



EuroForth 2009



25th EuroForth Conference

4th – 6th September, 2009

University of Exeter
England

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 25th EuroForth finds us in Exeter for the first time. The three previous EuroForths were held in Cambridge, England (2006), in Schloss Dagstuhl, Germany (2007), and in Vienna, Austria (2008). Information on earlier conferences can be found at the EuroForth home page (<http://www.euroforth.org/>).

Since 1994, EuroForth has a refereed and a non-refereed track.

For the refereed track, one paper was submitted, and one was accepted (100% acceptance rate). For more meaningful statistics, I include the numbers since 2006: nine submissions, five accepts, 56% acceptance rate. The paper was sent to three program committee members for review, and they produced three reviews. I thank the authors for their paper, and the reviewers for their reviews.

Several papers were submitted to the non-refereed track in time to be included in the printed proceedings. In addition, the printed proceedings include the slides for talks that will be presented at the conference without being accompanied by a paper.

These online proceedings also contain late presentations that were too late to be included in the printed proceedings. Also, some of the presentations included in the printed proceedings were updated to reflect the slides that were actually presented. Workshops and social events complement the program.

This year's EuroForth is organized by Peter Knaggs.

Anton Ertl

Program committee

Sergey N. Baranov, Motorola ZAO, Russia
M. Anton Ertl, TU Wien (chair)
David Gregg, University of Dublin, Trinity College
Ulrich Hoffmann, FH Wedel University of Applied Sciences
Phil Koopman, Carnegie Mellon University
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart, University of Teesside
Reuben Thomas, Adsensus Ltd.

Contents

Refereed papers

Ilya E. Tarasov, Veniamin G. Stakhin, Anton A. Obednin: A Forth-Based CAD for System-Level Microelectronic Design 5

Non-refereed papers

Campbell Ritchie and Bill Stoddart: Formulating Type Tagged Parse Trees as Forth Programs 11
M. Anton Ertl: A Look at Gforth Performance 23
Stephen Pelc: Porting Forth Applications and Libraries 32

Presentations

Gerald Wodni, M. Anton Ertl: SWIG-Gforth-Extension (updated) 39
Gerald Wodni: Wrongforth – A reversible Forth (updated) 42

Late presentations

Ulrich Hoffmann: Hardware/Software Co-Design Microcore 46
Duncan Louttit: Writetrack: An Inventor’s Experience 52
Bernd Paysan: Rebuild from Scratch: Internet 2.0 56
Jürgen Pfitzenmaier: Forth Type Checker 60
Klaus Schleisiek: Microcore Status Report 68

A Forth-Based CAD for System-Level Microelectronic Design

Ilya E. Tarasov (Measurement systems), Veniamin G. Stakhin, Anton A. Obednin (IDM-Plus)

Abstract

As part of a Federal project to reconstruct Russian microelectronics, the Zelenograd Innovation and Technological Center was chosen to coordinate R&D works to develop a set of IP cores and an appropriate software platform for cores integration, modeling, and hardware-software co-simulation. This project is carried out under a state contract, supported by the Ministry of Science and Education of the Russian Federation. This article describes features of use of the Forth-based engine for the decision of a problem of creation microelectronic CAD.

Overview

The main subcontractor for this project is the IDM-Plus fabless company, located in the town of Zelenograd near Moscow. The company successfully operates at the microelectronic market, including collaboration with China and Taiwan foundries and the Cadence company. Among their products is a family of 16-bit stack processors (defined as 1894xx), so IDM-Plus is experienced with the stack architecture. Since 2004 a partnership of IDM-Plus and Measurement Systems was established, based primarily on a collaborative research in Forth technologies.

Measurement Systems, located in Kovrov, was founded in 2000 as an R&D company. It manufactures the high-intellectual measurement systems and precision sensors with strong signal processing part. The base technologies of the company include FPGA prototyping of system-on-chip devices, original filtering and statistical algorithms, and a complete set of Forth technologies, including original versions of Forth translators for PC (PM-Forth in 1999, Quark-Forth in 2006), several cross-translators for microcontrollers and FPGA-implemented Forth processors.

Partnership of IDM-Plus and Measurement Systems based on architectural research works and system software development, provided by Measurement Systems, resulted in an RTL design and an FPGA prototype of a new system-on-chip. Its further silicon implementation is being performed by IDM-Plus. This is a common practice for microelectronic design, when separate teams work on high-level and topology levels of a silicon device.

Design flow

Within this project it was necessary to develop an original design flow, suitable for both, worldwide foundries and an 'Angstrom' foundry in Zelenograd. Also, since the cost of mask set grows dramatically with each new technology step, more attention should be paid to the modeling stage [1, 2]. After carefully preformed prototyping and algorithms testing, a decision on acceptability of the silicon implementation must be taken. Furthermore, not only technical questions, but also marketing ones must be resolved, because a silicon implementation of a 130-nm (and deeper) chip is profitable only for high volumes; in case of moderate volumes an FPGA implementation should be considered. Such high risks of getting a non-profitable (or even non-working) device turns silicon developers to assembling large systems from pre-designed, verified building blocks (IP-cores). With an appropriate set of cores the electronic engineer or programmer is able to compose an optimal hardware configuration for the device under construction. From this point of view, it is important to carefully separate architectural

solutions from a low-level topology design. This means that the system-level specialist must be liberated from designing cores at the low level and refocused to rapidly assemble a software model or an FPGA prototype of a new device, when evaluating it in the field.

