36th EuroForth Conference

September 4-6, 2020

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 36th EuroForth finds us mostly at home, thanks to Covid19, and the conference is being held on the Internet. The two previous EuroForths were held in Hamburg, Germany (2019) and in Edinburgh, Scotland (2018). Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there have been no submissions to the refereed track. Nevertheless, I thank the program committee for their willingness to review the papers. I thank the authors for their papers.

Late papers will be included in the final proceedings (`http://www.euroforth.org/ef20/papers/`).

These online proceedings also contain presentations that were too late to be included in the proceedings available during the conference.

You can find these proceedings, as well as the individual papers and slides, and links to the presentation videos on `http://www.euroforth.org/ef20/papers/`.

Workshops and social events complement the program. This year's EuroForth is organized by Gerald Wodni.

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
Ulrich Hoffmann, FH Wedel University of Applied Sciences
Matthias Koch, Institute of Quantum Optics, Leibniz University Hannover
Jaanus Pöial, Tallinn University of Technology
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas, SC3D Ltd.

Contents

Smart Stacks in VHDL

Ulrich Hoffmann and Andrew Read

EuroForth 2020

Introduction

This work originates from our development of seedbed, a minimal VHDL implementation of the seedForth [1] target. A softcore stack processor needs hardware implementations of stacks. *Smart Stacks* are an approach based on abstracted stack operations rather than register level control . An innovation of smart stacks is that they offer built-in exception handling.

Stacks have been implemented countless times before in VHDL and whilst we are not aware of similar ideas in publication, we do not claim originiality.

A stack as memory

The simplest stack is memory with a stack pointer. An illustrative VDHL entity is shown below:

(During this paper we present code snippets in VDHL which are intended to be self-explanatory. For readers not familiar with VHDL, it suffices to note that VHDL design units are known as entities, and that each entity comprises an interface plus one or more architectures. VHDL designs are hierachical, so that entities may instantiate other entities within their design.)

```
entity stack_1 is

    generic(width : natural;

        depth : natural );

    port(    clk : in std_logic;

        rst : in std_logic;

        input : in std_logic_vector(width - 1 downto 0);

        stack_pointer_n : in integer range 0 to depth - 1;

        write_enable : in std_logic;

        output : out std_logic_vector(width - 1 downto 0);

        stack_pointer : out integer range 0 to depth - 1

    );

end entity;
```

A VDHL module which instantiates this stack will be responsible for providing the new value of the stack pointer `stack_pointer_n` and setting `write-enable` on each clock cycle. If `stack_1` is coded with good-practice VHDL, then the vendor synthesis tools will instantiate it efficiently as FPGA block RAM.

A stack with operations

The example above is simple and efficient, but perhaps not very scalable. We would prefer just to instruct the stack on each clock cycle, and let it take care of write enable and the stack pointer by itself. `stack_2` accomplishes just that by making the `stack_pointer` an output only and adding a `stack_op` input:

```
entity stack_2 is

    generic(width : natural;

        depth : natural );

    port(    clk : in std_logic;

        rst : in std_logic;

        input : in std_logic_vector(width - 1 downto 0);

        stack_op : in stack_op_type;

        output : out std_logic_vector(width - 1 downto 0);

        stack_pointer : out integer range 0 to depth - 1;

        err_under : out std_logic;

        err_over : out std_logic

    );

end entity;
```

We have introduced a user-defined VHDL type

```
type stack_op_type is

    (s_nop, s_push, s_drop, s_replace, s_reset);
```

`stack_2` knows what is should do with the stack pointer and write enable depending on the operation.

Beyond the expected `s_push` and `s_drop` , `s_nop` is necessary because synchronous hardware logic updates registers every clock cycle. If no update is desired this must be specified. The `s_replace` operation completes the orthogonal set: register write is enabled but the stack pointer remains unchanged. `s_reset` allows the stack to be reset (meaning the stack pointer will be returned to its initialization value) without triggering `rst <= '1'` across the whole design.

Now that `stack_2` is managing its own stack pointer, it must also handle stack overflow or underflow conditions and raise exception signals. These are included in the port interface.

Implementing the stack with operations

`stack_2` is implemented in a straightforward manner around a VHDL `case` statement, as the following except illustrates:

```
case stack_op is

    when s_push =>

        we <= '1';    sp_n <= sp_inc;

    -- other cases



    when s_nop =>

        we <= '0';    sp_n <= sp;

end case;
```

This VHDL coding format is very scalable. As we introduce further complexity into our stacks, we need just set the appropriate signals for each operation in the `case` statement. VHDL synthesis tools handle `case` statements well and

produce optimized and efficient logic from them.

A stack with smart operations

Consider the Forth word `dup`. We could implement `dup` by routing the output of `stack_2` to the input and instructing `s_push`. Only a little more logic is requried for `?dup`, and `drop` is trivial. This stack seems to be quite useful!

But a moment's thought suggests that `swap` is going to be more difficult. We'll need a temporary register in the instantiating entity and the operation will take two clock cycles... and we don't want to even think about `rot`. The problem is that the underlying stack entity, which is built from simple memory, only outputs and inputs the top-of-stack item.

That entity can be modified such that it becomes possible to update the top *three* stack locations in the same clock cycle. Writing behavioral VHDL code to accomplish this is not difficult, but it may not be straightforward (and is beyond the scope of this memo) to write the VDHL code in such a way that the synthesis tools translate the design into an efficient combination of registers and block RAM .

With that modification achieved our repetoire of stack operations expands greatly. `stack_3`, our new entity, also outputs both the top-of-stack and next-on-stack items `tos` and `nos`:

```
entity stack_3 is

    generic(width : natural;

        depth : natural );

    port(    clk : in std_logic;

        rst : in std_logic;

        input : in std_logic_vector(width - 1 downto 0);

        stack_op : in stack_op_type;

        tos : out std_logic_vector(width - 1 downto 0);

        nos : out std_logic_vector(width - 1 downto 0);

        stack_pointer : out integer range 0 to depth - 1;

        err_under : out std_logic;

        err_over : out std_logic

    );
end entity;
```

```
type stack_op_type is

    (s_nop, s_push, s_drop, s_replace, s_reset,

    s_nip, s_replaceAndNip, s_dup, s_ifDup,

    s_swap, s_rot, s_over,

    s_depth

    );
```

`stack 3` is responsible for its own stack maniputations and so we chose to call it a *smart stack*. Two further extensions:

1. `s_depth` writes the value of the stack pointer itself onto the stack
2. `s_replanceAndNip` supports arithmetic operations with the signature `( x1 x2 -- x3 )`

Exception handling

Forth's `catch` and `throw` (Milendorf [2]) necessitate some special stack handling. For example, when a exception is thrown the return stack should be appropriately restored so as to facilitate onward program flow after the exception. Implementations of exception handling in Forth typically rely on hooks provided by the Forth virtual machine to read and write stack pointers directly.

Such an approach is also possible in a softcore Forth processor, but there are reasons to hesitate:

1. If we allow software to update stack pointers we rupture the encapsulation that abstracts stacks as hardware entities which ought to manage themselves.
2. Thinking from a hardware perspective might identify a better-performing and more efficient way to accomplish exception handling.

A smart stack with an embedded exception stack

Let's bring two of our stacks together in a single module. We will instantiate `stack_2` in parallel with `stack_3` inside a new entity, `stack_4`, and expand our set of stack operations.

Here are the three new operations concerned with exception handling:

```
s_saveSP, s_restoreSP, s_dropSP
```

1. `s_saveSP` sets up a new exception frame. The stack pointer of `stack_3` is pushed onto `stack_2`.
2. `s_restoreSP` throws an exception. The stack pointer of `stack_3` is updated with the top-of-stack value from `stack_2`, which is simultaneously popped off the stack.
3. `s_dropSP` completes exception handling when a subroutine exits normally. The top-of-stack value of `stack_2` is dropped but the stack pointer of `stack_3` is not affected. In this way the exception frame is discarded.

The instantiation of `stack 2` has taken the role of an embedded exception stack. `stack_4` now encapsulates exception handling through appropriate stack operations.

Exception handling is therefore fast (the stack pointer can be updated in a single clock cycle) and atomic (exception handling is fundamental operation rather then being written in software which could itself be liable to exceptions).

Using smart stacks with exception handling

Our seedbed softcore processor utilizes a number of stacks, principally the parameter stack, the return stack and a subroutine stack. Each of these has an embedded exception stack. The processor implements global exception handling by passing a relevant exception instruction to all of the stacks, which in turn handle the exception locally.

Incidentally, mimicking the actual behavior of Forth's `catch` and `throw` requires a slightly expanded set of operations, for example:

```
s_saveSPAndPush, s_restoreSPAndPush, s_dropSPAndDrop
```

take care of requirements such as recyling the throw code to the top-of-stack after throwing an exception, or placing a zero on stack after dropping an exception. Implementing these additional operations in the VHDL `case` statement is straightforward since the instantiations of `stack_2` and `stack_3` are separate entities which can be controlled independently.

seedbed is both a vehicle to extend experimentation with seedForth into hardware, and a successor to the N.I.G.E. Machine [3]. The N.I.G.E. Machine incorporated hardware exception handling on a global level [4] but the approach described in this paper is certainly more elegant.

Conclusion

We have developed a *smart stack* approach to hardware stacks in VDHL which focuses on abstraction and scalability. Combining two smart stacks, encapsulated as a single entity, provides simple exception handling.

This work is a spin-off of our research and development in seedForth. We welcome correspondence.

Ulrich Hoffmann (FH Wedel University of Applied Sciences), uh@fh-wedel.de

Andrew Read, andrew81244@outlook.com

References

[1] http://www.complang.tuwien.ac.at/anton/euroforth/ef18/papers/

[2] http://www.euroforth.org/ef98.html/

[3] http://www.complang.tuwien.ac.at/anton/euroforth/ef12/papers/

[4] http://www.complang.tuwien.ac.at/anton/euroforth/ef14/papers/

EuroForth 2020
Extending the VALUE concept

Abstract

For many years, variables that return their addresses have been a standard part of Forth. Some years ago, an alternative concept of values, that return their contents, was introduced. This has proved to be so useful, that we have extended the concept to elements of different size, arrays and structures.

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

We were all brought up with VARIABLEs, and for many years we were perfectly happy with them. Their operators @ (fetch) and ! (store) were placed after the variable, in the proper Forth-like way. Then along came VALUEs, which turn out to be rather useful, especially if you think about them in a Forth-like way too. It turns out that they get really very useful if their concept is extended to other types of data storage.

2. Why VARIABLEs are not so great

It's not VARIABLEs themselves, it's their operators. If @ and ! were used *only* for VARIABLEs, that would be fine. But they're also used to access elements in structures and arrays. And the problem is, you don't feel you know exactly what they do. Actually, what you know is, they fetch and store data that is the same size as the version of Forth (note: not necessarily the size of the operating system). But what if you are trying to write code that works in different sizes of Forth? Suddenly there are horrible problems. The access of structure elements of fixed sizes doesn't work any more.

