MOVING FORTH

Part 2: Benchmarks and Case Studies of Forth Kernels

by Brad Rodriguez

This article first appeared in The Computer Journal #60 (March/April 1993).

BENCHMARKS

By now it must seem that the answer to every design question is "code it and see." Obviously you don't want to write the entire Forth kernel several different ways just to evaluate different schemes. Fortunately, you can get quite a good "feel" with just a small subset of the Forth kernel.

Guy Kelly [KEL92] examines the following code samples for 19 different IBM PC Forths:

NEXT ...the "inner interpreter" that chains from one Forth word to another in the "thread". This is used at the end of every CODE definition, and is one of the most important factors in speed of Forth execution. You've already seen the pseudo-code for this in ITC and DTC; in STC it's just CALL/RETURN.

ENTER ...also called DOCOL or DOCOLON; the Code Field action that causes a high level "colon" definition to be executed. This, too, is crucial for speed; it is used at the start of every colon definition. Not needed in STC.

EXIT ...called ;S in fig-Forth; the code that ends the execution of a colon definition. This is essentially the high-level subroutine return, and appears at the end of every colon definition. This is just a machine code RETURN in STC.

NEXT, ENTER, and EXIT indicate the performance of the threading mechanism. These should be coded to evaluate ITC vs. DTC vs. STC. They also reflect the quality of your register assignments for IP, W, and RSP.

DOVAR ...a.k.a. "variable"; the machine code fragment that is the Code Field action for all Forth VARIABLEs.

DOCON ...a.k.a. "constant"; the machine code fragment that is the Code Field action for all Forth CONSTANTs.

DOCON and DOVAR, along with ENTER, show how efficiently you can obtain the Parameter Field address of a word being executed. This reflects your choice for the W register. In a DTC Forth, this also indicates whether to put a JUMP or CALL in the Code Field.

LIT ...a.k.a. "literal"; is a Forth word that fetches a cell value from the high-level thread. Several words use such in-line parameters, and this is a good indicator of their performance. It reflects your choice for the IP register.

@ ...the Forth memory-fetch operator, shows how quickly memory can be accessed from high-level Forth. This word usually benefits from TOS in stack.

! ...the Forth memory-store operator, is another indicator of memory access. This consumes two items from the stack, and illustrates efficiency of Parameter Stack access. It's a good indicator of the TOS-in-memory vs. TOS-in-register tradeoff.

+ ...the addition operator, is a representative example of all the Forth arithmetic and logical operators. Like the ! word, this benchmarks stack access, and it's a clear demonstration of any TOS-in-register benefit.

This is an excellent set of code samples. I have a few additional favorites:

DODOES ...is the Code Field action for words built with DOES>. This doesn't yield any new benchmark comparisons, although it does reflect the usefulness of W, IP, and RSP. I include it because it's the most convoluted code in the Forth kernel. If you can code the logic of DODOES, everything else is a snap. The intricacies of DODOES will be described in a subsequent article.

SWAP ...a simple stack operator, but still educational.

OVER ...a more complex stack operator. This gives a good idea of how easily you can access the Parameter Stack.

ROT ...a still more complex stack operator, and the one most likely to need an extra temporary register. If you can code ROT without needing an "X" register, you probably don't need an "X" register for anything.

0= ...one of the few unary arithmetic operators, and one of the most likely to benefit from TOS-in-register.

+! ...a most illustrative operator, combining stack access, arithmetic, memory fetch and store. This is one of my favorite benchmarks, although it is less frequently used than the other words in this list.

These are among the most-used words in the Forth kernel. It pays to optimize them. I'll show examples of all of these, including pseudo-code, for the 6809. For the other CPUs, I'll use selected examples to illustrate specific decisions.

