A MACHINE FORTH SPECIFICATION BASED ON THE PSC1000 CAPABILITIES

Federico de Ceballos
Universidad de Cantabria
fcebcab@ciccp.es

November, 2000

Abstract

Machine Forth, a new Virtual Machine model, has recently been proposed as an alternative to ANS-Forth plus assembler for different processors. MF is the native language of the F21 processor and is said to be easily adaptable to other hardware.

This paper studies the convenience of using the PSC1000 instruction set (more powerful than the one in the F21) as a model for a VM. The model is compared with ANS-Forth both in the PSC1000 and in a conventional processor.

Introduction

This paper presents a set of additional instructions for a Forth Virtual Machine taken from the PSC1000 native instruction set. The resulting model is labelled Machine Forth as is follows the trend initiated by Machine Forth for the F21 of exporting capabilities of a particular processor to common usage.

The original specification of Machine Forth, as laid down be Chuck Moore and Jeff Fox, can be found at [1]. A commented description is also available at [3]. In order to fully understand this paper, some knowledge of both Forth and MF is assumed.

Reuben Thomas [4] has made some effort in porting the F21 MF to an ARM processor. By doing so, he found some advantages and also some limitations in the coding. The intent of this paper is completely different, as we shall only study the effect of using additional registers without limiting the usage of normal Forth words (that are available in the PSC1000 but not in the F21).

The Machine Forth Model

In this paper we shall discuss the benefits of using three additional registers as user registers in the Virtual Machine. Further optimisations would be available if we considered some subtle differences between the PSC1000 instruction set and the Forth Virtual Machine[1].

The address register (*A*) allows indirect memory access. It has a post-increment version (as in the F21) and also a pre-decrement one.

[1] The complete list of instructions available in the PSC1000 can be found in [2].

The top of the return stack (*R*) can also be used as an auxiliary address register with the same modes of operation as *A*.

An index register (*I*) allows fast loops using FOR..NEXT as in Chuck Moore's cmForth. Complex loops as DO..LOOP are not used.

Table 1 shows the words we shall be using, where *T* is the top of the data stack, *[x]* means an indirect memory access and *Dpush*, *Dpop*, *Rpush* and *Rpop* mean pushing / popping a register to / from the Data stack or the Return Stack.

@R+ Dpush(T) T=[R] R=R+cell @-R Dpush(T) R=R-cell T=[R] @R Dpush(T) T=[R] !R+ [R]=T R=R+cell Dpop(T) !-R R=R-cell [R]=T Dpop(T) !R [R]=T Dpop(T,Data) R@ Dpush(T) T=R R! R=T Dpop(T)	@A+ Dpush(T) T=[A] A=A+cell @-A Dpush(T) A=A-cell T=[A] @A Dpush(T) T=[A] !A+ [A]=T A=A+cell Dpop(T) !-A A=A-cell [A]=T Dpop(T) !A [A]=T Dpop(T) A@ Dpush(T) T=A A! A=T Dpop(T)
I@ Dpush(T) T=I I! I=T Dpop(T)	>R Rpush(R) R=T Dpop(T) R> Dpush(T) T=R Rpop(R)
n FOR .. NEXT *is used to repeat a block of code.* FOR *puts* n *into* I*, which gets decremented at* NEXT*. The loop is exited when* I *equals zero.*	

Table 1

These words may seem a lot, and their names can be daunting at first. However, they are quite orthogonal and their meaning can be derived from a few simple rules:

- @ means fetch and *!* means store.
- A prefix indicates that a register (*A*, *R* or *I*) has to be used instead of memory.
- A suffix indicates the source register (*A* or *R*) for an indirect memory access. This register can be pre-decremented, post-incremented or left as it is.

Additionally, some convenience words are defined in terms of the previous ones.

```
: +A! ( n -- )     A@ + A! ;
: A>R ( R: -- n )   A@ >R ;
: R>A ( R: n -- )   R> A! ;
: I>R ( R: -- n )   I@ >R ;
: R>I ( R: n -- )   R> I! ;
```

Table 2

The first word allows moving though memory in intervals different from one cell.

The last four words allow saving and restoring *A* and *I* into the return stack. I preferred not to save the FOR counter automatically as this would mean that the top of the return stack would not be available for indirect memory access. Even if this technique forces the user to do some manual housekeeping, I believe it doesn't impair legibility.

Programming Style

The address registers allow the programmer to access data sequentially without the current address taking a precious place near the top of the stack. Its effect is more important when several addresses are being used concurrently.

