**Instruction set**

There are four commands in the instruction set. They are identified by the two least significant bits of the 32 bit word.

|  |  |  |
| --- | --- | --- |
| Type | LSB | Description |
| Routine Call | 00 | This is a subroutine call, given an absolute address in the memory space. Instructions are word aligned, so this is designed so that the whole 32 bit value is the physical address of the word, *not* the value shifted right twice. |
| Core Commands | 01 | This indicates a core command. The command number is starts from the 2nd bit of the word. This is normally shifted right twice to increase the probability that a switch statement or similar will implement as a table. |
| Definition/String | 10 | The upper 30 bits contain 4 x 7 bit ASCII values. This represents four characters. The upper 2 bits of the word have the following meanings – either to place the text of a definition in memory, or to create code to generate a string.   * 11 Start of definition name * 10 Continuation of definition name * 00 Copy characters to string buffer * 01 Append characters to string buffer. |
| Literal | 11 | This pushes the 30 bit literal from the upper 30 bits of the stack on the data stack. This literal is sign extended from the 30th bit. e.g. $FFFFFFFF is $FFFFFFFF not $3FFFFFFF |

**Definition**

A word definition is stored as follows

|  |  |  |
| --- | --- | --- |
| Offset from address | Contents | Notes |
| -16 | Start of name | A Definition string with 11 in the upper 2 bits |
| -12 | Continuation of name | A Definition string with 10 in the upper 2 bits |
| -8 | End of name | A Definition string with 10 in the upper 2 bits |
| -4 | Link | Addr of previous word in the dictionary, 0 if none. |
| +0 | Code | First instruction of definition |
| +4 | Code | Second instruction of definition. |

**Code Header**

At the start of the code image there is a word array of data, this is set up as follows. These values can be changed if (for example) development is done and the state is to be saved.

|  |  |
| --- | --- |
| Offset | Contents |
| 0 | Length of code / Next free word. The address of the word after the end of the code. Used for C, , ALLOT and so on. |
| 4 | Head of the word list currently in use, used when defining new words. |
| 8 | Initial value of the Data Stack Pointer (set by the run time). This and the RSP in the runtime are set automatically, it is for ABORT type instructions, so you can do (e.g. 8 @ dsp! 12 @ rsp! 16 @ >r ; resets both stacks and runs from the first position) |
| 12 | Initial value of the Return Stack Pointer (set by the run time) |
| 16 | Address of the first instruction of \_\_main, the word used to start the code. Can be changed if required. |
| 20 | Program first word. |

**Core Commands**

These core commands are implemented atomically. Each has a constant associated with it, this is generated automatically from the list in the python based compiler in various required formats.

