Beetle and pForth:
a Forth virtual machine and compiler

R. R. Thomas

St John's College

Computer Science Tripos Part II 1995

11,100 words approximately

Project Originator: R. R. Thomas
Project Supervisor: M. Richards

Original Aims

The aim of the project was to develop a virtual processor specification, a portable C implementation of it, an ANSI standard Forth compiler to run on it, and a user interface and debugger for the system. Depending on the time available after the processor and compiler had been developed, the user interface was to be either a GUI or a simple text interface. Optionally, an assembly code version of the virtual processor was to be produced for the ARM processor.

Work completed

All work was completed except for the optional assembly code version of the virtual processor, and a tiny part of the C implementation of the virtual processor. A text-based user interface was produced. All parts of the system were documented, and thoroughly tested and debugged.

Special difficulties

None.

Contents

Acknowledgements

Martin Richards's demonstration of his BCPL-oriented Cintcode virtual processor convinced me that the project was worth attempting. He also gave valuable advice on Beetle's design and the writing of the dissertation, and supplied code to return a key-press without buffering on UNIX systems.

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].

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

Eugenia Cheng proof-read the dissertation aiming to help me avoid writing learned gibberish. I hope she succeeded occasionally.

All trademarks and registered trademarks are acknowledged.

Typographical notes

Virtual processor registers and instructions, and program code and variables are shown in `Typewriter` font. Terms being defined are shown in **bold** type. Function names are shown in **bold** followed by empty parentheses, thus: **function()**.

Numbers followed by “h” are in hexadecimal.

Addresses are given in bytes and refer to the address space of the Beetle virtual processor except where stated.

Chapter 1

Introduction

The aim of the project that this dissertation describes was to produce a system suitable for teaching the language Forth. This would consist of a virtual processor, called Beetle, which would run an ANSI standard Forth compiler, called pForth (standing for “portable Forth”). The virtual processor would be designed to match the needs of a Forth compiler, and would be implemented in ANSI C, to make the whole system easily portable. The compiler would be written in a mixture of virtual processor assembly code and Forth, and would be provided in compiled form, ready to run on the virtual processor. A simple debugger and user interface would be provided from which the compiler could be run and monitored. The debugger would also be used to aid the porting of the Forth compiler to Beetle.

The Forth compiler and virtual processor were based on earlier work by the author, but both were re-designed and almost completely rewritten, and extended to meet the needs of the project.

The next two sections provide an introduction to the Forth language and the Beetle virtual processor.

1.1 Forth

Forth is an unusual language developed by Charles Moore, a computer programmer, in the late 1960s, to control a radio telescope and provide data analysis facilities. With his system, telescope control, data acquisition and storage, and interactive analysis on a graphical terminal were supported concurrently on what was even then a small computer. To this day, Forth’s main uses are in control and embedded systems, where its combination of compactness and speed are unrivalled.

Forth is an interpreted language, though not in the conventional sense: it provides an interactive environment in which code may be entered and executed immediately, but programs are nevertheless compiled.

Forth is stack-based: it has a **data stack** for calculations and parameter passing, and a **return stack** to hold subroutine return addresses. A rich set of stack operators is provided, and all arithmetic and logic operations implicitly take stack items as their arguments, and return their result on the stack.

Forth’s implicit use of the data stack leads to reverse Polish notation being the natural form for arithmetic. For example, the phrase

```
3 5 -
```

pushes the numbers 3 and 5 on to the data stack, then subtracts the top stack item (5) from the second item (3), and returns -2 on top of the stack.

Forth's syntax is extremely simple. It has two sorts of token: the **word**, which is any sequence of non-space characters, and the **number**, which is any sequence of characters that is not a currently defined word, and that may be interpreted as an integer in the current number base.

The Forth interpreter can be in one of two states, interpreting or compiling. When interpreting, words in the input stream are executed, and numbers are pushed on to the data stack; when compiling, subroutine calls to words and code to push numbers on to the stack is compiled. Code is compiled into the **dictionary**, which is the collection of words in the system.

The words `:` and `;` are used to define new words. The word `:` scans the input stream for the next space-delimited token, which it uses as the name of the word to be defined. It compiles header information for the new word, including its name and a link to the next word in the dictionary. Typically, it also compiles subroutine entry code. It then places the interpreter in compilation mode. The word `;` compiles subroutine exit code and returns the interpreter to interpretation mode.

Here is a definition:

```
: GCD   ( n1 n2 -- n3 )   ?DUP IF  TUCK MOD  RECURSE  THEN ;
```

`GCD` is a word that calculates the greatest common divisor of two numbers. The phrase

```
( n1 n2 -- n3 )
```

is a comment; the word `(` scans the input stream until it finds a close bracket, and discards the input stream up to that point. This comment is a **stack comment**: it gives the **stack effect** of `GCD`, that is, the number and type of the stack items it consumes and returns; in this case it takes two numbers and returns one.

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 dashes serve merely 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.

When `GCD` is executed, the stack should look like this:

```
n1 n2
```

The word `?DUP` duplicates the top item on the stack if it is not zero, so if `n2` is not zero, then it is duplicated by `?DUP`. The stack is now

```
n1 n2 n2
```

`IF` then consumes `n2`. `IF`, like all control structures, is a special sort of word, an **immediate** word. This means that it is executed even when the interpreter is in compiling mode. When executed, it compiles a conditional branch, and leaves the address of the branch on the stack. `THEN` later uses the address to resolve the branch. At run-time, the conditional branch is taken if the top of stack is zero. Note that `THEN` is akin to `ENDIF` in other languages, not `THEN`. As we are assuming that `n2` is not zero, the conditional branch is not taken. The stack is now

```
n1 n2
```

`TUCK` copies the second item on the stack under the first:

```
n2 n1 n2
```

and `MOD` takes the remainder when the second number on the stack is divided by the top number, leaving

```
n2 (n1 mod n2)
```

Finally, `RECURSE` does exactly that. `GCD` implements Euclid's algorithm using the obvious recursive method. An iterative version is also possible:

```
: GCD   ( n1 n2 -- n3 )   BEGIN  ?DUP WHILE  TUCK MOD  REPEAT ;
```

In this case the flag generated by `?DUP` is tested and consumed by `WHILE`, which performs a conditional branch to after `REPEAT` if the flag is zero; if not, execution continues, and when the `REPEAT` is reached, branches back to the `BEGIN`.

Forth programs are typically written with many words with short definitions. These used to be stored in blocks of sixteen by sixty-four characters; increasingly, ordinary text files are used, but the Forth compiler described here, pForth, uses blocks. An advantage of blocks is that small portions of code can be compiled, tested, edited, deleted from the dictionary, and reloaded quickly; the Forth development cycle is much shorter and more often iterated than that of traditional compiled languages.

Forth is a good teaching language because it is interactive and its compiler is so simple. It is possible to learn the language and understand how its compiler works in the time taken just to learn how to program in a modern high-level language such as Modula-3. Forth is primitive, but not old-fashioned; it enforces no particular style of programming. Also, Forth is extensible: user-defined words have the same status as precompiled words, and new control structures and data-structure defining words can be created with ease. This combination of flexibility and extensibility allows modern programming methods such as functional programming and object-orientation to be applied to Forth in a way that is impossible in most other languages.

1.2 Beetle

Beetle is a simple virtual processor intended to run Forth compilers. It has a byte-code instruction set, called bForth, most of whose instructions directly correspond to Forth words. As Forth performs most of its data manipulation, such as calculation and parameter passing, on a data stack, most of these instructions manipulate items on the data stack of the virtual machine. For example, `DUP` duplicates the top item on the stack, and `1+` adds one to the top item. Other instructions deal with control flow, and use addresses stored in the instruction stream, or manipulate the return stack. The registers `SP` (Stack Pointer) and `RP` (Return stack Pointer) point to the top item on the data and return stacks respectively. Both stacks grow downwards in memory.

Beetle's execution cycle works as follows: a word is fetched from the address pointed to by the register `EP` (Execution Pointer) into the `A` (Accumulator) register. `EP` is then incremented by four. The least-significant byte of `A` is copied into the `I` (Instruction) register, and the instruction represented by that opcode is executed. At the same time, `A` is shifted arithmetically right by one byte. The instructions in `A` are copied into `I` and executed in this manner until `A` is empty; the next byte copied into `I` will be either 00h or FFh. Both these opcodes cause another instruction fetch, and execution continues.

Beetle's memory is byte-addressed; the machine word, called a **cell**, is four bytes.

Chapter 2

Preparation

Much of the preparation required for the project went into the writing of the project proposal: the project had to be clearly defined, and a precise work plan, with goals and dates, written. There were many directions the project could have taken; those outlined in the following sections were chosen and fitted together.

Also required were a thorough knowledge and understanding of the ANSI Forth standard [1]. This was initially read at the start of the project, so that the implications for Beetle's design were understood; it was again consulted during the development of pForth.

The rest of this chapter is concerned with the planning of the project. First, the old Forth compiler and virtual processor specification are described, then the changes and improvements to them that were needed, and finally the parts of the project that were entirely new.

2.1 Original components

2.1.1 Forth compiler

The Forth compiler was originally written for Acorn RISC OS, which runs on the ARM processor. It was written to comply loosely with the Forth-83 standard [5] (the standard assumes a 16-bit machine word, while the ARM has a 32-bit machine word). The language extensions added to perform file-handling and other I/O followed the structure of RISC OS, as the standard prescribed only line-oriented console I/O and block-based mass-storage, where data is stored in numbered 1024-byte blocks rather than files. Several machine-specific optimisations had been built into the compiler, tying it firmly to the ARM architecture. The compiler was generated by a BBC BASIC program from assembly and Forth source code, the latter being interpreted by a crude partial Forth compiler written in BBC BASIC.

The ANSI standard contains many extensions to Forth-83, covering areas such as exception handling, local variables and floating-point arithmetic which were not covered by previous standards. In addition, the language is specified in terms of its semantics, rather than a concrete execution model as used in previous standards. This results in many subtle changes to established features of the language.

The ANSI standard requires many implementation-defined behaviours and dependencies to be documented. The original compiler was not documented.

2.1.2 Virtual processor specification

Beetle was also based on the Forth-83 standard, and the original version of the compiler. It had notable deficiencies: there was no provision for address exception checking, it was tied strongly to a particular

compiler, and it contained no optimisations to improve execution speed. No standard I/O was provided; although not a normal part of a processor specification, this is useful for a virtual processor which is intended to be portable.

Most seriously, the specification lacked an external interface, a standard means by which other programs might control Beetle.

2.2 Changes required

2.2.1 Forth compiler

The nature of the changes required to the Forth compiler were such that it was almost entirely rewritten. It was felt to be worth changing the compiler incrementally rather than writing a completely new version because at all times a working version of the compiler would be available for testing. This is especially useful as Forth is an interactive language, so a complete compiler is much easier to test than one which is incomplete.

The first set of changes was intended to make the compiler's architecture more easily portable to other machine architectures. This consisted of removing ARM-based optimisations such as a loop stack for holding the index and increment of Forth's `DO...LOOP` construct, and segregating machine-dependent code, such as the caching of stack items in registers, from the rest of the compiler. A few dependencies such as the use of twos-complement arithmetic were retained: the new compiler would require target machines to have these characteristics. Such environmental dependencies were chosen to allow a simpler and more efficient implementation of the compiler, while reducing its portability as little as possible; for example, almost all modern computers use twos-complement arithmetic in hardware.

The second set of changes made the compiler ANSI conformant. Some optional features specified in the standard were added to ease the addition of the cross-compiler (see below); others were added simply because it made the other necessary changes easier to make.

As one of the aims of the ANSI standard is to make Forth source code more portable between different compilers, some of the changes necessary to improve portability could be effected by changes to make the compiler ANSI conformant. Thus the first two sets of changes were made in parallel.

The third set of changes added the ability to perform cross-compilation, so that pForth could recompile itself to run under Beetle.

2.2.2 Virtual processor specification

Beetle was modified in five main ways: first, an exception mechanism including address checking was added, secondly, it was changed to reflect the ANSI Forth standard rather than the Forth-83 standard, so that it would more directly support the Forth compiler in its new form; thirdly, it was changed from a byte-stream to a word-stream design, to increase efficiency; fourthly, an external interface was specified, so that programs implementing it would provide a well-defined, standard interface to clients. Finally, the entire specification was carefully rewritten to be more precise and clearer, and errors were removed.

2.3 New components

2.3.1 Virtual processor implementation

An implementation of Beetle was to be written in ANSI C, and designed to run on any machine which uses twos-complement arithmetic and supports an ANSI C compiler; thus, it had to be written in strictly correct ANSI C. Features such as I/O which are machine-dependent had to be user-configurable so that the program could easily be configured and compiled to run on any appropriate machine. Little consideration was given to performance: it was anticipated that as the system was not designed for intensive processing, but for teaching Forth, its performance would be adequate. The C processor had to be tested on different machine architectures to ensure that it really was portable.

2.3.2 User interface

Since Forth provides an interactive environment, and since the main purpose of Beetle is to run Forth compilers, only an extremely simple low-level interface was required, primarily to aid debugging of the cross-compiler. Once a Forth compiler for Beetle had been generated successfully, it would be possible for even low-level debugging facilities to be written in Forth and run in the Forth environment.

Chapter 3

Implementation

3.1 Changes to the Forth compiler

An ANSI compliance document for pForth, the portable Forth compiler that resulted from the changes to the original ARM compiler which are described below, is in appendix D.

