ISSN 0265-5195

Forthwrite FIGUK

Issue 106 April 2000

John Tasgal
0161 7739365
john@tcl.prestel.co.uk

An Introduction to Machine Forth

John Tasgal

There has been a lot of interest in Chuck Moore's recent work on special processors and the languages that go with them (see issue June 1999 for an interview with Moore). These are Forths that deviate from the classical model to match the hardware more closely. The differences challenge our assumptions about standard Forth; could it become both simpler and better?

John Tasgal has researched both Machine Forth and Color Forth and expounds these differences. Alongside the Color Forth article (in the next issue) is a commentary on some of Chuck's published code, showing how complex code can be written with a simpler Forth.

Machine Forth (MF) is a development, principally by Charles Moore and Jeff Fox, of classical Forth. Its aim is to simplify both the design of stack chips and the Forth-style languages they use. It is a low-level Forth closely mapped to the underlying hardware.

There are two quite separate parts to this language:

The first part is a core which is the instruction set of a MISC (Minimal Instruction Set Computing) chip. The second part is an extension to the instruction set to allow word and dictionary building etc.

As far as I know there is no standard and so I have chosen to use the Ultra Technology F21 chip as the 'reference' for the MF instruction set. This article describes the 'programming model' of a MISC chip. I only describe hardware where relevant.

Notation

T(n)	The n'th bit of register T
T(n1 .. n0)	A bitfield in register T from bit n1 down to bit n0

Phrases

To present the structural templates below in compact form these abbreviations are used for phrases, that is, a sequence of tokens :

`flag?`	A phrase which leaves a value in T(19..0) for use by `IF`
`carry?`	A phrase which leaves a value in T(20) for use by `-IF`

`<tt`	The phrase executed when `IF` is true
`<ff`	The phrase executed when `IF` is false

The MISC Chip

- There are 5 registers, 2 circular stacks, and a 5-bit opcode with 27 instructions decoded.
- All on-chip registers are 21 bits wide.
- The MSB, bit 20, is used for memory control for instructions which access external memory; as carry for the add instruction; and as an ordinary bit for the others.
- `IF` reads bits 19..0 and jumps if they are false. It does not pop the stack (unlike a classical `IF`). It is therefore called 'non-destructive'.
- `-IF` jumps if bit 20 is false. This too is non-destructive.
- `2/` on the F21 shifts all bits T(20..0) right. It can therefore be used either as `2/` or `U2/`, or for multiple-precision arithmetic.

The Registers

- **PC** The Program counter
- **A** The Address register for memory access
- **T** Top of data stack, the implied operand for arithmetic, logic and `IF` instructions
- **S** The 'subtop' register, the second on the data stack.
- **R** Top of return stack

The Circular Stacks

- **The Data Stack (S2 .. S11)** A 16-element circular stack below T and S
- **The Return Stack (R1 .. R10)** A 16-element circular stack below R

The Instruction Set

Control

- **`ELSE`** Unconditional jump
- **`IF`** Non-Destructive `IF`. Jump if T(19..0) is false (leaves stack untouched)
- **`-IF`** Non-destructive jump if-carry-false

- **CALL** A Subroutine call. Push PC+1 to R
- **RET** Return from Subroutine. Pop R to PC

A Register

A	**(-- A ; T = A)**	Push A to T
@A	**(-- n0 ; T = ^A)**	Fetch contents of memory at address A and push to T.
@A+	**(-- n0 ; T = ^A, A=A+1)**	Fetch A and push to T. Increment A. ('Auto Post-Increment')
!A	**(n0 -- ; mem(A) = n0)**	Pop T to memory at address A
!A+	**(n0 -- ; mem(A) = n0, A=A+1)**	Pop T to memory at address A. Increment A
A!	**(a0 -- ; A = T)**	Pop T to A

R Register and the Return Stack

POP	(-- r0 ; r0 -R- ; T = R)	Pop R and push to T
PUSH	(n0 -- ; -R- n0 ; R = T)	Pop T and push to R
@R+	(-- n0 ; T = ^R, R=R+1)	Fetch from address in R, push to T. Increment R
!R+	(n0 -- ; mem(R) = n0, R=R+1)	Pop T to memory at address R. Increment R

Data Stack Manipulation

DUP	(n0 -- n0 n0)	Push T to T
DROP	(n0 --)	Pop T
OVER	(n1 n0 -- n1 n0 n1)	Push S to T

Arithmetic

+	(n1 n0 -- n0' ; T = T + S)	Add S to T.
+*	(n1 n0 -- n1 n0' ; T = T + S {T(0)=1})	If T(0) is true, add S to T non-destructively. A multiply step.
	(n1 n0 -- n1 n0 ; {T(0)=0})	If T(0) is false, do nothing.

