Stephen Pelc, stephen@vfxforth.com
Wodni & Pelc GmbH
9 September 2025

Forth for ARM 64 CPUs

Trials and tribulations

ARM's 64 bit CPUs are very different beasts from the 32 bit ones. The instruction mix Is completely different and appears to be derived from the Power PC. Cache behaviour Is quite different and poorly documented. VFX Forth (64 bit) has been ported to the ARMv8-A architecture and an alpha release is expected later in 2025. The current Arm® Architecture Reference Manual for the A profile is 14,568 pages long. Despite my reservations, the more I use this CPU, the more I come to appreciate it. The code density is far better than expected.

In the beginning ...

Many years ago, the 32 bit ARM CPU appeared. Over the years various instruction sets and derivatives appeared. As a company, ARM has always made frequent changes to the instruction set to support the silicon. Unlike other companies, an ARM instruction set supports today's architecture. 64 bit ARMs are based on the ARMv8 architecture from 2011, and ARMv9 already exists. ARMv9 is basically has the ARMv8 instruction set with extensions.

Under some customer pressure, we started a port of VFX Forth to the ARMv8A architecture.

About the ARM64

The ARM 64 bit CPU in native mode has very little to do with ARM32, although the original ARM32 and Thumb-2 ISAs are supported for legacy reasons and I shall discuss these no further. There are several ARM64 assembler books out there; in the main they are useless for people who have assembler experience and projects under their belt. You will find the

“Compiler Writer’s Guide to the Power PC” of use, and PowerPC assembler experience of value.

The instruction set consists of 32 bit instructions only, with a number of conditional operations derived from PowerPC. Many of the instructions have three operands. There are 32 registers, including a zero register and a subroutine return register. There are many pseudo-instructions which offer a different syntax to the base instruction. Although initially confusing, the instruction set makes sense after a while. Despite this, a description of the **BFM**, **SBFM** and **UBFM** instructions for human beings would be very useful.

The big change is in the cache architecture, which is a real pain. We have not finished with it yet.

Instruction set

I will only discuss the basic and integer portions of the instruction set here. There is a full set of basic instructions plus enough special instructions for supporting cryptography, security and ARM32 that one could already call the instruction set baroque.

Poor code density has been a problem for many RISC or load/store architectures. A side effect of improving code density for ARM64 has been a selection of immediate value encodings -

1) Arithmetic - 16 bit, 12 bit+shift, 12 bit, 8 bit, 7 bit, 6 bit.
2) Logical - 13 bit mask about 5000 options. Used by **AND**, **ORR** and **EOR**.
3) Branch offsets - 26 bits, 19 bits, 14 bits
4) Memory offsets - s19 bits, u12 bits, s9 bits, s7 bits, 0

The branch instructions result in a call range of +/-128 Mbytes. Most conditional branches have a +/-1 Mbytes branch range. This is a vast improvement over many other CPUs.

Because of the impact of mispredicted branches on performance, conditional instructions reduce both code size and improve performance by avoiding conditional branches, e.g. the end of **WITHIN**

```
cmp      tos, x17            \ (n1-n2)-(n3-n2)
csinv    tos, xzr, xzr, .ls  \ cy -> -1, ncy -> 0
```

Conditional select invert

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register. See: **`CSEL`**, **`CSET`**, **`CSETM`**, **`CSINC`**, **`CSINV`**, and **`CSNEG`**.

Caches

The cache system for ARM64 is quite different to that for ARM32. There L1 caches for both code and data. The minimum cache line size is 64 bytes, but is permitted to be larger. The size can be read by application programs running at execution level 0 (EL0). A limited range of cache maintenance instructions can be run at EL0 that permit the cache to be flushed by code running at EL0.

Tools

The main tools we use for porting are the VFX cross compiler. Testing is performed on a 64 bit version of ARM linux. We have used both UTM and Parallels to host these on Apple Silicon Macs. Apple produce a tool called Rosetta that enables x64 applications to run on Apple Silicon. Two problems with Rosetta have forced us to abandon it.

1) x64 Forth cross compilers are slowed down by a factor of 100 or more.

2) There are bugs in Rosetta that have forced other projects to abandon it.

We have therefore moved back to cross-compiling on an x64 box, copying the output to an ARM Linux and then debugging.

For both the cross compiler and the target code there are five sections that change for each target

1) Assembler - an ARM64 assembler is provided with a prefix notation that closely follows the ARM64 standard notation.

2) Disassembler - it is almost impossible to debug compiled native code without one.

3) Code generator and optimiser - produces faster and shorter code than that produced by combining patterns. The VFX code generator is an analytical compiler that tracks which registers are used and what they contain.

