Threaded Code Designs for Forth Interpreters

P. Joseph Hong

Introduction

Forth interpreters can utilize several techniques for implementing threaded code. We will classify these techniques to better understand the mechanisms underlying "threaded interpretive languages", or TILs. One basic assumption we will make is that the TIL is implemented on a typical microprocessor (which is usually the case).

The following are the elements of any threaded interpretive language. These must be designed together to make the interpreter work.

TIL Instructions

In a TIL, an instruction is also called a *word* (which is not really the same as a 16-bit memory word). In an instruction sequence, each instruction word could be represented by a token, a pointer, or some other entity. For convenience, we shall call this entity a *thread-code*. Each thread-code corresponds to one instruction word, and there are two types of these words:

1. *Primitives:* these are words (instructions) representing the small, basic actions that define the operations of the language. Primitives are the first level of execution for the TIL, and are defined using the machine language of the host processor.

2. *Secondaries:* a secondary word is defined as a sequence of primitives. To execute a secondary, each primitive in its definition is executed in turn. This is the equivalent of a microprocessor program calling a subroutine. A secondary word may also contain other secondaries in its definition. In this case they are just like ordinary subroutines which call other subroutines.

The sequence of instructions in a secondary's definition is a list of thread-codes, and sometimes we shall call this list a *sub-word list*.

The Interpreter

This is the engine that handles the sequence of instructions. To do this, it needs the following elements:

1. *Instruction Pointer:* keeps track of the current thread-code in a sub-word list. Called IP for short, it is also known as the Program Counter.

2. *Return Stack:* used for storing return addresses for subroutines.

3. *Interpreter Primitives:* the three interpreter primitives are called *next* , *call* , and *return* . These three are also known as *next* , *colon* , and *semi* in some reference literature.

Let's take a simple example to see how these elements interact to form the Threaded Code Interpreter. Take a look at Figure 1. We have a sequence of thread-codes in the processor's memory. The instruction pointer, IP, points to the start of the sequence. To interpret an instruction word, we run *next* . The action of

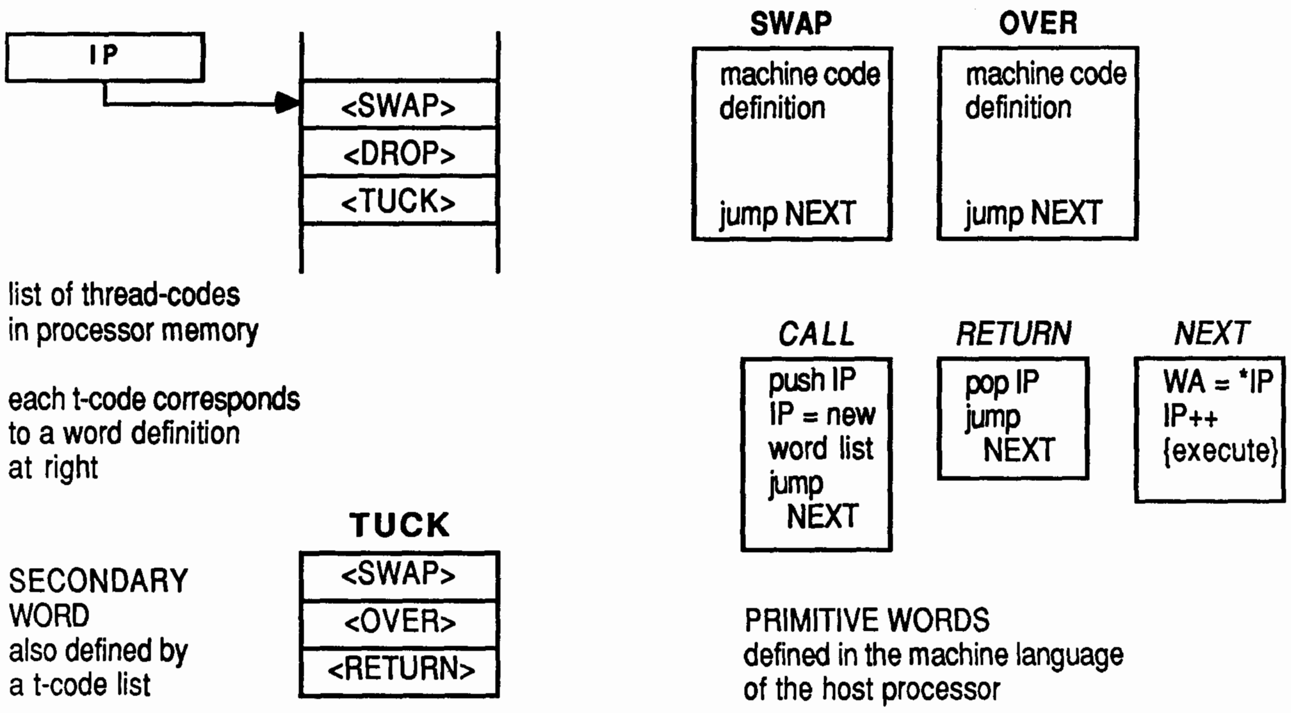
IP
<SWAP>
<DROP>
<TUCK>
list of thread-codes in processor memory
each t-code corresponds to a word definition at right
SWAP
machine code definition
jump NEXT
OVER
machine code definition
jump NEXT
CALL
push IP
IP = new word list
jump NEXT
RETURN
pop IP
jump NEXT
NEXT
WA = *IP
IP++
{execute}
TUCK
<SWAP>
<OVER>
<RETURN>
SECONDARY WORD also defined by a t-code list
PRIMITIVE WORDS defined in the machine language of the host processor