There are design tools in the market, oriented to high-level system design, like Coware Platform Architect, Mentor Graphics Visual Elite, Xilinx Platform Studio, etc. Some common trends can be highlighted for this class of CAD systems:

- Early integration of embedded software, that enables complex hardware and software cosimulation.
- Modeling at the transaction level (TLM, Transaction Level Modeling), that enables performance speedup compared to RTL level [1]. At this level of abstraction, the developer must use only pre-verified blocks, because at the transaction level we are at risk to write TLM, which can't be implemented with the current level of silicon technology (for example, considering too fast or too large circuit may fail at the layout implementation phase, while the transaction level will be passed successfully).
- Integration with industry standard software tools, such as topology-level CAD software, and, from the other side, mathematical and DSP software tools and high-level languages.
- Using script languages for automated creation of a project and running the design flow in a batch mode. As an example, a Tcl scripting language is widely used in Xilinx software tools, enabling to assemble an FPGA project with a single batch command without any user interaction. This greatly helps in running many iterations while creating a large chip, replacing high cost 'silicon' iterations with cheap modeling iterations on a PC.

Newly developed CAD software, named 'Quark CAD', belongs to the class of system-level design tools and is intended to design a processor-based chips, including multiprocessor system-on-chip (SoC) devices. This project utilizes a software Forth machine, based on the 'Quark Forth' translator, previously developed at Measurement Systems. After analyzing the successful usage of the PM-Forth translator, several features was selected as a basis for this new implementation of the Forth machine:

- Native machine code of x86 processor with separate code and data spaces.
- DLL implementation, interaction with the shell program via the 'Evaluate' word and several exported functions, providing access to stack, memory, and a special 'virtual screen'.
- Output from the Forth-machine, based on a virtual screen in the main memory, similar to a Canvas object in some object-oriented software development tools. This eliminates the necessity of `wm_paint` message handling for an application program, while the shell program is responsible for proper rendering the contents of a virtual screen.
- Wide usage of vector words, replacing nearly all system interface words (PIXEL, EMIT, INTERPRET, OK, etc.) for maximal portability.

While developing the Quark CAD system, a baseline version of Quark Forth was adapted to the new task. First, its assembler version was replaced with a C++ single `quark.h` source text for further build-in into the `gcc` software tool. Due to the Linux compatibility requirement, the Qt library was chosen as the main GUI engine, providing virtual screen visualization, file and GUI operations, and handling the main part of Forth source texts, represented as scripts in the CAD being developed (Fig. 1)

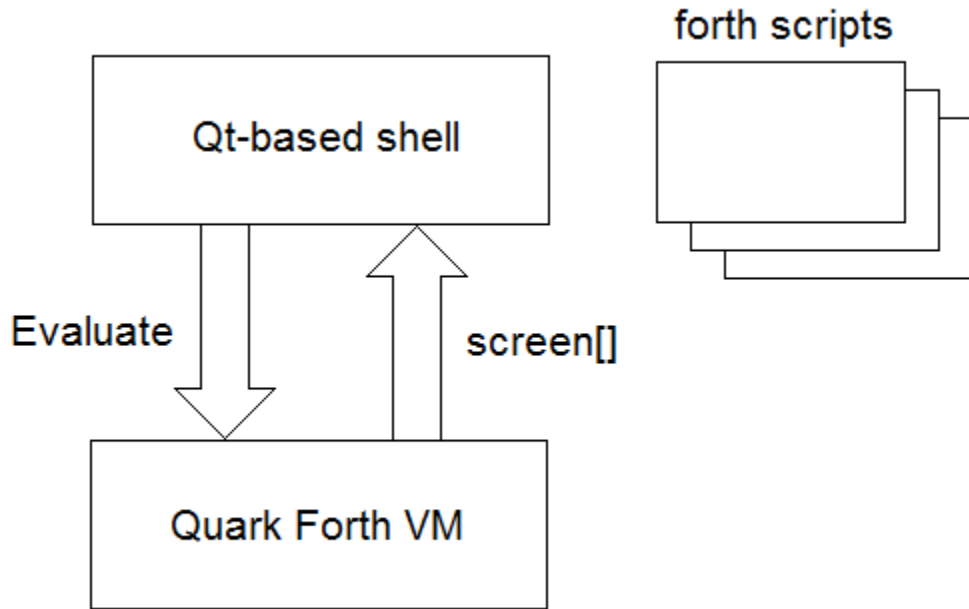


Fig.1. Interaction between C++ Qt and Quark Forth VM in Quark CAD

As shown in Fig. 1, all interactions between the C++ and the Forth parts of the project are strictly limited to a single 'Evaluate' function, which accepts a string to be interpreted by the Quark VM. This VM also is able to open, create, and read/write files without the shell, because basic POSIX-compatible file operations are wrapped by Forth words. So, simple replacing the scripts to load the VM can change the whole functionality of this software. This looks very useful for such complex area as microelectronic design, because many software components (RTL representations, models, etc.) may require an upgrade or reviewing after CAD release. Representing the main part of CAD functionality as Forth scripts, evaluated by VM at the runtime brings great flexibility to this software and makes the component upgrade process transparent and cheap, because no more replacements of executable files are needed. Indeed, it is a common and predictable feature of any Forth-based software, which has been exploited for newly developed CAD.

Forth based product features

Each component in the silicon chip has a several representations for different phases of the design flow (Fig. 2).

