Implementing Forth on the 80386

John E. Lecky
62 Overlook Dr.
So. Burlington, VT 05401

Background

The introduction of a wide range of commercial 32-bit microprocessors by Fairchild, Intel, Motorola, National and others has intensified the drive for native 32-bit Forth implementations. These new processors offer two important advantages as Forth system hosts. First, their 32-bit address spaces provide for the maintenance of enormous Forth dictionaries without resorting to overlaying or segmentation which slow execution and frustrate programmers. Coping with such large dictionaries is increasingly important as Forth is called on to solve more and more complex problems requiring massive programs and data structures.

The second advantage of these new processors is their speed. Forth is often the language of choice for solving real-time control problems. Maintaining execution speed in the face of large control routines and data sets is critical in these applications. The new processors exploit the latest design and fabrication technologies to permit clock rates of up to 30 MHz. At the same time, many instructions execute in fewer clock cycles than were required by earlier processors as more efficient parallel processing is performed internally.

The purpose of this paper is to describe a Forth system that has been developed for one of these processors, the Intel 80386. The 80386, now a key player in the new line of personal computers offered by IBM, will soon enjoy the level of support already enjoyed by some of the earlier entries in the 32-bit market. The 80386 has a "real" mode in which it can execute existing 8086 programs. This feature makes it an attractive system host, as the wide range of existing hardware expansion products already available in the IBM PC-compatible marketplace can be utilized. Similarly, any of the multitude of software products running under MS-DOS[1] or other operating systems can be used as *extensions* to a Forth-PC system, creating a very powerful and heavily supported workstation or system controller at comparatively low cost.

While the basic architecture of the 80386 closely resembles that of the 8086, a larger instruction set, more orthogonal addressing capabilities, and vastly improved instruction execution times make 80386 Forth an essentially new implementation opportunity. In addition, the hostility of the Intel architecture toward multiple-stack machines makes the job somewhat invigorating.

The 80386-based Forth system described herein has been built around the Intel iSBC 386/20P Single Board Computer running on MULTIBUS I.[2] The system includes floppy and hard disk support, a full-screen editor, and a complete, standard syntax 80386 assembler. The system also interfaces with a framegrabber and an array processor to allow it to serve as an industrial vision engine.[3] The complete package is targeted for porting to the 386-PC environment as hardware becomes available.

Execution Threading

Several concerns confront system authors in designing a 32-bit Forth. First, dealing with 32-, 16-, and 8-bit data requires the invention of some new primitives. This problem is easily solved by simply extending the solutions already in use in 16-bit Forth. Second, the possibility of huge dictionaries raises concerns about dictionary search, and therefore compilation, speed. Search speed may be maintained, however, by

[1]MS-DOS is a trademark of MicroSoft Corporation.

[2]iSBC and MULTIBUS are trademarks of Intel Corporation.

[3]This system was developed under contract with Control Automation, Inc., Princeton, NJ.

Fairchild, Intel, Motorola, National and IBM are trade names used herein to refer to Fairchild Camera and Instrument Corporation, Intel Corporation, Motorola Inc., National Semiconductor Corporation, and International Business Machines Corporation.

employing the standard technique of hashing into multiple (i.e., 256) search paths. Other porting issues have been addressed and conquered before; in fact, the only critical design decision involves the manner in which execution will "thread," or flow from one word to another.

In designing Forth for speed, two threading approaches are commonly used; direct and subroutine. In the direct threaded approach, Forth definitions are compiled as a list of code field addresses. This list is headed by the address of the word **DOCOLON**, and terminated by the address of the word **EXIT**. Table 1 shows how these words are typically coded on the 8086 and the 80386, along with execution times in clock cycles. The direct threading into and out of a colon definition requires 119 clocks on the 8086. The same job can be done in 52 clocks on the 80386 due to more efficient instruction execution.[4] These times represent a measure of the overhead involved in executing colon definitions; looked at another way, they measure the time needed to execute a Forth word which does nothing, as in **: NULL ;**

