Stephen Pelc
MicroProcessor Engineering Ltd
133 Hill Lane
Southampton SO155AF
UK

http://www.mpeforth.com

stephen@mpeforth.com

Introduction

Since its first release in 1998, all versions of VFX Forth have been 32 bit Forth implementations. Although there has been almost no customer demand for a 64 bit version, two events have conspired to force the decision to move to 64 bits.

1) Fashion - never to be underestimated in software choices.
2) Operating system reluctance to host 32 bit applications.
   Apple have already dropped support for 32 bit applications from macOS Catalina and iOS 11.
   Linux distributions are increasingly shipping without the 32 bit library support.

Beta test versions of VFX Forth 64 are now (August 2020) available for macOS and x64/amd64 Linux from
http://soton.mpeforth.com/downloads/VfxCommunity
Windows and ARM64 Linux will follow in due course.

Process

All MPE Forths have a cross-compiled kernel, which is used to produce further builds. The first stage is thus to select a host for the cross compiler. We attempted to use a well-known 64 bit FOSS compiler as a host, but for commercial reasons, we needed to support macOS first, and that Forth treated macOS as a second-class citizen. Somewhat reluctantly, we decided to use VFX Forth 32 for macOS as the first host.

Once the first 64 bit target was stable, the assembler, disassembler and code generator were ported to VFX Forth 64, and the x64 cross compiler was ported.

By customer demand, we then ported the ARM/Cortex cross compiler to the 64 bit host.

We can split the jobs into

1) x64 assembler and disassembler
2) code generator
3) shared library interface
4) Floating point
5) porting 32 bit code to 64 bit code
6) testing

x64 Asm and Dasm

Although the AMD64 instruction set is said to be “upward compatible” with x86, the compatibility level still leads to an impact on assembler code. There are three areas that cause trouble or code expansion.

1) I’m dreaming of a REX byte - used to select 64 bit data and R8..R15
2) Not all instructions have 8, 16, 32 and 64 bit operations because of
3) Special cases

The REX byte rules are mostly consistent, but cause problems when byte register selection is affected by whether a REX byte is present - goodbye AH, BH, CH and DH.

There are no POP 32 bit operations. PUSH and POP default to 64 bits and only have 16 and 64 bit forms. Dealing with data size caused the most problems in both the assembler and the disassembler. I was wrong in a design decision made early on, but was too tired to change it! Bad design decisions make you bleed even if the system appears to work sooner.

There's also a peculiarity in that the SIB byte fully decodes all four bits of a register field, but the MODR/M byte does not. In consequence, R12 and R13 are a bit special.

Because the 64 bit operating systems take floating point arguments in XMM registers, you have to extend the assembler and disassembler to support at least a subset of the SSE2 instructions. The base AMD64 instruction set is much more than x86 with 16 registers. There is a 64 bit literal form to load a register, but it's a big instruction and not needed very much if you accept the zero and sign extension rules - some of the redundant addressing modes using the MODR/M byte come back into use.

In order to aid code portability, we chose to treat all literals as 32 bit items in the 32 bit hosted cross compiler. This worked with a few exceptions, and a simple 64 bit literal specifier was portable into the 64 bit hosted cross compiler.

Code Generator

The code generator was in many ways the least of our problems. The existing x86 code generator uses tables for mosts of its CPU definition operations. Converting these to AMD64 was mostly just tedious rather than difficult. The major changes were in defensive programming - far more defining of data size rather than accepting the default.

The biggest changes come from dealing with absolute addresses and literals. Unless you use one of the new instructions with 64 bit absolute addresses, absolute addressing is replaced at the CPU level by PC relative addressing with 32 bit offsets. Making literals work reliably on both 32 bit and 64 bit hosts was tedious, but probably beneficial in that it forced us to convert to a 64 host as soon as possible.

Literal handling changes infected the rest of the code generator in that the x86 code generator can handle literals with very few special cases; whereas the AMD64 needs decisions depending on 32 or 64 bit literal handling.

Other topics that cause problems are sign and zero extension because the instruction set is not regular and the need to bring the floating point stack pointer and top of float stack into the VM register set.