Fig. 1: The threaded code interpreter. The primitives CALL, RETURN, and NEXT perform the execution of the Primitive and Secondary words.

next is:

1. pick up the thread-code indicated by IP,
2. increment IP to point to the next in line,
3. execute the word (from 1) by running its machine-language definition.

In Figure 1, the thread-code for **SWAP** is picked up, and then executed. When the definition for **SWAP** ends, it jumps to *next* itself. The next word is **DROP** , and it is executed just like the first word.

The next word in the sequence, **TUCK** , is a secondary. After the interpreter picks it up, it identifies it as a secondary, so it executes the *call* primitive. The action of *call* is to:

1. push the current IP onto the stack,
2. change IP to point to the sub-word list of the secondary (this destroys the previous information in the IP), then
3. perform *next* , thus executing this sequence just like the original.

All secondaries terminate with the word *return* . The action performed by *return* is:

1. pop the value on top of the stack, and load this into IP, thus restoring the previously saved value of IP, so we are once again pointing to the original sub-word list.
2. perform *next* , continuing execution of the original sub-word list.

General Model of Threaded Code

We will use a general model (Figure 2) to show differences in threaded code designs. The "classical" Forth word contains 4 distinct fields in its structure: the Name Field (NF), the Link Field (LF), the Code Field (CF), and the Parameter Field (PF). The first two are concerned with management of the Forth dictionary, and we will exclude these from our attention. The last two, the CF and the PF, are the fields that are concerned with the implementation of the threading. In our threading model, we have only the Code Field and the Parameter Field (CF and PF). The two are usually found together in memory, and each CF and each PF have unique addresses.

The Code Field indicates the type of the word: primitive or secondary. The Parameter Field can contain several things:

- for a Primitive word, the machine-language definition;
- for a Secondary word, the thread-code sequence of primitives and secondaries in its definition;
- for other types of words, collections of data and words.

The interpreter is responsible for making use of the information in these two fields to execute programs. For instance, when the interpreter picks up a word, it checks its Code Field (CF). For a primitive, it will execute the machine language (ML) definition in the Parameter Field (PF). When that ML definition finishes, it will return to the Interpreter via *next* , as we have seen above.

When the interpreter picks up a secondary, instead of jumping to the PF, it executes *call* , which interprets the secondary word list, and returns to the previous word via *return* .

The problem that the interpreter faces is: how does it distinguish between a primitive and a secondary, and how does it find the proper word definition? The answer to this is in precisely the way a threaded code is represented.