Table 1: Direct Threading on the 8086 and 80386

	8086 Implementation			80386 Implementation		
	Code		Time	Code		Time
DOCOLON:	INC	DI	63 clocks	ADD	EDI,4	27 clocks
	INC	DI				
	DEC	BP		SUB	EBP,4	
	DEC	BP				
	MOV	[BP],SI		MOV	[EBP],ESI	
	MOV	SI,DI		MOV	ESI,EDI	
	LODSW			LODSD		
	XCHG	AX,SI		XCHG	EAX,ESI	
	JMP	[SI]		JMP	[ESI]	
EXIT:	MOV	SI,[BP]	56 clocks	MOV	ESI,[EBP]	25 clocks
	INC	BP		ADD	EBP,4	
	INC	BP				
	LODSW			LODSD		
	XCHG	AX,SI		XCHG	EAX,ESI	
	JMP	[SI]		JMP	[ESI]	

Subroutine threading contrasts sharply with direct threading in that the microprocessor's own subroutine call and return capability transfers control from one word to another. Compiled Forth words consist of a list of machine code **CALL** instructions, optionally interspersed with other machine code instructions. Execution is terminated by a machine code **RET** instruction. In a 16-bit Forth, subroutine threading increases the storage requirements of compiled Forth words by 50% as the **CALL** opcode byte must be stored with each two-byte address. This overhead decreases to 25% in 32-bit Forth, as the pointers to the routines must be four bytes long at the outset. Table 2 shows the coding for this simple subroutine threading approach on the 8086 and 80386, along with execution times in clock cycles. Subroutine threading into and out of a colon definition takes 27 clock cycles on the 8086 and only 19 clocks on the 80386, again owing to more efficient instruction execution on the 80386.

Table 3 summarizes these threading speeds, assigning direct threading on the 8086 (the slowest) a speed of 1.0. This direct clock count comparison ignores the 80386's advantage in clock rate-- a 16 MHz part is presently available, and 25 and 30 MHz parts are planned by Intel. Adhering to the present, Table 4 projects

[4]Branching instructions on the 80386 (JMP, CALL, RET, etc.) have an execution time which is dependent on the complexity of the instruction located at branch destination. This effect is caused by prefetch queue reloading delays. Timing calculations herein assume an average reload delay of 1 clock cycle, which has proven reasonable experimentally.

Table 2: Subroutine Threading on the 8086 and 80386

8086 Implementation		80386 Implementation	
Code	Time	Code	Time
CALL addr	19 clocks	CALL addr	8 clocks
RET	8 clocks	RET	11 clocks

threading times for the 8086 running at 8 MHz and the 80386 running at 16 MHz. The threading times for the 8086 have been experimentally verified for several versions of Forth running on the 8086. As a result of inevitable wait-states incurred by a 16 MHz CPU in accessing RAM, the actual subroutine threading time on the 80386 averages 1.5 μS. The iSBC 386/20P board uses a cache subsystem to eliminate some of these wait-states, although cacheing is of limited utility in executing a threaded language like Forth.

Threading Type	Processor	
	8086	80386
Direct	1.0	2.3
Subroutine	4.4	6.3

Table 3: Relative Threading Speed At Equal Clock Rates

Threading Type	Processor	
	8086	80386
Direct	14.9 μS	3.25 μS
Subroutine	3.38 μS	1.19 μS

Table 4: Projected Threading Overhead (Null Definition Execution Time) 8086 at 8 MHz, 80386 at 16 MHz

It is clear that subroutine threading offers large speed advantages over the direct threaded approach. Threading speed is very important in Forth, as high-level routines which make few calls to primitives can easily spend more than half their execution time threading. The *execution* of primitives, however, is somewhat complicated by subroutine threading, as explored next.