A few changes were made to take advantage of the extra registers. An additional eight registers is more than the code generator can use regularly. Two registers are dedicated to **DO** … **LOOP** handling. One becomes the float stack pointer. A few more will become shared between the data and return stack models.

The code generator is a work in progress and will improve significantly over the next year or so. What has been important to us is to achieve both a suitable base-line performance and sufficient reliability to port existing code easily.

Shared library interface

In the 32 bit world, macOS, Windows and Linux use essentially the same assembly level interface to external functions in shared libraries. In the 64 bit world, macOS and Linux use the same interface, and Windows goes its own way again. Naturally, the list of callee-saved registers is quite different. The following discussion is only for macOS and Linux, which both use the System V AMD64 ABI.

There is a big emphasis on calling through registers, six integer and eight XMM registers being reserved for parameter passing. Unlike the 32 bit world, return values of up to 128 bits may be returned in registers. This particularly affects the macOS Cocoa interface, which uses many co-ordinate pairs of 64 bit floats. These are returned in two XMM registers.

To add a little extra excitement, the person in charge of the Cocoa interface decided that he wanted a more “Forth-like” interface to Cocoa and Objective-C. Explaining that the MPE **EXTERN:** interface was the way it was to avoid the nastinesses caused by primitive notations, we now have a “more Forth-like” interface as well. It will work up to a point, but can still be fooled in admittedly rare circumstances.

```
FUNCTION: CGContextGetPathCurrentPoint ( cgcontext -- ) ( F: -- x y )
FUNCTION: CGContextMoveToPoint ( cgcontext -- ) ( F: x y -- )

Extern: CGPoint CGContextGetPathCurrentPoint(CGContextRef c);
Extern: void CGContextMoveToPoint(CGContextRef c, CGFloat x, CGFloat y);
```

Floating point

As an implementer, I dislike floating point. It is a substantial cause of technical support issues that are really mathematical issues. You really need a “mathmo” to write, maintain and support floating point packages. A good “mathmo” is a rare beast.

In the 64 bit world, SSE is the default interface to the outside world. SSE However, SSE is limited to IEEE 64 bit floats and multiple 64 bit items. We really notice the lack of precision compared to the NDP’s 80 bit format. We then have some choices to make.

SSE code is supposed to be faster than NDP code. Until we have finished an SSE2 optimiser I cannot comment, but unoptimised SSE code is substantially slower than optimised NDP code. The advantage of SSE arises when we can use the SIMD instructions. Given lack of practice in doing this in Forth, a significant development path opens up. Our options seem to be:

1) Go straight to optimised SSE and suffer the loss of precision in favour of potential future enhancements,
2) Stick with NDP and provide conversion operators,
3) Stay with NDP but allow the **EXTERN:** interface to select between an SSE float stack and an NDP system that converts to SSE on the fly.

At present option 3 is looking like a good solution as it preserves the precision of 80 bit floats while leaving the door open for SSE extensions. Option 3 is also a good test of search order and recogniser order management issues.

Porting 32 bit code to 64 bits

Somewhat to my surprise, the vast majority of our 32 bit code ported to 64 bits with little or no change. However, we have been scrupulous in ensuring that structures and addresses are manipulated by words such as **CELL+** rather than **4+**.

The biggest set of problems stemmed from C’s decision to leave **int** as 32 bit, **long** changes to 64 bit, and so on. This also influenced many structure declarations and their alignments. We also noted that whereas MacOS structures were in the main relatively easy to convert, the Linux people had gone to town on wholesale changes. If you are going to break code, just break it properly.

Whereas the macOS 64 bit signal interface was fairly straightforward to port from 32 bits, the Linux 64 bit interface required many changes to structure definitions, much reading of obscure C header files and a lot of swearing. I was thankful that alcohol is cheap in Spain.

Overall, checking operating system structure definitions has taken longer than porting Forth directly. The only real Forth problem we had was in porting the AAPCS (Arm Advanced Procedure Call Standard) for Cortex CPUs from a 32 bit hosted cross compiler to a 64 bit hosted cross compiler. One portion of the code used the same alignment words for host and target navigation rather than using the (already) provided target navigation words. This bug broke a mission-critical test application until it was fixed.

Testing

As the section above indicates, testing is vital.

