4

Token Threaded Forth and the Extended Address Space

Terry Holmes

Information Appliance, Inc.
530 University Avenue
Palo Alto, CA 94301

Abstract

This paper outlines a proposal for token threaded code to extend FORTH beyond its traditional 64K boundary while maintaining the compactness of 16-bit FORTH systems. Comparisons are made with the efficiencies and functional characteristics of indirect- and direct-threaded extended-address-space implementations on the MC68000 microprocessor.

Introduction

The intention of this paper is to examine ways in which FORTH systems may effectively align themselves with the capabilities of current microprocessors – in particular, here, the MC68000. The technical terminology follows Ritter and Walker (Ritter, 1980) in identifying three of their four varieties of threaded code structure. These are direct threaded code (DTC) consisting of lists of addresses of machine code; indirect threaded code (ITC) consisting of lists of addresses of machine code; and token or table threaded code (TTC) consisting of lists of values representing offsets into a table of pointers. Advantages and disadvantages of token threading will be discussed.

The Tyranny of the Address Space

There are a number of characteristics of conventional threaded code implementations (both DTC and ITC) which stem directly from the use of absolute addresses as compilation tokens. Compiled code consisting of only addresses must execute where it was complied, and since the addresses of FORTH primitives vary with the machine used, it probably will run only on the machine on which it was compiled. The penalty for choosing an address-dependent rather than tokenized approach to compilation is the necessity of a class of rather complex programs called target (or meta-) compilers to generate code not intended for execution in the immediate memory or processor environment.

Using a DTC or ITC model on the MC68000 processor requires either compiling 32-bit addresses, or compiling 16-bit addresses but restricting code to a 64K partition. The alternate scheme proposed here involves compiling 16-bit numbers and storing 32-bit addresses in a lookup table for use at execution time. This means four extra bytes are required with each word defined. Assuming each dictionary entry requires 20-30 bytes of storage, this threading technique would be 10-15% less memory efficient than conventional 16-bit FORTH systems, but 40-45% more efficient than 32-bit ITC/DTC systems, which would require 40-60 bytes of storage per entry. Thus, 16-bit token systems, like 32-bit ITC/DTC systems, have the capacity to use the extended MC68000 address range, but are less wasteful of memory resources.

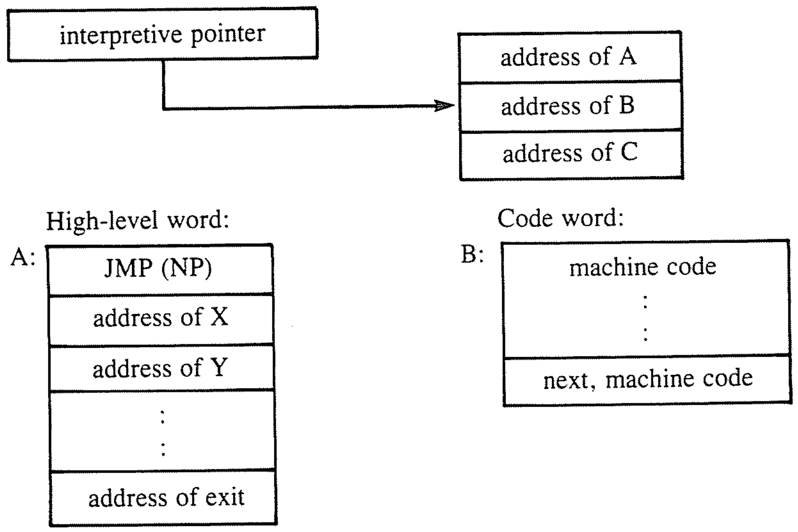
Implementation Particulars

The most straightforward approach to the use of the MC68000 address space involves using 32-bit wide stacks. This entails a small speed penalty while allowing the use of full 32-bit addresses as stack arguments.

The following threading comparison considers the inner interpreter (NEXT) and the nesting and unnesting routines (**:** and **;**) as used for extended-range DTC, ITC and TTC implementations. The interpretive pointer (IP), return stack pointer (RP), parameter stack pointer (SP), and nesting routine pointer (NP) are all assumed to be permanently assigned address registers.

