decimal 32) onto the stack. WORD then returns the address of the next blank delimited string (leading delimiters are ignored). The first byte in the string contains the string length (number of characters). Subsequent bytes contain the string data. A blank is appended to the string but not included in its length.

WORD determines its position in the input stream by examining BLK. If the input stream is already exhausted when WORD is called (BLK equals zero and >IN equals #TIB or BLK is non-zero and >IN equals 1024) then WORD returns a string of length zero. Otherwise, WORD adjusts >IN to the character following the delimiter (>IN is set to the size of the input stream if the delimiter is not found).

GET_WORD may be defined to return: 1) the address of a counted string containing the next word name from the input stream and 2) a flag indicating if the input stream was exhausted:

```
: GET_WORD   ( -- addr flag )
   BL WORD                  ( get next word )
   DUP @                    ( get word length )
   0 <> ;                   ( return true if zero length )
```

2.5.2 Implementation

Several words must be introduced before defining the text interpreter. The control structures IF-ELSE-THEN, BEGIN-UNTIL and BEGIN-WHILE-REPEAT are required here but are discussed in section 3.8.

FIND is used to search the dictionary for a word definition (using the currently active search order). FIND is passed the address of a counted string which contains the word name. If the word is not found, FIND returns this address and a value of zero. Otherwise, it returns the CFA (code field address or compilation address) of the word and a value of either minus one or one. Minus one is returned if the word is non-immediate. One indicates the word is immediate.

EXECUTE accepts a CFA and calls the address interpreter to execute the word definition at that address (refer to section 2.6).

CONVERT converts a counted string into a 32-bit value. CONVERT expects a 32-bit start value and the address of a counted string. It then determines the value of each digit (using BASE) beginning with the first character in the string. The start value is multiplied by the value of BASE before each digit is added. Conversion stops once an unconvertible character is encountered. CONVERT returns a 32-bit value and the address of the first unconvertible character.

[COMPILE] LITERAL compiles a 16-bit number as discussed in section 3.1. [COMPILE] DLITERAL performs the same function for 32-bit numbers. Section 3.1 also describes the word ." ("dot-quote") which is used in the form ." ccc" and causes ccc to be output to the terminal screen. ABORT" is similar to ." but expects a flag. If the flag is true, ABORT" ccc" displays ccc, clears the stacks, sets STATE to interpreting and accepts new input. This is used to restart the text interpreter after an error occurs.

COUNT takes the address of a counted string and returns address+1 and the length of the string. TYPE uses these (the address of a character string and a length) to output the character data to the terminal screen.

The text interpreter is usually implemented as a headerless secondary. A simple text interpreter can be defined as shown below:

```
0 CONSTANT FALSE            ( a false flag )
-1 CONSTANT TRUE            ( a true flag )
1 CONSTANT 32-BIT           ( 32-bit number flag )

: DEFINED   ( addr1 -- addr2 n flag )
   FIND                     ( do FIND plus return true )
   DUP 0<> ;                ( if the word was found )

( test value returned by FIND )
: NOT_IMMEDIATE   ( n -- flag )
   1 <> ;                   ( true for non-immediate words )

( test if STATE is compiling )
: COMPILING   ( -- flag )
   STATE @ 0 <> ;           ( return true if compiling )
```

```
( convert a string to a 32-bit number and return a flag )
( which indicates:                                       )
(                  0 = invalid number                    )
(                  1 = valid 32-bit number               )
(                 -1 = valid 16-bit number               )
: NUMBER?   ( addr -- d1 n1 )
   0. ROT CONVERT          ( attempt conversion )
   @ DUP                   ( first unconvertible character )
   ASCII . =               ( is a dot for 32-bit numbers )
   IF DROP 1
   ELSE BL =               ( a space for 16-bit numbers )
   THEN ;                  ( otherwise it is invalid )

( compile or interpret a number as returned by NUMBER? )
: PROCESS_NUMBER   ( d1 n1 -- )
   32-BIT =                ( check NUMBER? flag )
   IF
      COMPILING
      IF [COMPILE]
      DLITERAL             ( compile 32-bit number )
      THEN                 ( or leave it on stack )
   ELSE
      DROP                 ( discard upper 16-bits )
      COMPILING
      IF [COMPILE]
      LITERAL              ( compile 16-bit number )
      THEN                 ( or leave it on stack )
   THEN ;

( execute or compile a word address )
: PROCESS_WORD   ( addr -- )
   DEFINED                 ( is word defined? )
   IF
      COMPILING            ( if compiling and non-immediate )
      NOT_IMMEDIATE AND
      IF
         ,                 ( enclose word address )
      ELSE
         EXECUTE           ( otherwise execute word )
      THEN
   ELSE
      DROP                 ( discard zero flag )
      DUP                  ( duplicate string address )
      NUMBER?              ( is word a valid number? )
      IF
         PROCESS_NUMBER    ( compile or stack number )
      ELSE
         DROP              ( discard erroneous number )
         ." *** "          ( begin error message )
         COUNT TYPE        ( print offending word and)
                           ( restart text interpreter )
         ABORT"  is invalid "
      THEN
   THEN ;
```

```
( text interpreter loop )
0 BLK !                        ( BLK to zero )
BEGIN                          ( text interpreter loop )
   BLK @ 0=                    ( check input stream )
   IF
      GET_INPUT                ( get input from keyboard )
   ELSE                        ( or continue previous input )
      RESTORE_INPUT_STREAM
   THEN
   BEGIN
      GET_WORD                 ( get next word )
      WHILE
      PROCESS_WORD             ( execute or compile each word )
   REPEAT
   BLK @ 0=                    ( if end of input line )
   IF
      ." OK"                   ( print "OK" and carriage )
      CR                       ( return, linefeed )
   THEN
   FALSE
UNTIL                          ( loop forever )
```

2.6 ADDRESS INTERPRETER

The address interpreter should execute as fast as possible since it determines how quickly the threaded code (TC) can be interpreted and hence how fast the language runs. Usually the address interpreter is coded in assembly language. Although the instruction set may change from machine to machine, the function of the address interpreter remains the same. The only real difference may be the length or the number and type of available registers.

A generic assembler language or pseudo-code can be used to illustrate the function and implementation of the address interpreter. The FORTH-83 Standard specifies the use of 16-bit addresses and byte (8-bit) or word (16-bit) data. Assume the machine:

1. has a byte addressible 16-bit address space

2. has two or more 16-bit registers to maintain:

   a) two stack pointers used to implement

      - ds, the data stack
      - rs, the return stack

   b) four index registers for

      - pc, the program counter used to sequence through machine instructions; points at the next machine code instruction to be executed
      - ip, the threaded code instruction pointer used to sequence through the TC; points at the next word address in the TC
      - wa, a temporary containing a pointer to the code address word (CAW) of the word definition being currently executed
      - ca, a temporary containing the CAW of the word being currently executed

   c) two general purpose register

      - ra, for intermediate results/calculations
      - rb, as above

3. has machine instructions for word operations such as:

a) ra = POP(ds)	pop word from stack and assign to ra
b) PUSH(rs) = ra	push ra onto return stack
c) ra = ra + 1	increment ra by one
d) ra = (ca)	fetch word pointed to by ca and assign to ra
e) ra = (ca++)	as above but then increments ca to point to the next word
f) ra = (--ca)	decrement ca to point at previous word, then fetch this word and assign it to ra
g) (ca) = ra	store ra to word pointed to by ca

h) ra = (ca+4) — fetch word pointed to by the address given by the sum of ca and 4

i) JMP (ca) — jump to address given in ca (same effect as pc = ca)

4. has similar instructions for byte (B) operations

a) ra =B (ca) — fetch byte pointed to by ca and assign to ra (only the least significant byte of ra is affected)

b) ra =B (ca++) — as above but then increments ca to point to the next byte

The address interpreter can be implemented on an actual machine by using a subset of the available machine registers or by simulating registers (as global words in memory) if there are not enough real registers.

Stack operators and special addressing modes may be simulated if necessary:

Instruction	Simulation
PUSH(ds) = ra	(--ds) = ra
ra = POP(ds)	ra = (ds++)
ra = (ca++)	ra = (ca) ca = ca + 2
ra = (--ca)	ca = ca - 2 ra = (ca)
ra =B (ca++)	ra =B (ca) ca = ca + 1

There may also be cases where a processor can't operate on an entire word in one operation. An example would be a processor with an 8-bit arithmetic logic unit. In this case

each word operation would require two byte operations - at least doubling the number of machine instructions.

Ultimately, Forth is best implemented on a machine which directly executes threaded code (see "Forth in Hardware"). Another choice might be a zero-address (stack) machine - since Forth relies heavily on its use of stack operators. The important thing to note is that the function of the address interpreter is independent of the underlying machine. In every case the address interpreter acts as a virtual machine which "executes" threaded code.

The pseudo-code for an ITC address interpreter is:

```
NEXT     wa = (ip++)          get word address from TC
                              and set ip to point at the
                              next word address
         ca = (wa++)          get code address word (CAW) and
                              set wa to point at the start
                              of the parameter field
         JMP (ca)             jump to machine code pointed
                              to by CAW

DOCOLON  PUSH(rs) = ip        save TC return address
         ip = wa              setup to execute TC in
                              parameter field of secondary
         JMP NEXT             continue execution

DOSEMI   FCW *+2[12]          CAW for primitive
         ip = POP(rs)         get TC return address
         JMP NEXT             continue execution at that address
```

---

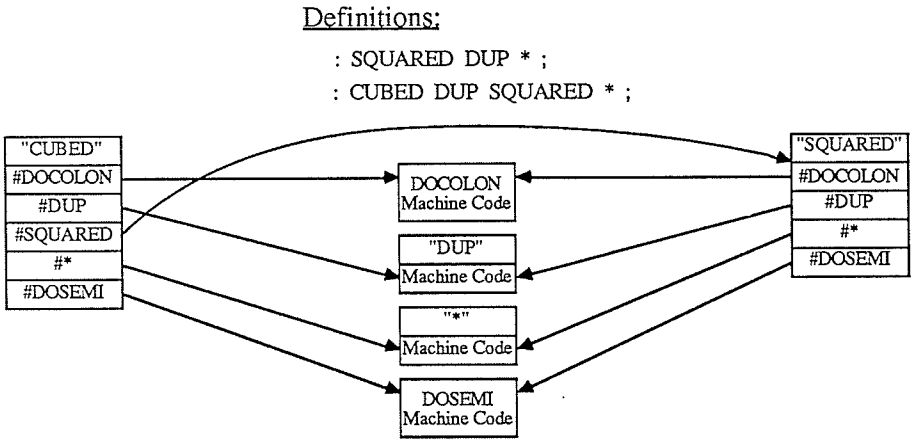
[12] FCW represents an assembler directive "form constant word". The asterisks represents an assembler directive which returns the current value of the program counter. Thus FCW *+2 assembles a two byte constant value which points at the following instruction.

The JMP NEXT instruction is often replaced by JMP (next), where next is a register dedicated to holding the address of NEXT. JMP NEXT can also be replaced by inline NEXT code. These techniques are used to decrease execution time.

The execution of the address interpreter is detailed in the following example. Assume that the words SQUARE and CUBED have been defined and compiled into the dictionary as shown in figure 2.11. The address interpreter and the word EXECUTE are also included in this figure. Note that DOSEMI has been placed immediately before NEXT to save the execution of a JMP instruction. DOSEMI was chosen instead of DOCOLON because DOSEMI may be called more frequently. DOSEMI and DOCOLON normally occur as a pair, however, DOSEMI does appear by itself in definitions created by high level defining words (see section 3.10).

If the expression 3 CUBED was entered, the text interpreter would push 3 and 2196 (the code address of CUBED - refer to figure 2.11) onto the stack. EXECUTE would then be called and the address interpreter would proceed as shown in figure 2.12. After CUBED has executed, the stack contains the result (27).

Figure 2.11: Sample Dictionary Entries and the Address Interpreter



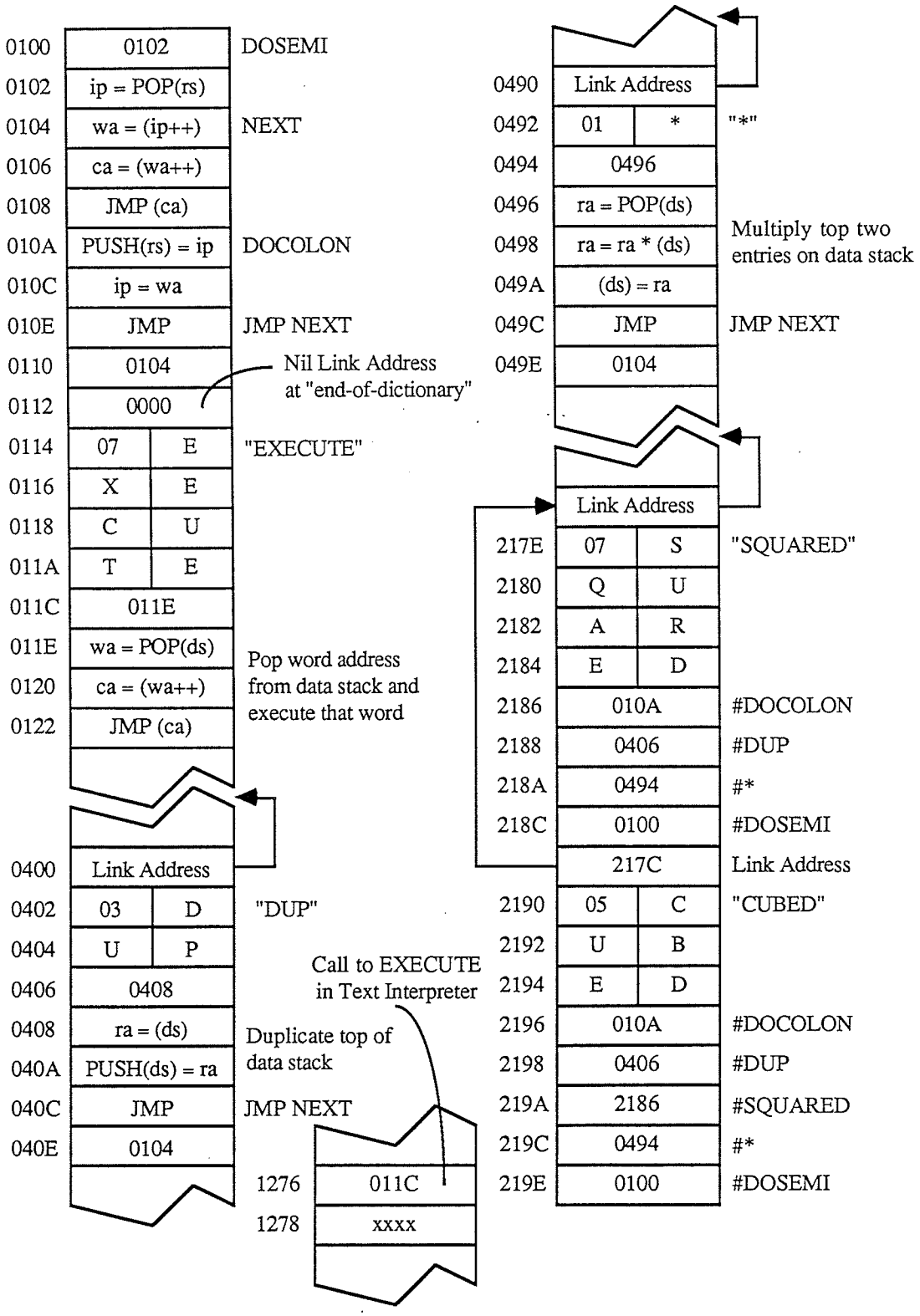

0100 0102 DOSEMI
0102 ip = POP(rs)
0104 wa = (ip++) NEXT
0106 ca = (wa++)
0108 JMP (ca)
010A PUSH(rs) = ip DOCOLON
010C ip = wa
010E JMP JMP NEXT
0110 0104
0112 0000
Nil Link Address at "end-of-dictionary"
0114 07 E "EXECUTE"
0116 X E
0118 C U
011A T E
011C 011E
011E wa = POP(ds)
0120 ca = (wa++)
0122 JMP (ca)
Pop word address from data stack and execute that word
0400 Link Address
0402 03 D "DUP"
0404 U P
0406 0408
0408 ra = (ds)
040A PUSH(ds) = ra
040C JMP JMP NEXT
040E 0104
Call to EXECUTE in Text Interpreter
Duplicate top of data stack
1276 011C
1278 xxxx
0490 Link Address
0492 01 * "*"
0494 0496
0496 ra = POP(ds)
0498 ra = ra * (ds)
Multiply top two entries on data stack
049A (ds) = ra
049C JMP JMP NEXT
049E 0104
Link Address
217E 07 S "SQUARED"
2180 Q U
2182 A R
2184 E D
2186 010A #DOCOLON
2188 0406 #DUP
218A 0494 #*
218C 0100 #DOSEMI
217C Link Address
2190 05 C "CUBED"
2192 U B
2194 E D
2196 010A #DOCOLON
2198 0406 #DUP
219A 2186 #SQUARED
219C 0494 #*
219E 0100 #DOSEMI

Figure 2.12: Address Interpreter Execution

ROUTINE	pc	instruction	ip	wa	ca	(ds)	(ds+2)	(ds+4)	(rs)	(rs+2)	comments
NEXT	0104	wa = (ip++)	1276	-----	-----	2196	0003	-----	-----	-----	Begin text interpreters
	0106	ca = (wa++)	1278	011C	-----	2196	0003	-----	-----	-----	call to EXECUTE
	0108	JMP (ca)	1278	011E	011E	2196	0003	-----	-----	-----	
EXECUTE	011E	wa = POP(ds)	1278	011E	011E	2196	0003	-----	-----	-----	Run EXECUTE
	0120	ca = (wa++)	1278	2196	011E	0003	-----	-----	-----	-----	
	0122	JMP (ca)	1278	2198	010A	0003	-----	-----	-----	-----	
DOCOLON	010A	PUSH(rs) = ip	1278	2198	010A	0003	-----	-----	-----	-----	Nest down one level to
	010C	ip = wa	1278	2198	010A	0003	-----	-----	1278	-----	begin CUBED
	010E	JMP NEXT	2198	2198	010A	0003	-----	-----	1278	-----	
NEXT	0104	wa = (ip++)	2198	2198	010A	0003	-----	-----	1278	-----	
	0106	ca = (wa++)	219A	0406	010A	0003	-----	-----	1278	-----	
	0108	JMP (ca)	219A	0408	0408	0003	-----	-----	1278	-----	
DUP	0408	ra = (ds)	219A	0408	0408	0003	-----	-----	1278	-----	Execute DUP in CUBED
	040A	PUSH(ds) = ra	219A	0408	0408	0003	-----	-----	1278	-----	
	040C	JMP NEXT	219A	0408	0408	0003	0003	-----	1278	-----	
NEXT	0104	wa = (ip++)	219A	0408	0408	0003	0003	-----	1278	-----	
	0106	ca = (wa++)	219C	2186	0408	0003	0003	-----	1278	-----	
	0108	JMP (ca)	219C	2188	010A	0003	0003	-----	1278	-----	
DOCOLON	010A	PUSH(rs) = ip	219C	2188	010A	0003	0003	-----	1278	-----	Nest down one level to
	010C	ip = wa	219C	2188	010A	0003	0003	-----	219C	1278	begin SQUARED
	010E	JMP NEXT	2188	2188	010A	0003	0003	-----	219C	1278	
NEXT	0104	wa = (ip++)	2188	2188	010A	0003	0003	-----	219C	1278	
	0106	ca = (wa++)	218A	0406	010A	0003	0003	-----	219C	1278	
	0108	JMP (ca)	218A	0408	0408	0003	0003	-----	219C	1278	
DUP	0408	ra = (ds)	218A	0408	0408	0003	0003	-----	219C	1278	Execute DUP in SQUARED
	040A	PUSH(ds) = ra	218A	0408	0408	0003	0003	-----	219C	1278	
	040C	JMP NEXT	218A	0408	0408	0003	0003	0003	219C	1278	
NEXT	0104	wa = (ip++)	218A	0408	0408	0003	0003	0003	219C	1278	
	0106	ca = (wa++)	218C	0494	0408	0003	0003	0003	219C	1278	
	0108	JMP (ca)	218C	0496	0496	0003	0003	0003	219C	1278	
*	0496	ra = POP(ds)	218C	0496	0496	0003	0003	0003	219C	1278	Execute * in SQUARED
	0498	ra = ra * (ds)	218C	0496	0496	0003	0003	-----	219C	1278	
	049A	(ds) = ra	218C	0496	0496	0003	0003	-----	219C	1278	
	049C	JMP NEXT	218C	0496	0496	0009	0003	-----	219C	1278	
NEXT	0104	wa = (ip++)	218C	0496	0496	0009	0003	-----	219C	1278	
	0106	ca = (wa++)	218E	0100	0408	0009	0003	-----	219C	1278	
	0108	JMP (ca)	218E	0102	0102	0009	0003	-----	219C	1278	
DOSEMI	0102	ip = POP(rs)	218E	0102	0102	001B	-----	-----	219C	1278	Denest back to CUBED
NEXT	0104	wa = (ip++)	219C	0102	0102	001B	-----	-----	1278	-----	
	0106	ca = (wa++)	219E	0494	0102	001B	-----	-----	1278	-----	
	0108	JMP (ca)	219E	0496	0496	001B	-----	-----	1278	-----	
*	0496	ra = POP(ds)	219E	0496	0496	0009	0003	-----	1278	-----	Execute * in CUBED
	0498	ra = ra * (ds)	219E	0496	0496	0003	-----	-----	1278	-----	
	049A	(ds) = ra	219E	0496	0496	0003	-----	-----	1278	-----	
	049C	JMP NEXT	219E	0496	0496	001B	-----	-----	1278	-----	
NEXT	0104	wa = (ip++)	219E	0496	0496	001B	-----	-----	1278	-----	
	0106	ca = (wa++)	21A0	0100	0496	001B	-----	-----	1278	-----	
	0108	JMP (ca)	21A0	0102	0102	001B	-----	-----	1278	-----	
DOSEMI	0102	ip = POP(rs)	21A0	0102	0102	001B	-----	-----	1278	-----	Denest back to
NEXT	0104	wa = (ip++)	1278	0102	0102	001B	-----	-----	-----	-----	text interpreter
	0106	ca = (wa++)	127A	xxxx	0102	001B	-----	-----	-----	-----	

2.7 BLOCK STORAGE SYSTEM

The block storage system (BSS) provides a simple low-level interface to backing store. The BSS views the backing store as a random access block oriented storage device. The device (floppy, hard disk, etc) is mapped into n blocks of 1024 bytes each. A block is accessed by its block number (zero to n-1) which maps back to a physical address on the storage device.

Blocks are used for storing source code and data. A block which contains source is called a screen. Each screen contains 16 lines of 64 characters each. Screen n is loaded by saying n LOAD. This sets BLK to n so that the input stream will be taken from block n. The screen is interpreted exactly as if it were entered at the keyboard. This allows screens to act as command lists - defining or executing words. For example, the word '-->' ("next block") can be placed at the end of a screen. This causes the next screen to be LOADed. Thus a series of consecutive screens can be interpreted by a single LOAD. An EDITOR vocabulary provides words for editing screens.

Data can be stored in any desired format within a block or range of blocks. High level defining words may be used to create "virtual" data structures which transparently access block storage. This facilitates a simple form of virtual memory.

The BSS performs demand paging between the block device and memory resident block buffers. Each of the two or more block buffers has an associated block number and an update flag. Block n is requested by saying n BLOCK. BLOCK first determines if the block is already in memory. If the block is not resident, it must be read from the block device into an available block buffer. This may require a modified buffer to be written out prior to reading in the new block. In any case, BLOCK ensures that the requested block is resident and returns a pointer to its block buffer.

Blocks are read or modified in memory using the fetch and store operators together with the address returned by BLOCK. The word UPDATE is used to mark the most recent BLOCK referenced. This ensures that the block buffer will be written out to backing store. SAVE-BUFFERS can be used to write out all UPDATEd buffers and mark them as unmodified. FLUSH performs SAVE-BUFFERS and unassigns all block buffers (sets their block numbers to zero) - this is useful when changing disks. BUFFER performs a subset of BLOCK's operations - n BUFFER assigns a block buffer to block n and returns a pointer to the buffer. The contents of the buffer are unspecified.

The above mentioned words form the BSS and provide a standard device-independent block interface. The only perceived difference between one storage device and another, is the number of blocks which can be stored.

At its lowest level, the BSS must translate a block request into one or more physical requests to the storage device. Consider a single-sided floppy disk drive as an example. Suppose the drive has 40 tracks, 9 sectors per track and 512 bytes per sector. Then record n (zero to 359) can be found on track [n/9], sector [n mod 9]. Block n could be mapped to records 2n and 2n+1. A request for block 10 (assuming block 10 was not already resident) would read records 20 and 21 into a block buffer.

The implementation of block related words is straightforward and not discussed in any greater detail. For further information refer to [Bro81]. An implementation of virtual arrays using the BSS is detailed in section 3.10.1.

2.8 META COMPILER

Meta-compilation is the process of compiling Forth source code (on the host machine) to produce an executable Forth nucleus (image) for a target machine. The host machine provides a complete software development environment. The target machine often provides only a subset of this environment and may contain hardware components not available on the host. In any case, the meta-compiler is written in Forth and runs on the host machine. Source code for the target application is compiled to produce a target image. This image is transferred to the target machine where it can then be executed.

Meta-compilation can be classified as either meta-compiling, cross-compiling or target-compiling. Meta-compiling invokes a true "compiler-compiler". The host and target machines are defined as having the same processor. Any definitions on the host machine can be compiled for the target machine. This includes defining words and immediate words - these are difficult to handle if the compiler being extended (for the target) is not the compiler running (on the host). The meta-compiler is used to recompile Forth for a different target machine.

Cross-compiling is meta-compiling on one processor to generate executable code for a different processor. Interactive testing of target source code is limited to machine independent Forth code. Target assembler and I/O dependent code cannot be tested.

Target compilation is the most common process performed by a meta-compiler. A target-compiler generates an optimal sized (reduced) nucleus based on the application program. The target image executes as a turnkey system containing only those elements of Forth which are needed by the application. An interactive application requires the text interpreter and search words. User available words must have headers so that they may be found in the dictionary. Closed target systems (those which do not require word searches) can have all headers removed from the dictionary. The text interpreter can also be removed in many instances.

In the case of a target-compiler, the target image is often placed in EPROM and installed on the target machine. This provides a turnkey system which executes Forth internally (invisible to the user). The ability to interactively develop software and to generate compact object code makes Forth ideal for creating ROM-based applications. In fact,

> "the cost advantage of using the Forth programming environment for producing ROMable applications code for inexpensive microcomputer systems is so great as to make attempts using other operating systems and languages non-competitive" [Pay84].

All types of meta-compilers are implemented using similar techniques. The basic idea is to create a target dictionary on the host machine. Once the application has been compiled, the target dictionary will contain the target image. A separate "ghost" dictionary may be used to contain header information for the target dictionary if the application is being target compiled. In any case, the process of meta-compilation requires using kernel words which are redefined to generate code in the target dictionary rather than the host dictionary.

The target image contains remote addresses which correspond to the addresses at which the image will reside once installed on the target machine. Since the image is position or machine dependent, it usually cannot be executed on the host. Some types of threaded code do allow the generation of relocatable code as discussed in sections 4.1 and 4.5. A more complete discussion of meta-compiler implementation can be found in [Loe81] and [RMB85].

Chapter III

LANGUAGE COMPONENTS

This chapter discusses the function and implementation of the basic elements of Forth's high level language. This includes sample definitions and the dictionary entries compiled for each. Assembly level code is presented using the generic assembly language described in the previous chapter.

3.1 LITERALS

There are two types of literals. A number literal causes its number to be pushed to the stack. A string literal causes its string to be displayed. Literals are compiled by the text interpreter into the form shown in figure 3.1. The text interpreter encloses the word address of a literal handler followed by its immediate data. The symbol '#' is used to denote a word address (e.g. #DOSEMI represents the word address of DOSEMI).

Numbers are compiled by using the word LITERAL which encloses the word address of the number literal handler, #DONUM, followed by the number (a 16-bit value). At execution time DONUM will push its number to the stack and increment the TC instruction pointer by two (to skip over the number data). DLITERAL and DONUMD are similarly used to compile 32-bit numbers.

Definitions:

```
: TESTA 1015 + ;
: TESTB 1015. + ;
: TESTC ." HI THERE" ;
```

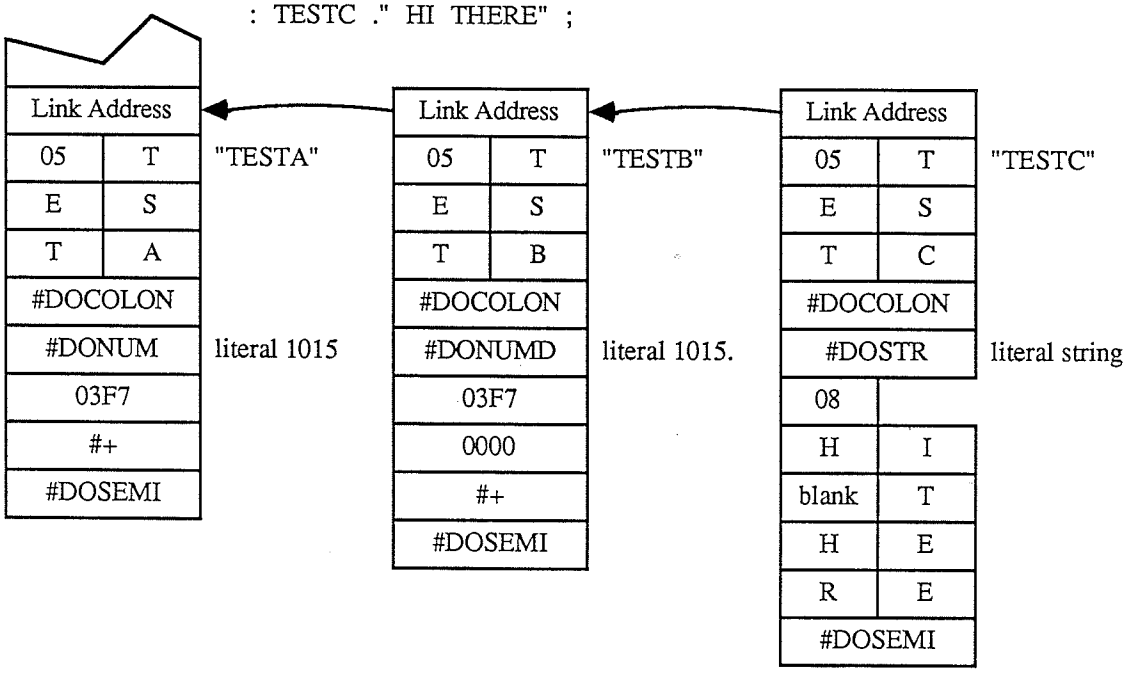


Figure 3.1: Literals

The word ." (pronounced "dot-quote") is used to compile string literals. Dot-quote is an immediate word that encloses the word address of the string literal handler, #DOSTR, followed by the string data. The string data is obtained by calling HEX 22 WORD to scan the input stream until a quote is encountered. When DOSTR is executed it will display the counted string and leave the TC instruction pointer pointing at the next word address (the byte following the last character of the string data).

