Machine Forth for the ARM processor

Reuben Thomas*
Computer Laboratory, University of Cambridge

23rd August 1999

Abstract

Fox and Moore [2] have recently proposed a new VM for Forth, called Machine Forth. Using a simple concrete model, it is said to be readily adaptable to different hardware, and to produce reasonably efficient code without needing to descend to assembler. It is also intended to be an excellent basis for Forth compilers. This paper examines these claims with respect to an implementation for the ARM processor, and compares a Machine-Forth based Forth system with a similar system using the ARM's machine model, and a conventional Forth system.

1 Introduction

In March 1999 there was a discussion on `comp.lang.forth` of Charles Moore's Machine Forth, a virtual machine model which he was said to be using for all his Forth programming, and had realised in several silicon designs such as the F21.

Jeff Fox, Moore's amanuensis, said that Moore felt the Machine Forth VM to be rather better for low-level Forth programming than the classic Forth VM, allowing one to get close to the machine while retaining the same VM no matter what processor is being used underneath. Furthermore, the simplicity of the design meant that it required no more than a small macro assembler to implement.

Intrigued, I decided to test the model for myself by implementing it on the ARM processor [3]. I wanted to see how the new instructions such as auto-increment addressing would ease programming, how the lack of old faithfuls such as `SWAP` and `ROT` would hinder it, how simple the system would be to build, and how it would compare with conventional Forth systems in terms of speed and code density. Also of interest was how much if at all the Machine Forth VM would need to be changed to match the ARM hardware.

My test drive of Machine Forth consisted of writing a Forth compiler in it, which in keeping with Moore's preoccupation with smallness was to be minimal. (Another obvious experiment would have been to port my usual compiler, an extensive ANS-compliant system, to Machine Forth.) I then rewrote the compiler in ARM code to compare writing directly for the ARM with writing in Machine Forth. Benchmarks were run on both systems, and on conventional Forth and C compilers.[1]

The discussion that follows assumes familiarity with the Machine Forth model, as detailed in the preliminary F21 data sheet [2]. Herefrom, "Machine Forth" is abbreviated "MF", and my version, with unashamed conceit, "MF32".

2 Changes to the machine model

As I had only the F21's specification as my guide to MF, I didn't know how Moore had implemented it on conventional processors. The following issues were dealt with:

*`rrt@sc3d.org`

[1] All the source code, and binaries for Acorn RISC OS, are available from `http://sc3d.org/rrt/`.

Data width Since the ARM has a 32-bit architecture, the data width was changed to 32 bits.

Testing carry The instruction `C=0` tests an extra bit stored with each stack item on the F21. This would be complex and slow to implement on a conventional processor, so the instruction was changed to test bit `T31`, which works for the original use of testing numbers to see if they are negative. It doesn't help with arithmetic carry unless registers are treated as 31-bit quantities, but with 32-bit registers multiple-precision operation is needed much less often than with 20-bit registers.

Stacks The F21 holds its stacks on-chip, so they are fixed-size; since the ARM has no hardware stacks, they can be of arbitrary size.

Subroutine call and return In common with most RISC processors, the ARM does not automatically push the subroutine return address on to the stack, but moves it into a register, thus avoiding memory accesses when calling leaf subroutines. Since always pushing the return address is both slow and uses 8 bytes per `call` instruction instead of `4`, a new register, `L` ("link"), was added, which caches the top of the return stack in the same way that `T` caches the top of the data stack. Two new instructions, `ret` and `:`, were added to allow leaf subroutine optimisation, and the semantics of instructions dealing with the return stack had to be changed slightly to take `L` into account.

Byte addressing Since the ARM is byte-addressed, the increment of `P` had to be 4. It also seemed sensible to add load and store instructions to deal with bytes.

Multiplication The ARM has hardware multiply, so `+*` was replaced by `*`. (`+*` has other applications apart from multiplication, but it seems reasonable to remove an instruction that would be rarely used when a multiply is provided.)

`nop` The ARM has no need for a no-op, so `nop` was removed. It could in any case be emulated with phrases such as `push pop` or `dup drop`.

OS access The instruction `swi` was added to allow the ARM's `SWI` (SoftWare Interrupt) instruction to be used to call the host operating system.

The revised VM model is:

2.1 Data elements

- Stacks: data `S`, return `R`
- Registers: `T`, `L`, `A`, `P`

2.2 Execution cycle

Perform instruction at `P`; if `P` is not altered by instruction then set `P` to `P`+4; repeat

2.3 Instruction set

Table 1 shows the instruction set and its semantics. The first column gives the instruction's name, the second the number of immediate operands. These are referred to as `V1`, `V2`, ... The third column gives the semantics. The following shorthands are used: "push `X` to `T`" means "push `T` to `S`, set `T` to `X`", and "pop `T` to `X`" means "evaluate `X`, set `X` to `T`, pop `T` from `S`".

