**VFORTH Description**

VFORTH is a 32 bit Forth derivative. Its initial platform is a simple 32 bit virtual machine.

Like OCF (and like Chuck Moore’s 1970 FORTH version !), all words are executed. No exceptions. Some words, when executed, do compilation though.

Words have two forms, which are distinguished between using a bar prefix.

DUP Executes a DUP

|DUP Executes the code that compiles a DUP. Call to DUP or inline code.

RFORTH code is driven by either Color (like Color FORTH) or Syntax as follows. Color FORTHs colour scheme is kept, but this is not Color Forth. Code is compiled to ‘object code’.

Executable Words

Directly executed words are in underline and/or Yellow.

In the example above DUP or DUP will execute DUP. A numeric constant will be pushed on the stack.

Compiled Words

Words that are compiled into the dictionary are in normal font and/or Green.

In the example above DUP or DUP will execute |DUP rather than DUP. This will normally either compile a call to DUP, or generate the inline code for DUP, but it also allows for words that are executed at Compile time. If (say) |IF exists but IF does not, then IF cannot be run directly, only in compile mode.

A numeric constant will cause code that generates that literal to be pushed on the stack. The dictionary is checked first. This will be presented to the system as |42 – compiles the code to push 42 on the stack – using the same syntax.

Dictionary Entries

Dictionary Entries are in bold and/or red. So **DROP** or DROP both define a word DROP. This does not put anything in code merely puts a reference in the dictionary (or overrides a currently existing one). So code like 8\* 2\* 4\* 2\* 2\* ; works. It will actually – record a reference to 8\*, compile the code for 2\* whatever that is, record a reference to 4\* and compile the code for 2\* twice, followed by the code for return. Compilation will be done by |2\* and |; respectively.

Comments

Comments are in italics and/or white. So *this Is a comment* and so is this. Comments by convention are *italicised like this* in editors (e.g. Kompozer or Blue Griffon) if the display system supports it. Obviously it has to support either bold/italic/underline or colour.

CSS Styling

<style type="text/css">  
em {

color: white;

}  
body {

font-size:x-large;  
font-family:consolas,tahoma,sans-serif;  
color: green;  
background-color:black;  
}  
strong {  
color: red;  
}  
u {  
text-decoration:none;  
color:yellow;  
}  
</style>

Code can be written in HTML editors that support italic bold underlined rendering as WYSIWYG (it will still work with <em>code</em> tags but it won’t be very readable. I have had success with both Kompozer and Blue Griffon.

The CSS on the right, when inserted at the top of a HTML file, makes it into a “Color FORTH” editor in ASCII.

A script will be created which takes the HTML and converts it to the internal format.

Design

Initially there will be a machine with about 40 primitives , non packed, on a 32 bit machine architecture with 4 byte words, lower end format (e.g. LSB first), byte addressed (e.g. a 386 structure).

Bootstrapping will be done via a cross compiler written in Python which assumes all code is entirely compiled and will just be a collection of FORTH definitions that get the basic system going.

This will be absolutely minimalistic. It will really just be an assembler for the primitives, which can build composites and has very basic branching facilities (basically GOTO and IF GOTO ! with labels).

It will also include |xxxx definitions for the primitives.

Later on the ‘code generation’ part of the assembler can output (say) 386 equivalents of the primitives rather than the actual primitive.

It’s purpose is to be sufficient to allow the normal usage of FORTH e.g. execution of a string of definitions, and definition of new definitions.

Python Quasi-Forth Cross-Compiler

[[definition]] Compiles a definition in place.

// Comment (to end of line)

<Constant> Compile a literal

<Definition> Compile a call to a definition or execute a primitive.

BR / BZ <label> Compile a branch, conditional or unconditional to

<label>: Define label (all labels are local to definition)

ALLOC n Allocate n words of memory

WORDSIZE Push word size on stack (4)

Back end options

getHere()

Get current compilation word address.

getWordSize()

Get the current word size.

compileHeader(name)

Compile definition header.

compileAllocate(count)

Compile the given number of words

compileLiteral(literal)

Compile code to a literal

compileCall(address)

Compile call to an address (e.g. another word)

compilePrimitive(primitive)

Compile the given primitive.

compileBranch(isConditional)

Compile a conditional (tos = 0) or unconditional branch.

setBranchTarget(branchAddress,targetAddress)

Adjust a branch target using the address of the compileBranch to targetAddress

FORTH primitives

@ ! c@ c! +! + - \* / and or xor

not 0= 0> 0< 0- 1+ 1- 2\* 2/ dup drop

swap rot over pick ; r> >r ,

*Note that 2\* and 2/ are logical shift lefts and rights, not arithmetic or division.*

Unusual primitives (e.g. not normal FORTH ones)

$br <addr> Branch to address in next word

$0br <addr> Branch to address in next word if tos is zero, tos dropped.

$lit <constant> Push constant in next word on stack *for architectures that use it.*

The VM doesn’t but it shouldn’t be precluded.

Forth structure

The dictionary is inlined as in most FORTHs. A dictionary record is defined as follows.

If the last word compiled is not the primitive return (i.e. it is a fall through) then $BR <code> is compiled. Example is “increment” (letters are 7 bit ASCII in 8 byte groups). Bit 7 of last word clear.

+0 address of previous word in chain, or zero if not applicable.

+4 1 <i> <n> <c> <r>

+8 1 <e> <m> <e> <n>

+12 0 0x00 0x00 0x00 <t>

+16 first word, and target address of $BR if relevant. Must be word aligned

FORTH VM

00000000-000000FF Execute given primitive

00000100-7FFFFFFF Call routine at Base+(100-7FFFFFFF)

80000000-FFFFFFFF Push sign extended constant on stack

First 256 bytes/64 words reserved.