As an example, consider the two versions for the inner product in matrix multiplication shown in table 3.

```
100 CONSTANT SIDE
SIDE CELLS CONSTANT SIDE-SIZE

: INNER-PRODUCT ( a[row][*] b[*][column] -- n )
   0                                  ( a  b  result )
   SIDE 0 DO
      >R                              ( a  b  ) ( R: result )
      OVER @ OVER @ * R> + >R         ( a  b  ) ( R: result' )
      SWAP CELL+                      ( b  a' ) ( R: result' )
      SWAP SIDE-SIZE +                ( a' b' ) ( R: result' )
      R>                              ( a' b' result' )
   LOOP NIP NIP ;

: INNER-PRODUCT ( a[row][*] b[*][column] -- n )
   I>R A>R  A! >R  0
   SIDE FOR  @R+ @A * +  SIDE-SIZE +A!  NEXT
   R> DROP R>A R>I ;
```

Table 3

In order to calculate an element in the result matrix, each element in a row of the first matrix has to be multiplied by the corresponding one in a column of the second matrix. Therefore, we need to advance a pointer one cell a time to move in through a row in the first matrix. Another pointer advances **side** cells at a time through a column in the second matrix.

In the first solution (ANS) both pointers are placed on top of the stack. To avoid too much stack gymnastics, the aggregate of all products is kept on the return stack.

The second solution (MF) avoids having the pointers in the stack. One is kept in the address register and the other on top of the return register. The data stack is now free to hold the aggregate of the different products. This results in code that is both simpler and, in my opinion, easier to understand.

One possibility available in Forth is the use of the loop counter as a pointer instead of an index. This approach sometimes complicates the code, but can be used to good advantage in lots of cases. As an example, see the matrix (or vector) addition algorithm shown in table 4 next page.

In the first version (ANS) the current addresses of the two source matrices are kept in the two top elements of the data stack, while the one of the destination matrix is in the loop index. At most, only four elements are kept in the data stack at the same time, which gives a clean code.

The second version (MF) uses *A* and *R* to hold the address of the sources matrices and the data stack for the destination address. Only three elements are at most in the data stack, giving an even cleaner code.

```
100 CONSTANT SIDE
SIDE DUP * CONSTANT #ELEMENTS
#ELEMENTS CELLS CONSTANT MATRIX-SIZE

: MATRIX+ ( a b c -- )
   MATRIX-SIZE BOUNDS DO
      OVER @ OVER @ + I !  \ stores sum
      CELL+ SWAP CELL+     \ watch the trick!  ( a b -- b' a' )
   CELL +LOOP
   2DROP ;

: MATRIX+ ( a b c -- )
   I>R A>R  SWAP A! SWAP >R
   #ELEMENTS FOR  @R+ @A+ + OVER ! CELL+  NEXT
   DROP R> DROP  R>A R>I ;
```

Table 4

Benchmark Results

This section provides the result of some benchmarks[1] with both ANS-Forth and Machine Forth code versions.

Processors Used

Two completely different systems have been used for these tests.

The first is a PSC1000 processor in a R2kup-b board (from Denison Mayes Group). It runs at 32 MHz (the processor doubles this internally) and is provided with 12 ns RAM. The PSC1000 is an advanced processor with single cycle capabilities without pipelines or instruction cache. It has two hardware stacks and is ideally suited to run Forth language code.

The second is an Intel 80486 DX processor in a SVE1 card for an AS990 system (from Siemens). It runs 32 MHz and is equipped with 25 ns RAM. The 80486 contains a cache made of 4 kB data and 4 kB code. It also has a prefetch queue and a pipeline that allows one instruction per cycle.

Both processors are true 32 bits. However, the 80486 is running in real mode (16 bits).

Compilers Used

For comparison purposes, the following compilers have been used:

SwiftX[2]	An ANS-Forth optimising compiler for the PSC1000.
MF 1000	A minimal compiler for the PSC1000 built by the author that covers the model presented here.
F 486	A direct threaded compiler for the 80486, built by the author[3].
MF 486	A modification of the above for the Machine Forth model.
ForthCMP[1]	An ANS-Forth optimising compiler for the 80x86 family.

[1] Complete source code is available from the author.
[2] This compiler is available from Forth Inc. For these tests the trial version has been used.
[3] It has been developed around the eForth model, especially the indirect threaded version with metacompiler by Bill Muench.

The code produced by SwiftX and MF 1000 is roughly of the same quality.

F 486 is a *bona fide* example of a DTC with some optimisations (TOS and loop counter in registers and lots of code words taking into account the 80486 timings). MF 486 only differs from it in the minimum details to cover MF. It keeps the address register in a hardware register but the top of the hardware stack is in memory (as no more index registers are available).

ForthCMP is used to compare DTC execution with the produce of a high quality compiler.

Adding Natural Numbers.

The first example adds the first 25000 positive integers[2]. Its purpose is to measure the loop overhead.

Swift X	MF 1000	F 486	MF 486	ForthCMP
6.25 ms	1.17 ms	25.0 ms	24.22 ms	6.25 ms
32 bytes	16 bytes	26 bytes	24 bytes	22 bytes

Table 5

The result of the MF 1000 is especially good, as the resulting loop is so small that is can be coded into a simple cell, so repeated memory access for code is not needed.

The Fibbonacci Numbers

The next example calculates the 30th element in the Fibbonacci sequence, using a recursive algorithm[3]. The results can be used to compare the two processors and, in the case of the 80486, the difference between native code generation and a threading mechanism.