The literal handlers are defined as headerless[13] primitives:

```
DONUM   FCW *+2                 CAW for primitive
        PUSH(ds) = (ip++)       fetch word at instruction
                                pointer and increment ip (by 2)
                                to point at the instruction
                                following the literal value
        JMP NEXT

DONUMD  FCW *+2                 CAW for primitive
        PUSH(ds) = (ip++)       get lower 16-bits
        PUSH(ds) = (ip++)       get upper 16-bits
                                ip is incremented (by 4)
                                to point at the instruction
                                following the literal value
        JMP NEXT

DOSTR   FCW *+2                 CAW for primitive
        ra =B (ip++)            get string length
DOSTR1  rb =B (ip++)            get next character in string
        PUTCHR(rb)              system call to output the
                                character in the least
                                significant byte of rb
        ra =B ra - 1            decrement count
        IF ra >B 0 BRA DOSTR1     repeat until all
                                  characters are displayed
        JMP NEXT
```

LITERAL and DLITERAL are defined as an immediate words:

```
( enclose DONUM literal handler followed )
( by a 16-bit value                      )
: LITERAL   ( -- n1 )
            ( n1 -- compiling )[14]
   #DONUM , ,
 IMMEDIATE
```

---

[13] Pre-compiled system words are stored as headerless entries if they need not be directly accessible to the user. A headerless entry has no name field - it requires less space but cannot be located in the dictionary. Its address must be "known" to the system.

[14] Compiling indicates stack parameters at compile-time. For example, LITERAL expects a 16-bit value at compile-time but returns a 16-bit value at execution-time.

```
( enclose DONUMD literal handler followed by lower and )
( upper 16-bits of the 32-bit value )
: DLITERAL    ( -- d1 )
              ( d1 -- compiling )
   #DONUMD , SWAP , , ; IMMEDIATE
```

LITERAL and DLITERAL can be compiled into a word definition (as is done in the text interpreter) by saying [COMPILE] LITERAL or [COMPILE] DLITERAL. [COMPILE] forces compilation of immediate words (normally they are executed at compile time).

Systems using an 8-bit processor will often have a byte literal handler for compiling numbers in the 8-bit range. A byte value occupies less space and can usually be fetched, extended and pushed to the stack faster than a 16-bit number (on an 8-bit machine). On the other hand, a 16-bit machine can often fetch and push a 16-bit value faster than an 8-bit value (since the 8-bits must be extended to 16-bits before being pushed). The generic machine supports word access and therefore precludes the use of a byte literal handler (since FORTH-83 chooses speed over compactness).

3.2 CREATEING A DICTIONARY ENTRY

CREATE is a defining word used to create a dictionary entry for a new word. All other defining words (CONSTANT, :, etc) call CREATE to begin a word definition. CREATE takes the following word from the input stream and creates a dictionary entry for it as shown in figure 3.2. The new header is linked to the previous dictionary entry in the compila-

tion vocabulary, the CAW is set to point at DOCREATE, and the system variable LAST is set to point at the new entry.

After CREATE has executed, the next available dictionary location will be the first byte of the new word's parameter field. DOCREATE will return a pointer to the parameter

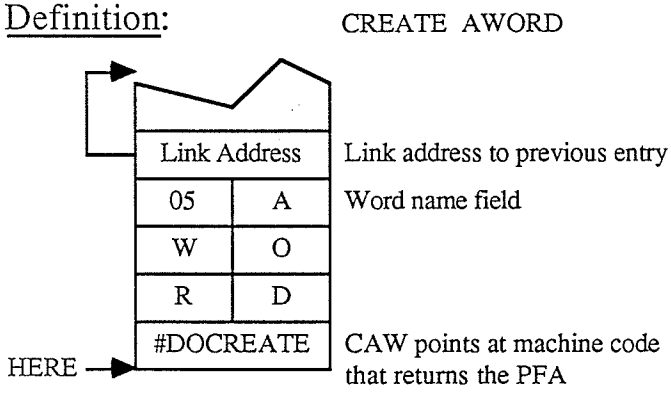


Figure 3.2: A CREATEd Dictionary Header

field when the new word is executed.

CREATE can be defined as shown below. The non-standard word WIDTH is a system variable containing the maximum number of characters to be stored in the name field (WIDTH = 31 for FORTH-83). LAST is used to keep a pointer to the most recent dictionary entry.

```
( get a word name from the input stream )
( returning a pointer to the string and the size of the )
( name field required to store this string )
: GET_NAME    ( -- addr n1 )
   BL                        ( push blank onto stack )
   WORD                      ( get word name from input )
                             ( stream, stack now has )
                             ( nameptr length )
   DUP C@                    ( get number of characters )
                             ( in word name )
   WIDTH @                   ( get maximum name size )
   MIN 1+ ;                  ( take minimum of these )
                             ( and add 1 to get length )
                             ( of name field )

( accepts a pointer to a string and the string length )
( encloses this string into the dictionary )
: SET_NAME    ( addr n1 -- )
   SWAP OVER                 ( stack now has )
                             ( length nameptr length )
   HERE                      ( point to name field )
                             ( length nameptr length dest )
   SWAP                      ( length nameptr dest length )
   CMOVE                     ( copy length bytes from )
                             ( nameptr to dest address )
                             ( this sets up name field )
   ALLOT ;                   ( enclose name field )
                             ( of specified length )

( accepts pointer to most recent dictionary entry and )
( links this entry into the CURRENT vocabulary )
: SET_LINK    ( addr -- )
   DUP                       ( get copy of entry address )
   CURRENT @ @               ( point to previous entry )
                             ( in CURRENT vocabulary )
                             ( stack has addr addr prev )
   SWAP !                    ( set link field of new )
                             ( entry to prev )
                             ( stack has addr )
   CURRENT @                 ( point to CURRENT vocab )
                             ( vocabptr )
   ! ;                       ( set vocab to point at )
                             ( new entry )

: CREATE    ( -- )
   HERE                      ( point to start of entry )
   LAST !                    ( set value of LAST )
   0 ,                       ( reserve link field )
   GET_NAME                  ( get name string )
   SET_NAME                  ( enclose name field )
   LAST @                    ( get entry address )
   SET_LINK                  ( link to previous entry )
   #DOCREATE , ;             ( enclose word address )
                             ( of DOCREATE )
```

3.3 IMMEDIATE WORDS

Immediate words are words which execute even if encountered during compilation. The text interpreter can tell if a word is immediate by checking a flag bit in the dictionary entry for that word. By convention, the immediate flag is stored as the most significant bit of the byte containing the length of the word's name. This bit is normally turned off but can be set on to indicate the word is immediate.

The word IMMEDIATE is used to mark the most recently created dictionary entry as an immediate word. Immediate words are executed (rather than compiled) during compilation; this allows them to perform compile-time functions (calculations, code generation, etc).

The effect of IMMEDIATE is illustrated by the following:

```
: MSG    ( -- )
   ." HERE GOES" ; IMMEDIATE OK

: AMSG   ( -- )
   ." HELLO " MSG ." THERE" ; HERE GOES OK

AMSG HELLO THERE OK
```

A more useful example would be the definition of ';' for terminating a colon definition. Semi-colon is defined in the pre-compiled portion of the dictionary. Its dictionary entry is shown in figure 3.3. A functional definition of ';' is discussed in section 3.5.

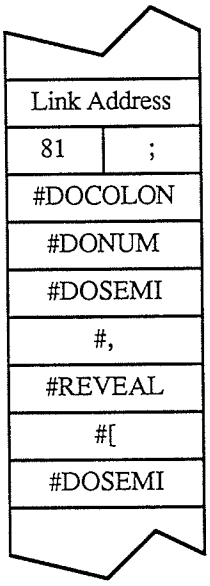
IMMEDIATE is defined as:

```
HEX                                ( numbers are in hexadecimal )
: IMMEDIATE    ( -- )
   LAST @ 2+                       ( get address of length byte )
   DUP C@                          ( fetch length byte )
   80 OR                           ( set immediate flag )
   SWAP C! ;                       ( store byte with flag set )
```

Figure 3.3: An Immediate Word

Definition:

: ; #DOSEMI , REVEAL [; IMMEDIATE



";" is a pre-compiled entry, note that immediate flag is set

enclose word address of DOSEMI

set state = interpreting

3.4 DEFINING PRIMITIVES

Primitives are defined using Forth's assembler language vocabulary. The words CODE and END-CODE are analogous to ':' and ';' used in defining a secondary. The assembly language code is entered in postfix notation (operand before

instruction) as discussed in section 3.12. Each instruction is represented by a Forth word which takes its operands from the stack, generates the object code for that instruction, and encloses this into the dictionary.

As an example, the fetch (@) and store (!) operators could be defined as:

```
CODE @    ( addr -- n1 )
   ra ds POP =              ( get address from stack )
   ds PUSH ra ( =           ( fetch word at this address )
                            ( and push it onto stack )
   NEXT END-CODE

CODE !    ( n1 addr -- )
   ra ds POP =              ( get address from stack )
   ra ( ds POP =            ( get value from stack and )
                            ( store it at this address )
   NEXT END-CODE
```

These would produce the dictionary entries shown in figure 3.4. CODE is a defining word which does a CREATE to build a dictionary entry. It then switches to the ASSEMBLER vocabulary to make assembler words available. NEXT encloses a JMP NEXT instruction in the dictionary and END-CODE terminates the definition.

FORTH-83 specifies that the word <name> defined by CODE <name> cannot be found in the dictionary until END-CODE is executed. After CODE creates <name>, it temporarily removes <name> from the CURRENT vocabulary; END-CODE later re-installs <name>. Two non-standard words, HIDE and REVEAL, have been defined to perform these functions needed by CODE and END-CODE. HIDE modifies the CURRENT vocabulary to point at its previous entry (the last word defined before <name>).

Now <name> cannot be found in the CURRENT vocabulary. REVEAL resets the CURRENT vocabulary to point at <name> so that <name> can be found.

HIDE and REVEAL are defined as:

```
: HIDE   ( -- )
   LAST @                    ( point to LFA - link )
                             ( field address of <name> )
   @                         ( fetch link to previous )
                             ( entry )
   CURRENT @ ! ;             ( update CURRENT vocabulary )
                             ( to point here )

: REVEAL   ( -- )
   LAST @                    ( point to entry for <name> )
   CURRENT @ ! ;             ( update CURRENT vocabulary )
                             ( to point to <name> )
```

Definitions for CODE and END-CODE are:

```
: CODE   ( -- )
         ( -- sys compiling )
   CREATE                    ( create entry for primitive )
   HIDE                      ( hide <name> temporarily )
   HERE DUP 2- !             ( set CAW to point at )
                             ( parameter field )
   ASSEMBLER ;               ( set CONTEXT to ASSEMBLER )

: END-CODE   ( -- )
             ( sys -- compiling )
   REVEAL ;                  ( make <name> visible )
```

These definitions often include code to verify that CODE/END-CODE are used as a pair. For example, CODE would push a special value (denoted by "sys") onto the stack and END-CODE would check the stack (for "sys") to determine if CODE had been used.

Definitions:

```
CODE @ ra ds POP = ds PUSH ra ( = NEXT END-CODE
CODE ! ra ds POP = ra ( ds POP = NEXT END-CODE
```

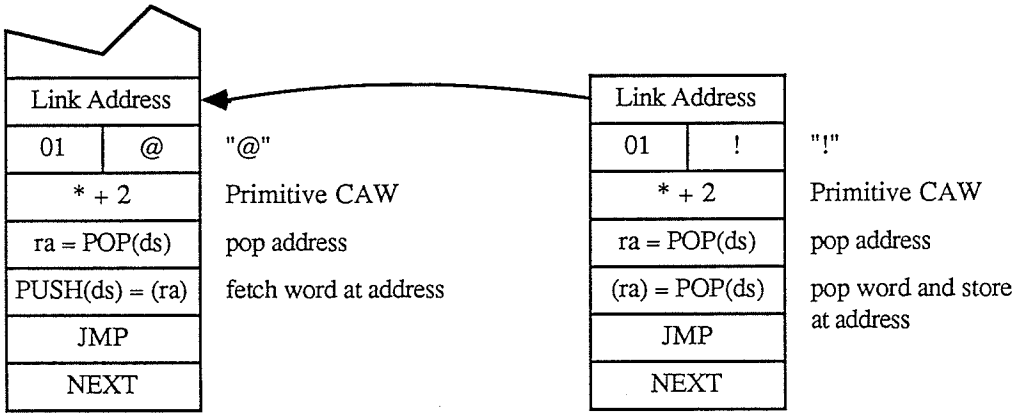


Figure 3.4: Primitives

3.5 DEFINING SECONDARIES

Secondaries are defined using the word ':' in the form

: <name> ... ;

Colon begins by setting CONTEXT to CURRENT so that the dictionary search order starts with the compilation vocabulary. Colon then executes CREATE and HIDE to create a "hidden" dictionary entry for <name>. The word address of DOCOLON is emplaced as the CAW for <name>. The last function of colon is to set STATE to compiling. Subsequent words will be compiled (or executed if immediate) by the text interpreter.

The definition is terminated by ';'. Semicolon is an immediate word which encloses the word address of DOSEMI, executes REVEAL and sets STATE to interpreting.

Colon and semicolon are defined as:

```
: :   ( -- )
   CURRENT @ CONTEXT !     ( set CONTEXT to CURRENT )
   CREATE                  ( create entry for definition )
   HIDE                    ( hide <name> temporarily )
   #DOCOLON                ( word address of DOCOLON )
   LASTCFA !               ( pointer to CAW - this )
                           ( will be explained later )
   !                       ( update CAW to DOCOLON )
   ] ;                     ( set state to compiling )

: ;   ( -- )
   #DOSEMI ,               ( enclose word address )
                           ( of DOSEMI )
   REVEAL                  ( make <name> visible )
   [                       ( set state to interpreting )
   ; IMMEDIATE             ( make ; immediate )
```

Refer back to figure 2.11 for an example of the dictionary entries compiled for the secondaries SQUARED and CUBED.

LASTCFA is a non-standard word which has been defined here since it can be used later. LASTCFA returns the code field address (CFA) of the most recent dictionary entry. The CFA points at the code address word (CAW - refer back to figure 2.3). LASTCFA can be defined as:

```
HEX                        ( numbers are in hexadecimal )
: LASTCFA   ( -- addr )
   LAST @                  ( get pointer to most recent )
                           ( dictionary entry - refer )
                           ( back to section 3.2 )
   2+                      ( skip over link field and )
                           ( point at length of name )
   DUP C@                  ( fetch byte containing length )
   7F AND                  ( mask off any immediate flag )
   + 1+ ;                  ( add length of name field to )
                           ( obtain pointer to CAW )
```

3.5.1 EXITing a Word

EXIT is an immediate word that encloses the word address of DOSEMI into the dictionary. It is normally used in conjunction with an IF statement (see section 3.8.1) to allow a conditional return prior to reaching the implicit return at "end-of-definition".

EXIT cannot be used within a DO loop construct since DO uses the return stack for maintaining temporary variables (see section 3.8.3). The return address within a DO loop is undefined.

EXIT is defined as:

```
: EXIT   ( -- )
   #DOSEMI ,              ( enclose word address )
                          ( of DOSEMI )
   ; IMMEDIATE            ( make EXIT immediate )
```

3.6 CONSTANTS AND VARIABLES

Constants and variables are part of a group of words which are neither primitives nor secondaries. These words have passive code bodies as opposed to primitives and secondaries (which have active code bodies). A passive code body contains data whereas an active code body contains machine code or threaded code which performs some action. The CAW for a passive code body always points to prologue code which manipulates the data.

Constants and variables are very similar in structure. This is shown by the dictionary entries compiled for:

```
5 CONSTANT MAX_OK    ( and )
VARIABLE APPLES_OK
```

as listed in figure 3.5. The variable or constant data is stored as the first (and only) word in the code body (also known as the parameter field). The dictionary entries for CONSTANT and VARIABLE will be explained in the following section.

As always, the CAW determines the word type (colon definition, constant, variable, etc.). All words of the same type share the same prologue code and have identical CAWs. A constant's CAW points to machine code which extracts the value from the parameter field of the word and pushes this onto the stack. In variables, the prologue code simply returns the address of the parameter field (referred to as the PFA - parameter field address). The fetch (@) and store (!) operators are then used to access the variable as follows:

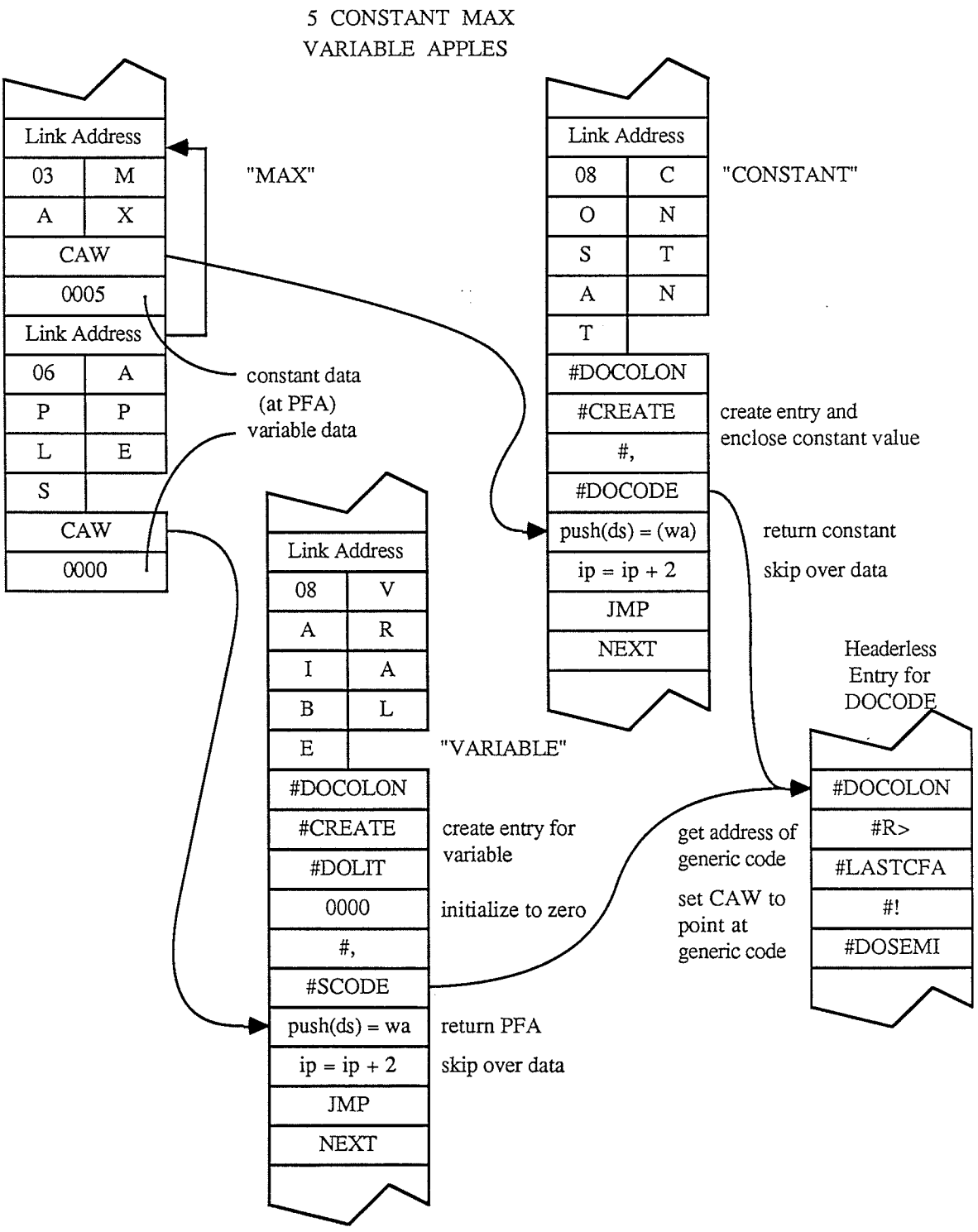
```
10 APPLES ! OK          ( pushes 10 and the PFA of )
                        ( APPLES to the stack )
                        ( ! stores 10 into the )
                        ( parameter field of APPLES )

APPLES @ OK             ( pushes the PFA of APPLES to )
                        ( the stack and calls @ to )
                        ( fetch the value from APPLES )
                        ( parameter field and push it )
                        ( <10> onto the stack )
```

CONSTANT and VARIABLE may be defined using ;CODE (see next section) as in:

```
: CONSTANT    ( n1 -- )
   CREATE                    ( create entry for constant )
   ,                         ( enclose value of constant )
   ;CODE                     ( set CAW to point at the )
                             ( following code )
   ds PUSH wa ( =            ( fetch word from parameter )
                             ( field to stack )
   NEXT END-CODE

: VARIABLE    ( -- )
   CREATE                    ( create entry for variable )
   0 ,                       ( enclose value of zero )
   ;CODE                     ( set CAW to point at the )
                             ( following code )

   ds PUSH wa =              ( return PFA of variable entry )

   NEXT END-CODE
```

Figure 3.5: CONSTANTs and VARIABLEs

Definitions:



3.7 LOW LEVEL DEFINING WORDS

Forth's most powerful facility is its ability to create defining words. Each defining word can create a new class of words with similar properties. As an example, the predefined word CONSTANT is a defining word used to create words which return a constant value. New defining words can be created as necessary. This allows the programmer to define new word classes such as special purpose data or control structures.

A defining word <defname> is created in the form

: <defname> defining code ;CODE generic code END-CODE

using high level defining code and assembly language generic code. <defname> can define a new word <name> by saying:

parameters(s) <defname> <name> (as in 5 CONSTANT MAX)

This creates an entry for <name> and sets <name>'s CAW to point at the generic code in <defname>. When <name> is invoked it will execute the generic code.

Compilation of <defname> proceeds as normal until ;CODE is encountered. ;CODE is an immediate word which executes REVEAL, encloses the word address of DOCODE into the dictionary, sets STATE to interpreting, and changes to the ASSEMBLER vocabulary. (DOCODE will be executed later when its word address is encountered during the execution of <defname>.) The generic code makes up the last portion of the dictionary entry for <defname>. Its object code is enclosed using assembler words, and is terminated by END-CODE.

When <defname> is executed, its defining code calls CREATE directly or indirectly to create a dictionary entry for <name>. DOCODE then alters the CAW of <name> to point to the generic code following ;CODE (which is enclosed following the word address of DOCODE in the dictionary entry for <defname> - see figure 3.5). This generic code is machine code which performs the function common to all words defined using <defname>.

DOCODE operates in the following manner. After DOCODE is invoked, the return stack will contain a pointer to the generic code (since it follows the word address of DOCODE, and DOCODE is a secondary). DOCODE can then pop this pointer from the return stack and store it in the CAW of the most recent dictionary entry (which will be <name>). When DOCODE completes it will return and exit from <defname>.

;CODE and DOCODE are defined as

```
: ;CODE   ( -- )
          ( sys1 -- sys2 compiling )
   REVEAL                  ( make <defname> visible )
   #DOCODE ,               ( enclose word address )
                           ( of DOCODE )
   [                       ( set state to interpreting )
   ASSEMBLER               ( set CONTEXT to assembler )
   ; IMMEDIATE             ( make ;CODE immediate )

: DOCODE   ( -- )
   R>                      ( pop return stack to get )
                           ( pointer to generic code )
   LASTCFA                 ( get pointer to CAW )
   ! ;                     ( store generic pointer into CAW )
```

A detailed example of the use of ;CODE was given in the previous section in which CONSTANT and VARIABLE were defined.

;CODE may check the stack to determine if CODE was used (sys1 is present) and leave sys2 for verification by END-CODE.

3.8 CONTROL STRUCTURES

Forth's control structures are implemented as immediate words. These words generate branch addresses and enclose code field addresses (CFAs) for primitives which perform the branch on false or unconditional branch operation. These primitives represent the branch instructions of the "virtual machine".

Control structures are only used within colon definitions as in:

: <name> ...[15] control structure ... ;

Most systems provide syntax checking to ensure proper use of control structures. This section discusses the implementation of control structures without specifying the details of syntax checking.

It is sufficient to note that each control word is responsible for verifying its own proper use. Each word typically places some special value ("sys") onto the stack. An error exists if the subsequent control word does not find the expected value. This simple approach is often used to

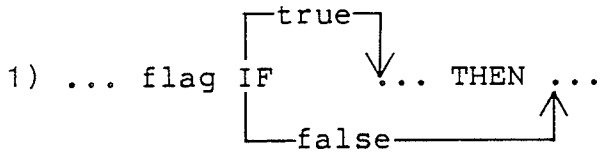
---

[15] The sequence ... is used to denote a series of words and/or numbers. This may include nesting of control structures.

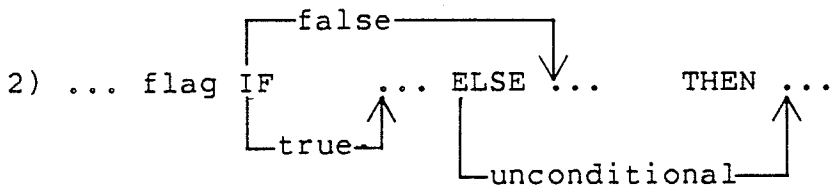
perform basic syntax checking at compile time.

3.8.1 IF Statements

The simplest control structure is the IF statement. Its syntax is:



OR



Code located immediately before the IF is an expression which returns a boolean flag of value true (an integer not equal to zero) or false (an integer equal to zero). If the flag is true then the words between IF and THEN (in the first case) or IF and ELSE (in the second case) are executed. If the flag is false then, in the second case, the words between ELSE and THEN are executed. In either case, execution continues at the word following THEN. The branch is taken only if the flag is false.

IF performs three functions at compile time. First it encloses the word address of DOIF (a primitive that performs the test and branch at execution time). Second it executes

HERE to place the address of the next available dictionary location onto the stack. Last it reserves two bytes (2 ALLOT or 0 ,) to store the branch address needed by DOIF. The top stack entry now contains a pointer to the branch address.

THEN assigns HERE to the branch address located by the pointer which was placed on the stack by IF (or ELSE). This updates the branch address to point at the word following THEN. At execution time, DOIF tests the flag and (if the flag is false) assigns its branch address to the TC instruction pointer. This has the effect of causing a branch to the word following THEN.

ELSE operates the same as IF except that it encloses the word address of DOBRA (to perform an unconditional branch to the word following THEN). In addition, ELSE updates DOIFs branch address in the same manner as THEN would in the case of an IF-THEN (no ELSE).

IF, THEN and ELSE are defined as:

```
: IF    ( flag -- )
        ( -- sys compiling )
   #DOIF ,                  ( enclose word address of DOIF )
   HERE                     ( point to branch address )
   0 ,                      ( temporary branch address )
   ; IMMEDIATE              ( make IF immediate )

: THEN    ( -- )
          ( sys -- compiling )
   HERE                     ( get IF or ELSE destination )
   SWAP !                   ( store this as branch )
                            ( address for DOIF or DOBRA )
   ; IMMEDIATE              ( make THEN immediate )
```

```
: ELSE    ( -- )
          ( sys1 -- sys2 compiling )
   #DOBRA ,                ( enclose word address )
                           ( of DOBRA )
   HERE                    ( point to branch address )
   0 ,                     ( temporary branch address )
   SWAP                    ( get IF destination )
   THEN                    ( store DOIF branch address )
   ; IMMEDIATE             ( make ELSE immediate )
```

DOIF and DOBRA are headerless primitives as follows:

```
DOIF     FCW *+2                 primitive CAW
         ra = POP(ds)            get flag from stack
         IF ra = 0 BRA DOIF1     branch if flag is false
         ip = ip + 2             skip over branch offset and
                                 continue at next word
         JMP NEXT
DOIF1    ip = (ip)               perform branch on false
                                 by setting ip = branch address
         JMP NEXT

DOBRA    FCW *+2                 primitive CAW
         ip = (ip)               perform unconditional branch
                                 by setting ip = branch address
         JMP NEXT
```

Note that DOIF could share DOBRA's code. Code can be shared in many other instances. This would produce a smaller kernel at the expense of slower execution.

An example of the TC produced by IF-THEN and IF-ELSE-THEN is shown in figure 3.6.