4) Required code - these are words that either cannot be written easily in high level Forth, or should be written in assembler for performance reasons.

5) Operating system interface - calling functions in shared libraries and providing callbacks that can be used by the operating system.

The following are examples of these.

```
code l!c(t) \ l addr --
    \ *G Store and Flush instruction cache line containing *\i{addr}.
    \ ** Use in stores of code.
      ldr   x0, [ psp ]!, # 8       \ get l
      str   w0, [ tos, # 0 ]        \ save l
      ic    ivau tos                \ flush
      dsb    SY                     \ ensure completion of the invalidation
      isb    SY                     \ ensure instruction fetch path sees
                                    \ new I cache state
      ldr    tos, [ psp ]!, # 8     \ restore TOS
      ret   x30
    end-code
```

```
dis l!c(t)
L!C(T)
( 0041:1850 A08740F8 )              LDR       X0, [ XPSP  ]!, # $8
( 0041:1854 400300B9 )              STR       W0, [ XTOS, # $0 ]
( 0041:1858 3A750BD5 )              SYS       $1BA9 XTOS
( 0041:185C 9F3F03D5 )              DSB       # $0F
( 0041:1860 DF3F03D5 )              ISB       # $0F
( 0041:1864 BA8740F8 )              LDR       XTOS, [ XPSP  ]!, # $8
( 0041:1868 C0035FD6 )              RET       XLR  ( NEXT/EXIT )
28 bytes, 7 instructions.

: InOvl?      \ addr1 -- addr2|0
\ *G Returns the overlay address (addr2) if the address (addr1)
\ ** is within an overlay, otherwise returns 0.
  ovl-link @
  begin                             \ -- addr *ovl
    dup
  while                             \ -- addr *ovl
    2dup OVI.end 2@ within if       \ -- addr *ovl
      nip  exit
    then
    ovi.link @
  repeat
  nip                               \ remove addr
;
```

```
dis inovl?
INOVL?
( 0042:9F88 00FF9AD2 )              MOVZ      X0, # $D7F8
( 0042:9F8C 2008A0F2 )              MOVK      X0, # $41 LSL # $10
( 0042:9F90 110040F8 )              LDUR      X17, [ X0, # $0 ]
( 0042:9F94 BA8F1FF8 )              STR       XTOS, [ XPSP, # $-8 ]!
( 0042:9F98 FA0311AA )              MOV       XTOS, X17
( 0042:9F9C 9E8F1FF8 )              STR       XLR, [ XRSP, # $-8 ]!
( 0042:9FA0 5F031FEB )              CMP       XTOS, XZR  LSL# $00
( 0042:9FA4 40020054 )              B .EQ     # $429FEC
( 0042:9FA8 50BF41A9 )              LDP    X16, X15, [ XTOS, # $18 ]
( 0042:9FAC BD6300D1 )              SUB       XPSP, XPSP, # $18
( 0042:9FB0 AF0300F9 )              STR       X15, [ XPSP, # $0 ]
( 0042:9FB4 A10F40F9 )              LDR       X1, [ XPSP, # $18 ]
( 0042:9FB8 A10700F9 )              STR       X1, [ XPSP, # $8 ]
( 0042:9FBC BA0B00F9 )              STR       XTOS, [ XPSP, # $10 ]
( 0042:9FC0 FA0310AA )              MOV       XTOS, X16
( 0042:9FC4 53A3FF97 )              BL        # $412D10     WITHIN
( 0042:9FC8 5F031FEB )              CMP       XTOS, XZR  LSL# $00
( 0042:9FCC BA8740F8 )              LDR       XTOS, [ XPSP  ]!, # $8
( 0042:9FD0 80000054 )              B .EQ     # $429FE0
( 0042:9FD4 BD230091 )              ADD       XPSP, XPSP, # $08
( 0042:9FD8 9E8740F8 )              LDR       XLR, [ XRSP  ]!, # $8
( 0042:9FDC C0035FD6 )              RET       XLR  ( NEXT/EXIT )
( 0042:9FE0 510340F8 )              LDUR      X17, [ XTOS, # $0 ]
( 0042:9FE4 FA0311AA )              MOV       XTOS, X17
( 0042:9FE8 CEFDFF54 )              B         # $429FA0
( 0042:9FEC BD230091 )              ADD       XPSP, XPSP, # $08
( 0042:9FF0 9E8740F8 )              LDR       XLR, [ XRSP  ]!, # $8
( 0042:9FF4 C0035FD6 )              RET       XLR  ( NEXT/EXIT )
112 bytes, 28 instructions.
 ok
```