The DTC implementation requires that each word definition begins with executable code (Figure 1). A high level FORTH word begins with a transfer to the nesting routine with a cost of 8 cycles, two bytes of code, and the permanent allocation of an address register (NP) dedicated to holding the address of the nesting routine. The execution of code words will require the accesssing of only one address, since the compiled value leads directly to executable code.

FIGURE 1: DTC diagram



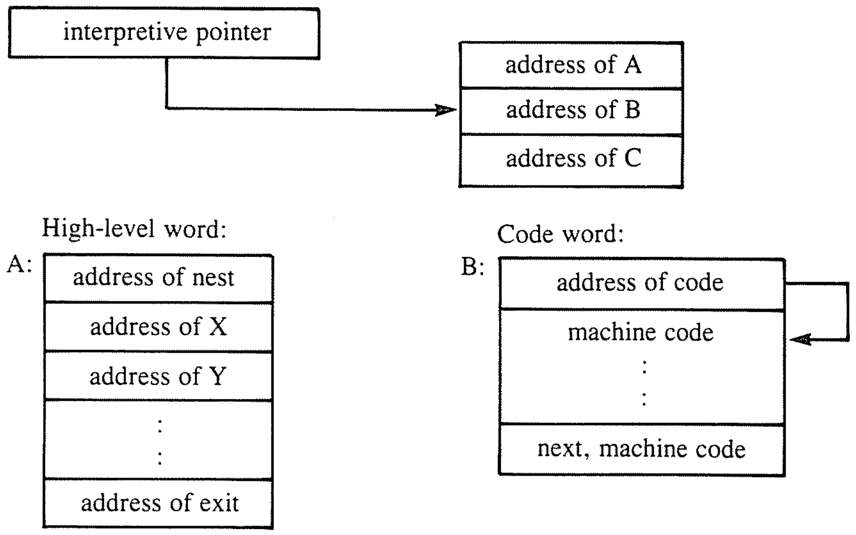
```
Code for DTC NEXT:
MOVE.L (IP)+,A0                      12    ; Get 32-bit execution address.
JMP (A0)                              8    ; Execute word or transfer to other
                                             routine.
                              total  20  cycles

DTC nesting routine:
JMP (NP)                              8    ; Transfer to nesting routine.

....
MOVE.L IP,-(RP)                      16    ; SaveIP to return stack.
LEA 2(A0),IP                          8    ; New Ip follows 2-byte jump to nest.
<NEXT>                               20    ; Ready to interpret new word.
                              total  52  cycles
```

DTC unnesting routine:

```
MOVE.L (RP)+,IP              12   ;  Recover old IP.
<NEXT>                       20   ;
                      total  32  cycles
```

The ITC implementation is noticeably slower as two memory fetches are required for code words to execute (Figure 2). To its advantage it does not require a transfer to be emplaced at the beginning of high level words, since that function is filled by a pointer located there.

FIGURE 2: ITC diagram



Code for ITC NEXT:

```
MOVE.L (IP)+,A1              12   ;  Get pointer to pointer.
MOVE.L (A1)+,A0              12   ;  Get address of code to execute.
JMP (A0)                      8   ;  Execute machine code.
                      total  32  cycles
```

ITC nesting routine:

