SEEING FORTH

JACK J. WOEHR - 'JAX' ON GEnie

"Remontons vers les faits moins visibles, mais plus importantes. Nous y verrons le retour á l'âge des Adeptes."

—Louis Pauwels and Jacques Bergier
Le Matin des Magiciens
Editions Gaillimard, 1960

The Grand Adept of Forth was and remains Charles Moore himself, whom some describe as the author of Forth and others as the discoverer of same.

Charles Moore is a tall, smiling, pleasant man in his forties with neat, dark hair and a balding dome which he covers with a tasteful cowboy hat. He also wears cowboy boots and is associated with a firm called Computer Cowboys.

Forth idealizes an imaginary processing unit.

Mr. Moore characterizes himself as "the one you can blame for all this." In a sense he is correct; a wind of freedom blows from the direction of Forth that is most disconcerting to those trapped in jobs which mandate the use of a traditional compiler.

Moore is cryptic when asked to describe his invention. He is a habitual iconoclast, as delighted at bursting the bubbles of his disciples as of his opponents.

"Forth, to me, is more of an approach than a specification for a programming language," he says when asked his opinion of attempts to standardize Forth.

Let us examine that approach.

Forth idealizes an imaginary processing

```
\ scasm32.f ...
\ assembler for SC32 in JForth
\ ©1989 jack j. woehr
\ permission to distribute and use freely granted
\ to Forth Interest Group MEMBERS ONLY !!!
\ pay yer dues, cheapskate!
\ and attend your local FIG Chapter regularly!
\ jax@well.UUCP JAX on GEnie
\ Minimal instruction assembler written in JForth for the Johns Hopkins
\ JPL 32-bit stack machine known as the SC32.
\ references:
\         Silicon Composers, Inc., 32-Bit
\         Stack-Chip Microprocessor Preliminary,
\         4/12/89.
\         ©1989, Silicon Composers, Palo Alto, CA
\ Examples:
\
\       CALL 1234567 ADDRESS ,
\
\ ALU/SHIFT NEXT U3 SOURCE S0 DEST PUSHS-POPR STACK 0 CIN DST<ALU BUSSRC
\       V ALUCOND FL<ALUCOND FLAG S0&SRC ALU ,
\

hex

only forth definitions also

vocabulary SCASM32

also SCASM32 definitions also

\ *** Instruction Logic

\ Instruction Types

00000000 constant call
20000000 constant branch
40000000 constant branch?
60000000 constant ALU/shift
80000000 constant load
A0000000 constant store
C0000000 constant load-addr-low
E0000000 constant load-addr-high
```

```
\ convenient op name aliases

load-addr-low  constant lal
load-addr-high constant lah

\ Next Bit

10000000 constant  #next

\ Src & Dst Bit Patterns

14 constant dst-field
18 constant src-field

00 constant s0
01 constant s1
02 constant s2
03 constant s3
04 constant r0
05 constant r1
06 constant r2
07 constant r3
08 constant u0
09 constant u1
0A constant u2
0B constant u3
0C constant pc
0D constant psw
0E constant zero

( 0F reserved)

\ Stack Bit Patterns

10 constant stack-field

00 constant nop \ This also applies for the Flag Field of the ALU/
Logic ops.
01 constant popr
02 constant pushr

( 03 reserved)

04 constant pops
05 constant pops-popr
06 constant pops-pushr

( 07 reserved )

08 constant pushs
09 constant pushs-popr
0A constant pushs-pushr

( 0B - 0F reserved )

\ ALU/Shift

\ Fields

0F constant subtype-field
0E constant bussrc-field
0A constant alucond-field
08 constant cin-field
07 constant flag-field
00 constant alu-operation-field
```

(Continued on next page.)

unit with an infinitely extensible instruction set. Such a processor not yet existing, Forth is asymptotic to the progress of Forth implementations. So we see that where Moore appears frustratingly vague to his eager hearers, he is actually being explicit.

If Moore is an adept, he must have a lineage. Dr. C.H. Ting, himself a Forth adept, compares the CISC (Complicated Instruction Set) style of Forth with the available academic models and proclaims Moore heir to Von Neumann. Von Neumann and his associates gave contours to serial computation conducted by electronic digital devices which held near-universal sway until recent years. Now the Harvard architecture rears up in belated challenge as we sit on the threshold of the parallel-computation age; but it is significant that the retooling of Von Neumannism inherent in Forth is of an age equal to the Harvard model, and it has progressed to a greater variety of implementations ahead of the evolution of the Harvard model, the latter requiring a much greater silicon investment before its benefits could be made manifest.