3. Why VALUEs are great

I list the reasons from least important to most important.

a) VALUEs return their content, not their address. It is easy to observe in most applications that there are far more fetches than stores. Therefore, using a VALUE instead of a VARIABLE results in a small simplification of the code. There is a direct correlation between code complexity and bug infestation! So, VALUEs should result in fewer bugs.

b) VALUEs are initialised at compilation time.
In theory this saves explicit initialisation in the code, but in practice we find that one does not often know an appropriate initialisation value at coding time.

c) If all VARIABLES are replaced by VALUEs, you don't need @ and ! any more. For those cases where you still need to fetch and store from arrays or structures, you can use words that specify the size that you need explicitly, such as C@, W!, I@, L! etc.

d) VALUEs work in just the same way as locals. Note that to avoid confusion, I never refer to "local variables" precisely because locals *do not* act like VARIABLEs. We are great believers in locals, because they improve code readability.

4. Operators or Modifiers?

VALUEs on their own, just returning their content, are quite happy in Forth. They behave just like CONSTANTs. However, when you need to store, or do other things to a VALUE, it seems at first to look un-Forth-like.

```
123 -> MYVALUE
```

The "operator" appears before the VALUE. To a brain steeped in Forth, this looks all wrong. However, if you decompile the result, you will find no sign of the word "->". What this has actually done is to modify the compilation action for the following VALUE.

As soon as you stop referring to words like "->" as operators, and start calling them "modifiers" instead, then all your Forth instincts are satisfied again.

As a further improvement, rather than fill our code with "magic numbers" we have also used an enumeration to describe all the modifiers.

```
ENUM VALMODS {
  VMOD@        \ Fetch
  VMOD!        \ Store
  VMODADDR     \ Address
  VMODINC      \ Increment
  VMODDEC      \ Decrement
  VMOD+!       \ Add
  VMODOFF      \ Zero
  VMOD-!       \ Subtract
  VMODSIZE     \ Size of
  VMODSET      \ Set
};
```

5. Extending the concept to different data sizes

In my opinion, only one additional size is needed - for floating point values.
This is already provided, along with its local equivalent, in some compilers but not others.

```
1.2E3 FVALUE MYFVAL  ok
MYFVAL F. 1200.  ok
4.56E7 -> MYFVAL  ok
```

6. Extending the concept to arrays

In my opinion, this is where things get really useful.

a) VINDEX

I propose VINDEX to create an indexed array of values.

```
: VINDEX nsize "name" -- ;
Exec: (unmodified) nindex -- ncontents
```

All the modifiers that can be used with VALUES are available.

```
10 VINDEX MYINDEX  ok
12 1 -> MYINDEX  ok
34 2 -> MYINDEX  ok
1 MYINDEX . 12  ok
2 MYINDEX . 34  ok
1 SIZEOF MYINDEX . 4 ok
```

Note that the SIZEOF applies to the individual element, not to the number of elements in the array.

Because we have not found value initialisation to be very useful, no initial value is specified for VINDEX. All elements are initialised to zero.

There seems to be no consensus about whether array indices should be zero based (to keep programmers happy) or one based (to keep "normal" people happy). Therefore, my implementation of VINDEX actually creates space for both possibilities.

```
10 0 -> MYINDEX  ok
100 10 -> MYINDEX  ok
0 MYINDEX . 10  ok
10 MYINDEX . 100  ok
```

Because -> is a modifier, not an operator, then the statements
```
12 1 -> MYINDEX
```
and
```
12 -> 1 MYINDEX
```
have the same effect.
However, in practice, the index itself is often a VALUE, and the statements
```
12 MYVAL -> MYINDEX
```
and
```
12 -> MYVAL MYINDEX
```
do not have the same effect.
Therefore, we have started as good programming practice always placing the modifier immediately before the word that it is modifying, even when the index is not a VALUE.

An interesting possibility now presents itself. We have seen that incorrect calculation of an index is a very common software bug, and can be very hard to trace.

It is now easy to constrain the index value applied to a VINDEX, and report the error.

```
1 11 -> MYINDEX
Invalid index 11 for MYINDEX length 10
 ok
```

The error message is reported on the terminal window, if in debug mode, or added to a log file if in normal run mode.

b) VMATRIX

I propose VMATRIX to create a two dimensional indexed array of values.

```
: VMATRIX nsizex nsizey "name" -- ;
Exec: (unmodified) nindexx nindexy -- ncontents
```

This operates in exactly the same way as VINDEX but with two indices.

```
10 20 VMATRIX MYMATRIX  ok
12 3 4 -> MYMATRIX  ok
3 4 MYMATRIX . 12  ok
```

It also checks for valid indices:

```
1 10 21 -> MYMATRIX
Invalid index 10 21 for MYMATRIX length 10 20
 ok
```

7. String indices

Following on from the use of VINDEX, I realised that the same concept could easily be used to create arrays of strings.

```
: STRINDEX narraysize nmaxlen "name" -- ;
Exec: (unmodified) nindex -- addr
```

In this case, being a string, the use of the unmodified name simply returns the string address. However, for a nice compatibility, -> does the string store.

```
10 100 STRINDEX MYSTRS  ok
Z" abc" 1 -> MYSTRS  ok
Z" xyz" 2 -> MYSTRS  ok
1 MYSTRS Z$. abc ok
2 MYSTRS Z$. xyz ok
1 SIZEOF MYSTRS . 100  ok
```

A possible security enhancement, not yet implemented, would be to constrain the length of the string during a store, and report errors in the same way as an indexing error.

8. Structures with values

Forth structures can have elements of any size - byte, word, int, cell, string etc. Normally, the word that defines the element returns a calculated address. A frequent problem for the programmer is to use the correct operator (C@. W@ etc.) for the size of the element. Mistakes happen, are often disastrous and can be very hard to find.

It occurred to me that if the elements of a structure were like values, then this type of mistake could be eliminated.

```
: VFIELD structlen size "name" -- structlen' ; ??? -- ???
```

defines a value type field of arbitrary length.

The following words are added for all standard sizes:

```
: VBYTE       _BYTE     VFIELD  ; \ Value type byte field
: VWORD       _WORD     VFIELD  ; \ Value type word field
: VINT        _INT      VFIELD  ; \ Value type int field
: VLONGLONG   _LONGLONG VFIELD  ; \ Value type longlong field
```

A demonstration:

```
STRUCT MYSTRUCT
  VINT         my1
  VWORD        myword
  VBYTE        mybyte
  100  VFIELD  mystring
END-STRUCT

MYSTRUCT BUFFER: MYSTR


: TESTER
  123 MYSTR -> my1
  45  MYSTR -> myword
  67  MYSTR -> mybyte
  Z" qwerty" MYSTR -> mystring
  MYSTR my1 .
  MYSTR myword .
  MYSTR mybyte .
  MYSTR MYSTRing z$. SPACE
  MYSTR SIZEOF my1 .
  MYSTR SIZEOF myword .
  MYSTR SIZEOF mybyte .
  MYSTR SIZEOF MYSTRing .
;

tester 123 45 67 qwerty 4 2 1 100  ok
```

9. Dynamic creation of value arrays

In a previous paper, I explained how VALUEs were created automatically when an XML file, created by the GTK visual design program GLADE, was loaded. Creating VALUEs programmatically requires a different form:

```
: ZVALUE z$name  ival -- ; Exec: (unmodified) -- val
```

It is also very useful to be able to create arrays of different elements sizes programatically. Therefore, alternative forms for these are also provided, such as:

```
: ZVINDEX nsize z$name -- ; Exec: (unmodified) nindex -- ncontents
```

These are used when creating system configuration settings directly from an SQL table, as I will explain in another paper.

10. Future ideas

In my continued search for ways to make code more concise and readable, I am always looking for how the best features of other languages can be adapted to Forth.

The PHP language is particularly good in its use of the results of an SQL query. The column names of the query result are useable in the code.

It should be possible to do something similar in Forth by dynamically creating locals, and loading them automatically with the values of the results. All SQL results are in string form, and numbers have to be converted. But by looking up the data type of a column in the result, it should be possible to convert automatically.

It might look something like this:

```
MAXOPERATORS VINDEX   OPNUMS
MAXOPERATORS STRINDEX OPNAMES

: GETOPS ( --- ) \ Get operator names & nos. from SQL into memory
  SQL| SELECT opnum,opname FROM OPERATORS |SQL> IF    \ Run query
    PROWS 0 DOROW                                    \ Each row
      ropnum  I -> OPNUMS                            \ Save no.
      ropname I -> OPNAMES                           \ Save name
    LOOP
  THEN
;
```

Here I imagine that ropnum and ropname are automatically created locals, named by adding a prefix to the column name of the result. "ropnum" returns a number, because it has looked up the column type, and because it is a numeric type, it has done the conversion from a string. "ropname" returns an address because the column type is some sort of string.

As yet, I have not figured out how to create locals dynamically!

11. Conclusion

The concept of VALUEs instead of VARIABLEs is quite useful on its own. When extended to include arrays and structures, it becomes very useful indeed.

The Grand Recognizer Unification

Bernd Paysan
net2o

M. Anton Ertl*
TU Wien

Abstract

There is an obvious similarity between the search order and a recognizer sequence, which has led to similarities in proposed words (e.g., `get-recognizer` is modeled on `get-order`). By turning word lists into recognizers, we unify these concepts. We also turn recognizer sequences (and be extension the search order) into a recognizer, which allows nestable recognizer sequences and wordlist sequences in the search order. The implementation becomes simpler, too.

1 Introduction

Data uniformity is a useful principle in programming, because it means that we can use the same program on more different data. Examples of uniformity are Unix's principles of 1) accessing many different kinds of things (files, devices, pipes) as file (using open, close, read and write) and 2) to organize files as sequences of bytes. Object-oriented programming allows to treat, e.g., circles and triangles uniformly as graphical objects, with higher-level code being able to e.g., draw graphical objects.

In the Forth world, examples of uniformity are words encompassing colon definitions, constants, variables, etc., and cells encompassing addresses, signed and unsigned integers.

Different types of data can be usefully unified if they have commonalities. In this paper we unify four things: recognizers, wordlists, recognizer sequences, and the search order. Their commonality is that you pass a string to them, and they either recognize it (and produce some data representing the string), or not (and produce a *not-found* result); see Fig. 1.

Section 2 gives examples of how each of the three other concepts works as recognizer, how you can implement existing interfaces, and in some cases how you can implement the concept. Section 3 describes the implementation of wordlists as recognizers in Gforth. In Section 4 we discuss related work.

*anton@mips.complang.tuwien.ac.at

	wordlist	recognizer
one	wordlist `find-name-in`	recognizer `execute`
many	search order `find-name`	recognizer sequence `recognize`

Figure 1: Similar concepts and searching words for recognizing a string

2 Grand Unified Recognizers

2.1 Wordlists as recognizers

We start by generalizing wordlists to also be recognizers: The *wid* is implemented as the xt of a `rec-nt`-like[1] recognizer that recognizes the words in the wordlist. Note that this requires changes pretty deep in the the Forth system (see Section 3), that's why we do not propose this change for standardization. The resulting wordlist recognizers have the stack effect

```
( c-addr u -- nt rectype-nt | rectype-null)
```

The benefit is that we can now use wordlists wherever recognizers are expected, e.g., in recognizer sequences (see Section 2.3).

But first, how can we use it as a wordlist? You can implement `find-name-in` as follows:

```
: find-name-in ( c-addr u wid -- nt | 0 )
    execute rectype-nt <> if 0 then ;
```

2.2 Recognizer sequences as recognizers

This part has been proposed[2] as a potential part of the recognizer proposal.