Instruction	Operands	Operation
#	1	push V to T
else	1	jump to V
T=0	1	jump to V if T = 0
C=0	1	jump to V if T31 = 0
call	1	set L to P+4, jump to V
ret	0	set P to L
:	0	push L to R
;	0	pop P from R
A@	0	push A to T
A!	0	pop T to A
@A	0	push [A] to T
!A	0	pop T to [A]
B@A	0	push [A]0-7 to T
B!A	0	pop T0-7 to [A]0-7
@A+	0	push [A] to T, add 4 to A
!A+	0	pop T to [A], add 4 to A
B@A+	0	push [A]0-7 to T, add 1 to A
B!A+	0	pop T0-7 to [A]0-7, add 1 to A
pop	0	push L to T, pop R to L
push	0	push L to R, pop T to L
@R+	0	push [L] to T, add 4 to L
!R+	0	pop T to [L], add 4 to L
B@R+	0	push [L]0-7 to T, add 1 to L
B!R+	0	pop T0-7 to [L]0-7, add 1 to L
com	0	one's complement T
2*	0	shift T one place left
2/	0	shift T arithmetically one place right
*	0	set T to [S]×T, pop S
-or	0	set T to exclusive-or of [S] and T, pop S
and	0	set T to and of [S] and T, pop S
+	0	set T to [S]+T, pop S
dup	0	push T to T
over	0	push S to T
drop	0	pop T from S
swi	3	pop V2 arguments from T into ARM registers R0 to RV2−1, call SWI V1, push V3 results from ARM registers R0 to RV3−1 to T

Table 1: The MF32 instruction set

3 The assembler

The assembler was easy to write, as promised. I added the usual Forth assembler control structures for `IF...THEN` conditionals and `BEGIN...REPEAT`/`UNTIL`/`AGAIN` loops, plus the MF variants based on `-IF`. After completing the compiler I added a simple peephole optimiser to the assembler to remove push–pop pairs; this probably took longer to get working than the rest of the assembler, though the code is short.

4 The compiler

The compiler has just enough tools to provide a minimal interpretive environment: an interpreter/compiler, numeric input and output, and a dictionary with the ability to create new definitions.

5 Difficulties with Machine Forth

5.1 Getting started

As expected, the code was hard to write at first because of the lack of familiar stack and arithmetic operators, and I found it hard to remember which of `A@`/`A!` and `@A`/`!A` was which (I eventually succeeded by remembering the latter are like `@A+`/`!A+`).

As time went on, I began to find common idioms such as using `A` and the `R` stack to permute items, sometimes using `R` to store loop indices or limits, and `DUP BEGIN DROP` to discard the flag in conditional loops (since MF's `if` and `-if` instructions do not pop `T`).

5.2 Thinking portably

Despite getting something of the feel of MF, I found it difficult not to compare different MF instruction sequences according to their ARM translation when considering how best to implement a word. For example, in my MF32 implementation, `0 lit` is always the fastest way to get 0 on top of the stack. On the F21 `dup dup -or` is often a better bet. This seems to contradict MF's portability, although both code sequences produce the same result when used on different architectures. However, this problem occurs in any portable language, and is simply more visible on MF because it is so simple.

5.3 Instruction cache synchronisation

One problem that any dynamic code generator for the StrongARM must cope with is that its instruction cache is not automatically synchronised with the data cache. A simple and safe solution was to add `CODE!` which synchronizes the instruction cache on the address stored to, and to make `,` use it to store into the dictionary (the only other word that directly compiles code in my MF32 system is `THEN`).

5.4 Adding primitives

I found that I needed several functions that are not provided by MF. I added `negate`, `minus`, `or`, `lshift` and `rshift`. I adopted the rule of thumb that if a word could be written with fewer ARM instructions than MF32 instructions, I would add it in ARM code; in fact, all those mentioned above were added as inline primitives, as if they were part of the MF32 assembler.

5.5 Other assembly-coded words

Several posts to `comp.lang.forth` have discussed the omission of `SWAP`. I ended up adding `SWAP` to my system, but it was only used four times; every other time a partial swap, with one quantity ending up in `A` or on the return stack, was more efficient. The extra inline primitives were similarly little used: the only one used more than two or three times was `OR`, which was used eight times.

`EXECUTE`, `MIN` and the OS-dependent `EMIT`, `SPACE`, `CR` and `BYE` were also written in ARM assembly. `EXECUTE` deserves a special mention: in F21 it's simply `push ret`, but it's a lot harder to write in MF32, though it can be done (think about it before looking at section 9 for the answer).

