Analysis of Several Extended-Memory 8086 Forth Implementation Approaches

Robert Berkey

2334 Dumbarton Ave.
Palo Alto, CA 94303

Abstract

Several 8086 system designs are evaluated that provide the Forth application programmer one megabyte of dictionary space. A flat (non-segmented) address space with a 32-bit virtual machine is chosen.

Introduction

This paper will first look at four approaches to implementing the stack and the corollary of Forth's addressing of main memory, then, at alternatives to threaded-code implementations, and finally, tradeoffs of these various approaches will be evaluated.

Stack Implementation

The stack is how the application programmer "sees", in the mind's eye, the architecture of the machine. Forth is a virtual architecture; the 8086 system designer's task is to choose a virtual architecture that provides an optimum interface between the application programmer and the physical architecture of the underlying machine. The choice of stack implementation has a major impact not only in modification of previously existing code, but also has a permanent impact at the application programmer level.

Four approaches to stack implementation were analyzed.

(1) 16-bit wide stack	Default segment addressing
(2) 16-bit wide stack	Explicit segment addressing
(3) 32-bit wide stack	Absolute addressing
(4) 32-bit wide stack	Segmented addressing

The approaches are organized first by stack width, which is the size of the virtual machine (the size of the machine from the application programmer's viewpoint). The 16-bit-wide stack has been the traditional machine size in the Forth community. With a standard Forth a 16-bit-wide stack provides direct addressing to 64k bytes of memory, and the register size for arithmetic is based on 16-bit operations. The 32-bit-wide stack allows user programs a register size of 32 bits for arithmetic operations and may allow direct addressing beyond 16 bits of memory.

Within each stack size two addressing methods have been studied. These will be detailed in the following discussion.

All of these stack approaches provide for the 8086 application programmer a full-range 20 bits of address space.

16-Bit-Wide Stack, Default Segmentation

The first of the four stack-implementation approaches discussed is the 16-bit-wide stack with default segmentation. In this approach a default segment address (a register) is maintained as a part of the Forth architecture.

For the Forth application programmer this approach is more complex than any of the other three approaches. It is, however, the fastest of the four approaches and it minimizes the impact on existing code. Forty-two primitives were coded and speed-optimized using this approach. The default segment is implicitly set by words that put an address on the stack, including `BLOCK` , `DOUSER`, `SP@`, `DOCREATE`, etc. The default segment is explicitly set with `!SEG` or `R>SEG`. `@SEG` fetches the current default segment to the data stack while `SEG>R` fetches the default segment to the return stack. One additional primitive required is `SEGLIT`, which is like `LIT`, but sets the default register rather than putting a literal value on the stack.

While most words keep the same input/output parameters, `CMOVE` requires application code modification. Its parameters change from (from to count --) to (from from-seg to count --). In a simple example only a `@SEG` need be added to convert the old code. Note that `PAD`, because it put an address on the stack, also sets the default segment.

```
9 BLOCK      PAD 1024 CMOVE becomes
9 BLOCK @SEG PAD 1024 CMOVE
```

In use, the default segment register is similar to a user variable such as `BLK`. Any word using BLK must know what other words might cause `BLK` to be modified. With a default segment, however, the effects are pervasive in code modifying the default. Words must either routinely preserve the default segment or be well documented.

16-Bit-Wide Stack, Explicit Segment

In this approach using a 16-bit-wide stack with explicit segments, all addresses use two stack elements. This is a straightforward approach, in tune with the 8086 architecture, and lacking the hidden features present in the other approaches. The Forth community considers simplicity to be a goal unto itself; often this is because simple techniques contain hidden strengths that are later beneficial.

Unfortunately, simplicity can be elusive. This approach maximizes the modification of existing code. The definition of `CMOVE` for the explicit segment approach has parameters (from from-seg to to-seg count --). The application programmer does not have to remember a hidden register as is needed in default segmentation, but in return he or she must keep track of and manipulate extra items on the stack. A variable under this approach must append two values to the data stack. This is why the explicit-segment approach maximizes modification of existing code—all stack manipulation sequences involving variables must be changed.

32-Bit-Wide Stack and 32-Bit Virtual Machine

The virtue of a 32-bit-wide stack is that a complete address can be contained in one stack element.