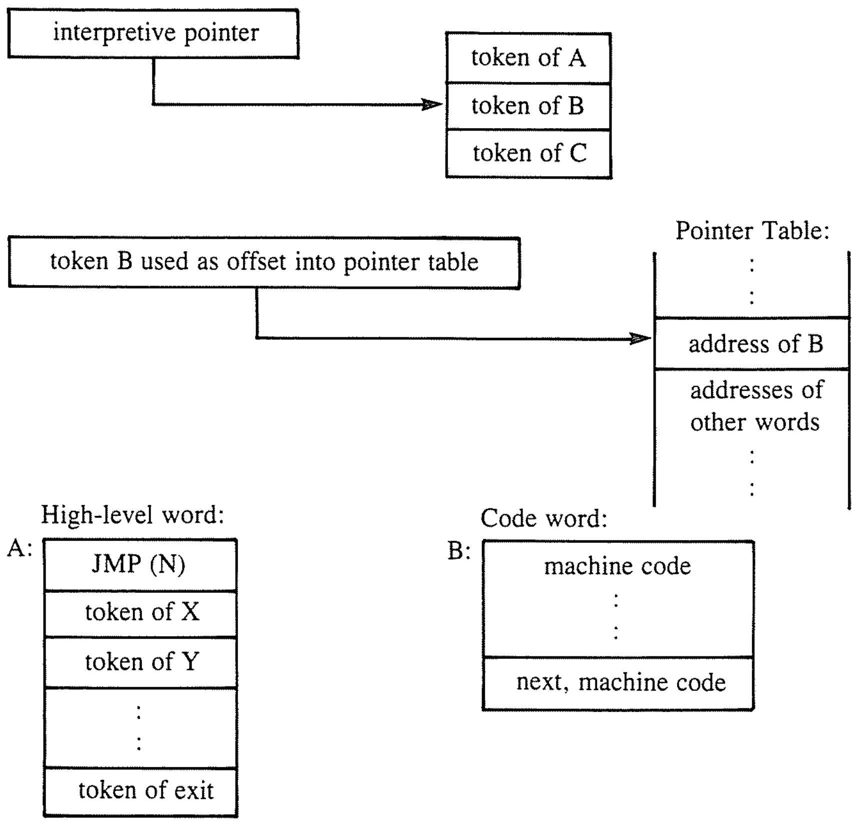
```
MOVE.L IP,-(RP)              16   ;  Save IP.
MOVE.L A1,IP                  4   ;  A1 is “work” register from  NEXT .
<NEXT>                       32   ;
                      total  52  cycles
```

ITC unnesting routine:

```
MOVE.L (RP)+,IP              12   ;  Old IP.
<NEXT>                       32   ;
                      total  44  cycles
```

The TTC implementation (a variant of the Ritter and Walker TTC) resembles the ITC version in that two memory accesses are required for code words, but high level words here require an emplaced transfer to nesting code in the same manner as DTC (Figure 3). First a 16-bit token is read from the interpreting word. Then a 32-bit execution address is fetched from the lookup table and control transfers to that location. Two versions are provided – one that will handle 16,384 tokens and a somewhat slower one that will handle up to 65,536 separate tokens.

FIGURE 3: TTC diagram



Code for TTC NEXT (16K version):

```
    MOVE (IP)+,D7                 8    ;  Get 16-bit token.
    MOVE.L D7,A1                  4    ;  D7 has offset to table in high bits.
    MOVE.L (A1),A0               12    ;  Get 32-bit address.
    JMP (A0)                      8    ;  Execute.
                        total    32  cycles
```

Code for TTC NEXT (64K version):

```
    MOVE (IP)+,D7                 8    ;  Get token.
    MOVE.L D7,A1                  4    ;  Get table address.
    ADD.L A1,A1                   6    ;  Shift left twice, address*4
    ADD.L A1,A1                   6    ;
    MOVE.L (A1), A0              12    ;  Get 32-bit execution address.
    JMP (A0)                      8    ;
                        total    44  cycles
```

```
TTC nesting routine:
    JMP (NP)                          8    ;  Transfer to nesting code.
    . . .
    MOVE.L IP,-(RP)                  16    ;  Save IP.
    LEA 2(A0),IP                      8    ;  New IP follows jump code.
    <NEXT>                           32    ;  Timing estimate uses smaller token
                                              space.
                              total  64  cycles

TTC unnesting routine:
    MOVE.L (RP)+,IP                  12    ;  Get old IP.
    <NEXT>                           32    ;
                              total  44  cycles
```

One comparison which the preceding analysis suggests follows from the execution times for the respective threading types. An empty high level word in DTC costs 84 cycles; in ITC costs 96 cycles; and in TTC costs 108 cycles. This rough estimation suggests that ITC may be 10% slower and TTC 20% slower than DTC. Taking into consideration that a large percentage of FORTH code consists of short stack and arithmetic operators, it may be that DTC is actually more efficient than this would indicate, and that TTC and ITC are approximately of the same efficiency.

The particular version of token FORTH demonstrated here is not machine independent because the prologue routine in high level words is represented by a particular MC68000 machine instruction. A third level of indirection (a pointer instead of the prologue routine) would be required for machine independence (Kogge, p.24). However, this would still require that the nesting routine be at a fixed location on any system that would use this code. A fourth level of indirection would yield truly address and machine independence (emplacing a token for nesting) at the cost of a significant sacrifice of execution speed (Ritter and Walker, p. 212).

