# Forth language

*Selecting a processor to suit the language, and control structures are subjects of Brian Woodroffe's second article illustrating why he designed his computer around Forth.*

**by B. Woodroffe**

Forth's speed is directly related to how efficiently the computer can execute the NEXT operation. The Table shows how NEXT is coded for some popular eight-bit microprocessors; the 6809 processor executes the operation quickly so a NEXT operation may be included at the end of code routine. This improves performance since the 'JMP NEXT' operation needed for most processors is avoided – in stark contrast to conclusions drawn from one manufacturer's benchmark tests[7].

NEXT is the virtual-machine instruction fetch so the choice of a processor to run Forth on should be dominated by speed and memory costs of the NEXT operation. Further, 6809 registers exactly match those required for Forth as can be seen in List 2. Machine code in the host computer represents the Forth machine, the Y register taking on the role of the Forth program counter. Following examples of simulating the virtual machine, in 6809 machine code, confirm that this processor is well suited to Forth.

### The stack

So far, only the control mechanism by which Forth transfers control from one word to the next has been described, but the language must also control and manipulate data. This, too, is done by means of a stack, but this storage area is known as a data stack, as opposed to the one previously described which is known as the 'return' or 'control' stack. Separation of the stacks simplifies things; normally, data and control operations use the same stack. The stack is further broken down into 'frames' with markers to denote which part is what. In Forth all operators, such as the words + and AND, may remove instructions from the stack, destroy them, manipulate them and push results back onto the stack many times. This has the advantage that operators need not be told where their operands are, which results in less code. A computer operating this form of addressing is known as a zero-address machine, for operand addresses are implicit in the instruction. These words may be in the machine code of the target computer or determined using words already defined.

Using a stack avoids problems caused by parentheses and operator precedence. As far as the computer is concerned the problem is solved, List 3, but programmers used to infix notation may find postfix notation (reverse-Polish notation) difficult, e.g.

| Postfix | Infix |
|---|---|
| 34+56+× | (3+4)×(5+6) |

List 2. Registers of the 6809 suit Forth requirements.

| 6809 register | | Forth usage | |
|---|---|---|---|
| S | stack pointer | RP | return stack pointer |
| U | user stack pointer | SP | data stack pointer |
| Y | index register | IP | instruction pointer |
| X | index register | W | current c.f.a. |
| D | accumulator | – | accumulator |

**Table. Coding and performance analysis of the Forth NEXT operation for popular eight-bit microprocessors.**

| Processor | 6809 | 6800 | Z80/8085 | 8088 | 6502 |
|---|---|---|---|---|---|
| Code | LDX 0,Y++<br>JMP [0,X] | JMP NEXT<br>LDX IP<br>INX<br>INX<br>STX IP<br>LDX 0,X<br>STX W<br>LDX 0,X<br>JMP 0,X | JMP NEXT<br>LDAX B<br>INX B<br>MOV L,A<br>LDAX B<br>INX B<br>MOV H,A<br>MOV E,M<br>INX H<br>MOV D,M<br>XCHG<br>PCHL | JMP NEXT<br>LODS AX<br>MOV BX,AX<br>MOV DX,BX<br>INC DX<br>JMP WORD PTR [BX] | JMP NEXT<br>LDY #1<br>LDY [IP],Y<br>STA W+1<br>DEY<br>LDY [IP],Y<br>STA W<br>CLC<br>LDA IP<br>ADC #2<br>STA IP<br>BCC L<br>INC IP+1<br>L JMP W−1 |
| Memory bytes | 4 | 17 | 14 | 11 | 28 |
| Processor clock cycles | 14 | 44 | 60 | 58 | 43 |
| Normal cycle time (μs) | 1 | 1 | 0.25 | 0.2 | 1 |
| Total time (μs) | 14 | 44 | 15 | 11.6 | 43 |
| Memory-access (ns) | 695 | 530 | 250(Z80) | 450 | 650 |
| Time for 450ns-access memory (μs) | 9 | 37 | 27 | 11.6 | 29.7 |
| Speed relative to 6800* | 4.11 | 1 | 1.37 | 3.19 | 1.25 |

*Value rises proportional to speed.