Forth, from its inception, has been remarkably easy to implement on a certain level. This was one of its most attractive points to early enthusiasts who found themselves in a race with rapidly changing hardware in the computer explosion of the seventies and early eighties. Forth seems alive; once "life" has been established—once a nucleus of indispensable instructions has been coded—the system awakens and begins to grow beneath the sculpting hand of the programmer.

The real-world emulations of the ideal Forth have culminated in our time with microprocessors specifically designed to execute the fundamental Forth instruction set. Yet Forth itself remains elusive, almost reticent, much like Moore himself. Perhaps we have come as close to the Muse as she will allow us to approach in this Digital Dispensation, and we shall now be forced to take refuge in standards, and in technique.

(Multitasking, continued from page 18.)

```
20 CONSTANT SCORE
```

The 20 (like all numbers) is placed on the stack, then CONSTANT is activated. CONSTANT is a defining word, and a defining word is always followed by a name to give to the 'thing' it is to define (in this case SCORE). CONSTANT places this name in the dictionary, reserves space for one number, and installs the number on the stack in this space. This completes the building; it then adds instructions about the run-time behavior of SCORE.[1] All constants have the same run-time behavior, which is to place on top of the stack the number stored as part of their structure.

CONSTANT could have been defined using the words CREATE (which starts the instructions on what to build) and DOES> (which starts the list of run-time behavior instructions) as follows:[2]

```
: CONSTANT
  CREATE , DOES> @  ;
```

CREATE starts the building process by adding a name to the dictionary, using the next word in the input string (the word after CONSTANT) for the name. The , (comma) reserves two bytes and initializes them by storing the number from the top of the stack at the end of the dictionary and advancing the dictionary pointer (the pointer to the next available free space at the end of the dictionary). DOES> starts the series of run-time behavior instructions with the minimum action, which is to return the address of the first thing CREATE built after the name. In the case of a constant, this is the address of the stored value, so the only other action needed is to read the value stored there with a normal fetch.

Returning to our new defining word, CREATE and DOES> are used to define the two functional parts of TIMER, as shown in Figure One. TIMER builds a name and the space for two 16-bit variables, the user value UV and the internal value IV. The run-time behavior given to the word defined by TIMER is to put the address of the user variable on the stack and read the real-time clock. Then the last value read is subtracted and the user variable is corrected.

[1]To be picky, F83 does not place the instructions there, it places a pointer to instructions. However, this is a point of implementation detail that can be ignored here.

[2]It isn't in most systems—it is defined as a primitive in the interests of speed—but it could have been.

(Continued.)

```
\ SubT Field Bit Patterns

0 constant alu/logic
1 constant shift/step

\ BusSrc Field Bit Patterns

0 constant dst<fl
1 constant dst<alu

\ ALU Condition Field Bit Patterns

( 00 constant 0) \ These conveniently are unambiguously themselves!
( 01 constant 1) \ Likewise with the Cin instructions.

02 constant V
03 constant _V
04 constant _((NxV)|Z)
05 constant (NxV)|Z
06 constant N
07 constant _N
08 constant Z
09 constant _Z
0A constant _(_C|Z)
0B constant _C|Z
0C constant NxV
0D constant _(NxV)
0E constant C  \ watch out with the hex numbers, always precede w/ 0 !!
0F constant _C

\ Cin Field Bit Patterns

( 00 constant 0) \ Conveniently, unambiguously themselves ...
( 01 constant 1) \ ... as w/ ALU Conditions above

02 constant FL'
03 constant _FL'

\ Flag Field Bit Patterns

( 0 constant nop ) \ Same as above in the Stack Field Bit Patterns

1 constant fl<alucond

\ ALU Operations

15 constant _(s0&src)
17 constant s0|_src
1D constant _s0|src
1F constant s0|src
20 constant 0
21 constant _s0
22 constant negl
23 constant s0
24 constant _src
2C constant src
2F constant s0xsrc
41 constant _s0+cin
43 constant s0+cin
44 constant _src+cin
45 constant _s0+_src+cin
46 constant _src-_cin
47 constant s0-src-_cin
49 constant _s0-_cin
```

```
4B constant s0-_cin
4C constant src+cin
4D constant src-s0-_cin
4E constant src-_cin
4F constant s0+src+cin
55 constant s0&src
57 constant _s0&src
5D constant S0&_src
5F constant _(s0|src)
6F constant _(s0xsrc)

\ Shift Instructions
\ Shift Fields

5 constant shift-field
4 constant shiftin-field
2 constant step-field
1 constant flagin-field

\ Shift Field Bit Patterns

\ Shift

( 0 constant nop)\ Once again, this is conveniently already defined

1 constant right
2 constant left

\ Shiftin

0 constant <alucond
1 constant <FL'

\ Step

0 constant step:src+cin
1 constant step:src-s0-cin
2 constant step:s0+src+cin(FL')
3 constant step:s0+src+cin(_FL')

\ Flagin

( 0 constant <alucond)        \ already defined above in Shiftin

1 constant <shiftoutput

\ *** Forming Instructions

\ Control Flow

: address      \ control-instruction address -- instruction'
      1FFFFFFF and or ;

\ Shifting Bit Pattern to Instruction Field

: shift-into-field    \ instruction bits field -- instruction'
      << or ;

\ Set Next Bit

: next \ instruction -- instruction'
      #next or ;

\ Src & Dst
```