Coding of Primitives under Direct and Subroutine Threading

In direct threaded Forth, the return stack is pointed to by the **BP** register and is managed separately by **DOCOLON** and **EXIT**. This leaves the CPU stack pointed to by the **SP** register free for use as the parameter stack. Conversely, in subroutine threaded Forth the CPU stack must be the return stack. Thus, a second register, usually **BP**, must be used as a pointer for the parameter stack. As the Intel architecture only provides **PUSH** and **POP** instructions which operate relative to the **SP** register, and as it has no register auto-increment or auto-decrement capabilities, interacting with the parameter stack at the machine level is cumbersome. The code for **DUP**, again on both processors and for both threading techniques, is shown in Table 5. Note that the more orthogonal addressing capability of the 80386 is used to advantage in both threading environments.

Performing a **DUP** is significantly more time consuming under subroutine threading on the Intel architecture. The other primitives are affected in a similar fashion; any word that modifies the depth of the stack must update the **EBP** register explicitly. Fortunately, the add and subtract immediate instructions on the 80386 execute in only two clock cycles, and so while a greater burden is placed on the Forth system author in developing the kernel, a much faster final product results from the huge savings in threading time.

Definition Interchangeability and Inline Code

Subroutine threading offers the additional advantage of simplifying the interface between colon definitions and **CODE** definitions. All subroutine threaded words, whether written in Forth or code, are compiled

Table 5: Coding of DUP for Direct and Subroutine Threading

Threading Type	8086 Implementation Code	8086 Implementation Time	80386 Implementation Code	80386 Implementation Time
Direct	MOV DI,SP PUSH [DI]	23 clocks	PUSH [ESP]	5 clocks
Subroutine	MOV DI,BP XCHG BP,SP PUSH [DI] XCHG BP,SP	31 clocks	MOV EAX,[EBP] SUB EBP,4 MOV [EBP],EAX	8 clocks

as machine code routines. This unification of the compiled form of all definitions is useful when code definitions must be called by both Forth words and code words without execution-time penalty. Furthermore, machine language instructions may be inserted with impunity *into the bodies* of Forth definitions. This powerful capability, known as **inline code,** permits *partial* optimization of debugged Forth definitions to increase their execution speed without necessitating complete translation into assembly language. This facility provides a very wide latitude in managing the trade-off between programmer hours and final execution speed.

Inline code has also been used to great advantage in the optimization of the Forth control flow constructs. For example, the execution time of **LOOP** was decreased by 40% by compiling it as:

```
CALL  (loop)
JNZ   (beginning of loop)
ADD   ESP,8
```

This sequence calls an index incrementing routine, **(loop)**, which leaves the zero flag set if the loop should terminate. The conditional branch instruction then either branches back to the word following the original **DO** or falls through to the **ADD** instruction which drops the index and limit off the return stack. This arrangement eliminates the conditional branch in the **(loop)** routine which would otherwise be needed to orchestrate this drop. Using this technique, the subroutine threaded Forth on the 80386 at 16 MHz executes 307,000 empty **DO-LOOP**s per second.

Conclusion

Several test programs were executed on a direct threaded, 8 MHz 8086 system and the subroutine threaded, 16 MHz 80386 system. The 80386 system proved between six and eight times faster than the 8086. The introduction of the 25 MHz 80386 will decrease instruction execution time by over 50%, although more clock cycles will be wasted on memory wait-states as the RAM struggles to keep up with the CPU. For this reason, the 25 MHz 80386 will probably run Forth ten to twelve times faster than does the 8086. Defining macros which assemble inline code for simple but inefficient constructs like **DROP DUP** or **DUP >R** can often increase execution speed by an additional order of magnitude.

The enormous, 4 gigabyte dictionary space of the 80386 opens up many new frontiers for effective Forth application. While Forth implementation on the 80386 is somewhat complicated by the Intel architecture and the choice of subroutine threading, these complications are transparent to Forth programmers. The resulting machine is nearly three times faster than a direct threaded 80386 Forth would be, at only moderate storage expense. At the same time, subroutine threading leads to a unified compiled definition structure that opens up new possibilities for implementation and optimization.

Acknowledgement

The author wishes to thank the University of Vermont, Intel Corporation, and Control Automation, Inc. for invaluable assistance and support throughout this project.