3.1.1 Overview

The first two sets of changes to the compiler, to make its architecture more easily portable, and to make the compiler ANSI compliant, were highly involved and were carried out in a minutely incremental manner, so are not easily discussed in general terms. When they were completed, it was straightforward to write the metacompiler, and the compiler was able to run an ANSI conformance suite successfully (although some tests failed not because the compiler was not ANSI conformant, but because the test suite had an insufficiently general interpretation of the ways in which double-length numbers can be implemented).

3.1.2 Portability and ANSI compliance

The changes to the Forth compiler to make it portable and ANSI compliant were carried out at the same time. One by one, the 350 or so words in the pForth kernel were compared against the standard, where an equivalent existed, and were modified so that they behaved as specified. At the same time, machine dependencies were removed.

The most involved changes were in the parsing routines, which were respecified by the standard to make them more flexible.

Also, to facilitate the development of the metacompiler, the dictionary mechanism was rewritten so that more than one dictionary could be present in the system.

Without giving a full description of both the workings of Forth compilers and the ANSI standard, it is almost impossible to discuss the changes more fully without being incomprehensible.

3.1.3 Metacompiler

3.1.3.1 Assembler for bForth

The first part of the metacompiler was an assembler for the target machine, Beetle. The source code, which occupies four Forth blocks (a "block" is sixteen lines of sixty-four characters), is given in appendix E.

In the first section of code, a vocabulary is set up for the assembler, and the number of bits per address unit is found and stored in the constant `/BITS` (pronounced "per-bits"). Then the words `CODE` and `END-CODE`, which begin and end an assembler definition, are defined. `INLINE` modifies an assembler definition so that when it is compiled, the machine-code of the definition is expanded inline, rather than compiling a subroutine call. This is mainly used for the primitive assembler words such as `DUP`: these contain only one machine instruction in their definition, and it would be ludicrous to compile a subroutine call to them every time they are used.

The variable `M0` resembles the identically named Beetle register; it points to address zero in the target compiler, and is subtracted from all addresses compiled by the cross-compiler.

The words `FITS` and `FIT,` are used to compile immediate operands for Beetle instructions such as `BRANCHI` (see section A.3.1); `FITS` determines whether an operand will fit in the remainder of an instruction cell, and `FIT,` assembles an immediate operand.

`OPLESS`, `OPFUL` and `OPADR` create words to assemble instructions with no operand, a numeric operand, and an address operand respectively. These together with the words `0OPS`, which makes repeated calls to `OPLESS`, and `BOPS`, which calls `OPADR`, are then used to define an assembler word for each machine instruction. These are named after the machine instructions, but preceded with "B" for "Beetle" to avoid confusion with Forth words with the same names.

3.1.4 pForth for Beetle

pForth for Beetle is constructed as follows. First, the assembler is loaded. Next, a new vocabulary is set up for the metacompiler. The various parts of the metacompiler are loaded, and a new dictionary is defined for the target compiler, together with a new hash table (links to all the words in the pForth dictionary are stored in a hash table).

A special vocabulary is set up for transient words. These are words that are normally executed during compilation, such as the Forth control-structure words (see section 1.1). The transient forms of these words compile code for Beetle, but execute on the ARM version of pForth. Also among the transients are words to create constants and variables. These also must compile run-time code that works on Beetle.

The source code for the Beetle version of pForth, which is almost identical to the ARM version except for the machine-dependent words, is now loaded; thanks to the transient words, it compiles correctly, building a binary image. A few patches to addresses in the image are then made.

Some addresses were not relocated (by having `M0` subtracted from them) during compilation, and are now relocated. Finally, the binary image is saved in a Beetle object file, using a Forth version of the Beetle interface call `save_object` (see section A.4.3).

3.2 Redesign of Beetle

The specification for Beetle resulting from the redesign is in appendix A. The changes which constituted the redesign are detailed below.

3.2.1 Exceptions

Originally, Beetle had no exception mechanism. This resulted in two weaknesses. Most importantly, there was no provision for checking for address references outside Beetle's address space, an obvious safety measure which can easily be built into most virtual processors. If made optional, it need not even lower performance. Also, a general exception mechanism simplifies the implementation of Forth's CATCH and THROW exception handlers. Finally, the ability to stop Beetle and return control to the program that called it is necessary. This can be used for signalling errors, for signalling that Beetle has finished, for communicating results, or to make the calling program perform some service for Beetle.

Two different exception mechanisms were designed: one which redirects the flow of control within Beetle, allowing the bForth program to handle the exception, and one which stops Beetle, and returns control and a reason code to the calling program. Beetle only generates internal exceptions; the bForth exception handler may then raise an external exception if necessary.

Internal exceptions use the THROW instruction, previously named ABORT (see section 3.2.4.1), which supports the ANSI standard Exception word set. Its action is to save EP in the register 'BAD ("tick-bad"), then put the contents of the register 'THROW ("tick-throw") into EP, which causes a branch to the exception handling routine. Conventionally, a number is placed on the data stack indicating the nature of the exception before THROW is executed. An advantage of this design is that virtual processor exceptions are the same as Forth compiler exceptions, and may be handled by high-level code in exactly the same way.

The external exception mechanism is implemented by the instruction HALT, which returns the top item on the data stack as a reason code to the program that called Beetle (or to the calling environment from a stand-alone Beetle).

3.2.2 Address checking

The registers CHECKED and -ADDRESS ("not-address") were added to control address-checking. The value of CHECKED determines whether or not address checking is performed. If it is, then when an address exception occurs the address which caused the exception is placed in -ADDRESS. An exception is then raised with THROW (see section 3.2.1).

Address checking is performed on all addresses processed by Beetle. These fall into two main classes. The first consists of addresses which are passed as parameters to bForth instructions, either on the data stack as with @, which fetches a cell from memory, or in the instruction stream, as with BRANCH, which performs an unconditional branch. The second consists of addresses which Beetle calculates itself, such as the value of EP, which is incremented before each instruction fetch, and may stray outside Beetle's address space.

3.2.3 Word-stream execution cycle

The original Beetle design used a straightforward byte-code instruction set, with cell-sized operands appearing in the instruction stream. While this may be a reasonable design when implemented on an eight-bit processor, it would be inefficient on current processors which often take at least as long to load a byte as a four-byte word. It was decided that the opcodes would be fetched a cell at a time. The instruction Accumulator register, A, was introduced to hold the current cell being decoded. As a result, branches are constrained to cell-aligned addresses.

Having adopted a cell-based execution cycle, three further questions had to be addressed. First, how should operands be represented in the instruction stream? Using unaligned operands would make decoding slow, but using aligned operands would waste space, and also slow Beetle down, as it would have to perform more instruction fetches. Secondly, how would execution continue over gaps in the instruction stream? Thirdly, how could this scheme be made to work on both little-endian and big-endian machines?

3.2.3.1 Operands

Two steps were taken to make operands space-efficient. First, all instructions which take operands in the instruction stream were split into two instructions. The first loads the operand from the address in EP. Figure 3.1 shows the relevant registers just before an instruction $\texttt{OP}_M$, which takes its operand from memory, is executed.

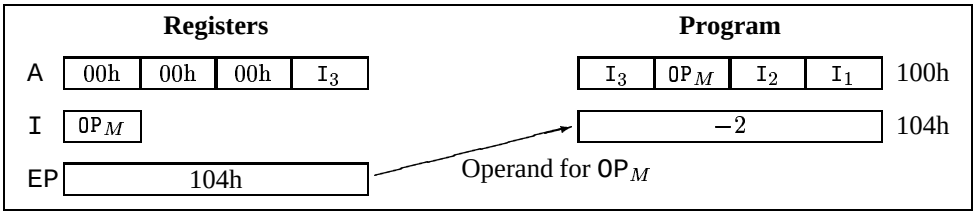

Figure 3.1: Instruction with operand in memory

The second takes the current value of A as its operand. This works as follows: when the instruction is placed in memory, the operand is placed in the remainder of the current cell. When this cell is loaded into A, the operand sits in the most significant bytes. Depending on where the instruction whose operand it is was assembled, the operand may occupy one, two or three bytes. Just after the instruction is decoded, A is shifted arithmetically right by one byte. Its value is now that of the operand, which is signed, as the effect of shifting A arithmetically is to sign-extend the operand. It may now be used by its instruction. Figure 3.2 shows the state of the registers just before an instruction $\texttt{OP}_I$, which takes an immediate operand, is executed.

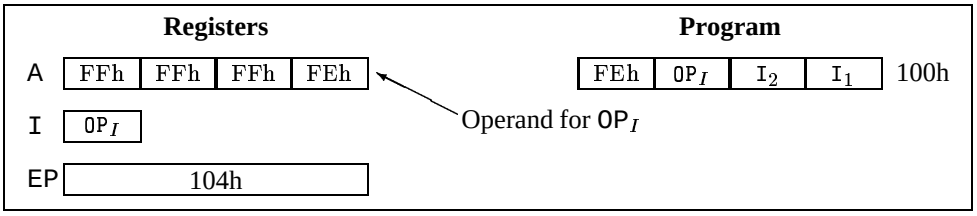

Figure 3.2: Instruction with immediate operand

Secondly, it was observed that there is no need to leave a gap when storing cell-sized operands. Since EP always points to the next cell of instruction opcodes or the next cell-sized operand, when an instruction with a cell-sized operand is stored, the remainder of the current cell may still be filled with instructions, regardless of whether they take operands or not. When they come to be executed, EP will point to their operands if they have one, or to the next instruction cell when the current cell is exhausted.

3.2.3.2 Bridging gaps

Although most gaps are rendered unnecessary by the techniques introduced in the last section, some gaps may still appear in the instruction stream where the flow of control divides and rejoins, as branches must be to a cell-aligned address. The instruction NEXT was introduced to cope. NEXT simply loads the next instruction cell from the cell pointed to by EP, and increments EP. NEXT has two opcodes, 00h and FFh, which are the two possible values of the least-significant byte of A once all the instructions in it have been executed. This byte will be loaded into I in the next cycle and NEXT executed. Thus NEXT performs the double function of a no-op filler and a prompt to do the next instruction fetch. Note that whether a gap in the instruction stream is one, two or three bytes, NEXT will bridge it in one cycle, moving immediately to the next instruction cell.

3.2.3.3 Endianness

Since Beetle deals mainly with cell-aligned cell-sized data and programs, no changes to the specification were necessary to deal with the difference between big-endian and little-endian processors. The difficulty came later when designing the external interface (see section 3.2.6). When transferring code between Beetles of different endianness, resexing would be required. Simply reversing the order of the bytes in every cell in a binary image would correctly resex bForth code and cell-sized data. However, it would jumble byte strings, so the instructions `C@` and `C!`, which load and store a byte respectively, were modified to refer always to the same byte in a cell. Thus, on a little-endian machine the phrase `0 C@` loads the byte at address 0, whereas on a big-endian machine it loads the byte at address 3. After resexing, these addresses hold the same byte on the different machines. Before this change, accessing the same byte with both byte and cell operators was prohibited; the change allowed the restriction to be removed.

The register `ENDISM` was added to store the endianness of the current Beetle. It is used to determine the manner of byte addressing, and to compare against the endianness of saved binary images, so that it can be decided whether resexing is necessary when they are loaded.

3.2.4 Support for the ANSI standard

Beetle was originally designed to support Forth-83 standard Forth compilers. In 1994 the ANSI standard was published, and it was decided that the Forth compiler to be used in this project, pForth, should comply with the new standard. Thus, Beetle had to be modified accordingly. The more substantial changes are detailed below.

3.2.4.1 `ABORT` `THROW`n out

Previously, there was an instruction `ABORT` which branched to the bForth routine at the address in the register `'ABORT`. This supported the primitive `ABORT` mechanism of Forth-83, which restarts the Forth interpreter, returning control to the user in the event of an error. The `'ABORT` register provided a means for applications to change the routine run by `ABORT`. The ANSI standard Exception word set (`CATCH` and `THROW`) allows application programs to handle errors in much the same way as the `CATCH` and `THROW` construct of LISP. This can be handled by the `ABORT` mechanism, so `ABORT` was renamed `THROW` and `'ABORT 'THROW`. Although `THROW` uses exception codes while `ABORT` did not, the code is not dealt with by the instruction itself, but by the exception handling routine to which `THROW` causes a branch.

3.2.4.2 Division conquered

The Forth-83 standard prescribed floored division, in which the quotient is always rounded towards minus infinity, and the remainder has the same sign as the divisor. This generates unfamiliar results: 10 ÷ -7 gives -2 remainder -4 rather than the more usual -1 remainder 3. The Forth-83 standard team preferred floored division as it removes inconsistencies around zero in signed division, but the previous Forth-79 standard prescribed the more usual symmetric division. As both were still in widespread use when the ANSI standard was defined, it required that both methods of division be made available.

It was decided that Beetle should still use floored division, partially because pForth relied on it, but the symmetric division instruction `S/REM` was introduced.

3.2.4.3 `LEAVE` not granted

The Forth-83 standard provided two words for exiting a `DO...LOOP` (see section 3.2.5.2) prematurely: `LEAVE`, which causes a branch to the end of the loop, and `LEAP`, which exits the current word. The ANSI

standard replaced LEAP by UNLOOP, which simply discards the current loop parameters. Thus LEAP is equivalent to UNLOOP EXIT, and LEAVE to UNLOOP followed by a branch. Having separate LEAVE and LEAP instructions was now felt to be unnecessary and both were removed.

3.2.4.4 Cells and address units

To ease the writing of portable programs, the ANSI standard contains words designed to manipulate cell-aligned addresses: for example, CELLS converts a number of cells into a number of address units (typically bytes), and CELL+ adds the size of a cell in address units to an address. Instructions to perform some of these functions were already available, but named 4*, 4+ and so on; they were renamed or new instructions added as appropriate.