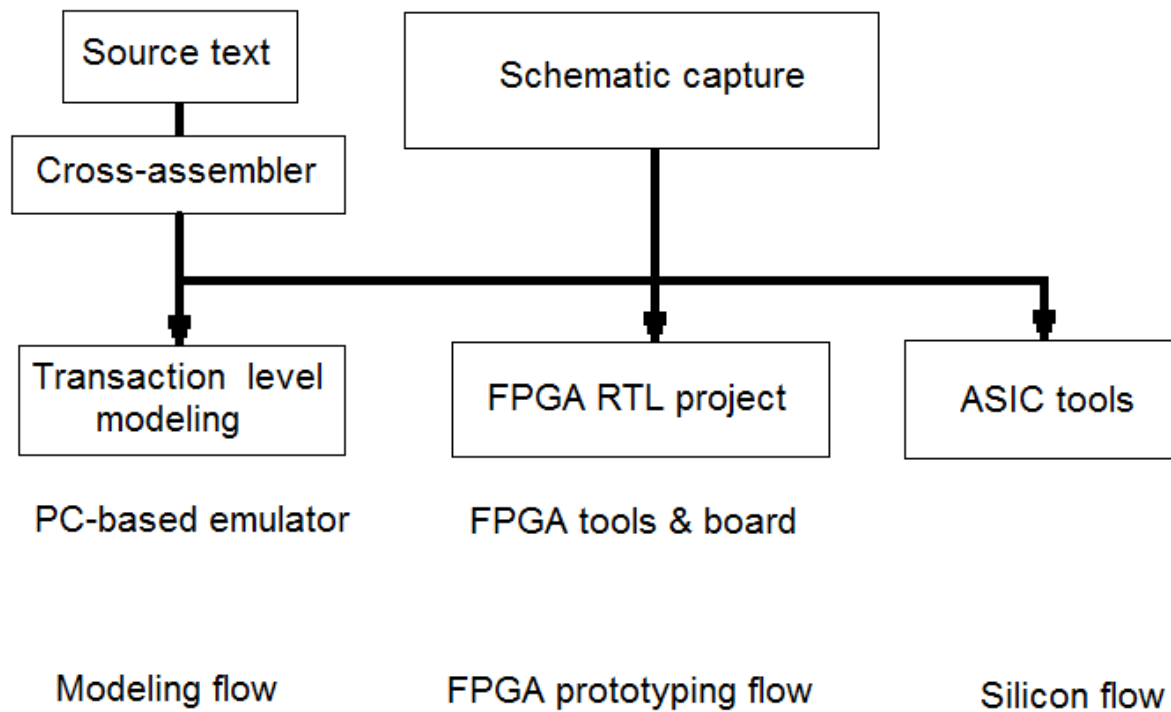


Fig. 2 System-on-chip design flow

As shown in fig.2, there are three main versions of SoC design flow, depending of the specialists involved. At early stages, when project requirements and specifications are still unprecise and need to be reviewed, the long-time FPGA prototyping and especially high-cost silicon implementation is unacceptable. Working with complex models at the hardware and software levels costs less, enables hundreds of iterations, when up to ten iterations of FPGA prototyping may be followed to validate a practical applicability of the developed device. Finally, ASIC implementation will challenge only specific technological, but not algorithmic or marketing issues. In the Quark CAD, all of the design-specific tasks run in Quark VM. There are some types of component representations, used in the design flow, including:

- Structural representation, required for building project graphics and, more important, creating a top-level structural VHDL file. This kind of file is based on a domain-specific Forth extension, which looks like a scripting file for the designer. For example, 'in data_a 32 bit', will create a bus with the name 'data_a' and the 32-bit width for the current component.
- Transaction level model (TLM) representation, written in Forth for each component. There are no domain-specific extension limitations, like for previous representation, so the component designer may use any Forth words he wants, and freely define all necessary words to properly represent the component model. Each model must provide the 'CLK' word, which is executed by the main modeling engine for each component in the system. Executable addresses of CLK words are collected when component models are loaded.
- Assembler representation, required for any type of processor component. There are no restrictions for assembler format, style, or limitations, so the developer may realize both Intel and AT&T formats, as well as a more compact postfix format. All processor cores, provided in the baseline configuration of Quark CAD, use the postfix format, written in pure Forth. No restrictions are present because only requirement to assembler is to create memory image for each processor component in the system. An external assembler program may be also used for

this purpose.

- RTL representation, written in VHDL, but combines to synthesizable project by configuration engine, running on Quark VM. Each component must have this type of representation for FPGA prototyping and silicon implementation; however, not for TLM. This allows programmers to write their own TLMs without knowledge of VHDL. After modeling is completed, the TLM representation may be used as a specification to for a corresponding RTL source file.
- GDSII representation required only for chip production.

Design tools are integrated in IDE, shown in Fig.3.