Definitions:

```
: MAX < IF SWAP THEN DROP ;
: MIN < IF DROP ELSE SWAP DROP THEN ;
```

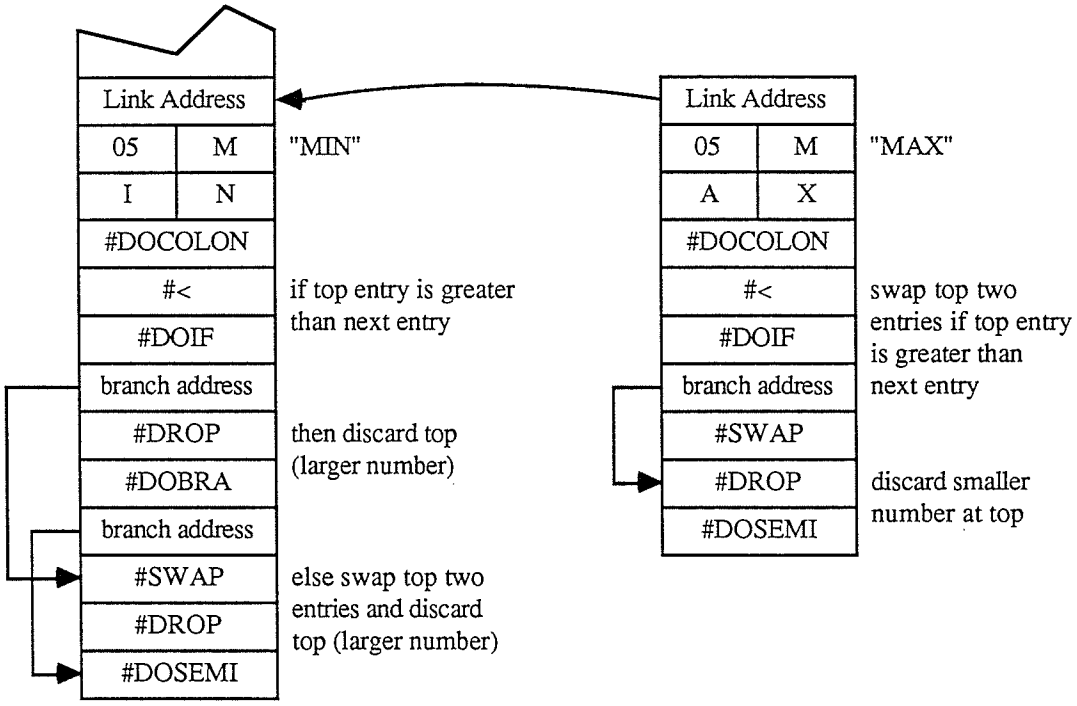
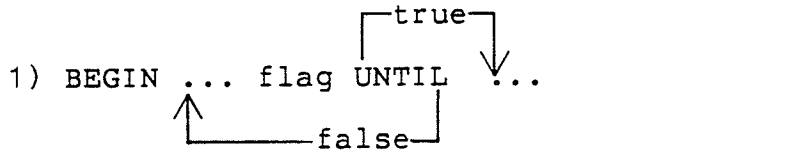


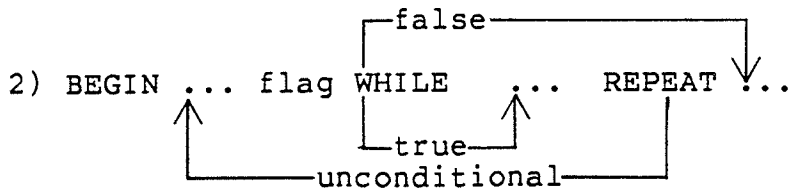
Figure 3.6: IF-THEN and IF-ELSE-THEN

3.8.2 BEGIN Loops

BEGIN loops are of the form:



OR



In the first case the code between BEGIN and UNTIL is repetitively executed as long as the flag remains false. If the flag is true then execution continues at the word following UNTIL. The body of the loop is executed at least once.

In the second case the code between BEGIN and WHILE typically contains only an expression used to the derive a flag. If the flag is true then the code between WHILE and REPEAT is executed followed by an unconditional jump back to the word after BEGIN. If the flag is false, execution continues at the word following REPEAT. The body of the loop may execute zero or more times.

BEGIN simply executes HERE to mark the beginning of the loop. UNTIL encloses the word address of DOIF and the branch address which was placed on the stack by BEGIN. If the flag is false then DOIF assigns its branch address to the TC instruction pointer. This causes a backwards branch to the word following BEGIN.

Systems with an 8-bit processor often use two versions of DOIF and DOBRA. These allow relative branches of up to 255 bytes in either direction. IF and ELSE use the forward versions (the offset is added to the TC instruction pointer) while UNTIL and REPEAT use the backward ones (the offset is subtracted). This produces a more compact and faster branch instruction. The generic implementation presented here does

not use 8-bit branch offsets for the same reason that 8-bit literals were not used (refer to section 3.1).

WHILE and REPEAT function exactly like IF and ELSE except that REPEAT does a backward branch to the word following BEGIN whereas ELSE does a forward branch to the word following THEN. Some non-standard systems use IF as a direct replacement for WHILE.

BEGIN, UNTIL, WHILE and REPEAT are defined as:

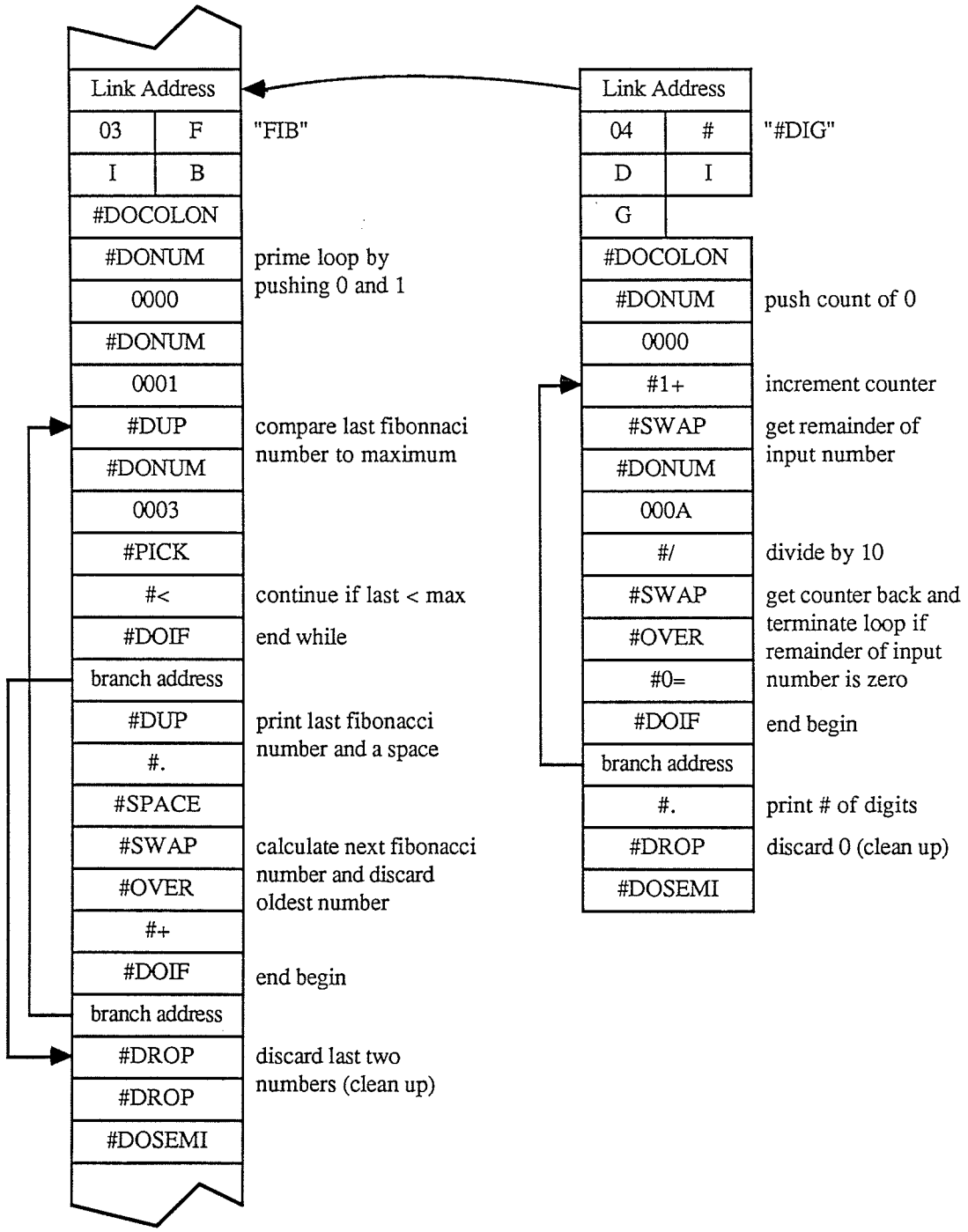
```
: BEGIN    ( -- )
           ( -- sys compiling )
   HERE                      ( point to start of loop )
   ; IMMEDIATE               ( make BEGIN immediate )

: UNTIL    ( flag -- )
           ( sys -- compiling )
   #DOIF ,                   ( enclose word address of DOIF )
   ,                         ( enclose branch address )
   ; IMMEDIATE               ( make UNTIL immediate )

: WHILE    ( flag -- )
           ( sys1 -- sys1 compiling )
   #IF , ; IMMEDIATE         ( has basically the same )
                             ( function as IF )

: REPEAT    ( -- )
            ( sys -- compiling )
   SWAP                      ( get BEGIN destination )
   #DOBRA ,                  ( enclose word address )
                             ( of DOBRA )
   ,                         ( enclose branch address )
   THEN                      ( store DOIF branch address )
   ; IMMEDIATE               ( make REPEAT immediate )
```

See figure 3.7 for examples of the BEGIN loop structures.

Figure 3.7: BEGIN-UNTIL and BEGIN-WHILE-REPEAT

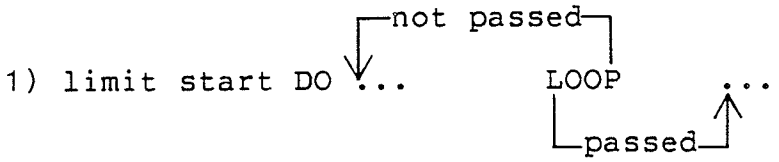
Definitions:

```
: FIB 0 1 BEGIN DUP 3 PICK < WHILE
      DUP . SPACE SWAP OVER + REPEAT DROP DROP ;

: #DIG 0 BEGIN 1+ SWAP 10 / SWAP OVER 0= UNTIL . DROP ;
```

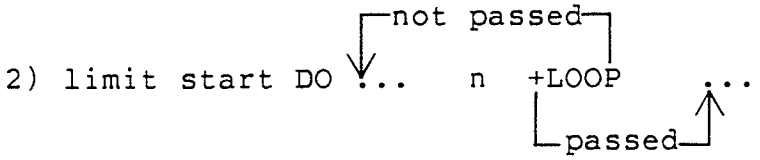


3.8.3 DO Loops

DO loop constructs have the syntax:



OR



Both forms of the DO loop are counted loops which expect a limit and start value to be placed on the stack. A loop index is assigned the start value and is incremented by one (for LOOP) and by n (for +LOOP). The loop is terminated when the index is incremented across the boundary from limit-1 to limit. When the loop terminates, execution continues at the word following LOOP (or +LOOP). Otherwise, execution continues at the word following the corresponding DO. The body of each loop is executed at least once.

The loop index can be accessed via the word I as in:

DO ... I ... LOOP (or) DO ... I ... +LOOP

I pushes the value of the current loop index onto the stack. J can be used to access the index of the next outer loop:

DO ... DO ... J ... +LOOP ... LOOP

Another word, LEAVE, may be used to prematurely exit a DO. Prior to FORTH-83, LEAVE was defined to terminate the loop upon encountering the next LOOP or +LOOP. This has been changed so that LEAVE now causes an immediate exit from the loop. LEAVE is normally used together with an IF statement as a means of conditionally exiting a DO loop as in:

```
DO ... flag IF ... LEAVE THEN ... LOOP ...
```

A few examples should clarify the use of DO loops:

```
: COUNT   ( end start -- )
   DO                          ( for i = start to end-1 )
      I . SPACE                ( print i,' ' )
   LOOP ; OK

: COUNTBY2   ( end start -- )
   DO                          ( for i = start to end-1 by 2 )
      I . SPACE                ( print i,' ' )
   2 +LOOP ; OK

: TABLE   ( n -- )
   1+ 1 DO                     ( for i = 1 to end )
      CR                       ( print carriage return )
      I 0 DO                   ( for i = 0 to i-1 )
         I J - . SPACE         ( print i-j,' ' )
      LOOP
   LOOP ; OK

: SUM-UP-TO   ( n -- )
   0                           ( current sum )
   1000 1 DO                   ( for i = 1 to 999 )
      I +                      ( sum = sum + i )
      OVER OVER <=             ( if input sum <= sum then )
      IF                       ( exit DO loop after )
         SWAP DROP             ( discarding input sum and )
         I SWAP                ( push i value )
      THEN
   LOOP
   . ."  in " . ; OK           ( print sum,' in ',i )

5 0 COUNT 0 1 2 3 4 OK

5 5 COUNT 5 6 7 ...
65534 65535 0 1 2 3 4 OK

10 3 COUNTBY2 3 5 7 9 OK
```

```
3 TABLE
-1
-2 -1
-3 -2 -1 OK

25 SUM-UP-TO 28 in 7 OK
```

Note that the sequence n DUP DO ... LOOP (as in the case of 5 5 COUNT) will cause the loop to execute 65,536 times - looping through all possible 16-bit indices.

At compile time, DO encloses the word address of DODO and executes HERE to mark the beginning of the loop. It then reserves two bytes to hold the branch address required by LEAVE. LOOP (or +LOOP) closes the loop by enclosing the word address of DOLOOP (or DO+LOOP) followed by the branch address which was placed on the stack by DO. In addition, the branch address following DODO is set to HERE (which points just beyond the end of the loop). LEAVE uses this address to exit from the DO loop.

DO loops use the return stack to maintain a copy of the LEAVE branch address, the loop index and the limit. At execution time, DODO pushes its LEAVE branch address, the limit and the start index onto the return stack. DOLOOP increments the current loop index (top of return stack) and compares it to the limit (second return stack entry). If the two are equal then the TC instruction pointer is incremented to continue execution at the word following LOOP. Otherwise, the branch address is assigned to the TC instruction pointer to effect a backward jump to the word following the

corresponding DO. DO+LOOP operates in a similar manner except that the loop index is incremented by n and the loop condition is more complicated to test.

I and J return the loop indices at the top of the return stack and the fourth entry respectively. LEAVE encloses the word address of DOLEAVE which pops the loop index, limit and branch address from the return stack. It then does an unconditional branch to the LEAVE branch address (immediately exits the DO loop). Older versions of DOLEAVE could simply set the loop index to limit-1 which would ensure loop termination on the next LOOP or +LOOP.

The function of the DO loop can be improved by adjusting the loop indices in order to simplify the branch condition. Instead of pushing [limit , start] we can push [limit + HEX 8000 , start - (limit + HEX 8000)]. For example, 5 2 DO ... LOOP would push the LEAVE branch address, HEX 8005 and HEX 7FFD to the return stack. DOLOOP and DO+LOOP then increment the loop index (top entry on the return stack) and branch as long as there is no arithmetic overflow. An overflow occurs when a number is incremented across the boundary from HEX 7FFF (the largest positive number) to HEX 8000 (the largest negative number). This reflects the boundary condition which terminates a DO loop (the index crossing from limit-1 to limit).

The above modification requires that I and J return the sum of the loop limit and loop index. This addition calculates the true value of the loop index: start = limit + HEX 8000 + (start - (limit + HEX 8000)) so that the (limit + HEX 8000) terms nullify each other. The overall effect is a DO loop which performs faster iterations at the expense of slightly slower start-up and indice access times.

Another method of implementing LEAVE is discussed in [Feu85] and [Hay85]. The goal of this implementation is that each LEAVE be compiled to DOLEAVE and a pointer to the word following LOOP (or +LOOP). This representation seems intuitively "ideal". The LEAVE branch address need not be kept on the return stack and J now returns the third return stack entry. At compile time, LEAVE encloses the word address of DOLEAVE and a pointer to the previous DOLEAVE pointer within the same loop. This forms a linked list of pointers. LOOP (or +LOOP) can traverse this list and update the DOLEAVE pointers to their proper values. At execution time, DOLEAVE removes the loop index and limit and does an unconditional branch to the pointer that follows. This version of LEAVE was not chosen since it increases compile time and complexity, and requires an additional 2(n-1) bytes for n LEAVEs within a loop. However, it does have the advantage of a slightly faster DODO.

DO loop control words may be defined as:

```
: DO    ( end start -- )
        ( -- sys compiling )
   #DODO ,                  ( enclose word address of DODO )
   0 ,                      ( reserve LEAVE branch address )
   HERE                     ( point to start of loop )
   ; IMMEDIATE              ( make DO immediate)

: LOOP    ( -- )
          ( sys -- compiling )
   #DOLOOP ,                ( enclose word address )
                            ( of DOLOOP )
   DUP ,                    ( enclose branch address )
   2- HERE SWAP !           ( set LEAVE branch address )
   ; IMMEDIATE              ( make LOOP immediate )

: +LOOP    ( incr -- )
           ( sys -- compiling )
   #DO+LOOP ,               ( enclose word address )
                            ( of DO+LOOP )
   DUP ,                    ( enclose branch address )
   2- HERE SWAP !           ( set LEAVE branch address )
   ; IMMEDIATE              ( make +LOOP immediate )

: LEAVE    ( -- )
   #DOLEAVE ,               ( enclose word address )
                            ( of DOLEAVE )
   ; IMMEDIATE              ( make LEAVE immediate )

CODE I    ( -- index )
   ra rs ( =                ( get top return stack entry )
   ra ra rs 2 (D + =        ( add 2nd return stack entry )
                            ( to calculate true index )
   ds PUSH ra =             ( push index onto stack )
   NEXT END-CODE

CODE J    ( -- index )
   ra rs 6 (D =             ( get 4th return stack entry )
   ra ra rs 8 (D =          ( add 5th return stack entry )
                            ( to calculate true index )
   ds PUSH ra =             ( push index onto stack )
   NEXT END-CODE
```

DODO, DOLOOP, DO+LOOP and DOLEAVE are headerless primitives as follows:

```
DODO     FCW *+2                 primitive CAW
         ra = POP(ds)            get start value
         rb = POP(ds)            get limit value
         PUSH(rs) = (ip++)       push LEAVE address onto
                                 return stack
         rb = rb + HEX 8000      adjust limit value
         PUSH(rs) = rb           push limit onto return stack
         ra = ra - rb            adjust start value
         PUSH(rs) = ra           push start onto return stack
         JMP NEXT

DOLOOP   FCW *+2                 primitive CAW
         (rs) = (rs) + 1         increment index by 1
         IF OVERFLOW BRA DOLP1   test exit condition
         ip = (ip)               branch back to continue loop
         JMP NEXT
DOLP1    rs = rs + 6             terminate loop - remove index
                                 and limit from return stack
         ip = ip + 2             skip over offset and continue
                                 execution at next word
         JMP NEXT

DO+LOOP  FCW *+2                 primitive CAW
         ra = POP(ds)            get increment value
         (rs) = (rs) + ra        add increment to index
         IF OVERFLOW BRA DOLP1   test exit condition
         ip = (ip)               branch back to continue loop
         JMP NEXT

DOLEAVE  FCW *+2                 primitive CAW
         rs = rs + 4             terminate loop - remove index
                                 and limit from return stack
         ip = POP(rs)            branch to LEAVE address
         JMP NEXT
```

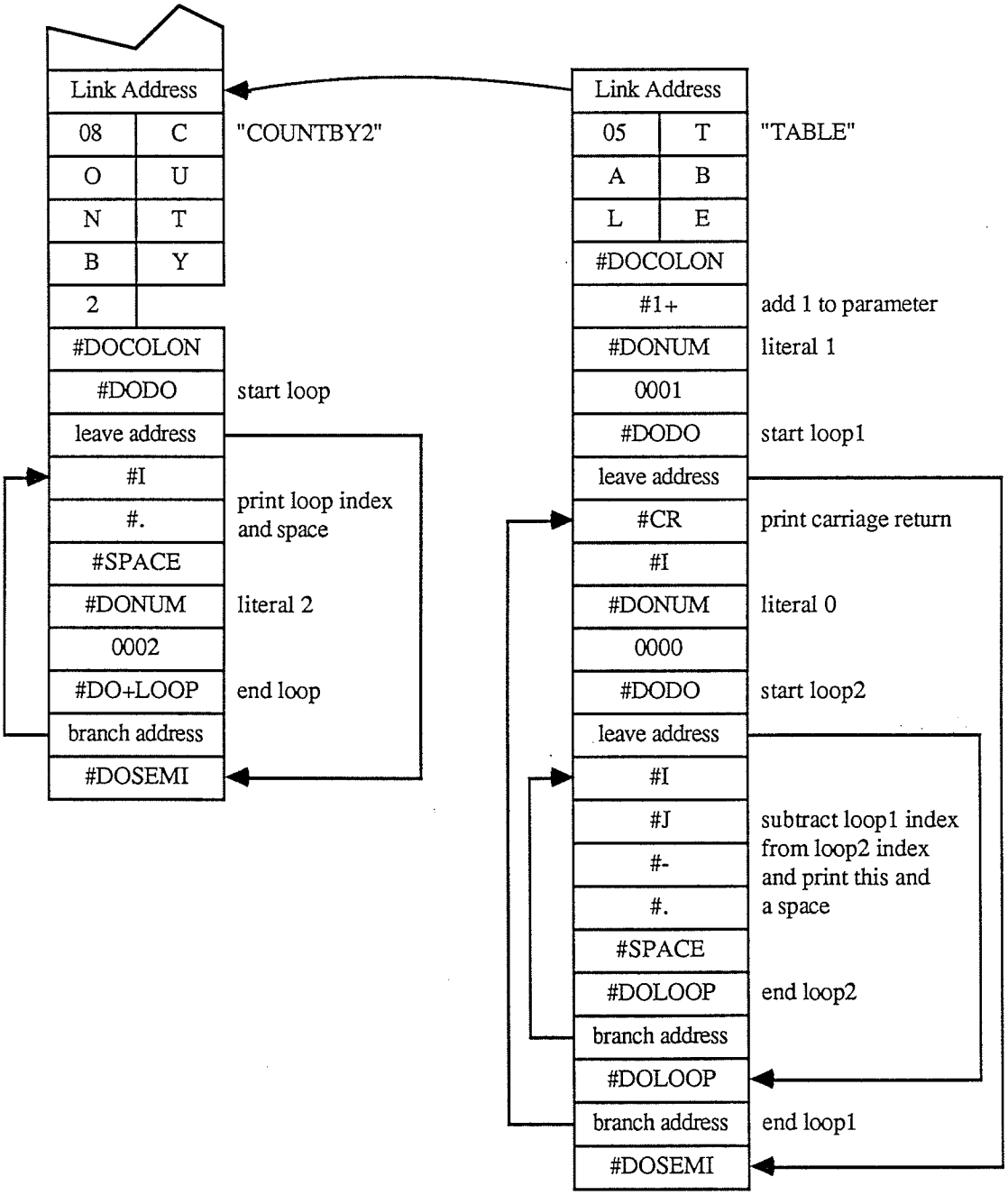
Figure 3.8 shows an example of dictionary entries compiled for DO loop constructs.

Figure 3.8: DO-LOOP and DO-+LOOP

Definitions:

: COUNTBY2 DO I . SPACE 2 +LOOP ;

: TABLE 1+ 1 DO CR I 0 DO I J - . SPACE LOOP LOOP ;



3.9 TERMINAL INPUT AND OUTPUT

Several words are provided for doing terminal input and output. EXPECT and TYPE were discussed in section 2.5. EXPECT accepts input from the keyboard and places this into a character buffer. TYPE is given the address and length of a character buffer which it outputs to the terminal screen. The remaining words are used to perform pictured numeric output or simple character I/O.

Character I/O is based on the words KEY and EMIT. KEY returns an ASCII character when a key is pressed. EMIT is passed a value which is output to the terminal screen as an ASCII character. The words CR (perform a carriage return and linefeed), SPACE (print a blank), SPACES (print n blanks) and TYPE are defined using EMIT. EXPECT requires both KEY and EMIT. As an example, CR could be defined as:

```
HEX                         ( numbers are in hexadecimal )
: CR   ( -- )
   0D EMIT                  ( print carriage return )
   0A EMIT ;                ( print linefeed )
```

Pictured numeric output is provided by a set of words which use the PAD (as described in section 2.3.4) for temporary storage of ASCII data. Digits are converted and stored to the PAD in right to left order. The variable HLD is used as a pointer to the most recent digit stored in the PAD. HLD is initialized by the word <# ("less-sharp").

Conversion assumes a 32-bit number is present on the stack. The word # ("sharp") generates the next digit by dividing the number by BASE, converting the remainder to an ASCII character and storing this into the PAD. The quotient remains on the stack (from which the next digit can be calculated). #S ("sharp-s") converts the remaining digits by calling # until the quotient is zero. #> ("sharp-greater") ends conversion by discarding the quotient and returning the address and length of the ASCII string which was accumulated into the PAD. This result is suitable for use with TYPE.

An ASCII character is output to the PAD using the word HOLD. SIGN generates a minus sign by calling '-' HOLD if the top stack entry is negative. The word . ("dot") is used to print a single 16-bit number. U. prints an unsigned 16-bit number. Pictured numeric output words are used as shown in the following example which prints an 32-bit integer as a date in the form yy/mm/dd:

```
: DATE   ( date -- )
   <#                        ( begin numeric conversion )
   # #                       ( convert dd digits )
   ASCII / HOLD              ( insert "/" )
   # #                       ( convert mm digits )
   ASCII / HOLD              ( insert "/" )
   # #                       ( convert yy digits )
   #>                        ( end numeric conversion )
   TYPE ; OK                 ( and print result )

122586. DATE 12/25/86 OK
```

Note the use of the word ASCII to push an ASCII character value onto the stack. ASCII / returns HEX 5C. Numeric output words can be defined as:

```
VARIABLE HLD                 ( pointer into PAD )
```

```
: <#    ( -- )
   PAD HLD ! ;                ( set HLD to point at PAD )

: #>    ( d1 -- addr n1 )
   2DROP                      ( discard 32-bit quotient )
   HLD @                      ( get address of PAD string )
   PAD OVER - ;               ( calculate string length )

: HOLD    ( char -- )
   -1 HLD +!                  ( decrement HLD )
   HLD @ C! ;                 ( store byte at HLD )

: SIGN    ( n -- )
   0< IF                      ( check if number is negative )
      ASCII - HOLD            ( insert "-" )
   THEN ;

( divides an unsigned 32-bit number by an unsigned )
( 16-bit number and returns an unsigned 16-bit     )
( remainder and an unsigned 32-bit quotient        )
: MU/MOD    ( ud1 u1 -- u2 ud2 )
   >R                         ( divisor onto return stack )
   0                          ( add zero extension )
   R@                         ( fetch divisor )
   UM/MOD                     ( divide 32-bit number by )
                              ( 16-bit number to get 16- )
                              ( bit remainder and quotient )
   R>                         ( fetch divisor )
   SWAP >R                    ( save partial quotient )
   UM/MOD                     ( get remainder and quotient )
   R> ;                       ( append partial quotient to )
                              ( form 32-bit result )

: #    ( +d1 -- +d2 )
   BASE @ MU/MOD              ( divide 32-bit number by BASE )
   ROT                        ( get remainder as a digit )
   9 OVER <                   ( convert digit to ASCII )
   IF 7 + THEN
   ASCII 0 HOLD ;             ( store digit to PAD )

: #S    ( +d1 -- 0 0 )
   BEGIN
      #                       ( convert a digit )
      2DUP D0=                ( continue conversion until )
   UNTIL ;                    ( 32-bit remainder is zero )

: .    ( n -- )
   DUP                        ( save copy of number )
   ABS                        ( take absolute value )
   0                          ( extend to 32-bit number )
   <# #S                      ( convert number to ASCII )
   ROT SIGN                   ( check copy for number sign )
   #> TYPE ;                  ( end conversion and print )
```

```
: U.    ( u -- )
   0                          ( extend to 32-bit number )
   <# #S #> TYPE ;            ( convert and print )
```

3.10 HIGH LEVEL DEFINING WORDS

High level defining words are similar to low level defining words; they allow the creation of new defining words. A high level defining word <defname> is created in the form:

: <defname> defining code DOES> generic code ;

whereas low level defining words are of the form:

: <defname> defining code ;CODE generic code END-CODE

The main difference is that the generic code of high level defining words is written in high level Forth rather than assembly language.

Compilation of <defname> is the same as that of any secondary. However, the immediate word DOES> is used, like a macro, to generate additional code at compile-time. DOES> encloses the word address of DOCODE (as in ;CODE), followed by machine code which causes execution to continue at the word following DOES> in the definition of <defname>. <defname> is used to define a new word <name> by saying:

parameter(s) <defname> <name>

as was done for low level defining words.

When <defname> is executed, its defining code creates a dictionary entry for <name>. DOCODE then alters the CAW of the most recent dictionary entry (which will be <name>).

The CAW is set to point at the machine code which was enclosed in <defname> by DOES>. This machine code acts like DOCOLON but pushes the PFA of <name> onto the stack and sets the TC instruction pointer to point at the word following DOES> in <defname>.

Thus when <name> is executed, the PFA of <name> is placed onto the stack and the generic code in <defname> is executed. For example, CONSTANT could be defined as:

```
: CONSTANT    ( n -- )
   CREATE                  ( create entry for constant )
   ,                       ( enclose constant value )
   DOES>                   ( alter the CAW to cause )
                           ( run-time execution of the )
                           ( following code )
   @ ;                     ( fetch constant value at PFA )
                           ( of constant )
```

The definition 5 CONSTANT MAX would create a dictionary entry for MAX and enclose 5 into the parameter field. Later, when MAX is executed, the PFA is placed on the stack and @ is executed to extract the value of the constant. Thus MAX would return the value 5.

In a similar fashion, the word VOCABULARY could be defined as:

```
: VOCABULARY    ( -- )
   CREATE                  ( create entry for vocabulary )
   LAST @                  ( get pointer to the entry )
                           ( just created )
   ,                       ( enclose this as the )
                           ( vocabulary pointer )
   DOES>                   ( alter CAW for generic code )
   CONTEXT ! ;             ( set CONTEXT to PFA so that )
                           ( CONTEXT points at this )
                           ( vocabulary variable )
```

This allows the creation of new vocabularies as previously discussed in section 2.3.1.

DOES> can be defined as:

```
: DOES   ( -- addr )
   #DOCODE ,                ( enclose word address )
                            ( of DOCODE )
   #DODOES                  ( copy DODOES machine code )
   HERE 10 CMOVE            ( into <defname> )
   10 ALLOT                 ( and enclose this code )
                            ( in the dictionary )
   ; IMMEDIATE              ( make DOES> immediate )
```

DODOES occupies 10 bytes of machine code:

```
DODOES  PUSH(rs) = ip       save TC return address
        PUSH(sp) = wa       return PFA
        ip = pc + 6         setup to continue execution
                            of TC following DOES>
        JMP NEXT
```

The instruction "ip = pc + 6" is used to set the TC instruction pointer to point at the generic TC which follows "JMP NEXT" in the body of <defname> (the word following DOES>). This instruction may need to be broken into several steps such as "ip = pc" and "ip = ip + 8" depending on the target machine.

An alternative implementation of DOES> can be made if the target machine has a jump to subroutine (JSR) instruction which pushes its return address onto the stack. In this case, DOES> can enclose a single "JSR DODOES" instruction, rather than copying all the DODOES code. DODOES can now use the return address (from JSR) as a pointer to the generic TC in <defname>.