CASE STUDY 1: THE 6809

In the world of 8-bit CPUs, the 6809 is the Forth programmer's dream machine. It supports two stacks! It also has two other address registers, and a wealth of orthogonal addressing modes second only to the PDP-11. ("Orthogonal" means they work the same way and have the same options for all address registers.) The two 8-bit accumulators can be treated as a single 16-bit accumulator, and there are many 16-bit operations.

The programmer's model of the 6809 is [MOT83]:

A - 8 bit accumulator
B - 8 bit accumulator

Most arithmetic operations use an accumulator as the destination. These can be concatenated and treated as a single 16-bit accumulator D (A high byte, B low).

X - 16 bit index register
Y - 16 bit index register
S - 16 bit stack pointer
U - 16 bit stack pointer

All addressing modes for X and Y can also be used with the S and U registers.

PC - 16 bit program counter
CC - 8 bit Condition Code register
DP - 8 bit Direct Page register

The 6800 family's Direct addressing mode uses an 8-bit address to reach any location in memory page zero. The 6809 allows any page to be Direct-addressed; this register provides the high 8 bits of address.

Those two stack pointers are crying out for Forth use. They are equivalent, except that S is used for subroutine calls and interrupts. Let's be consistent and use S for return addresses, leaving U for the Parameter Stack.

W and IP both need to be address registers, so these are the logical use for X and Y. X and Y are equivalent, so let's arbitrarily assign X=W, and Y=IP.

Now a threading model can be chosen. I'll scratch STC and TTC, to make this a "conventional" Forth. The limiting factor in performance is then the NEXT routine. Let's look at this in both ITC and DTC:

```
ITC-NEXT: LDX ,Y++   (8) (IP)->W, increment IP
          JMP [,X]   (6) (W)->temp, jump to adrs in temp

DTC-NEXT: JMP [,Y++] (9) (IP)->temp, increment IP,
                         jump to adrs in temp
                         ("temp" is internal to the 6809)
```

NEXT is one instruction in a DTC 6809! This means you can code it in-line in two bytes, making it both smaller and faster than JMP NEXT. For comparison, look at the "NEXT" logic for subrou~ tine threading:

```
        RTS           (5) ...at the end of one CODE word
        JSR nextword  (8) ...in the "thread"
        ...               ...start of the next CODE word
```

STC takes 13 clocks to thread to the next word, compared with 9 clocks for DTC. This is because subroutine threading has to pop and push a return address, while simple DTC or ITC threading between CODE words does not.

Given the choice of DTC, you have to decide: does a high-level word have a Jump or Call in its Code Field? The driving consid~ eration is how quickly can you obtain the address of the parame~ ter field which follows? Let's look at the code to ENTER a colon definition, using symbolic Forth register names, to see this illustrated:

using a JSR (Call):

```
        JSR ENTER   (8)
        ...
ENTER:  PULS W      (7) get address following JSR into W reg
        PSHS IP     (7) save the old IP on the Return Stack
        TFR  W,IP   (6) Parameter Field address -> IP
        NEXT        (9) assembler macro for JMP [,Y++]
                    37 cycles total
```

using a JMP:

```
        JMP ENTER   (4)
         ...
ENTER:  PSHS IP     (7) save the old IP on the Return Stack
        LDX -2,IP   (6) re-fetch the Code Field address
        LEAY 3,X    (5) add 3 and put into IP (Y) register
        NEXT        (9)
                    31 cycles total

                    (CPU cycle counts are in parentheses.)
```

The DTC 6809 NEXT doesn't use the W register, because the 6809 addressing modes allow an extra level of indirection automatically. The JMP version of ENTER has to re-fetch the Code Field address -- NEXT didn't leave it in any register -- and then add 3 to get the Parameter Field address. The JSR version can get the Parameter Field address directly by popping the return stack. Even so, the JMP version is faster. (Exercise for the student: try coding the JSR ENTER with S=PSP and U=RSP.)