Swift X	MF 1000	F 486	MF 486	ForthCMP
1.52 s	1.45 s	9.68 s	9.55 s	1.64 s
36 bytes	32 bytes	38 bytes	38 bytes	28 bytes

Table 6

Here, all versions run code compiled from the same source. The difference between Swift X and MF 1000 is due to their packaging of instructions. The difference between F 486 and MF 486 is probably due to alignment of code words.

Matrix Addition

This example calculates the sum of two matrices of 100*100 elements.

Swift X	MF 1000	F 486	MF 486	ForthCMP
8.13 ms	4.22 ms	40.8 ms	24.5 ms	10.3 ms
52 bytes	32 bytes	54 bytes	48 bytes	70 bytes

Table 7

[1] Available from Tom Almy.
[2] For the 80486, the result overflows. However, this doesn't affect the timings.
[3] As before, there is an overflow with 16-bit arithmetic.

The results show how simpler code with fewer instructions can be executed faster. The improvement is more or less of the same order in the PSC 1000 (48% reduction) and in the 80486 (40% reduction).

Matrix Multiplication

This example calculates the product of two matrices of 100*100 elements. The results of the 80486 may vary, as this processor has an early-out multiplication algorithm. The values given here are for the typical case.

Swift X	MF 1000	F 486	MF 486	ForthCMP
1.32 s	1.05 s	6.21 s	3.42 s	1.86 s
140 bytes	96 bytes	118 bytes	112 bytes	188 bytes

Table 8

In the DTC cases, this example is similar to the previous one. For the PSC1000 the difference is not so important, as multiplication is a relatively slow operation.

It should be noted that the absolute reduction in time in these last two examples is greater when moving from F 486 to MF 486 than when moving from MF 486 to ForthCMP.

Prime Numbers

The last example finds the prime numbers among the 25000 first positive integers. It does this by means of the Erasthotenes sieve, packing each number into a bit.

Swift X	MF 1000	F 486	MF 486	ForthCMP
96.1 ms	92.8 ms	569 ms	563 ms	109 ms
224 bytes	172 bytes	202 bytes	198 bytes	183 bytes

Table 9

Even if the ANS and MF versions have different coding, there is not much difference in the results.

Summary of Results

Table 10 shows the timing for all tests. I haven't tried finding any sort of average, as the relative weights would vary a lot from one real application to other.

	Swift X	MF 1000	F 486	MF 486	ForthCMP
Adding	6.25 ms	1.17 ms	25.0 ms	24.22 ms	6.25 ms
Fibbonacci	1.52 s	1.45 s	9.68 s	9.55 s	1.64 s
Sum of mat.	8.13 ms	4.22 ms	40.8 ms	24.5 ms	10.3 ms
Product of mat.	1.32 s	1.05 s	6.21 s	3.42 s	1.86 s
Sieve	96.1 ms	92.8 ms	569 ms	563 ms	109 ms

Table 10

Table 11 shows the memory usage for all tests. MF 1000 gives the best result, especially when considering that both F 486 and MF 486 have 16-bit tokens and that ForthCMP code is in 16-bit mode.

	Swift X	**MF 1000**	**F 486**	**MF 486**	**ForthCMP**
Adding	32 bytes	16 bytes	26 bytes	24 bytes	22 bytes
Fibbonacci	36 bytes	32 bytes	38 bytes	38 bytes	28 bytes
Sum of mat.	52 bytes	32 bytes	54 bytes	48 bytes	70 bytes
Product of mat.	140 bytes	96 bytes	118 bytes	112 bytes	188 bytes
Sieve	224 bytes	172 bytes	202 bytes	198 bytes	183 bytes
TOTAL	484 bytes	348 bytes	438 bytes	420 bytes	491 bytes

Table 11

Conclusions

These results show that the usage of extra registers can improve the code, both in raw speed and in size. As it could be expected, this improvement can be important both in the PSC1000 and in a threaded model, where the number of instructions has greater weight than their relative complexity.

It is easy to find some tests that make a great difference between the models, but there are also many situations where the extra registers are not needed.

Furthermore, when a good optimising compiler is available (as it is the case of ForthCMP for the 80x86) it can give better results with the standard model that a naive compiler (as it is the case for different threading mechanisms) with an improved model.

When using ANS Forth, it is possible to use off-the-shelf words. These words have to be coded from scratch in the MF model. (An example of this can be the SQRT function used to limit the search of composite numbers in the sieve.)

However, if space is not critical, both models can coexist peacefully in the same program.

References

[1] Moore, Charles and Fox, Jeff. *F21 Microprocessor Preliminary Specifications*. 1995.

[2] Patriot Scientific Corporation. *PSC1000 Microprocessor Reference Manual*. 1999.

[3] Tasgal, John. *An Introduction to Machine Forth*. Forthwrite, April 2000.

[4] Thomas, Reuben. *Machine Forth for the ARM processor*. EuroForth 1999.