It is straightforward to implement a recognizer sequence as recognizer:

[1] https://forth-standard.org/proposals/recognizer
[2] https://forth-standard.org/proposals/nestable-recognizer-sequences

```
: rec-sequence ( xt1 .. xtn n "name" -- )
  create dup , dup , 0 ?do , loop
does> ( c-addr u -- ... rectype )
  {: c-addr u addr :}
  addr cell+ @ cells addr 2 cells +
  dup >r + r> ?do
    c-addr u i @ execute
    dup rectype-null <> if
      unloop exit then
  1 cells +loop
  rectype-null ;
```

This implementation stores the maximum size in the first cell, and the current size in the second cell, followed by the recognizers in the sequence.

The benefit of having recognizer sequences as recognizers is that we can, e.g., put it in another recognizer sequence, i.e., recognizer sequences become nestable. So even if each sequence is short, a sequence can contain an unlimited number of basic recognizers.

We have the following accessor words for recognizer sequences:

```
\ we expect the following words
\ is-defer? ( xt -- f )
\ is-rec-sequence? ( xt -- f )

\ helper word
: follow-defers ( xt1 -- xt2 )
  begin
    dup is-defer? while
      defer@
  repeat  ;

: get-rec-sequence ( xt -- xt1 .. xtn n )
  follow-defers dup is-rec-sequence? 0= if
    drop 0 exit then
  >body cell+ dup cell+ over @
  dup >r cells rot + ?do
    i @ -1 cells +loop
  r> ;

: set-rec-sequence ( xt1 .. xtu u xt -- )
  follow-defers
  dup is-rec-sequence? 0= -12 and throw
  >body {: u addr :}
  u addr @ > -49 and throw
  u addr cell+ !
  addr 2 cells + dup u cells + rot ?do
    i !
  1 cells +loop ;
```

2.3 Search order as recognizer

With wordlists as recognizers, we can implement the search order as a recognizer sequence:

```
: rec-nothing ( c-addr u -- rectype-null )
  \ recognizer that recognizes nothing
  2drop rectype-null ;

' rec-nothing dup 2dup 2dup 2dup 2dup
  2dup 2dup 2dup 16 rec-sequence rec-nt0

: get-order ( -- wid1 ... widu u )
  ['] rec-nt0 get-rec-sequence ;

wordlist constant root-wordlist

: only ( -- )
  root-wordlist 1 rec-nt0 set-rec-sequence ;

: set-order ( wid1 ... widn n -- )
  dup -1 = if drop only exit then
  ['] rec-nt0 set-rec-sequence ;
```

You can put non-wordlist recognizers in the search order, if they have the stack effect

```
( c-addr u -- nt rectype-nt | rectype-null)
```

In particular, you can put recognizer sequences containing wordlists in the search order. Putting other kinds recognizers in the search order will result in `find-name` not working properly.

Locals

One issue in implementing standard Forth is how to find locals. The standard does not really specify the details, but the text interpreter certainly has to find them when they are in scope, they probably should not be in the search order, and whether `find-name` (or `find`) should find them is not entirely clear and has been answered differently by different systems.[3]

I think that `find-name` should find locals, and here I show how that can be done. I assume that there is a system-specific `rec-loc` that recognizes locals and behaves like `rec-nt` (it can be implemented as a temporary wordlist, or with a separate mechanism). First, we implement `rec-nt`:

```
defer rec-nt

' rec-nt0 ' rec-loc 2 rec-sequence rec-locals

: activate-locals ( -- )
  ['] rec-locals is rec-nt ;
: deactivate-locals ( -- )
  ['] rec-nt0 is rec-nt ;

deactivate-locals
```

With that, `find-name` is easy to implement:

[3] https://forth-standard.org/standard/locals#reply-426

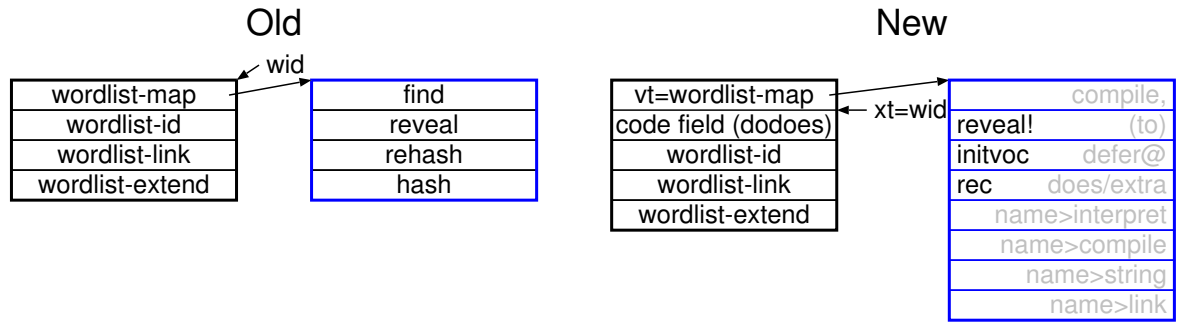


Figure 2: Wordlist implementations in Gforth

```
: find-name ( c-addr u -- nt|0 )
  ['] rec-nt find-name-in ;
```

3 Wordlist Implementation

The previous section shows possible implementations of recognizer sequences and the search order. This section discusses the implementation of wordlists.

Figure 2 shows two implementations of a wordlist in Gforth. We look at the old implementation first, because the new implementation is based on it.

3.1 Old wordlist implementation

In the old implementation a wordlist is a structure with the following fields:

wordlist-map This field points to a virtual method table (map) of methods for the wordlist; the methods will be explained below.

wordlist-id This (badly named) field contains the start of the linked-list representation of the wordlist. In hashed wordlists the linked-list representation is not used for searching (for performance reasons), but it is used for `words` and `traverse-wordlist`. For hashed wordlists it is also used as persistent representation of the wordlist, from which the hashed representation can be recreated whenever it is convenient.

wordlist-link This field points to the next wordlist in the linked list of wordlists (`voclink`), used in various places where all wordlists or all words are needed.

wordlist-extend This field may contain additional data. In hashed wordlists it contains a unique number for the wordlist, which is used to ensure that two words with the same name, but in different wordlists are sorted into different hash table buckets.

The method fields in the wordlist-map contain the xts of the method implementations. In the old implementation, methods were invoked with sequences like

```
dup wordlist-map @ reveal-method perform
```

instead of with separate words. The method fields are:

find-method `( addr len wid - nt|0 )` `find-name-in` for the wordlist; E.g., for a normal `wordlist`, this implements a case-insensitive hashed lookup.

reveal-method `( nt wid - )` inserts nt into the wordlist.

rehash-method `( wid - )`

hash-method `( wid - )` Both the hash and the rehash methods are used nowadays for putting all the entries of a hashed wordlist into the hash table (do nothing for a linked-list wordlist). There used to be a difference between the methods in earlier times.

3.2 New wordlist implementation

The new wordlist implementation is a nameless recognizer word (cf. Figure 3 of [PE19]). The fields are essentially the same as in the old implementation: `wordlist-id`, `wordlist-link` and `wordlist-extend` are exactly the same; they reside in the parameter field of the nameless word.

A new field is added: the code field of the word; the wordlist recognizers are implemented as `create...does>` words, so the code field contains `dodoes`[4].

Both wordlists and (with the new Gforth header [PE19]) words have a virtual method table, pointed to by `wordlist-map` in wordlists, and by vt in word headers. We chose to use the vt as `wordlist-map`.

[4] `Dodoes` is the code address of a piece of native code that pushes the parameter field address and then executes the xt in the does/extra field

```
find-method ( addr len wid -- nt|0 )
```

is replaced by

```
rec-method ( addr len wid --
      nt rectype-nt | rectype-null )
```

This rec-method must be invoked by `execute`ing the wid, so the does/extra method serves as the rec-method.

The other methods now have invocation words: `reveal!` for `reveal-method`, and `initwl` for what used to be `rehash-method`/`hash-method`[5].

For the `reveal!` and `initwl` methods there are no natural places in the word header. We reused the `(to)` (aka `defer!`) entry for `reveal!` and the `defer@` entry for `initwl`. This means that users who apply `defer@` and `defer!` to a wid will not get an error message, but some other behaviour that is probably unexpected; but then, it is typical of Forth's approach to type-checking that the caller of a word only passes parameters of the right type (i.e., only the xts of deferred words to `defer@`), and gets arbitrary behaviour otherwise.

Alternative approaches would have been to 1) append additional entries for the `reveal!` and `initwl` methods to the virtual method table; or to 2) have a wordlist-map field separate from the vt, which points to a method table containing the `reveal!` and `initwl` method xts.

Implementing wordlists as Gforth words also offers other options that we have not implemented (yet): `Name>string` could return the name of the vocabulary or the constant associated with the wordlist (if there is one), making it easier to implement, e.g., `order`. `Name>link` could follow `wordlist-link`, allowing to implement `voclink` (the list of wordlists) as a wordlist itself.

4 Related work

Matthias Trute developed the idea of recognizers into a real, workable implementation [Tru11] and proposal for standardization [Tru15].

Ertl [Ert15, Section 4.1, 5.1] has presented other unifying approaches: A variant of recognizers that generate temporary (or permanent) words, with `find-name` to find them. The paper also discusses the disadvantages of these approaches, and these have eventually led to deciding against these and for the RfD approach.

Ertl mentioned the idea of a "virtual wordlist consisting of a changeable search order of sub-wordlists" in 2003[6].

[5] There has been no difference between these methods in recent years, so they have been combined into one; also, we have not given this method a field name.

[6] <2003Apr18.141759@a0.complang.tuwien.ac.at>

5 Conclusion

By unifying recognizers, wordlists, recognizer sequences, and the search order we can write words which can deal with all these things. The benefits already start when implementing these concepts: the search order becomes just another recognizer sequence, simplifying the implementation of `find-name`, `get-order` and `set-order`.

References

[Ert15] M. Anton Ertl. Recognizers — why and how. In *31st EuroForth Conference*, pages 77–78, 2015. 4

[PE19] Bernd Paysan and M. Anton Ertl. The new Gforth header. In *35th EuroForth Conference*, pages 5–20, 2019. 3.2

[Tru11] Matthias Trute. Recognizer — interpreter dynamisch verändern. *Vierte Dimension*, 27(2):14–16, 2011. 4

[Tru15] Matthias Trute. Forth recognizer — request for discussion. 3rd RfD, Forth200x, 2015. 4

Stephen Pelc
MicroProcessor Engineering Ltd
133 Hill Lane
Southampton SO155AF
UK

http://www.mpeforth.com

stephen@mpeforth.com

The VFX Forthv5.1 kernel incorporates dual-behaviour words and recognisers. This talk discusses our experience over the last year with these changes. Dual-behaviour words are a standards-compliant solution to needing words that have separate interpretation and compilation behaviour. Previous papers called these words NDCS words (non-default compilation semantics). Recognisers are a fashionable solution to providing a user-extensible text interpreter. Our experience converting two OOP packages to use recognisers is discussed, together with problems involved in supporting two floating point formats.