Either way, the code for EXIT is the same:

```
EXIT:   PULS IP     pop "saved" IP from return stack
        NEXT        continue Forth interpretation
```

Some registers remain to allocate. You could keep the User Pointer in memory, and this Forth would still be pretty fast. But the DP register would go to waste, and there's not much else it can do. Let's use the "trick" described above, and hold the high byte of UP in the DP register. (The low byte of UP is implied to be zero).

One 16-bit register is left: D. Most arithmetic operations need this register. Should it be left free as a scratch register, or used as the Top-Of-Stack? 6809 instructions use memory as one operand, so a second working register may be unnecessary. And if a scratch register is needed, it's easy to push and pop D. Let's write the benchmark primitives both ways, and see which is faster.

NEXT, ENTER, and EXIT don't use the stack, and thus have identical code either way.

DOVAR, DOCON, LIT, and OVER require the same number of CPU cycles either way. These illustrate the earlier comment that putting TOS in register often just changes where the push or pop takes place:

```
        TOS in D        TOS in memory   pseudo-code

DOVAR:  PSHU TOS        LDD  -2,IP      address of CF -> D
        LDD  -2,IP      ADDD #3         address of PF -> D
        ADDD #3         PSHU D          push D onto stack
        NEXT            NEXT

DOCON:  PSHU TOS        LDX  -2,IP      address of CF -> W
        LDX  -2,IP      LDD  3,X        contents of PF -> D
        LDD  3,X        PSHU D          push D onto stack
        NEXT            NEXT

LIT:    PSHU TOS        LDD  ,IP++      (IP) -> D, increment IP
        LDD  ,IP++      PSHU D          push D onto stack
        NEXT            NEXT

OVER:   PSHU D          LDD  2,PSP      2nd on stack -> D
```

```
        LDD  2,PSP      PSHU D          push D onto stack
        NEXT            NEXT
```

SWAP, ROT, 0=, @, and especially + are all faster with TOS in register:

```
        TOS in D        TOS in memory   pseudo-code

SWAP:   LDX  ,PSP (5)   LDD  ,PSP (5)   TOS -> D
        STD  ,PSP (5)   LDX 2,PSP (6)   2nd on stack -> X
        TFR  X,D  (6)   STD 2,PSP (6)   D -> 2nd on stack
        NEXT            STX  ,PSP (5)   X -> TOS
                        NEXT

ROT:    LDX  ,PSP (5)   LDX  ,PSP (5)   TOS -> X
        STD  ,PSP (5)   LDD 2,PSP (6)   2nd on stack -> D
        LDD 2,PSP (6)   STX 2,PSP (6)   X -> 2nd on stack
        STX 2,PSP (6)   LDX 4,PSP (6)   3rd on stack -> X
        NEXT            STD 4,PSP (6)   D -> 3rd on stack
                        STX  ,PSP (5)   X -> TOS
                        NEXT

0=:     CMPD #0         LDD  ,PSP       TOS -> D
        BEQ  TRUE       CMPD #0         does D equal zero?
                        BEQ  TRUE
  FALSE:LDD  #0         LDD  #0         no...put 0 in TOS
        NEXT            STD  ,PSP
                        NEXT
  TRUE: LDD  #-1        LDD  #-1        yes...put -1 in TOS
        NEXT            STD  ,PSP
                        NEXT

@:      TFR TOS,W (6)   LDD [,PSP] (8)  fetch D using TOS adrs
        LDD  ,W   (5)   STD  ,PSP  (5)  D -> TOS
        NEXT            NEXT

+:      ADDD ,U++       PULU D          pop TOS into D
        NEXT            ADDD ,PSP       add new TOS into D
                        STD  ,PSP       store D into TOS
                        NEXT
```