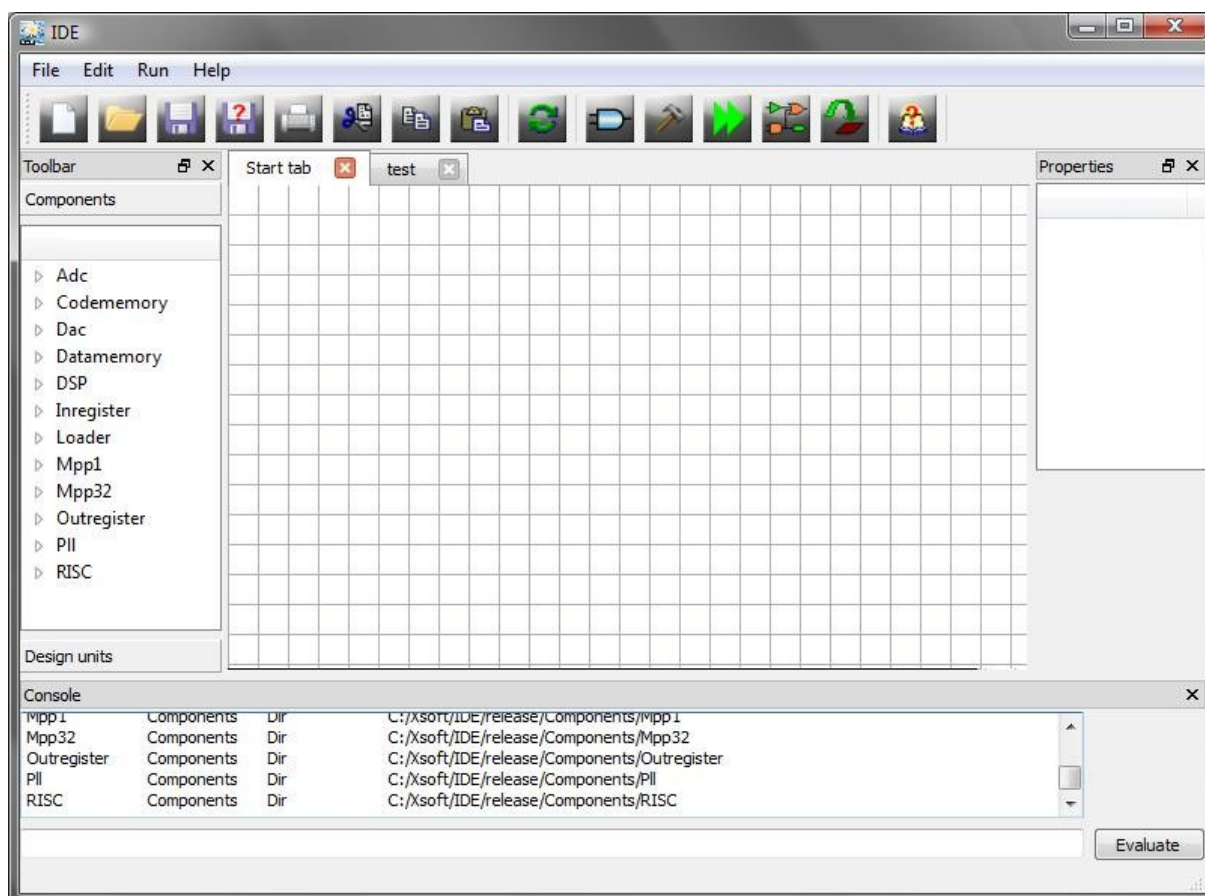


Fig. 3. The Quark CAD design tool

The main window, denoted as 'start tab', shows the content of the Quark VM virtual screen. Any other tabs are text editors from the Qt library, used for viewing and editing source texts and model reports. The 'Evaluate' button at the bottom of the window runs evaluation of the text, entered in the text field near it. The content of this text field is copied directly into the Forth terminal input buffer, when the VM interpreter runs.

Quark CAD stack processor cores

Under the contract conditions, several processor cores must be provided as part of a standard component library of the CAD tool. There are a general-purpose RISC processor, a DSP core, and a matrix of processor elements in this library. All processor cores may be connected to the synchronous system bus for sharing peripheral devices. Fixed priority arbitration is used in

case of more than one processor in the system. Stack operations and a minimal set of Forth words are supported by all cores being developed.

The RISC-processor, named QuarkR, is a 32-bit general-purpose processor. Its main features are listed below:

- Harvard architecture with a 3-stage pipeline
- 32 general-purpose registers
- 3-address and stack-based register file access, when the top of data stack is initially located in R31
- Independent return stack (32 cells deep), switchable to a stack in the external data memory
- 32-bit wide command
- Base Forth commands implemented in hardware as a command set extension (no mode switching is required)

DSP:

- Independently running MAC engines, 1 or 2 blocks in each DSP core. Each DSP block may work as one 32x32, two 32x16, or four 16x16 independent multipliers with corresponding accumulators
- Control processor unit, 32-bit stack core with 8-bit wide commands; command set is similar to that of QuarkR

Matrix processors have two modifications, one of them with a grid of 32-bit compact stack cores:

- 4 cells deep data stack, 4 cells deep return stack
- Harvard architecture with separate code and data memories for each core
- Up to 32x32 processor unit grid
- Local links, global column buses and system bus interface

Conclusions

From project description above, we can make several assumptions about usage style of integrated development systems based on Forth-engine. They have found preliminary acknowledgement while QuarkCAD was in service at developers laboratories.

- The combined interpreting&compiling nature of Forth makes very useful script-rich style of large, complex system assembling. In fact, C++ code has about nothing influence on the actually characteristic of CAD output, while all of results depends only on scripts being run. This allow a wide range of IP cores to be modeled.
- Possibility of just-in-time defining of additional commands allow deep integration of different CAD flow steps, including flexible and re-engineering interfaces to external tools. For example, a full path to VHDL code generation provided by one of Forth script, when another, added later, scripts are interacts with FPGA downloading tools.
- Working on the scripts, developers team found a little need of complex programming technologies, some of them was added to Forth last time. Nevertheless, Quark VM, which closer to F83 than to F94, provide enough capabilities to solve a hard enough task, such as system-level chip design. We relied less on the built-in possibilities of the Forth, than on added by our scripts. Thus, in the course of project performance even base possibilities of the Forth have allowed to achieve good results.

