The Beetle Forth Virtual Processor

Reuben Thomas

29th October 1993; revised 4th May 1995

Abstract

The design of the Beetle Forth virtual processor is described. Beetle's purpose is to provide an easily portable environment for ANS Forth compilers: to move a compiler from one system to another only Beetle and the I/O libraries need be rewritten. Like most interpreters, Beetle gains portability and compactness at the expense of speed, but it retains flexibility by providing instructions to call machine code and access the operating system.

Typographical notes

bForth instructions and Beetle's registers are shown in `Typewriter` font; interface calls are shown in **Bold** type, and followed by empty parentheses. Quoted pronunciations of instructions and registers are given with components separated by dashes; single letters should be pronounced as the name of the letter.

Addresses are given in bytes and refer to Beetle's address space except where stated. Addresses are written in hexadecimal; all hex numbers are followed by "h".

1 Introduction

Beetle is a simple virtual processor designed to enable the easy implementation of ANS Forth compilers, such as pForth [5], on different systems. It has twelve registers, two stacks, and an instruction set, called bForth, of ninety-two instructions. The instruction set is based on the Core Word Set of ANS Forth [1]. This paper gives a full description of Beetle, but certain implementation-dependent features, such as the size of the stacks, are purposely left unspecified, and the exact method of implementation is left to the implementor in many particulars.

Beetle is self-contained, and performs I/O via the `LIB` instruction, which provides access to a standard library which mimics ANS Forth I/O words. The operating system and machine code routines on the host computer may be accessed using the `OS` and `LINK` instructions. Beetle supports the saving and loading of simple object modules.

Beetle may exist either as a stand-alone system, or embedded in other programs. A small interface is provided for other programs wishing to control Beetle.

Since Beetle is heavily oriented towards supporting Forth compilers, it is useful to understand how Forth compilers operate in order to understand Beetle and to use it properly. An excellent introduction to Forth and Forth compilers is [3]. An overview of the language and its compilers is also provided in [1]. For an implementation of Beetle, see [6].

2 Architecture

Beetle's address unit is the byte, which is eight bits wide. Characters are one byte wide, and cells are four bytes wide. The cell is the size of the numbers and addresses on which Beetle operates, and of the items placed on the stacks. The cell size is fixed to ensure compatibility of object code between implementations on different machines; the size of the address unit, character and cell has been chosen with a view to making efficient implementation of Beetle possible on the vast majority of current machine architectures.

Cells may have the bytes stored in big-endian or little-endian order. The address of a cell is that of the byte in it with the lowest address.

2.1 Registers

The registers, each with its function and pronunciation, are set out in table 1.

Register	Pronunciation	Function
EP	"e-p"	The Execution Pointer. Points to the next cell from which an instruction word may be loaded.
I	"i"	The Instruction. Holds the opcode of an instruction to be executed.
A	"a"	The instruction Accumulator. Holds the opcodes of instructions to be executed, and immediate operands.
MO	"m-zero"	The address of Beetle's address space on the host system, which must be aligned on a four-byte boundary.
MEMORY	"memory"	The size in bytes of Beetle's address space, which must be a multiple of four.
SP	"s-p"	The data Stack Pointer.
RP	"r-p"	The Return stack Pointer.
'THROW	"tick-throw"	The address placed in EP by a THROW instruction.
ENDISM	"endism"	The endianness of Beetle: 0 = Little-endian, 1 = Big-endian.
CHECKED	"checked"	0 = address checking off, 1 = address checking on.
'BAD	"tick-bad"	The contents of EP when the last exception was raised.
-ADDRESS	"not-address"	The last address which caused an address exception.

Table 1: Beetle's registers

EP, A, MEMORY, SP, RP, 'THROW, 'BAD and -ADDRESS are cell-wide quantities; I, ENDISM and CHECKED are one byte wide, and MO's size depends on the implementation; it would normally have the same width as addresses on the host computer. The values of MEMORY, 'BAD and -ADDRESS are available in Beetle's address space; 'THROW must be physically held there so that it can be changed as well as read by programs. Their addresses relative to MO are shown in table 2.

Register	Address
'THROW	0h
MEMORY	4h
'BAD	8h
-ADDRESS	Ch

Table 2: Registers which appear in Beetle's address space

To ease efficient implementation, Beetle's stack pointers may only be accessed by bForth instructions (see section 3.7).

2.2 Memory

Beetle's memory is a contiguous sequence of bytes numbered from 0 to MEMORY − 1.

2.3 Stacks

The data and return stacks are cell-aligned LIFO stacks of cells. The stack pointers point to the top stack item on each stack. To **push** an item on to a stack means to store the item in the cell beyond the stack pointer and then adjust the pointer to point to it; to **pop** an item means to make the pointer point to the second item on the stack. The stacks grow downwards in memory as new items are added. Instructions that change the number of items on a stack implicitly pop their arguments and push their results.