The virtual machine, the size of the machine from the application programmer's viewpoint, becomes 32 bits, so that there are few 65535 limitations. An addition now adds two 32-bit numbers. `D+` now adds two 64-bit numbers. It is relevant to note that, as Forth sought a balance between 8- and 16-bit virtual machines, the 8-bit machines fell to the wayside. Machines that are 8 bits internally are routinely implemented as 16-bit virtual machines at the Forth application-programmer level.

A major advantage of the 32-bit stack is that current code is largely supported. Consider

```
AVARIABLE intervening-code 1 SWAP +!
```

In the default-segment approach discussed first, this will work only if the intervening code did not change the default segment. In the explicit-segment approach the SWAP just before the +! must be changed to ROT ROT. In either of the 32-bit approaches, this code does not need to be modified.

To support existing code, 2@ continues to add two 16-bit items to the stack; the upper sixteen bits of each stack element would be sign extended from bit 15. D@ D! and D+! are used for manipulating 32-bit data from one stack element. For example: CREATE XYZ -1 , -1, creates a word XYZ with a 32-bit parameter field. XYZ 2@ puts a double -1 on the data stack, while XYZ D@ puts a single -1 on the data stack.

32-Bit-Wide Stack, Absolute Address

One approach using a 32-bit-wide stack makes use of absolute addresses. "Absolute address" refers to the application programmer level. The address space appears flat, with 20 bits (one megabyte) of implementable range. Internally, including the threaded-code interpretive pointers on the return stack, addresses are still segmented.

This approach is the only standardizable, portable structure of the four analyzed. This means that a Forth programmer coming in off the street would need to learn nothing about segmentation. The programmer could read and write the code as it is. Additionally, later code transfers to other processors is supported. Nuclei designed for other 32-bit machines will be more useful for code written with this approach.

The 20-bit absolute address is also the same address the 8086 chip hardware presents to its address lines, but the 8086 opcodes do not include operators appropriate for manipulating these numbers. The conversion of a segment address to absolute address requires eight shift operations and additional processing, with about the same cycle time as twenty additions or two NEXTs or half of a division. For this reason, this approach is the slowest of those considered.

32-Bit-Wide Stack, Segmented Address

The 32-bit-wide stack with segmented address is similar in impact to the 32-bit absolute-address approach. In the segmented-address approach, an address contained in one stack element contains two disjoint components: a 16-bit segment and a 16-bit offset. Only in address calculations involving carries into or from the high sixteen bits does a problem occur. The primary tradeoff is that of increased speed versus increased system dependencies.

A likely application programmer mistake here and with either of the 16-bit approaches would be to subtract a few bytes from the address of the parameter field (APF) to look at the header. Since (with one implementation) the APF offset is always 42, subtraction of more than 42 bytes would create address wrap-around and an offset greater than 65000. For a 16-bit virtual machine the dump would show the high end of the segment. For the 32-bit segmented approach an even more peculiar result occurs; the borrow during the subtract reduces the segment by one, so that the actual memory dumped would be sixteen bytes below that dumped by the 16-bit machines. With the 32-bit absolute-address approach no surprise occurs.

Threaded Code Approaches

In this analysis the choice of threading approach affects only execution speed and memory compactness, and any threading approach could be used with any of the stack approaches discussed previously.

The most obvious threading approach when converting from 16-bit addressing to 20-bit addressing is to make each unit of threaded code a 32-bit address consisting of offset and segment. For each word compiled by a colon definition, two 16-bit cells are added to the dictionary. This tends to double the memory used by a given application, and NEXT is slower than in a traditional system.

Another threading approach considered, referred to here as 4-seg threading, would allow direct access to only 256k of memory. In 4-seg threading, each compilation address of the threaded code would consist of sixteen bits. The lower two bits would indicate which of four 64k segments contained the remainder of the address. The boundary problems this creates can be overcome with appropriate management of the task partitions, but a faster approach exists.

With segment-threaded code each unit of the threaded code consists of one 16-bit segment address (see Segment-Threaded Code in this same issue). This allows direct access to the megabyte of memory without incurring an execution speed penalty in NEXT, and without incurring the memory doubling caused by the 32-bit threaded code. The 8088 and the post-8086 Intel architectures have not been considered in these evaluations.

Tradeoffs Evaluation

In the tradeoffs evaluation in Figure 1 (see also Appendix A), tradeoff categories as listed in the Forth Standard have been adopted. Each category and approach is compared with a traditional 16-bit Forth. The numbers listed are a considered evaluation of the negative impact. Generality for the traditional system is crossed out because it lacks dictionary size beyond a 64k limit. Execution speed is factored in twice.