List 3. Some 6809-code arithmetic routines including add, subtract and two's complement.

```
"+"     FDB $+2
        PULU D
        ADDD 0,U
        STD 0,U
        NEXT
MINUS   FDB $+2
        LDD #0
        SUBD 0,U
        NEXT
@       FDB $+2 (fetch)
        LDD [0,U]
        STD 0,U
        NEXT
!       FDB $+2 (store)
        PULU X
        PULU D
        STD 0,X
        NEXT
DUP     FDB $+2
        LDD 0,U
        PSHU D
        NEXT
OVER    FDB $+2
        LDD 2,U
        PSHU D
        NEXT
SWAP    FDB $+2
        PULU D,X
        EXG D,X
        PSHU D,X
        NEXT
DROP    FDB $+2
        LEAU 2,U
        NEXT
```

NEXT is defined as a macro instruction.

Parameters are also passed between separate lists using the stack. The word consumes as many stack elements as required and pushes back its results. Some defined Forth words for subtracting and doubling the top of the stack respectively are

```
"–"FDB DOCOL    "2*"FDB DOCOL
   FDB MINUS        FDB DUP
   FDB ADD          FDB PLUS
   FDB SEMIS        FDB SEMIS.
```

### Language control structures

As has been shown, Forth passes control from one item in a word to the next and results are calculated. These words can be either machine-code words or pointers to other words. How control may be diverted to form if-then-else or repeat-until structures is the following subject, starting with an explanation of how Forth tests for true or false conditions by simply considering a non-zero value at the top of the data-stack as a true condition. Examples of conditions that create these flags are '0=', '0<', '=' and '<' in the form of code words or Forth words, as appropriate, Lists 4, 5. Diversion of control is carried out by Forth

List 4. Code routines leaving a flag at stack top.

```
0EQUAL  FDB $+2
        LDD #1      assume true (i.e.
                    zero)
        LDX 0,U++   get operand, set
                    6809 flags
        BEQ 0E1
        DECB        was <>0 so set
                    Forth flag
0E1     STD 0,U     put back Forth flag
        NEXT
0LESS   FDB $+2
        LDB #1      prepare true
        LDA 0,U     get sign to A
        BMI 0L1     -?
        CLRB        no, leave false
0L1:    CLRA
        STD 0,U
        NEXT
```

List 5. Forth routines leaving a flag.

```
"="     FDB DOCOL
        FDB SUB
        FDB 0EQUAL
        FDB SEMIS
"<"     FDB DOCOL
        FDB SUB
        FDB 0LESS
        FDB SEMIS
">"     FDB DOCOL
        FDB SWAP
        FDB LESS
        FDB SEMIS
```

words BRANCH and 0BRANCH, the former taking the next storage cell as a branch offset and the latter branching or not depending on the value at the top of the stack. If the flag is false, the threaded-code instruction pointer, ip, is incremented by the offset value contained in the next program storage cell. When the flag is true, this offset is skipped and execution continues with the next word. Controlled loops may also be constructed. Using 'begin . . . until' structures, statements between are executed so long as the flag at the top of the stack remains false. Iterative loop type structures such as '100 TIMES DO' are handled by taking initial and limit loop indexes off the data stack and storing them on the control stack. At the potential end of the loop the current index is incremented and compared with the limit. If the limit is exceeded a branch is executed as described above, otherwise the indexes are deleted and the offset skipped to continue execution, List 6.

List 6. Code for diverting control flow if the flag at the top of the stack is false.

```
0BRANCH: FDB S+2    6809 code
         LDD ,U++   test and delete
                    Forth flag
         BNE 0B1    <>0, branch if true
         LDX 0,Y    get jump offset in
                    X
         LEAY Y,X   add offset
         NEXT
0B1:     LEAY 2,Y   skip over offset
         NEXT
BRANCH   FDB S+2
         LDX 0,Y
         LEAY Y,X
         NEXT
```

```
: ROOTS                  ( stack ..c b a start defining new word 'ROOTS' )
   SWAP MINUS            (        ..c a -b )
   OVER                  (        ..c a -b a )
   DUP +                 (        ..c a -b 2a quicker than 2* )
   /                     (        ..c a -b/2a )
   ROT ROT               (        ..-b/2a c a save -b/2a )
   /                     (        ..-b/2a c/a )
   OVER DUP*             (        ..-b/2a c/a -b/2a*-b/2a )
   +                     (        ..-b/2a b**2/4a-a/c )
   DUP 0<                ( is top less than 0, ie imaginary roots? )
   IF                    ( test flag )
     DROP DROP           ( delete partial results, send <cr><lf> to terminal )
     CR ." imaginary roots" ( and print message )
   ELSE
     CR                  ( real roots, send <crc><lf> to terminal )
     0                   ( convert 16-bit positive number to 32 bits )
     SQRT                ( get back square root )
     OVER OVER +         ( duplicate both parts of answer and get 1st result )
     ." roots are" .     ( print message and first answer )
     ." , and" - .       ( print message and other answer )
   ENDIF                 ( continue execution )
   CR ;                  ( send <cr><lf> and stop compiling return to execution )
```

List 7. Forth code used to calculate the roots of a quadratic equation. The stack is represented across the page with the top of the stack at the right.