! and +! are slower with TOS in register:

```
        TOS in D        TOS in memory   pseudo-code

!:      TFR TOS,W (6)   PULU W   (7)    pop adrs into W
        PULU D    (7)   PULU D   (7)    pop data into D
        STD  ,W   (5)   STD  ,W  (5)    store data to adrs
        PULU TOS  (7)   NEXT
        NEXT

+!:     TFR TOS,W (6)   PULU W   (7)    pop adrs into W
        PULU TOS  (7)   PULU D   (7)    pop data into D
        ADDD ,W   (6)   ADDD ,W  (6)    add memory into D
        STD  ,W   (5)   STD  ,W  (5)    store D to memory
        PULU TOS  (7)   NEXT
        NEXT
```

The reason these words are slower is that most Forth memory- reference words expect the address on the top of stack, so an extra TFR instruction is needed. This is why it's a help for the TOS register to be an address register. Unfortunately, all the 6809 address registers are spoken for...and it's much more important for W, IP, PSP, and RSP to be in address registers than TOS. The TOS-in-register penalty for ! and +! should be outweighed by the gains in the many arithmetic and stack operations.

CASE STUDY 2: THE 8051

If the 6809 is the Forthwright's dream machine, the 8051 is the nightmare. It has only one general-purpose address register, and one addressing mode, which always uses the one 8-bit accumulator.

All of the arithmetic operations, and many of the logical, must use the accumulator. The only 16-bit operation is INC DPTR. The hardware stack must use the 128-byte on-chip register file. [SIG92] Such a CPU could give ulcers.

Some 8051 Forths have been written that implement a full 16-bit model, e.g. [PAY90], but they are too slow for my taste. Let's make some tradeoffs and make a faster 8051 Forth.

Our foremost reality is the availability of only one address register. So let's use the 8051's Program Counter as IP -- i.e., let's make a subroutine-threaded Forth. If the compiler uses 2- byte ACALLs instead of 3-byte LCALLs whenever possible, most of the STC code will be as small as ITC or DTC code.

Subroutine threading implies that the Return Stack Pointer is the hardware stack pointer. There are 64 cells of space in the on-chip register file, not enough room for multiple task stacks. At this point you can

a) restrict this Forth to single-task;
b) code all of the Forth definitions so that upon entry they move their return address to a software stack in external RAM; or
c) do task switches by swapping the entire Return Stack to and from external RAM.

Option (b) is slow! Moving 128 bytes on every task switch is faster than moving 2 bytes on every Forth word. For now I choose option (a), leaving the door open for (c) at some future date.

The one-and-only "real" address register, DPTR, will have to do multiple duty. It becomes W, the multi-purpose working register.

In truth, there are two other registers that can address external memory: R0 and R1. They provide only an 8-bit address; the high 8 bits are explicitly output on port 2. But this is a tolerable restriction for stacks, since they can be limited to a 256-byte space. So let's use R0 as the PSP.

This same 256-byte space can be used for user data. This makes P2 (port 2) the high byte of the User Pointer, and, like the 6809, the low byte will be implied to be zero.

What is the programmer's model of the 8051 so far?

```
   reg 8051   Forth
  adrs name   usage

     0  R0    low byte of PSP
     1  R1
     2  R2
     3  R3
     4  R4
     5  R5
     6  R6
     7  R7
  8-7Fh        120 bytes of return stack
    81h  SP    low byte of RSP (high byte=00)
 82-83h  DPTR  W register
    A0h  P2    high byte of UP and PSP
    E0h  A
    F0h  B
```

Note that this uses only register bank 0. The additional three register banks from 08h to 1Fh, and the bit-addressable region from 20h to 2Fh, are of no use to Forth. Using bank 0 leaves the largest contiguous space for the return stack. Later the return stack can be shrunk, if desired.

The NEXT, ENTER, and EXIT routines aren't needed in a subroutine threaded Forth.

What about the top of stack? There are plenty of registers, and memory operations on the 8051 are expensive. Let's put TOS in R3:R2 (with R3 as the high byte, in Intel fashion). Note that B:A can't be used -- the A register is the funnel through which all memory references must move!