Classifications of Threading Designs

We classify the threading types according to the representations we find in the Code Field and in the Parameter Field. For instance, if a pointer is found in CF, and the thread-codes found in the PF are *pointers* to words, we classify the type as a Pointer-Pointer threaded code.

Pointer-Pointer Type

The Code Field contains a pointer to some executable code, as follows:

Primitives: a pointer to its own PF.

Secondaries: a pointer to the *call* primitive.

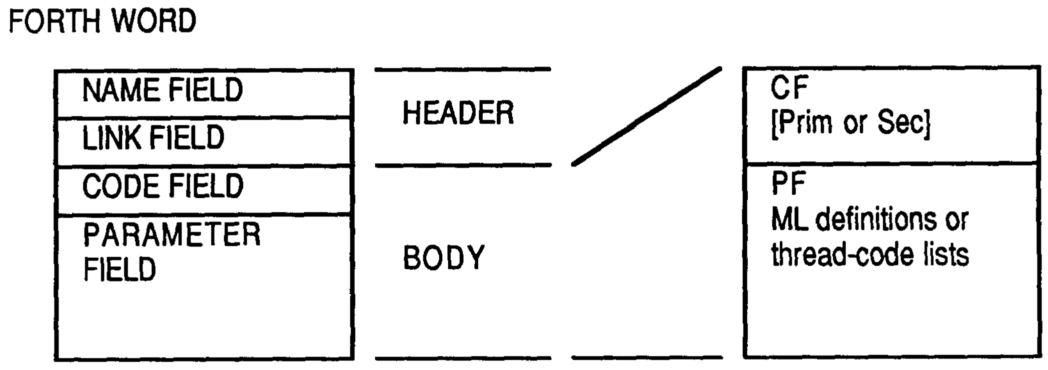

Fig. 2: General model of Threaded Code. The threading is in the Body of the Words.

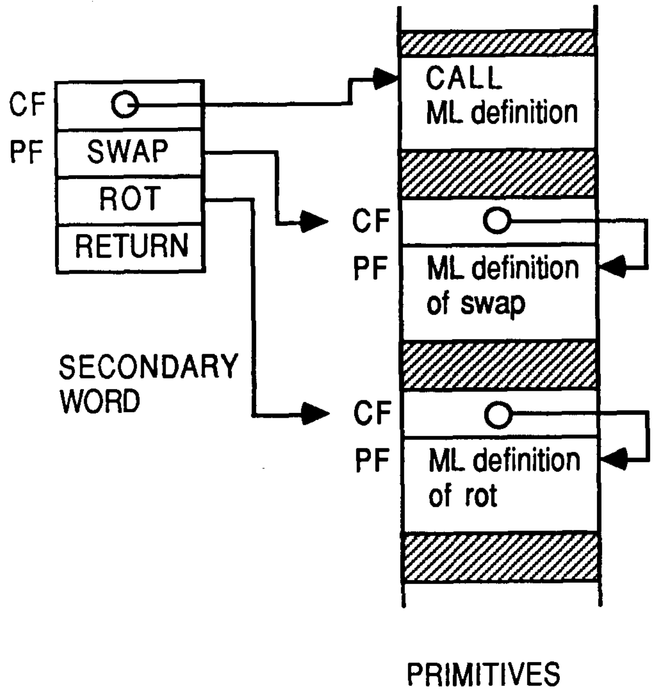

Fig. 3: Pointer-pointer threaded code.

The Parameter Field contains:
Primitives: the machine-language definition of the word, ended with a jump to *next*.
Secondaries: a list of pointers to the sub-words in the definition, terminated with a pointer to *return*.

This interpreter's initial action is simple: it looks at the pointer in CF, and then jumps to that location. Automatically, the proper action is executed. That code then returns control to the interpreter through *next* when it is finished (Figure 3).

The concept behind this is that each thread-code is a pointer ... essentially, the address of the word's definition in the memory space. The list of sub-words in a secondary's definition is a list of addresses, each pointing to the CF of the sub-word in question.

This is one of the most common types of threaded-code. The design is quite flexible, and is easily supported by commonly-available microprocessors.