Introduction

VFX Forth v5.1 contains a different approach to compilation semantics in Forth and it incorporates recognisers.

Other Forths have different approaches to handling the general problem first defined in ANS Forth, that of "non-default compilation semantics" (NDCS) but in general the solutions involve defining separate actions for the interpret and compile time actions. For example, the definition of **DO** is:

```
: DO            \ Run: n1|u1 n2|u2 -- ; R: -- loop-sys ; 6.1.1240
 NoInterp  ;
ndcs: ( -- )  s_do,  3  ;
```

Our approach to this problem has been described in EuroForth 2018 and 2019 papers.

Recognisers are this decades's solution to the problem of providing a text interpreter that is extensible by the application.VFX Forth uses them for installing two different floating point handlers (NDP and SSE) and for handling two different OOP packages (ClassVFX and CIAO).

Dual behaviour words

The definition below shown how the separate interpretation and compilation actions do not interfere with each other.

```
: ."            \ "ccc<quote>"  --
\ *G Output the text up to the closing double-quotes character.
  [char] " word $.  ;
ndcs: ( -- )  compile (.")  ",  ;
```

Since we introduced this notation, we have had no problems with it and the code passes all the tests we have tried.

Recognisers

The text interpreter in VFX Forth is basically as below. It is assumed that **recognize** returns the address of a type structure holding the interpret, compile and postpone actions of the returned type.

```
: interpret \ --
  begin  ?stack parse-name dup
  while  forth-recognizer recognize  state @ abs cells + @ execute
  repeat
  2drop
;

: postpone  \ "<name>" -- ; POSTPONE <name> ; 6.1.2033
  parse-name forth-recognizer recognize  2 cells + @ execute
; immediate
```

For us, the big benefit of recognisers is to be able have fine grain control of the parsing. For example, NDP and SSE floats require different handlers; SSE is limited to 64 bit floats whereas NDP can handle 80 bit floats. In an environment which uses Forth source libraries, we may need to be able to switch between OOP packages. When switching to a new OOP package, we have to be able to remove float package notations completely. To do this we need vocabulary search order control as well as recogniser order control.

Recogniser order control is a relatively new idea, but we have been using wordlist search order control for decades. ANS Forth introduced a clumsy pair or words.

```
: GET-ORDER     \ -- widn...wid1 n  ; 16.6.1.1647
\ *G Return the list of WIDs which make up the current search-order.
\ ** The last value returned on top-of-stack is the number of WIDs
\ ** returned.

: SET-ORDER     \ widn...wid1 n -- ; unless n = -1 ; 16.6.1.2197
\ *G Set the new search-order. The top-of-stack is the number of WIDs
\ ** to place in the search-order. If N is -1 then the minimum search
\ ** order is inserted.
```

A more useful pair (invented at Forth Inc.) for daily use is:

```
: -ORDER   \ wid --
\ *G Remove all instances of the given wordlist from the *\fo{CONTEXT}
search order.

: +ORDER   \ wid --
\ *G The given wordlist becomes the top of the search order. Duplicate
\ ** entries are removed.
```

A similar pair can easily be designed for recognisers. I'm not going to get into the naming arguments that recognisers have suffered from for several years. Later recogniser proposals seem to have adopted the idea of a recogniser "stack" which is really just an ordered array. It is possible to use the same structure mechanism for both recognisers and wordlist search order control.

At present, most systems that use recognisers do it in very similar ways. However, the recogniser management words as present in most proposals are inadequate or ugly. For full use of recognisers, wordlist control is equally necessary. For example, if I have to switch between NDP and SSE floats as I may have to do when interfacing to macOS or Linux, I may need to select

```
  NDPfloats ( — )
  SSEfloats ( — )
  integers ( — )
```

These three words need to manipulate both recognisers and wordlists. Once we can do this in a sane way, we can then use the same techniques to switch between OOP packages. In turn this enables us to compile libraries that use different OOP packages without conflict. MPE supported five OOP packages in VFX Forth 32. It's a dreadful position to be in. Since none of the OOP designers in the Forth world seem prepared to compromise on a standard notation, the only solution I can see is for their parsers and compilers to be removable and installable at will. For MPE, this aspect of recognisers is the convincing use case; recognisers for literals did nothing for us as we already supported the common notations.

From the user's point of view, nobody really notices recognisers and they have caused no technical support except for the one or two users who wanted to expand notations. Once explained, they went away happy.

From a standards point of view, the original simple set of words has proven to be enough. We (MPE) are firmly convinced that trying to automate the postpone behaviour is potentially dangerous as we have no idea what clever Forth programmers will get up to with recognisers. The difficult part of recognisers is the set of recogniser management words.

Converting two OOP packages to recognisers

There are two OOP packages in VFX that needed to be rock-solid with recognisers. CIAO (C Inspired Active Objects) was written over 20 years ago by a long-departed programmer to ease interfacing to C++. The package has its fans.

ClassVFX was designed by and is used by Construction Computer Software at the heart of a large application of 1.4 million lines of Forth source code.

As with many other complex and capable OOP systems in Forth, both packages take over 1000 lines of source code and contain ugly code. As a result, the original code was not thrown away for a new version, but the code was hacked to fit the parser and action model of recognisers. The parser actually performs all the required interpretation and compilation actions - yes, it remains state-smart as it always has been, and the action structure contains **NOOP**s for the interpretation and compilation actions, and a **THROW** for the **POSTPONE** action.

This rather brutal approach to code conversion was somewhat regrettable, but did prove the robustness and flexibility of recognisers. People with time on their hands are welcome to design a new implementation that manages to fully separate the parsing and action parts, but I haven't found a way yet.

CIAO previously used a hook in the text interpreter loop and took full advantage. To provide the same behaviour with recognisers, two parsers and action tables are required. One parser runs first in the recogniser sequence, and the other runs last.

Full source code is provided in the downloadable versions of VFX Forth.

Acknowledgements

Gerald Wodni persuaded me that recognisers have merit.
Anton Ertl convinced me that all standards have bugs.
Willem Botha taught me about big Forth applications.
Pat Gillespie has tolerated my apparent refusal to retire.

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.

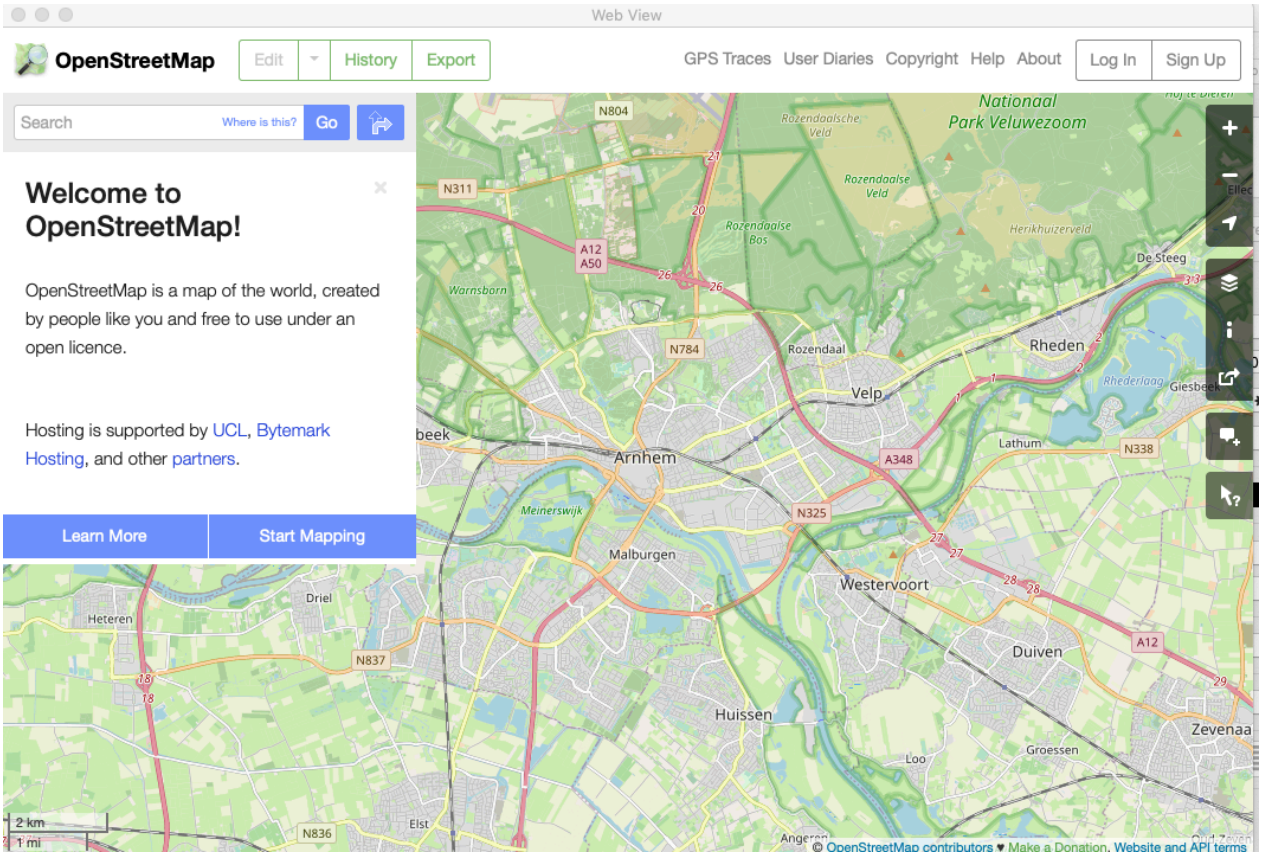
We did reasonably well with the assembler testing. However, testing an assembler before you really know how it is going to be used just leads to testing without exploring the important corner cases. That's why good testers have a mentality of their own that has to be respected.

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

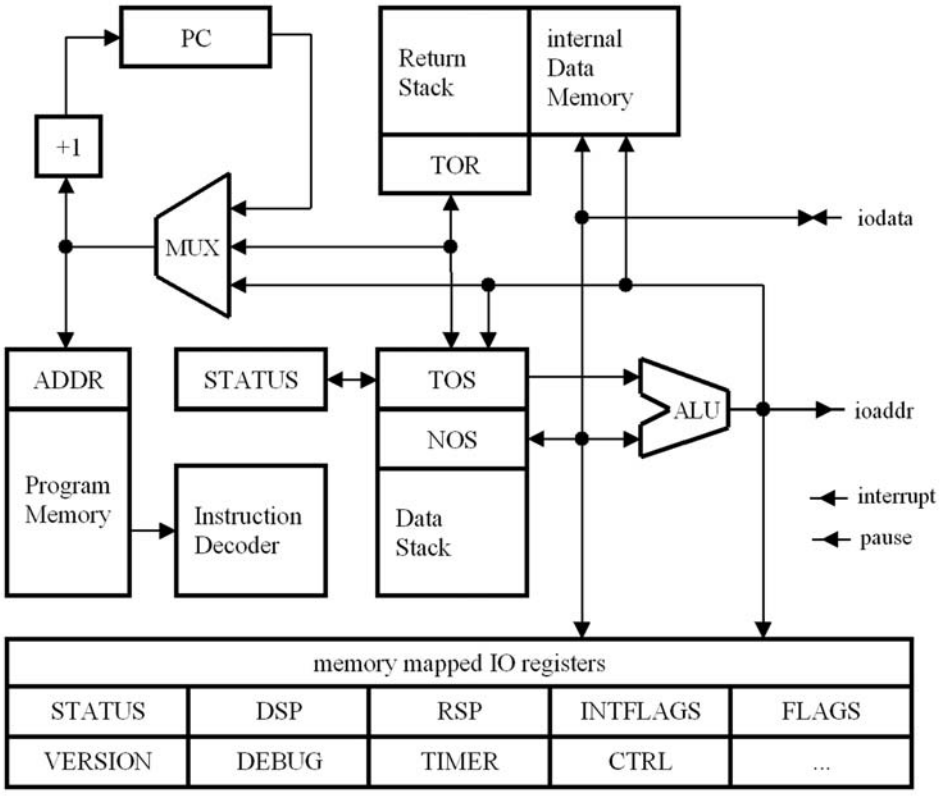
µCore Overview

Klaus Schleisiek - kschleisiek@freenet.de

µCore (pronounced "microCore") is a processor written in VHDL for synthesis into FPGAs. It is optimized for embedded real time control. µCore is an embodiment of the virtual machine of the Forth programming language [1] and hence Forth is µCore's assembler.