The data stack is used for passing values to instructions and routines and the return stack for holding subroutine return addresses and the index and limit of the Forth DO...LOOP construct. The return stack may be used for other operations subject to the restrictions placed on it by its normal usage: it must be returned before an EXIT instruction to the state it was in directly after the corresponding CALL, and before a (LOOP), (+LOOP), or UNLOOP to the state it was in before the corresponding (DO).

In what follows, for "the stack" read "the data stack"; the return stack is always mentioned explicitly.

2.4 Operation

Before Beetle is started, MO, MEMORY and ENDISM should be set to implementation-dependent values; 'THROW should be set to point to the exception handler, and EP to the bForth code that is to be executed. CHECKED should be set to 0 or 1 as desired. The other registers should be initialised as shown in table 3, except for I and A, which need not be initialised.

Register	Initial value
SP	MEMORY − 100h
RP	MEMORY
'BAD	FFFFFFFFh
-ADDRESS	FFFFFFFFh

Table 3: Registers with prescribed initial values

MEMORY should be copied to 4h; its value and those of ENDISM and CHECKED must not change while Beetle is executing. Next, the action of NEXT should be performed (see section 3.11): A is loaded from the cell to which EP points, and four is added to EP.

Beetle is started by a call to the interface calls **run()** or **single_step()** (see section 4.3). In the former case, the execution cycle is entered:

```
begin
    copy the least-significant byte of A to I
    shift A arithmetically 8 bits to the right
    execute the instruction in I
repeat
```

In the latter case, the contents of the execution loop is executed once, and control returns to the calling program.

The execution loop need not be implemented as a single loop; it is designed to be short enough that the contents of the loop can be appended to the code implementing each instruction.

Note that the calls **run()** and **single_step()** do not perform the initialisation specified above; that must be performed before calling them.

2.5 Termination

When Beetle encounters a `HALT` instruction (see section 3.10), it returns the top data stack item as the reason code, unless `SP` does not point to a valid cell, in which case reason code -258 is returned (see section 2.6). After a call to **single_step()** which terminates without an exception being raised, reason code 0 is returned.

Reason codes which are also valid exception codes (either reserved (see section 2.6) or user exception codes) should not normally be used. This allows exception codes to be passed back by an exception handler to the calling program, so that the calling program can handle certain exceptions without confusing exception codes and reason codes.

2.6 Exceptions

When a `THROW` instruction (see section 3.10) is executed, an **exception** is said to have been **raised**. Some exceptions are raised by other instructions, for example by `/` when division by zero is attempted; these also execute a `THROW`. The exception code is the number on top of the stack at the time the exception is raised.

Exception codes are signed numbers. -1 to -255 are reserved for ANS Forth exception codes, and -256 to -511 for Beetle's own exception codes; the meanings of those that may be raised by Beetle are shown in table 4. ANS Forth compilers may raise other exceptions in the range -1 to -255 and additionally reserve exceptions -512 to -4095 for their own exceptions (see [1, section 9.3.1]).

Code	Meaning
-9	Invalid address (see below).
-10	Division by zero attempted (see section 3.4).
-23	Address alignment exception (see below).
-256	Illegal opcode (see section 3.14).
-257	Library routine not implemented (see section 3.12).

Table 4: Exceptions raised by Beetle

Exception -9 is raised whenever an attempt is made to access an invalid address (not between zero and `MEMORY − 1` inclusive), either by an instruction, or during an instruction fetch (because `EP` contains an invalid address). Exception -23 is raised when a bForth instruction expecting an address of type `a-addr` (cell-aligned), is given a non-aligned address. When Beetle raises an address exception (-9 or -23), the offending address is placed in `-ADDRESS`.

The initial values of 'BAD and -ADDRESS are unlikely to be generated by an exception, so it may be assumed that if the initial values still hold no exception has yet occurred.

Address and alignment exceptions are only raised if CHECKED is 1. When CHECKED is 0, a faster implementation of Beetle may be used—this is especially useful for stand-alone Beetles.

If SP is unaligned when an exception is raised, or putting the code on the stack would cause SP to be out of range, the effect of a HALT with code -258 is performed (although the actual mechanics are not, as that too would involve putting a number on the stack). Similarly, if 'THROW contains an invalid address, the effect of HALT with code -259 is performed.

3 Instruction set

The bForth instruction set is listed in sections 3.2 to 3.13, with the instructions grouped according to function. The instructions are given in the following format:

NAME "pronunciation" 00h (*before* -- *after*)
R: (*before* -- *after*)

Description.

The first line consists of the name of the instruction followed by the pronunciation in quotes, and the instruction's opcode. On the right are the stack comment or comments. Underneath is the description. The two stack comments show the effect of the instruction on the data and return (R) stacks.

Stack comments are written

(*before* -- *after*)

where *before* and *after* are stack pictures showing the items on top of a stack before and after the instruction is executed (the change is called the **stack effect**). An instruction only affects the items shown in its stack comments. The brackets and dashes serve merely to delimit the stack comment and to separate *before* from *after*. **Stack pictures** are a representation of the top-most items on the stack, and are written

$i_1\ i_2 \ldots i_{n-1}\ i_n$

where the i_k are stack items, each of which occupies a whole number of cells, with i_n being on top of the stack. The symbols denoting different types of stack item are shown in table 5.