Token-Pointer Type

The Code Field contains a token signifying if this is a primitive or a secondary.
The Parameter Field contains:
Primitives: the machine-language definition of the word, ended with a jump to *next*.
Secondaries: a list of pointers to the sub-words in the definition, terminated with a pointer to *return*.

The only difference from the above is the contents of CF. The interpreter has to recognize the token in CF. If the token represents a primitive, it jumps straight to the PF. For a secondary, it jumps to *call*. The interpreter must know the location of *call*, as the token does not provide this information (Figure 4).

Token-Token Type

The Code Field contains a token signifying if this is a primitive or a secondary.
The Parameter Field contains:
Primitives: the machine-language definition of the word, ended with a jump to *next*.
Secondaries: a list of tokens of the sub-words in the definition, terminated with the token for *return*.

This is a big change from the token-pointer type. For a primitive, the interpreter's action is the same, but for a secondary, the process is a bit more complicated (Figure 5 on next page).

Instead of pointers to the sub-words, we use tokens. A token is just a simple number, and we use this number as an index into an **array** that contains the actual addresses of all the words defined in the system. When the interpreter picks up a sub-word token, it goes to the array, uses the token as an index, and then gets the address. With the word address in hand, the interpreter runs the *call* primitive to execute the word.

This two-step approach only affects the way sub-words are found. We have replaced the pointers that lead directly to the sub-words. Instead, we have tokens leading to the addresses, which in turn lead to the sub-words.

Clearly, the token-token type is more difficult to implement. The reason they are used is that tokens are designed to take up much less **space** than actual CPU

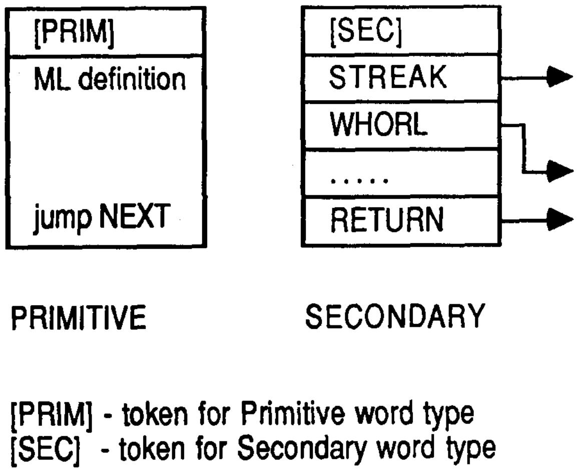
[PRIM]
ML definition
jump NEXT
[SEC]
STREAK
WHORL
.....
RETURN
PRIMITIVE
SECONDARY
[PRIM] - token for Primitive word type
[SEC] - token for Secondary word type

Fig. 4: Token-pointer threaded code.