The modified DOES> and DODOES are as follows:

```
   HEX                          ( numbers are in hexadecimal )
   : DOES    ( -- addr )
      #DOCODE ,                 ( enclose word address )
                                ( of DOCODE )
      xxxx ,                    ( enclose machine opcode )
                                ( for JSR instruction )
      #DODOES ,                 ( enclose address of DODOES )
                                ( as the operand of JSR )
      ; IMMEDIATE               ( make DOES> immediate )

DODOES  PUSH(rs) = ip           save TC return address
        ip = POP(sp)            set ip to return address
                                of JSR instruction
                                (points at generic code)
        PUSH(sp) = wa           return PFA
        JMP NEXT
```

Prior to FORTH-83, high-level defining words were created in the form:

: <defname> <BUILDS defining code DOES> generic code

The difference in syntax does not affect the function of <defname>. <BUILDS was defined as:

: <BUILDS 0 CONSTANT

to create a word and reserve a pointer (initially zero) at the start of the parameter field. DOES> would then set the CAW to point at DODOES and update the zero pointer to point at the generic code. DODOES would use this pointer to locate the generic code at run-time.

3.10.1 Examples

High level defining words are often used to create new data structures or even new control structure words. An example is a vector; a one-dimensional array. A new defining word could be added to facilitate the definition of vectors:

```
: VECTOR   ( index -- addr )
   CREATE                    ( create new word )
   2* ALLOT                  ( allocate element storage )
  DOES>
   SWAP 2* + ; OK            ( calculate element address )
```

Vectors can now be defined by saying n VECTOR <name>. This defines a vector <name> and allocates 2n bytes in <names>'s parameter field. This allows up to n integer elements to be stored in <name>. At execution time, <name> expects a subscript I (from zero to n-1) and returns the address of the Ith element in <name>. This address is calculated as the PFA of <name> (returned by DOES>) plus I*2. For example:

```
100 VECTOR Y OK              ( define Y with 100 elements )
                             ( integer Y[100] )
14 25 Y ! OK                 ( store 14 into 25th element )
                             ( Y[25] = 14 )
25 Y @ . 14 OK               ( fetch and print Y[25] )
```

The above definition of VECTOR does not do any subscript range checking; this feature could be added by saying:

```
: VECTOR   ( index -- addr )
   CREATE                    ( create entry )
   DUP ,                     ( save vector size #bytes in )
                             ( entry's parameter field )
   2* ALLOT                  ( allocate storage )
  DOES>
   OVER OVER                 ( duplicate subscript and PFA )
   @ < IF                    ( if subscript < vector size )
      2* + 2+                ( compute address of element )
                             ( skip over size field )
   ELSE                      ( subscript too large or negative )
      ." BAD SUBSCRIPT "     ( print error message )
      SWAP DROP              ( discard bad subscript )
      2+                     ( return address of first element )
   THEN ;
```

Run-time error checking of this type is normally added only during the debugging phase (if necessary). The programmer controls the manner in which the error is handled. In the

above example, additional information could be printed to fully describe the error. A general purpose error-handling routine might be called, or program execution may be halted.

If the application required vector operators then the definition of VECTOR could be changed to return only the PFA. Operators would then be passed the address of a vector so that you could say X Y V+, where X and Y are vectors and V+ performs the vector addition X = X + Y.

Yet another alternative would be to define ARRAY which would allow the definition of an N-dimensional array. VECTOR could then be replaced by n 1 ARRAY <name> versus n VECTOR <name>.

An example of the dictionary entry produced by DOES> is shown in figure 3.9.

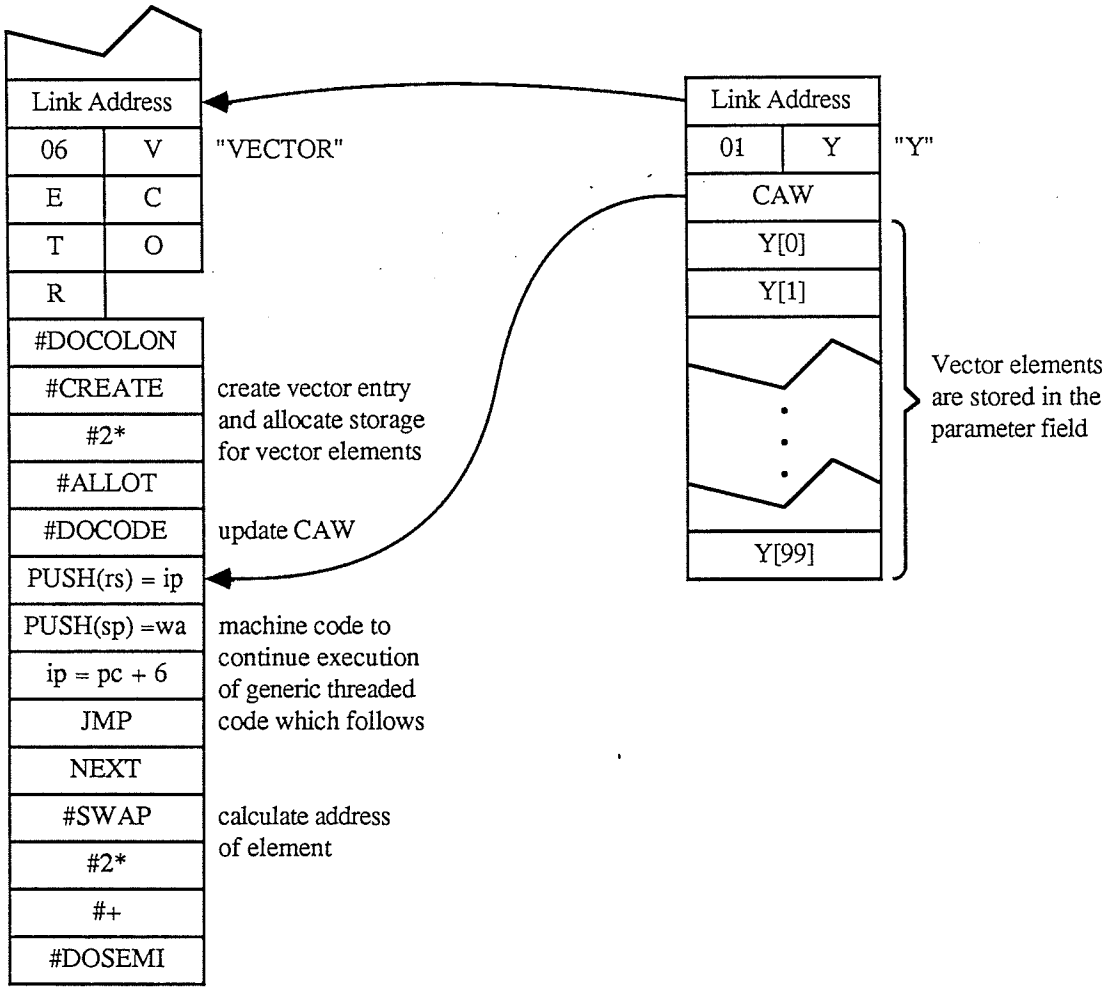
Virtual data types can be created by combining high level defining words with calls to the block storage system. Arrays were previously defined to allocate a parameter field for storing array elements. Virtual arrays are similar except that they allocate blocks for storing array elements. This allows the definition of very large arrays (limited only by the available backing store - not by the amount of main memory). As an example:

```
VARIABLE NEXT-BLOCK OK      ( next available block number )
100 NEXT-BLOCK ! OK         ( start at block 100 )

( allocate n blocks )
: BLOCK-ALLOT   ( n -- )
   NEXT-BLOCK @ +           ( add n to NEXT-BLOCK )
   + NEXT-BLOCK ! ; OK
```

Figure 3.9: A High level Definition using DOES>

Definitions:

```
: VECTOR CREATE 2* ALLOT DOES> SWAP 2* + ;
100 VECTOR Y
```



```
( create virtual array defining word )
: VARRAY   ( index -- addr )
   CREATE                    ( create word name )
   NEXT-BLOCK @ ,            ( enclose first block number )
   1- 512 / 1+               ( calculate # blocks required )
                             ( to store array elements )
   BLOCK-ALLOT               ( allocate blocks )
  DOES>
   @                         ( at execution time get )
                             ( block # of first block )
   SWAP                      ( stack has blk# index )
   2*                        ( convert index to byte offset )
   1024 /MOD                 ( get block offset and byte )
                             ( offset within the block )
                             ( stack has blk# byteoff blkoff )
   ROT                       ( byteoff blkoff blk# )
   +                         ( get blk# containing element )
   BLOCK                     ( get addr of block buffer )
                             ( reads in block if necessary )
                             ( stack has byteoff bufaddr )
   + ; OK                    ( get addr of element within )
                             ( the block buffer )

10000 VARRAY X OK
5276 X @ OK                  ( fetch X[5276] )
4296 X ! UPDATE OK           ( store this to X[4296] )
                             ( ensure block is changed )
```

3.11 RECURSION

Recursive definitions use the word RECURSE to compile the word address of the latest dictionary entry. This allows a word to call itself (normally a word cannot be located in the dictionary until its definition has been completed). A recursive definition of the factorial function is:

```
( calculate factorial for n >= 0 )
: FACTORIAL   ( n -- n! )
   DUP IF                    ( if n > 0 then calculate )
      DUP 1- RECURSE *       ( n * factorial n )
   ELSE
      DROP 1                 ( if n = 0 return 1 )
   THEN ;
```

RECURSE is defined as:

```
: RECURSE    ( -- )
   LASTCFA ,                ( enclose word address of )
                            ( latest definition )
   ; IMMEDIATE              ( make RECURSE immediate )
```

3.12 ASSEMBLER

The ASSEMBLER vocabulary contains a set of words which allow the generation of native machine code for primitive word definitions. Assembler words are named to correspond with the appropriate mnemonic or addressing mode which they imply. These words are used to implement the assembly language. Regular Forth expressions can also be intermixed with assembler words. This allows calculation of operand addresses or immediate data.

Assembly code is written in postfix notation. All operands are pushed onto the stack. An instruction (assembler word) retrieves its operands from the stack, generates specific machine code based on these operands, and encloses this machine code into the dictionary. Each instruction acts as a compiling word. This process allows incremental assembly by each operator.

The number, type and complexity of assembler words is directly related to the instruction set of the target machine. An orthogonal instruction set will simplify the definition of assembler words. For instance, the Motorola 6809 assembler is simpler than that of the Intel 8086. A small instruction set with fewer addressing modes will also be easier to handle.

The definition of assembler words is highly dependent on the target machine. This section is only a basis for understanding the concepts behind an incremental postfix assembler. A more complete discussion of an assembler vocabulary can be found in [Loe81] and [Per83].

Assembler word sets are generally implemented by grouping instructions according to their machine code formats. Each format specifies a class of similar instructions. High level defining words are then created for each class. These words are used to define instructions. Each instruction can then be defined by opcode and format. Operand/addressing information is added in later during assembly. Fixed field values, such as register numbers, are defined as constants.

Control structures similar to those of high level Forth can be defined to provide structured assembly language constructs. The stack is used to maintain information necessary in calculating branch addresses or relative branch offsets. The structured approach largely eliminates the need for line labels. No symbol table is required (the dictionary serves this function if necessary). As an example, the word COPYSTR could be defined to copy one null terminated string to another:

```
CODE COPYSTR   ( src dest -- )
   rb ds POP =             ( rb points at destination )
   ra ds POP =             ( ra points at source )
   BEGIN
      rb (+ ra (+ =b       ( copy src byte to dest )
   0= UNTIL                ( repeat until src byte null )

   NEXT END-CODE
```

It is necessary to study an actual implementation in order to view the assembly process in greater detail. The FIG-FORTH[16] 8080 assembler [Fei81] offers a good example. The Intel 8080 is a simple 8-bit microprocessor which has only 112 instructions and four addressing modes. Instruction formats and machine registers are shown in figure 3.10 Addressing modes include:

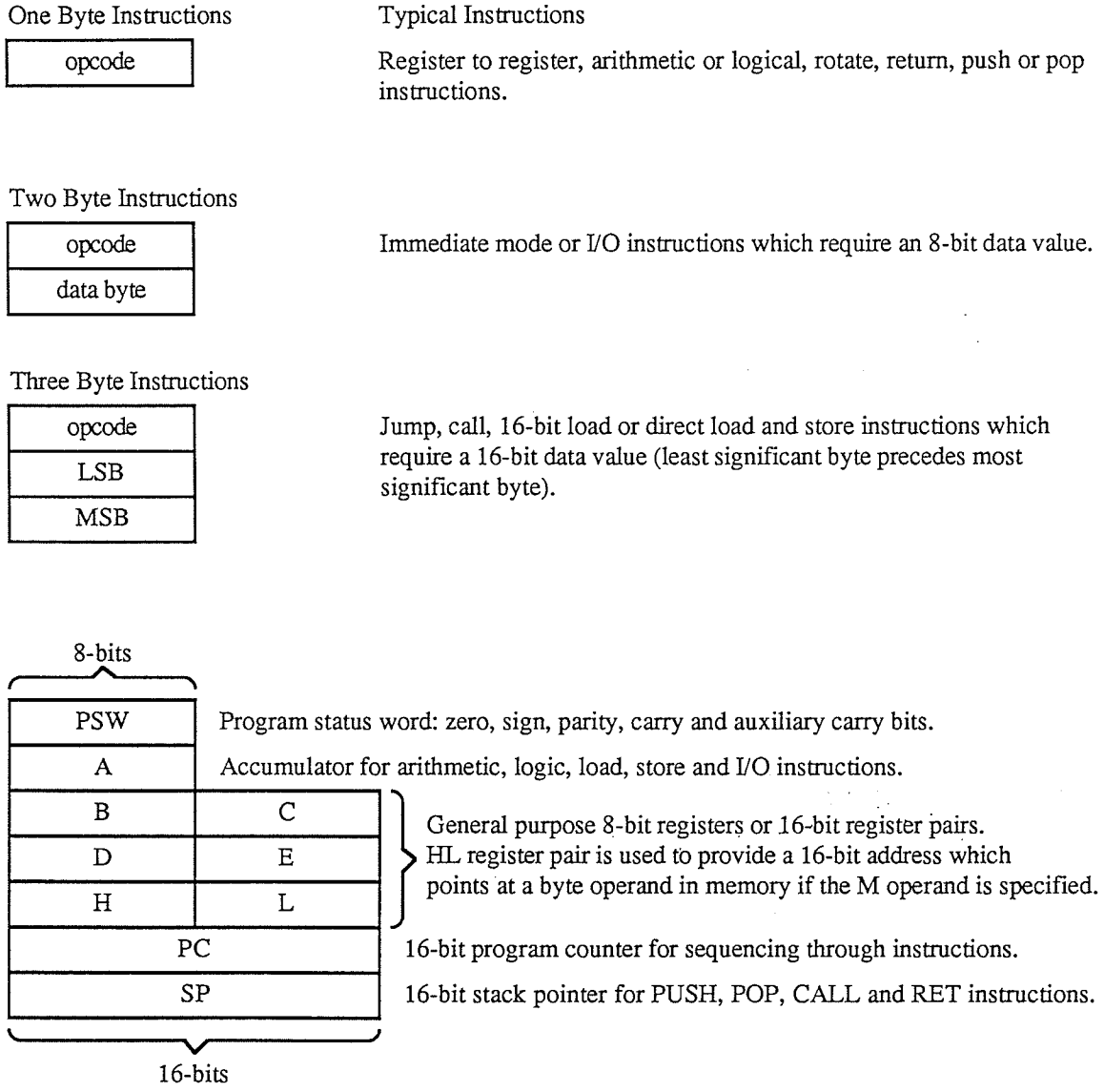
- direct - a 16-bit address follows the instruction opcode and points to a byte operand in memory.
- register - the instruction opcode specifies a register or register pair as the operand(s).
- register indirect - the instruction opcode specifies the register pair which contains a 16-bit address that points to a byte operand in memory.
- immediate - an 8-bit or 16-bit data operand follows the instruction opcode.

Indirect memory access is provided by the STAX and LDAX instructions which use either the BC or DE register pairs. Other instructions, such as ADD, implicitly use the HL register pair if the M (memory) operand is specified (rather than an 8-bit register A, B, C, D, E, H or L).

The 8080 assembler word set is defined after grouping instructions according to their general format (refer to figure 3.11). Many instruction opcodes contain bit fields

[16] FIG-FORTH is a public domain Forth available through the Forth Interest Group.

Figure 3.10: Intel 8080 Registers and Instruction Formats



which specify a register operand. Each register has its own 3-bit pattern. Register pairs require only the first two bits of this pattern. Instructions may use a 3-bit pattern that specifies the M or PSW operand in place of a regular register. Likewise, a register pair pattern might specify SP (rather than BC, DE or HL).

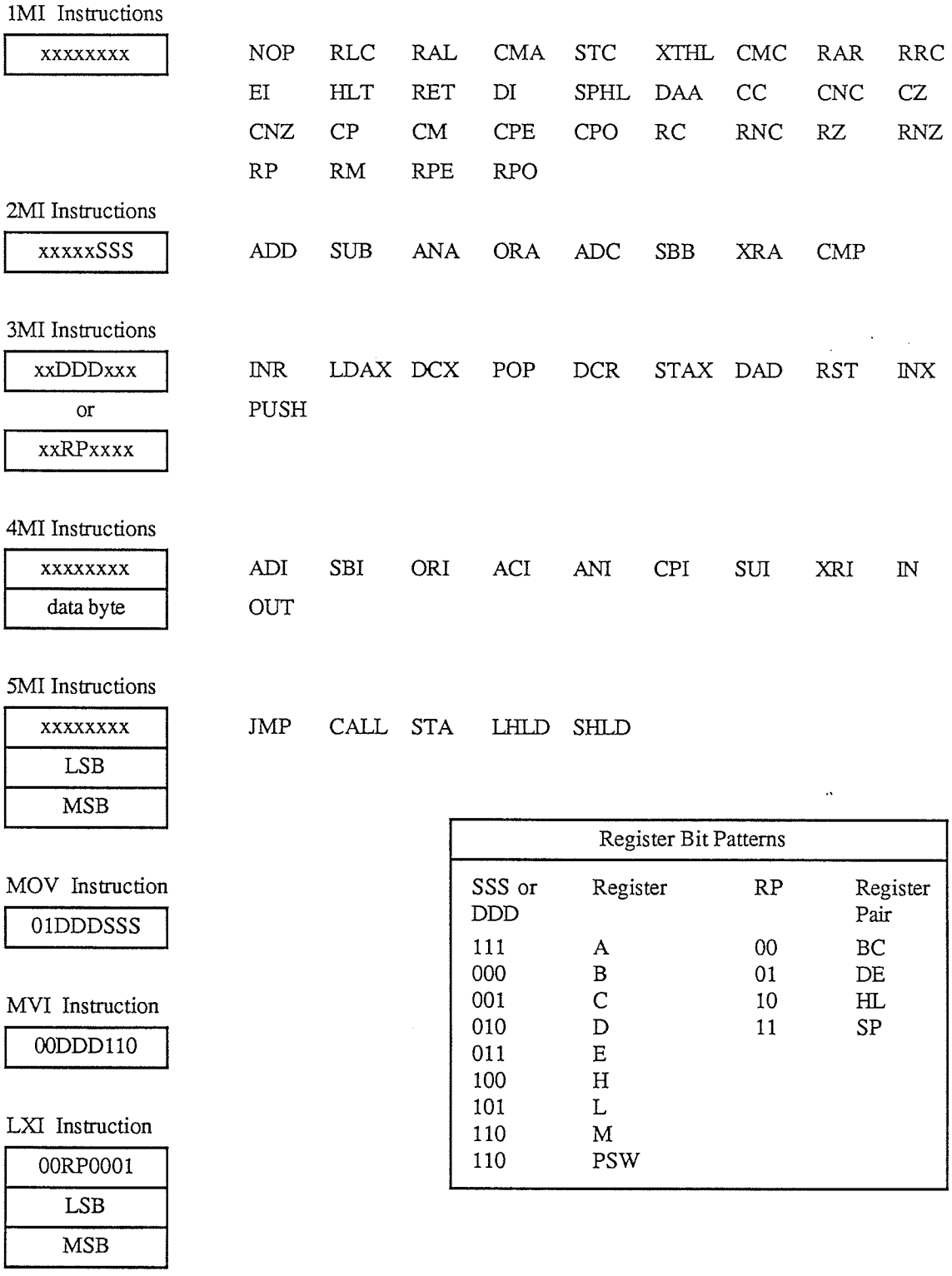
Assembler word definitions should exploit similarities between instruction formats. For instance, all arithmetic and logic instructions take place between the accumulator and a given register (or memory). Each of these instructions can be formed by adding a register bit pattern to a fixed opcode specific to the instruction. Registers are defined as CONSTANTs which push their bit patterns onto the stack. An instruction can then add its opcode with the top stack entry and enclose the result into the dictionary. This generates the appropriate machine code for 2MI-type instructions (see below). 3MI-type instructions are similar except that the register bit pattern must be shifted three bits to the left.

```
: 8*  ( n1 -- n2 )
   DUP + DUP + DUP + ;
```

can be defined to perform the shift operation. Registers are defined as:

```
( begin definition of ASSEMBLER vocabulary )
HEX VOCABULARY ASSEMBLER IMMEDIATE
```

Figure 3.11: Intel 8080 Instruction Classes



Register Bit Patterns			
SSS or DDD	Register	RP	Register Pair
111	A	00	BC
000	B	01	DE
001	C	10	HL
010	D	11	SP
011	E		
100	H		
101	L		
110	M		
110	PSW		

```
( define ASSEMBLER word set )
ASSEMBLER DEFINITIONS

7 CONSTANT A        0 CONSTANT B        1 CONSTANT C
2 CONSTANT D        3 CONSTANT E        4 CONSTANT H
5 CONSTANT L        0 CONSTANT BC       2 CONSTANT DE
4 CONSTANT HL       6 CONSTANT M        6 CONSTANT SP
6 CONSTANT PSW
```

The simplest instructions consist of a single fixed opcode. (Note that the following numbers are given in hexadecimal.) 1MI defines instructions which enclose their opcode into the dictionary.

```
: 1MI    ( opcode -- )
   CREATE C, DOES > C@ C, ;
```

The sequence C9 1MI RET creates the word RET which, when executed, encloses the byte C9 into the dictionary.

The next set of instructions are formed by adding the top stack entry (a register bit pattern) to an opcode and enclosing the result.

```
: 2MI    ( opcode -- )
   CREATE C, DOES> C@ + C, ;
```

The sequence 80 2MI ADD defines the ADD instruction. E ADD would push E (3) and call ADD which would calculate 80 + 3 and enclose the byte 83. This corresponds to the instruction "ADD E" (add register E to the accumulator).

3MI is similar to 2MI but shifts the register bit pattern left by three bits before adding it to the opcode.

```
: 3MI    ( opcode -- )
   CREAT C, DOES> C@ SWAP 8* + C, ;
```

The POP instruction is defined as C1 3MI POP. The sequence DE POP corresponds to the instruction "POP DE" and encloses the byte D1 (C1 + 2*8). This instruction will pop a 16-bit value from the stack into the DE register pair.

Instructions that require a single byte of immediate data are defined using 4MI. 4MI is similar to 1MI but encloses a fixed opcode followed by a data byte taken from the top stack entry.

```
: 4MI    ( opcode -- )
   CREATE C, DOES> C@ C, C, ;
```

The CPI instruction is defined as FE 4MI CPI. The instruction 60 CPI ("CPI 60") compares the contents of the accumulator with the value 60. This instruction encloses the two bytes FE and 60.

5MI is nearly the same as 4MI but us used to define instructions that require two bytes of immediate data.

```
: 5MI    ( opcode -- )
   CREATE C, DOES> C@ C, , ;
```

The sequence C3 5MI JMP defines the JMP instruction. 21E4 JMP would enclose the bytes C3, E4 and 21 in that order. This represents the "JMP $21E4" instruction which causes the processor to continue executing at address 21E4.

Most of the 8080 assembler instructions can now be defined:

```
00 1MI NOP      76 1MI HLT      F3 1MI DI       FB 1MI EI
07 1MI RLC      0F 1MI RRC      17 1MI RAL      1F 1MI RAR
E9 1MI PCHL     F9 1MI SPHL     E3 1MI XTHL     EB 1MI XCHG
27 1MI DAA      2F 1MI CMA      37 1MI STC      3F 1MI CMC
C0 1MI RNZ      C8 1MI RZ       D0 1MI RNC      D8 1MI RC
E0 1MI RPO      E8 1MI RPE      F0 1MI RP       F8 1MI RM
C9 1MI RET

80 2MI ADD      88 2MI ADC      90 2MI SUB      98 2MI SBB
A0 2MI ANA      A8 2MI XRA      B0 2MI ORA      B8 2MI CMP

09 3MI DAD      C1 3MI POP      C5 3MI PUSH     02 3MI STAX
0A 3MI LDAX     04 3MI INR      05 3MI DCR      03 3MI INX
0B 3MI DCX      C7 3MI RST

D3 4MI OUT      DB 4MI IN
C6 4MI ADI      CE 4MI ACI      D6 4MI SUI      DE 4MI SBI
E6 4MI ANI      EE 4MI XRI      F6 4MI ORI      FE 4MI CPI

22 5MI SHLD     2A 5MI LHLD     32 5MI STA      3A 5MI LDA
C4 5MI CNZ      CC 5MI CZ       D4 5MI CNC      DC 5MI CC
E4 5MI CPO      EC 5MI CPE      F4 5MI CP       FC 5MI CM
CD 5MI CALL     C3 5MI JMP
```

The remaining instructions: MOV, MVI and LXI do not fit into any of the above instruction classes. These misfits can be defined as:

```
: MOV   ( src-reg dest-reg -- )
   8* 40 + + C, ;

: MVI   ( byte -- )
   8* 6 + C, C, ;

: LXI   ( addr -- )
   8* 1+ C, , ;
```

As an example, the instruction D A MOV ("MOV A,D") would push D (2) and A (7) onto the stack. MOV would then calculate and enclose the byte 7A (7*8 + 40 + 2). This instruction will assign the contents of the D register to the A register (accumulator).

The assembler words CODE and END-CODE are defined as discussed in section 3.4. The word NEXT may be defined as:

```
( address of NEXT in address interpeter )
0104 CONSTANT #NEXT

: NEXT #NEXT JMP ;
```

The conditional jump instructions were not defined since they can be compiled from higher level words. These words are similar to the Forth control structures discussed in section 3.8. The same word names (IF, ELSE, THEN, etc) can be used without conflict within the ASSEMBLER vocabulary.

The word ?PAIRS is used to perform syntax checking. ?PAIRS expects two matching values to be present on the stack. This is used to verify matching IF-THEN, ELSE-THEN, BEGIN-UNTIL and other paired control words.

```
: ?PAIRS <> IF ABORT" control structure error" THEN ;
```

The words IF, UNTIL and WHILE expect a conditional jump opcode to be present on the stack. This opcode defines the test and branch condition. Each conditional opcode is defined as a CONSTANT. For example, C2 CONSTANT 0= can be used in the form 0= IF ... THEN. 0= specifies the JNZ instruction which will jump around the IF and continue execution following THEN if the PSW indicates a non-zero result. The word NOT adds in a bit which reverses the test condition. The sequence 0= NOT would place the JZ (jump if zero) opcode onto the stack.

Conditional opcodes and control structures are defined as:

```
C2 CONSTANT 0=       D2 CONSTANT CS    ( carry set )
F2 CONSTANT 0<       E2 CONSTANT PE    ( parity even )

: NOT    ( jmp-op -- jmpn-op )
   + ;

: IF     ( -- )
         ( jmp-op -- sys compiling )
   C, HERE 0 , 2 ;

: THEN   ( -- )
         ( sys -- compiling )
   2 ?PAIRS HERE SWAP ! ;

: ELSE   ( -- )
         ( sys1 -- sys2 compiling )
   2 ?PAIRS C3 IF ROT SWAP THEN 2 ;

: BEGIN   ( -- )
          ( -- sys compiling )
   HERE 1 ;

: UNTIL   ( -- )
          ( sys jmp-op -- compiling )
   SWAP 1 ?PAIRS C, , ;

: WHILE   ( -- )
          ( jmp-op -- sys compiling )
   IF 2+ ;

: REPEAT   ( -- )
           ( sys -- compiling )
   >R >R 1 ?PAIRS C3 C, , R> R> 2 - THEN ;
```

An example using control structures is shown below. UPPER accepts a pointer to a string and the string length. UPPER converts all lowercase characters to uppercase by subtracting 20 ("a" - "A") from characters greater than or equal to "a". Note the use of ASCII to generate the immediate data. This example assumes the 8080 stack is used as the data stack.

```
( sample definition using 8080 assembler )
CODE UPPER   ( addr length -- )
   D POP                    ( get string length )
   H POP                    ( get string address )
   BEGIN
   D A MOV                  ( continue while length )
   E ORA                    ( is not zero )
   0= NOT
   WHILE
      M A MOV               ( get character at HL )
      ASCII a CPI           ( check if character )
      CS NOT                ( is >= "a" )
      IF
         ASCII a            ( convert lower to uppercase )
         ASCII A - SUI
         A M MOV            ( store result into string )
      THEN
      D DCX                 ( decrement length )
      H INX                 ( increment character pointer )
   REPEAT
   NEXT                     ( goto address interpreter )
   END-CODE
```

The 8080 code generated for UPPER corresponds to the 8080 assembly language code:

```
UPPER:
        DW    *+2           ; primitive CAW
        POP   D             ; pop arguments
        POP   H
LOOP:
        MOV   D,A           ; check string length
        ORA   E
        JZ    DONE          ; exit if length is zero
        MOV   A,M           ; otherwise translate any
        CPI   A,'a          ; lower to uppercase chars
        JCS   AHEAD
        SUI   'a - 'A
        MOV   M,A
AHEAD:
        DCX   D             ; decrement string length
        INX   H             ; increment character pointer
        JMP   LOOP          ; continue to end of string
DONE:
        JMP   NEXT          ; goto address interpreter
```

Control structures normally eliminate the need for line labels. If necessary, line labels can be defined using:

```
: DECLARE_LABEL   ( n -- )
   0 CONSTANT ;

: SET_LABEL   ( -- )
   ' 2+ HERE SWAP ! ;      ( ' <name> returns the CFA )
```

Labels must be defined in advance (any word must be defined before being used). A label is a CONSTANT which pushes an address onto the stack. SET_LABEL locates a word and stores the value of HERE into the word's parameter field (assuming the word is a CONSTANT in a 16-bit ITC system). This equates the CONSTANT (label) to the current dictionary pointer. For example:

```
DEFINE_LABEL LOOP          ( define a label )

( create an infinite loop word )
CODE FOREVER   ( -- )
   SET_LABEL LOOP          ( generates no code )
   LOOP JMP END-CODE       ( "LOOP:  JMP  LOOP" )
```

Definitions such as:

```
VARIABLE TEMP              ( reserve two bytes )
CREATE BUFFER 80 ALLOT     ( reserve an 80 byte buffer )
```

may be used to define data areas for use within an assembly language definition. The words TEMP and BUFFER each return a memory address suitable for assembly. As an example, a pointer to the buffer would be initialised by saying BUFFER HL LXI (load register pair HL with the address of BUFFER).