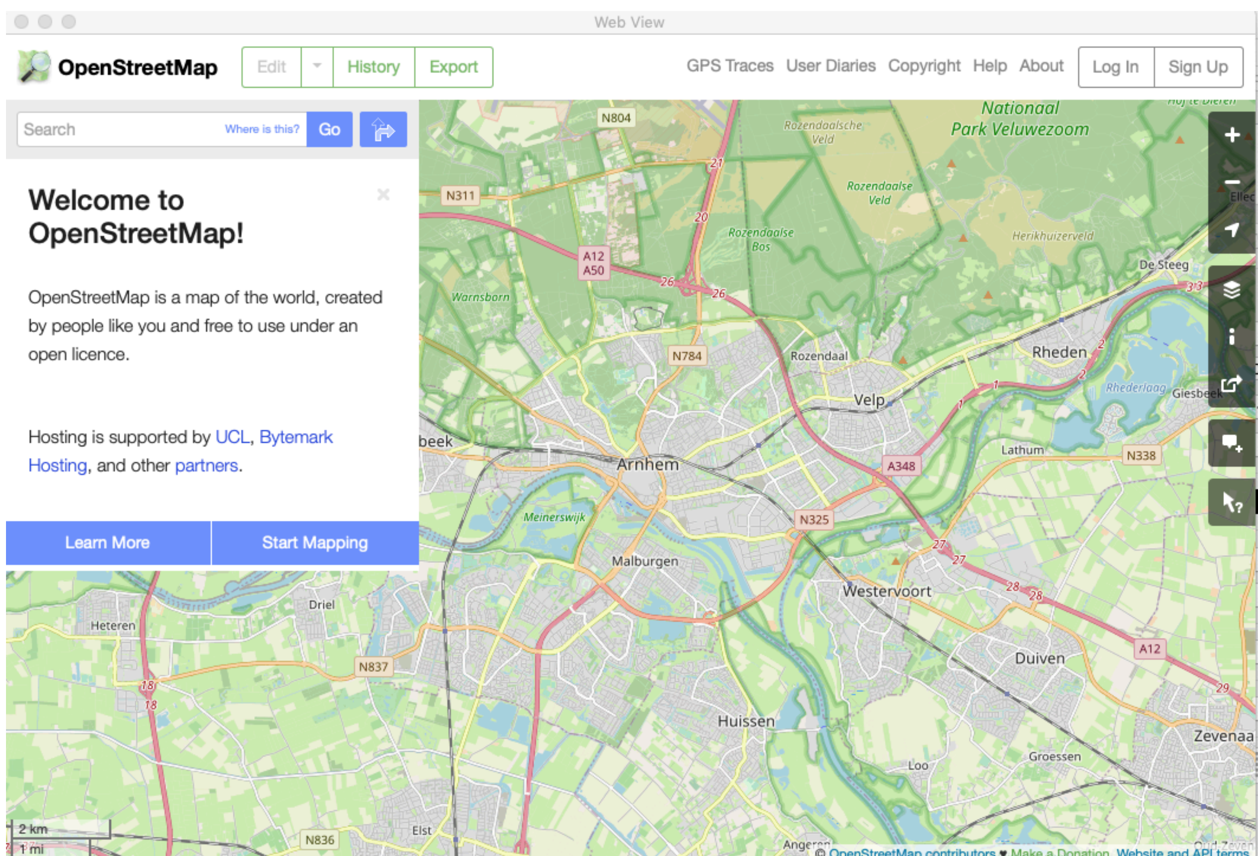
We did reasonably well with the assembler testing. However, testing an assembler before you really know how it is going to be used just leads to testing without exploring the important corner cases. That's why good testers have a mentality of their own that has to be respected.

Testing a code generator is hard. You end up with a set of regression tests that simply protect against repeating the same mistakes.

The Gerry Jackson test suite is a life-saver.

Results

We now have a stable 64 bit for AMD64/x86_64 CPUs. Roelf Toxopeus has ported his Cocoa interface to VFX Forth 64. An application is shown.



Acknowledgements

This software exists because of the efforts of

Robert Sexton
Roelf Toxopeus
Ward McFarland
Bruno Degazio
Gerald Wodni