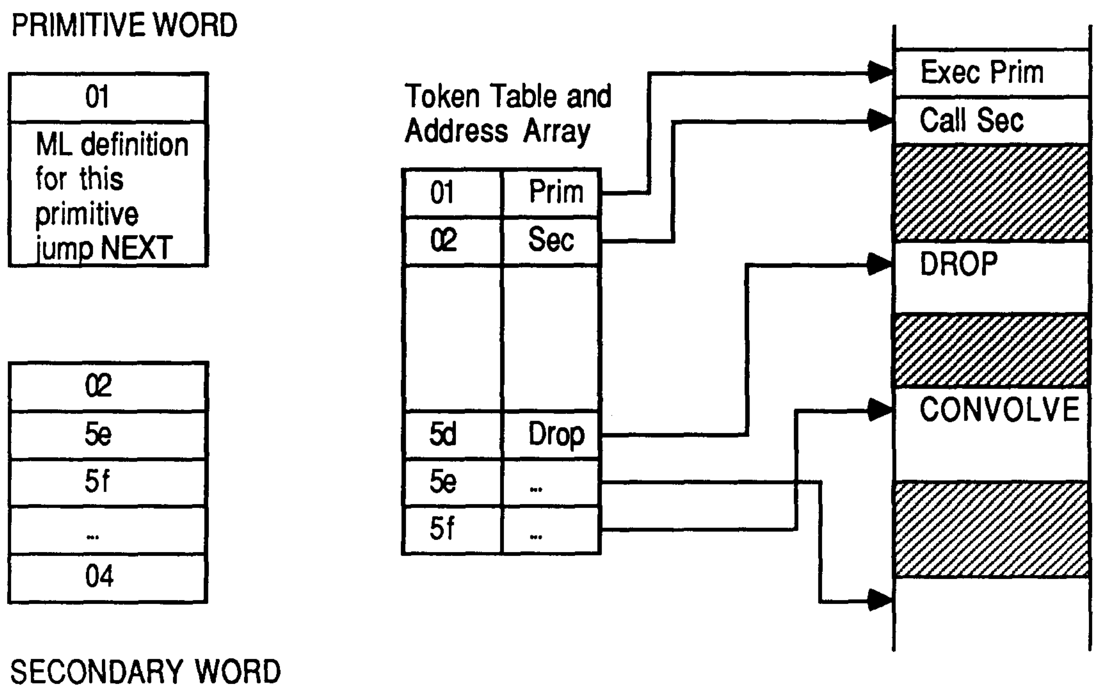

Fig. 5: Token-token threaded code. To execute a word, the token in CF is followed to determine if it is a Primitive or a Secondary. Thread codes in a Secondary index the actual adresses in the Token Table.

addresses. Using a token in a definition leads to memory savings.

For example, a CPU with 24-bit addresses can use 16-bit tokens. This saves 8 bits, or one byte, each time a token is used for a sub-word. The memory saved for the whole system can be substantial. With a 32-bit CPU, tokens are practically necessary. Some memory is taken up by the address array, but the space savings make up for this many times over.

Code-Pointer Type

The Code Field contains some sort of executable code, as follows:

Primitives: the machine-language definition of this word.

Secondaries: machine-language code to get to *call* .

The Parameter Field contains:

Secondaries: a list of pointers to the sub-words in the definition, terminated with a pointer to *return* .

To execute the word, the interpreter only has to jump to the CF. For a primitive, the definition gets executed right away. A secondary needs some machine-language code to jump to *call* . It also has to pass the PF address to *call* , so that the interpreter knows where to find the word list (Figure 6).

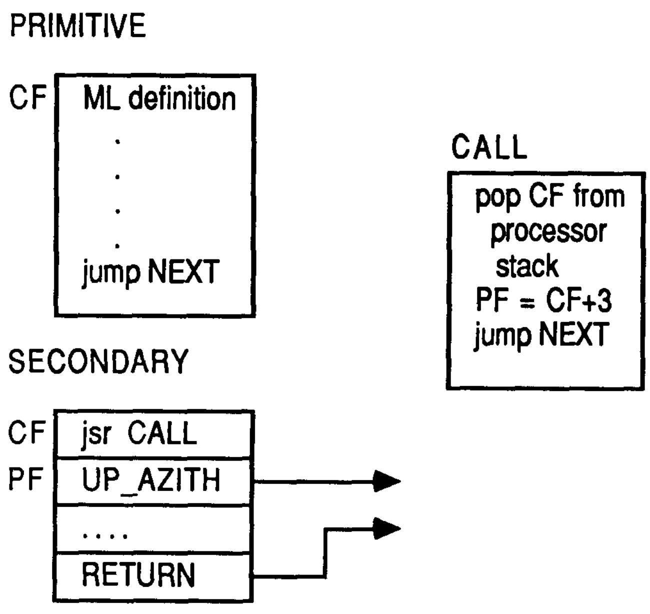

Fig. 6: Code-pointer threaded type.

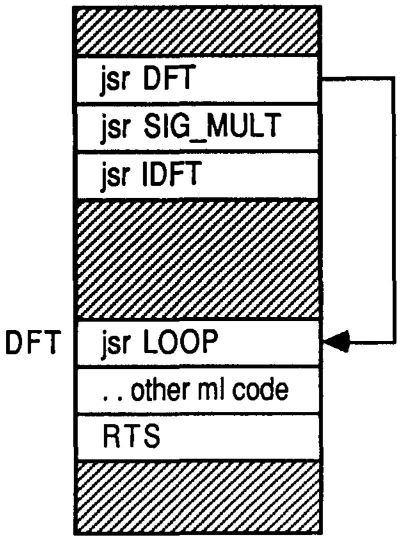

Fig. 7: Subroutine threading.