Chapter IV

VIRTUAL MACHINE

A high-level language (HLL) can be either compiled or interpreted. FORTRAN source code is usually compiled to machine code whereas APL is interpreted. Each approach has its advantages: machine code executes faster; interpreted code offers the possibility of improved debugging aids. Many other tradeoffs are involved in the decision to translate or interpret a language.

An intermediate language implementation offers a compromise between compilation to machine code and HLL interpretation. The HLL is translated to an intermediate language (IL) which is independent of the target machine. This allows a single compiler to generate code suitable for a variety of machines. A good example is UCSD Pascal, which translates Pascal source code to an IL called p-code. Forth performs a similar translation to an IL known as threaded code (TC). These ILs are interpreted by a "virtual machine". The virtual machine "executes" the p-code or TC instructions by emulating them on the target machine. Thus the IL remains machine independent.

ILs can again be either translated or interpreted. An interpreted IL will naturally be slower. Ideally the pro-

grammer would like to have the option of interpreting or compiling the IL. (This option is also desired at the source level). Program development could be done in an interactive environment - interpreting the IL (or source) and providing powerful debugging tools. The IL (or source) could later be compiled to achieve faster execution times.

Any IL must make compromises between various degrees of machine independence, execution speed and size of code generated. Forth has chosen TC for its ease of interpretation, compactness and reasonably fast execution. TC represents every program as a sequence of subroutine calls. The lowest level subroutines are used to implement primitives which form the basis of Forth. Compactness is achieved by using subroutines versus inline code; this increases interpretation overhead. TC is, however, able to maintain its speed by being very close to the machine level.

Forth has traditionally been implemented using indirect threaded code (ITC) as described in the previous chapter. Forth's address interpreter (also known as the inner interpreter) acts as the virtual machine which executes TC. Forth's primitives form the instruction set of the virtual machine - each primitive containing machine code which emulates an instruction.

This chapter briefly describes the various types of TC and the implementation of their address interpreters. Al-

ternatives to TC implementations are also discussed. These include macro substitution and compilation to machine code. Both alternatives can be implemented in Forth or some other language.

Since all forms of TC are logically equivalent, it is possible to substitute one for the other. Thus a single compiler could allow the choice of TC based on requirements such as speed or code size. Forth may also be used to redefine itself for a different type of TC.

4.1 SUBROUTINE THREADED CODE

Each form of TC must uniquely specify a subroutine call plus provide a subroutine call/return mechanism and a way to advance to the next TC instruction. Subroutine threaded code (STC) uses the machine instructions JSR (jump to subroutine) and RTS (return from subroutine). These instructions are available (in one form or another) on most machines. Thus STC is directly supported by hardware. There is no need for an address interpreter. The ITC address interpreter routines DOCOLON, DOSEMI and NEXT (refer back to section 2.6) are replaced by JSR, RTS and JSR/RTS respectively. Note that the JSR/RTS pair has the effect of advancing the program counter to the next STC instruction.

An STC implementation of Forth would require the text interpreter to enclose a JSR opcode before each word address

being compiled. Similar changes would be made to CREATE and other defining or compiling words. For instance, semi-colon would terminate a colon definition by enclosing an RTS op-

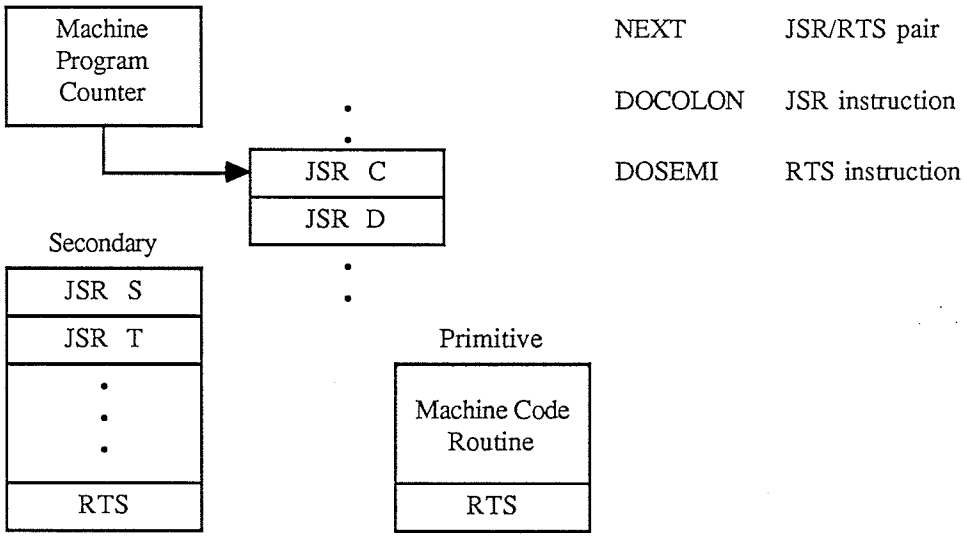


Figure 4.1: Subroutine Threaded Code

code. The format of STC bodies is shown in figure 4.1.

The main advantage of STC is that it is fast. No interpretation is required. STC can also be intermixed with in-line machine code. This is not feasible with other forms of TC. One final advantage of STC is its efficient use of processor resources. It does not require additional machine registers to be dedicated for use by an address interpreter.

The disadvantage is that STC requires each word address to be preceded by a JSR opcode. Code size can be as much as

double that of ITC. STC will also produce machine and position dependent code. Position independent code can be generated by using a relative JSR instruction (if one is available).

Unlike most other types of TC, which require an address interpreter, STC is directly executed. This precludes the possible introduction of debugging aids within the address interpreter. High level debug/trace capabilities cannot be easily added.

4.2 DIRECT THREADED CODE

In direct threaded code (DTC), the JSR opcode is removed so that DTC consists only of word addresses (see fig 4.2). As with all forms of TC (other than STC) an address interpreter is required.

DTC and ITC are very similar - the only difference being that DTC bodies do not begin with a code address word (CAW). Recall that the CAW was used to point to a prologue procedure (this would point at DOCOLON in the case of a colon definition). In DTC, the prologue procedure is included at the start of each DTC body.

The DTC address interpreter is simpler than that for ITC. The address interpreter fetches the next word address and jumps to machine code at this address. This works fine for primitives - they contain machine code followed by a jump back to NEXT.

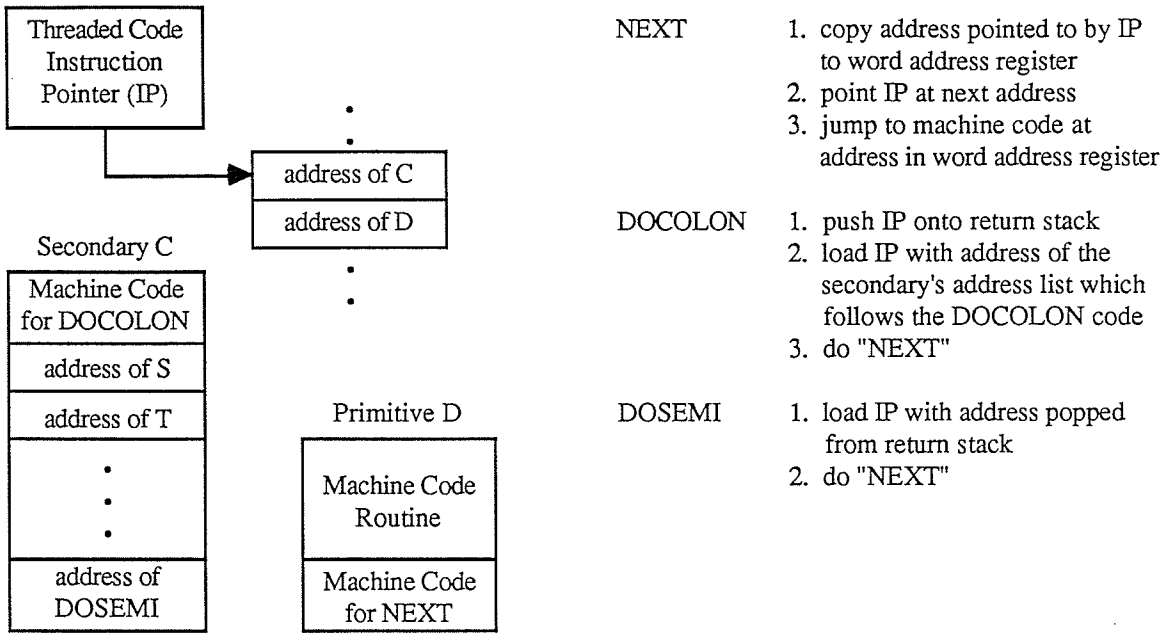


Figure 4.2: Direct Threaded Code

Secondaries begin with a short machine code procedure. This prologue procedure performs the function of DOCOLON. The TC following the prologue code will be interpreted as usual. The last word address is a pointer to DOSEMI which functions exactly as its ITC version.

DTC has the advantage of being faster than ITC. This is due to a simpler address interpreter which jumps directly to machine code. Primitives use slightly less memory since the CAW, associated with ITC, is not present.

On the other hand, DTC duplicates the prologue code once for each secondary (rather than storing a pointer to common code). This prologue code is short (4-20 bytes) but can occupy a large amount of memory when duplicated thousands of times. DTC is still machine and position dependent.

4.3 INDIRECT THREADED CODE

The use of indirect threaded code (ITC) was discussed in detail in section 2.6. Figure 4.3 reviews the structure of

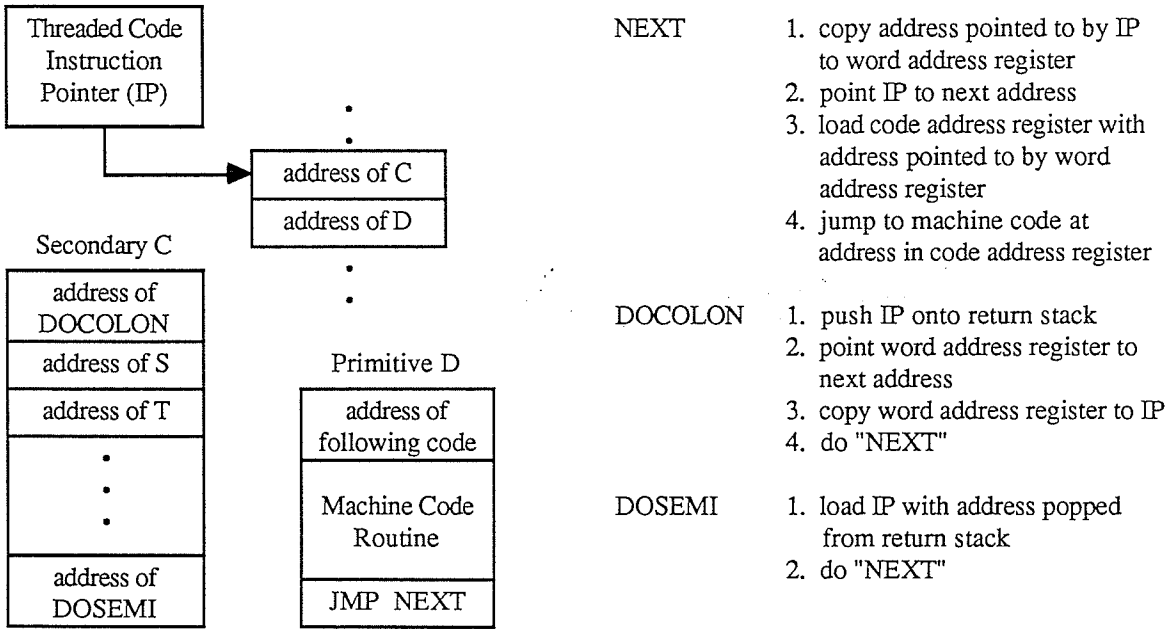


Figure 4.3: Indirect Threaded Code

ITC for comparison with other types of TC.

ITC builds upon DTC by adding a CAW which points at the prologue code. This added step of indirection separates machine code from ITC and allows ITC to become machine independent. The CAW also reduces memory requirements by eliminating the need to duplicate prologue code.

The disadvantage of ITC is its slower execution speed due to the added step of indirection in the address interpreter. ITC remains position dependent.

4.4 RETURN THREADED CODE

Return threaded code (RTC) is an interesting combination of STC and DTC [Bur84]. Like STC, RTC is directly executed and requires no address interpreter. RTC uses STC primitives and DTC secondaries as shown in figure 4.4. This trick requires using the processor's return stack pointer (SP) as the TC instruction pointer. The data and return

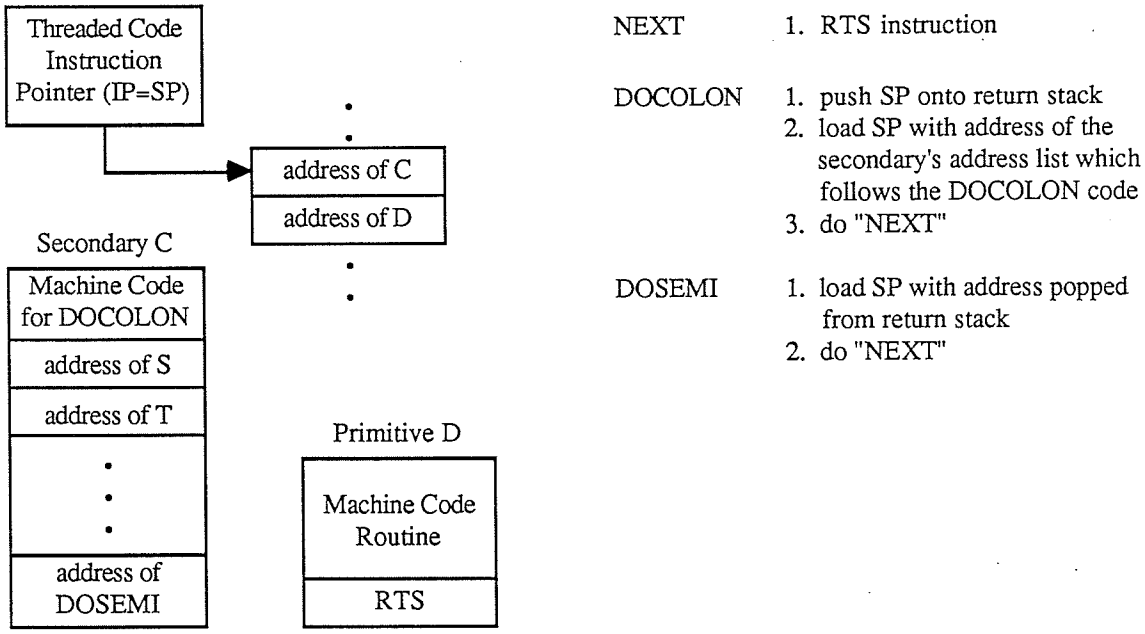


Figure 4.4: Return Threaded Code

stacks must be implemented using registers other than SP.

In RTC, a single RTS instruction performs the NEXT function (incrementing SP and jumping to the next word address). The TC acts as a pre-defined set of return stack entries. Successive RTS instructions "eat" their way through the TC,

executing each word in the address list. There is very little overhead involved in executing a list of primitive word addresses. Secondaries require entry/exit code as in DTC.

RTC can execute faster than STC if the ratio of primitives executed to secondaries executed exceeds a certain value. This ratio depends on primitive and secondary overhead which varies from processor to processor. The following table shows the ratio required for RTC to outperform STC under several conditions.

Processor	Address Length	Ratio
MC68000	16 bits 32 bits	11:9 3:2
MC6809	16 bits	17:7

RTC and STC share much the same advantages and disadvantages. RTC, however, has the added DTC advantage of compact code and the disadvantage of not being able to directly include inline machine code. RTC may not produce compact code on some machines. For example, the MC68000 always requires 32-bit return addresses. A 64K byte RTC system would generate double the code required by DTC (16-bit addresses). This problem does not occur with most other processors.

4.5 TOKEN THREADED CODE

Token threaded code (TTC) can be classified as either direct TTC (DTTC) or indirect TTC (ITTC). These forms are analogous to DTC and ITC. The difference is that TTC replaces each word address with a token which is used as an index into a table of pointers (see figures 4.5 and 4.6). The address interpreter performs a table lookup to determine the word address.

TTC has the advantage of being very compact. It typically requires 50% less than DTC or ITC. Each token is a table index rather than a full address. For example, a 32-bit machine could use a 16-bit token. The use of an 8-bit token is sufficient provided there will be no more than 256 word definitions.

The disadvantage of TTC is its slower execution speed. The address corresponding to each token must be located and fetched from a table. TTC requires this separate table to be allocated for storage of pointers. However, the memory used by the table is normally small in comparison to the amount saved by using TTC.

DTTC is faster than ITTC but slower than ITC. DTTC is position independent only if its prologue code is position independent. Since DTTC contains inline code, it is also machine dependent.

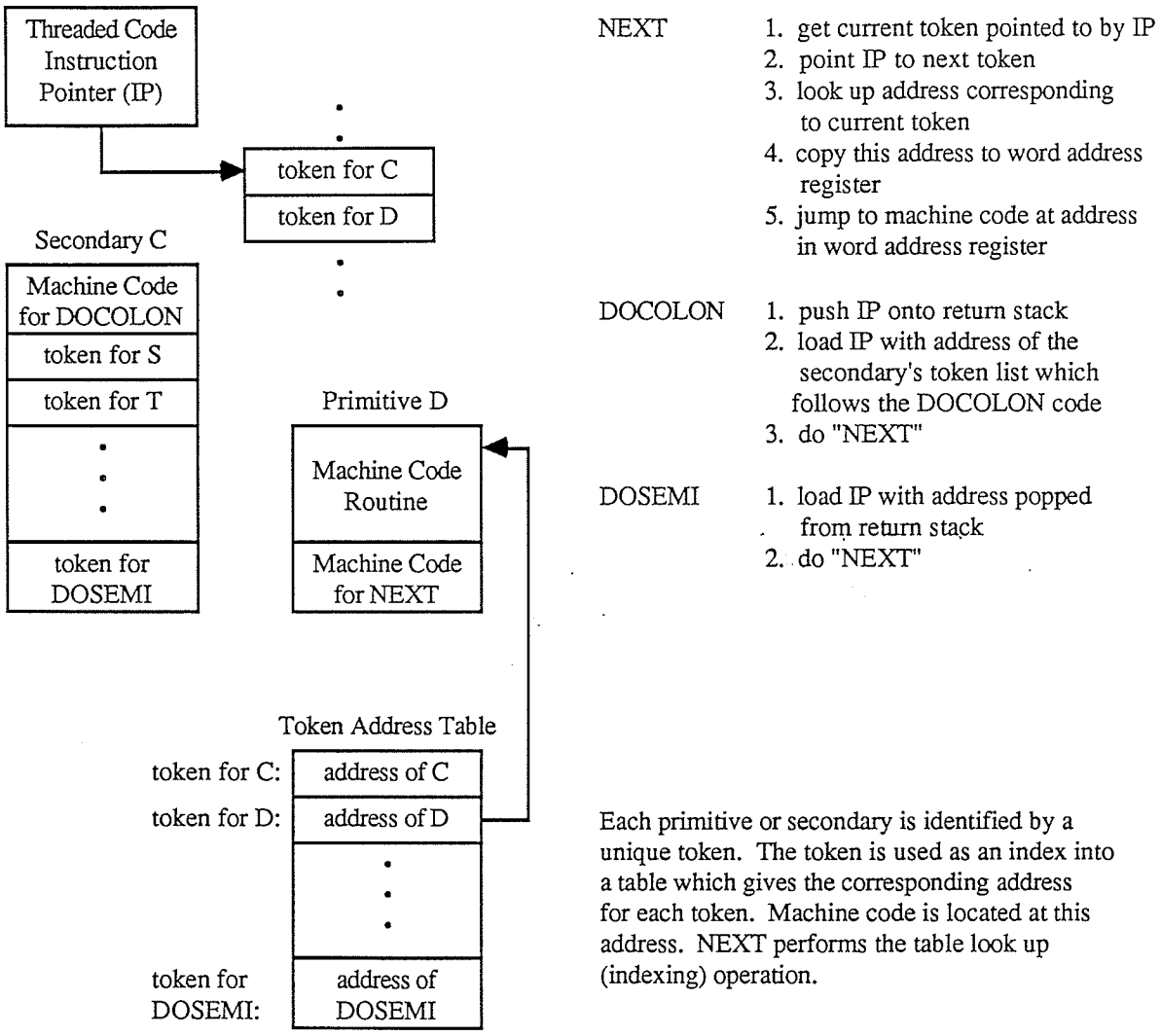


Figure 4.5: Direct Token Threaded Code

ITTC is slower than DTTC but is both machine and position independent because it is composed entirely of tokens.

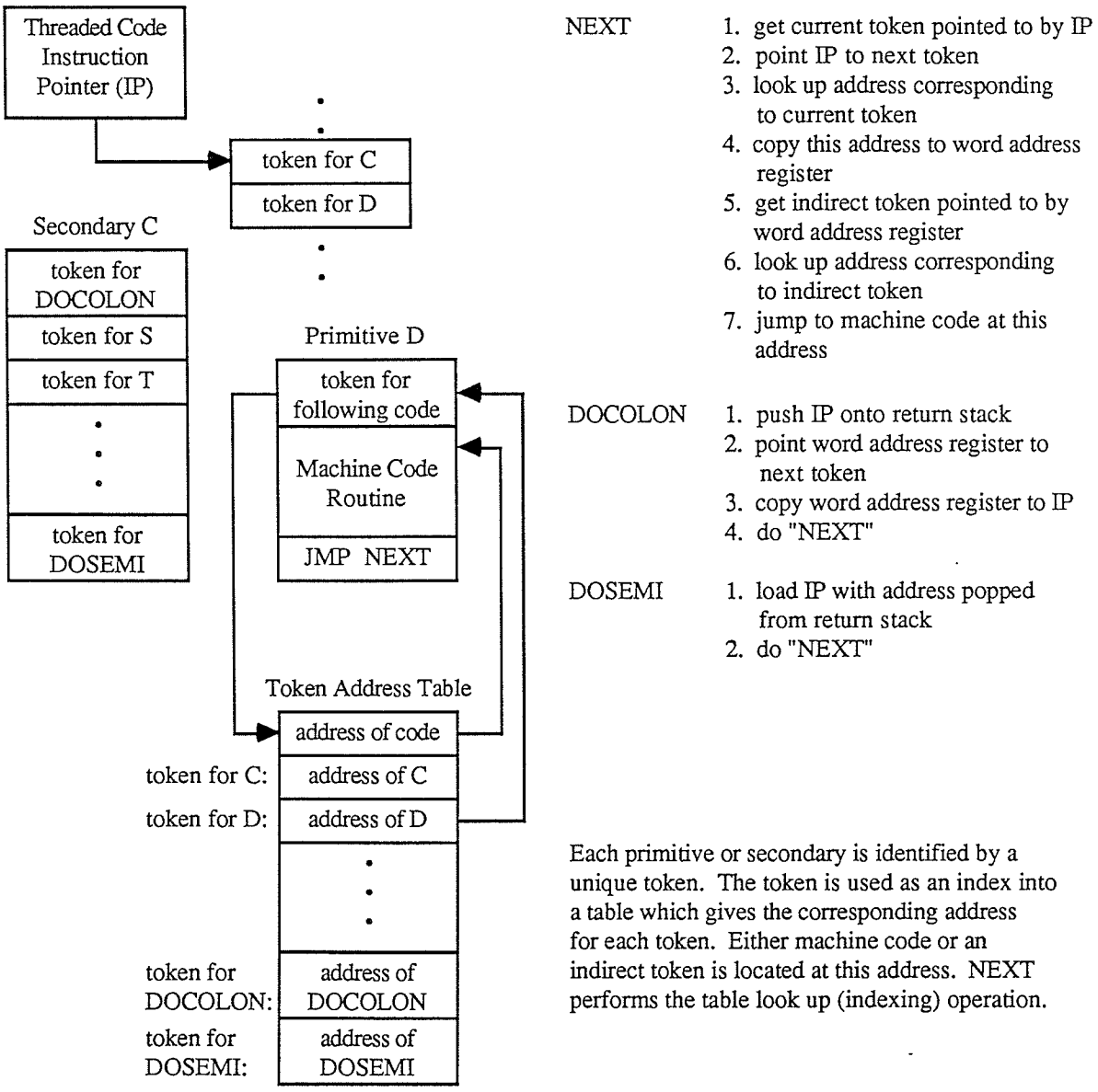


Figure 4.6: Indirect Token Threaded Code

4.5.1 Table Lookup

A TTC address interpreter spends most of its time performing table lookups. For this reason, its important that the table lookup (or vectored jump) operation be as fast as

possible. Some processors requires this operation to be performed in several steps:

1. shift the index (token) left to obtain an offset into the table,
2. add the offset to the base address of the table (calculate the address of the table entry),
3. fetch the pointer at this address and
4. jump to the machine code pointed to by this pointer.

Other processors, such as the National Semiconductor NS32016, can perform this operation with a single instruction (a memory scaled indexed indirect jump). In any case, the best solution is the fastest one and is highly dependent upon the target machine (especially upon its addressing capabilities).

In general, the lookup table is placed anywhere in memory and each token provides an integer index into the table. This requires steps 1) and 2) to be performed. The table lookup can be improved by eliminating either or both of these steps.

If the table is placed beginning at address zero (this may be at the start of any segment or at logical address zero in segmented or memory mapped systems respectively), then the base address of the table (zero) no longer needs to be added to the offset and step 2) is eliminated.

Similarly, if each token is defined as an offset rather than an index, step 1) is eliminated. This, of course, will reduce the number of available tokens and thus allow fewer word definitions. Using 16-bit tokens and 32-bit addresses, the number of available tokens is reduced from 65,536 to 16,384 (for large tokens this reduction is acceptable).

4.6 MODIFIED THREADED CODE

Additional types of TC can be defined by combining or modifying the basic types: STC, DTC, ITC and TTC. Such combinations can provide a more application or machine specific form in which to implement the virtual machine. In general, however, most permutations offer few benefits over the traditional forms of TC. Any advantages are usually outweighted by a large increase in execution time due to the increased complexity of the address interpreter. A few examples are discussed in the following paragraphs.

Indirect subroutine threaded code (ISTC) may be defined by modifying STC. The temptation here is to reduce code size by eliminating the JSR opcodes required by STC. An address interpreter is added to fetch the next subroutine address and perform an indirect JSR to that address. This closely resembles the DTC address interpreter. Both must fetch the next address and either JMP (DTC) or JSR (ISTC). ISTC and DTC share the advantage of compact code, however, ISTC is slower since it requires a JSR or RTS for each DTC JMP or JMP NEXT. DTC remains the better choice.

TTC can be modified to suit 8-bit systems with limited memory. In this case it is best to use 8-bit tokens for compact code. The limitations of 8-bit tokens are solved by allowing a certain range of tokens to be followed by an additional 8-bit token extension. For example, the most significant bit of each token could be used as an extension flag. Primitives would be referenced by one byte (a token value from zero to 127). Up to 32,768 other words could be represented by a token in the range 128 to 255 followed by an extension byte. The added complexity of the address interpreter is offset by the fact that most code is composed of single byte tokens. This approach provides relocatable, reasonably fast TC for 8-bit systems. A similar approach is discussed in [Far83].

Other forms, such as status threaded code [Bue84], may also be useful in improving the Forth development environment. In these cases, the TC may contain additional control information. The address interpreter is further complicated in exchange for better debugging facilities or language features. In extreme instances the virtual machine can be used to alter the functionality of Forth itself. This can result in data driven or object oriented systems with a higher level of interpretation and a correspondingly large overhead.

4.7 HIGH-LEVEL LANGUAGE

The virtual machine and Forth itself can be implemented in Forth or some other high-level language. Two components must be implemented. First, a memory management scheme is needed to provide storage and access of the dictionary (containing virtual machine instructions, data, etc) and stacks. Second, an interpreter must be written to execute the virtual machine instructions (Forth primitives).

The choice of memory allocation and use is dependent on the high level language. Typically a large array is defined to encompass available memory. Alternatively, memory may be dynamically allocated as required. In either case, a "chunk" of memory is allocated towards a data structure or structures used to represent the dictionary and stacks.

The virtual machine is usually best implemented as a case statement. This approach closely resembles token threaded code. Each primitive is assigned a virtual machine opcode (token value). The virtual machine fetches the next opcode from memory and executes the case statement based on the opcode. Each case contains code which performs a primitive's function. The case may simply invoke a procedure which emulates the primitive, however, this incurs the overhead of a subroutine call. It is advisable not to use separate procedures for each primitive [Bla83].

The basic structure of the virtual machine is shown by the following Pascal code:

```
Program Forth (Input, Output);
Const
   StackSize  = 1000;
   MemorySize = 10000;

   { virtual machine opcodes }
   DOCOLON    = 1;
   DOSEMI     = 2;
   ADD        = 3;
   .
   .
   LastOp     = N; { last opcode used }
Type
   MemAddress = 1..MemorySize
   Memory     = Array [MemAddress] of Integer;
   Stack      = Record
                   StackTop  : Integer;
                   StackData : Array [1..StackSize] of Integer;
                End;
Var
   Mem            : Memory;
   DataStack      : Stack;
   ReturnStack    : Stack;
   .
   .
Procedure Push(OntoStack : Stack; Data : Integer);
Begin
   StackTop := StackTop - 1;
   OntoStack.StackData [StackTop] := Data;
End;

Function Pop(Var FromStack : Stack) : Integer;
Begin
   TopEntry := FromStack.StackTop;
   FromStack.StackTop := FromStack.StackTop + 1;
   Pop := FromStack.Stackdata [TopEntry];
End;

Function Empty(FromStack : Stack): Boolean;
Begin
   Empty := FromStack.StackTop > StackSize;
End;
   .
   .
```

```
Procedure Execute(PC : MemAddress)
{ execute a word definition pointed to by PC }
{ assuming return stack is empty on entry    }
Var
   OpCode : 1..LastOp;
   Temp   : Integer;
Begin
   { execute the word definition }
   Repeat

      { fetch next instruction }
      OpCode := Mem [PC];
      PC := PC + 1;

      { execute it }
      Case OpCode Of
      DOCOLON:
         Begin
         Push( ReturnStack, PC );
         PC := Mem [PC];
         End;
      DOSEMI:
         PC := Pop( ReturnStack );
      ADD:
         Begin;
         Temp := Pop( DataStack );
         Temp := Temp + Pop( DataStack );
         Push( DataStack, Temp );
         End;
      .
      .
      End;

   { continue nested word execution if return stack }
   { contains any return word addresses             }
   Until Empty( ReturnStack );
End;
   .
   .
{ mainline - text interpreter }
Begin
   { set stacks to empty }
   { print startup message }
   .
   .
End.
```

Note that the text interpreter is best written in the HLL (otherwise it must be coded as a series of virtual machine opcodes). The text interpreter performs the usual diction-

ary search, compile and execute procedures. Less restrictive languages offer a better means of implementation. For example, the C language can implement stack operators in a single instruction (or macro). PUSH and POP macros may be defined as:

```
#define PUSH(stack,data) *--stack = data
#define POP(stack) *stack++
```

These are used as in "x = POP(stack);" or "PUSH(stack,x);". Memory can be more effectively used by allowing byte and integer data to be intermixed. Opcodes may then be stored in a single byte (if there are fewer than 256 primitives).