Symbol	Data type
flag	flag
true	true flag
false	false flag
char	character
n	signed number
u	unsigned number
n|*u*	number (signed or unsigned)
x	unspecified cell
xt	execution token
a-addr	cell-aligned address
c-addr	character-aligned address

Table 5: Types used in stack comments

Types are only used to indicate how instructions treat their arguments and results; Beetle does not distinguish between stack items of different types. In stack pictures the most general argument types with which each instruction can be supplied are given; subtypes may be substituted. Using the phrase "$i \Rightarrow j$" to denote "i is a subtype of j", table 6 shows the subtype relationships. The subtype relation is transitive.

u ⇒ *x*
n ⇒ *x*
char ⇒ *u*
a-addr ⇒ *c-addr* ⇒ *u*
flag ⇒ *x*
xt ⇒ *x*

Table 6: The subtype relation

Numbers are represented in twos complement form. `a-addr` consists of all unsigned numbers less than `MEMORY`. Numeric constants can be included in stack pictures, and are of type *n*|*u*.

Each type may be suffixed by a number in stack pictures; if the same combination of type and suffix appears more than once in a stack comment, it refers to identical stack items. Alternative *after* pictures are separated by "|", and the circumstances under which each occurs are detailed in the instruction description.

The symbols *i*x*, *j*x* and *k*x* are used to denote different collections of zero or more cells of any data type. Ellipsis is used for indeterminate numbers of specified types of cell.

If an instruction does not modify the return stack, the corresponding stack picture is omitted. Some instructions have two forms, the latter ending in "I". This denotes Immediate addressing: the instruction's argument is included in the instruction cell (see section 3.1), rather than being placed separately in the next available cell.

3.1 Programming conventions

Since branch destinations must be cell-aligned, some instruction sequences may contain gaps. These must be padded with `NEXT` (opcode 00h).

Literals and branch addresses should be placed in memory as follows. If a literal (see section 3.9) or branch address (see section 3.8) will fit in the rest of the cell directly after its instruction (see below), it should be placed there, and the immediate form of the instruction used. Otherwise it should be placed in the cell after the instruction. Further instructions may still be stored in the current cell. If more than one literal or branch instruction is encoded in one instruction cell, the literal values follow each other in successive cells.

Given an instruction cell with n bytes free, a literal will fit into it if it can be represented as an n-byte twos complement number. Immediate mode branch destinations are given as the relative cell count from the value `EP` will have when the instruction is executed (rather than the address of the instruction cell containing the instruction) to the address of the destination instruction cell (not as absolute addresses). The literal or branch is stored with the bytes in the same order as for a four-byte number, at the most significant end of the instruction cell.

3.2 Stack manipulation

These instructions manage the data stack and move values between stacks.

`DUP` "dupe" 01h (*x* -- *x x*)

Duplicate *x*.

`DROP` 02h (*x* --)

Remove *x* from the stack.

`SWAP` 03h (x_1 x_2 -- x_2 x_1)

Exchange the top two stack items.

`OVER` 04h (x_1 x_2 -- x_1 x_2 x_1)

Place a copy of x_1 on top of the stack.

`ROT` "rote" 05h (x_1 x_2 x_3 -- x_2 x_3 x_1)

Rotate the top three stack entries.

`-ROT` "not-rote" 06h (x_1 x_2 x_3 -- x_3 x_1 x_2)

Perform the action of `ROT` twice.

`TUCK` 07h (x_1 x_2 -- x_2 x_1 x_2)

Perform the action of `SWAP` followed by `OVER`.

`NIP` 08h (x_1 x_2 -- x_2)

Perform the action of `SWAP` followed by `DROP`.

`PICK` 09h ($x_u \ldots x_1$ x_0 *u* -- $x_u \ldots x_1$ x_0 x_u)

Remove *u*. Copy x_u to the top of the stack. If $u = 0$, `PICK` is equivalent to `DUP`. If there are fewer than $u + 2$ items on the stack before `PICK` is executed, the memory cell which would have been x_u were there $u + 2$ items is copied to the top of the stack.

`ROLL` 0Ah (x_u $x_u - 1 \ldots x_0$ *u* -- $x_u - 1 \ldots x_0$ x_u)

Remove *u*. Rotate $u + 1$ items on the top of the stack. If $u = 0$ `ROLL` does nothing, and if $u = 1$ `ROLL` is equivalent to `SWAP`. If there are fewer than $u + 2$ items on the stack before `ROLL` is executed, the memory cells which would have been on the stack were there $u + 2$ items are rotated.

`?DUP` "question-dupe" 0Bh (*x* -- 0 | *x x*)

Duplicate *x* if it is non-zero.

`>R` "to-r" 0Ch (*x* --)
R: (-- *x*)

Move *x* to the return stack.