An issue that applies to the present DTC and TTC implementations is the allocation of the address registers. There are potentially many candidates for the use of these limited resources, such as stack pointers and prologue branches to `VARIABLE` , `CONSTANT` , etc. One solution is to place all of the internal routines in a contiguous area and allow the other prologue code pieces to be of the form: " `JMPd(NP)` " where "d" is the number of bytes from the beginning of the common code area to the beginning of the code for this particular routine. These instructions require four instead of two bytes per usage.

Difficulties with Tokens

The chief disadvantage of token systems is the fragmentation of the dictionary into a pointer table and a code portion, and the memory management problems that arise from this. In fact, it is probably advantageous to make a third piece out of the dictionary headers. Strategies to accommodate a table lookup scheme in mixed RAM/ROM systems may require complex decisions to be made in `NEXT` . One such approach in RAM-based systems is to approximate the functional characteristics of conventional systems by having contiguous table and code areas grow from low memory. As a page of tokens (256 bytes or 64 pointers) is exhausted, another page can be allocated by moving all code forward in memory, and appropriately updating the pointer (token) table. Forgetting words from the dictionary would, conversely, contract the table and code areas. Clearly, this scheme is more complex than with conventional FORTH systems.

Further Comments on Tokens

A primary distinguishing feature of token systems is transportability and relocatability inherent at the object code level. This is enforced by a discipline which disallows absolute addresses in the code space – all addresses being segregated into the lookup table. Moving code to a different memory environment involves adding a fixed displacement to all addresses in the lookup table. Unneeded words can be purged from the code space by moving code down over deleted material and updating the lookup table as required. Even tokens may be recycled by placing unused tokens on a list of free tokens. Indiscriminate use of this capability could lead to the removal of words called by other routines which are still active. Whether the system should protect the user from making such mistakes is an open implementation issue facing the designer in this kind of mutable programming environment.

Traditionally, FORTH system designers have struggled with two opposing notions. One is to extend FORTH with special functions and operators to assist words from which all of FORTH can be generated. Commercially available systems often have 5000-8000 bytes (300-500 FORTH words) as a basic configuration. A designer working within the exisiting paradigm must pursue a course toward a small but functional nucleus, avoiding special temporary constructs which may have little future use but will be locked into the address space of the system. A token system, on the contrary, need not be so constrained if it has been designed to allow the redefinition and removal of words.

Experience to Date

The FORTH system described in this paper has not been implemented exactly as suggested here, but I offer the following observations from having implemented a similar TTC system. First, the process of target compilation conformed closely to the normal compilation of FORTH words, so that the target compiler needed to be only about 6 screens long. Most of the remaining difficulty with target compilation resulted from cross-talk between vocabularies – the system performing the target compilation used words whose names also appeared in the target dictionary under construction. Second, this experimental system allowed only one instance of a name to appear in a vocabulary. Subsequent attempts to define the word would actually alter the operation of the original word by updating the entry in the token table to point to the most recent code. With the redefining technique modified in this way, existing words are quickly changed and tested without recompiling large amounts of source code. Forward referencing can be done rather easily in this system by creating a word without a code section. When the word is redefined later, the code portion of the word is defined.

This experimental system followed the FORTH-83 Standard only in the operation on most stack and arithmetic primitives. The 32-bit stack as well as vocabulary and dictionary structures were of course incompatible with the Standard.

Conclusions

Three strategies for expanding FORTH out of its 64K partition have been considered. Direct and indirect threaded code requires the compilation of 32-bit addresses – DTC being the choice where execution speed is foremost. Token threaded systems will probably be the choice for integrated packages requiring large amounts of code due to the great compaction achieved. Also, token systems offer greater flexibility and portability of object code.

References

Barnhart, Joe. "Forth and the Motorola 68000," *Dr. Dobb's Journal,* Vol. 7 No. 9, September 1983, p. 34.

Kogge, Peter M., "An Architectural Trail to Threaded-Code Systems," *IEEE Computer,* Vol. 15 No.3, March 1982, p. 22.

Ritter, Terry, and Walker, Gregory. "Varieties of Threaded Code for Language Implementation," *BYTE,* Vol. 5 No. 9, September, 1980.

Manuscript received August 1984.