References

1. C. Rowen. Engineering the Complex SoC
2. The International Technology Roadmap for Semiconductors: 2007 <http://public.itrs.net>

Formulating Type Tagged Parse Trees as Forth Programs.

Dr Campbell Ritchie and Dr Bill Stoddart
Formal Methods and Programming Research Group
Teesside University, UK

August 24, 2009

Abstract

We describe a two pass compilation technique for converting infix expressions to postfix, in which a first pass produces a type tagged tree and a second pass provides type checking and generates Forth code.

The expression language we consider includes sets and sequences, ordered pairs, relations, functions, lambda expressions, strings and arithmetic expressions. The type theory we employ in our infix expression language is an extension of that of the B Method, which is based on the use of power set and product operations as type constructors.

The main interest as a Forth research topic is the ease with which the compiler can be implemented in Forth. The first pass compiler is a set of mutually recursive functions that produce a type tagged tree. The second pass is implemented by providing a Forth definition corresponding to operations found at the non-terminal nodes of the parse tree, and over which is distributed the responsibility of performing type checking and compilation of the final executable code. Due to the close correspondence between parse trees and postfix expressions, the parse tree can be identified with a Forth program and the execution of the second pass of the compiler with the execution of this program.

1 Introduction

We consider the translation to postfix (Forth) of an expression language whose terms include sets, ordered pairs, higher order functions, lambda expressions, strings and arithmetic expressions. We use strong typing on set expressions, so that all elements of a set must be of the same type. Set and sequence expressions can be nested. Some forms may be represented in multiple ways, e.g. small sets of integers can be represented as bitsets, as well as in a standard format. Functions may be represented as executable code, or as sets of ordered pairs with element lookup.

The translation to Forth ensures the source expression is correctly typed, and deals with any conversions required to cope with differing data representations.

This work is part of an attempt to create a reversible high level language with a formal semantics and with a compiler that targets a reversible version of Forth, the “Reversible Virtual Machine” or RVM.

We use a two pass compiler in which the first pass produces a type tagged parse tree. Since there is a close correspondence between parse trees and postfix expressions, we are able to think of the tree as a Forth program.

These ideas, in a basic form, were presented at last years Euro-Forth, but using an expression language limited to integer and floating point arithmetic expressions. When compiling the expression $1+2.5$, the output from the first pass is “ 1” INT “ 2.5” FLOAT +_, and this is a Forth program which produces the output from the second pass, and its type. INT and FLOAT are constants denoting types. The operation +_ expects four arguments, these being two expression strings and their types, which can be either INT or FLOAT. The operation +_ generates two stack outputs, the postfix expression, “1 FLOAT 2.5 F+” and its type, FLOAT. The implementation technique provides the possibility of operators which are polymorphic, providing different functions according to the type of their arguments.

Analogously to +_ we have operations -, *_ and so on. Each operation in the language has a corresponding operation definition used in the second pass compiler, whose name is formed by appending an underscore to the original operation name. This is used in place of the original name in the output generated by the first pass of the compiler, and is executed during the second pass of the compiler.

This convention is maintained in the present paper. However, we now have two problems that were not present in the previous discussion. Firstly, since strings are now a part of our expression language, we need to represent string expressions which contain string expressions. We do this by using the facility, provided by Unicode, of using opening and closing quotes, as in:

“ “jim” ↦ 1234, “fred” ↦ 2345 ”

A second problem is that as well as the atomic types INT and FLOAT we now have type expressions of arbitrary complexity. We represent these types by the strings that will denote them in the final Forth code.

The remainder of the paper is structured as follows. In Section 2 we discuss aspects of our expression language, including sets, sequences, ordered pairs, and types. In section 3 we discuss the formal syntax of the expression language, using the syntax definition to derive the functions used to implement the first pass of the compiler. In Section 4 we consider lexical analysis and the first pass of the compilation process that generates type tagged trees. In Section 5 we discuss the second pass, showing how type information is passed up the tree and how type checking and nested set and sequence structures are handled. In Section 6 we conclude and discuss future work.

2 Sets and Types

We write mathematical expressions in maths font and fragments of Forth code in teletype font. The set extension $\{1, 2, 3\}$ is written in RVM_Forth as INT { 1 , 2 , 3 , }. As is usual in Forth, we adopt a programming style in which each lexical item is a Forth operation. INT provides the type of the set elements. The operation { opens a new set construction. The commas within the set construction represent an operation that takes an element from the stack and compiles it into the current set, and the operation } closes the set construction and leaves a reference to the set on the stack.

The set $\{\{1, 2\}, \{4\}\}$ has elements which are sets of integers. It may be represented in RVM_Forth as

```
INT SET { INT { 1 , 2 , } , INT { 4 , } , }
```

The following set of string and integer pairs:

$\{\text{"Bill"} \mapsto 2673, \text{"Campbell"} \mapsto 2680, \text{"Frank"} \mapsto 2680\}$ may be rendered in Forth as:

```
STRING INT PAIR { " Bill" 2673 ↦ , " Campbell" 2680 ↦ ,  
" Frank" 2680 ↦ , }.
```

We refer to sets of pairs as “relations”. The set of left hand elements (in this case $\{\text{"Bill"}, \text{"Campbell"}, \text{"Frank"}\}$) is the relation’s domain, and the set of right hand elements is its range. We can apply relations as functions. If the relation just given is applied to the argument “Bill” the result will be 2673. If the relation is called R this would be represented in the infix expression language as $R(\text{"Bill"})$ and in Forth as “ Bill” R APPLY.