Harvard architectures

The 8051 uses a "Harvard" architecture: program and data are kept in separate memories. (The Z8 and TMS320 are two other examples.) The 8051 is a degenerate case: there is physically no means to write to the program memory! This means that a Forthwright can do one of two things:

a) cross-compile everything, including the application, and give up all hope of putting an interactive Forth compiler on the 8051; or

b) cause some or all of the program memory to also appear in the data space. The easiest way is to make the two spaces completely overlap, by logically ORing the active-low PSEN* and RD* strobes with an external AND gate.

The Z8 and TMS320C25 are more civilized: they allow write access to program memory. The implications for the design of the Forth kernel will be discussed in subsequent articles.

CASE STUDY 3: THE Z80

The Z80 is instructive because it is an extreme example of a non- orthogonal CPU. It has four different kinds of address registers! Some operations use A as destination, some any 8-bit register, some HL, some any 16-bit register, and so on. Many operations (such as EX DE,HL) are only defined for one combination of registers.

In a CPU such as the Z80 (or 8086!), the assignment of Forth functions must be carefully matched to the capabilities of the CPU registers. Many more tradeoffs need to be evaluated, and often the only way is to write sample code for a number of different assignments. Rather than burden this article down endless permutations of Forth code, I'll present one register assignment based on many Z80 code experiments. It turns out that these choices can be rationalized in terms of the general principles outlined earlier.

I want a "conventional" Forth, although I will use direct threading. All of the "classical" virtual registers will be needed.

Ignoring the alternate register set, the Z80 has six address registers, with the following capabilities:

```
BC,DE - LD A indirect, INC, DEC
        also exchange DE/HL

   HL - LD r indirect, ALU indirect, INC, DEC, ADD, ADC,
        SBC, exchange w/TOS, JP indirect

IX,IY - LD r indexed, ALU indexed, INC, DEC, ADD, ADC,
        SBC, exchange w/TOS, JP indirect  (all slow)

   SP - PUSH/POP 16-bit, ADD/ADC/SUB to HL/IX/IY
```

BC, DE, and HL can also be manipulated in 8-bit pieces.

The 8-bit register A must be left as a scratch register, since it's the destination for so many ALU and memory reference operations.

HL is undoubtedly the most versatile register, and at one time or another it is tempting to use it for each of the Forth virtual registers. However, because of its versatility -- and because it is the only register which can be fetched byte-wise and used in an indirect jump -- HL should be used for W, Forth's all-purpose working register.

IX and IY might be considered for the Forth stack pointers, because of their indexed addressing mode, which can be used in ALU operations. But there are two problems with this: it leaves SP without a job; and, IX/IY are too slow! Most of the operations on either stack involve pushing or popping 16-bit quantities. This is one instruction using SP, but it requires four using IX or IY. One of the Forth stacks should use SP. And this should be the Parameter Stack, since it is used more heavily than the Return Stack.

What about Forth's IP? Mostly, IP fetches from memory and autoincrements, so there's no programming advantage to using IX/IY over BC/DE. But speed is of the essence with IP, and BC/DE are faster. Let's put IP in DE: it has the advantage of being able to swap with HL, which adds versatility.

A second Z80 register pair (other than W) will be needed for 16- bit arithmetic. Only BC is left, and it can be used for addressing or for ALU operations with A. But should BC be a second working register "X", or the top-of-stack? Only code will tell; for now, let's optimistically assume that BC=TOS.

This leaves the RSP and UP functions, and the IX and IY registers unused. IX and IY are equivalent, so let's assign IX=RSP, and IY=UP.