(Continued on next page.)

The last value read is then updated, and we exit with the address of the user variable still on the stack.

A definition for (READ_CLOCK) to suit the IBM PC and F83 is given in Figure Two; it returns a number which is incremented 1193180/65536 times per second (a strange number, granted, but that is how IBM designed it). After this (or a substitute that suits your hardware) and TIMER are entered, the following can be used as a test:

```
TIMER CLOCK
: TEST
  BEGIN CLOCK
  @ DUP U. 0<= UNTIL
  ." Timed out!"  ;
```

Then, if you enter the line:

```
180 CLOCK ! TEST
```

a series of decreasing numbers (the user variable) will be printed—which lasts just under ten seconds on my system—before the "Timed out!" message appears.

To complete the task, the multitasker must be used. Multitasking has been part of almost all versions of Forth except fig-FORTH, the first of the public-domain versions. It is not, however, part of the standard. Unlike time-sliced multitasking, in which each task has to surrender the processor to the next task after a pre-determined time interval whether it "likes" it or not, F83 (like most versions of Forth) uses a cooperative scheme. In this, a task passes control only when it is ready, thus simplifying the job of keeping track of who is doing what, and making the task interchange very fast. The cost is that one cannot predict reliably exactly when task interchange will take place, and if one task gets into an endless loop that does not contain the voluntary transfer word PAUSE, everything else stops for good. This latter case is the fault of the programmer, not the language. With care, the task latency time can be made very small, especially since all F83 words having to do with human interaction—and whose execution times are, therefore, unpredictable—already contain the task interchange word PAUSE.

Different tasks share all resources other than the stacks, although a group of variables has to be assigned to each task to keep a record of internal processor information during the time when other tasks have control. The tasks involved in the multitasking are linked into a circular list, each receiving control from the preceding one

and passing it to the succeeding one. Each task on the list can be active or asleep. In the latter state, it passes control on as soon as it receives it. Otherwise, it executes until the word PAUSE is encountered, either explicitly or as part of an input or output word. A task can be activated by use of the word AWAKE. Multitasking can be turned off or on by the words MULTI and SINGLE. If absolutely essential, these could be used within a task if, for some reason, the task had to retain control for a certain period even though some input or output words (which would normally cause a task interchange) are to be executed.

The user-interface words involved in multitasking in F83 are given in Figure Three. The use of these words is demonstrated in Figure Four. First, we use the special defining word TASK: to build a task that prints 20 asterisks on the screen and link it into the round robin (which at the moment only consists of the outer interpreter, which is handling our keyboard input).

Note that the STOP is essential. Otherwise, when 20 asterisks have been printed and the task is over, disaster will strike as the computer tries to execute the stack for PRINT*S! Also note that we have an inner loop just to slow things down a bit, otherwise all the asterisks will appear before we have a chance to do anything. This inner loop is a good neighbor and gives everyone else a go by, including the word PAUSE in the loop.

Nothing unusual happens on the screen, as we have not turned on multitasking. We can change that easily by entering MULTI. Still no asterisk appears; this is because when a task is built and linked, it is placed in the sleeping condition. Hence, we must enter PRINT*S WAKE to wake it up. We can carry on typing at the keyboard, but on the screen our input will appear mixed with asterisks. Well, it will until 20 asterisks are printed, then things will return to normal.

Entering PRINT*S WAKE again will *not* cause another batch of asterisks to appear. The task will resume with the (nonexistent) word after STOP, and disaster will strike. As it stands, PRINT*S is a one-shot model only!