If the relation is inverted and applied to the value 2680 there is a choice of results: “Frank” or “Campbell”. Such choices may be made non-deterministically and be revised on backtracking.

Our types consist of basic sets, such as INT and STRING, together with the constructors SET and PAIR. In this paper we are only concerned with the postfix representation of types. If T is a type, T SET is the type whose elements are sets of elements of type T . If U is also a type, T U PAIR is the type whose elements are ordered pairs, with the first element of each pair belonging to T and the second element to U .

We use sets as a general way of representing data. As well as the set operations of union and intersection, we provide operations that are more specifically related to data updates and data queries. If R is a relation and U a relation of the same type, $R \oplus U$ is the relation R updated by entries from U . This expression is represented in Forth as R U OVERRIDE.

Type checking for an override operation consists of checking that both arguments are relations of the same type. An operation that requires a slightly more complex type analysis is “domain restriction”, denoted by \triangleleft . If R is a relation of elements between of type T and elements of type U , and if S is a set of elements of type T , then $S \triangleleft R$ is the relation from T to U consisting of the pairs in R whose first elements are in S . In terms of type checking with our postfix type language, we need to check that the postfix representation of the set S , which we wrote as S, is of type T SET for some postfix type T and that the postfix type of R is T U PAIR SET.

Sequences, in our canonical set representations, are just sets of ordered pairs where the domain elements run from 1 to n . The type of a sequence of elements from T is INT T PAIR SET. Not all data of this type are sequences of course. Since sequences are just sets, it is permissible to take the union or intersection of two sequences, though the result will only be a sequence under certain special conditions. As an example of where taking the union of two sequences can be useful, suppose we are given two sequences s and t and we want to test whether s is a prefix of t . A suitable test is $s \subseteq t \wedge (s \cup t) = t$.

3 Expression Grammar and Compilation Functions

Our grammar is written in Hehner's Bunch Theory. Appendix A describes this notation. Appendix B gives the full grammar. We provide sufficient comments in the text for a reader to follow our general approach without consulting the appendices.

The top level rule for our grammar is:

$$E = E \text{ "\(\rightarrow\)" } E_0, E_0$$

E is the bunch of strings in our infix expression language. Terminal symbols are shown in quotes. The maplet symbol \mapsto is an infix symbol which yields an ordered pair; it is the lowest precedence operator in the grammar, and is left associative. E_0 is the bunch of strings from our expression language which do not contain \mapsto at the top level. In parsing an expression we first look for the lowest precedence symbol, and since it is a left associative symbol we scan from the right to find the rightmost occurrence of such a symbol in the expression. Let PE be the function that takes a string which is a valid infix expressions, and returns the string that would be generated by the first pass of the compiler, then if e is any string from E and e_0 any string from E_0 we have the following properties:

$$\begin{aligned} PE(e \text{ "\(\rightarrow\)" } e_0) &= PE(e) PE(e_0) \text{ "\(\rightarrow_)" } \\ PE(e_0) &= PE_0(e_0) \end{aligned}$$

These cover the two cases, where the expression to be parsed contains a maplet symbol at the top level, and where it does not. Note that we compile the tagged maplet symbol $\mapsto_$, which will be used in the second pass to process type information and produce a maplet operator for the final code.

The next level of precedence contains the symbols $\setminus, \cup, \cap, \oplus$. Again these are left associative. The associated grammar rule is:

$$E_0 = S \text{ "\(\setminus)" } S_0, S \text{ "\(\cup)" } S_0, S \text{ "\(\cap)" } S_0, S \text{ "\(\oplus)" } S_0, E_1$$

Here, S is the bunch of all set expression, S_1 the bunch of strings from S without any of $\setminus, \cup, \cap, \oplus$ at the top level, and E_1 the bunch of expressions from E_0 without any of $\setminus, \cup, \cap, \oplus$ at the top level.

The rule tells us that any string from E_0 is either a string from S followed by one of $\setminus, \cup, \cap, \oplus$ followed by a string from S_0 , or else it is a string from E_1 .

Let PS , PS_0 and PE_1 be functions that parse strings from S , S_0 and E_1 respectively. Let op be one of \cup, \cap, \oplus , e_0 a string from E_0 , s a string from S , s_0 a string from S_0 , and e_1 a string from E_1 . Let $space$ be a sting containing just a space character. Then we can characterise PE_0 , the function to compile code for strings from the bunch E_0 , with the following equations.

$$\begin{aligned} PE_0(s \text{ } op \text{ } s_0) &= PS(s) PS_0(s_0) \text{ } space \text{ } op \text{ "_)" } \\ PE_0(e_1) &= PE_1(e_1) \end{aligned}$$

Moving to the next level of precedence we have the symbols for domain restriction and domain subtraction, which are of necessity right associative:

$s \triangleleft r$ has the same type as r and hence $s_1 \triangleleft s_2 \triangleleft r$ must parse as $s_1 \triangleleft (s_2 \triangleleft r)$. The grammar rule for this level is:

$$E_1 = S_2 \triangleleft S_1, S_2 \triangleleft -S_1, E_2$$

Now if e_1 is a string from E_1 , s_1 a string from S_1 , s_2 a string from S_2 , op a string from “ \triangleleft ”, “ $\triangleleft-$ ” and e_2 a string from E_2 we can characterise the compiling function PE_1 as follows:

$$\begin{aligned} PE_1(s_2 \text{ op } s_1) &= PS_2(s_2) PS_1(s_1) \text{ space op “}_-” \\ PE_1(e_2) &= PE_2(e_2) \end{aligned}$$

Other grammar rules involving binary operators lead to compiling functions in the same way.