`R>` "r-from" 0Dh (-- *x*)
R: (*x* --)

Move *x* from the return stack to the data stack.

`R@` "r-fetch" 0Eh (-- *x*)
R: (*x* -- *x*)

Copy *x* from the return stack to the data stack.

3.3 Comparison

These words compare two numbers (or, for equality tests, any two cells) on the stack, returning a flag, true with all bits set if the test succeeds and false otherwise.

`<` "less-than" 0Fh (n_1 n_2 -- *flag*)

flag is true if and only if n_1 is less than n_2.

`>` "greater-than" 10h (n_1 n_2 -- *flag*)
flag is true if and only if n_1 is greater than n_2.

`=` "equals" 11h (x_1 x_2 -- *flag*)
flag is true if and only if x_1 is bit-for-bit the same as x_2.

`<>` "not-equals" 12h (x_1 x_2 -- *flag*)
flag is true if and only if x_1 is not bit-for-bit the same as x_2.

`0<` "zero-less" 13h (*n* -- *flag*)
flag is true if and only if n is less than zero.

`0>` "zero-greater" 14h (*n* -- *flag*)
flag is true if and only if *n* is greater than zero.

`0=` "zero-equals" 15h (*x* -- *flag*)
flag is true if and only if *x* is equal to zero.

`0<>` "zero-not-equals" 16h (*x* -- *flag*)
flag is true if and only if *x* is not equal to zero.

`U<` "u-less-than" 17h (u_1 u_2 -- *flag*)
flag is true if and only if u_1 is less than u_2.

`U>` "u-greater-than" 18h (u_1 u_2 -- *flag*)
flag is true if and only if u_1 is greater than u_2.

3.4 Arithmetic

These instructions consist of monadic and dyadic operators, and numeric constants. All calculations are made without bounds or overflow checking, except as detailed for certain instructions.

Constants:

`0` "zero" 19h (-- 0)
Leave zero on the stack.

`1` "one" 1Ah (-- 1)
Leave one on the stack.

`-1` "minus-one" 1Bh (-- −1)
Leave minus one on the stack.

`CELL` 1Ch (-- 4)
Leave four on the stack.

`-CELL` "minus-cell" 1Dh (-- −4)
Leave minus four on the stack.

Addition and subtraction:

`+` "plus" 1Eh (n_1|u_1 n_2|u_2 -- n_3|u_3)
Add n_2|u_2 to n_1|u_1, giving the sum n_3|u_3.

`-` "minus" 1Fh (n_1|u_1 n_2|u_2 -- n_3|u_3)
Subtract n_2|u_2 from n_1|u_1, giving the difference n_3|u_3.

`>-<` "reverse-minus" 20h (n_1|u_1 n_2|u_2 -- n_3|u_3)
Perform the action of `SWAP` (see section 3.2) followed by `-`.

`1+` "one-plus" 21h ($n_1|u_1$ -- $n_2|u_2$)
Add one to $n_1|u_1$, giving the sum $n_2|u_2$.

`1-` "one-minus" 22h ($n_1|u_1$ -- $n_2|u_2$)
Subtract one from $n_1|u_1$, giving the difference $n_2|u_2$.

`CELL+` "cell-plus" 23h ($n_1|u_1$ -- $n_2|u_2$)
Add four to $n_1|u_1$, giving the sum $n_2|u_2$.

`CELL-` "cell-minus" 24h ($n_1|u_1$ -- $n_2|u_2$)
Subtract four from $n_1|u_1$, giving the difference $n_2|u_2$.

Multiplication and division (note that all division instructions raise exception -10 if division by zero is attempted, and round the quotient towards minus infinity, except for `S/REM`, which rounds the quotient towards zero):

`*` "star" 25h ($n_1|u_1$ $n_2|u_2$ -- $n_3|u_3$)
Multiply $n_1|u_1$ by $n_2|u_2$ giving the product $n_3|u_3$.

`/` "slash" 26h (n_1 n_2 -- n_3)
Divide n_1 by n_2, giving the single-cell quotient n_3.

`MOD` 27h (n_1 n_2 -- n_3)
Divide n_1 by n_2, giving the single-cell remainder n_3.

`/MOD` "slash-mod" 28h (n_1 n_2 -- n_3 n_4)
Divide n_1 by n_2, giving the single-cell remainder n_3 and the single-cell quotient n_4.

`U/MOD` "u-slash-mod" 29h (u_1 u_2 -- u_3 u_4)
Divide u_1 by u_2, giving the single-cell remainder u_3 and the single-cell quotient u_4.

`S/REM` "s-slash-rem" 2Ah (n_1 n_2 -- n_3 n_4)
Divide n_1 by n_2 using symmetric division, giving the single-cell remainder n_3 and the single-cell quotient n_4.

`2/` "two-slash" 2Bh (x_1 -- x_2)
x_2 is the result of shifting x_1 one bit toward the least-significant bit, leaving the most-significant bit unchanged.

`CELLS` 2Ch (n_1 -- n_2)
n_2 is the size in bytes of n_1 cells.