Figure 1. Tradeoffs Evaluation

Threading:	Tradi-	—Segment-threading—				32-bit	4-seg
Stack width:	tional	—16-bit—		—32-bit-wide-stack—			
Addressing:	System	Def-seg	Exp-seg	—Seg—	—Absolute-address—		
Portability	0	6	16	2	0	0	0
Simplicity	0	8	4	2	1	1	1
Clarity	0	9	0	2	1	1	1
Generality	xxx	0	0	0	0	0	4
Execution Speed	0	0	1	4	6	12	15
" "	0	0	1	4	6	12	15
Memory Compactness	0	6	6	6	6	20	3
Compilation Speed	0	1	1	1	1	1	1
Historical Continuity	0	3	3	2	0	0	0
Teachability	0	4	3	2	1	1	1
Totals	xxx	37	35	25	22 best	48	41

The fewer the points, the better the score. Points are a considered evaluation of undesirability. The preferred approach is segment threaded, has a 32-bit-wide stack, and uses absolute addresses.

The primary organization of the table is by thread type. One thread type, segment threading, is further divided, into two 16-bit virtual machines and two 32-bit virtual machines.

The bottom row shows the result of the conclusions; as in golf, the smallest score is best. Within segment threading the scores of the two 16-bit approaches are close, but the 32-bit approaches are preferred. Of the two 32-bit approaches, the preference is for absolute addresses.

The other threading techniques are shown using the preferred 32-bit absolute-addressing approach. These other threading approaches are substantially less preferred than the segment-threaded approach.

The evaluation numbers represent the opinion of the author. In addition to consideration of the subjective content, specific system requirements might substantially alter the conclusion drawn here.

A significant feature of this tradeoffs evaluation is that, of the segment-threaded approaches, the slowest has been picked. The need to use 8086 substitutes, such as the 8088 and the 80286, may also impact the conclusions drawn.

The conclusion of this tradeoffs evaluation is to recommend segment threading, a 32-bit stack, and absolute addressing.

Appendix A: Timing Comparison Estimate in Cycles

Intel does not provide complete timing information for the 8086. Intel claims their numbers to be "typically within 5-10%", but has not provided adequate documentation on the impact of the timing of the bus interface unit. "The time provided assumes that the instruction has already been prefetched and that it is waiting in the instruction queue" (Intel 1983, p. 2-50). It is well known that even with the fastest memory, certain sequences of code execute 100% slower than the published timings. Another factor affecting timings is the number of wait states on the memory board.

Experience with physical timings shows that the impact of the bus interface unit has significant impact on the choice of optimum routine for Forth implementations on the 8086. One peculiar effect of the bus interface unit is that occasionally a code routine should be aligned on an odd address boundary for faster execution. In some cases the choice of in-line NEXT varies from code word to code word. Most of the following timings have not been physically timed, rather they are based on the available Intel documentation modified by an estimate as to any impact of the 8086 bus interface unit.

Significantly different numbers should be expected for the 8088 and other Intel processors that run 8086 code.

Threading:		Tradi-	Segment-threading				32-bit	4-seg
Stack width:		tional	16-bit		32-bit			
Addressing:		System	Def Seg	Exp Seg	32-bit Seg	Absolute-address		
Key Measure	Weight							
`LITERAL`	2	126	124	124	144	144	+16ea.	+24ea.
`DOCOLON`	6	402	426	426	426	426	(1)	(1)
`EXIT`	6	360	366	366	366	366		
`DOVARIABLE`	2	106	98	114	114	198		
`BRANCH`	1	56	54	54	54	54		
`?BRANCH`	1	70	69	69	81	81		
`0=`	1	75	64	64	86	86		
`@`	1	70	70	76	88	104		
`!`	1	67	71	77	81	99		
`OVER`	3	198	213	213	297	297		
`+`	3	204	198	198	297	297		
	27							
TOTAL CYCLES		1734	1753	1781	2034	2152	2584	2800
%		1.00	1.01	1.03	1.17	1.24	1.49	1.61

(1) The effect of `NEXT` is included once in each key measure and does not show explicitly. The only timing difference in the last three columns is in the value of `NEXT` .

Reference

Intel. *The 8086 Family User's Manual.* Intel Corporation, May 1983.

Manuscript received August 1984.