5.6 Lack of stack juggling

MF's lack of stack access words mean that the usual pressure Forth applies to the programmer to avoid stack juggling and to factor into small pieces is even stronger. At first it seems like a strait-jacket, but I found that only one or two words were seriously constrained by the lack of `PICK` and `ROLL` (`NUMBER` in particular was tricky). Even then, it was probably largely my fault for not being sufficiently accustomed to MF.

5.7 Division

For number input and output, division is a must. MF doesn't have a division instruction, and neither does the ARM, so I used assembler routines.

5.8 Reading and writing the stack pointers

A serious omission is not being able to read and write the stack pointers: this is important not so much for implementing `DEPTH` as for resetting the stack in `QUIT`. When the stack is implemented in hardware as on the F21 there is no need, but in software it is necessary to avoid a memory leak. I avoided the problem by making `QUIT` simply restart the system by branching to the initialisation code which sets up the stack pointers.

6 ARM Forth

Having finished MF32 for the ARM, I rewrote the compiler in ARM code. This sounds like what Moore calls "Hardware Forth", but there's a difference: while the cmForth compiler's structure was dictated by the machine (in this case, the Novix Forth chip), ARM Machine Forth is exactly the same as Machine Forth, just using the ARM machine model rather than the MF model.

The code was not only shorter, but at least as easy to write (indeed, many of the words worked first time, and they were not just naïve translations from the MF32 versions, but were reworked to take advantage of the ARM). The ARM's small, regular instruction set is comparable in size with Machine Forth (it has about 20 basic instructions), but is richer in arithmetic and logical operations, addressing modes, and has features that Machine Forth lacks, such as conditional execution.

On the other hand, the fact that the ARM is a register machine leads to duplication in the compiler: some words that are useful for interactive use, such as `DUP` and `+`, are rarely used in compiled code, where the ability to address registers can be exploited.

	MF32	ARM Forth
Definitions	109	175
Total size/cells	1478	1446
Code size/cells	1180	854
Instructions	754 MF + 52 ARM	854 ARM
Source length/bytes	16065	19679

Table 2: Comparison of the MF32 and ARM Forth systems

	ANSI	MF32	ARM Forth	C
Time/s	6.48	7.30	1.94	0.92
Code size/cells	32	37	25	22

Table 3: Random number benchmark data

7 Comparison of MF32 and ARM Forth

7.1 Size

Table 2 shows some comparisons of the MF32 and ARM Forth systems.

This gives a code density of 6.0 bytes, or 1.5 ARM instructions, per MF32 instruction. The peephole optimiser saved about 100 instructions, or 9% of the total code generated.

While the ARM Forth system has 66 more definitions than the MF32 system, as the assembler for the ARM chip is more complex than that for Machine Forth, the binary is 32 cells smaller, and contains 326 fewer cells of code. Further, most of the words occurring in both systems required fewer ARM instructions than MF32 instructions to write; it seems that ARM Forth's code density is roughly double that of ARM MF32.

7.2 Speed

Two benchmarks were run on the MF32 system, the ARM Forth system, a naïve subroutine-threaded ANSI Forth compiler,[2] and GNU C.[3] The code for the first, a simple random number generator, is shown in figure 1,[4] and the code size and timings in tables 3 and 4. 10,000,000 iterations of the first test were run, and for the second, a simple prime finder, primes up to 10,000 were found.[5]

The ARM Forth versions are by far the smallest and quickest of the Forth versions, but

[2] aForth 0.75, available from `http://sc3d.org/rrt/`.

[3] GNU C 2.96 for ARM, using `-O2`.

[4] The C code is omitted for reasons of brevity; it is a literal translation of the ANS Forth version, and can be found in the Machine Forth distribution.

[5] It would have been better to run well-known benchmarks, such as Ertl's integer suite [1], but there was not enough time to translate them into Machine Forth and ARM Forth.

	ANSI	MF32	ARM Forth	C
Time/s	3.79	5.39	1.31	1.77
Code size/cells	39	57	28	54

Table 4: Primes benchmark data

```
VARIABLE SEED  -1 SEED !
: RANDOM   SEED @  DUP 1 LSHIFT  SWAP 0< IF 495090497 XOR
   THEN DUP SEED ! ;
: TEST   0 DO  RANDOM DROP  LOOP ;
```

(a) ANSI

```
CREATE SEED  -1 ,
: RANDOM   SEED @A  -IF 495090497  ELSE 0  THEN
   SWAP 1 LSHIFT XOR  DUP SEED !A ;
: TEST   BEGIN  RANDOM DROP  1 - 0UNTIL  DROP ;
```