Sign functions:

`ABS` "abs" 2Dh (n -- u)
u is the absolute value of n.

`NEGATE` 2Eh (n_1 -- n_2)
Negate n_1, giving its arithmetic inverse n_2.

Maxima and minima:

`MAX` 2Fh (n_1 n_2 -- n_3)
n_3 is the greater of n_1 and n_2.

`MIN` 30h (n_1 n_2 -- n_3)
n_3 is the lesser of n_1 and n_2.

3.5 Logic and shifts

These instructions consist of bitwise logical operators and bitwise shifts. The result of performing the specified operation on the argument or arguments is left on the stack.

Logic functions:

`INVERT` 31h `( x1 -- x2 )`
Invert all bits of x_1, giving its logical inverse x_2.

`AND` 32h `( x1 x2 -- x3 )`
x_3 is the bit-by-bit logical "and" of x_1 with x_2.

`OR` 33h `( x1 x2 -- x3 )`
x_3 is the bit-by-bit inclusive-or of x_1 with x_2.

`XOR` "x-or" 34h `( x1 x2 -- x3 )`
x_3 is the bit-by-bit exclusive-or of x_1 with x_2.

Shifts:

`LSHIFT` "l-shift" 35h `( x1 u -- x2 )`
Perform a logical left shift of `u` bit-places on x_1, giving x_2. Put zero into the least significant bits vacated by the shift. If `u` is greater than or equal to 32, x_2 is zero.

`RSHIFT` "r-shift" 36h `( x1 u -- x2 )`
Perform a logical right shift of `u` bit-places on x_1, giving x_2. Put zero into the most significant bits vacated by the shift. If `u` is greater than or equal to 32, x_2 is zero.

`1LSHIFT` "one-l-shift" 37h `( x1 -- x2 )`
Perform a logical left shift of one bit-place on x_1, giving x_2. Put zero into the least significant bit vacated by the shift.

`1RSHIFT` "one-r-shift" 38h `( x1 -- x2 )`
Perform a logical right shift of one bit-place on x_1, giving x_2. Put zero into the most significant bit vacated by the shift.

3.6 Memory

These instructions fetch and store cells and bytes to and from memory; there is also an instruction to add a number to another stored in memory.

`@` "fetch" 39h `( a-addr -- x )`
`x` is the value stored at `a-addr`.

`!` "store" 3Ah `( x a-addr -- )`
Store `x` at `a-addr`.

`C@` "c-fetch" 3Bh `( c-addr -- char )`
If `ENDISM` is 1, exclusive-or `c-addr` with 3. Fetch the character stored at `c-addr`. The unused high-order bits are all zeroes.

`C!` "c-store" 3Ch `( char c-addr -- )`
If `ENDISM` is 1, exclusive-or `c-addr` with 3. Store `char` at `c-addr`. Only one byte is transferred.

`+!` "plus-store" 3Dh `( n|u a-addr -- )`
Add `n|u` to the single-cell number at `a-addr`.

3.7 Registers

As mentioned in section 2.1, the stack pointers SP and RP may only be accessed through special instructions:

SP@ "s-p-fetch" 3Eh (-- *a-addr*)
a-addr is the value of SP.

SP! "s-p-store" 3Fh (*a-addr* --)
Set SP to *a-addr*.

RP@ "r-p-fetch" 40h (-- *a-addr*)
a-addr is the value of RP.

RP! "r-p-store" 41h (*a-addr* --)
Set RP to *a-addr*.

3.8 Control structures

These instructions implement unconditional and conditional branches, subroutine call and return, and various aspects of the Forth DO...LOOP construct.

Branches:

BRANCH 42h (--)
Load EP from the cell it points to, then perform the action of NEXT.

BRANCHI "branch-i" 43h (--)
Add A × 4 to EP, then perform the action of NEXT.

?BRANCH "question-branch" 44h (*flag* --)
If *flag* is false then load EP from the cell it points to and perform the action of NEXT; otherwise add four to EP.

?BRANCHI "question-branch-i" 45h (*flag* --)
If *flag* is false then add A × 4 to EP. Perform the action of NEXT.

EXECUTE 46h (*xt* --)
R: (-- *a-addr*)
Push EP on to the return stack, put *xt* into EP, then perform the action of NEXT.

@EXECUTE "fetch-execute" 47h ($a\text{-}addr_1$ --)
R: (-- $a\text{-}addr_2$)
Push EP on to the return stack, put the contents of *a-addr* into EP, then perform the action of NEXT.

Subroutine call and return:

CALL 48h (--)
R: (-- *a-addr*)
Push EP + 4 on to the return stack, then load EP from the cell it points to. Perform the action of NEXT.

CALLI "call-i" 49h (--)
R: (-- *a-addr*)
Push EP on to the return stack, then add A × 4 to EP. Perform the action of NEXT.

`EXIT` 4Ah `( -- )` `R: ( a-addr -- )`

Put `a-addr` into `EP`, then perform the action of `NEXT`.