Characteristics

1. Dual stack (data stack & return stack)
2. Harvard architecture (8 bit program memory & independent data memory)
3. Configurable word width for data memory & stacks
4. Postfix instruction set architecture (literal operands/addresses precede rather than follow instructions that need them in the instruction stream)
5. Single cycle execution, no pipelining
6. Interrupt & Pause traps
7. Multiple data and return stack areas for efficient multitasking
8. Deterministic program execution
9. Hardware/Software co-design environment
10. Malleable instruction set
11. Instantiation



µCore block diagram

μCore Overview

1 Dual Stack

μCore does not have general purpose registers, it has a "data stack" for numerical computations and a "return stack" for program flow control.

The data stack resides in a private RAM area managed by the DataStackPointer (DSP). Its top two items are held in registers (TopOfStack - TOS and NextOfStack - NOS).

The return stack is mapped into the data memory managed by the ReturnStackPointer (RSP). Its top item is held in a register (TopOfReturnstack - TOR).

The pros-and-cons of a stack versus register architecture:

1) Parameter passing.
   Both architectures quite often suffer from the fact that the arguments for a subroutine call are not in the registers resp. not in the stack order, in which the subroutine expects them. As a result, registers have to be re-shuffled resp. the stack has to be re-ordered.
   The number of registers is finite and hence the number of arguments that may be passed to a subroutine. The stack has a finite depth in hardware as well, but if needed, the bottom part of the stack may be swapped out and in of data memory by a background process at runtime.
2) Interrupt processing
   The stack architecture holds all arguments on stacks already and therefore, no registers need to be saved and restored.
3) Code optimization
   "Register allocation" became a research topic long ago whereas "stack allocation" is an esoteric topic. But three generations of research happened and the results are already of production quality.

   When μCore's assembler μForth is used, the programmer himself is responsible for stack allocation and stack re-ordering.
4) Instruction set impact
   A register architecture needs address bits in the instruction word whereas a stack architecture does without. A subroutine always takes its arguments from the top of stack downwards and leaves its results on the stack.

Looking at the technical merits, the stack architecture has several advantages especially for real time applications. Its major disadvantage is its unfamiliarity.

2 Harvard Architecture

μCore's instructions and the program memory are 8 bits wide. The configurable width data memory (see below) is independent of the fixed width program memory. Two instructions, which allow writing and reading program memory, may or may not be instantiated and therefore, μCore can be made safe from corrupting its program during runtime.

During cold boot, μCore may always write into program memory and therefore, it may fetch its program from an external, non-volatile memory. The boot loader is a μForth program that has to be compiled before synthesis.

3 Configurable Data Memory & Stack Word Width

In μCore the word width of the data memory and the stacks is configurable and must be defined before synthesis as needed by the application. μCore does not deal with 8 bit "bytes". That is data memory is always accessed in units of the instantiated data memory and stack width, which simplifies the memory interface considerably.

The instantiated word width may be any odd or even number of bits. 12 bits is a practical minimum, because it limits the program memory's address range to 4096 instructions. The maximum width is only limited by the available FPGA resources. This configurability is rendered possible by μCore's postfix instruction set architecture (see below).

4 Postfix Instruction Set Architecture

μCore's postfix instruction set extends Forth's postfix syntax into the hardware realm. The principle has been inherited from the Transputer, and it makes the configurable data word width possible.

When instructions include immediate numbers (literal, offset, address etc.), the number of bits needed for these numbers depends on the processor's data word width. In μCore, literals and opcodes are kept separate and literals precede rather than follow opcodes that need them. Numbers of any magnitude may be constructed by a sequence of literal instructions.

μCore's instructions are 8 bits wide. An instruction's most significant bit determines whether it is interpreted as an opcode (MSB not set) or as a literal (MSB set). A "lit flag" in the status register is set by literals and reset by opcodes. This leads to four different cases for instruction interpretation:

lit flag	MSB	interpretation
0	0	An opcode that does not need a literal.
0	1	A literal following an opcode. The least significant 7 bits are pushed on the stack as a 2s-complement number in the range -64 .. 63, and the lit flag is set.
1	0	An opcode following a literal. If it is just a noop, the literal remains on the stack as a number. The lit flag is reset.
1	1	A literal following another literal. The literal in TOS is shifted 7 bit positions to the left and the 7 least significant bits of the instruction are appended. This covers the following number ranges: 1 lit -64 .. 63 2 lits -8192 .. 8191 3 lits -1048576 .. 1048575 etc. This way numbers of any magnitude can be constructed.

Opcodes do neither include literal information nor register addresses due to the stack. Therefore, each of the remaining 127 opcodes can be used for different semantics. This is plenty. The "core" opcode set has 47, the "extended" and "float" set another 29 opcodes. This leaves room for 42 application specific opcodes.

Opcodes like branch and call do even have different semantics depending on the lit flag. When it is set, they take the number in TOS as a branch offset. Otherwise, they take it as an absolute target address.

Instructions (literals as well as opcodes) are always self-contained and therefore, interrupts can be accepted after each instruction.

5 Single Cycle Execution, No Pipelining

In general, a single instruction needs one active clock transition to execute.

On a hardware level (VHDL), μCore's registers have a clock and a clock enable input. The enable input allows embedding μCore into environments with a clock frequency that is above μCore's timing requirements. Constant **cycles** in architecture_pkg.vhd defines the number of clock cycles, which are needed to execute one instruction. In addition, the top-level enable input may be used to temporarily halt μCore. Typically, μCore's worst-case asynchronous signal delay is less than 40 ns (25 MHz) for Xilinc, Altera, and Lattice FPGAs.

Reading the FPGA's internal blockRAM memories requires the execution of a sequence of two uninterruptible instructions. The first instruction latches the memory address inside the blockRAM, the second instruction pushes the memory's data output on the stack.

To this end a general mechanism allows to chain sequences of uninterruptible instructions. It can also be used for read-modify-write instructions.

6 Interrupt & Pause Traps

Interrupts are a well known concept in computing and almost every existing processor implements it. Hardly known is its Janus-faced sibling, the Pause, another inheritance from the Transputer. A Pause will e.g. be raised by a UART, when the processor intends to read a character, which has not been received yet. The difference between interrupts and pauses is as follows:

Interrupt: An event did happen that was **not** expected by the software.

On an interrupt the processor pushes the status register on the data stack, raises the InterruptInService (IIS) status bit that disables further interrupts, and executes a call to the interrupt trap location. The status register is a collection of single bit flags that characterize the processor state besides the PC.

Globally, interrupts may be enabled or disabled via the "InterruptEnable" (IE) status bit. Individual interrupts sources are enabled or disabled writing the Intflags register, which holds a flag for each interrupt source. Reading Intflags allows locating an interrupt's source(s).

Interrupt processing is terminated by the IRET instruction, which returns from the interrupt service routine and restores the status register from the data stack, thereby resetting the IIS bit again.

Pause: An event did **not** happen that was expected by the software.

On a pause the processor aborts the current instruction, which caused the pause signal to be raised, does not increment the Program Counter (PC), and executes a call to the pause trap location.

In a single task system, the pause trap would immediately execute an EXIT (return from subroutine), thereby returning to the instruction that caused the pause previously. This loop repeats until the pause signal will no longer be raised.

In a multitask system, the pause trap would call the scheduler, and another task can be given the opportunity to run. Eventually, the task that caused the pause will be activated again and continue normal execution as soon as the reason for raising the pause signal has vanished, i.e. the missing event did happen.

Using the pause mechanism, resource locking can be completely realized in hardware. E.g. an ADC with an integrated 8-channel multiplexer in a multitasking environment can then be programmed in μForth:

```
<channel_number> ADC !
ADC @ Sample !
```

or in C[1] as follows:

```
ADC = <channel_number>;
Sample = ADC;
```

Storing the <**channel_number**> into the memory mapped **ADC** interface will initiate conversion of that channel and set the ADC's semaphor flag in the flags register. Should the ADC be in use by another task, the semaphor will have been set and a pause will be raised until eventually the semaphor will have been reset. In the next line, the conversion result will be read from the **ADC**, stored in varable **Sample**, and the semaphor will be reset. Should the ADC be still busy converting when a read attempt is made, pause will be raised until eventually the ADC has finished conversion.

This way the hardware takes care of both mutual exclusion of the ADC resource as well as waiting for the AD-conversion to finish without having to probe flags in software loops.

7 Multiple Data- and Return-Stack Areas for Efficient Multitasking

The number of data- and return-stack memory areas can be configured so that each task has its own set of stacks. This speeds up task scheduling, because only internal registers TOS, NOS, TOR, DSP, and RSP need to be redirected. A task switch @ 25 MHz takes 7 μsec to put the current task to sleep and start another one.

---

[1] A μCore back-end for the LCC C compiler has been implemented for an earlier 32 bit version of μCore. In addition, LCC itself has been modified for stack- rather than register-allocation. This work has been done at Fachhochschule Aargau, Windisch (CH), supported by the Hasler Foundation.

8 Deterministic Program Execution

μCore's program execution is fully deterministic. There is no pipeline. There is no cache memory that may have to be loaded before the next instruction can be executed. Therefore, μCore's time to execute a certain sequence of instructions is predictable, which is a prerequisite for verifyable real-time systems.

9 Hardware/Software Co-Design Environment

μCore's design environment consists of the following elements:

- VHDL source code for μCore on a target system,
- the μForth cross-compiler and debugger running on a host computer,
- an RS232 umbilical link that connects host and target,
- an interactive command line interpreter to inspect and control the target,
- a disassembler,
- and a single-step tracer.

Both the VHDL hardware description as well as the μForth cross-compiler share a common file **architecture_pkg.vhd**, which characterizes μCore's architecture and opcodes. Therefore, the cross-compiler will always be in-sync with the hardware description.

The cross-compiler is able to produce program memory initialization code for the VHDL simulator and therefore, μForth program execution can be observed in the VHDL simulator.