3.2.5 Simplification and portability

Beetle had been based heavily on the old pForth compiler. It was felt that its design could be simplified, both to remove some idiosyncrasies, and to make porting easier.

3.2.5.1 Halving doubles

Traditionally Forth compilers have used a 16-bit cell size, and 32-bit double numbers have been used for high precision arithmetic. Beetle was originally designed to support double numbers, but it was felt that 64-bit precision was unnecessary and only complicated the design, and the double number instructions were removed.

3.2.5.2 Removing the loop stack

As well as subroutine return addresses, the return stack traditionally holds the index and count of the Forth DO...LOOP construct, which is like the FOR statement of languages such as Pascal and BASIC. Beetle provided a third stack for this purpose, which allowed greater freedom in the use of the return stack for temporary storage of data values, but since the ANSI standard specifically prohibits such use in standard programs, the loop stack was felt to be redundant, and was removed.

3.2.5.3 Arithmetic right shift

The arithmetic right shift instruction >> was found to be unused in the pForth compiler, and was removed. The instruction 2/, which performs a right shift by one place, remains.

3.2.6 External interface

To allow implementations of Beetle to be written which could be used by other programs in a well-defined way, an external interface had to be specified (see section A.4.3). This was split into three parts:

3.2.6.1 Object module format

An object module format was specified so that binary images of Beetle's memory could be saved and reloaded.

3.2.6.2 Library format

Beetle provides an instruction `LIB` to access input and output functions. Users may write their own libraries. A standard format for libraries was defined to allow users to add them to a Beetle in a consistent manner.

A standard library containing terminal I/O functions was also added to the specification. The functions are based on words from the ANSI Forth standard.

3.2.6.3 Calling interface

A language-independent calling interface which allows programs to make a Beetle execute, either continuously or single-stepping, as well as to save and load binary images and load libraries, was defined.

A call to save an executable stand-alone Beetle was also defined, as was the behaviour of a stand-alone Beetle, so that programs written for Beetle could be made into stand-alone applications.

3.2.7 Recursion

Two instructions, `RUN` and `STEP`, were added to mimic the interface calls which cause Beetle to execute a program, or single step. This makes it easier for debuggers for bForth programs to run on Beetle themselves.

3.3 Implementation of Beetle in ANSI C

The ANSI C implementation of Beetle is described in appendix B.

The most important consideration in the design and implementation of C Beetle was that it should, if possible, work when compiled with any ANSI C system. One concession was made to ease of implementation: C Beetle only works on ANSI C systems which use twos-complement arithmetic. Since this applies to the overwhelming majority of modern C compilers and computers, it was not thought to be a heavy restriction.

3.3.1 Omissions

C Beetle does not meet the full specification for an embedded Beetle: it omits some features which were considered to be either of limited use in a portable Beetle, or difficult to implement portably. These are detailed in section B.2.

The following features were omitted because they would have been difficult to render portably:

The `OS` instruction is intended for direct access to the operating system, and thus cannot be implemented portably. The `LIB` instruction already provides operating-system independent access to services such as I/O.

The save_standalone() interface call would be difficult to implement in a manner which is both useful and portable: while it could merely write an executable shell script which passes the desired binary image to an embedded C Beetle, this would not give either of the intended benefits of stand-alone Beetles, which are increased speed (the Beetle interpreter can be a hand-optimised custom written version) and smaller size.

The load_library() interface call is not useful on a portable system as it is difficult to implement portable libraries within the restrictions imposed by the library file format.

The instructions RUN and STEP (see section 3.2.7) were omitted because there was not enough time to implement and test them properly.

3.3.2 Portability

Although the only requirement of C Beetle is that the C compiler use twos-complement arithmetic, it does contain several other machine dependencies. These are isolated in the header file bportab.h, and are detailed in section B.3.1.

Types must be defined to match Beetle's fundamental types, the byte and cell. The endianness of the machine on which Beetle is to be compiled must be specified, as although this can be discovered at run-time, the byte load and store instructions can be compiled more efficiently if the endianness is known in advance. Whether the C compiler uses floored or symmetric division must also be specified.

Finally, macros must be supplied that perform an arithmetic right shift (the ANSI C standard does not specify that the >> operator must do this on signed types, only that it may), and call a C function given its address (as type void (*)()).

Code is provided that tests most of these settings, thus enabling a program using an embedded Beetle to check that the Beetle has been compiled correctly.

In bportab.h the endianness setting is used to define a macro FLIP(x) which returns x on little-endian machines, and x with the least significant two bits inverted on big-endian machines. This is used by the byte addressing instructions C@ and C!. Beetle is thus slightly less efficient on big-endian machines, but the difference is slight in comparison with the other overheads introduced by the C compiler, and in practice these instructions account for less than 3% of instructions executed (see table 4.1).

Macros are also defined to perform floored and symmetric division. They both use the C division and remainder operators, / and %, with the results adjusted for the form of division that the C compiler does not use.

3.3.3 Calling interface and register access

The main header, which provides the external interface to Beetle given in section A.4.3, is beetle.h. This provides function prototypes for the interface calls, and global variables for the registers. The registers are mostly C variables, and are accessed as such, except for two which are #defined constants. These are ENDISM and CHECKED, which are prohibited from changing while Beetle is running by the specification.

An extra call was added to the interface, **init_beetle()** (see section B.3.4). This, when passed a pointer to a byte array, its size, and an initial value for EP, initialises Beetle's registers, using the array as the memory. It also checks that Beetle has been compiled properly, and halts with advice on how to change bportab.h before recompiling if not.

3.3.4 Interpreter

The contents of the execution loop is implemented in the obvious manner: first, I is set to the least-significant byte of A and A is shifted arithmetically one byte to the right, then a large switch statement decodes the opcode and performs the appropriate action. If an invalid opcode is found, the appropriate exception is raised.

The implementation of the instruction set is discussed below. Where a group of instructions has been implemented in a similar manner, such as the logical operators AND, OR, XOR and INVERT, only one is described. The instructions are split into sections under the headings used in section A.3. The specification of each instruction whose implementation is described is quoted from section A.3 with the C code that

implements it (the code that checks the validity of addresses is omitted). See section A.3 for an explanation of the instruction specification layout.

3.3.4.1 Stack manipulation

OVER (x_1 x_2 -- x_1 x_2 x_1)

Place a copy of x_1 on top of the stack.

```
SP--;
*SP = *(SP + 2);
```

The simple stack operators such as DUP, SWAP and OVER shuffle the topmost items on the stack. Here, OVER first decrements the stack pointer to make room for the stack item to be copied, then copies it from what is now the third position on the stack (SP + 2). The stack pointer SP has type CELL * in the C implementation, so that SP is not a Beetle address, but a C pointer.

ROLL (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.

```
temp = *(SP + *SP + 1);
for (i = *SP; i > 0; i--) *(SP + i + 1) = *(SP + i);
*++SP = temp;
```

ROLL is the most complex stack operator to implement: to rotate the uth item on the stack to the top it must first be copied, then the rest of the stack shuffled down, before replacing it on top of the stack.

?DUP (x -- 0 | x x)

Duplicate x if it is non-zero.

```
if (*SP != 0) { SP--; *SP = *(SP + 1); }
```

?DUP is a conditional operator. The implementation is obvious, but relies on the fact that C means the same thing by zero as Beetle: a cell with all bits clear. The fact that C takes a low-level view of datatypes is often exploited in C Beetle to make the implementation simpler. It is important that this is realised, because in a few cases the correspondence breaks down, and the C implementation must be thought through more carefully.

>R (x --)
R: (-- x)

Move x to the return stack.

```
*--RP = *SP++;
```

Here we see the use of the return stack pointer RP, which is used in exactly the same way as SP.

3.3.4.2 Comparison

> (n_1 n_2 -- *flag*)

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

```
SP++;
*SP = (*SP > *(SP - 1) ? B_TRUE : B_FALSE);
```

The conditional tests such as > can be implemented directly with the C equivalents. Two symbols, B_FALSE and B_TRUE, are defined in the header file bportab.h to be Beetle's values for false and true flags, because C uses the value 1 for true rather than a cell with all bits set, as Beetle uses.

3.3.4.3 Arithmetic

1	(-- 1)
Leave one on the stack.	
*--SP = 1;	

The benefit of requiring that the C compiler use twos-complement number representation and arithmetic is that no translation is needed when loading and storing numbers to and from Beetle. Here, the constant 1 has the obvious definition.

+	($n_1 \mid u_1$ $n_2 \mid u_2$ -- $n_3 \mid u_3$)
Add $n_2 \mid u_2$ to $n_1 \mid u_1$, giving the sum $n_3 \mid u_3$.	
SP++; *SP += *(SP - 1);	

+, like most of the arithmetic operators, leaves one less item on the stack than it consumes, so it must increment the stack pointer (SP++). Because the C compiler and Beetle both use twos-complement arithmetic, + can use C's addition operator directly on values in Beetle's memory.

/MOD	(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.	
temp = MOD(*(SP + 1), *SP, i); *SP = DIV(*(SP + 1), *SP); *(SP + 1) = temp;	

/MOD uses the macros DIV(a, b) and MOD(a, b, t) to perform the division and remainder operations. These are defined so that they produce a floored quotient regardless of the C compiler's method of division. The variable i is used by the MOD macro as a temporary variable.

MAX	(n_1 n_2 -- n_3)
n_3 is the greater of n_1 and n_2.	
SP++; *SP = (*(SP - 1) > *SP ? *(SP - 1) : *SP);	

Finding the maximum of two numbers requires a branch on most processors, even though it is a common operation. One of the advantages of a virtual processor is that the instructions can perform complex actions, because complex operations may take up little memory in an interpreter, whereas algorithmic complexity is almost inevitably related to gate complexity in real processors.

3.3.4.4 Logic and shifts

AND	(x_1 x_2 -- x_3)
x_3 is the bit-by-bit logical "and" of x_1 with x_2.	
SP++; *SP &= *(SP - 1);	

The logical operators are implemented in the same simple manner as the arithmetic operators, again using C's equivalents.

LSHIFT	(x_1 u -- x_2)
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.	

```
                    SP++;
    *(SP - 1) < 32 ? (*SP <<= *(SP - 1)) : (*SP = 0);
```

The logical shifts cannot just use C's operators, because Beetle's specification says that the value returned when the shift is greater than thirty-two places must have all bits cleared, whereas the C standard leaves the value implementation-defined (as does the ANSI Forth standard). Thus the test

```
*(SP - 1) < 32
```

is introduced, and the code

```
(*SP = 0)
```

to cope when the shift is greater than 32.

3.3.4.5 Memory

@	(*a-addr* -- *x*)
x is the value stored at *a-addr*.	

```
                *SP = *(CELL *)(*SP + M0);
```

The instruction @ fetches a cell from memory. The C code for it demonstrates how Beetle addresses are manipulated with C pointers: `M0` is a pointer of type `BYTE *`, and thus pointer arithmetic can be used to give a pointer to a Beetle address by adding it to `M0`. This pointer can then be cast as `CELL *` so that the cell's contents can be fetched.

3.3.4.6 Registers

SP@	(-- *a-addr*)
a-addr is the value of SP.	

```
                    SP--;
        *SP = (CELL)((BYTE *)SP - M0) + CELL_W;
```

SP@ is the most complicated of the four instructions which set and read the values of the two stack pointers. First, `SP--` makes room for the value being read on the stack. Next, the Beetle address of `SP` is calculated from the C pointer by pointer subtraction. As `SP` is a `CELL *`, it must first be cast to `BYTE *` so that `M0` can be subtracted from it. Finally, `CELL_W`, the number of bytes in a cell, is added, as it is the value of `SP` before decrementing that is required.

3.3.4.7 Control structures

BRANCH (--)

Load EP from the cell it points to, then perform the action of NEXT.

```
EP = (CELL *)(*EP + M0);
NEXT;
```

BRANCH causes an unconditional branch by assigning to EP, the execution pointer. The address must be converted from Beetle form to a C pointer using the same method as @. Then the macro NEXT is executed, which has the same effect as the NEXT instruction (see section 3.3.4.10).

BRANCHI (--)

Add A × 4 to EP, then perform the action of NEXT.

```
EP += A;
NEXT;
```

BRANCHI is different from BRANCH in two ways: its operand is in A, and it is a displacement in cells, not an absolute address. This happens to make its implementation much simpler.

(DO) (x_1 x_2 --)
R: (-- x_1 x_2)

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

```
*--RP = *(SP + 1);
*--RP = *SP++;
SP++;
```

(DO) illustrates simple manoeuvring between the return and data stacks. Note that the two items end up in the same order on the return stack as they were in on the data stack; the obvious manipulation

```
*--RP = *SP++; *--RP = *SP++;
```

reverses the order.

(LOOP) (--)
R: ($n_1|u_1$ $n_2|u_2$ -- | $n_1|u_1$ $n_3|u_3$)

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.

```
(*RP)++;
if (*RP == *(RP + 1)) { RP += 2; EP++; }
 else { EP = (CELL *)(*EP + M0); NEXT; }
```

(LOOP) combines a conditional stack operation like ?DUP with a branch like BRANCH.

UNLOOP (--)
R: (x_1 x_2 --)

Discard the top two items on the return stack.

```
RP += 2;
```

UNLOOP performs part of the function of the Forth word LOOP. It is used when a loop is exited, to discard the loop index and limit.

3.3.4.8 Literals

(LITERAL)	(-- *x*)
Push the cell pointed to by EP onto the stack, then add four to EP.	
`*--SP = *EP++;`	

(LITERAL) is used to encode immediate constants in programs. It pushes the next cell in the instruction stream on to the data stack.