`DO...LOOP` support:

`(DO)` "bracket-do" 4Bh `( x1 x2 -- )` `R: ( -- x1 x2 )`

Move the top two items on the data stack to the return stack.

`(LOOP)` "bracket-loop" 4Ch `( -- )` `R: ( n1|u1 n2|u2 -- | n1|u1 n3|u3 )`

Add one to $n_2|u_2$; if it then equals $n_1|u_1$ discard both items and add four to `EP`, otherwise load `EP` from the cell to which it points and perform the action of `NEXT`.

`(LOOP)I` "bracket-loop-i" 4Dh `( -- )` `R: ( n1|u1 n2|u2 -- | n1|u1 n3|u3 )`

Add one to $n_2|u_2$; if it then equals $n_1|u_1$ discard both items, otherwise add `A` × 4 to `EP`. Perform the action of `NEXT`.

`(+LOOP)` "bracket-plus-loop" 4Eh `( n1|u1 -- )` `R: ( n2|u2 n3|u3 -- | n2|u2 n4|u4 )`

Add $n_1|u_1$ to $n_3|u_3$; if $n_3|u_3$ thereby crosses the $(n_2|u_2 - 1)$ to $n_2|u_2$ boundary discard both items and add four to `EP`, otherwise load `EP` from the cell to which it points and perform the action of `NEXT`.

`(+LOOP)I` "bracket-plus-loop-i" 4Fh `( n1|u1 -- )` `R: ( n2|u2 n3|u3 -- | n2|u2 n4|u4 )`

Add $n_1|u_1$ to $n_3|u_3$; if $n_3|u_3$ thereby crosses the $(n_2|u_2 - 1)$ to $n_2|u_2$ boundary discard both items, otherwise add `A` × 4 to `EP`. Perform the action of `NEXT`.

`UNLOOP` 50h `( -- )` `R: ( x1 x2 -- )`

Discard the top two items on the return stack.

`J` 51h `( -- x1 )` `R: ( x1 x2 x3 -- x1 x2 x3 )`

Copy the third item on the return stack to the data stack.

3.9 Literals

These instructions encode literal values which are placed on the stack.

`(LITERAL)` "bracket-literal" 52h `( -- x )`

Push the cell pointed to by `EP` on to the stack, then add four to `EP`.

`(LITERAL)I` "bracket-literal-i" 53h `( -- x )`

Push the contents of `A` on to the stack. Perform the action of `NEXT`.

3.10 Exceptions

These instructions give access to Beetle's exception mechanisms.

THROW "bracket-throw" 54h (--)

Put the contents of EP into 'BAD, then load EP from 'THROW. Perform the action of NEXT. If 'THROW contains an out of range or unaligned address stop Beetle, returning reason code -259 to the calling program (see section 4.3).

HALT 55h (*x* --)

Stop Beetle, returning reason code *x* to the calling program (see section 4.3). If SP is out of range or unaligned, -258 is returned as the reason code.

3.11 Miscellaneous

(CREATE) implements part of the Forth CREATE...DOES> construct. NEXT performs an instruction fetch when Beetle runs out of instructions in the A register.

(CREATE) "bracket-create" 56h (-- *a-addr*)

Push EP on to the stack.

NEXT 00h/FFh (--)

Load the cell pointed to by EP into A then add four to EP.

3.12 External access

These instructions allow access to Beetle's libraries, the operating system and native machine code.

LIB 57h (*i*x n* -- *j*x*)

Call library routine *n*. The parameters passed and returned depend on *n*. If the library routine is not currently available, raise exception -257.

OS "o-s" 58h (*i*x* -- *j*x*)

Make an operating system call. OS typically takes an operating system call number then makes that call and returns any result, with other parameters and result data being communicated via the stack or a shared data area.

LINK 59h (*i*x* --)

Make a subroutine call to the routine at the address given (in the host machine's format, padded out to a number of cells) on the data stack. The size and format of this address are machine-dependent.

3.13 Recursion

These instructions are provided to simplify the task of writing a low-level debugger for Beetle that runs directly on Beetle. Such a debugger might be written in pForth. The instructions perform roughly the same function as the interface calls **run()** and **single_step()**.

RUN 5Ah (*a-addr*$_1$ *a-addr*$_2$ *a-addr*$_3$ *x* -- *n*)

Save the current values of EP, A, SP, RP, 'THROW, 'BAD and -ADDRESS and load *a-addr*$_1$ into SP, *a-addr*$_2$ into RP, *a-addr*$_3$ into EP, and *x* into A. Enter the execution loop. If Beetle halts, restore the contents of the registers to those saved, and push the reason code on to the stack.

STEP 5Bh (*a-addr*$_1$ *a-addr*$_2$ *a-addr*$_3$ *x* -- *n*)

Save the current values of EP, A, SP, RP, 'THROW, 'BAD and -ADDRESS and load *a-addr*$_1$ into SP, *a-addr*$_2$ into RP, *a-addr*$_3$ into EP and *x* into A. Perform the contents of the execution loop once. Restore the contents of the registers to those saved, and push the reason code on to the stack.