Thus the Z80 Forth register assignments are:

```
   BC = TOS   IX = RSP
   DE = IP    IY = UP
   HL = W     SP = PSP
```

Now look at NEXT for the DTC Forth:

```
DTC-NEXT: LD A,(DE) (7) (IP)->W, increment IP
          LD L,A    (4)
          INC DE    (6)
          LD A,(DE) (7)
          LD H,A    (4)
          INC DE    (6)
          JP (HL)   (4) jump to address in W
```

alternate version (same number of clock cycles)

```
DTC-NEXT: EX DE,HL  (4) (IP)->W, increment IP
NEXT-HL:  LD E,(HL) (7)
          INC HL    (6)
          LD D,(HL) (7)
          INC HL    (6)
          EX DE,HL  (4)
          JP (HL)   (4) jump to address in W
```

Note that cells are stored low-byte first in memory. Also, although it might seem advantageous to keep IP in HL, it really isn't. This is because the Z80 can't JP (DE). The NEXT-HL entry point will be used shortly.

Just for comparison, let's look at an ITC NEXT. The pseudo-code given previously requires another temporary register "X", whose contents can be used for an indirect jump. Let DE=X, and BC=IP. TOS will have to be kept in memory.

```
ITC-NEXT: LD A,(BC) (7) (IP)->W, increment IP
          LD L,A    (4)
          INC BC    (6)
          LD A,(BC) (7)
          LD H,A    (4)
          INC BC    (6)

          LD E,(HL) (7) (W)->X
          INC HL    (6)
          LD D,(HL) (7)
          EX DE,HL  (4) jump to address in X
          JP (HL)   (4)
```

This leaves "W" incremented by one, and in the DE register. As long as this is done consistently, there's no problem -- code needing the contents of W knows where to find it, and how much to adjust it.

The ITC NEXT is 11 instructions, as compared to 7 for DTC. And ITC on the Z80 loses the ability to keep TOS in a register. My choice is DTC.

If coded in-line, DTC NEXT would require seven bytes in every CODE word. A jump to a common NEXT routine would only use three bytes, but would add 10 clock cycles. This is another of the tradeoff decisions in designing a Forth kernel. This example is a close call; let's opt for speed with an in-line NEXT. But sometimes NEXT is so huge, or memory is so tight, that the prudent decision is to use a JMP NEXT.

Now let's look at the code for ENTER. Using a CALL, the hardware stack is popped to get the Parameter Field address:

```
        CALL ENTER  (17)
        ...
ENTER:  DEC IX      (10) push the old IP on the return stack
        LD (IX+0),D (19)
        DEC IX      (10)
        LD (IX+0),E (19)
        POP DE      (10) Parameter Field address -> IP
        NEXT        (38) assembler macro for 7 instructions
```

Actually it's faster to POP HL, and then use the last six instructions of NEXT (omitting the EX DE,HL):

```
        CALL ENTER  (17)
        ...
ENTER:  DEC IX      (10) push the old IP on the return stack
```

```
        LD (IX+0),D (19)
        DEC IX      (10)
        LD (IX+0),E (19)
        POP HL      (10) Parameter Field address -> HL
        NEXT-HL     (34) see DTC NEXT code, above
                    119 cycles total
```

When a JP is used, the W register (HL) is left pointing to the Code Field. The Parameter Field is 3 bytes after:

```
        JP ENTER    (10)
        ...
ENTER:  DEC IX      (10) push the old IP on the return stack
        LD (IX+0),D (19)
        DEC IX      (10)
        LD (IX+0),E (19)
        INC HL      ( 6) Parameter Field address -> IP
        INC HL      ( 6)
        INC HL      ( 6)
        NEXT-HL     (34)
                    120 cycles total
```

Again, because of the alternate entry point for NEXT, the new value for IP doesn't actually have to be put into the DE register pair.

The CALL version is one cycle faster. On an embedded Z80, a one- byte RST instruction could be used to gain speed and save space. This option is not available on many Z80-based personal computers.

CASE STUDY 4: THE 8086

The 8086 is another instructive CPU. Rather than go through the design process, let's look at one of the newer shareware Forths for the IBM PC: Pygmy Forth [SER90].