Subroutine threading

This type of threading does not follow the general model of a threaded code (Figure 7). There are neither CFs nor PFs. The list of sub-words is replaced with a sequence of subroutine calls. Each subroutine can contain its own mix of ML and subroutine calls. This design takes advantage of the microprocessor's own subroutine call and return facility to simulate threading. In effect, there is no interpreter. The CPU itself is the actual processor.

Double-pointers

This is the general case of token-type threading. The pointers found in the sub-word list lead to another set of pointers, sometimes called *vectors* . These *vectors* give the actual addresses of the sub-words. To interpret a word, the system picks up the first pointer to get the vector, and then reads the vector to get the actual address (Figure 8). Like a token type of threading, it is a two-step process.

In contrast to the token type, vectors are, generally, allowed to reside anywhere in the system (instead of only in an address array). This makes the vector system extremely flexible.

What are double-pointers used for? In one Forth system that I have used, the threading took advantage of the segment registers in the 80x86 processors. Machine language definitions and secondary words were stored in one segment, while the link vectors were in another. The double-pointers allowed alteration of a single word's definition without needing a system recompilation ... the definition was compiled, and then only the vector was changed.

Double-pointers might also be used for implementing modularity and relocatability. Only the vectors

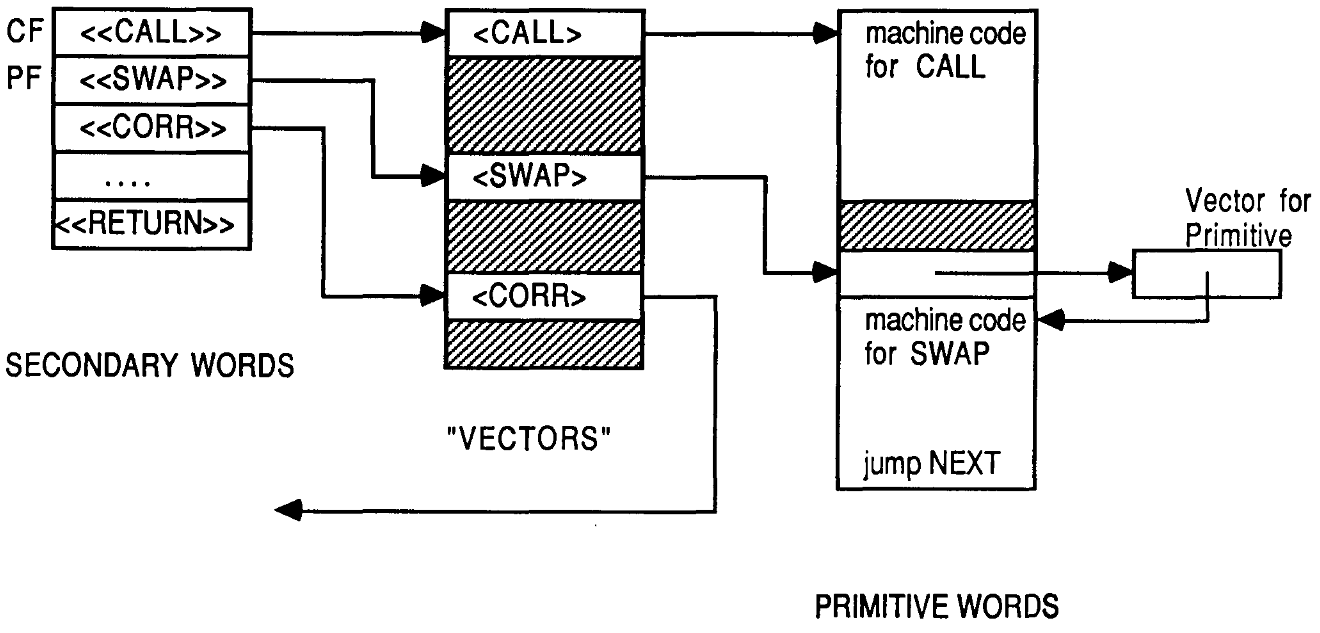

Fig. 8: Double-pointer threaded code. Each thread-code first points to a 'vector', which points to the actual primitive or secondary definitions.

need be changed to link-in "outside" code. Similarly, some programs may require on-the-fly redirection. Vectors can be changed during run-time, changing the routine's action as desired or as needed.