If, during this batch of asterisks, we had managed to type
PRINT*S SLEEP <cr>

the output of asterisks would have stopped at once. The same would happen if we were

(Continued.)

```
: source       \ instruction register -- instruction'
       src-field shift-into-field ;

: dest  \ instruction register -- instruction'
       dst-field shift-into-field ;

\ Stack Action

: stack\ instruction stackop -- instruction'
       stack-field shift-into-field ;

\ Load and Store

: zero-extended-offset  \ instruction addr-offset -- instruction'
       0ffff and or ;

: zeo   \ i a-o -- i'  \ convenient alias
       zero-extended-offset ;

\ ALU/Shift Instructions

\ ALU Operations

: subtype      \ instruction subtype -- instruction'
       subtype-field shift-into-field ;

: bussrc       \ instruction bussrc -- instruction'
       bussrc-field shift-into-field ;

: alucond      \ instruction type -- instruction'
       alucond-field shift-into-field ;

: cin   \ instruction type -- instruction'
       cin-field shift-into-field ;

: flag  \ instruction type -- instruction'
       flag-field shift-into-field ;

: alu   \ instruction operation --
       alu-operation-field shift-into-field ;

\ Shift Operations

: shift\ instruction shift-op -- instruction'
       shift-field shift-into-field ;

: shiftin      \ instruction shifter-input -- instruction'
       shiftin-field shift-into-field ;

: step  \ instruction step-op -- instruction'
       step-field shift-into-field ;

: flagin       \ instructions flagin --
       flagin-field shift-into-field ;

\ *** Assembly Buffer Management

   4 constant buff-hdr-size  \ link to next allocated buffer | 0
1000 constant buff-size      \ each buffer same size
   4 constant opsize         \ each op is a longword on SC32

variable 1st-buffer          \ as it says
variable last-buffer         \ latest allocated buffer
```

```
variable asm-ptr   \ the "dictionary pointer" for our assembler

: asm-ptr.init \ --    \ start off set to zero
      asm-ptr off ;

: here         \ -- next-available-assembly-relative-address
      asm-ptr @ ;

: (there)              \ addr -- offset-in-any-buffer
      buff-size mod ;

: there        \ here -- real-address
      (there) last-buffer @ buff-hdr-size + + ;

\ allocate $1004 bytes for assembly and a buffer-linking header.

: buff-err?    \ --
      abort" No Buffer Memory" ;

: get-buffer   \ -- absmemaddr|0
      [ buff-hdr-size buff-size + ] literal MEMF_CLEAR
      [ forth ] exec? call exec_lib allocmem [ scasm32 ] ;

: free-buffer  \ reladdr --
      >abs
      [ buff-hdr-size buff-size + ] literal
      [ forth ] call exec_lib freemem drop [ scasm32 ] ;

: free-all-buffers     \ --
      1st-buffer @
      begin
      dup @ swap free-buffer
      dup 0=
      until drop ;

: get-1st-buffer       \ --
      get-buffer dup
      if >rel dup 1st-buffer ! last-buffer !
      else true buff-err?
      then ;

: get-subsequent-buffer       \ --
      get-buffer dup
      if >rel dup last-buffer @ ! last-buffer !
      else true buff-err?
      then ;

: manage-buffers       \ --
      here (there) 0= here 0> and
      if get-subsequent-buffer then ;

\ *** Output File Handling

create out-filename 100 allot
variable out-fileptr
variable outfile-buff

: outfile-err? \ t|f --
      abort" Couldn't Open Output File" ;

: writefile-err?       \ t|f --
      abort" Error While Writing Output File" ;

: out-filename.default         \ --
      0" ram:scasm32.out" 0count 1+ out-filename swap cmove ;
```

(Continued.)

to type SINGLE, although this would "lock" the processor on the one task in which it occurred. The other tasks would not be asleep, but would start as soon as MULTI was issued, without having to be awakened. If the task in which the SINGLE command occurred didn't have MULTI later, or had no way of inputting the MULTI command (i.e., didn't involve the outer interpreter), there would be no way short of a system reset to regain control.

BACKGROUND: adds a new task into the round robin. How can one remove one that is no longer needed? The simple answer is, you cannot. You can assign new instructions to the old task name, but you must not FORGET the old task, as the circular list would be broken and disaster would strike when the processor tried to move around it. To assign a new set of instructions, the word ACTIVATE is used to associate the new instructions with the old name and wake it up immediately. We will use ACTIVATE to assign a new version to PRINT*S, one which will be reusable. It is essential to realize that ACTIVATE can only be used in a colon definition, because of the way it handles the return stack; attempts to use it interactively will cause a system crash.