(b) MF32

```
CREATE SEED  -1 ,

CODE RANDOM
  LINK,
  R @! 1 DB STM,          save R0
  495090497 LITERAL       put magic on stack
  ' SEED BL,              call SEED (leaves address in A)
  0 A 0@ LDR,             load value of SEED
  0 0 # CMP,              compare SEED with 0
  0 0 1 #LL MOV,          shift value left once
  0 0 T MI EOR,           if SEED < 0, EOR with magic
  T 0 MOV,                copy SEED to top of stack
  0 A 0@ STR,             save new seed
  R @! 1 IA LDM,          reload saved R0
  UNLINK,
END-CODE

CODE TEST
  LINK,
  R @! 1 DB STM,          save R0
  0 T MOV,                copy top of stack to R0
  T S 4 #@+ LDR,          DROP
  BEGIN,
    ' RANDOM BL,          call RANDOM
    T S 4 #@+ LDR,        DROP
    0 0 1 # SET SUB,      decrement and test counter
  EQ UNTIL,
  R @! 1 IA LDM,          reload saved R0
  UNLINK,
END-CODE
```

(c) ARM Forth

Figure 1: The random-number benchmark

surprisingly the ANSI Forth code is both smaller and quicker than the MF32 version.[6] The C benchmark results need a little explanation: the first benchmark is a little smaller and even faster than ARM Forth because loading the random number generator seed does not require a subroutine call, as it does for a `CREATE`d variable. The second benchmark is much larger because there are several small functions, and about half the generated code is function entry and exit sequences.

In an earlier version of this paper, I foolishly wrote "MF32 will obviously lie somewhere between subroutine threaded code and natively-compiled code in speed". Although it would be equally foolish on the basis of a few benchmarks to say that that is false, MF's claims for speed and code density need further scrutiny, at least on typical desktop architectures.[7]

7.3 Ease of programming

ANSI Forth and MF32 were about equally easy to program in. Contrary to the experience of writing the compilers, ARM Forth was much harder, mostly because it was hard to test the ARM words interactively. Also, ARM code is much more long-winded and harder to read than Forth. In particular, ARM Forth is harder to factor, because definitions which return flags would naturally modify the ARM flags register in ARM Forth rather than returning a flag value on the stack, but this is currently impossible, as the flags are preserved across subroutine calls. Perhaps this should be changed.

However, it is rather unfair to compare writing Machine Forth and ARM code as vehicles for Forth, as the former was designed for it, whereas the latter was not. Using a more natural machine code style, ARM assembler is just as easy to write as Machine Forth, and if used from within a conventional Forth compiler, it can be tested interactively too.

8 Conclusions

This has been a small experiment, and it's never wise to draw firm conclusions about programming in Forth from writing Forth compilers. It is clearly the case that MF is easy to implement, but that's obvious just from looking at the machine model. Whether it makes it easier to write fast, small, portable code is debatable. For me, Machine Forth falls between two stools: for high level code, I'd rather write in full-blown Forth, and for low-level code, I'd rather have the power, speed, and compactness of assembler (used from within a Forth compiler, of course!).

This impression is perhaps biased by the fact that the processors I've programmed extensively (the 6502, 68000 and ARM) have small and simple instruction sets. On an unfamiliar processor, especially one with a large and complex instruction set, the simplicity of MF might help me to produce reasonable code more quickly than by learning the native assembly language.

The portability of Machine Forth might also be cited in its favour, but that is only an advantage when writing high-level code; when MF is being used as a replacement assembly language the code being written is machine-dependent anyway, and portability is irrelevant.

However, MF does offer some useful ideas. Its explicit use of an address latch is a simple way to improve the performance of small Forth compilers that cannot afford to have an optimiser. Even more interesting are its non-destructive conditionals, which could easily be used in traditional Forth.

In conclusion, Machine Forth's judicious mixture of novelty and classic simplicity merit careful study, though I for one will not be abandoning the traditional combination of Forth and assembler in its favour.

[6] The primes benchmark spends about 60% of its time executing the software division routine, so is of less significance, though it still shows the same trend as the random number benchmark.

[7] Similar code density has been reported on `comp.lang.forth` for an Intel implementation.

9 Answer to exercise

`EXECUTE` can be written in MF32 as

```
A! pop A@ push push ;
```

10 Acknowledgements

Jeff Fox was kind enough to read and comment on the paper, a referee made several helpful comments, Hanno Schwalm pointed out the weakness of the primes benchmark, and Marcel Hendrix asked for the comparison with C.

References

[1] M. Anton Ertl. Performance of Forth systems, 1996. `http://www.complang.tuwien.ac.at/forth/performance.html`.

[2] Charles Moore and Jeff Fox. Preliminary specification of the F21, 1995. `http://pisa.rockefeller.edu:8080/MISC/F21.specs`.

[3] VLSI Technology Inc., Eaglewood Cliffs, NJ. *Acorn RISC Machine family Data Manual*, 1990.