Other types

Combinations of the pointer, token, and double-pointer representations are possible. The discussions above hold for these types as well.

Comparisons and Comments

Obviously, there is great variety in the techniques of code threading. Each type affects the implementation of the interpreter, making it simpler or more complex. We'd like to keep an interpreter simple, since this leads to high-speed execution. Complex threading types lead to lower interpreter performance, but they also have some advantages. One more point of comparison is the amount of memory the threading takes up in a word. These points have to be balanced against each other to evaluate the design.

The simplest interpreter types are the subroutine thread and the code-pointer thread. The subroutine thread does not even need an interpreter at all! This makes it very fast. However, the compilation of this design is different from the general model of a TIL. It also requires more memory: one CALL instruction per subroutine in the definition, which usually takes one additional byte.

The code-pointer case uses a very simple interpreter since it immediately knows where to find the next code to execute. This feature, however, adds a certain amount of overhead per secondary word. For an 8-bit system, three bytes is the usual minimum. Still, the simplicity makes this technique quite fast.

The pointer-pointer type is, again, considered the general type. The design is straight-forward, neat, and flexible. The speed of execution and the space requirements are usually acceptable for general use.

The token and double-pointer threads are more complex. This may lower the performance of the interpreter. However, these designs are usually implemented on more powerful processors, so the complexity does not really penalize execution times. The advantages of space savings and flexibility may be more important. Sometimes, even space saving is not the primary concern. Since the more powerful processors can access greater and greater amounts of memory, flexibility in managing this memory becomes a much more important consideration.

Conclusion

The designs depicted above are all taken from various Forths that have been implemented over the years. For anyone interested in implementing yet another Forth system, these examples may prove very useful. I hope this discussion has enlightened many in the intricacies of threaded code.

References:

(1) Brodie, L. (1981), *Starting Forth*, Prentice-Hall.

(2) Harris Semiconductor (1988), *RTX 2000 Instruction Set*, Harris Corporation, Melbourne, FL.

(3) Harris Semiconductor (1988), *RTX 2000 Real Time Express Microcontroller Data Sheet*, Harris Corporation, Melbourne, FL.

(4) Haydon, G. (1983), *All about Forth: An Annotated Glossary, 2nd Ed.*, Mountain View Press, Mountain View, CA.

(5) Koopman, P. (1989), *Stack Computers: the new wave* , Ellis Horwood Limited, GBR.

(6) Loeliger, B. (1985), *Threaded Interpretive Languages*, Byte Books.

(7) Ritter, T. and G. Walker (1980), "Varieties of Threaded Code for Language Implementation" *Byte*, Sept 1980, pp. 206-227.

(8) Tracy, M. (1990), "Zen for Embedded Systems" *Dr. Dobb's Journal*, Jan. 1990.

Joseph Hong is currently studying for his Master's Degree in Electrical Engineering at the State University of New York at Stony Brook. His interests include computer architecture, signal processing, and real-time systems. He can be reached via pjhong@ic.sunysb.edu and is eager to hear from other TIL enthusiasts.

❑ ❑ ❑

AUTHORS WANTED!

The SIGForth Newsletter is a forum for readers in all walks of life, with varying degrees of knowledge of the Forth language. Although the general tone of the publication is aimed at the professional programmer, we would like to invite authors to contribute regardless of the complexity or degree of sophistication of their intended topics.

Technical papers, case histories, observations, conjecture, news items, letters to the editor, cartoons and other writings are hereby solicited from potential authors. Please send contributions to the editor at the address given on the inside front cover, or via GEnie.