Implementing the virtual machine in a HLL has several advantages. It makes the program easier to read, write and understand. Also, the implementation is far more portable (as opposed to an assembly language version).

The disadvantages of this approach are twofold. First, the virtual machine will be slower - this is inherent in the translation from HLL to machine code. Second, Forth's assembly language facility generally cannot be implemented. The user only has access to existing "built-in" primitives. Primitives can be added only by modifying and re-compiling the HLL source code for the "execute" procedure.

4.8 MACRO EXPANSION

Forth has been likened to a macro processor. In a sense this is true. Forth words are "expanded" at execution time. TC represents a list of macro instructions.

It is possible to create a defining word which allows the definition of true macros. Macro words could be defined as immediate words which compile a copy of themselves. Each macro duplicates its code rather than allowing the text interpreter to compile the word address. Macro expansion occurs at compile-time rather than execution-time.

The most common approach to implementing macros has been to add an extra macro header field to each dictionary entry [Gre84a]. This field is similar to the word name length field but contains the code body length and a flag indicating if the word is a macro. The text interpreter is modified to check the macro flag during compilation (execution remains the same). If the word is a macro then the appropriate number of bytes are copied from its code body to the dictionary.

Another approach has been to create a high-level defining word MACRO [Yng85]. MACRO acts like ':' but reserves a length field at the start of the code body. END-MACRO later sets this field to the code body length. The DOES> portion of MACRO either copies or executes the code body depending on STATE.

Macros have been incorporated into several commercially available Forth systems. A macro is defined in the form MACRO <name> ... END-MACRO. Macros allow a subroutine call to a secondary to be replaced by the TC of that secondary. This improves execution speed marginally but rapidly increases code size (especially if macros are nested). Control structures can be used within macros as long as they generate position-independent code (relative branches must be used).

The benefits of macros are best realized with the use of subroutine threaded code (STC). Recall that STC is directly executed and therefore allows the inclusion of inline machine code. Macros can be used to generate inline code for both primitives and secondaries. STC and macros may be combined at any level and operate in synergy.

Other types of TC cannot be effectively combined with inline machine code. Each macro must contain entry/exit code which bypasses the address interpreter. A single macro begins with address or token data that causes the address interpreter to indirectly jump to the inline code that follows. The macro must end with code which adjusts the TC instruction pointer and returns to the address interpreter. Overhead for a single macro actually results in slower execution. Improved speed is only possible if several words can be replaced by a set of adjacent macros. Overhead is then reduced by placing entry code before the first macro and exit code after the last macro.

The remainder of this chapter discusses optimization techniques based on the use of macros and STC. Examples are given using the Motorola 68000 assembly language and execution times. A 64K byte address space is assumed. The data stack pointer (ds) is kept in address register A6. The return stack pointer (rs) is kept in address register A7 (sp) which is used by the 68000 JSR (jump to subroutine) and RTS (return from subroutine) instructions. JSR pushes a full 32-bit return address which is later popped by RTS.

Optimization techniques often depend upon the processor being used. In general, registers should be used in place of memory for high-access variables. It may be worth-while to keep the top stack entry in a register [Bur84]. Short forms of instructions should be used whenever possible (eg. a short relative branch versus a long jump).

Macros can be used to replace both secondaries and primitives. The best candidates for replacement are common primitives such as SWAP, DROP, DUP, etc. These primitives are very short machine code procedures. Often, the time required to JSR and RTS is greater or very close to the execution time of the primitive function itself. In this case, the primitive should be replaced with a macro which generates inline code to perform the primitive function. This produces much faster code.

A comparison of STC and macro implementations of the word DROP is shown below. The table indicates total execution time in machine cycles. Total size of code generated is given in bytes. STC size does not include code required by subroutines; it only includes code that is actually compiled (ie. the subroutine call). Values preceded by a '+' are added to obtain the total execution time and code size. Actual execution time can be calculated by dividing cycle time by clock speed. For example, an 8 Mhz 68000 executes the STC version of DROP in 42 / 8 = 5.25 microseconds.

Description	Machine Code		Cycle Time	Bytes
STC for DROP	jsr.w	DROP	+18	+4
DROP primitive	addq.w	#2,ds	+8	2
	rts		+16	2
			Total /42	/4
macro DROP	addq.w	#2,ds	8	2

The STC for DROP requires 4 bytes (for the subroutine call) and takes 42 cycles to execute. A macro version of DROP surprisingly requires only 2 bytes and executes in 8 cycles. This is clearly an improvement in both code size and execution time. Similar results can be obtained for other primitives.

It is important to note that primitives are the most frequently used words [Tin85]; they represent the instruction set of the virtual machine. Macros provide another way of

implementing this virtual instruction set. Macros can also produce a faster set of primitives and thus dramatically increase overall execution speed.

The usefulness of macros in STC is directly related to the instruction set, addressing modes and number of registers available on the target machine. In essence, a processor such as the Motorola 6800 will require more instructions to perform the same function as a 68000. Therefore, 6800 macros will be larger and the tradeoff between code size and efficiency will make macros less attractive. More complex primitives should not be written as 6800 macros; they generate too much code.

STC macros can be implemented as words which either duplicate their code or directly enclose machine code. In the first case, the word DROP might be defined as:

```
ASSEMBLER                    ( use ASSEMBLER vocabulary )
MACRO DROP    ( n -- )
   WORD 2 ds ADDQ
   END-MACRO
```

In the second case, DROP would be written as:

```
HEX ( use hexadecimal numbers )
: DROP    ( n -- )
   5496 ,                    ( "addq.w  #2,ds" )

   ; IMMEDIATE
```

The first approach (using a MACRO defining word) is clearly a better solution (the second approach provides a "quick and dirty alternative"). MACRO also allows secondaries to be defined as in:

```
: MACRO SQUARED    ( n1 -- n2 )
   DUP * END-MACRO
```

The advantage of macros is improved execution speed. The disadvantage may be increased code size.

4.9 COMPILATION

Compilation to machine code is the next logical step beyond STC and the use of macros. A conventional incremental compiler could be designed to translate Forth source code into machine code - thus eliminating the "virtual machine" and its inherent overhead. This approach allows a "smart" compiler to generate optimized code for the target machine.

Compilers must generally perform six major functions:

1. lexical analysis to determine which language symbols are present in the source code,
2. syntactic analysis to determine the structure of the source code representation,
3. semantic analysis to identify the actions (or meaning) of the source code,
4. storage allocation to determine storage requirements, to specify algorithms for execution-time addressing and to generate intermediate code based on these actions,
5. target code generation to translate intermediate code to assembler (or a similar machine-dependent representation) and

6. assembly to translate assembler code into machine code.

A more detailed discussion of compiler functions and implementation can be found in [Cal79].

The implementation of a Forth compiler should be very straightforward [Buv84]. Lexical analysis must only be concerned with the type of word being scanned. A word may be a number, a regular word or an immediate word. Syntactic and semantic analysis are largely eliminated due to Forth's postfix notation. Forth was designed to be easily compiled since its original compiler (the text interpreter) had to be small enough to remain in memory at all times. In short, a native code compiler for Forth must be simpler than that of most other languages. The task of implementing this compiler would be further lessened by the use of existing compiler related development tools and literature.

A single-pass Forth compiler is sufficient to allow code generation. Multiple passes could be used to perform code optimization. However, it is questionable as to how much optimization is possible within the standard definition of Forth. All words, including primitives, pass arguments and results via the stack. It is unlikely that the stack could be effectively replaced by registers. Not only are registers limited in number, but the compiler must be able to predict what the stack will look like at run-time in order

to compile code which accesses the proper registers. This becomes impossible once the stack is conditionally manipulated. Other forms of optimization can, in most cases, be done using a single-pass compiler.

Rather than create a conventional compiler, it is simpler and nearly as effective to use Forth's text interpreter. The text interpreter already functions as a single-pass compiler. Forth's hierarchy of word definitions would undoubtedly be compiled into a form similar to subroutine threaded code (STC) even if a conventional compiler were used. This must be the case since each word is functionally a subroutine call. Therefore a combination of STC and macros can form the basis for generating machine code. Macros already represent optimized primitives (by eliminating subroutine call overhead).

Compiler features can be incorporated into either the text interpreter or words themselves. The latter approach is in keeping with Forth's use of immediate words to perform compile-time actions. Several optimizations can be accommodated by defining "smart" immediate words. These words could view themselves in context with the surrounding input stream. For example, a common sequence of words such as SWAP DROP can be compiled into a single function. SWAP would look ahead and compile machine code to either SWAP or SWAP+DROP. If SWAP finds DROP it must, of course, adjust the input stream to continue following DROP. Alternatively,

SWAP could be compiled as usual and DROP could look back and overwrite SWAP with SWAP+DROP if DROP finds itself preceded by SWAP. The effect of this form of optimization (compression) is shown below:

Description	Machine Code		Cycle Time	Bytes
macro SWAP	mov.l	(ds),d0	+12	+2
	swap	d0	+4	+2
	mov.l	d0,(ds)	+12	+2
macro DROP	addq.l	#2,ds	+8	+2
			Total /36	/8
SWAP+DROP	mov.w	(ds)+,d0	+8	+2
	mov.w	d0,(ds)	+8	+2
			Total /16	/4

The normal sequence SWAP DROP requires 8 bytes and 36 cycles. The combined SWAP+DROP performs the same function in only 4 bytes and 16 cycles. Other common word sequences can be similarly compressed.

Number literals can be optimized by making a change to the definition of LITERAL. Recall that the text interpreter calls LITERAL to compile a number. Normally LITERAL compiles a call to the number literal handler and then encloses the number. At execution time the number literal handler must be called to push its value onto the stack. This process can be accelerated by modifiying LITERAL so that it compiles machine code which directly pushes a number onto the stack. For example, the number 1234 can be compiled as follows:

Description	Machine Code		Cycle Time	Bytes
DOLIT primitive	mov.l	(rs),a0	+12	2
pushes number	mov.w	(a0)+,d0	+8	2
to stack and	mov.w	d0,-(ds)	+8	2
adjusts return	mov.l	a0,(rs)	+12	2
address to skip	rts		+16	2
over number				
Normal code	jsr.w	DOLIT	+18	+4
compiled by	#1234		nil	+2
LITERAL			Total /74	/6
Optimized code	mov.w	#1234,-(ds)	12	4

A conventional LITERAL generates 6 bytes of code and takes 74 cycles to push its number onto the stack. The optimized LITERAL reduces this to 2 bytes and 12 cycles. Further optimization may be possible for smaller numbers.

LITERAL can also be improved by allowing it to compress certain sequences of the form <number><primitive>. Thus combinations such as 5 + or 80 AND might be reduced to a single function. In these cases it is more appropriate to have each primitive determine whether it can optimize itself if preceded by a number (rather than having LITERAL determine which primitives can be optimized and then generating the proper code). This allows words to be individually responsible for their own optimization. In keeping with the previous example, the sequence 1234 + can be compiled as:

Description	Machine Code		Cycle Time	Bytes
Optimized	mov.w	#1234,-(ds)	+12	+4
LITERAL ADD	mov.w	(ds)+,d0	+8	+2
macro	add.w	d0,(ds)	+12	+2
		Total	/32	/8
Optimized ADD	addi.w	#1234,(ds)	16	4

The optimized ADD reduces execution time from 32 to 16 cycles and generates only 4 bytes of code versus 8. Addition or subtraction of numbers from 1 to 8 can be further improved by using the ADDQ and SUBQ instructions. These "quick" instructions reduce code size to 2 bytes and execution time to 12 cycles. Processors other than the Motorola 68000 may also provide instructions for handling small integers.

CONSTANTs and VARIABLEs can be optimized by compiling code which directly performs their functions. A CONSTANT can be translated to its number value and compiled as a LITERAL. Similarly, a VARIABLE can be replaced by code which pushes the variable's address onto the stack.

Sequences such as <variable><memop> can be compressed. <memop> may be any word used to access memory. Suppose VARIABLE X was defined; then the sequence X ! could be compressed as shown below. Assume that the PFA (parameter field address) of X is 2E10 (the value of X is stored at this address).

Description	Machine Code		Cycle Time	Bytes
STC for X !	jsr.w	X	+18	+4
	jsr.w	STORE	+18	+4
DOVAR code in X	lea	4(pc),a0	+8	4
which pushes	mov.w	a0,-(ds)	+8	2
the PFA of X to	rts		+16	2
the stack	(2 data bytes)		nil	2
STORE (!) code	mov.w	(ds)+,a0	+8	2
to fetch address	mov.w	(ds)+,(a0)	+12	2
and store value	rts		+16	2
here			Total/104	/8
Optimized X !	mov.w	(ds)+,$2e10	16	6

This compression requires only 16 cycles and 6 bytes compared to the original 102 cycles and 8 bytes. Sequences of the form <number><variable><memop> (0 X !) can be likewise compressed. Similar, although less dramatic, results are obtained for CONSTANTs. Compression techniques may also be applied to access of user defined data structures.

Control structures are the next candidates for optimization. The basic flow of control is performed by the primitives DOIF, DOBRA, DOLOOP and DO+LOOP as discussed in section 3.8. Control structures can be optimized by replacing these primitives with machine code which directly performs the branch or test and branch operations. Control words can select the best type of branch to use (based on the branch offset and available branch instructions). This may be a long jump or, more often, a short relative branch. In either case, execution time is greatly reduced by performing a

direct processor branch rather than a subroutine call to emulate the branch.

Compression can be used to combine logical comparisons and control words. Logical comparison primitives would have already been replaced with machine code which pushes the result (a boolean flag) onto the stack. Control words would normally pop this flag, test it, and take appropriate action. The compare, test and branch functions can be combined to eliminate the need to push, pop and test the flag. Most processors maintain a condition code register which indicates the result of a comparison. This result is used to control conditional branches. Therefore, the condition code register replaces the need for a flag. The sequence 0= UNTIL is compressed as shown below:

Description	Machine Code		Cycle Time	Bytes
Optimized 0=	tst.w	(ds)	+8	+2
test top stack	beq.b	*+2	+10/8	+2
entry and set	clr	(ds)	+12	+2
flag to 0 or -1	mov	#-1,(ds)	+12	+4
Optimized UNTIL	tst.w	(ds)+	+8	+2
(DOIF primitive)	beq.b	BEGIN	+8/10	+2
			Total 58/58	/14
Compressed	tst.w	(ds)+	+8	+2
0= UNTIL	bne.b	BEGIN	+10/8	+2
			Total 18/16	/4

Similar compression is possible for sequences such as 5 < IF or 1000 > UNTIL. Numbers are compiled as immediate data within compare instructions:

Description	Machine Code		Cycle Time	Bytes
5 < IF	cmpi.w	#5,(ds)+	+12	+4
	bge.b	ELSE or THEN	+10/8	+2
		Total	22/20	/6
1000 > UNTIL	cmpi.w	#1000,(ds)+	+12	+4
	ble.b	BEGIN	+10/8	+2
		Total	22/20	/6

DO loops can be further improved by using primitive loop instructions which may be available on the target machine. Examples include the Intel 8086 LOOP (decrement CX register and branch on condition), National Semiconductor NS32016 ACB (add increment to register, compare and branch on condition) and the Motorola 68000 DBcc (decrement register and branch on condition) instructions. These instructions generally require the loop index to be kept in a register. This poses a problem for nested loops - each loop should have its own register. Forth has traditionally overcome this problem by placing loop indices and limits onto the return stack (refer back to section 3.8.3).

To achieve maximum speed it is necessary to allocate registers for loop indices. This cannot be effectively done without the use of a larger, smarter compiler which can view the interdependency and use of loops within the hierarchy of word definitions. Such a compiler may not be practical due to its complexity. Looping instructions can still be used provided that all loops use the same dedicated loop regis-

ter. Each loop must begin by saving this register, and must end by restoring it.

The loop register can be saved (pushed) onto the return stack and later restored (popped) from it. Inner loops can access the return stack to obtain the indices of outer loops (recall the words I and J serve this purpose). The current loop index is always kept in the loop register. The innermost loop will be most efficient since the loop register is saved and restored only once. All outer loops will implicitly save and restore the register each time a loop is executed.

Proper use of looping instructions is highly dependent on their function. For example, the 68000 DBcc and 8086 LOOP instructions decrement the loop register by one - restricting their use to DO ... LOOP constructs; whereas the 32016 ACB instruction can add any integer between -8 and +7 to the register. The calculation of initial values for loop index and limit should be tailored to the loop instruction. These calculations must be the same for both DO ... LOOP and DO ... +LOOP so that I and J can return proper indices based on the reverse calculation.

Loop instructions can be effectively replaced by two instructions: 1) decrement register and 2) branch on condition. This sequence allows LOOP and +LOOP to use a dedicated loop register regardless of existing (or non-existing)

loop instructions. In any case, the setup and termination of loops must be modified to use the loop register.

DODO, DOLOOP, DO+LOOP and DOLEAVE are modified as follows: DODO pushes a copy of the loop register onto the return stack rather than pushing the start value of the loop index. DODO also loads the loop index register with the start value. DOLOOP and DO+LOOP pop (restore) the loop register from the return stack rather than popping and discarding the loop index. Finally, DOLEAVE should be modified to compile a jump or branch to the loop termination code compiled by LOOP or +LOOP. This removes the need to push a LEAVE address onto the return stack during loop setup.

Loop setup and termination time is slightly improved by storing the loop index in a register instead of on the return stack. The major increase in performance is due to the faster increment/decrement of the loop register. Relative performance between loop implementations (assuming the loop is taken) is shown by the DOLOOP code:

Description	Machine Code		Cycle Time	Bytes
Normal STC	jsr.w	DOLOOP	+18	+4
	branch address		nil	+2
STC DOLOOP	mov.l	(rs),a0	+12	2
	addq.w	#1,2(rs)	+16	2
	bvs	exit loop	+8	2
continue loop	mov.w	(a0),a0	+8	2
	mov.l	a0,(rs)	+8	2
	rts		+16	2
exit loop	addq.w	#4,rs	8	2
	lea	2(a0),a0	8	4
	mov.l	a0,(rs)	8	2
	rts		16	2
		Total	/86	/6
Macro DOLOOP	addq.w	#1,(rs)	+12	+2
continue loop	bvc.b	DO	+10	+2
exit loop	addq.w	#4,rs	8	+2
		Total	/22	/6
Loop register	addq.w	#1,reg	+4	+2
continue loop	bvc.b	DO	+10	+2
exit loop	mov.w	(rs)+,reg	8	+2
	addq.w	#2,rs	8	+2
		Total	/14	/8
Loop instruction	dbf	reg,DO	+10	+4
exit loop	mov.w	(rs)+,reg	8	+2
	addq.w	#2,rs	8	+2
		Total	/10	/8

Note that the code size is either 6 or 8 bytes whereas the execution time varies from 10 to 86 cycles. Loop setup and termination times are excluded since they do not fall within the body of the loop (plus they are roughly equivalent for each implementation). These are nonetheless shown for comparison.

4.10 SUMMARY

The choice of a virtual machine implementation will vary depending on system requirements. Execution speed, code size, existing hardware and other factors must be considered. These will limit the number of viable implementation alternatives.

Compilation to machine code involves the use of STC in conjunction with macros, compression techniques and loop optimization. The resultant code is reasonably compact and extremely fast. Thus optimized machine code offers an excellent alternative to standard TC, especially if execution speed is important.

Machine code and STC do, however, have several disadvantages. Debugging aids are limited by replacing the "virtual machine" (address interpreter) with direct execution. The address interpreter can incorporate high-level debugging facilities into the execution of TC (eg. tracing word execution). In contrast, machine code requires special attention and hardware support in order to implement breakpoints, tracing and single-stepping of machine instructions. Higher-level debugging aids cannot easily be implemented due to the semantic gap between HLL and machine code. On the other hand, the semantic gap between Forth and TC is very small, thus facilitating Forth-level debugging.

Macros and optimization techniques may be impractical. This is generally the case for simple 8-bit processors such as the Motorola 6800 or Mostek 6502. These processors have few registers and addressing modes. This results in excessive use of temporary variables and a need to save/restore registers. Macro generated code can be quite large. These factors make other forms of TC more attractive and consign machine code compilation (and associated techniques) to use with more powerful processors.

Chapter V

FORTH IN HARDWARE

Forth's virtual machine can be implemented on any standard processor using the techniques previously discussed. The other alternative is to implement the virtual machine in hardware rather than software. A "Forth Engine" is created to directly execute the TC and Forth primitives. The advantage is faster execution speed. A text interpreter is still required to translate Forth source code into TC.

A Forth engine can be viewed as both a high level language (HLL) processor and a reduced instruction set computer (RISC). The Forth engine is an indirect execution type HLL processor since it executes an intermediate language (IL) [Chu81]. The engine is also classified as a RISC due to its simple instruction set architecture [Wal85b].

Forth is particularly well suited to hardware implementation. Unlike many HLLs, Forth closely resembles its IL (TC) and its IL can be easily implemented in hardware. This combination affords an HLL processor with a small semantic gap between HLL (Forth) and machine code (TC). The benefits of a RISC (simple hardware and fast execution times, to name a few) can also be realized. In short, the simplicity of Forth can be exploited in hardware [For84].

Several papers have discussed proposals for a Forth instruction set. [Sol83] and [SoR84a] describe an abstract Forth machine which executes F-code. There are only 22 F-code instructions from which a Forth nucleus has been implemented. [KRA81] presents a 32-bit machine with 100 instructions. Up to four 8-bit opcodes can be fetched in parallel.

The first Forth machine was probably an English Ferranti computer built around 1973. This machine was redesigned to accommodate a Forth-like instruction set [Moo80]. More recent endeavors have taken the form of single-chip microcomputers, microcoded processors, bit sliced or discrete component designs and, most recently, custom Forth processors.

5.1 SINGLE-CHIP MICROCOMPUTERS

Single-chip microcomputers can be used to create Forth machines. Examples include the Rockwell R65F11 and F68K [Dum84]. Each chip contains a microprocessor, RAM, ROM and may include I/O lines, timers and other peripheral devices. The single-chip microcomputer is converted into a Forth machine by programming the address interpreter and primitives into on-chip ROM. The remainder of the Forth kernel is placed in external ROM. This arrangement gives the appearance of a Forth processor, however, the implementation is still bound in software. There is no increase in performance. These chips serve as low cost, dedicated Forth machines for control applications.

The R65F11 is based on the Mostek 6502 architecture. Its 3K ROM contains the address interpreter, 133 primitives and higher level words plus a small "micro monitor" (text interpreter). The F68K utilizes a Mostek 68200 (based on the Motorola 68000) with a 4K on-chip ROM. The F68K is functionally identical to the R65F11 but runs 20 times faster.

5.2 MICROCODED PROCESSORS

Most popular 16/32-bit microprocessors are microcoded. Each instruction is executed as sequence of microcode instructions. The microcode contains control information necessary to perform the low level steps which make up each machine instruction. Microcode is kept in a fast ROM (usually on-chip) known as the control store. Processors such as the Motorola 68000 have a fixed control store which defines the instruction set. If possible, this control store could be altered; thereby modifying the instruction set. Thus it would be possible to take a standard processor and alter its microcode to implement the instruction set of a Forth machine. Unfortunately, processors with fixed control stores can only be mask programmed at production time. It is therefore impractical to use a processor with fixed control store. (Although IBM has modified the 68000 control store to emulate a large portion of the IBM 370 mainframe instruction set.)

The best solution is to use a processor which has a writeable control store (WCS). This allows the control store to be re-programmed. The NCR/32 is one example of a processor which has a WCS. A high performance polyFORTH II system has been ported to the NCR 9300 - a 32-bit virtual memory machine based on the NCR/32 chip set [McB84]. The improvement in execution speed is shown by the cycle times for several primitives.

Primitive	NCR/32	68000 16-bit DTC
DUP	9	38
DROP	10	34
SWAP	12	54
OVER	12	42
ROT	17	66
@	10	46
!	15	46
+	11	46
1+	8	38
AND	11	46

5.3 BIT SLICED OR DISCRETE COMPONENT

A Forth machine may be implemented using bit sliced elements which act as processor building blocks. These functional blocks may be combined together with a microcode instruction sequencer and control store ROM. This allows a custom design and definition of microcode tailored to Forth. Non-microcoded bit sliced designs are also possible. Bit sliced elements may be replaced or incorporated with discrete logic components. Faster logic families, such as ECL,

may be used. Special hardware may be added for dynamic address translation, stack validation, barrel shifting, etc.

The development of a bit sliced or discrete component Forth machine is no easy task. A good design should exploit the use of parallel operations in hardware. This type of Forth machine is practical only if money and time are small considerations (since high speed parts can be very expensive and hard to work with).

Several very quick Forth machines have been built using Advanced Micro Devices AMD290x series bit sliced components [GrD84][Bal84]. [VaS84] describes one implementation which separates data and instructions into separate memories in order to allow parallel access to these. [Vic84] defines a machine with 8-bit opcodes and 32-bit words. Separate stack, task environment, code and data memories are utilized. Multitasking and process control words are implemented along with fast context switching.

Several board-level Forth processors are commercially available [Col85]. For example, the Metaforth MF16LP is a single-board processor built in bipolar logic [Pou85]. The MF16LP has 16-bit data paths and a 32-bit address bus. A partial WCS is provided. It has been demonstrated that a full FORTH-79[17] system can be implemented with a minimum set of 27 instructions (out of 40 or so available). The final

---

[17] FORTH-79 is an older Forth standard published in 1979.

production model MF16LP should execute at least five million Forth instructions per second.

5.4 CUSTOM FORTH PROCESSORS

The next step is to implement the Forth engine as a custom Forth processor. The viability of HLL processors has been proven by efforts such as SOAR (SmallTalk on a RISC) [UBF84], Scheme-79 (Lisp on a chip) [SHS81], Acaps (Ada-directed processor) and the Texas Instruments CLM (Compact Lisp Machine) [Col85]. The LEM language and M3L processor is another example of the desire to close the semantic gap between hardware and software [SPB82].

RISC type HLL processors have been recently developed for direct execution of TC. These processors are specifically designed to execute Forth (although other HLL compilers are possible). Forth chips currently available include the NOVIX NC4000P and Symbolic Controls 4thCPU-40. These chips are similar in design and performance - only the NC4000P is discussed in further detail.

The NC4000P was the first available Forth chip made possible by the advent of affordable ASIC (application specific integrated circuit) technology. The NC4000P development team, headed by Charles Moore (the inventor of Forth), was able to complete the chip design and simulation from May through July of 1984. The layout was released to Mostek in

August, and a working (first time in silicon) chip was available in December of that same year.

The NC4000P incorporates a parallel architecture on a high-speed, 8MHz, 4000-gate CMOS chip [Mur85][GMB85][Bro85]. This 16-bit Forth microprocessor allows simultaneous access to main memory, data stack, return stack and two I/O buses (see figure 5.1 [GMB85]). Most instructions execute in one cycle (125 nanoseconds) and can perform up to five operations in parallel. Combinations of Forth words (eg. SWAP DROP) are compressed into a single instruction. The execution speed of eight million instructions per second results in an effective speed of over ten million Forth words a second!

Each instruction follows the hard-wired (non-microcoded) RISC approach. Each 16-bit opcode contains bit fields which directly control one or more of the 40 operations available to the processor. These operations combine to create a set of over 170 single-opcode instructions (many of these correspond to Forth words or word sequences). Most instructions, including subroutine calls, jumps and loops are executed in a single clock cycle. Division, multiplication and square root step instructions must be executed 16 times to perform a 16-bit operation. The times instruction sets a repeat-instruction counter which allows any instruction to be repeatedly executed. This is useful for division, multiplication, block memory moves, etc. Memory fetch (@) and store (!) operations require two cycles.

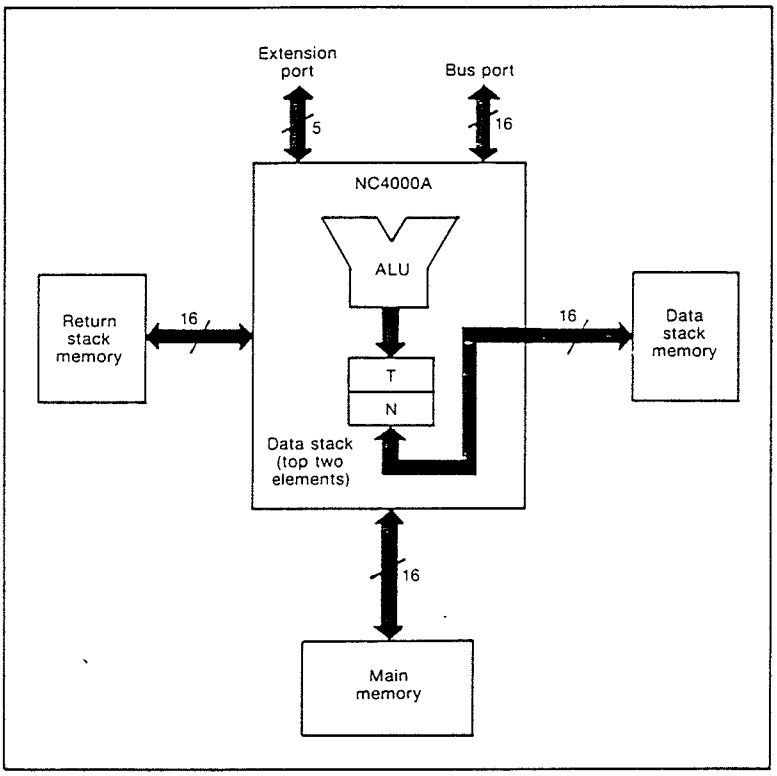


Figure 5.1: NC4000P Block Diagram

Main memory, return and data stacks are kept in external RAM. Fast static CMOS RAM is required to keep up with the processor. All memory is accessed as 16-bit words. The program counter and stack pointer registers contain word (not byte) addresses. Main memory can contain up to 64K words (programs must be stored in the first 32K). A 5-bit extension port allows extended memory addressing of up to 4 MB. The return stack can accommodate 256 entries. The data stack holds up to 258 entries (the top two entries are kept in on-chip registers). Finally, an extremely flexible 16-bit I/O port is provided as an interface to external devices.