Bitwise

▪ COM	(n0 -- n0' ; T = NOT(T))	Complement T. Invert each bit.
▪ AND	(n1 n0 -- n0' ; T = S AND T)	AND S to T
▪ -OR	(n1 n0 -- n0' ; T = S XOR T)	Exclusive OR S to T
▪ 2*	(n0 -- n0' ; T = T * 2)	Shift Left one bit. Write 0 to T(0)
▪ 2/	(n0 -- n0' ; T = T div 2)	Shift Right one bit. WriteT(20..1) to T(19..0). Write 0 to T(20).

Miscellaneous

▪ #	(-- n0 , | <number)	Fetch a number from PC+1 and push to T. Increment PC .
▪ NOP	( )	Do nothing for 1 cycle.

The Extensions

Very few words need to be added to an assembler based on the above instruction set to produce a working Forth system. The main categories are:

Definitions

▪ `:`	Colon starts a new definition
▪ `;`	Return. Does *not* end a definition
▪ `CREATE ... DOES`	To allow new types
▪ `CODE ... ENDCODE`	For machine code

Control Structures

These structures have the same meanings as Classical Forth but the flag/carry remain on the stack after execution.

▪ `flag? IF <tt THEN <ff`	If flag? is true execute <tt
▪ `carry? -IF <tt THEN <ff`	If carry? is true execute <tt
▪ `flag? IF <tt ELSE <ff THEN`	If flag? is true execute <tt, else execute <ff
▪ `carry? -IF <tt ELSE <ff THEN`	If carry? is true execute <tt, else execute <ff

▪ **`( index ) BEGIN ... NEXT`**	A loop with an single index
▪ **`BEGIN  flag?  WHILE <tt REPEAT`**	While flag? is true execute <tt
▪ **`BEGIN carry? -WHILE <tt REPEAT`**	While carry is true execute <tt
▪ **`BEGIN ...  flag?  UNTIL`**	Loop until flag? is true
▪ **`BEGIN ... carry? -UNTIL`**	Loop until carry? is true

These allow words of various types to be defined; a dictionary to be built; and for the control of program flow. Numerous other words for arithmetic, logic, and Operating System functions can then be added to this extension.

Differences From Classical Forth

The Semicolon
This doesn't end a definition; it means simply 'return'. Definitions run into one another.

The Address Registers
This moves addressing from the the data stack to a register, either A or R.
Both registers also have auto-post-incrementing instructions.
This changes the style of Forth as pointer arithmetic becomes the method of choice over the use of `DO ... LOOP`'s with indexes.

Non-Destructive Conditionals
In Classical Forth, `IF` destroys the top of stack. However in Machine Forth `IF`, and therefore all the conditionals based on it, are non-destructive. This removes the need to use `DUP` when conditionals repeatedly test a flag.

But, it may lead to more use of `DROP` to remove a flag which would have been destroyed by a conventional conditional. This suggests that the behaviour of other words and if necessary the program structure itself should be adapted to optimise the use of non-destructive conditionals, rather than simply copying a program written using the destructive versions.

This is another example of a change in programming style.

Tail-Recursion Optimisation
In any definition the return action of the word before a semicolon, and of the semicolon itself, can always be compiled into a single return.

```
        word1 ..... lastword ;
```

As nothing happens between **`lastword`** returning and '**`;`**' returning, the **`lastword`** return is superfluous.

A more elaborate example is the recursive call at the end of a `WHILE` loop. If we have a series of nested calls then the last instruction is in each case a return. At runtime this produces '**`; ; ; ; ;`**' viz. a sequence of returns.

The point is that when these calls unwind all that happens is that a sequence of returns are executed, one after the other. Nothing is done between them. The only necessary return is the first one pushed onto the return stack (and so the last to be executed).

Removing these superfluous returns is known as *tail-recursion optimisation*. Most Machine Forth compilers (and also Color Forth) contain a 'tail-recursion optimiser'.

Two syntaxes are currently in use to indicate that this optimisation is to be carried out:

- A special token '**`-;`**' (hyphen semicolon)
- A *smart semicolon,* which involves recognising the '**`lastword ;`**' pattern.

Tail-recursion optimisation is achieved through a compiler optimisation, and also by the syntax itself. The syntax is so designed that the programmer, simply by writing a semicolon after a recursive jump, causes the compiler to build a single return instead of nested returns. Therefore nested returns are eliminated *at the design stage* through a syntactical feature.

This is really a very unusual and elegant approach to this problem.

Next Issue
The remaining two articles in this series appear in the next issue of Forthwrite, describing Charles Moore's newest Forth, Color Forth, which builds on Machine Forth. This is followed by a detailed commentary on some of Chuck's Color Forth code to see how it is used in practice.