|  |  |  |
| --- | --- | --- |
| Word | Stack | Notes |
| @ | (addr – data) | Read one 32 bit word. Must be on a 4 byte boundary.  *Addresses are all byte addresses.*  *Least Significant Byte first in word.* |
| ! | (data addr -) | Write one 32 bit word. Must be on a 4 byte boundary |
| c@ | (addr – data) | Read one 8 bit byte |
| c! | (data addr -) | Write one 8 bit byte |
| +! | (value addr -) | Add value into memory. Must be on a 4 byte boundary |
| + | (n1 n2 – n3) | Add two 32 bit values |
| - | (n1 n2 – n3) | Subtract two 32 bit values (result is n1 – n2) |
| \* | (n1 n2 – n3) | Multiply two 32 bit values |
| / | (n1 n2 – n3) | Divide two 32 bit values (result is n1 / n2). If n2 = 0 result is 0 |
| and | (n1 n2 – n3) | Bitwise and two 32 bit values |
| or | (n1 n2 – n3) | Bitwise or two 32 bit values |
| xor | (n1 n2 – n3) | Bitwise exclusive or two 32 bit values |
| not | (n1 – n2) | One’s complement top of stack. |
| 0= | (n1 – n2) | Push Truth(n1 = 0) on stack (False 0, True $FFFFFFFF) |
| 0> | (n1 – n2) | Push Truth (n1 > 0) on stack |
| 0< | (n1 – n2) | Push Truth (n1 < 0) on stack |
| 0- | (n1 – n2) | Two’s complement negate top of stack |
| 1+ | (n1 – n2) | Increment top of stack |
| 1- | (n1 – n2) | Decrement top of stack |
| 2\* | (n1 – n2) | Shift top of stack left |
| 2/ | (n1 – n2) | Shift top of stack right (most significant bit 0, e.g. not arithmetic) |
| dup | (n1 – n1 n1) | Duplicate top of stack value |
| drop | (n1 - ) | Drop value on top of stack |
| swap | (n1 n2 – n2 n1) | Swap values on top of stack |
| rot | (n1 n2 n3 – n3 n1 n2) | Rotate top three values |
| over | (n1 n2 – n1 n2 n1) | Duplicate 2nd value down on stack |
| ; | ( - ) | Return from subroutine |
| r> | ( - n1) | Pop top of return stack to data stack |
| >r | (n1 - ) | Pop top of data stack to return stack |
| $$branch | (addr - ) | Branch to given address (only used for SELF word, *nothing* else). |
| if | (n1 - ) | if n1 is not zero advance PC to after next *then* or *;* IF/THEN and FOR/NEXT should not be nested in a single word. |
| then | ( - ) | Marker for if (note, ; is also a terminating marker) but has no other effect. |
| for | (n1 - ) | Push PC then n1-1 to return stack. (loops from n-1 downto 0) |
| next | ( - ) | (Works with for) Decrement top of return stack.   * If < 0 then drop counter and loop address * If >= 0 then set PC to second value on stack. |
| dsp! | (n1 - ) | Set data stack address.  *All stacks are pre-decrement push, post-increment pull.* |
| dsp@ | ( - n1) | Get data stack address |
| rsp! | (n1 - ) | Set return stack address |
| rsp@ | ( - n1) | Get return stack address |
| $$hwio | *various* | Hardware interface functionality. TOS is always the command, but the in and out values depend on the operation. |

**Compile/Execute**

All words are executed. Words are of two types, normal words and compiling words. The latter are equivalent to immediate words, in that they generate code.

The compiler given a word “fred” will look for fred in the dictionary and either execute it or compile it according to the colour of the word, as normal. However when compiling, it will look for “fred|” in the dictionary instead, and if this is found it will execute it, if it is not found, it will compile the word as a call, if it exists, and error otherwise.

This allows invisible optimisation for primitives and similar - “fred|” is “do the compilation for fred”. It also allows immediate words like variable and allot.

Primitive words like dup to exist in two forms, one executable and one precompiling e.g.

* DUP is a word which executes the primitive ‘6’ (assuming that is the opcode for DUP)
* DUP| is a word which executes 6 , e.g. it compiles the code for primitive ‘6’.

There are special cases of this, primarily those which use the return stack R> >R because their execution versions have to allow for the return address (e.g. R> would be (R>) (R>) (DROP) (>R) to preserve the return address to the caller – it throws the second value on the stack. (all values are opcodes). The compiled version would throw the first address on the stack, so can just compile the opcode for R>

The words “; if for next self” should only ever exist in their compiled forms (e.g. if|) as when executed from the command line the words are executed sequentially.

**Boot Compiler**

There are syntactic differences between the normal compiler and the boot compiler, because the boot compiler does not have colour.

|  |  |  |
| --- | --- | --- |
| Word | Example | Notes |
| :<name> | :count | Inserts a definition (see earlier definition) updating the “head” value in the program header |
| self | self | Compiles a branch back to the start of the current function, it is equivalent to <word> ; for recursive loops. |
| allot:<size> | allot:42 | Allocate 42 words (e.g. 168 bytes). Increments the head pointer, does nothing else. |
| $<hexadecimal> | $A74 | Generate code to push literal A74 base 16 on the stack. |
| “<string>” | “Hello world” | Generate string operations to put string “Hello world” in the buffer (clearing it first) |
| %<colour><string> | %4fred | Compile a colorForth string “ fred” in colour 4 (BBC Micro colours). Note the preceding space. (Used for testing) |

Additional to these are primitives which are compiled inline, word calls, and decimal constants which do not take notice of base settings. Other syntaxes (e.g. variable) are also different.