A umbilical link consisting of a 2-wire UART (rxd, txd) connects μCore's debugger with the mating debugger on the host and allows to

- load object code into μCore for execution,
- control μCore interactively from a command line (Forth style),
- display the data stack and dump data memory to the host's display,
- upload/download data memory areas without delaying μCore's program execution,
- single step code on a subroutine level displaying the data stack at each step.

10 Malleable Instruction Set

μCore's instruction set consists of 47 "core", 24 "extended", and 5 "float" instructions. In addition, 26 "software traps" are available, which do a single cycle call to fixed program memory locations.
42 opcodes are unused or software traps. They may be used for application specific instructions.

In the core version, the extended instructions will be emulated by macros or subroutines. Therefore, a core version has the same functionality as the extended version but it runs slower consuming fewer FPGA resources.

Adding a new opcode to μCore is rather simple and consists of three steps:

1. Define the binary code for **op_newname** as a constant in **architecture_pkg.vhd**.
2. Add a new when clause to the instruction decoder case statement in **uCntrl.vhd** for the semantics of **op_newname**.
3. Define a name for **op_newname** in **opcodes.fs** to make it known to μForth.

μCore Overview

The core instruction set

Data stack:	drop, dup, ?dup, swap, over, rot, -rot
Return stack:	>r, r>, r@
Branches:	branch, 0=branch, next, call, exit, iret
Data memory:	ld, st
Unary arithmetic:	not, 0=, 0<, shift, ashift, c2/, c2*
Binary arithmetic:	+, +c, -, swap-, and, or, xor, um*, *, um/mod
Flags:	status-set, ovfl?, carry?, time?, <
Traps:	reset, interrupt, pause, break

11 Instantiation

μCore has been instantiated on these FPGA families:
Xilinx (XC2S), Lattice (XP2), Altera (EP2), and Actel/Microsemi (A3PE).

Reference implementations on a Lattice LFXP2-8 prototyping board with minimal external IO and hardware multiplier have been made with different μCore configurations. The clock's timing constraint in the synthesizer and in the place-and-route tool had been set to 25 MHz.

Instruction set	word width	SLICES	data memory	program memory	maximum clock
core	16	988	6k	8k	33 MHz
extended	16	1199	6k	8k	30 MHz
core	27	1259	4k	8k	33 MHz
extended	27	1608	4k	8k	28 MHz
extended and floating point	27	1808	4k	8k	26 MHz
core	32	1432	3k	8k	33 MHz

Literature

[1] Leo Brodie: "Thinking Forth", http://thinking-forth.sourceforge.net/

A Note on Parsing Source Code

Corrolary to "Poor Man's Recognizer"

I pondered some more on the relationship between gforth's **[**, **]**, and **parser**.

When the Forth system is used for cross compilation, we must be able to switch back and forth between interpreting and compiling using **[** and **]** - just the same as in a normal host only system. But the **parser** action will be different depending on whether we want to produce code for the host or for the target system.

In gforth, **parser** is deferred, which is already a good starting point for differences in host or target code production, and **[** and **]** are defined as follows:

```
: ]  ( -- )   ['] compiler is parser   1 state ! ;
: [  ( -- )   ['] interpreter is parser   0 state ! ;
```

Bad luck if I have to modify **parser** depending on my cross compiler's needs, because **[** and **]** will overwrite whatever I may have assigned to **parser**. Actually, I invented this almost 40 years ago in volksForth. It seemed a good idea at the time but it was clearly a mistake.

This was one of my bad ideas, which I

A more useful implementation will look like this:

```
: ]  ( -- )   1 state ! ;
: [  ( -- )   0 state ! ;
: host-parser  ( c_addr u -- )
   state @ IF  compiler  ELSE  interpreter  THEN ;
' host-parser IS parser
```

8-Sep-2020, Klaus Schleisiek

Poor Man's Recognizer

µForth is the cross compiler for µCore. When I started the project on top of gforth_062, I found a simple and powerful solution to cross compilation by patching the way `[` and `]` behave. The current version of gforth_079 uses Matthias Trute's recognizer mechanism and I gave them a try as an alternative to patching.

The result of this endeavor was so complicated that I returned to the patch solution, which I regard as much more readable. It can be implemented on all versions of gforth above 062 albeit minute differences, because gforth's interpreter/compiler is a moving target. Alternatively, the patch version can be implemented in a way that it reads like recognizer code, though drastically simplified.

In order to clarify the issues, I will first present and discuss the patch solution, followed by the gforth recognizer solution, and finally followed by the "Poor Man's Recognizer" solution.

µForth has two very different modes of operation:

1. `host-compile` compiles into the host's dictionary in order to add capabilities to µForth.

2. `target-compile` compiles into the target's dictionary in order to produce code for µCore.

These two modes require very different parsers, which are both different from the native gforth parser, because µForth includes an OOP package for the sake of operator overloading (i.e. a `@` may do very different things depending on the object that preceded it). This would be straightforward, if `[` and `]` were deferred, but they are not and therefore, `[` and `]` have to be patched instead. For the sake of argument, I will only discuss the `target-compile` mode.

1 Patching

The patch magic is done in a portable way by `becomes`, which takes an xt and the name of an existing word:

```
: becomes ( <word> new-xt -- ) \ make existing <word> behave as new-xt
   >r   here   ' >body dp !
   postpone AHEAD   r> >body dp !   postpone THEN
   dp !
;
Variable 'interpreter   ' interpreter 'interpreter !
Variable 'compiler      ' compiler    'compiler !

:noname ( -- )  'interpreter @ IS parser   state off ;   becomes [
:noname ( -- )  'compiler @    IS parser   state on  ;   becomes ]
```

After patching, everything still works ok, because we initialized both `'interpreter` and `'compiler` with gforth's native code. Now we can define the new parser for the target.

First some lower level words:

`d>target ( d.host -- d.target )` converts a host's double number into a double number for µCore, because very often the data width of the host is greater than µCore's data width.

`ClassContext`, a variable, points to a linked list of methods. If it is zero, Forth's dictionary will be searched. `search-classes` searches an object's list of methods down its inheritance order.

`debugger-wordlist`, a variable pointing to a wordlist of commands, which will be searched with preference when interactively debugging the target system.

`t_lit, ( n -- )` compiles a number on the host's stack as a literal in the target system.

`>t` transfers a number on the host's stack to the target's stack.

`not-found ( addr len -- )` displays the string that didn't match and aborts.

Now we are prepared to define the target's interpreter/compiler classical style, everything in one single definition:

```
: target-compiler ( addr len -- )
\ search for methods
   ClassContext @ IF  2dup search-classes ?dup
                      IF  nip nip name>int execute  EXIT THEN
                      not-found
                  THEN
\ search for commands while debugging
   dbg? IF  2dup debugger-wordlist search-wordlist
            IF  nip nip name>int execute  EXIT THEN
        THEN
\ search the target's dictionary
   2dup find-name ?dup IF  nip nip name>int execute  EXIT THEN
\ try to convert to an integer number
   2dup 2>r snumber? ?dup 0= IF  2r> not-found  THEN  2rdrop
   comp? IF  0> IF  d>target swap t_lit,  THEN  t_lit,  EXIT THEN
   dbg?  IF  0> IF  d>target swap >t      THEN  >t      EXIT THEN
   drop
;
```

When an object has been compiled or executed, `ClassContext` will have been set to its methods list to be searched. If a method is found, it will be executed producing target code and we are done. If nothing was found, we display the not-found message and abort.

Otherwise, we check whether we are interactively debugging, in which case we will search `debugger-wordlist` next. If it was a command, it will be executed and we are done. If it was no command, we will search the target's dictionary. If it was a target word, it will be executed producing target code and we are done.

If it was neither a command, nor a target word, `snumber?` will try to convert the string into a number. When successful, we may be

1. compiling. In this case, we have to compile the number as literal(s) into the target code and we are done.
2. debugging. In this case we transfer the number on the host's stack to the target's stack and we are done.
3. interpreting. In this case we throw away the double number flag and we are done, leaving the number on the host's stack.

You must admit, this is VERY different from what the native gforth compiler does.

Finally, we can define

```
: target-compile ( -- )
   ['] target-compiler dup 'interpreter !   dup 'compiler !   IS parser
   Targeting on   Only Target also
;
```

that will activate the `target-compiler`, set `Targeting` to true and set the dictionary search order for target words.

2 Recognizers

Ok, now gforth's recognizers. The lower level words we need have already been explained above.

The debugger words are handled separately, so we define our first recognizer:

```
: rec-debugger ( addr u -- nt rectype | rectype-null )
   dbg? IF  debugger-wordlist find-name-in
            dup IF  rectype-name  EXIT THEN  dup
         THEN  2drop   rectype-null
;
```

Hm, we see that we have to do more than just define a recognizer, we also have to define recognizer types like `rectype-name` and `rectype-null`, which have been defined in gforth_079.

Now come the methods and the Forth dictionary. Remember, no matter whether we are in interpret or compile mode, we always execute the command of the target code compiler. So we have to define our first rectype, because the standard one, `rectype-name`, does not do what we need.

```
:noname   name>int execute-;s ;  \ interpret action
dup                              \ compile action, same as above
' lit,                           \ postpone action
rectype: rectype-target
```

Now we can define the recognizer for methods compilation,

```
: rec-methods ( addr u -- nt rectype | rectype-null )
   ClassContext @ 0= IF  2drop rectype-null  EXIT THEN
   2dup search-classes ?dup IF  nip nip rectype-target  EXIT THEN
   not-found
;
```

and the one for the Target's dictionary:

```
: rec-target ( addr u -- nt rectype | rectype-null )
   find-name ?dup IF  rectype-target  EXIT THEN  rectype-null
;
```

Ok that takes care of the methods and normal Forth words. Now we must turn to numbers. Again, we can not use the normal `rectype-num` and `rectype-dnum`, because we are compiling code for the target.

```
' noop
:noname ( n -- )  comp? IF  t_lit,  EXIT THEN  dbg? IF  >t  EXIT THEN  drop ;
dup
rectype: rectype-tnum

' noop
:noname ( d -- )  d>target swap
   comp? IF  t_lit, t_lit,  EXIT THEN
   dbg?  IF  >t     >t      EXIT THEN  2drop
;
dup
rectype: rectype-tdnum

: rec-tnum ( addr u -- n/d table | rectype-null )
   snumber? ?dup 0= IF  rectype-null  EXIT THEN
   0> IF  rectype-tdnum  ELSE  rectype-tnum  THEN
;
```

Now we have a single/double integer recognizer for the target and we are prepared to define the recognizer for target compilation consisting of four single recognizers:

```
$Variable target-recognizer
align here target-recognizer !
4 cells , ' rec-tnum A, ' rec-target A, ' rec-debugger A, ' rec-methods A,
```

Finally we can define:

```
: target-compile ( -- )
   target-recognizer TO forth-recognizer
   Targeting on   Only Target also
;
```

that will activate the `target-recognizer`, set `Targeting` to true and set the dictionary search order for target words.

Frankly, I find the patch version of the first chapter a lot easier to read.

Why is this so?

In the patched version, everything is close together in one definition: From string through wordlist lookup to the final execute. Therefore, you may modify everything when needed. We could even do without patching, if `[` and `]` were deferred. Its drawback: Everything is merged into a single definition with a more or less complex conditional structure.