3.3.4.9 Exceptions

THROW	(--)
Put the contents of EP into 'BAD, then load EP from 'THROW. Perform the action of NEXT.	

`*(CELL *)(M0 + 8) = BAD = (CELL)((BYTE *)EP - M0);`
`EP = (CELL *)(*THROW + M0);`
`NEXT;`	

As a copy of 'BAD must be held in Beetle's memory (at M0 + 8), a double assignment is made. The effect of loading EP from 'THROW is to branch through a vector to the exception handling routine. NEXT (see section 3.3.4.10) then performs an instruction fetch after the branch.

HALT	(*x* --)
Stop Beetle, returning reason code *x* to the calling program.	
`return (*SP++);`	

HALT stops Beetle, consuming the item on top of the data stack and returning it as the reason code.

3.3.4.10 Miscellaneous

(CREATE)	(-- *a-addr*)
Push EP onto the stack.	
`*--SP = (CELL)((BYTE *)EP - M0);`	

(CREATE) is used in the run-time code of most data structures to push the address of their data on to the stack. Since the code to do so is short, it can be arranged that this address is EP.

NEXT	(--)
Load the cell pointed to by EP into A, and add four to EP.	
`A = *EP++;`	

NEXT performs an instruction fetch by loading A from the address in EP and incrementing EP by one cell.

3.3.4.11 External access

`LIB` (*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.

```
if ((UCELL)(*SP) > 8) { *--SP = -257; goto throw; }
                else lib((UCELL)*SP++);
```

`LIB`'s implementation is almost self-explanatory. The file `lib.c` implements the standard library. Note that library routines up to 8 are available: although the standard library only contains routines up to 3, C Beetle adds some file-handling routines to implement the Forth block system. These are not documented.

`LINK` (*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 is machine dependent.

```
LINK;
```

A typical definition of the `LINK(f)` macro is

```
SP++; (*(void (*)(void))*(SP - 1))().
```

3.3.5 Address checking

If `CHECKED` is set to one, then all addresses used by each instruction are checked before they are used, to ensure that they are both in range, and, for aligned addresses, aligned. The macros used to accomplish this are `CHECKC(a)` to check character-aligned addresses

```
if ((UCELL)((BYTE *)(a) - M0) >= MEMORY) {
      *(CELL *)(M0 + 12) = ADDRESS = (BYTE *)(a) - M0;
      goto invadr;
}
```

and `CHECKA(a)` to check cell-aligned addresses

```
CHECKC(a);
if ((unsigned int)(a) & 3) {
      *(CELL *)(M0 + 12) = ADDRESS = (BYTE *)(a) - M0;
      goto aliadr;
}
```

where the branches to `invadr` and `aliadr` cause the appropriate address exceptions to be raised.

3.3.6 Implementation and testing

The implementation centred around the testing. Each group of instructions (arithmetic, branching, exceptions etc.) was first implemented in the **single_step()** interface call, and a test program was then written to test them. At the same time, a range of routines useful for debugging was developed, to perform tasks

such as displaying the stack and assembling programs. Some of these later turned out to be useful in the implementation of the user interface (see section 3.4.1).

When all the instructions had been implemented, the **run()**, **load_object()** and **save_object()** interface calls were also implemented and tested in a similar way.

A sample test program and its output are given in appendix F.

3.4 Design and implementation of the user interface

The user interface is described in appendix C.

The user interface was required to have only the minimal functionality needed to debug pForth running on Beetle. Thus the main design aims were that it should, if possible, be as portable as C Beetle, and simple to implement. Implementation took two weeks for the main bulk of the interface; some debugging was done and additions made while it was in use thereafter. This made it by far the quickest component to program.

3.4.1 Command set

The command set was thus chosen according to the functions already available among the debugging utilities and the additional functions that were vital to the user interface's usability for debugging pForth.

From the first set came the stack manipulation commands, which pop and push numbers to and from the stacks, and display the stacks, and the commands that load and save object files. From the second set came the register and memory assignment and display commands, including the memory dump command `DUMP`, and the execution commands. Here the programming effort lay in writing the parsing routines, and in performing full error checking; the operation of the commands is trivial. The only command requiring much additional programming for its operation was `DISASSEMBLE`. This works rather like Beetle's execution cycle, but displays the instructions instead of executing them.

Although C's standard library contains many functions for parsing, some additional programming was required. First, commands may be abbreviated; a function was written to compare a string of any length with all the commands and find the first with which it matches. The C function `strtol` was used to parse numbers, and was wrapped in error-checking and base detection code (the user interface uses the convention that hex numbers are followed by "h" or "'H").

The user interface was much simplified by having only two argument formats for most commands: either a single number, or a pair of numbers. In the latter case, the numbers can be separated by a plus sign, in which case they are taken as a base and offset, or not, when they are taken as the start and end of a range. These formats were general enough to be used for the disassemble command, the memory dump command, and the save object file command. The file commands additionally require a file name.

3.4.2 Error handling

The user interface was designed to perform comprehensive error checks. This often introduces difficulties when escape routes for error conditions have to be found, but because of the flat structure of the program, this was not a problem: most error conditions simply cause a premature exit from the function in which they occur, and return up the calling hierarchy; errors in argument parsing cause a `longjmp` which returns to the main loop of the user interface, to await another line of input.

3.4.3 Adding commands

The commands `FROM` and `INITIALISE` were added to the design after the user interface had already been written and tested. However, the program is simple enough that these commands were added easily and without error: the only modifications required were to add their names to the enumeration and string array that hold all the commands, to increment the variable holding the total number of commands, and to add the code to implement the commands. At the last minute, after all the project programming was complete, another command, `COUNT`, was added, to count how many times each bForth instruction is executed when single-stepping. This was added in under half-an-hour, and worked first time.

Chapter 4

Evaluation

4.1 pForth

The RISC OS version of the compiler does its job, which is to cross-compile itself to run under Beetle. It is now ANSI compliant, and runs an ANSI conformance suite, which tests most of the Core word set, successfully. Most of the other words are themselves tested thoroughly by running the cross compiler.

Similarly, the Beetle version of the compiler works, although it cannot cross-compile itself, as the cross-compiler is hard-wired to compile from ARM to Beetle. It too runs the ANSI conformance suite successfully.

4.2 Beetle's specification

Having continuously combed the specification for errors while the C implementation was being developed, the specification is now about as correct as it could be without being formalised. But is it a good specification? There are several points to make.

4.2.1 Design

Only about one third of possible opcodes are used. This seems wasteful: a whole bit is unused in every opcode. Either extra instructions could be added, or perhaps the bit could be used to replace the `EXIT` instruction, and cause a return from subroutine; this method has been used on hardware processors which use Forth as their assembly language [7]. But in the former case the interpreter would start to grow, and lowered locality of reference might lower efficiency, while in the latter every processing step would be slowed by the need to check the top bit. If new opcodes were introduced, should they implement simple or complex instructions? For example, implementing memory move (the Forth word `MOVE`) as a single instruction would bring enormous performance benefits. A vasty wilderness is the end of this argument, which needs to be ordered by research.

SOD32, another Forth-oriented virtual processor written in C (no reference available) has only thirty-two instructions, and encodes six of these in every four-byte cell, together with an optional return from subroutine bit. Yet its code is only 16% more compact than that of Beetle, which runs 40% faster (see section 4.3.3). The benchmark compiled into 4,508 bytes on SOD32, and 5,372 on Beetle. This compares with 10,448 bytes for SOD32's precompiled dictionary, and 16,596 for Beetle's. C Beetle itself is 24,300 bytes long (for a program which simply loads and runs pForth, and hence includes only the **run()** and **load_object()** functions). The SOD32 interpreter is 21,484 bytes long.

Some measurements were made of instruction usage: the instructions obeyed during a run of the ANSI conformance suite were recorded, and each instruction's frequency as a proportion of total execution is shown in table 4.1. 6,788,027 instructions were executed in total.

Instruction	%	Instruction	%	Instruction	%	Instruction	%
NEXT00	9.75	DUP	4.15	DROP	1.76	SWAP	3.00
OVER	5.77	ROT	0.53	-ROT	0.30	TUCK	2.16
NIP	0.52	PICK	0.36	ROLL	0.18	?DUP	0.65
>R	0.83	R>	0.83	R@	1.90	<	0.11
>	0.96	=	2.49	<>	0.57	0<	0.15
0>	0.09	0=	0.25	0<>	0.00	U<	0.00
U>	0.01	0	0.59	1	0.86	-1	0.19
CELL	0.01	-CELL	0.45	+	2.81	-	0.23
>-<	0.22	1+	2.19	1-	0.47	CELL+	2.20
CELL-	0.41	*	0.02	/	0.00	MOD	0.00
/MOD	0.00	U/MOD	0.00	S/REM	0.00	2/	0.06
CELLS	0.46	ABS	0.00	NEGATE	0.07	MAX	0.00
MIN	0.18	INVERT	0.19	AND	2.40	OR	0.02
XOR	0.17	LSHIFT	0.01	RSHIFT	0.19	1LSHIFT	0.16
1RSHIFT	0.00	@	3.48	!	0.31	C@	2.29
C!	1.00	+!	0.12	SP@	0.03	SP!	0.00
RP@	0.00	RP!	0.00	BRANCH	1.42	BRANCHI	0.00
?BRANCH	5.34	?BRANCHI	0.00	EXECUTE	0.06	@EXECUTE	0.01
CALL	1.36	CALLI	10.50	EXIT	11.93	(DO)	0.70
(LOOP)	0.00	(LOOP)I	1.62	(+LOOP)	0.08	(+LOOP)I	0.75
UNLOOP	0.26	J	0.00	(LITERAL)	1.63	(LITERAL)I	3.39
THROW	0.00	HALT	0.00	(CREATE)	1.66	LIB	0.20
LINK	0.00	RUN	0.00	STEP	0.00	NEXTFF	0.00

Table 4.1: Instruction frequencies during execution of the ANSI conformance suite

The table shows a reasonable distribution of frequencies: although theoretically an instruction set in which each instruction has the same frequency of execution is attainable, in practice a roughly inverse linear degradation of frequency is the best that can be expected [6]. Subroutine call and exit dominate, together with the NEXT instruction (opcode 00h), which is not surprising, as the Forth compiler is mainly composed of subroutine calls, and the function of NEXT is performed at least every fourth instruction. The reason that NEXT has a frequency of only 9.75% is that its function is performed implicitly by other instructions such as CALL, BRANCH and (LOOP); most of pForth consists of CALLI instructions. Most of the arithmetic and comparison operators are well used, although it seems that some of the division instructions might be removed without loss as none has a frequency greater than 0.01%. However, most of the instructions whose frequency is less than 0.01% are instructions which are needed, but only rarely used, such as HALT and the register manipulation instructions SP@, SP!, RP@ and RP!. Some of these could perhaps be combined into a single instruction.

Neither BRANCHI nor ?BRANCHI seems to be used much. This is because most branches compiled are forward branches, where the Forth compiler always uses a non-immediate branch. This is because it compiles more code before the branch is resolved (for example, while compiling the IF...THEN construct), so it must ensure that there is always enough room for the destination address. Optimising the code when the branch is resolved would be tricky, as other branches and address references would have to be modified.

Notice that EXIT is executed slightly more than CALL and CALLI combined. This is because EXECUTE and @EXECUTE also cause a subroutine call; they take their address from the stack rather than the instruction stream.

It seems that it was a good idea to include instructions to place constants on the stack, and to add and subtract constants; `1+` and `CELL+` are especially well used.

4.2.2 Implementability

Despite intentional imprecision in the specification, it was not possible to produce a full implementation of Beetle in portable C. Whether this is a problem with the specification depends on whether a portable C implementation is considered desirable: in a didactic system it probably is. In that case perhaps the specification should be simplified, retaining only those features which can be rendered portably, at the loss of usability for practical applications, or perhaps it should be layered, so that different levels of conformance are laid out for different classes of implementation.

4.2.3 Address checking

There is still at least one area of ambiguity in the specification: address checking. It is not clear whether library routines ought to check addresses internally, and it is not clear when during instruction execution addresses should be checked. The C implementation does check addresses within library routines, and performs all address validation before executing an instruction, so that instructions are never aborted halfway through. This ought to be made explicit.

4.2.4 An embarrassment

When taking measurements of instruction execution frequency (see section 4.2.1), it was found that `NEXT` with opcode FFh was never executed. This instruction was put in the instruction set to be executed after negative immediate numeric literals. However, `(LITERAL)I` performs the function of `NEXT` itself, so `NEXT` with opcode FFh is never executed. The author comforts himself with the thought that, were the instruction set extended to top-bit-set instructions, `NEXT` with opcode FFh would certainly be needed.

4.2.5 Experimentation and generality

For a system which is intended for teaching the mechanisms of Forth compilers, it might be argued that Beetle is not general enough: it supports a particular model of compilation, and it is difficult to see how radically different Forth compilers, with a different number of stacks or different methods of compiling control structures, could be supported as easily as pForth. If Beetle were designed again, the extremely specialised instructions such as `(DO)`, `(LOOP)` and `(CREATE)` could be replaced by more general building-block instructions which could be used to implement a wider variety of Forth compilers.

4.3 C Beetle

The C implementation of Beetle is the most obviously successful part of the project. There are three considerations: correctness, portability and speed.

4.3.1 Correctness

To check that the C implementation of Beetle correctly fulfilled the specification, eighteen test programs were constructed and run as it was being written. One of these together with sample output is shown in

appendix F. For the final version of Beetle all the tests run correctly on all three machines on which Beetle was tested (see section 4.3.2).