The version of our example shown in Figure Five is much better; when awakened after running to completion, it just loops and runs again. The original version is replaced by this improvement just by executing NEW-PRINT*S.

For speed, you should have only enough tasks in the circular list to service the maximum number that must run concurrently, and use task redefinition to move tasks into and out of the list. Task interchange is fast, but it doesn't take zero time.

Vast possibilities arise from the ability to run tasks, freeze them, and later restart them; for tasks to start and stop other tasks; and for tasks to be able to grab all the processing power for time-critical routines by issuing SINGLE and, afterwards, MULTI. However, the virtues of simplicity are nowhere stronger than in multitasking. All tasks must cooperate, and the problem of keeping in mind the possible effects of all combinations of events rapidly becomes daunting.

I do not like the name used for the word BACKGROUND:, as it suggests to me a master-slave relationship rather than a cooperative arrangement. Also, the allocation of 400 bytes is not always ideal, although 100 bytes of whatever quantity you allocate

will go for the return stack and the rest for the data stack (unless you change TASK:). Of course, in the spirit of Forth, if you don't like it, change it.

The formal definition of BACKGROUND: is given in Figure Six. It is a short definition, and it is easy to modify the number of bytes required for the two stacks. After modification, it could be saved as MULTITASK or COOP-MEMBER or any other name which takes your fancy. Similarly, I would prefer IS-NOW for ACTIVATE, but that is a personal matter. If you wish to change the name, it can easily be done with:

```
: IS-NOW  ACTIVATE ;
```

which make the two names mean the same. If you wish to allocate more or less than 100 bytes to the return stack, you will need to redefine TASK: and then use your new definition in a new version of BACKGROUND:. To find where to change TASK:, decompile (e.g., SEE TASK:) and then re-enter it, changing the 100 just past halfway through the definition to whatever number you prefer. The data stack will get the difference between what you put in your version of TASK: and the total allocation for stacks you define in your version of BACKGROUND:.

Interrupts will be needed for very rapidly occurring events but, for most other situations, the timer and multitasker described above will give you real-time control. For further detail on the multitasker in F83, see the shadow screens of the source code or chapter 23 of *Inside F83* by C.H. Ting.

Tim Hendtlass is principal lecturer in scientific instrumentation in the physics department of the Swinburne Institute of Technology. He discovered Forth in 1980, used it in more and more instrumentation, and introduced it as the laboratory language for all undergraduate students majoring in scientific instrumentation.

(Continued.)

```
: open-outfile \ --    \ what the heck is going on here?
       out-filename new Ofopen
       dup out-fileptr !
       0= outfile-err?
       out-fileptr @ fclose
       out-filename Ofopen dup
       0= outfile-err?
       out-fileptr ! ;

: close-outfile       \ --
       out-fileptr @ fclose ;

: (get-outfile-buff)  \ -- 0|abs_addr
       here MEMF_CLEAR
       [ forth ] exec? call exec_lib allocmem [ scasm32 ] ;

: get-outfile-buff    \ --
       (get-outfile-buff) dup 0= abort" Couldn't Get Outfile Buff"
       >rel outfile-buff ! ;

: free-outfile-buff                    \ --
       outfile-buff @ >abs here
       [ forth ] call exec_lib freemem drop [ scasm32 ] ;

: fill-outfile-buff                    \ --
       here buff-size /mod             \ how many 4k buffs to consoli-
date?
       outfile-buff @                  \ get the file buffer
       1st-buffer @ rot                \ get the first asm buffer
       0 do                            \ JForth DO is a ?DO
              dup buff-hdr-size +      \ move to data area
              2 pick                   \ get file buffer address
              buff-size cmove>         \ move data
              @ swap buff-size + swap      \ get next data buff,
inc file buff adr
         loop
       rot dup                         \ is there a modulus remaining?
       if                              \ yes, copy rest of data
              swap buff-hdr-size + -rot cmove>
       else
              2drop drop               \ drop data-adr filebuf-adr 0ct
       then ;

: write-outfile       \ --
       out-fileptr @ outfile-buff @ here fwrite
       0= abort" Error Writing Assembly to File" ;

: save-assembly       \ --
       open-outfile
       get-outfile-buff fill-outfile-buff
       write-outfile close-outfile free-outfile-buff ;

\ *** Assemble to Memory

\ all SC32 operands are longwords

: ,           \ longword --
       manage-buffers here there ! 4 asm-ptr +! ;

\ *** Initialization

: scasm32.init \ --
       scasm32 get-1st-buffer asm-ptr.init out-filename.default ;

: wrapup              \ --
       save-assembly free-all-buffers ;

decimal
```