The NC4000P is able to directly execute a form of direct threaded code. This allows Forth programs to execute over 100 times faster than possible with software emulated Forth machines. The NC4000P itself is said to offer a ten fold increase in performance over standard 16-bit microprocessors [Wal85a]. High-level Forth programs can execute ten times faster than equivalent 68000 assembler routines. Novix claims that the NC4000P executes the Sieve[18] benchmark in 0.3 seconds versus a 68000 assembly version (0.49 seconds) and Pascal version (14.0 seconds). Tests were performed using an 8 Mhz 68000 and 8 MHz NC4000P [Wol85].

Subroutine call overhead is virtually eliminated - the NC4000P requires only one cycle to perform a subroutine call (other 16-bit processors require 15 or more clock cycles). This contributes greatly toward increased performance since it has been shown that procedural languages such as Forth, C and Pascal spend up to 40% of their time as subroutine call overhead [PaP82].

The NC4000P's execution unit and opcode format are shown in figure 5.2 [GMB85]. The first four opcode bits indicate the instruction type. The remaining twelve bits control the ALU (arithmetic logic unit) and register operations. The first bit in an opcode indicates if the instruction is a subroutine call. If the bit is zero, the remaining fifteen

---

[18] The Sieve of Eratosthenes is a simple high-level language benchmark.

bits give the subroutine address. The instruction sequencer places this address on the address bus in time to latch the first instruction of the subroutine on the next cycle. This

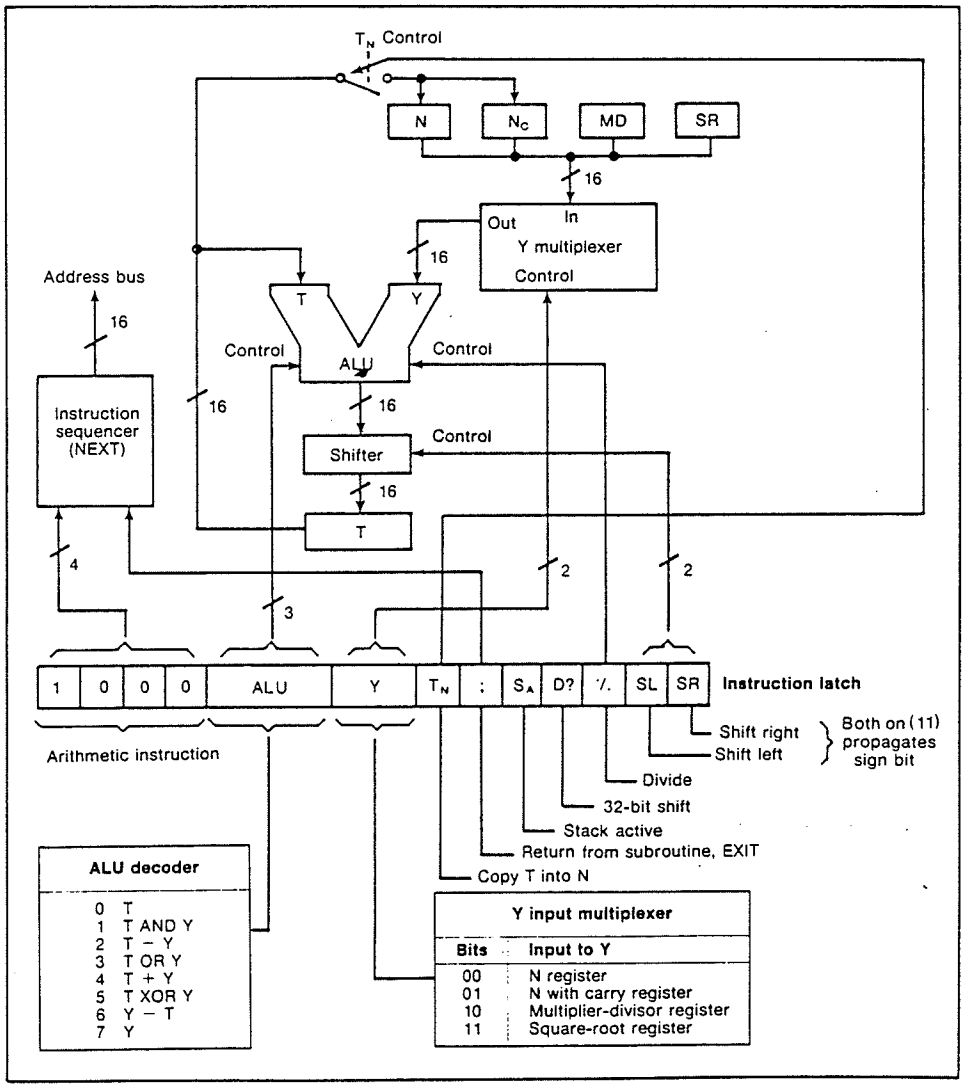


Figure 5.2: NC4000P Instruction Set Architecture and Execution Unit

allows a subroutine call to be executed in one cycle.

The 15-bit address limits programs to the first 32K words of memory. This restriction is counteracted by the compact object code. For example, the sequence OVER SWAP - is compressed into a single instruction: the second stack entry (register N) is output to the ALU while the ALU subtracts the top stack entry (register T) from register N.

Opcodes whose first bit is one are used to define arithmetic instructions, jumps, loops, etc. Each of these instructions can be combined with an EXIT bit (bit five in the opcode). When the EXIT bit is set, the NC4000P executes the instruction and performs an RTS (return from subroutine) at the same time. Thus, in most cases, the last instruction of a subroutine is executed in parallel with the RTS. An RTS effectively requires no cycle time!

Jump and loop instructions contain an address offset within the opcode. This allows the instruction sequencer to determine the next program counter address in time for the next cycle. All control flow instructions can be executed in one cycle.

The NC4000P represents a state-of-the-art, single-chip, high-performance Forth engine. A single-board Forth computer has been designed around the NC4000P. This board contains polyFORTH II in ROM and has eight stack segments to allow multitask switches in less than five microseconds. The polyFORTH compiler (text interpreter) is modified to al-

low compression of certain word sequences into single NC4000P instructions. The NC4000P is "hidden" from the programmer. The only noticeable differences are increased speed and the lack of an assembly language facility (Forth now replaces assembler)!

A list of Forth words and word sequences which are directly implemented by the NC4000P is given in the appendix "NC4000P Instruction Set" [GMB85].

Chapter VI

CONCLUSIONS

A standard Forth system is relatively simple to implement. This is largely due to the use of a virtual machine model and a postfix notation. Forth is certainly lacking in some areas but does succeed in providing an extensible language within an integrated, interactive, incremental programming environment.

This chapter briefly discusses a list of enhancements to counteract Forth's shortcomings and improve the Forth development environment. Comments on the future of Forth are also included.

6.1 SYSTEM ENHANCEMENTS

Forth system components can be enhanced in a number of ways. New features such as Forth component libraries can be added [Jam85a]. Other features may precipitate from the type of hardware being used.

A virtual memory mapped machine presents Forth with a very large address space. Files could simply be implemented as arrays. The single-level store mechanism would eliminate the need for a block storage system. Words would be dynamically paged in and out as necessary.

6.1.1 Alternate Dictionary Structures

The dictionary is used to maintain word definitions and allow them to be located by the text interpreter. Dictionary entries are typically stored as a linked linear list. This organization requires a linear search for each word being compiled. Structures other than a linked linear list can be used to reduce search time and increase compilation speed. Hashed and binary search methods are the most popular alternatives.

[Col81] describes a four-way hashing scheme. Each vocabulary contains four hash chains; each chain pointing to a linked linear list of word definitions. A hash function is used to determine which chain to search ([Col81] hashes on two bits taken from the first character of the word name). This hash reduces the search to one quarter of the vocabulary; increasing search speed by roughly four times. The improvement in search speed depends on the number of hash chains and upon how evenly the hash function distributes words amongst the chains.

Binary search methods dramatically increase compilation speed within a large dictionary. The dictionary structure must accommodate the binary search, word redefinition, chronological ordering of words and the FORGET mechanism. Fast search speed is offset by slower insertion and deletion of words. [KoN84c] describes one implementation in which binary

search compilation is slower than linear search compilation for dictionaries containing less than 70 words.

The binary dictionary structure can be implemented in a variety of ways. Some implementations may restrict word redefinitions, use of vocabularies and the FORGET mechanism. These restrictions simplify the dictionary structure and search procedure.

A binary tree is one possible solution. Each dictionary entry requires a pointer to its father, left child, right child and the next entry in chronological order. This implementation uses more memory and requires (slower) indirect pointer access. FORGET is complicated and time consuming.

[KoN84c] removes the link fields from each header and places these in a separate pointer array. Pointers are kept in alphabetical order according to the word name each points to. A binary search is performed by successively halving a search interval within the pointer array. Insertion and deletion of words is relatively simple. FORGET <name> can locate <name> and remove all pointers above or equal to <name>'s address. Chronological ordering and redefinitions are preserved. Vocabularies have not been implemented.

The dictionary structure is very flexible. Some systems split header and code body fields into separate areas of memory. This allows headers to be manipulated without affecting the main body of the dictionary [Joo82b][KoN84b].

Other systems extend headers to include fields for code size, a RAM/ROM flag, view block number, shadow block number, etc. The RAM/ROM flag and code size are necessary when attempting to relocate a word definition. The view field gives the source screen number in which the word is defined. Shadow blocks are used to document each word or source screen. Additional fields can be incorporated to meet specific design criteria.

6.1.2 Extending the Address Space

The FORTH-83 Standard specifies 16-bit addresses. There is clearly a need to extend the address space beyond 64K bytes. Several 32-bit systems have removed this restriction by using 32-bit address and data stacks exclusively. This trend will reduce portability between older 16-bit and newer 32-bit systems.

A solution has been proposed which defines byte (C:8-bit), word (W:16-bit) and long (L:32-bit) data and address operators [BrS84]. Existing operators use the "normal" stack width (either word or long depending on the machine). The new operators can be used to guarantee code compatibility. Examples include:

```
W@L   ( L-addr -- W )
C!L   ( C L-addr -- )
L@W   ( W-addr -- L )
```

Operators for converting between types are also provided.

Token threaded implementations allow Forth to maintain its compact 16-bit address size within an extended address space [Hol84].

6.1.3 Decompiler/Editor

A decompiler/editor can be added to eliminate the need to store source code as screens [Bue84]. Each dictionary entry already contains the word name. Two new fields must be added to each dictionary entry in order to facilitate a decompiler. The first field contains the length of the code body. The second field contains a pointer to a comment list.

Threaded code (TC) can be easily decompiled. Each address or token corresponds to a word whose name is "known" to the system or can be found in the dictionary. The decompiler lists the name of each word found in the TC. High-level defining words, literals, CONSTANTs and VARIABLEs can be identified by their code address words or by additional information stored in the dictionary.

The decompiler should be integrated with a full screen editor that automates the decompile-edit-compile cycle. This decompiler/editor can be written in Forth and invoked as EDIT <name> (edit the word <name>). Since the "source code" is kept in the dictionary, it will be necessary to include the words SAVE and RESTORE which save or restore a

dictionary image, vocabulary or word to and from backing store. The editor may be expanded to highlight undefined words and display word calling relationships [Fil83]. Multiple versions of words can be kept and selected for testing. These extensions greatly enhance the development environment.

6.1.4 Forth-like Languages

Several programming languages have precipitated from Forth. These include: STOIC, PISTOL, REPTIL, RTL, IPS, SPHERE, FIFTH and MAGIC/L. Other languages such as SNAP and MACRO SPITBOL offer similar characteristics. Most of these languages have incorporated new features which should be considered as possible extensions to Forth.

STOIC (STack Oriented Interactive Compiler) [SaB83] offers several suggestions. STOIC uses a vocabulary stack as a means of specifying the dictionary search order. This seems to be a reasonable approach.

The text interpreter is modified to use a compile buffer rather than compiling directly to the dictionary. This feature allows control structures to be used outside of colon definitions as in:

```
0> 10 0 DO I = LOOP 0 1 2 3 4 5 6 7 8 9 OK
```

Note the '0>' prompt which indicates the level of nested control structures. This is useful for multi-line definitions.

String literals allow defining words to take all of their operands from the stack. For example, a colon definition no longer needs to look ahead to the next word in the input stream:

```
'SQUARE : DUP * ; OK
```

PISTOL (Portably Implemented Stack Oriented Language) [Ber83] is very similar to STOIC but is written in both C and Pascal language versions. The high-level language implementation was chosen to provide a high degree of portability. In addition, a large portion of PISTOL is written in PISTOL itself. PISTOL, like STOIC, uses string literals and a compile buffer. PISTOL improves the command line prompt by displaying the nesting level and the stack depth if these are non-zero.

REPTIL (REverse Polish Threaded Interpretive Language) [Uri84] again extends the STOIC and PISTOL prompts by displaying the current number base and the type of control structure nesting. The prompt 2D:?> indicates a conditional branch nested within a colon definition (two nested levels with the current number base set to decimal). This feature allows syntax errors to be easily recognized.

RTL (Relocatable Threaded Language) [Bue84] uses a modified form of token threaded code (TTC) so that all words are independently relocatable. This implementation also allows any word to be redefined retroactively without having to re-

compile words which call the modified word. Word names, code and variables are separated in memory so that the dictionary can be compacted to recover space generated by deleting words. A decompiler/editor is provided as discussed in section 6.1.3. Another utility allows the conversion of TTC to direct or indirect threaded code if increased execution speed is required [Bue85].

RTL features have been recently incorporated into REPTIL [Uri85]. The most significant departure from Forth is the replacement of source code screens with a decompiler/editor. Source code is regenerated from tokens into a "pretty-printed" structured format. This feature enforces a standard format and improves readability.

MACRO SPITBOL [DeM77] is a Forth-like implementation of the SNOBOL4 string processing language. MACRO SPITBOL uses a postfix compiler which generates indirect threaded code. Memory is divided into stack, static and dynamic store areas. The stack contains pointers to SPITBOL objects or code blocks. Operand type checking and type conversion are performed (if necessary) by each operator. A store allocator manages requests for memory blocks taken from dynamic store. A three-pass (mark blocks, relocate pointers and copy blocks) garbage collector is used to compact dynamic store if an allocation request fails.

SPHERE [Sol84] is much like STOIC but includes a pre-emptive, priority-driven multitasking executive. Each task has its own local data structures and stacks. The highest priority task is always run first (priorities may dynamically change). If several tasks have highest priority then a round-robin approach is used to give each task a CPU time-slice. Tasks communicate using global variables as mailboxes. Executive calls are provided for doing process synchronization.

MAGIC/L [EpG85] attempts to combine the advantages of Forth with a more traditional syntax. MAGIC/L is an interactive, extensible, threaded interpretive language. A large incremental compiler provides for a more stable and maintainable syntax. MAGIC/L data types resemble those of C and Pascal. Arrays, record structures and local variables can be defined.

SNAP resembles MAGIC/L but is said to

> "combine the best of BASIC, LISP and FORTH with the readability of Pascal" [Pou84].

SNAP is an interactive, incrementally compiled language. Each function definition specifies its parameters and the number of entries it returns. This information is used to validate the parameter stack size during execution. Local variables, arrays and list handling functions are provided. Scoping lids allow definitions to be grouped into modules. Each module specifies which of its functions and variables are visible outside of that module.

6.1.5 Debugging Aids

Forth is capable of providing a large set of hardware and software debugging aids. Forth acts as a powerful interactive monitor or command language: words like @ and ! can be used to access memory or peripheral devices; test routines can be quickly developed. FIFTH, an extension of Forth, has been developed to support an interactive hardware and software debugging environment [LiW85]. FIFTH provides an extensible command language, a set of debugging tools and tool-building primitives.

For software debugging, Forth has access to the dictionary, data and return stacks. All compile-time information is available at run-time. This allows complete source-level debugging.

[Sor84b] describes a simple break point utility for non-CODE words. A break point is generated by replacing a word address with the break point monitor address. This causes a trap to the break point monitor at execution-time. The monitor can restore the word address and optionally move the break point forward to provide single-stepping.

[Lew83] describes a VERIFY utility. VERIFY <name> executes <name> and displays the before and after effects of <name> upon the stack. VERIFY can also generate a memory access listing by temporarily patching !, @, C!, and C@ during word execution.

The above utilities can be implemented on any threaded code system. More powerful utilities may be built into a debug version of the address interpreter. Additional information such as a trace flag, break point flag, execution count, etc., can be added to each dictionary entry. The address interpreter will access or update this information at execution-time. Debugger words can be defined for controlling the debugging environment. For example:

Word	Function
DEBUG-ON	install debug address interpreter
TRACE <name>	set trace flag for <name>
FREQUENCY <name>	display number of times name was executed

All words can be listed and referred to by name. The user is able to breakpoint or trace individual words. Execution may continue until a specified number of words have been executed or until a particular word has executed n times. Many other features are possible: user-defined traps, type-checking primitives, full-screen display showing source code, stacks and highlighting word execution, etc.

6.1.6 File System

A file system extension word set can be written "on top of" Forth's block storage system. Implementations such as FORTHDOS [Ree82] and FMS [Rus81] have followed this approach. Each file is given a name and a block allocation

bitmap indicating which blocks are allocated to the file. The file word set allows files to be created, deleted, copied, opened, closed, etc. Directory listing, sequential and random access I/O words are also included.

A different approach can be taken if Forth is run as a task under some other operating system. In this case, the file word set is used as an interface to the host file system. A proposed word set, modeled after the UNIX file system, is described in [Bra84].

Forth has also been adapted to allow source code to be saved to and loaded from a file on the host machine. Some systems have taken the reverse approach by implementing Forth's block storage system via the host file system. A standard file word set has yet to be developed.

6.1.7 Multitasking

A multitasking Forth system allows several words (tasks) to be simultaneously executed. This feature is extremely useful when designing complex real-time applications. Multitasking capabilities can be easily added to Forth. Each task is given its own local stack and a copy of the registers used by the virtual machine.

A set of words can be defined for creating and controlling tasks. A simple non-preemptive round-robin task scheduler can be used [Pou85b]. Alternatively, each task may be

given a time-slice according to its priority. In any case, the task scheduler can switch tasks by saving the virtual machine environment of the current task and restoring the environment for the new task. For example, we might say:

```
20 40 TASK: ALARM
            BEGIN
               TEMPARATURE @ 100 <
               WHILE
                  PAUSE
            REPEAT
            BELL ." It's Boiling!" ;

ALARM RUN
```

which defines the task ALARM with a 20 byte return stack and 40 byte data stack. ALARM is placed into the ready task queue by saying ALARM RUN. ALARM checks the variable TEMPERATURE and PAUSEs (relinquishes control back to the task scheduler) as long as TEMPERATURE is less than 100. The task scheduler continues to execute the next ready task.

Forth may be useful in a heterogeneous multiprocessor system. A multitasking, multiprogrammed Forth could distribute machine-independent tasks amongst available processors. Each task would contain relocatable, machine-independent code (ie. indirect token threaded code). A task could execute on any processor since all processors behave as the same type of virtual machine (disregarding peripherals).

Semaphores, queues [LeM83] and mailboxes [Dob85] can be defined to provide intertask communication and synchronization. Hardware may also be incorporated to provide fast context switching [Vic84].

6.1.8 Interrupt Handling

Many real-time applications make use of hardware interrupts to signal events such as a timer running out or a sensor begin triggered. These situations require a mechanism which would allow an interrupt to invoke a Forth word. [Oze82] concludes that

> "Interrupt mechanisms are implementation dependent and the standardization of interrupt handling words, if possible, is not a simple task."

A set of interrupt handling words can be designed for a particular processor. Most processors have some form of vectored interrupt. Interrupt number n fetches the nth interrupt vector and jumps to this address. Interrupt vectors must be modified to invoke the appropriate Forth word. For example, n INTERRUPT <name> could associate the word <name> with interrupt n.

The interrupt mechanism should allow interrupt routines to maintain all the properties of regular Forth words. However, interrupt routines should preserve system stacks and processor registers. Code which saves and restores processor registers should be automatically generated if necessary. The user must also be able to define interrupt routines in assembly or high-level Forth. A more complete discussion of asynchronous words, based on a process model, can be found in [WiP84].

6.2 LANGUAGE ENHANCEMENTS

Extended language features include forward references, scope control and mixed language subroutines [Seb85]. Assembly language words may be allowed to call other assembly and high-level Forth definitions [Sim85]. Other enhancements are discussed in the following sections.

6.2.1 Data Structures

More traditional data structures can easily be added to Forth. A heap data structure is discussed in [Dre86]. Alternate parameter stack definitions are given in [Hel81].

[Bra85a] describes one method of defining record structures. Each record may contain bit and byte fields as shown below:

```
struct ( control-register )
   4 bits    command
   4 bits    error-num
   2 field   data-ptr
   4 field   dma-base
   2 field   dma-count
constant control-register-size
```

Each structure keeps a byte count on the stack so that this may be used in defining a constant such as 'control-register-size'. The control register structure could be included within another structure by saying 'control-register-size field control-reg'.

[Bas83] discusses the general implementation of any data structure in Forth. One example details the use of multi-dimensional arrays of the form:

```
DIM 2 2 4 ARRAY TOTALS              ( integer array )
DIM 20 10 CHARACTER ARRAY NAMES     ( string array )
DIM 8 8 BYTE CHECKER-BOARD          ( byte array )
```

6.2.2 Floating Point

Floating point (FP) arithmetic is often considered an important feature of a language. FP does simplify programming in some instances where a large range of numbers are required. The major drawback to using FP is that FP operations are slow (unless hardware support is available). On the other hand, fixed point operations are fast and are usually sufficient for most problems. Fixed point requires the programmer to keep track of the decimal point (eg. dollar values can be stored as cents - an integer value with two implied decimal places).

Many complex applications have been written using fixed point arithmetic. These include fast fourier transformations, linear regression and solving of differential equations. In lengthy calculations fixed point can give a dramatic increase in performance over FP. Nonetheless, FP is necessary in many applications.

FP is usually added as an extension word set [DuT84]. FP operators are analogous to the double word operators except that they are prefixed with F as in F+, F*, F<, F@, FDUP, etc. Additional words are provided for FP transcendental functions, FCONSTANTs, FVARIABLEs, number display and conversion between FP and double integer values.

FP words may use the standard parameter stack or a separate FP stack (depending on the implementation). There is still some debate as to whether or not a separate FP stack should be required.

6.2.3 Infix Expressions

A separate INFIX vocabulary can be easily added to allow the use of infix algebraic expression [Hel82]. This is done by defining infix operators such as *, /, +, -, (and) that compile postfix code. Each operator is represented by a duple containing the word address and precedence of the operator. A temporary stack is used to hold duple entries.

Operators are created by a high-level defining word. Each operator compares its precedence with the precedence of the top duple entry. If the operator precedence is greater or equal, then the operator pushes its duple onto the temporary stack. Otherwise, the operator pops all duples of higher precedence and compiles or executes them depending on STATE. This process performs the infix to postfix conversion.

6.2.4 Named Parameters and Local Variables

Named parameters and local variables enhance Forth's readability and simplify programming. Parameters and local variables are normally kept on the stack and manipulated us-

ing DUP, SWAP, ROT, etc. Stack operators cannot easily access more than three values; they are also inconvenient when several values must be preserved for repeated use. These limitations force the programmer to consider an order of operations which maintain and provide access to parameters and local variables while minimizing stack manipulation. This results in code that is difficult to read; we must keep track of stack entries at each step within a word definition.

The need for named parameters and local variables is evident from the large number of papers discussing this topic [Joo82a] [Bar82] [Gla83] [KoN84a] [HaP85] [Lyo85] [Sto85] [Gre84b]. Several methods are proposed for implementing named parameters and local variables in Forth. The general idea is to define a parameter packet or stack frame which is kept on the data stack, return stack or an additional stack.

A stack frame is dynamically allocated on entry to a colon definition and is released upon exit. Parameters and local variables are kept within the stack frame and accessed relative to the frame pointer (FP). FP points at a specific entry in the stack frame (eg. the first parameter or first local variable). A parameter's address is calculated as the sum of FP and the parameter's offset within the stack frame. The FP is saved on entry (before being set for the current frame) and later restored on exit.

Their are two approaches to defining named parameters. The first approach provides a common set of parameter names (PAR1, PAR2, ...PARn or P1, P2, ...Pn) which return the address of the nth parameter. Alternatively, operators such as @n and !n could be used to access the nth parameter. These assume each parameter is a 16-bit (constant length) stack entry. The second approach is more difficult to implement but allows user-defined parameter names.

[Joo82a] discusses a simple method of implementing parameter access. For example:

```
: FACTORIAL   ( n -- n! )
   1 INPUT                  ( one input parameter )
   0 PAR @                  ( fetch first parameter )
   IF
      0 PAR @ 1- RECURSE    ( calculate n-1! )
      0 PAR @ *
   ELSE
      1
   THEN
   0 PAR !                  ( store result )
   1 OUTPUT ;               ( one output value )
```

Note that 0 PAR @ can be replaced by PAR0 @ or @0 (the later is fastest). INPUT and OUTPUT can be used to check the stack for the proper number of input parameters and output results.

[Gla83] allows the non-recursive factorial definition:

```
<DECLARE VAR N LOCAL RESULT
<DEFINE FACTORIAL   ( n -- n! )
   N 1 >   IF
      N -> RESULT           ( set RESULT to N )
      N 1 DO I RESULT * -> RESULT LOOP
      RESULT
   ELSE
      1
   THEN ;
```

The word '->' is used to pop the stack and assign this value to the following variable. This corresponds to the 'TO' concept discussed in section 6.2.6.

[Sto85] details a comprehensive implementation for passing arguments by value or address. Named local variables and loop indices are included. For example:

```
: FACTORIAL   ( n -- n! )
   { num n   internal result   index count }
   n 1 >=
   IF
      n to result          ( set result to n )
      n 1 with count       ( use count as index )
      DO count result * to result LOOP
      result
   ELSE
      1
   THEN ;
```

User-defined parameter types are also allowed.

The advantages of named parameters and local variables are obvious. The penalty for these is roughly a 10-30% increase in execution speed (and possibly code size).

6.2.5 Scope Control

[Gre84b] proposes a simple method of implementing local variables and word definitions. A LOCAL vocabulary is defined such that LOCAL DEFINITIONS causes subsequent words to be compiled as separate header and code body fields. Code bodies are compiled to the dictionary while headers are placed in a local header buffer. This buffer (LOCAL vocabulary) is searched first so that local definitions have precedence over all others.

LOCAL DEFINITIONS clears the local header buffer and changes the compilation vocabulary to LOCAL. END-LOCAL can also be used to clear the header buffer. Local definitions are provided as a result of being able to clear the header buffer. Separate headers are used to create "Orphans" (headerless word definitions). Any locals defined before LOCAL DEFINITIONS or END-LOCAL are no longer accessible. This allows a single level of scope control as shown below:

```
LOCAL DEFINITIONS          ( begin new set of locals )
VARIABLE A ...
: UTILITY ... ;


APPLICATION DEFINITIONS    ( start or continue global )
VARIABLE B ...             ( definitions which use )
: PLOT ... ;               ( preceding locals )


END-LOCAL                  ( make locals inaccessible )
```

6.2.6 The QUAN and TO Concepts

The QUAN and TO concepts allow a single word to perform several functions based on a prefix modifier [Dow83]. Each word contains multiple code field addresses (CFAs or executable word addresses). The prefix modifier determines which CFA will be executed. The TO concept selects its CFA at execution-time; QUAN selects its CFA at compile-time.

The TO concept is designed to prevent unintended access by the ! operator. VALUE is similar to VARIABLE but is used to define words which store or return a value. The function of a VALUE word is determined at execution-time according to

the %VAL flag. %VAL controls which CFA is executed. TO is the prefix modifier used to set %VAL. The TO concept can be illustrated as follows:

```
0 VARIABLE %VAL OK              ( create %VAL flag )

: TO   ( -- )
   1 %VAL ! OK                  ( set %VAL )

: VALUE    ( n -- if %VAL is zero )
           ( -- n if %VAL is non-zero )
   CREATE ,                     ( create initial value word )
   DOES>                        ( execution-time function )
      %VAL @                    ( checks %VAL flag )
      IF
         ! 0 %VAL !             ( set value and reset %VAL )
      ELSE
         @                      ( fetch value )
      THEN OK

3 VALUE COUNT OK                ( define a VALUE word )
7 TO VALUE OK                   ( set COUNT to 7 )
COUNT . 7 OK                    ( print value of COUNT )
```

The function of VALUE is determined by the presence or absence of TO.

The QUAN concept is similar to TO but selects one of three CFAs at compile-time. A QUAN can return a value, store a value or return the address of a values storage location. The prefix modifiers IS and AT are immediate words:

```
0 QUAN X OK                     ( define X )
5 IS X OK                       ( set X to 5 )
X . OK                          ( print value of X )
AT X . 14728 OK                 ( print address of X data )
```

The QUAN concept reduces execution time by compiling the appropriate CFA rather than selecting it at run-time. QUANs are also faster since they are accessible in one operation (X or IS X) whereas variables (X ! or X @) and TOs (TO X) require two operations.

QUAN and TO are interesting in that they provide a form of data hiding. The prefix modifiers IS, AT and TO are data-independent. We can say 5 TO X or 5 25 TO XARRAY (set the 25th element of the VALUE_ARRAY X).

[Sto85] takes the QUAN concept a step further by allowing the user to define any number of CFA's and associated prefix modifiers. This allows new data types to be given a set of modifiers which control the method of access. Each data type has its own methods. For matrices we might say:

```
4 4 MATRIX X OK                ( define matrices X and Y ) 4 4
MATRIX Y OK
17 to 4 2 of X OK              ( set X[4,2] to 17 )
4 2 val of X . 17 OK           ( fetch and print X[4,2])
X to Y OK                      ( copy matrix X to Y )
```

The appropriate CFA is compiled based on the value of ACTION-KEY. Each prefix modifier sets one bit in ACTION-KEY. This technique improves readability and allows prefixed data objects to select an access method for their data.

6.2.7 Generic Operators and Data Type Checking

Forth normally defines one set of operators for each data type [HoF83]. This can result in a large number of operators (F+, D+, +, etc). In addition, Forth makes no provision for type checking. A typeless stack allows boolean, integer, long and address data to be intermixed. Forth words have no way of verifying proper operand types. Generic operators (or words) require data type information.