Also, the correctness of many of the Beetle arithmetic and logic instructions was effectively double-checked by the ANSI conformance suite, which showed up errors that the test programs had missed.

Finally, the ability of C Beetle to support a Forth compiler running a complex program suggests that the implementation is largely correct.

4.3.2 Portability

C Beetle was compiled on three systems: an Acorn Risc PC running Acorn RISC OS on an ARM610, a DECstation 3100 running DEC ULTRIX on a MIPS R3000, and a SparcStation running Solaris 2 on four SuperSPARCs. The SPARC machine was big-endian; the other two were little-endian. All the tests and the ANSI conformance suite running on pForth worked on all three systems, except for some of the division tests in the ANSI conformance suite, which failed on the ARM version owing to bugs in the C compiler. The objective of portability seems to have been achieved; testing C Beetle on a wider range of machines would ensure this.

4.3.3 Speed

The benchmark used for all comparisons was a single run of the ANSI conformance suite on pForth. Three comparisons were made.

The first was between the different machines running C Beetle. Table 4.2 shows the results of running the benchmark with and without address checking. The times shown are the average of three runs.

System	Checking on	Checking off
ARM	79s	30s
MIPS	41s	26s
SPARC	9.7s	6.0s

Table 4.2: Comparison of interpreted Beetles

Even the slowest system takes a reasonable time to perform the task; the fastest, a modern teaching system, is extremely quick, and demonstrates that Beetle is certainly fast enough for teaching use, when typically many short programs are run interactively, which would be even faster than the long test sequence used here.

It seems that address checking slows Beetle down by at least a factor of two, although this effect is masked on the quicker systems which spend most of the time performing I/O (the source file for the benchmark is read continuously as the benchmark progresses).

The second test was to compare pForth running on the interpreted Beetle with the native ARM version. The latter ran the test in 5.8s. This should be compared with the interpreted version without address checking, as the native version performs no checks, and thus is only about five times faster. Considering the overheads involved in running an interpreter which is itself written in a compiled language, Beetle seems surprisingly fast.

The last test was to run the benchmark on another virtual processor running a Forth compiler. This was done on the SPARC system, using the SOD32 interpreter (no reference available). The benchmark ran in 14s, so it seems that Beetle is reasonably fast for an interpreter.

4.4 User interface

The user interface did all that it was supposed to, but no more. When debugging pForth on Beetle, a text editor displaying a hex and ASCII dump of the binary object file containing pForth was also used. Then code could be studied from a disassembly in the user interface, while literal numbers could be observed in the text editor, as could strings, especially the header fields of Forth words, which contain their names.

Thus, a unified disassembly with ASCII and hex dump command would be useful (perhaps even replacing both `DISASSEMBLE` and `DUMP`). A search command which could find numbers, strings and particular instructions in memory would be useful. Also, although `STEP TO` provides a single primitive breakpoint, a proper breakpoint mechanism would have made debugging easier. Finally, the size of the memory given to Beetle should be variable, for example by a command-line parameter to the user interface.

However, especially if Beetle's `STEP` and `RUN` instructions were implemented, it would be far better to write a debugger in Forth, which would make the functionality available more directly to the Forth programmer. Perhaps the user interface should be left as it is, and only used as a tool of last resort.

Chapter 5

Conclusions

Essentially, the project has been successful. What was promised in the project report work plan (see page 71) has been delivered. The code works correctly, and is reasonably efficient, and the system is well documented.

The rest of this chapter consists of some more specific reflections.

5.1 Quality

An important part of the success of the project is the quality of the programs and documents which were produced. The material completed, both specification and program, is stable, well-tested, and, apart from pForth, well documented. Many projects would require much additional work after the dissertation is completed to make them useful and usable. This project is already both, and would be completed by the completion of C Beetle with the two Beetle instructions omitted, and the writing of a teaching manual for pForth, which would be a lengthy work in itself, and is outside the scope of a project such as this.

In total, over 6,500 lines of code were produced, and over 15,000 words of documentation (printed as the appendices). The code was all thoroughly tested: C Beetle by the suite of programs written to test it and pForth by the ANSI compliance suite, and by dozens of small tests carried out on the RISC OS version as it was being made ANSI compliant. The user interface was tested extensively in use, as was C Beetle, by running pForth.

Equally importantly, the documentation has been carefully proofed, and its accuracy and clarity checked several times.

5.2 Exceptions in C Beetle

Address checking is not actually as straightforward as might appear. This is because checking whether or not a pointer points to an array is difficult, if not impossible, in ANSI standard C. When writing C Beetle common sense was used: pointers are normally simple addresses, and arrays are normally stored contiguously in memory, so by comparing a pointer with the addresses of the first and last elements of the array, the check can be performed. This seems to work on the systems on which C Beetle was tested, but is not guaranteed to work. In a future virtual processor with address checking, more thought might be given to how to implement it in standard C, and to whether another implementation language might be better.

The exception cases of instructions should arguably have been tested as thoroughly as the ordinary cases; only one test program was used to test exception cases.

Testing might be aided by a formal specification for Beetle. However, formal methods are expensive in terms of time and effort (which is why they are most often used in the future tense). Moreover, for a virtual processor, which is a short program, almost all of which should be executed reasonably often, formal methods may not help remove bugs, as most will turn up in use anyway. Nevertheless, a formal specification may still help with accurate implementation, and more importantly, improve the quality of the design.

5.3 Use of Beetle for teaching

The original aim of this project, to produce a virtual processor and Forth compiler for teaching Forth, has often been submerged beneath considerations of the individual components. To put the entire system to this use, pForth would need to be documented in the same detail as Beetle. pForth would also need to be extended to make it a more usable environment for developing Forth programs. Then the system could usefully be used for teaching not only Forth, but also the elements of virtual processors, especially as the source code for Beetle is short (754 lines).

5.4 Separation

One aspect of this project has not been mentioned at all so far. It is that the project was not really a whole, but was undertaken as four separate projects: the design of Beetle, its implementation, the construction of the user interface using it, and the conversion of pForth to run on it.

Although the different sub-projects influenced one another, a physical separation of code and function was maintained between them throughout. The body of code resulting from the project can be divided cleanly into the three programs corresponding to the three programming sub-projects. This is a strong indication that the programs have been well engineered.

The separation between the parts of the project arises from the strong well-defined interfaces between them, which in turn come from having designed each part of the project before implementing it, and not letting the designs mix. Crucial to this was that each part of the project was documented separately, as though four authors were collaborating on the project and needed to understand each other’s components, and as though this dissertation would never be written.

Bibliography

[1] ANS X3.215-1994 (a copy is available in the library while dissertations are being marked).

[2] ANS X3.4-1974.

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

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

[5] Forth Standards Team, *Forth-83 Standard* and Appendices.

[6] M. Richards, private communication.

[7] Harris Corporation, *The RTX 2000 Hardware Reference Manual* (Harris Corporation).

[8] See appendix A

[9] See appendix B

[10] See appendix D

[11] See appendix C

Appendix A

The Beetle Forth Virtual Processor

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.

A.1 Introduction

Beetle is a simple virtual processor designed to enable the easy implementation of ANS Forth compilers, such as pForth [10], 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 [9].

A.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.

A.2.1 Registers

The registers, each with its function and pronunciation, are set out in table A.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.
M0	"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 A.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 M0'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 M0 are shown in table A.2.

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

Table A.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 A.3.7).

A.2.2 Memory

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

A.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.

A.2.4 Operation

Before Beetle is started, `M0`, `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 A.3, except for `I` and `A`, which need not be initialised.

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

Table A.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 A.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 A.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.

A.2.5 Termination

When Beetle encounters a HALT instruction (see section A.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 A.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 A.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.

A.2.6 Exceptions

When a THROW instruction (see section A.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 A.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 A.3.4).
-23	Address alignment exception (see below).
-256	Illegal opcode (see section A.3.14).
-257	Library routine not implemented (see section A.3.12).

Table A.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.

A.3 Instruction set

The bForth instruction set is listed in sections A.3.2 to A.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 A.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 A.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 A.6 shows the subtype relationships. The subtype relation is transitive.

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.

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

Table A.6: The subtype relation

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 A.3.1), rather than being placed separately in the next available cell.

A.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 A.3.9) or branch address (see section A.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.

A.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.

A.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.

A.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 A.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.

A.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 (x_1 -- x_2)

Invert all bits of x_1, giving its logical inverse x_2.

`AND` 32h (x_1 x_2 -- x_3)

x_3 is the bit-by-bit logical “and” of x_1 with x_2.

`OR` 33h (x_1 x_2 -- x_3)

x_3 is the bit-by-bit inclusive-or of x_1 with x_2.

XOR "x-or" 34h (x_1 x_2 -- x_3)

x_3 is the bit-by-bit exclusive-or of x_1 with x_2.

Shifts:

LSHIFT "l-shift" 35h (x_1 u -- x_2)

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 (x_1 u -- x_2)

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 (x_1 -- x_2)

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 (x_1 -- x_2)

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.

A.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*.

A.3.7 Registers

As mentioned in section A.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*.

A.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-addr*$_1$ --)
R: (-- *a-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 (x_1 x_2 --)
R: (-- x_1 x_2)

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

(LOOP) "bracket-loop" 4Ch (--)
R: (n_1|u_1 n_2|u_2 -- | n_1|u_1 n_3|u_3)

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: (n_1|u_1 n_2|u_2 -- | n_1|u_1 n_3|u_3)

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 ($n_1 | u_1$ --)
R: ($n_2 | u_2$ $n_3 | u_3$ -- | $n_2 | u_2$ $n_4 | u_4$)

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 ($n_1 | u_1$ --)
R: ($n_2 | u_2$ $n_3 | u_3$ -- | $n_2 | u_2$ $n_4 | u_4$)

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 $\times$ 4 to EP. Perform the action of NEXT.

UNLOOP 50h (--)
R: (x_1 x_2 --)

Discard the top two items on the return stack.

J 51h (-- x_1)
R: (x_1 x_2 x_3 -- x_1 x_2 x_3)

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

A.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.

A.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 A.4.3).

HALT 55h (x --)
Stop Beetle, returning reason code x to the calling program (see section A.4.3). If SP is out of range or unaligned, -258 is returned as the reason code.

A.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.

A.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.

A.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.

A.3.14 Opcodes

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

A.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.

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 A.7: Beetle's opcodes

A.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 A.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).

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.

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 A.8: Object module format

A.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 A.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).

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 A.9: Library format

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.

A.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:

run () : integer
: Start Beetle by entering the execution cycle as described in section A.2.4. If Beetle ever executes a `HALT` instruction (see section A.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 A.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 A.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 A.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.

A.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).

A.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 A.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.

Appendix B

An implementation of the Beetle virtual processor in ANSI C

B.1 Introduction

The Beetle virtual processor [8] provides a portable environment for the pForth Forth compiler [10], a compiler for ANSI Standard Forth [1]. To move pForth between different machines and operating systems, only Beetle need be rewritten. However, even this can be avoided if Beetle is itself written in ANSI C, since almost all machines have an ANSI C compiler available for them.

Writing Beetle in C necessarily leads to a loss of performance for a system which is already relatively slow by virtue of using a virtual processor rather than compiling native code. However, pForth is intended mainly as a didactic tool, offering a concrete Forth environment which may be used to explore the language, and particularly the implementation of the compiler, on a simple architecture designed to support Forth. Thus speed is not crucial, and on modern systems even a C implementation of Beetle can be expected to run at an acceptable speed.

C Beetle provides only the virtual processor, not a user interface. A simple user interface is described in [8].

The interface to an embedded Beetle is described in [8]. This paper only describes the features specific to this implementation.

B.2 Omissions

Certain features of Beetle cannot be rendered portably in C, and so have been left out of this implementation. Thus, this implementation does not fully meet the specification for an embedded Beetle.

The `OS` instruction is not implemented, as it depends on the operating system of the host machine, and this implementation of Beetle is meant to be portable. If executed, `OS` does nothing.

The interface call **save_standalone()** is not implemented, as it is difficult to implement portably without it merely using C Beetle to run an object file, which lacks the usual advantages of stand-alone programs, speed and compactness. For similar reasons, **load_library()** is not implemented either; the use of `LINK` to access C functions is recommended instead.

The recursion instructions `STEP` and `RUN` are not implemented, although they may be added in a future version.

Machine type	Symbol
MS DOS	`MSDOS`
RISC OS	`riscos`
Unix	`unix`

Table B.1: Supported machine types

B.3 Using C Beetle

This section describes how to compile C Beetle, and the exact manner in which the interface calls and Beetle's memory and registers should be accessed.

B.3.1 Configuration

It is impossible to write an ANSI C implementation of Beetle that will run unaltered on any machine. The few features that are machine-dependent are defined in a machine header file, which is included by `bportab.h`. The appropriate symbol for the machine on which C Beetle is to be compiled must be defined, and the machine header file will then be included automatically. The machine header files available at the time of writing are shown in table B.1, together with the corresponding symbol that should be defined. These symbols are automatically defined by GNU C on the corresponding machines; if another compiler is used, the appropriate symbol should be defined. If C Beetle is to be compiled on a machine type not in the list, a new header file must be added to the directory `bportab`, modelled on the existing header files there, and `bportab.h` must be changed to load it.

The machine type `Unix` refers to most Unix machines. Systems tested successfully include System V Release 4, DEC OSF/1 and DEC ULTRIX. Some problems were encountered, which had effects ranging from impaired operation to non-compilation, but most are too machine and installation specific to be worth describing. The general observation may be made that when compiling on a 64-bit architecture many error messages may be generated by the compiler about pointer conversions. These occur because offsets into Beetle's address space are represented as four-byte numbers. As long as Beetle is never allocated more than 4Gb for its memory, this will not be a problem.