3.14 Opcodes

In table 7 are listed the opcodes in numerical order. All undefined opcodes (5Ch–FEh) raise exception -256.

Opcode	Instruction	Opcode	Instruction	Opcode	Instruction
00h	NEXT	1Fh	-	3Eh	SP@
01h	DUP	20h	>-<	3Fh	SP!
02h	DROP	21h	1+	40h	RP@
03h	SWAP	22h	1-	41h	RP!
04h	OVER	23h	CELL+	42h	BRANCH
05h	ROT	24h	CELL-	43h	BRANCHI
06h	-ROT	25h	*	44h	?BRANCH
07h	TUCK	26h	/	45h	?BRANCHI
08h	NIP	27h	MOD	46h	EXECUTE
09h	PICK	28h	/MOD	47h	@EXECUTE
0Ah	ROLL	29h	U/MOD	48h	CALL
0Bh	?DUP	2Ah	S/REM	49h	CALLI
0Ch	>R	2Bh	2/	4Ah	EXIT
0Dh	R>	2Ch	CELLS	4Bh	(DO)
0Eh	R@	2Dh	ABS	4Ch	(LOOP)
0Fh	<	2Eh	NEGATE	4Dh	(LOOP)I
10h	>	2Fh	MAX	4Eh	(+LOOP)
11h	=	30h	MIN	4Fh	(+LOOP)I
12h	<>	31h	INVERT	50h	UNLOOP
13h	0<	32h	AND	51h	J
14h	0>	33h	OR	52h	(LITERAL)
15h	0=	34h	XOR	53h	(LITERAL)I
16h	0<>	35h	LSHIFT	54h	THROW
17h	U<	36h	RSHIFT	55h	HALT
18h	U>	37h	1LSHIFT	56h	(CREATE)
19h	0	38h	1RSHIFT	57h	LIB
1Ah	1	39h	@	58h	OS
1Bh	-1	3Ah	!	59h	LINK
1Ch	CELL	3Bh	C@	5Ah	RUN
1Dh	-CELL	3Ch	C!	5Bh	STEP
1Eh	+	3Dh	+!	FFh	NEXT

Table 7: Beetle's opcodes

4 External interface

Beetle's external interface comes in three parts. The calling interface allows Beetle to be controlled by other programs. The library format provides a simple mechanism for Beetle to access I/O and other system-dependent functions via the LIB instruction. User-written libraries may also be used, allowing Beetle to benefit from previously written code, code written in other languages, and the speed of machine code in time-critical situations. The object module format allows compiled code to be saved, reloaded and shared between systems. pForth is loaded from an object module.

4.1 Object module format

The first six bytes of an object module should be the ASCII codes of the letters "BEETLE"; next should come an ASCII NUL (00h), then the one-byte contents of the `ENDISM` register of the Beetle which saved the module. The next four bytes should contain the number of cells the code occupies. The number must have the same endianness as that indicated in the previous byte. Then follows the code, which must fill a whole number of cells. The format is summarised in table 8 (the bytes in each cell are shown in the order in which they are stored in the file, regardless of the endianness of the machine on which the file is written).

Cell	**Contents**
1	42h 45h 45h 54h
2	4Ch 45h 00h `ENDISM`
3	Length l
4	1st cell of code...
⋮	⋮
$l+3$	...lth cell of code

Table 8: Object module format

Object modules have a simple structure, as they are only intended for loading an initial memory image into Beetle, such as the pForth compiler. Forth does not typically support the loading of compiled code into the compiler, nor there is any need, as compilers are fast, and an incremental style of program development, with only a little source code being recompiled at a time, is typically used.

4.2 Library format

The first six bytes of a library file should be the ASCII codes of the letters "BEETLE"; next should come FFh followed by the one-byte contents of the `ENDISM` register of the Beetle which saved the library. Next is a cell containing the number of library routines in this library. After this come the routines: first a cell containing the number of the routine (the same as that passed to `LIB` to call that routine), then a cell with the length of the routine in bytes, then the machine code itself, padded if necessary with 00h to a whole number of cells. The number of routines, routine numbers and lengths should be stored with the same endianness as that indicated earlier. The format is summarised in table 9 (the bytes in each cell are shown in the order in which they are stored in the file, regardless of the endianness of the machine on which the file is written).

If relocation tables or other data are needed for the machine code to work, they should be included with the code; it is up to the implementation how to decode the machine code sections of the library file.