Data type checking is facilitated by adding a separate type stack. Type information could be incorporated into the data stack but would not be as easily accessible. A separate type stack allows a quick block memory comparison for expected types. Word definitions must include parameter type declarations that are compiled into a type list which is suitable for direct comparison to the type stack. This is necessary in order for execution-time type checking to proceed as quickly as possible. Dynamic type checking is implicitly required by the stack. All operands and results push their value to the data stack and their type to the type stack.

Dyadic generic operators such as + and * can use the type information as row and column indices into a table of execution addresses. Addresses along the diagonal correspond to words that operate on the same data type (F*, D*, etc). Words not found on the diagonal must perform the appropriate conversion between types. Unary generic operators only require an array of word addresses.

Generic operators and dynamic type checking can be implemented in high-level Forth. Type checking can be done by each operator and word or may be incorporated into the address interpreter. In any case, the user should be able to define new types and methods by which operators use these types. Type checking is desired during program development but may be turned off for improved execution speed.

This typed environment leads to higher levels of data abstraction such as the introduction of a single stack in which each entry contains a type field and a pointer to an object of that type. Dynamic memory allocation and garbage collection follow from this. Further study indicates that hardware support is necessary in order to achieve reasonable execution speed with this complexity. Separate type and data stacks offer a more practical software solution.

6.3 THE FUTURE OF FORTH

Forth was originally designed to operate in a memory limited environment. This restriction no longer applies to most Forth systems, although it may be enforced for certain applications. There is no doubt that memory size is increasing faster than ever due to higher density, lower cost chips. Forth no longer needs to be as memory conscious within its development environment. Enhancements to Forth are including larger compilers, more elaborate dictionary structures, advanced editors, debuggers and other development tools.

Forth is interactive and extensible at all levels. This gives the programmer unprecedented control over the entire machine and software environment [Jam85b]. The question arises: what is wrong with Forth? Many software developers have continued to reject Forth despite its proven abilities and advantages.

The main problems are lack of application tools and support for software modules. Code can be structured but undisciplined. Forth does not require words to be organized into cleanly defined modules. This organization is necessary for large or multiprogrammer applications. Modules provide data hiding which in turn simplifies program development and maintenance.

Most Forth systems do not provide application tools. Instead, Forth becomes a language kit which allows (forces) the programmer to define his own set of tools. As a result, application programmers must acquire an in-depth knowledge of their Forth system. Application development tools should exist to simplify the application programmers task. These tools are, of course, separate "add-on" components.

Most of the enhancements mentioned in this chapter can and should be incorporated into a Forth development system. At present, no such system is commercially available. Existing systems implement only a subset of possible enhancements. A "complete" system should be implemented using relocatable token threaded code for development; allowing conversion to faster TC types or compilation to machine code depending on final product requirements. The implementation should provide a full-screen editor/decompiler/debugger, improved word definition syntax, scoping control (modules) and type checking facilities.

Forth has already found its niche in real-time process control applications. Forth is both a language and a real-time operating system. An enhanced Forth would meet other needs and appeal to a larger audience. Would this enhanced language still be considered an extension to Forth or would it become a new language in itself? This question is left to the reader. In any case, Forth seems destined to change by nature of its extensibility.

Future Forth systems may be built around Forth machines: direct-execution Forth processors capable of executing tens of millions of Forth instructions per second. Fast on-chip stacks and a large cache with separate instruction and data streams will be used.

Forth has the advantage of being easy to implement in hardware. This is illustrated by the NC4000P Forth processor: a 4000-gate implementation which is small in comparison to the Inmos Transputer (250,000 transistors) and the Intel 80386 (275,000 transistors). The Texas Instruments 32-bit Lisp processor is reportedly ten times the complexity of a Motorola 68000.

Future Forth processors will be able to take advantage of chip space freed up by their RISC architecture. Stack and cache areas can be placed on-chip. Floating point, virtual memory and dynamic type checking hardware could also be placed on-chip if space permits them.

In conclusion, the future of Forth looks very promising. Forth is gaining interest due to a recognition of its abilities. In addition, Forth holds the promise of an improved development environment: an interactive, extensible language coupled with a powerful set of development tools and boosted by the performance of a high-level language Forth processor.

Appendix A

TARGET MACHINES

Burroughs 5500
Univac 1108
IBM 360
HP 2100
PDP 10
Varian 620
GA SPC-16
IV-Phase
RCA 1802
IBM Series 1
Motorola 6800
Intel 8086/186/286
Motorola 68000/020
Zilog Z8000
Mostek 6502
National 32032
NCR/32
IBM 1130
Honeywell 316
Data General Nova
PDP 8
PDP 11/xx
Mod-Comp II
CDC-6400
Computer Automation LSI-4
Honeywell Level 6
Interdata
Intel 8080
Texas Instruments TI-9900
Zilog Z80
Motorola 6809
National 32016
VAX 11/xxx

Appendix B

NC4000P INSTRUCTION SET

Table 1. Instructions corresponding to single Forth words

Stack manipulation

DUP	(n — n n)	Push copy of top of stack onto stack
DROP	(n —)	Discard top stack element
OVER	(a b — a b a)	Push copy of second (next) stack element onto top of stack
SWAP	(a b — b a)	Reverse order of top two stack elements

Arithmetic and logic

+	(a b — sum)	Add top two elements as 16-bit two's complement integers
)c	(a b — sum)	Add with carry
−	(a b — a−b)	Subtract top eleMent from second element, as 16-bit two's complement integers
−c	(a b — a−b)	Subtract with carry
OR	(a b — or)	Bit-wise logical OR
AND	(a b — and)	Bit-wise logical AND
XOR	(a b — xor)	Bit-wise logical XOR
2/	(n — n/2)	Arithmetic-shift T register right one bit
2'	(n — n'2)	Arithmetic-shift T register left one bit
0<	(n — ?)	Return true flag (hexadecimal FFFF) if n is negative; otherwise false
D2/	(d — d/2)	Double-length arithmetic-shift right
D2'	(d — d*2)	Double-length arithmetic-shift left
•'	(d — d)	Multiplication step
•−	(d — d)	Signed multiplication step
•F	(d — d)	Fractional multiplication step
/'	(d — d)	Division step
/''	(d — d)	Last division step
S'	(d — d)	Square-root step

Return stack control

R >	(— n)	Pop top of return stack onto data stack
R@	(— n)	Copy top of return stack onto data stack
#!	(— n)	Copy loop index onto data stack
>R	(n —)	Push top of data stack onto return stack

Structure control

if		Jump if T register contains zero
else		Jump unconditionally
#loop		Jump and decrement loop counter if it is not zero
times	(n —)	Set repeat instruction counter
call		Jump to subroutine
EXIT		Return

Memory and I/O access

@	(adr — n)	Fetch value at memory address (2 cycles)
!	(n adr —)	Store value at memory address (2 cycles)
@	(adr — n)	Fetch value at local memory address (2 cycles)*
!	(n adr —)	Store value at local memory address (2 cycles)*
I@	(adr — n)	Fetch value from internal register
I!	(n adr —)	Store value in internal register
n (no name)	(— n)	16-bit literal fetch (2 cycles)
n (no name)	(— n)	5-bit literal fetch*

* Distinguished from preceding instruction(s) by internal structure

Table 2. Instructions corresponding to multiple Forth words

Stack manipulation

SWAP DROP	DROP DUP
SWAP −	SWAP −c
OVER +	OVER +c
OVER −	OVER −c
OVER SWAP −	OVER SWAP −c
OVER OR	OVER XOR
OVER AND	R) SWAP R)
R) DROP	

Full literal fetch

n +	n +c
n −	n −c
n SWAP −	n SWAP −c
n OR	n XOR
n AND	

Short literal fetch

nn +	nn +c
nn −	nn −c
nn SWAP −	nn SWAP −c
nn OR	nn XOR
nn AND	

Data fetch

@ +	@ +c
@ −	@ −c
@ SWAP −	@ SWAP −c
@ OR	@ XOR
@ AND	

DUP @ SWAP nn + (incrementing fetch)
DUP @ SWAP nn − (decrementing fetch)

Data store

DUP !
SWAP OVER ! nn + (incrementing store)
SWAP OVER ! nn − (decrementing store)

Extended address data fetch

nn X@ +	nn X@ +c
nn X@ −	nn X@ −c
nn X@ SWAP −	nn X@ SWAP −c
nn X@ OR	nn X@ XOR
nn X@ AND	

Extended address data store

DUP nn X!

Local data fetch

nn @ +	nn @ +c
nn @ −	nn @ −c
nn @ SWAP −	nn @ SWAP −c
nn @ OR	nn @ XOR
nn @ AND	

Local data store

DUP nn !	DUP nn ! +
DUP nn ! −	DUP nn ! SWAP −
DUP nn ! OR	DUP nn ! XOR
DUP nn ! AND	

Internal data fetch

nn I@ +	nn I@ −
nn I@ SWAP −	nn I@ SWAP OR
nn I@ XOR	nn I@ AND
DUP nn I@ +	DUP nn I@ −
DUP nn I@ SWAP −	DUP nn I@ OR
DUP nn I@ XOR	DUP nn I@ AND

Internal data store

DUP nn I!	DUP nn I! +
DUP nn I! −	DUP nn I! SWAP −
DUP nn I! OR	DUP nn I! XOR
DUP nn I! AND	nn I@!

REFERENCES

[Ant84] Anthony, T.R., "Laser Beveling in polyFORTH," The Journal of Forth Application and Research, Vol. 2, No. 2, 1984, pp. 5-24.

[Bal84] Ballard, B., "FORTH Direct Execution Processors in the Hopkins Ultraviolet Telescope," The Journal of Forth Application and Research, Vol. 2, No. 1, 1984, pp. 33-47.

[Bar82] Bartholdi, P., "Another Aid for Stack Munipulation and Parameter Passing in FORTH," 1982 Rochester Forth Conference, May 1982, pp. 221-230.

[Bar83] Barnhart, J., "Forth and the Motorola 68000," Dr. Dobb's Journal, No. 83, Sept 1983, pp. 18-26.

[Bas83] Basile, J., "Implementing Data Structures in FORTH," The Journal of Forth Application and Research, Vol. 1, No. 2, 1983, pp. 5-15, 79-80.

[Bel73] Bell, J.R., "Threaded Code," Comm. ACM, Vol. 16, No. 6, June 1973, pp. 370-372.

[Ber83] Bergmann, E.E., "PISTOL, A Forth-like Portably Implemented STack Oriented Language," Dr. Dobb's Journal, No. 76, Feb 1983, pp. 12-15.

[Bla83] Blaser, P., "The Development of a C-based Forth," 1983 Rochester Forth Conference, June 1983, pp. 161-164.

[BoF83] Boni, R., and Forsley, L.P., "Real-Time Interactive Spectroscopy (abstract)," 1983 Rochester Forth Conference, June 1983, p. 221.

[Bra84] Bradley, M., "Operating System File Interface," 1984 Rochester Forth Conference, June 1984, pp. 291-293.

[Bra85a] Bradley, M., "Structured Data with Bit Fields," 1984 Rochester Forth Conference, June 1984, pp. 188-192.

[Bra85b] Brakefield, J.C., "An Alternate Dictionary Structure," The Journal of Forth Application and Research (1985 Rochester Forth Conference), Vol. 3, No. 2, 1985, p. 112.

[Bro81] Brodie, L., Starting FORTH, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Bro85] Brodie, L. "A Threaded-Code Microprocessor Bursts Forth," Dr. Dobb's Journal, No. 108, Oct 1985, pp. 42-50.

[BrS84] Bradley, M., and Sebok, W., "Compatible Forth on a 32-Bit Machine," The Journal of Forth Application and Research, Vol. 2, No. 4, 1984, pp. 33-38.

[BSR84] Bailey, G., Sanderson, D., and Rather, E.D., "clusterFORTH, A High-Level Network Protocol, 1984 Rochester Forth Conference, June 1984, pp. 245-256.

[Bue84] Buege, B., "Status Threaded Code," 1984 Rochester Forth Conference, June 1984, pp. 103-104.

[Bue85] Buege, B., "Conversion of a Token Threaded Language to an Address Threaded Language," The Journal of Forth Application and Research (1985 Rochester Forth Conference), Vol. 3, No. 2, 1985, pp. 113-115.

[Bur84] Burger, A., "On Faster Forth," Byte, Vol. 9, No. 13, Dec 1984, p. 24.

[Buv84] Buvel, R., "A Forth Native-Code Cross Compiler for the MC68000," Dr. Dobb's Journal, No. 95, Sept 1984, pp. 68-107.

[Cal79] Calingaert, P., Assemblers, Compilers, and Program Translation, Computer Science Press, Rockville, Maryland, 1979.

[Car85] Caruso, D., "Development System Breaks Productivity Barrier," Electronics, July 8, 1985, pp. 36-39.

[Cha83] Chamberlin, M., "The Multiple Mirror Telescope Observatory Mount Servo Control System (Abstract)," 1983 Rochester Forth Applications Conference, June 1983, p. 223.

[Chu81] Chu, Y., "High-Level Computer Architecture," IEEE Computer, July 1981, pp. 7-8.

[Col81] Colburn, B.R., "Implementing a Four-way Hashed Vocabulary Structure for the FIG-FORTH Model," 1981 Rochester Forth Conference, May 1981, pp. 221-228.

[Col83] Colburn, D., "A Better Algorithm for Sieve Benchmarks in Forth," Dr. Dobb's Journal, No. 83, Sept 1983, pp. 9-10.

[Col85] Cole, B.C., "A Pride of New CPUs Runs High-Level Languages," Electronics, Nov 25, 1985, pp. 58-60.

[Con85] Cone, L.L., "Skycam: An Aerial Robotic Camera System," Byte, Vol. 10, No. 10, Oct 1985, pp. 122-132.

[DEF84] Dixon R.D., Edmonson W.M., Franklin R.D., and Sloan J.L., "Extensions of Forth for Functional Programming," 1984 Rochester Forth Conference, June 1984, pp. 228-233.

[DeM77] Dewar, R.K., and McCann, A.P. "MACRO SPITBOL - A SNOBOL 4 Compiler," Software - Practice & Experience, Vol. 7, No. 1, Jan. 1977, pp. 95-113.

[Dew75] Dewar, R.B.K., "Indirect Threaded Code," Comm. ACM, Vol. 18, No. 6, June 1975, pp. 330-331.

[Dob85] Dobbins, R.W., "Extending the Multi-Tasker: Mailboxes" FORTH Dimensions, Vol. 7, No. 4, Nov/Dec 1985, pp. 25-27.

[Dow83] Dowling, T., "The QUAN Concept Expanded," The Journal of Forth Application and Research, Vol. 1, No. 2, 1983, pp. 69-71.

[Dre86] Dress, W.B., "A Forth Implementation of the Heap Data Structure," The Journal of Forth Application and Research, Vol. 3, No. 3, 1986, pp. 39-49.

[DuI84] Duff, C.B., and Iverson, N.D., "Forth Meets Smalltalk," The Journal of Forth Application and Research, Vol. 2, No. 3, 1984, pp. 7-26.

[Dum83] "A Robotic Application for Contamination Free Assembly," The Journal of Forth Application and Research, Vol. 1, No. 1, 1983, pp. 33-41.

[Dum84] Dumse, R., "The R65F11 and F68K Single-Chip FORTH Computers," The Journal of Forth Application and Research, Vol. 2, No. 1, 1984, pp. 11-21.

[DuT84] Duncan, R., and Tracy, M., "The FVG Standard Floating-Point Extension," Dr. Dobb's Journal, No. 95, Sept 1984, pp. 110-115.

[EMS81] Elmore, M.J., and Miller, D., and Schwabe, J., "The impact of 16-bit microprocessors on software development tools," Computer Design, June 1981, pp. 111-115.

[EpG85] Epstein, A., and Gilliatt, C.H., "The MAGIC/L Programming Language," The Journal of Forth Application and Research, Vol. 3, No. 2, 1985, pp. 9-21.

[Far83] Farden, D.C., "A Non-Conventional Implementation of FORTH for the Z80 with CP/M," 1983 Rochester Forth Conference, June 1983, pp. 165-174.

[FeG82] Feuer, A., and Gehanin, N., Comparing and Assessing Programming Languages, Prentice-Hall, Englewood Cliffs, N.J., 1984.

[Fei81] Feierback, G., "Forth - the language of Machine Independence," Computer Design, June 1981, pp. 117-121.

[Feu85] Feucht, D.L., "Improved Forth-83 DO LOOP," FORTH Dimensions, Vol. 7, No. 3, Sept/Oct 1985, pp. 28-29.

[Fil83] Filipski, A.J., "A Forth Word Editor Based on Calling Relationships," SOFTFAIR Proceedings, July 1983, pp. 251-255.

[For84] Forsley, L., "A Review of RISC and Forth Machine Literature," The Journal of Forth Application and Research, Vol. 2, No. 1, 1984, pp. 85-88.

[FST83] FORTH Standards Team, FORTH-83 STANDARD, Mountain View Press, Mountain View, CA., Aug. 1983.

[Gla83] Glass, H., "Towards a More Writeable Forth Syntax," 1983 Rochester Forth Conference, June 1983, pp. 125-132.

[GMB85] Golden, J., and Moore, C., and Brodie, L., "Fast processor chip takes its instructions directly from Forth," Electronic Design, Mar 21, 1985, pp. 127-138.

[GoR83] Goldberg, A., and Robson, D., Smalltalk-80 : The language and its implementation, Addison-Wesley, Reading, Mass., 1983.

[GrD84] Dixon, R.D., and Grewe, R.C., "A Forth Machine for the S-100 System," The Journal of Forth Application and Research, Vol. 2, No. 1, 1984, pp. 23-32.

[Gre84a] Greene, R.L., "Faster FORTH," Byte, Vol. 9, No. 6, June 1984, pp. 127-129, 418-424.

[Gre84b] Greene, R.L., "A Proposal for Implementing Local Words in Forth," The Journal of Forth Application and Research, Vol. 2, No. 4, 1984, pp. 39-42.

[HaP85] Hart, J.R., and Perona, J., "Local Variables," The Journal of Forth Application and Research (1985 Rochester Forth Conference), Vol. 3, No. 2, 1985, pp. 159-162.

[Har80] Harris, K., "FORTH Extensibility: Or, How to Write a Compiler in Twenty-Five Words or Less," Byte, Vol. 5, No. 8, Aug 1980, pp. 164-184.

[Har81] Harris, K.R., "Transportable Control Structures," 1981 Rochester Forth Conference, May 1981, pp. 97-107.

[Hay85] Hayes, J., "Another Forth-83 LEAVE," FORTH Dimensions, Vol. 7, No. 1, May/June 1985, pp. 36-37.

[Hel81] Helmers, P., "Alternate Parameter Stacks," 1981 Rochester Forth Conference, May 1981, pp. 279-281.

[Hel82] Helmers, P.H., "A Parse-tial Solution to RPN," 1982 Rochester Forth Conference, May 1982, pp. 251-257.

[HoF83] Hofert, D.K., and Forsley, L.P., "Generic Operators for Use with Infix, Postfix, and Prefix Notations (Abstract)," 1983 Rochester Forth Conference, June 1983, p. 155.

[Hol84] Holmes, T., "Token Threaded Forth and the Extended Address Space," The Journal of Forth Application and Research, Vol. 2, No. 4, 1984, pp. 21-26.

[Jam80] James, J.S., "What is FORTH? A Tutorial Introduction," Byte, Vol. 5, No. 8, Aug 1980, pp. 100-126

[Jam85a] James, J.S., "Forth Component Libraries," FORTH Dimensions, Vol. 7, No. 4, Nov/Dec 1985, pp. 38-40.

[Jam85b] James, J.S., "What's Wrong with Forth?," The Journal of Forth Application and Research (1985 Rochester Forth Conference), Vol. 3, No. 2, 1985, pp. 167-169.

[JoB83] Johnson, H.E., and Bonissone, P.P., "Expert System for Diesel Electric Locomotive Repair," The Journal of Forth Application and Research, Vol. 1, No. 1, 1983, pp. 7-16.

[Jon86] Jonak, J.E., "Experience with a FORTH-like language," ACM SIGPLAN, Vol. 21, No. 2, Feb 1986, pp. 27-36.

[Joo82a] Joosten, R., "Parameters," 1982 Rochester Forth Conference, May 1982, pp. 276-281.

[Joo82b] Joosten, R., "Separate Headers," 1982 Rochester Forth Conference, May 1982, pp. 281-284.

[Kog82] Kogge, P.M., "An Architectural Trail to Threaded-Code Systems," IEEE Computer, Mar 82, pp. 22-32.

[KoN84a] Korteweg, S., and Nieuwenhuyzen, H., "Stack Usage and Parameter Passing," The Journal of Forth Application and Research, Vol. 2, No. 3, 1984, pp. 27-50.

[KoN84b] Korteweg, S., and Nieuwenhuyzen, H., "Separate Headers," The Journal of Forth Application and Research, Vol. 2, No. 2, 1984, pp. 43-52.

[KoN84c] Korteweg, S., and Nieuwenhuyzen, H., "Binary Search," The Journal of Forth Application and Research, Vol. 2, No. 4, 1984, pp. 43-49.

[KRA81] Komusin, B., Rust, T., and Armstrong, D., "Virtual Machine Group Report," 1981 Rochester Forth Conference, May 1981, pp. 59-63.

[LeM83] Laeary, R.C., and McClimens, D.P., "Message Passing with Queues," The Journal of Forth Application and Research, Vol. 1, No. 2, 1983, pp. 17-33.

[Lew83] Lewis, S.M., "VERIFY - A Useful FORTH Programming Tool," 1983 Rochester Forth Conference, June 1983, pp. 239-249.

[Lin84] Lindbergh, D.J., "Development of OMNITERM 2, A MS-DOS Communications Program in MMSFORTH," 1984 Rochester Forth Conference, June 1984, pp. 199-200.

[LiW85] Linden, F., and Wilson, I., "An Interactive Debugging Environment," IEEE Micro, Aug 1985, pp. 18-31.

[Loe81] Loeliger, R.G., Threaded Interpretive Languages, Byte Books, Peterborough, N.H., 1981.

[LyJ85] Lynk, E.T., and Johnson, H.E., "Forth-Based Software for Real-Time Control of a Mechanically-Scanned Ultrasonic Imaging System," The Journal of Forth Application and Research, Vol. 3, No. 1, 1985, pp. 7-24.

[Lyo85] Lyons, G.B., "Stack Frames and Local Variables," The Journal of Forth Application and Research, Vol. 3, No. 1, 1985, pp. 43-52.

[McB84] McBride, M., "polyFORTH on the NCR/32," The Journal of Forth Application and Research, Vol. 2, No. 1, 1984, pp. 77-84.

[McG84] McGuire, T.E., "Kitt Peak Multi-Tasking FORTH-11," The Journal of Forth Application and Research, Vol. 2, No. 2, 1984, pp. 57-67.

[Mei79] Meinzer, K., "IPS - An Unorthodox High Level Language," Byte, Vol. 4, No. 1, Jan. 1979, pp. 146-159.

[Moo74] Moore, C.H., "FORTH: A new way to program a minicomputer," Astronomical Astrophysics Supplement. 15, pp. 497-511.

[Moo80] Moore, C.H., "The Evolution of FORTH - An Unusual Language," Byte, Vol. 5, No. 8, Aug 1980, pp. 76-92.

[Mot82] Motorola, MC68000 16-Bit Microprocessor User's Manual, Third Edition, Prentice-Hall, Englewood Cliffs, N.J., 1982.

[Mur85] Murphy, R.W., "Under the Hood of a Superchip: The NOVIX FORTH Engine," The Journal of Forth Application and Research (1985 Rochester Forth Conference), Vol. 3, No. 2, 1985, pp. 185-188.

[Oze82] Ozegovic, J., "The FORTH Solution of the Hardware Interrupts Handler," 1982 Rochester Forth Conference, May 1982, pp. 259-263.

[PaP82] Patterson, D.A. and Piepho, R.S., "RISC Assessment: A High-Level Language Experiment," Proceedings of the 9th Annual Symposium on Computer Architecture, IEEE, New York, NY, 1982, pp. 26-29.

[Pay84] Payne, W.H., "ROMable Forth applications code development," IEEE Software, Vol. 1, No. 4, Oct 1984, pp. 100-102.

[Per83] Perry, M.A., "A 68000 Forth Assembler," Dr. Dobb's Journal, No. 83, Sept 1983, pp. 28-42.

[PBW78] Phillips, J.B., Burke, M.F., Wilson, G.S., "Threaded-code for Laboratory Computers," Software - Practice & Experience, Vol. 8, No. 1, Jan. 1978, pp. 257-263

[Pou84] Poutain, D., "POP and SNAP," Byte, Vol. 9, No. 11, Oct 1984, pp. 381-388.

[Pou85a] Poulo, R.J., "Language Tradeoffs for Real-Time Programming Applications," 1985 Rochester Forth Conference, June 1985, pp. 210-215.

[Pou85b] Poutain, D., "Multitasking FORTH," Byte, Vol. 10, No. 3, Mar 1985, pp. 363-371.

[Pou86] Poutain, D., "Object Oriented Extensions to Forth," The Journal of Forth Application and Research, Vol. 3, No. 3, 1986, pp. 51-73.

[RaM76] Rather, E.D., and Moore, C.H., "The FORTH Approach to Operating Systems," Proc. ACM '76, Oct. 1976, pp. 233-240.

[Rat85] Rather, E.D., "Fifteen Programmers, 400 Computers, 36,000 Sensors and FORTH," The Journal of Forth Application and Research, Vol. 3, No. 2, 1985, pp. 46-74.

[Ree82] Reece, P., "A Disk Operating System for FORTH," Byte, Vol. 7, No. 4, Apr 1982, pp. 322-358.

[RiW80] Ritter, T., and Walker, G. "Varieties of Threaded Code for Language Implementation," Byte, Vol. 5, No. 9, Sept 1980, pp. 206-227.

[RMB85] Robinson, H., Morse, P.D., and Bowhill, S.A., "Design of a Forth Target Compiler," Dr. Dobb's Journal, No. 108, Oct 1985, pp.52-92.

[Rus81] Rust, T., "A File Management System," 1981 Rochester Forth Conference, May 1981, pp. 183-186.

[SaB83] Sachs, J.M., and Burns, S.K., "STOIC, an Interactive Programming System for Dedicated Computing," Software - Practice & Experience, Vol. 13, No. 1, Jan 1983, pp. 1-16.

[Seb85] Sebok, W.L., "Combining Forth and the Rest of the Computer World: Dynamic Loading of Subroutines Written in Other Languages and Their Use as Forth Words," The Journal of Forth Application and Research (1985 Rochester Forth Conference), Vol. 3, No. 2, 1985, pp. 211-214.

[Sho83] Shooman, M.L., Software Engineering, McGraw-Hill, N.Y., 1983.

[SHS81] Sussman, G., Holloway, G., Steel, G, and Bell, A., "Scheme-79: Lisp on a Chip," IEEE Computer, July 1981, pp. 8-21.

[Sim85] Simard, D., "Another Subroutine Technique," FORTH Dimensions, Vol. 7, No. 2, July/Aug 1985, pp. 25-26.

[Sla83] Slater, D., "A State Space Approach to Robotics," The Journal of Forth Application and Research, Vol. 1, No. 1, 1983, pp. 17-22.

[Sol83] Solntseff, N., "An Instruction Set Architecture for Abstract Forth Machines," 1983 Rochester Forth Conference, June 1983, pp. 175-183.

[Sol84] Solley, E.L., "SPHERE: An In-Circuit Development System with a Forth Heritage," 1984 Rochester Forth Conference, June 1984, pp. 25-31.

[SoR84a] Solntseff, N., and Russell, J.W., "An Approach to a Machine-Independent Forth Model," 1984 Rochester Forth Conference, June 1984, pp. 121-139.

[SoR84b] Solntseff, N., and Russell, J.W., "A Break Point Utility for Forth," 1984 Rochester Forth Conference, June 1984, pp. 176-187.

[SPB82] Sansonnet, J.P, Percebois, C., Botella, D., and Perez, J., "A Hardware Support for Interactive Programming Environments," Integrated Interactive Computing Systems, ECICS Proceedings, Sept 1982, pp. 125-133.

[Sto85] Stoddart, B., "Readable and Efficient Parameter Acess via Argument Records," The Journal of Forth Application and Research, Vol. 3, No. 1, 1985, pp. 63-82.

[Tin85] Ting, C.H., "F83 Word Usage Statistics," FORTH Dimensions, Vol. 7, No. 4, Nov/Dec 1985, pp. 12-15.

[Tod81] Tody, D., "Moving Forth into the Eighties," 1981 Rochester Forth Conference, May 1981, pp. 331-364.

[UBF84] Ungar, D., Blau, R., Foley, D., and Patterson, D., "Architecture of SOAR: Smalltalk on a RISC," Proc. Eleventh International Symposium on Computer Architecture, 1984, pp. 188-197.

[Uri84] Urieli, I., "Hello, A REPTIL I Am," 1984 Rochester Forth Conference, June 1984, pp. 236-243.

[Uri85] Urieli, I., "REvised REcursive AND? 'REPTIL :IS," The Journal of Forth Application and Research (1985 Rochester Forth Conference), Vol. 3, No. 2, 1985, pp. 229-231.

[VaS84] Vaughan, J.C., and Smith, R.L., "The Design of a Forth Computer," The Journal of Forth Application and Research, Vol. 2, No. 1, 1984, pp. 49-64.

[Vic84] Vickery, C., "QFORTH: A Multitasking FORTH Language Processor," The Journal of Forth Application and Research, Vol. 2, No. 1, 1984, pp. 65-75.

[Wal85a] Wallen, L., "Rather sees Novix collaboration as Forth booster," Electronics Week, May 13, 1985, p. 46.

[Wal85b] Wallich, P., "Towards simpler, faster computers," IEEE Spectrum, Vol. 22, No. 8, Aug 1985, pp. 38-45.

[Wil80] Williams, G., "Threads of a FORTH Tapestry," Byte, Vol. 5, No. 8, Aug 1980, pp. 6-10 & 128-134.

[WiP84] Winterle, R.G., and Poehlman, W.F.S., "Asynchronous Words for Forth," 1984 Rochester Forth Conference, June 1984, pp. 32-41.

[Wol85] Wolfe, A., "Real-Time OP Systems Join the Computing Mainstream," Electronics, Aug 19, 1985, p. 46.

[Yng85] Yngve, V.H., "Synonyms and Macros," FORTH Dimensions, Vol. 7, No. 3, Sept/Oct 1985, pp.11-18.