4g and FAIL
(or: Be careful what you joke about!)

Glyn Faulkner
EuroForth 2023

2023-09-15

Historical context

My final slide last year:

Where next?

How about a parameterised Forth interpreter generator?

```
[marsu@celaeno 4g]$ ./4g -t ITC -T -m ANSI -o forth
Indirect-threaded x86_64 Linux ANSI Forth
Options: top-of-stack in register, linked-list dictionary
Generating forth.S
gcc -m64 forth.S -o forth
Done
[marsu@celaeno 4g]$ ./forth
```

Ask me how this is going next year!

This is how it's going. . .

4g, the Forth-generator

Used approach from Peter Knaggs' EuroForth paper[1] to build a "matched-pair" of ANSI-ish Forths, sharing as much source code as possible:

Direct-threaded:

```
.macro $next
    lodsl
    jmp *%eax
.endm
#include "common.S"
.section .flat
.align 4
docol:
    $pushrs %esi
    pop %esi
$next
```

Indirect-threaded:

```
.macro $next
    lodsl
    jmp *(%eax)
.endm
#include "common.S"
.section .text
.align 4
docol:
    $pushrs %esi
    lea 4(%eax), %esi
$next
```

[1]Peter Knaggs *Using Test Driven Development to build a new Forth interpreter*, http://www.euroforth.org/ef21/papers/knaggs.pdf

Problems...

- Not a scaleable approach!
  - new-runtime system required for every possible configuration or mapping of Forth to machine registers
  - multiple inter-dependent source files makes development slow and painful

It gets worse...

"common.S" is not so common! With `SP` mapped to `%esp`, `dup` looks like this...

```
code dup  ( x -- x x )
    pop %eax
    push %eax
    push %eax
end-code
```

...but with `SP` in `%esi` it might look like this...

```
code dup  ( x -- x x )
    lodsl
    lea -4(%esi), %esi
    mov %eax, (%esi)
    lea -4(%esi), %esi
    mov %eax, (%esi)
end-code
```

Assembler Macros!

```
code dup  ( x -- x x )
    $popds RegX
    $pushds RegY
    $pushds RegY
end-code
```

But if we have top-of-stack in a register, we want dup to look something like this:

```
code dup  ( x -- x x )
    push %ebx
end-code
```

What reasonable definition of `$pushds` and `$popds` can give us this?

- Macro complexity rapidly explodes!
- Debugging becomes a nightmare.
- Generated code is hard to read and modify.

What a mess!

F.A.I.L.: The Forth Abstract Instruction Language

The original insight:

```
code dup  ( x -- x x )
    ...
```

Hmm. That stack comment looks suspiciously compilable!

```
    ( x -- x x )

    1. Move top-of-stack to register X
    2. Re-pack the stack with two copies of register X
```

Stack shuffling 1

```
: dup  (   x -- x x ) ;
```

. . . becomes. . .

```
code dup
    # pop stack to x
    pop %eax
    # push x to stack twice
    push %eax
    push %eax
    $next
end-code
```

Stack shuffling 2

Data stack pointer is somewhere exotic? No problem?

```
code dup
    # pop stack to x
    mov (%eax), %edi
    add $4, %eax
    # push x to stack twice
    sub $4, %eax
    mov %edi, (%eax)
    sub $4, %eax
    mov %edi, (%eax)
end-code
```

Stack shuffling 2

Data stack pointer is somewhere exotic? No problem?

```
code dup
    # pop stack to x
    mov (%eax), %edi
    add $4, %eax
    # push x to stack twice
    sub $4, %eax
    mov %edi, (%eax)
    sub $4, %eax
    mov %edi, (%eax)
end-code
```

The assembly output is less than optimal, but quite readable.

Recall that my goal is to automate the boring parts of bringing up a new Forth-like language.