The recognizer version gets rid of the complex conditional structure, but it has newly invented complexities: Separation of the search action from semantic interpretation (rectypes), and the final execute is completely hidden. Therefore, in order to understand the recognizer code, you must make yourself a mental image that merges the search action and its semantic interpretation, and you better understand the underlying recognizer machine that does the final execute for you. Let alone the definition of the final -recognizer, which is not a colon definition but a data structure, which has to be read backwards. Challenging conditions for reliable code that ought to be easy to understand.

3 Poor Man's Recognizer

As a compromise, here is a simplified version for a recognizer type construct, which does not need all the overhead of gforth's recognizer mechanism.

Foremost, we want to get rid of the convoluted control structure of the patch version. I.e. we want to be able to write down an easy to read specification for `target-compiler`, which behaves identical to the patched version of the first chapter as follows:

```
: target-compiler ( addr len -- )
   method-find debugger-find target-find target-number not-found
;
```

In the Recognizer chapter we finally defined `target-recognizer` as a reverse list of simple sub-recognizers, each of which only does a single string matching attempt.

Instead, the simplified version defines a colon definition, which basically does the same thing. It has the limitation that it can not change its behaviour dynamically. But who needs this flexibility? After all, the parser has a much lower change rate compared to the dictionary's search order.

Each sub-recognizer has identical stack behaviour:

It takes a counted string as input argument.

If there is no match, it leaves the counted string unchanged handing it over to the next sub-recognizer.

If there is a match, the input arguments are dropped, the return address into `target-compiler` is thrown away and therefore, we continue to execute the word that called `target-compiler`.

If none of our sub-recognizers did match, we bump into `not-found`.

These are the definitions of the sub-recognizers:

```
: method-find ( addr len -- addr len | rdrop )
   ClassContext @ 0= ?EXIT
   2dup search-classes ?dup
   IF  rdrop nip nip   name>int execute  EXIT THEN
   not-found
;
: debugger-find ( addr len -- addr len | rdrop )
   dbg? 0= ?EXIT
   2dup debugger-wordlist search-wordlist
   IF  rdrop nip nip   name>int execute  EXIT THEN
;
: target-find ( addr len -- addr len | rdrop )
   2dup find-name ?dup IF  rdrop nip nip   name>int execute  THEN
;
: target-number ( addr len -- addr len | rdrop )
   2dup 2>r   snumber? ?dup 0= IF  2r> EXIT THEN  2rdrop rdrop
   comp? IF  0> IF  d>target swap t_lit,  THEN  t_lit,  EXIT THEN
   dbg?  IF  0> IF  d>target swap >t      THEN  >t      EXIT THEN
   drop
;
```

For completeness, here is the host's interpreter/compiler:

```
: host-find ( addr len -- addr len | rdrop )
   2dup find-name ?dup 0= ?EXIT   rdrop nip nip
   comp? IF  name>comp  ELSE  name>int  THEN   execute
;
: host-number ( addr len -- addr len | rdrop )
   2dup 2>r snumber? ?dup 0= IF  2r> EXIT THEN   2rdrop rdrop
   comp? IF  0> IF  swap postpone Literal  THEN  postpone Literal  EXIT THEN
   drop
;
: host-compiler ( addr len -- )   host-find host-number not-found ;
```

4 Summary

I am sorry to say, but Matthias Trute's recognizer mechanism is over-engineered and therefore, it is difficult to understand, explain, and extend. An indication for this is the sheer number of articles, which try to explain it. In addition, it is more flexibel than actually needed.

The "Poor Man's Recognizer" had been a vague idea for several years. After writing the first two chapters of this paper, I finally implemented it. It exceeded my expectations as far as complexity is concerned and therefore, this is what I will add to my Forth toolbox.

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

REPLACE THE REGISTRY WITH WHAT?

a) Basic configuration data — *A BIT OF THIS*
Configuration file

b) System configuration data — ***LOTS*** *OF THIS*
PROBLEM

c) Per PC configuration data — *A BIT OF THIS*
Configuration file

d) Small amounts of persistent data — *A BIT OF THIS*
Configuration file

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

THE GOOD OLD REGISTRY

* One of the best bits of Windows

* Gradually made less and less useful over the years

* Essentially, a hierarchical database

* Amazingly, there is no really good alternative in Linux

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

Configuration files – easy
Lots of possibilities
Libconfig is our favourite

Typical FORTH interface to Libconfig

```
: DATABASECONF ( --- ) \ Database settings
  Z" Database" CONFKEY
\    <length>     <address>       <Name>                          <Default>          <action>
     HOST_NAME_MAX ZDBHOST         Z" IP_address_of_normal_database"  Z" 192.168.0.10"  CONF$
     HOST_NAME_MAX ZDBCENTREAL     Z" IP_address_of_central_database" Z" 192.168.0.100" CONF$
     MAX_DB_USER   DBUSER          Z" Database_user"               Z" Nick"           CONF$
     MAX_DB_PASS   DBPASS          Z" Database_password"           Z$SECRET           CONF$
     MAX_DB_NAME   DBNAME          Z" Database_name"               Z" Tracknet"       CONF$
                   ADDR DBPORT     Z" Database_port_number"        3306               CONFVAR
;
```

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

We used it for:

a) Basic configuration data — *A BIT OF THIS*
e.g. Where to find the database

b) System configuration data — ***LOTS*** *OF THIS*
e.g. How many Kgs do you put in a washing machine?

c) Per PC configuration data — *A BIT OF THIS*
e.g. On this PC, are you allowed access to this dialog?

d) Small amounts of persistent data — *A BIT OF THIS*
e.g. Go back to the same report that you selected last time

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

WHAT TO DO ABOUT SYSTEM CONFIGURATION DATA?

* Lots of it
* Needs to be easily editable
  (But only by the configuration engineer)
* Needs to be structured
* Needs to be accessible from all devices on the control network
* Needs to be very flexible, and easily extensible

THIS SUGGESTS
a) A database table
b) A tabbed dialog box
c) The dialog box to be dynamically generated

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

HOW TO EDIT THE DATA?

a) Basic configuration data — *A BIT OF THIS*
Need to set this manually, before the program will run

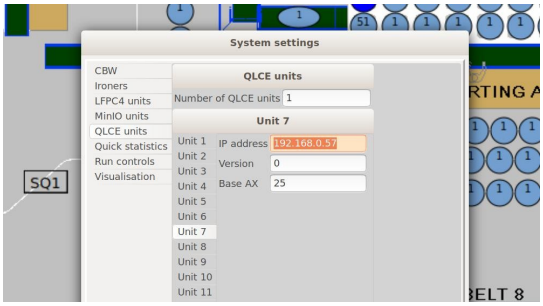
b) System configuration data — ***LOTS*** *OF THIS*
Dialog boxes ***ESSENTIAL*** *for managing the complexity*

c) Per PC configuration data — *A BIT OF THIS*
SIMPLE, so could be set manually

d) Small amounts of persistent data — *A BIT OF THIS*
Never needs setting

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

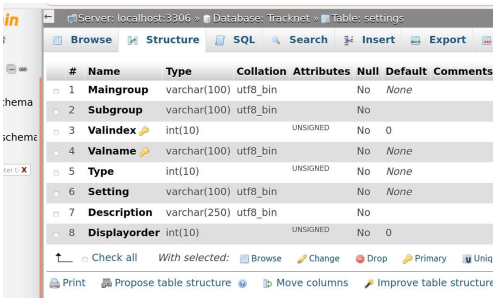


QLCE = Quad Loadcell to Ethernet

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

Database table columns



Euroforth 2020 “Rome”

A radical alternative to the Windows registry

NOW FOR THE RADICAL BIT

DURING COMPILATION, AT AN EARLY STAGE…

1. Read in the database config file
Now you know how to get to the DB

2. Read in the DB settings table

3. For every **Valname…**

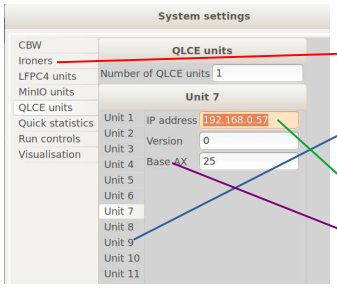
a) Does it exist as a Forth word?
b) If not, dynamically create a VALUE, VINDEX or STRINDEX,
according to **Valindex** and **Type**
c) Set the value from **Setting**

4. You are then free to use these new words in the rest of the compilation process

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

Tying the DB table to the dialog



Name	Type
Maingroup	varchar(100)
Subgroup	varchar(100)
Valindex	int(10)
Valname	varchar(100)
Type	int(10)
Setting	varchar(100)
Description	varchar(250)
Displayorder	int(10)

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

SUMMARY

New Forth words are created, not in code, but from entries in a database table.

Euroforth 2020 “Rome”

A radical alternative to the Windows registry

Specify Forth VALUE types in the DB

Name	Type
Maingroup	varchar(100)
Subgroup	varchar(100)
Valindex	int(10)
Valname	varchar(100)
Type	int(10)
Setting	varchar(100)
Description	varchar(250)
Displayorder	int(10)

Valname
Name of Forth value type word

Valindex
0= it’s a single Forth VALUE
0<> = it’s indexed value
(VINDEX, STRINDEX etc.)

Type
bool, int, string etc.

EuroForth 2020 "Rome"

Preparing for 64 bit

(Praeparatio ad LXIV frenos)

```
-1.0 0 UD.R
340282366920938463463374607431768211446 ok
```


EuroForth 2020 "Rome"

Preparing for 64 bit

Why 64 bit anyway?

Numerical accuracy? ✗

Addressing range? ✗

Interface with:
a) Operating system
b) Libraries ✓

EuroForth 2020 "Rome"

Preparing for 64 bit

Previous experience?

16 bit to 32 bit

FIG-like 16 bit

MPE Forth on WIN32S

1993

Can't remember a thing about it!

EuroForth 2020 "Rome"

Preparing for 64 bit

Critical differences

1

A CELL is no longer an int

Int is still 32 bits

Therefore, @ and ! don't work with ints any more

EuroForth 2020 "Rome"

Preparing for 64 bit

Critical differences

2

Careless Extern: declarations don't work any more

e.g. Enumerations are not ints

EuroForth 2020 "Rome"

Preparing for 64 bit

Critical differences

3

Anything ending with a _t

The good news is, the Linux 2038 problem goes away!

EuroForth 2020 "Rome"

Preparing for 64 bit

Solutions

1

A bit radical, this…..

Get rid of @ and !

a) Use VALUEs instead of VARIABLEs

b) Fetch / store ints in structures using L@ and L!

EuroForth 2020 "Rome"

Preparing for 64 bit

Solutions

1

Even more radical...

Get rid of fetch & store completely

Access structure elements like VALUEs

See my main paper

EuroForth 2020 "Rome"

Preparing for 64 bit

Solutions

2

a) Go through every Extern: and ensure prototypes match exactly

b) Go through every type definition and ensure size is correct

EuroForth 2020 "Rome"

Preparing for 64 bit

Solutions

3

Check every access to to things that end in _t

EuroForth 2020 "Rome"

Preparing for 64 bit

Did it work?