The following types must be defined in a machine header file:

`BYTE:` an unsigned eight-bit quantity (Beetle's byte).

`CELL:` a signed four-byte quantity (Beetle's cell).

`UCELL:` an unsigned four-byte quantity (an unsigned cell).

The following symbols should be defined if appropriate:

`BIG_ENDIAN` should be defined in the makefile to define the symbol of the same name whenever the C compiler is invoked if Beetle is compiled on a big-endian machine.

`FLOORED` should be defined if the C compiler performs floored division.

The following macros should also be defined:

`ARSHIFT(n, p)` should be set to a macro that assigns to `n` the result of shifting it right arithmetically `p` places, where `p` may range from 0 to 31.

`LINK` should be set to a macro that, calls the C function at the machine address held on top of Beetle's stack. This address will typically occupy one cell, but may occupy more. The macro must then alter `SP` so that the address is popped from the stack.

`GETCH` should be set to a macro that returns the next key-press without buffering and echo (like Curses's `getch()`).

`PUTCH(c)` should be set to a macro that prints the character `c` to `stdout` without buffering.

`NEWL` should be set to a macro that prints a `\n` on `stdout` without buffering.

The register `CHECKED` must be set (in `beetle.h`) at compile-time: set to one, address checking will be enabled, and set to zero it will be disabled. Its value cannot be altered at run-time. `MEMORY` can similarly be altered from its default value of 16384 if desired.

The C compiler must use twos-complement arithmetic. The settings in `bportab.h` are tested when the Beetle tests are run. If any is incorrect, the changes that should be made are listed.

B.3.2 Compilation

The utility Make is required to compile Beetle as supplied; this is available on most systems. First, edit the makefile, which is called `Makefile`, so that it will work with the C compiler and linker to be used. The variables `CCflags`, `Linkflags`, `CC` and `Link` may need to be changed. Then set `Touch` so that it will, when prepended to a filename, form a command that changes the timestamp of the file to the current time (on many systems, `touch` is the correct command).

Now run Make with `Makefile` as the makefile, and C Beetle should compile. To test the Beetle, run the script file `btests`. The Beetle object files can be made separately as the target `beetle`; the test programs can be made as the target `btests`.

B.3.3 Registers and memory

Beetle's registers are declared in `beetle.h`. Their names correspond to those given in [8, section A.2.1], although some have been changed to meet the requirements for C identifiers. C Beetle does not allocate any memory for Beetle, nor does it initialise any of the registers. C Beetle provides the interface call **init_beetle()** to do this (see section B.3.4).

The variables `I`, `A`, `MEMORY`, `BAD` and `ADDRESS` correspond exactly with the Beetle registers they represent, and may be read and assigned to accordingly, bearing in mind the restrictions on their use given in [8] (e.g. copies of `BAD` and `ADDRESS` must be kept in Beetle's memory). `THROW` is a pointer to the Beetle register `'THROW`, so the expression `*THROW` may be used as the Beetle register. `CHECKED` is a constant expression which may be read but not assigned to.

`EP`, `M0`, `SP` and `RP` are machine pointers to the locations in Beetle's address space to which the corresponding Beetle registers point. Appropriate conversions (pointer addition or subtraction with `M0`) must therefore be made before using the value of one of these variables as a Beetle address, and when assigning a Beetle address to one of the corresponding registers. Examples of such conversions may be found in `execute.c`, where the bForth instructions are implemented.

The memory is accessed via `M0`, which points to the first byte of memory. Before Beetle is started by calling **run()** or **single_step()**, `M0` must be set to point to a byte array which will be Beetle's memory.

Interface call	Object file
run()	`run.o`
single_step()	`step.o`
load_object()	`loadobj.o`
save_object()	`saveobj.o`
init_beetle()	`storage.o` and `tests.o`

Table B.2: Object files corresponding to interface calls

B.3.4 Using the interface calls

The operation of the interface calls (except for **init_beetle()**) is given in [8]. Here, the C prototypes corresponding to the idealised prototypes used in [8] are given.

Files to be loaded and saved are passed as C file descriptors. Thus, the calling program must itself open and close the files.

`CELL run()`
: The reason code returned by **run()** is a Beetle cell.

`CELL single_step()`
: The reason code returned by **single_step()** is a Beetle cell.

`int load_object(FILE *file, CELL *address)`
: If a filing error occurs, the return code is -3, which corresponds to a return value of `EOF` from **getc()**.

`int save_object(FILE *file, CELL *address, UCELL length)`
: If a filing error occurs, the return code is -3, which corresponds to a return value of `EOF` from **putc()**.

load_library() and **save_standalone()** are not implemented (see section B.2).

In addition to the required interface calls C Beetle provides **init_beetle()** which, given a byte array, its size and an initial value for `EP`, initialises Beetle:

`int init_beetle(BYTE *b_array, long size, UCELL e0)`
: `size` is the length of `b_array` in *cells* (not bytes), and `e0` is the Beetle address to which EP will be set. The return value is -1 if `e0` is not aligned or out of range, and 0 otherwise. All the registers are initialised as per [8], and those held in Beetle's memory as well are copied there. Various tests are made to ensure that Beetle has compiled properly, and the program will stop and display diagnostic messages if not.

Programs which use C Beetle's interface must `#include` the header file `beetle.h` and be linked with the object files corresponding to the interface calls used; these are given in table B.2. `opcodes.h`, which contains an enumeration type of Beetle's instruction set, and `debug.h`, which contains useful debugging functions such as disassembly, may also be useful; they are not documented here. (To use the functions in `debug.h`, link with `debug.o`.)

B.3.5 Other extras provided by C Beetle

C Beetle provides the following extra quantities and macro in `beetle.h` which are useful for programming with Beetle:

`B_TRUE`: a cell with all bits set, which Beetle uses as a true flag.

`B_FALSE`: a cell with all bits clear, which Beetle uses as a false flag.

`CELL_W`: the width of a cell in bytes (4).

`NEXT`: a macro which performs the action of the `NEXT` instruction.

Appendix C

A simple user interface for the Beetle virtual processor

C.1 Introduction

The Beetle virtual processor [8] provides a portable environment for the pForth Forth compiler [10], a compiler for ANSI Standard Forth [1]. C Beetle [8] is a version of Beetle written in ANSI C so that it is itself easily portable. This paper describes a simple user-interface for Beetle which uses C Beetle, as it is itself written in C, though with suitable calling-interface glue it could be made to work with a native code Beetle. The user-interface provides access to Beetle's registers, allows the stacks to be displayed, and provides disassembly and single-stepping.

C.2 Compilation

As supplied, the program consists of one C file, `uiface.c`, which is part of the C Beetle distribution, and is compiled when the entire distribution is made, resulting in the executable file `uiface`. It can be made separately as the Make target `uiface`. `uiface` is a command which takes no parameters.

There is a possible difficulty with compilation: on Unix and similar systems, the macros `GETCH` and `PUTCH(c)` required by C Beetle in `bportab.h` are unwieldy to define. They should be set to `getchar()` and `putchar(c)` respectively (`NEWL` may be defined as `putchar('\n')`), and in the same place the symbol `unix` should be defined (if it is not defined automatically by the C compiler). Extra code is then included from `noecho.c` to initialise and reset the keyboard before and after invoking Beetle within the user-interface. If this does not work, then `noecho.c` and `uiface.c` must be modified.

C.3 Initialisation

When the user-interface is started, an embedded Beetle with 16384 cells of memory is created. The registers which are set to system-dependent values are initialised as shown in table C.1.

`I` is uninitialised. `A` is set to zero: this has the effect that when a `STEP` or `RUN` command is given, a `NEXT` instruction will be performed. Thus, when initialisation is performed by a `LOAD` command (see section C.4.5), the Beetle may be started with `RUN` or `STEP` immediately after the `LOAD` command, without the need for `FROM`. The memory from byte 16 upwards is zeroed.

Register	Initial value
EP	10h
MEMORY	10000h
'THROW	0h

Table C.1: Registers with system-dependent initial values

C.4 Commands

The user-interface is command-driven. All commands and register names may be abbreviated to their first few letters; where commands start with the same letters, there is a set order of precedence, aimed at giving the most commonly used commands the shortest minimum abbreviations (see section C.5). All commands are case-insensitive.

If an unrecognised command is given, or the command has too few arguments, or they are badly formed, an error message is displayed. Command lines containing extraneous characters after a valid command are generally accepted, and the extra characters ignored.

Numbers are all integral, and may be given in either decimal or hexadecimal (which must be followed directly by "h" or "H"), with an optional minus sign.

For some arguments a machine instruction may also be used, preceded by "O", for opcode. The value of a machine instruction is its opcode. Opcodes are byte-wide values; when used as a cell, the most significant three bytes are set to zero.

The syntax of the commands is shown below; literal text such as command names and other characters are shown in `Typewriter font`; meta-parameters such as numbers are shown in angle brackets, thus: $\langle number \rangle$. Vertical bars separate alternatives, and square brackets enclose optional syntax.

There are three types of numeric meta-parameter: $\langle number \rangle$, which is any number; $\langle address \rangle$, which is a valid address (between zero and MEMORY − 1 inclusive, or zero and MEMORY if assigning to RP, R0, SP or S0); and $\langle value \rangle$, which is a number or an opcode.

C.4.1 Registers

Beetle's registers may be displayed by typing their name. The names used are slightly different from the names given in [8], and in fact follow the names used for the corresponding variables in C Beetle. This is to avoid the necessity of typing awkward characters in the names of registers such as 'THROW and -ADDRESS. The registers which have different names from those in [8] are given in table C.2.

Register	Name
'THROW	THROW
'BAD	BAD
-ADDRESS	ADDRESS

Table C.2: Registers and their user-interface names

The registers may also (where appropriate) be assigned to using the syntax

$$\langle register \rangle = \langle value \rangle$$

where ⟨*value*⟩ is in the form given in section C.4. An error message is displayed if an attempt is made to assign to a register such as CHECKED, which cannot be assigned to, or to assign an unaligned or out of range address to a register which must hold an aligned address, such as SP. The FROM command (see section C.4.4) should be used in preference to assigning to EP.

Two additional pseudo-registers are provided by the user-interface: they are called S0 and R0, and are the address of the base of the data and return stacks respectively. They are set to the initial values of RP and SP, and are provided so that they can be changed if the stacks are moved, so the stack display commands will still work correctly.

The command REGISTERS displays the contents of EP, I and A, useful when following the execution of a program.

C.4.2 The stacks

The stacks may be manipulated crudely using the registers SP and RP but it is usually more convenient to use the commands

>D ⟨*number*⟩
D>

which respectively push a number on to the data stack and pop one, displaying it, and

>R ⟨*number*⟩
R>

which do the same for the return stack.

The command DATA displays the contents of the data stack, and RETURN the contents of the return stack. STACKS displays both stacks.

If a stack underflows, or the base pointer or top of stack pointer is out of range or unaligned, an appropriate error message is displayed.

C.4.3 Memory

The contents of an address may be displayed by giving the address as a command. If the address is cell-aligned the whole cell is displayed, otherwise the byte at that address is shown.

A larger section of memory may be displayed with the command DUMP, which may be used in the two forms

DUMP ⟨*address*⟩ + ⟨*number*⟩
DUMP ⟨*address*$_1$⟩ ⟨*address*$_2$⟩

where the first displays ⟨*number*⟩ bytes starting at address ⟨*address*⟩, and the second displays memory from address ⟨*address*$_1$⟩ up to, but not including, address ⟨*address*$_2$⟩. An error message is displayed if the start address is less than or equal to the end address or if either address is out of range.

A command of the form

⟨*address*⟩ = ⟨*value*⟩

assigns the value ⟨*value*⟩ to the address ⟨*address*⟩. If the address is not cell-aligned, the value must fit in a byte, and only that byte is assigned to. When assigning to an aligned memory location, a whole cell is assigned unless the number given fits in a byte, and is given using the minimum number of digits required. This should be noted the other way around: to assign a byte-sized significand to a cell, it should be padded with a leading zero.

C.4.4 Execution

The command `INITIALISE` initialises Beetle as in section C.3.

The command `STEP` may be used to single-step through a program. It has three forms:

`STEP`
`STEP` ⟨*number*⟩
`STEP TO` ⟨*address*⟩

Without arguments, `STEP` executes one instruction. Given a number, `STEP` executes ⟨*number*⟩ instructions. `STEP TO` executes instructions until `EP` is equal to ⟨*address*⟩.

The command `TRACE`, has the same syntax as `STEP`, and performs the same function; in addition, it performs the action of the `REGISTERS` command after each bForth instruction is executed.

The command `RUN` allows Beetle to execute until it reaches a `HALT` instruction, if ever. The code passed to `HALT` is then displayed. The code is also displayed if a `HALT` instruction is ever executed during a `STEP` command.

The command `FROM` sets the point of execution. It has two forms:

`FROM`
`FROM` ⟨*address*⟩

Without arguments, `FROM` performs the function of Beetle's `NEXT` instruction, that is, it loads `A` from the cell pointed to by `EP`, and adds four to `EP`. With an argument, `FROM` sets `EP` to ⟨*address*⟩, and then performs the function of `NEXT`. `FROM` should be used in preference to assigning directly to `EP`.