Return stack 1

Works also with the return stack

```
: >r   ( x --   ) (r:    -- x ) ;
```

```
code >r
    # pop stack to x
    pop %eax
    # push x to return stack
    sub $4, %ebp
    mov %eax, (%ebp)
    $next
end-code
```

Return stack 2

Using both stacks at once...

```
: r@ (   -- x ) (r: x -- x ) ;

code r@
    mov (%ebp), %eax  ;  add $4, %ebp
    push %eax
    sub $4, %ebp  ;  mov %eax, (%ebp)
    $next
end-code
```

Return stack 3

What about a word with atypical behaviour. . .

```
: exit    (r: IP --   ) ;
```

(`IP` is the hardware register holding the Forth instruction pointer)

```
code exit
    # pop return stack to IP
    mov (%ebp), %esi
    add $4, %ebp
    $next
end-code
```

Words that actually *do* something 1

Stack shuffling isn't Turning complete! (probably)

Words that actually *do* something 1

Stack shuffling isn't Turning complete! (probably)

We can already map the stack onto virtual registers...

```
: dup   (    x -- x x ) ;
```

...which map onto machine-registers.

```
: dup   (    %eax -- %eax %eax ) ;
```

So what if we imagine an assembly-like syntax that works on virtual registers?

```
: +    ( a b -- c )    a b  c  +   ;
```

Borrowing ideas from QEmu's TCG intermediate representation, (almost) all of my primitives have separate parameters for source and destination registers.

(I regret my syntax choice here: the first + is defining the Forth word and the second is a FAIL primitive).

Words that actually *do* something 2

The resulting assembly looks like this...

```
code +
    pop %ecx
    pop %eax
    add %ecx, %eax
    push %eax
    $next
end-code
```

push and pop

`push` and `pop` have their own instructions (of course!).

```
<dest> <ptr> pop
<src> <ptr> push
```

Special case:

- the *only* FAIL instructions that modify a register in-place.
- `<ptr>` follows target register in both cases for ease of reading which stack is being accessed.

Common uses:

```
\ pop the data-stack to x
x  SP pop
\ push y to the return-stack
y  RP push
\ threaded-code NEXT
W  IP pop
W     execute
```

Branching

```
: (brn) (   -- )
    IP IP @
;
: (brz) ( n -- )
    \ "pop" through IP
    \ to reg b
    b  IP pop
    n  (0=) if
        b IP move
    then
;
```

```
code (brn)
    mov    (%esi), %esi
    $next
end-code
code (brz)
    pop %eax
    mov (%esi), %ecx
    add $4, %esi
    test %eax, %eax
    jnz 1f
        mov     %ecx,  %esi
    1:
    $next
end-code
```

Condition specification syntax using `(0=)` as an "argument" to `if` is awkward. Is there a better way?

Complications! 1

x86 has some *nasty* instructions:

- `div` and `idiv` have four implicit arguments (two source and two destination)
- `mul` and `imul` can clobber `%edx` (which might be Forth's stack- or instruction-pointer!)
- `shl` `shr` `sar` and `sal` require the number of places shifted to be in `%cl`

…and lots of register aliasing:

- `%eax`, `%ax`, `%ah` and `%al` all refer to the same hardware register!

```
: sm/rem  ( a b c -- d e )
    a b c  d e  sm/rem ;
```

…needs to produce something like this:

```
code sm/rem  ( l h d -- r q )
    pop %ecx
    pop %edx
    pop %eax
    idiv %ecx
    push %edx
    push %eax
    $next
end-code
```

`h`, `l`, `r`, and `q` *must* be in the correct machine registers.

How to solve?

Complications! 2

Current answer: cheat!

- Check abstract instruction's register affinity
- Run rudimentary liveness analysis.
- Allocate required registers if possible.
- Otherwise throw a compilation error.

Two possible solutions

- QEmu-style "helper functions"
  - compilation guaranteed to succeed
  - **but** run-time overhead of switching out of "Forth-mode" and into e.g. "C-mode"
- More advanced register allocator
  - spilling registers could handle some tricky cases
  - some x86 instructions can work directly with a value in memory, no register allocation required.
  - **but** compilation failure is still a possibility
  - **unanswered questions:** how does spilling work if our stack pointers are the registers we want to spill?

What next?

F.A.I.L.

- Port the start-up code for the runtime to FAIL.
- Easier configuration (currently requires editing an Awk script!)
- Use FAIL words inside FAIL words (currently `$next` has to be an assembly macro!)
- Better register allocation
- Support more threading models: Token, Subroutine. . .
- x86_64 and ARM support
- Re-write in Forth! (currently Awk!)
- More flexible instruction generation (optimise for size, speed, readability. . .)
- Abstract away the dictionary implementation
- Selectable back-end (GNU as, nasm, C, machine code. . .)

4g

- Package as a commandline tool (currently a Makefile!)
- More complete and correct ANSI support
- Add other Forth “models”: eForth, F83, F77?
- Port some of my own Forths!

Any Questions?