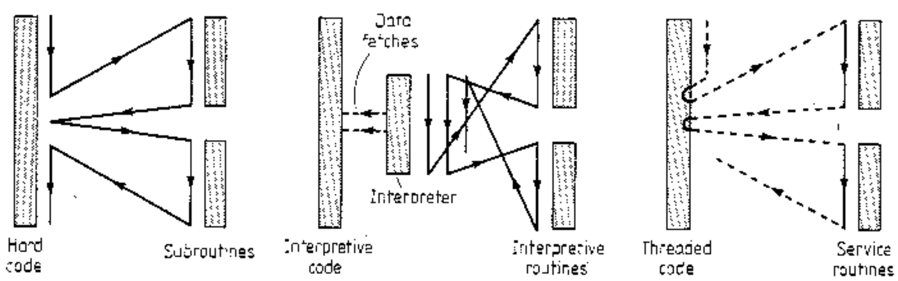
## Using Forth

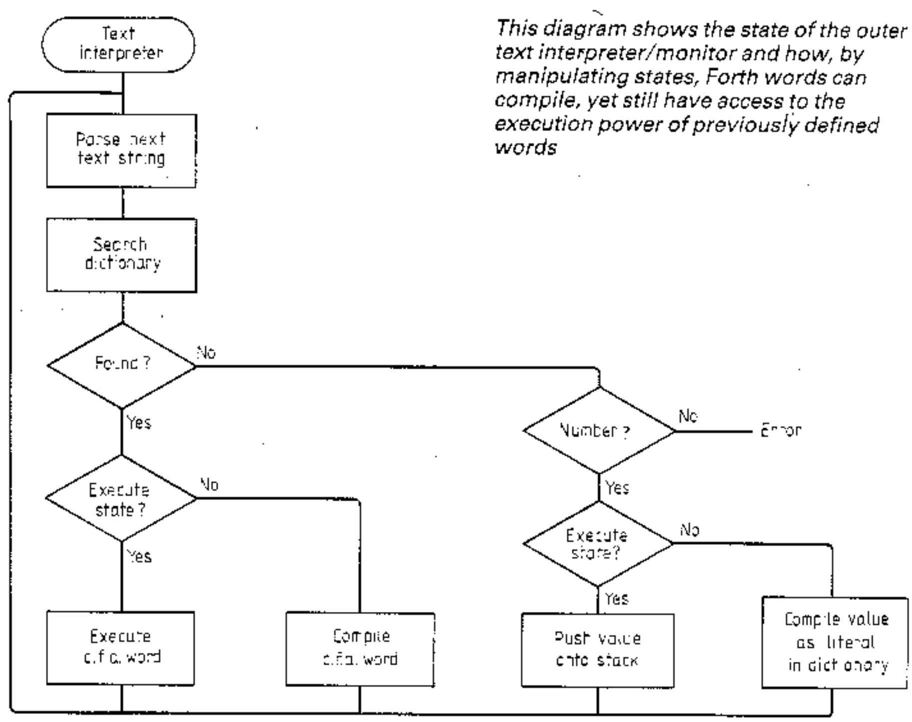
List 7 is an example of a Forth routine for calculating the roots of a quadratic equation, given that the indexes are on the stack. Forth has the shortcoming that it only handles integer arithmetic so non-integer results will be incorrect. The program example illustrates a number of Forth concepts, e.g., stack manipulation, passing parameters and terminal output. Words used in the program are explained in the next article, as are the dictionary and compiler.

## Reference


7. Intel iAPX88 Book, July 1981, appendix pp. 20-36.


*Three flow diagrams compare, from left to right, hard code, interpretive code and threaded code.*




*This diagram shows the state of the outer text interpreter/monitor and how, by manipulating states, Forth words can compile, yet still have access to the execution power of previously defined words*