The command `DISASSEMBLE` disassembles bForth code. It may be used in the two forms

`DISASSEMBLE` ⟨*address*⟩ `+` ⟨*number*⟩
`DISASSEMBLE` ⟨*address*$_1$⟩ ⟨*address*$_2$⟩

where the first disassembles ⟨*number*⟩ bytes starting at address ⟨*address*⟩, and the second from address ⟨*address*$_1$⟩ up to, but not including, address ⟨*address*$_2$⟩. The addresses must be cell-aligned, and the number of bytes must be a multiple of four. An error message is displayed if the start address is less than or equal to the end address, or if either the address or number of bytes is not aligned or is out of range.

The command `COUNTS` displays, if `CHECKED` is 1, the number of times that each Beetle instruction has been executed since the last initialisation (including loads).

C.4.5 Object modules

The command

`LOAD` ⟨*file*⟩ ⟨*address*⟩

initialises Beetle as in section C.3, then loads the object module in file ⟨*file*⟩ into memory at address ⟨*address*⟩. The address must be cell-aligned; if it is not, or if the module would not fit in memory at the address given, or there is some filing error, an error message is displayed.

The command SAVE saves an object module. It has the two forms

SAVE ⟨*file*⟩ ⟨*address*⟩ + ⟨*number*⟩
SAVE ⟨*file*⟩ ⟨*address*$_1$⟩ ⟨*address*$_2$⟩

where the first saves ⟨*number*⟩ bytes starting at address ⟨*address*⟩, and the second saves from address ⟨*address*$_1$⟩ up to, but not including, address ⟨*address*$_2$⟩. The addresses must be cell-aligned, and the number of bytes must be a multiple of four. An error message is displayed if the start address is less than or equal to the end address, or if either the address or number of bytes is not aligned or out of range.

The module is saved to the file ⟨*file*⟩. An error message is displayed if there is some filing error, but no warning is given if a file of that name already exists; it is overwritten.

C.4.6 Exiting

The command QUIT exits the user-interface. No warning is given.

C.5 Command abbreviations

Below are listed the commands, each with its minimum abbreviation. Register names may be abbreviated similarly, as long as the abbreviation does not clash with a command abbreviation: if there is an ambiguity, it is assumed that the command was intended.

Command	Minimum abbreviation
>D	>
>R	>R
COUNTS	C
DISASSEMBLE	D
D>	D>
DATA	DA
DUMP	DU
FROM	F
INITIALISE	I
LOAD	L
QUIT	Q
REGISTERS	R
R>	R>
RETURN	RET
RUN	RU
STEP	S
SAVE	SA
STACKS	ST
TRACE	T

Table C.3: Minimum abbreviations of commands

Appendix D

The pForth portable Forth compiler

D.1 Introduction

pForth is a Forth compiler which complies with the ANSI Forth standard [1]. It is designed to be easily portable between different host environments and processors (the "p" in pForth stands for "portable"). It has been implemented on Acorn RISC OS, and on the Beetle virtual processor [8]. It is designed to be used as a teaching tool, and to this end is written mostly in standard Forth, so that the workings of the compiler can be examined and understood by students learning the language; the compiler itself can be used to illustrate the language and the ANSI standard. Some primitive functions are written in assembly code, and the compiler has a few environmental dependencies, such as requiring twos-complement arithmetic, which are exploited to make the system simpler.

Because it is designed to be easily understood and ported, the compiler is simple, using few optimisations, and with little error checking. It does not implement the whole of the ANSI standard, notably omitting compilation from text files, and double number and floating point arithmetic.

D.2 Documentation required by the ANSI standard

Section D.2.1 contains the ANS labelling for pForth; the other sections give the documentation required in [1, section 4.1], laid out like the corresponding sections in the standard.

D.2.1 Labelling

pForth is an ANS Forth System

providing the Core Extensions word set (except `CONVERT`, `EXPECT`, `SPAN` and `UNUSED`),

providing the Block Extensions word set,

providing `D+`, `D.`, `D.R`, `D0=`, `D>S`, `DABS`, `DNEGATE`, `M+` and `2ROT` from the Double-Number Extensions word set,

providing the Exception Extensions word set,

providing `(`, `BIN`, `CLOSE-FILE`, `CREATE-FILE`, `OPEN-FILE`, `R/O`, `R/W`, `READ-FILE`, `REPOSITION-FILE`, `W/O` and `WRITE-FILE` from the File Extensions word set,

providing `.S`, `?`, `WORDS`, `AHEAD`, `BYE`, `CS-PICK`, `CS-ROLL` and `FORGET` from the Programming-Tools Extensions word set,

providing the Search-Order Extensions word set,

providing `-TRAILING`, `BLANK`, `CMOVE`, `CMOVE>` and `COMPARE` from the String Extensions word set.

D.2.2 Implementation-defined options

D.2.2.1 Core word set

- Aligned addresses are those addresses which are divisible by four.
- When given a non-graphic character, `EMIT` passes the code to the host environment's character output routine.
- `ACCEPT` allows the input to be edited by pressing the backspace key or equivalent to delete the last character entered (or do nothing if there are currently no characters in the input).
- The character set corresponds with one of the permitted sets in the range $\{32 \ldots 126\}$ but is otherwise environment-dependent.
- All addresses are character-aligned.
- All characters in any character set extensions are matched when finding definition names.
- Control characters never match a space delimiter.
- The control-flow stack is implemented using the data stack. All items placed on the stack are single cells except for *`do-sys`* elements, which occupy two cells.
- Digits larger than thirty-five are represented by characters with codes starting at the first character after "Z", modulo the size of the character set.
- After input terminates in `ACCEPT`, the cursor remains immediately after the entered text.
- `ABORT"`'s exception abort sequence is to execute `ABORT`.
- The end of an input line is signalled by pressing the return key or equivalent.
- The maximum size of a counted string is 255 characters.
- The maximum size of a parsed string is $2^{32} - 1$ characters.
- The maximum size of a definition name is 31 characters.
- The maximum string length for `ENVIRONMENT?` is 255 characters.
- Only one user input device (the keyboard) is supported.
- Only one user output device (the terminal display) is supported.
- There are eight bits in one address unit.
- Number representation and arithmetic is performed with binary numbers in twos-complement form.
- Types *`n`* and *`d`* range over $\{\text{-}2^{31} \ldots 2^{31} - 1\}$, types *`+n`* and *`+d`* over $\{0 \ldots 2^{31} - 1\}$ and *`u`* and *`ud`* over $\{0 \ldots 2^{32} - 1\}$.
- There are no read-only data-space regions.
- The buffer at `WORD` is 256 characters in size.
- A cell is four address units in size.

- A character is one address unit in size.
- The keyboard terminal input buffer is 256 characters in size.
- The pictured numeric output string buffer is 256 characters in size.
- The scratch area whose address is returned by `PAD` is 256 characters in size.
- The system is case-sensitive.
- The system prompt is “ok”.
- All standard division words use floored division except `SM/REM`, which uses symmetric division.
- When true, `STATE` takes the value 1.
- When arithmetic overflow occurs, the value returned is the answer modulo the largest number of the result type plus one.
- The current definition cannot be found after `DOES>` is compiled.

D.2.2.2 Block word set

- `LIST` displays “Block” followed by the block number in decimal, then the block as sixteen lines each of sixty-four characters, numbered from nought to fifteen in decimal.
- \ discards up to the next multiple of sixty-four characters when used in a block.

D.2.2.3 Exception word set

- Exceptions -1, -2, -10, -11, -14 and -56 may be raised by the system. Exception values -256 to -511 are reserved for the environment executing pForth to raise exceptions. Value -512 is used by the word `(ERROR")`. Other exceptions in the range {-255...-1} may be raised by the host environment.

D.2.2.4 File word set

The implementation-defined options depend on the host operating system.

D.2.2.5 Search-Order word set

- The search order may contain up to eight word lists.
- The minimum search order consists of the single word list identified by `FORTH-WORDLIST`.

D.2.3 Ambiguous conditions

The following ambiguous conditions are recognised and acted upon; all other ambiguous conditions are ignored by the System (although some of them may result in action being taken by the host machine, such as addressing a region outside data space resulting in an address exception). Dashes denote general ambiguous conditions which could arise because of a combination of factors; asterisks denote specific ambiguous conditions which are noted in the glossary entries of the relevant words in the standard.

D.2.3.1 Core word set

– If a *name* that is neither a valid definition name nor a valid number is encountered during text interpretation, the *name* is displayed followed by a question mark, and ABORT is executed.

– If a definition name exceeds the maximum length allowed, it is truncated to the maximum length (31 characters).

– If division by zero is attempted, -10 THROW is executed. By default this displays the message "division by zero" and executes ABORT.

– When a word with undefined interpretation semantics is interpreted, the message "compilation only" is displayed, and ABORT is executed.

– If the data stack has underflowed when the "ok" prompt would usually be displayed by QUIT, ABORT" is executed with the message "stack underflow". All other stack underflow conditions are ignored.

* If RECURSE appears after DOES>, the execution semantics of the word containing the DOES> are appended to that word while it is being compiled.

* If the argument input source is different from the current input source for RESTORE-INPUT, the flag returned is true.

* If data space containing definitions is de-allocated, those definitions continue to be found by dictionary search, and remain intact until overwritten, when the effects depend on exactly what is overwritten, but will probably include name lookup malfunction and incorrect execution semantics.

* If IMMEDIATE is executed when the most recent definition does not have a *name*, the most recent named definition in the compilation word list is made immediate.

* If a *name* is not found by ', POSTPONE, ['] or [COMPILE], the *name* is displayed followed by a question mark, and ABORT is executed.

* If POSTPONE or [COMPILE] is applied to TO, the semantics of TO are appended to the current definition so that when the definition is executed in interpretation mode, the interpretation semantics of TO are performed, and in compilation mode, the compilation semantics.

D.2.3.2 Block word set

– If a correct block read was not possible because the blocks file could not be opened, the message "blocks file not found" is displayed and ABORT is executed. Other reasons for a block read failing are not detected or handled.

* If a program alters BLK directly input is redirected to the block number stored in BLK; >IN retains its current value. If this is larger than the size of a block, other effects will occur.

* If there is no current block buffer, the buffer whose number is contained in the variable VALID is updated.

D.2.3.3 Double-Number word set

* If *d* is outside the range of *n* in D>S, the least-significant cell of the number is returned.

D.2.3.4 Programming-Tools word set

* If the compilation word list is deleted by FORGET, new definitions will still be added to the defunct word list; if the relevant data structures are subsequently overwritten, incorrect effects will probably occur.
* If FORGET cannot find *name*, *name* is displayed followed by a question mark, and ABORT is executed.

D.2.3.5 Search-Order word set

* Changing the compilation word list during compilation has no effect; changing the compilation word list before DOES> or IMMEDIATE causes the most recent definition in the new compilation word list to be modified; in the former case this may cause the next definition in memory to be partially overwritten.
* If the search order is empty, PREVIOUS has no effect.
* If ALSO is executed when the search order is full, the last word list in the search order is lost.

D.2.4 Other system documentation

D.2.4.1 Core word set

– No non-standard word provided uses PAD.
– The terminal facilities available are a single input (the keyboard), and a single output (the terminal display).
– The available program data space is dependent on the memory available in the host environment.
– 4096 cells of return stack space is available.
– 4096 cells of data stack space is available.
– The system dictionary space required depends on the implementation, and is typically under 32 kilobytes.

D.2.4.2 Block word set

– No multiprogramming system is provided, so there are no additional restrictions on the use of buffer addresses.
– The number of blocks available depends on the system configuration.

Appendix E

bForth Assembler

```
KERNEL VOCABULARY ASSEMBLER  ALSO ASSEMBLER DEFINITIONS
MARKER DISPOSE
: BITS   S" ADDRESS-UNIT-BITS" ENVIRONMENT?
   INVERT ABORT" ADDRESS-UNIT-BITS query not supported" ;
BITS  DISPOSE  CONSTANT BITS/

FORTH DEFINITIONS
: CODE   HEADER ;
ASSEMBLER DEFINITIONS
: END-CODE  ALIGN ;

: INLINE   ( char -- )   LAST >INFO 2 + C! ;

: FITS   ( x addr -- flag )   DUP ALIGNED >-<  BITS/ * 1-
   1 SWAP LSHIFT  SWAP DUP 0< IF  INVERT  THEN  U> ;
: FIT,   ( x -- )   HERE DUP ALIGNED >-<  0 ?DO  DUP C,
   BITS/ RSHIFT  LOOP  DROP ;

VARIABLE M0
: OPLESS   CREATE C,  DOES> C@ C, ;
: OPFUL   CREATE C,  DOES> C@  OVER HERE 1+ FITS IF  1+ C, FIT,
   ELSE C,  0 FIT,  ,  THEN ;
: OPADR   CREATE C,  DOES> C@  OVER HERE 1+ ALIGNED - CELL/
   DUP HERE 1+ FITS  IF  SWAP 1+ C, FIT, DROP  ELSE DROP C,
   0 FIT,  M0 @ - ,  THEN ;

: 0OPS   SWAP 1+ SWAP DO  I OPLESS  LOOP ;
: BOPS   SWAP 1+ SWAP DO  I OPADR  2 +LOOP ;

HEX
41 00 0OPS
BNEXT00 BDUP    BDROP   BSWAP   BOVER   BROT    B-ROT   BTUCK
BNIP    BPICK   BROLL   B?DUP   B>R     BR>     BR@     B<
B>      B=      B<>     B0<     B0>     B0=     B0<>    BU<
BU>     B0      B1      B-1     BCELL   B-CELL  B+      B-
B>-<    B1+     B1-     BCELL+  BCELL-  B*      B/      BMOD
B/MOD   BU/MOD  BS/REM  B2/     BCELLS  BABS    BNEGATE BMAX
BMIN    BINVERT BAND    BOR     BXOR    BLSHIFT BRSHIFT B1LSHIFT
```

```
B1RSHIFT B@     B!      BC@     BC!     B+!     BSP@    BSP!
BRP@    BRP!

44 42 BOPS      BBRANCH  B?BRANCH
47 46 0OPS      BEXECUTE B@EXECUTE
48    OPADR     BCALL
4B 4A 0OPS      BEXIT    B(DO)
4E 4C BOPS      B(LOOP)  B(+LOOP)
51 50 0OPS      BUNLOOP  BJ
52    OPFUL     B(LITERAL)
5B 54 0OPS      BTHROW   BHALT    B(CREATE) BLIB   BOS    BLINK
                BRUN     BSTEP
DECIMAL

FORTH DEFINITIONS  PREVIOUS
```

Appendix F

A typical C Beetle test program

This appendix contains the source and output (when compiled in debugging mode) of one of the C Beetle test programs. This program comes from the file `aritmtit.c`, and tests the arithmetic instructions.