4.3 Calling interface

The calling interface is difficult to specify with the same precision as the rest of Beetle, as it may be implemented in any language. However, since only basic types are used, and the semantics are simple, it is expected that implementations in different language producing the same result will be easy to program. A Modula-like syntax is used to give the definitions here. Implementation-defined error codes must be documented, but are optional. All addresses passed as parameters must be cell-aligned. There are six calls which a Beetle must provide:

Cell	Contents
1	42h 45h 45h 54h
2	4Ch 45h FFh ENDISM
3	Number of calls
4	1st call number
5	1st call length l
6	1st cell of code...
⋮	⋮
$l+5$	...lth cell of code
⋮	⋮
⋮	further calls
⋮	⋮

Table 9: Library format

run () : integer
Start Beetle by entering the execution cycle as described in section 2.4. If Beetle ever executes a HALT instruction (see section 3.10), the reason code is returned as the result.

single_step () : integer
Execute a single pass of the execution cycle, and return reason code 0, unless a HALT instruction was obeyed (see section 3.10), in which case the reason code passed to it is returned.

load_object (*file, address*) : integer
Load the object module specified by *file*, which may be a filename or some other specifier, to the Beetle address *address*. First the module's header is checked; if the first seven bytes are not as specified above in section 4.1, or the endianness value is not 0 or 1, then return -2. If the code will not fit into memory at the address given, or the address is out of range, return -1. Otherwise load the bForth code into memory, resexing it if the endianness value is different from the current value of ENDISM. The result is 0 if successful, and some other implementation-defined value if there is a filing system or other error.

save_object (*file, address, length*) : integer
Save the *length* cells in Beetle's memory starting at *address* as an object module under the filename or other specifier *file*. The result is 0 if successful, -1 if there is a Beetle error (the address is out of range or the area extends beyond MEMORY), and some other implementation-defined value if there is a filing system or other error.

load_library (*file*) : integer
Load the library specified by *file*, which may be a filename or some other specifier. Return 0 if successful, or some other implementation-defined value if not. It is up to the implementation whether particular library calls may be loaded more than once; if this is allowed, the old version should be overwritten by the new.

save_standalone (*file, size, start, copied, libs*) : integer
Write an executable stand-alone Beetle to the file *file*, which may be a filename or some other specifier. The executable should have `MEMORY` equal to *size*, and the first *copied* cells should have the contents of the *copied* cells in the current Beetle starting at *start*. The library calls specified in the list of cells *libs* should be linked to the executable, and the current values of the registers should be stored. The result is 0 if successful, -1 if the area extends beyond `MEMORY`, or some other implementation-defined value if there is a filing system or other error. See section 4.4 for the behaviour required of the stand-alone Beetle.

Beetle must also provide access to its registers and address space through appropriate data objects.

4.4 Stand-alone Beetles

A stand-alone Beetle should perform the following steps when it is executed:

1. Initialise the registers with the values they held when the stand-alone Beetle was saved.
2. Perform the action of `NEXT`.
3. Perform the action of a call to **run()**.

If the Beetle stops with a reason code, this should be returned to the calling environment if this is supported; otherwise it may be ignored. The stand-alone program should then terminate.

Any library calls which were linked to the stand-alone Beetle must execute correctly when called by the `LIB` instruction. Any other parameter passed to `LIB` should raise exception -257 (library call not implemented).

5 Libraries

Beetle has one standard library, the core I/O library. Its routines mimic the four system-dependent I/O words in the ANS Forth Core Word Set. The descriptions below are identical to those given for bForth instructions (see section 3), except that the opcode is replaced by the routine number, which is passed to `LIB` to call the routine.

`BL` “b-l” 0 `( -- char )`
`char` is the character value for a space.

`CR` “c-r” 1 `( -- )`
Cause subsequent output to appear at the beginning of the next line.

`EMIT` 2 `( x -- )`
If ***`x`*** is a graphic character in the implementation-defined character set, display ***`x`***. The effect of `EMIT` for all other values of ***`x`*** is implementation-defined. When passed a character whose character-defining bits have a value between 20h and 7Eh inclusive, the corresponding character from the ASCII code [2] is displayed.

`KEY` 3 `( -- char )`
Receive one character ***`char`***, a member of the implementation-defined character set. Keyboard events that do not correspond to such characters are discarded until a valid character is received, and those events are subsequently unavailable. Any standard character returned by `KEY` has the numeric value specified by the ASCII code [2].

For more precise information on the behaviour of the library calls, see the descriptions of the corresponding words in [1, chapter 6].

Acknowledgements

Leo Brodie's marvellous books [3, 4] turned my abstract enthusiasm for a mysterious language into actual knowledge and appreciation.

I have taken or extrapolated the pronunciations of Forth words from [1].

Martin Richards's demonstration of his BCPL-oriented Cintcode virtual processor convinced me that this project was worth attempting. He also gave valuable advice on Beetle's design and proof-read this paper.

Tony Thomas read an earlier draft of this paper, and gave advice on making it more understandable to readers without a knowledge of Forth.

References

[1] ANS X3.215–1994.

[2] ANS X3.4–1974.

[3] L. Brodie, *Starting Forth* (2nd ed., Prentice-Hall; ISBN 0–13–843079–9).

[4] L. Brodie, *Thinking Forth* (Prentice-Hall; ISBN 0-13-917568-7).

[5] R. R. Thomas, *The pForth portable Forth compiler* (unpublished).

[6] R. R. Thomas, *An implementation of the Beetle virtual processor in ANSI C* (unpublished).