Bit early to tell!

cc64 - Small C on the C64

EuroForth 2020
Philip Zembrod - pzembrod@gmail.com

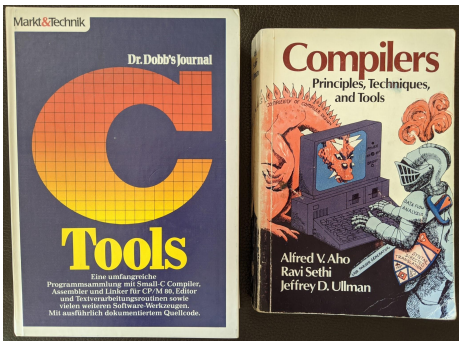
Why write a C compiler on the C64 in 1989?

I thought I could.

Long-standing interest in compilers

Inspirations: Dragon book +
Ron Cain's and James E. Hendrix'
8080 Small-C compiler

Existing C64 C & Pascal compilers
felt slow, opaque, impractical.



ASSI/M macros

"s65+": macros + string functions
+ conditional assembly

Functions, local and global vars,
recursion, arrays, pointers

Syntax still assembly-ish
Next: real compiler

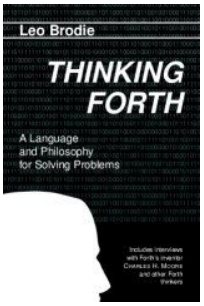
```
function findloc
char name^/start^
getargs name/start
 char cptr^
 move locptr,cptr
 decr cptr
 while "comp start,cptr",lt
  sub cptr^,cptr
  test stracmp(name/cptr/#maxnamlen)
  if eq
   sub #of_name,cptr
   return cptr
  endif
  sub #of_name+1,cptr
 endwhile
 return 0
endfunc
```

Why write it in Forth?

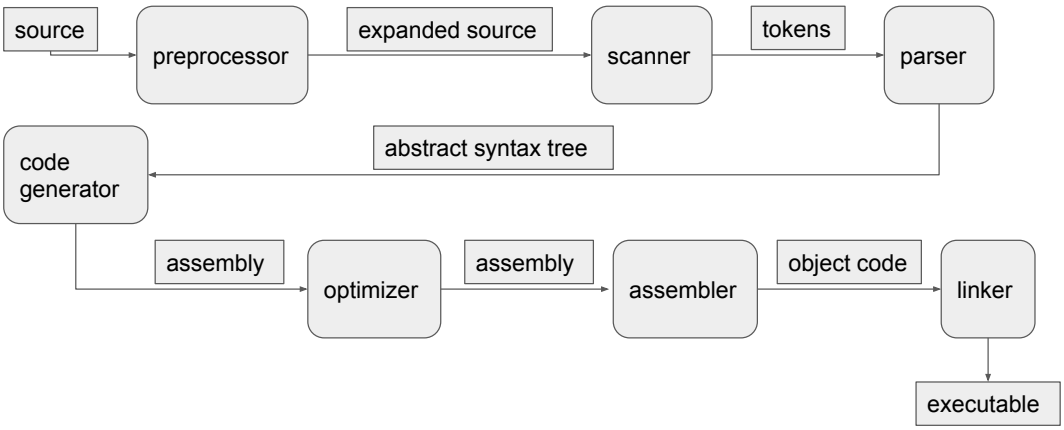
Why not in s65+? macro-expanding 100x: assembling slow

Encountered UltraForth

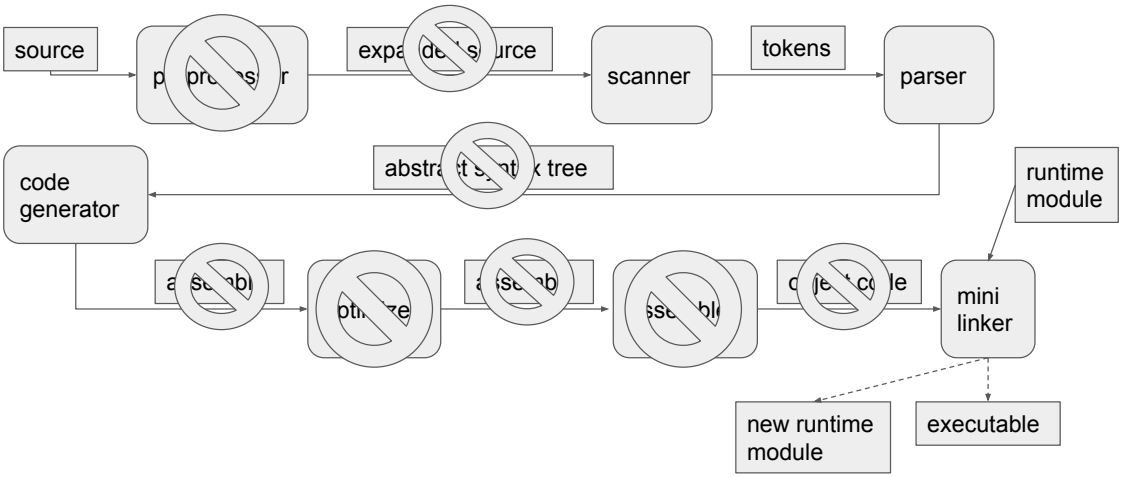
- compact code
- transparent, fast enough, practical
- fascinating
- … the known advantages of Forth ...



Typical classic C compiler



cc64 simplifications



cc64 parser

- Parsers
  - top-down vs bottom-up
  - hand-written vs grammar-generated
  - en.wikipedia.org/wiki/Comparison_of_parser_generators
  - Forth-generating parser generators are rare
- cc64: top-down recursive-descent parser
  - \+ easy to understand
  - – needs deep rstack
  - hand-crafted: more fun
- Limitations
  - K&R, int & char only, only 1 pointer level
  - not supported: unsigned, long, floats, void, struct, union, typedef, type casts, goto

Example: || and &&

```
: l-or ( -- obj )
     l-and
     BEGIN  <l-or> #oper# comes? WHILE
     do-l-or.1  l-and
     do-l-or.2  REPEAT ;

: l-and ( -- obj )
     bit-or
     BEGIN  <l-and> #oper# comes? WHILE
     do-l-and.1  bit-or
     do-l-and.2  REPEAT ;
```

Defining word

```
<or>   ' do-or
1   ' bit-xor   binary bit-or

<xor>  ' do-xor
1   ' bit-and   binary bit-xor

<and>  ' do-and
1   ' equal     binary bit-and

<==>   ' do-eq
<!=>   ' do-ne
2   ' comp      binary equal
```

```
<<>    ' do-lt
<<=>   ' do-le
<>>    ' do-gt
<>=>   ' do-ge
4   ' shift     binary comp

<<<>   ' do-shl
<>>>   ' do-shr
2   ' sum       binary shift

<+>    ' do-add
<->    ' do-sub
2   ' product   binary sum

<*>    ' do-mult
</>    ' do-div
<%>    ' do-mod
3   ' unary     binary product
```

Challenge: 6502

Only 3 registers: a, x, y, all 8 bit. 256 bytes hardware stack.
High-level languages be like :-(

Approach: virtual 16 bit stack machine with virtual a in 6502 a/x:

```
a&: .lda#     <& # lda  >& # ldx  ;a

a:  .pha      pha  txa  pha  ;a

a:  .sta-zp   $zp sta   $zp+1 stx ;a
```

Soft stack for local variables

Codegen do-xyz using code templates .xyz

```
   ' / ' .#div ' .div# ' .div
binop do-div

   ' mod ' .#mod ' .mod# ' .mod
binop do-mod

   ' * ' .mult# ' .mult# ' .mult
binop do-mult
```

```
\ divide a by zp, remainder in zp
a: (.divmod  $divmod jsr ;a

: .div#  .ldzp#          (.divmod ;
: .#div  .sta-zp .lda#   (.divmod ;
: .div   .sta-zp .pla    (.divmod ;

: .mod#  .div#   .lda-zp ;
: .#mod  .#div   .lda-zp ;
: .mod   .div    .lda-zp ;
```

Code templates

```
a&: .ldzp#  tay
  <& # lda  $zp   sta
  >& # lda  $zp+1 sta  tya  ;a

: (a: ( -- sys )
  create  here 0 c,  20
  assembler ;

: ;a ( sys -- )
  20 ?pairs
  current @ context !
  here over - 1- swap c! ;
```

```
: a, ( par I b -- step )
  dup $f and $7 =
    IF 2/ 2/ 2/ 2/ 2*
    atab + @ execute
    ELSE b, 2drop 1 THEN ;

: a&: (a: does> ( par pfa -- )
  count bounds DO
  dup I dup c@ a, +LOOP drop ;

: a: (a: does> ( pfa -- )
    count bounds
    DO 0 I dup c@ a, +LOOP ;
```

Other challenges

- Memory size
- Keep overview of source code
  - 3 disks 170 kB each
  - -> many printouts, 12 screens per page
- Moving screens around
- C64 charset: \^_{|}~ missing
- Text editor: missing
- Testing … 😓

… graduation … pause 1996-2019 ...

Restart with emulator

- From screen to stream sources
  - ufscr2file.c, ascii2petscii.c, petscii2ascii.c
  - Simple INCLUDE implementation
- VICE emulator with 4 disk drives
  - 1 Linux-dir-backed
  - 3 d64 disk images
- Automate build & tests
  - Script VICE with --keybuf param
  - Terminate VICE when file “notdone” deleted
  - Re-learn GnuMake
- Tests, tests, tests, some fixes, release ... github.com/pzembrod/cc64

VolksForth cross-pollination

- Ported to C64/C16 VolksForth:
  - VICE automation infra
  - INCLUDE
- Automated tests
- Automated target compile
- Integrated INCLUDE into core
- Binary flavours full and lite: with and without block words
- Fixed C16-32k
- Took lite binaries back to cc64
  - -> cc64 on c16-64k/Plus4!

Further ideas

- Port to Commander x16
- Port to Linux (host)
- cc64 unit/component tests
- ANSI function param syntax
- Profile memory and CPU. Why is so slow?
- Output assembler source
- Relocating symbolic linker
- C library

Thank you for listening!

Questions?

Forth and IDEs

M. Anton Ertl, TU Wien

Integrated Development Environment (IDE)

- Editor (aka IDE) is the hub
  Invokes compiler
  Processes error messages
  Runs program
  Debugging interface
  Crossreferencing
- Language-specific: Smalltalk-80, Turbo Pascal (1983), HolonForth (1989)
- No choice of editor
- Cross-language: Eclipse, IntelliJ IDEA, Visual Studio Code
- Gforth supports Emacs (and vi) as IDE

Language Server Protocol (LSP)

- Interface between language systems and IDEs
- Traditional development: $O(|\text{languages}| \times |\text{IDEs}|)$
- LSP development: $O(|\text{languages}| + |\text{IDEs}|)$
- Editor choice: any editor that supports LSP
- Future work: Add LSP support to Forth systems

Traditional Forth approach

- Forth command line is the hub

- Show source code of word
  or edit it with *your favourite* editor
  Load program
  Run program
  Debugging
  Crossreferencing
  Documentation
  Show generated code
  Show other system state

Demo

Implementation Cost

64-bit Gforth right after startup

	size (KB)
dictionary	548
native code	751
locate	32
where	619
backtrace	548

Conclusion

- Many languages: editor as hub
- Forth: command line as hub
  works with your favourite editor
- `locate`
  `where`
  backtrace
  `help`
  `see`
- also covers system code