Pygmy is a direct-threaded Forth with the top-of-stack kept in register. The 8086 register assignments are:

```
   AX = W         DI = scratch
   BX = TOS       SI = IP
   CX = scratch   BP = RSP
   DX = scratch   SP = PSP
```

Most 8086 Forths use the SI register for IP, so that NEXT can be written with the LODSW instruction. In Pygmy the DTC NEXT is:

```
NEXT:  LODSW
       JMP AX
```

This is short enough to include in-line in every CODE word.

High-level and "defined" Forth words use a JMP (relative) to their machine code. The ENTER routine (called 'docol' in Pygmy) must therefore get the Parameter Field address from W:

```
ENTER:  XCHG SP,BP
        PUSH SI
        XCHG SP,BP
        ADD AX,3    Parameter Field address -> IP
        MOV SI,AX
        NEXT
```

Note the use of XCHG to swap the two stack pointers. This allows the use of PUSH and POP instructions for both stacks, which is faster than using indirect access on BP.

```
EXIT:   XCHG SP,BP
        POP SI
        XCHG SP,BP
        NEXT
```

Segment model

Pygmy Forth is a single-segment Forth; all code and data are contained within a single 64 Kbyte segment. (This is the "tiny model" in Turbo C lingo.) All of the Forth standards issued to date assume that everything is contained in a single memory space, accessible with the same fetch and store operators.

Nevertheless, IBM PC Forths are beginning to appear that use multiple segments for up to five different kinds of data [KEL92,SEY89]. These are:

CODE ...machine code
LIST ...high-level Forth threads (a.k.a. THREADS)
HEAD ...headers of all Forth words
STACK ...parameter and return stacks
DATA ...variables and user-defined data

This allows PC Forths to break the 64K limit, without going to the expense of implementing a 32-bit Forth on a 16-bit CPU. Implementation of a multi-segment model, and the ramifications for the Forth kernel, are beyond the scope of this article.

STILL TO COME...

Subsequent articles will look at:
- design tradeoffs in the Forth header and dictionary search
- the logic of CONSTANTs, VARIABLEs, and other data structures
- the defining word mechanisms, CREATE...;CODE and CREATE...DOES>
- the assembler vs. metacompiler question
- the assembler and high-level code that comprises a Forth kernel
- multitasking modifications to the kernel

REFERENCES

[KEL92] Kelly, Guy M., "Forth Systems Comparisons," Forth Dimensions XIII:6 (Mar/Apr 1992). Also published in the 1991 FORML Conference Proceedings. Both available from the Forth Interest Group, P.O. Box 2154, Oakland, CA 94621. Illustrates design tradeoffs of many 8086 Forths with code fragments and benchmarks -- highly recommended!

[MOT83] Motorola Inc., 8-Bit Microprocessor and Peripheral Data, Motorola data book (1983).

[SIG92] Signetics Inc., 80C51-Based 8-Bit Microcontrollers, Signetics data book (1992).

Forth Implementations

[PAY90] Payne, William H., Embedded Controller FORTH for the 8051 Family, Academic Press (1990), ISBN 0-12-547570-5. This is a complete "kit" for a 8051 Forth, including a metacompiler for the IBM PC. Hardcopy only; files can be downloaded from GEnie. Not for the novice!

[SER90] Sergeant, Frank, Pygmy Forth for the IBM PC, version 1.3 (1990). Distributed by the author, available from the Forth Interest Group. Version 1.4 is now available on GEnie, and worth the extra effort to obtain.

[SEY89] Seyward, H., Elehew, W. R., and Caven, P., LOVE-83Forth for the IBM PC, version 1.20 (1989). A shareware Forth using a five-segment model. Contact Seyward Associates, 265 Scarboro Cres., Scarborough, Ontario M1M 2J7 Canada.

Author's note for web publication: the files formerly available on the GEnie online service are now available from the Forth Interest Group FTP server, ftp://ftp.forth.org/pub/Forth.

Continue with Part 3 | Back to publications page