Now let us see how a set extension is compiled. The grammatical form of a set extension is:

“{” L “}”

Where L is a list of expressions, with grammatical description:

$L = E, L \text{ “,” } E$

The set extension $\{1, 0, x + 1\}$ will compile to:

{_ “ 1” “ INT” ,_ “ 0” “ INT” ,_ “ x” “ INT” “ 1” “ INT” +_ ,_ }_

Let the function that performs parsing of set extensions be PSE and the function that parses a list be PL . Let $list$ be a string from L , and e a string from e . Then we can characterise PSE and PL with these equations.

$$\begin{aligned} PSE(\text{“ {” } list \text{ “}”}) &= \text{“ {” } } PL(list) \text{ “}” \\ PL(list, e) &= PL(list) \text{ “,” } PE(e) \\ PL(e) &= PE(e) \end{aligned}$$

4 Lexical Analysis and First Pass Compilation

Whereas classical lexical analysis reads and distinguishes tokens by reading left to right, we have a bidirectional lexical analyser which finds the lowest priority connective at each scan. This is implemented by two Forth functions. **RL-LEX** scans from right to left and is used to search for left associative connectives, and **LR-LEX** scans from left to right and is used to search for right associative connectives. These words take as input parameters a string address and a sequence of tokens to be searched for. Where tokens are multi-character and one token may be a prefix of another, the longer token is placed first in the sequence to prevent spurious detection of the shorter token.¹ If a token is found, the returned values are the part of the expression string that lies before the token, the part that lies after, and the token. If no token is found, the returned values are the expression string and two null values.

Whilst searching we only check for tokens at the “top level”. We are not at the top level if we are currently inside some kind of bracket structure. The four types of bracket defined in the expression language are precedence brackets (...), set brackets (...), sequence brackets [...], and matching string quotes “...”. Whilst performing a lexical scan, if a bracket is detected, we stop looking for the

¹Where a symbol at a lower order of precedence is a prefix of a symbol with a higher order of precedence there is a potential problem which we have not tried to solve.

given tokens and look instead for the relevant brackets until we return to the top level. For example if scanning right to left and a closing set bracket is found, we then look for opening and closing set brackets until the original bracket is matched; we ignore any sequence or structuring brackets, but we must change mode if a quote bracket is detected, since within quotes any set brackets that occur are part of a string literal rather than part of the expression's structure.

The first pass compiler has a separate Forth function for each syntactic category in the expression grammar. We obtain a collection of mutually recursive functions. We use a uniform naming convention in which, for example, strings from the syntactic category E are translated to postfix as described by the mathematical function PE , which is implemented by the Forth function `PE`. The top level function `PE`, which can parse any expression, is the last function to be defined, but is required by many of the functions that precede it, e.g by functions that handle bracketed expressions. To handle this situation RVM-Forth has a defining work `OP` used in these circumstances as:

```
NULL OP PE ( now we can refer to PE but not execute it)

.... (define all compiler functions)

: P ( define the functionality required of PE) ... ;
' P to PE ( assign the functionality of P to PE)
```

Within the definition of `P` we look for the rightmost maplet symbol at the top level:

```
: P ( az1 -- az2, parse an expression from E, leaving az2 the
  first pass postfix translation of the expression az1. )
  (: VALUE e :)
  e STRING [ "↦" , ] RL-LEX
  VALUE BEFORE VALUE AFTER VALUE OP-STRING
  OP-STRING NULL =
  IF ( e did not contain a ↦ symbol at the top level)
    BEFORE PEO
  ELSE
    BEFORE RECURSE AFTER PEO
    SPACE^ OP-STRING _^ AZ^
  THEN
1LEAVE ;
```

Within this definition, `AZ^` performs catenation of asciiz strings (the string form used in this project), `SPACE^` appends a space to a string, and `_^` appends an underscore.

As a second example we describe PE1, which looks for the right associative domain restriction and domain subtraction operations.

```

: PE1 ( az1 -- az2, parse an expression from E1, leaving az2,
the first pass postfix translation of the expression az1. )
(: VALUE e :)
e STRING [ "←, ◁" , ] LR-LEX
VALUE BEFORE VALUE AFTER VALUE OP-STRING
OP-STRING NULL =
IF ( e did not contain ← or ◁ at the top level)
  BEFORE PE2
ELSE
  BEFORE PS2 AFTER PS1
  SPACE^ OP-STRING _^ AZ^
THEN
1LEAVE ;

```

We see that these functions are very similar. They either fail to find any symbols at the current precedence level and fall through into a function which deals with higher precedence symbols, or they find a symbol, apply appropriate functions to parse the text before and after the symbol, and concatenate the results followed by the operator with an appended underscore.