F.1 Source for the arithmetic instructions test

The revision record and program description are omitted. First come the header files and an array called `correct`. This array is found in most test programs, although its type and exact function varies. It always contains values which are compared with others generated in the course of the test. In this case, it contains stack pictures, which are compared with the state of the data stack during the test.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "beetle.h"  /* main header */
#include "btests.h" /* Beetle tests header */
#include "opcodes.h" /* opcode enumeration */
#include "debug.h"   /* debugging functions */


char *correct[] = { "", "0", "0 1", "0 1 -1", "0 1 -1 " QCELL_W,
    "0 1 -1 " QCELL_W " -" QCELL_W, "0 1 " QCELL_W " -" QCELL_W " -1",
    "0 1 " QCELL_W " -5", "0 1 -1", "0 2", "0 3", "0 2", "2 0",
    "2 " QCELL_W, "2 0", "2 0 -1", "2 0 -1 " QCELL_W, "2 0 -" QCELL_W,
    "2 -" QCELL_W, "-2 -1", "2", "2 -1", "0", "1", QCELL_W, "2", "",
    QCELL_W, "-" QCELL_W, QCELL_W, QCELL_W, QCELL_W " 1", QCELL_W,
    QCELL_W " -" QCELL_W, "-" QCELL_W, "-" QCELL_W " 3", "-1 -1",
    "-1", "-1 -2", "1 1" };
```

Next comes the `main` function. It returns zero for success and one for failure, in case it is being used by another program (such as a program running a battery of tests). First it sets up a Beetle with a call to `init_beetle`:

```
int main(void)
{
    int i;
```

```
    init_beetle((BYTE *)malloc(1024), 256, 16);
    here = EP;
    S0 = SP; /* save base of stack */
```

It also sets the point of assembly, `here`, to `EP`, and records the base of the stack in `S0` for later use when examining and displaying the stack. Next, the test program is assembled.

```
    start_ass();
    ass(O_ZERO); ass(O_ONE); ass(O_MONE); ass(O_CELL);
    ass(O_MCELL); ass(O_ROT); ass(O_PLUS); ass(O_PLUS);
    ass(O_MINUS); ass(O_PLUS1); ass(O_MINUS1); ass(O_SWAP);
    ass(O_PLUSCELL); ass(O_MINUSCELL); ass(O_MONE); ass(O_CELL);
    ass(O_STAR); ass(O_SWAPMINUS); ass(O_SLASHMOD); ass(O_SLASH);
    ass(O_MONE); ass(O_MOD); ass(O_PLUS1); ass(O_CELLS);
    ass(O_SLASH2); ass(O_DROP); ass(O_CELL); ass(O_NEGATE);
    ass(O_ABS); ass(O_ABS); ass(O_ONE); ass(O_MAX);
    ass(O_MCELL); ass(O_MIN); ass(O_LITERALI); ilit(3);
    ass(O_SSLASHREM); ass(O_DROP); ass(O_LITERALI); ilit(-2);
    ass(O_USLASHMOD); ass(O_NEXTFF);
    end_ass();
```

Finally, the test program is run, and an appropriate message printed. If debugging is switched on via definition of the symbol `B_DEBUG` some diagnostics are displayed during the run. These are displayed in the next section.

```
    NEXT;   /* load first instruction word */

    for (i = 0; i <= instrs - instrs / 5; i++) {
#ifdef B_DEBUG
        show_data_stack();
printf("Correct stack: %s\n\n", correct[i]);
#endif
     if (strcmp(correct[i], val_data_stack())) {
         printf("Error in AritmtiT: EP = %ld\n", val_EP());
         exit(1);
     }
        single_step();
        if (I == O_NEXT00) i--;
#ifdef B_DEBUG
        printf("I = %s\n", disass(I));
#endif
    }

    printf("AritmtiT ran OK\n");
    return 0;
}
```

F.2 Output from arithmetic instructions test

```
Data stack:
Correct stack:
```

```
I = 0
Data stack: 0
Correct stack: 0

I = 1
Data stack: 0 1
Correct stack: 0 1

I = -1
Data stack: 0 1 -1
Correct stack: 0 1 -1

I = CELL
Data stack: 0 1 -1 4
Correct stack: 0 1 -1 4

I = NEXT00
Data stack: 0 1 -1 4
Correct stack: 0 1 -1 4

I = -CELL
Data stack: 0 1 -1 4 -4
Correct stack: 0 1 -1 4 -4

I = ROT
Data stack: 0 1 4 -4 -1
Correct stack: 0 1 4 -4 -1

I = +
Data stack: 0 1 4 -5
Correct stack: 0 1 4 -5

I = +
Data stack: 0 1 -1
Correct stack: 0 1 -1

I = NEXT00
Data stack: 0 1 -1
Correct stack: 0 1 -1

I = -
Data stack: 0 2
Correct stack: 0 2

I = 1+
Data stack: 0 3
Correct stack: 0 3

I = 1-
Data stack: 0 2
Correct stack: 0 2

I = SWAP
```

```
Data stack: 2 0
Correct stack: 2 0

I = NEXT00
Data stack: 2 0
Correct stack: 2 0

I = CELL+
Data stack: 2 4
Correct stack: 2 4

I = CELL-
Data stack: 2 0
Correct stack: 2 0

I = -1
Data stack: 2 0 -1
Correct stack: 2 0 -1

I = CELL
Data stack: 2 0 -1 4
Correct stack: 2 0 -1 4

I = NEXT00
Data stack: 2 0 -1 4
Correct stack: 2 0 -1 4

I = *
Data stack: 2 0 -4
Correct stack: 2 0 -4

I = >-<
Data stack: 2 -4
Correct stack: 2 -4

I = /MOD
Data stack: -2 -1
Correct stack: -2 -1

I = /
Data stack: 2
Correct stack: 2

I = NEXT00
Data stack: 2
Correct stack: 2

I = -1
Data stack: 2 -1
Correct stack: 2 -1

I = MOD
Data stack: 0
Correct stack: 0
```

```
I = 1+
Data stack: 1
Correct stack: 1

I = CELLS
Data stack: 4
Correct stack: 4

I = NEXT00
Data stack: 4
Correct stack: 4

I = 2/
Data stack: 2
Correct stack: 2

I = DROP
Data stack:
Correct stack:

I = CELL
Data stack: 4
Correct stack: 4

I = NEGATE
Data stack: -4
Correct stack: -4

I = NEXT00
Data stack: -4
Correct stack: -4

I = ABS
Data stack: 4
Correct stack: 4

I = ABS
Data stack: 4
Correct stack: 4

I = 1
Data stack: 4 1
Correct stack: 4 1

I = MAX
Data stack: 4
Correct stack: 4

I = NEXT00
Data stack: 4
Correct stack: 4

I = -CELL
```

```
Data stack: 4 -4
Correct stack: 4 -4

I = MIN
Data stack: -4
Correct stack: -4

I = (LITERAL)I
Data stack: -4 3
Correct stack: -4 3

I = S/REM
Data stack: -1 -1
Correct stack: -1 -1

I = DROP
Data stack: -1
Correct stack: -1

I = (LITERAL)I
Data stack: -1 -2
Correct stack: -1 -2

I = U/MOD
Data stack: 1 1
Correct stack: 1 1

I =
AritmtiT ran OK
```

Project proposal

Part II CST Project Proposal

Beetle and pForth: A Forth Virtual Machine and Compiler

Proposer: R. R. Thomas
St John's College

Originator: R. R. Thomas
Special resources required: 6 Mb extra disk space (CUS)
Supervisor and Director of Studies: M. Richards
Overseers: A. Hopper, A. M. Pitts

The project

The aim of the project will be to develop a virtual processor, Beetle, a Forth compiler to run on it, pForth, itself written in Forth, and a user-interface to the system. Beetle, written in C, will be easily portable, and its object modules will be executable on all the machines on which it runs, with automatic adjustment for endianness. Two versions of Beetle will be produced: one to embed in other programs, and one which simply bootstraps itself, allowing standalone software to be developed. pForth will comply with the ANSI Standard for Forth (hereafter ANS Forth). Depending on the time available after the processor and compiler have been developed, the user-interface might be a GUI or a simple text interface. The user interface will allow simple low-level debugging. One of the design aims of Beetle is that it should be easy to embed in any user-interface or other program, by providing a suitable interface of routines callable by a controlling program; another is that it be easily recodable in machine code for faster execution on a particular machine. Such a recoding will be performed for the ARM processor series.

Beetle will be tested at a low level (instruction by instruction), and by its ability to run pForth. pForth will be tested by a partial ANSI conformance suite, and its ability to compile a range of ‘real’ programs such as a parser generator, a Life simulator, and itself. The low-level tests will be carried out on as many machine architectures as possible, and the compiler tests on as many machines as there are user-interfaces for (the same number, if the simple text-based interface is written portably).

Resources required

6 Mb of extra disk space on CUS will be used for storing backups of project files, and for compiling Beetle for portability testing. Other resources I will use are my own computers: an Atari ST, and an Acorn Risc PC. In case of hardware failure, I have two Atari systems, and the UCS has two Acorn RISC OS machines. In any case, all of the work except for the GUI and machine code version of the virtual processor can be carried out on any machine with an ANSI C compiler and LaTeX (which will be used to typeset the dissertation). Since strict ANSI C will be used to code Beetle and the text-based user-interface, it is expected that, although different C compilers may be used on the different machines, there will be no compatibility problems. The extra disk space will be vital if all my own computers fail and I have to complete the project on CL/UCS equipment.

The starting point

The design of the virtual processor, my own, already exists. However, it will be remodelled, chiefly to increase efficiency: the handling of branches and immediate addressing will be redesigned, endianness will be incorporated, a standard object and library module formats will be designed, and standard libraries corresponding to some of the ANS Forth standard word sets will be added. The calling interfaces to Beetle must also be designed. A preliminary version of the pForth compiler has been written running under Acorn RISC OS; it must have a meta-compiler* added to it, so that it can recompile itself to run under the virtual processor, and an assembler for Beetle’s machine code, and it will be made ANSI conformable. I have the public domain source of a text-file browser on the Atari which I would extend to form the GUI, if it is written; I would need to add the debugger, and a simple terminal emulator to handle console I/O for the virtual processor.

* A Forth meta-compiler is an extension to the compiler which allows it to compile itself; this facility can be used to modify the structure of the compiler, or to cross-compile it, as in this case.

Work plan

While the work plan must give targets to be met, it should also allow for reasonable flexibility owing to external pressures, and changes in the project's design. Thus the work plan is presented in phases, with goals to be achieved during each phase. By the end of each phase, all goals in the phase should be achieved, but their order of execution is unspecified.

Michaelmas term

Goal 1: Implementation of Beetle

- **Subgoal a:** redesign Beetle
- **Subgoal b:** design the implementation of Beetle
- **Subgoal c:** design the testing schedule
- **Subgoal d:** implement and test Beetle

Goal 2: Design of metacompiler

- **Subgoal a:** assess changes/additions to pForth that are needed
- **Subgoal b:** design metacompiler to meet these criteria

Goal 3: ANSIfication of pForth

- **Subgoal a:** discover words to be added/changed in pForth
- **Subgoal b:** design and write changes/additions
- **Subgoal c:** comply with documentation requirements of ANS Forth

Christmas vacation

Goal 1: Write Introduction of dissertation

Goal 2: Write up part of Implementation dealing with Beetle

Goal 3 (optional): Work on GUI (decide whether to proceed therewith)

Checkpoint at the progress report (3rd February)

Beetle and the design for the porting of pForth should be complete (i.e. all goals above, and part of goal 2, subgoal a below); the user interface should also be well under way (at least subgoal a).

Lent term

Goal 1: Write user interface

- **Subgoal a:** design user interface and testing schedule
- **Subgoal b:** implement and test

Goal 2: Implement pForth on Beetle

- **Subgoal a:** design and write assembler for Beetle in pForth
- **Subgoal b:** design and implement porting, with any special fixes needed

Goal 3: Write parts of dissertation covering work done so far

The schedule for this depends heavily on how much writing is done during the rest of the project.

Easter vacation

Goal 1 (optional): Write most of final component (GUI or ARM-optimised Beetle). This will be planned shortly after the progress report when the feasibility of attempting it has been assessed, and a decision to proceed has been made.

Goal 2: Finish the dissertation as far as possible

Easter term

Overspill: Work as necessary on code

Goal: Finish, edit and publish the dissertation

Note: there will be very little time available (I want to spend most of my time on revision). If necessary, most of the Easter term lecture courses can be avoided.