Where parsing functions are more complex, it is because the particular symbol found reveals more about the before and after text. For example consider the grammar rule:

$$E_2 = S_1 \text{ "←" } E, W_1 \text{ "∧" } W_2, S_1 \text{ "▷" } S_2, S_1 \text{ "−▷" } S_2, S_1 \uparrow A, S_1 \downarrow A, E_3$$

In this case, after the string belonging to E_2 is split into before and after strings by RL-LEX, the functions to be applied to the before and after text depend on the symbol that has been found. For example if the symbol is "←" we are appending a value to a sequence. The text before the symbol is a set expression from S_1 , to be processed by PS1, and following text is an expression from E_3 , to be processed by PE3. If, however, the symbol found is the catenation symbol "∧", we are concatenating two sequences or two string expressions. In that case the before text is from W_1 and the after text is from W_2 , and these are to be processed by PW1 and PW2 respectively. Implementation of PE2 thus requires a case analysis based on the symbol found.

The lexical analysers have a wider use than detecting infix symbols. For example to parse a function application, which has syntactic form given by the equation:

$$F = S_2 \text{ "(" } L \text{ ")"}, F \text{ "(" } L \text{ ")"}$$

We start from the right of the expression, move back one character, and search for the token "(" . The before string is then the function expression and the after string the argument list. To parse the argument list with function PL we search right to left for a comma. If none is found the argument list is a single expression to be proceeded with PE, if a comma is found the following text is an expression to be processed with PE and the before text is again an argument list to be recursively processed by PL.

The first pass compiler collects type information from numeric literals and numeric strings and from identifiers, whose type is held in a symbol table (assumed to be already in existence). The result of a first pass compilation is a postfix expression containing a mixture of special operators and type tagged literals and identifiers. The special operators, which will run during the second pass, are named with the names of the corresponding operators in the original expression, with additional underscores: $\mapsto_ \sqcup \sqcap$ and so on. We could have re-used the existing names and written the definition of these second pass operations in a separate word list. However, the present choice of names serves to help the reader (and ourselves) to remember which pass of the compiler is currently being discussed.

5 Second Pass Type Checking

The intermediate postfix code produced by the first pass of the compiler may be viewed as a tree which has the literals and identifiers which appear at its leaves tagged with type information. As the intermediate code is executed, the types of more complex sub-expressions, such as set structures, are derived, along with the postfix code for these sub expressions.

We illustrate the technique by considering the compilation of some set extensions, where, in addition to deriving the types of the sets concerned, the second pass compiler must perform type checks to ensure that all sets are homogeneous in the sense that any set may only contain elements of one particular type.

A simple example would be compilation of 1,3,5. The first pass produces the following text:

```
{_ " 1" " INT" ,_ " 2" " INT" ,_ " 3" " INT" }_
```

In the following trace we see how this is translated into two strings, representing a Forth set expression and the type of the set. In the trace, strings which are on the stack are represented by the form they take in the Forth source code, and the distinction is made by whether they occur in the "Forth Code" column, or the "Stack" column. NULL, a constant with value zero representing lack of information about the types of a set element, is shown in the same way.

Forth Code	Stack
	Empty
{_	" {" NULL
" 1" " INT"	" {" NULL " 1" " INT"
, _	" { 1 , " " INT"
" 3" " INT"	" { 1 , " " INT" " 3" " INT"
, _	" { 1 , 3 , " " INT"
" 5" " INT"	" { 1 , 3 , " " INT" " 5" " INT"
}_	" INT { 1 , 3 , 5 , }" " INT SET"

The `{_` word places an initial set expression string (consisting of just an open set brace) onto the stack followed by a NULL, signifying lack of information about the type of the set's elements.

The word `,_` takes four string arguments, a partial set expression, a representation of the type of the set elements (or NULL if the type is not yet known), an expression giving the current element, and an expression giving its type. It

checks whether the type of the new element is the same as that of previous elements, and extends the partial set expression to include the new element.

Finally, `}_` takes the same four string arguments as `,_`. It checks the type of the final element, extends the set expression to include this element and adds the closing brace, then prepends the type of the set element, returning the resulting string as its first return value. Its second return value is the type of the set, formed by appending “ SET” to the element type.

The use of the stack during the second pass provides for this approach to cope seamlessly with nested set structures.

We now look at an example of a relation in which some elements are strings. We consider the expression:

```
{ “ joe” ↦ 90, “ Methuselah” ↦ 900 }
```

The first pass compilation yields

```
{_ “ “ joe”” “ 90” ↦_ ,_ “ “ Methuselah”” “ 900” ↦_ }
```

and here we need nested quotes; we are building a calculus of partial expression strings, and in such a partial expression string we now have a string literal. After the second pass we obtain two stack items, the expression string: “ STRING INT PAIR { “ joe” 90 ↦ , “ Methuselah” 900 ↦ , }” and its type, which is “ STRING INT PAIR SET”.

The new second pass operation introduced by this example is `↦_`. This expects four string arguments: a first expression, its type, a second expression, and its type. It produces two results: a new expression combining the previous expressions as a pair, and the type of the pair. It checks the types are equal, then concatenates the expressions and appends a maplet symbol to obtain the new expression. If T and U are the expression types, then the new expression has type T U PAIR.

The second pass compiler for operations handling set and sequence descriptions are shown below, omitting `[_` and `]_` which are very similar to `{_` and `}_`.

```
: {_ “ { ” NULL ( NULL represents the lack of type information at
the start of a set expression construction ) ;
```

```
: ,_ ( set-exp:$ type1:$ element-exp:$ type2:$ -- exp' type2 )
(: VALUE set-exp VALUE type1 VALUE element-exp VALUE type2 :)
type1 NULL <> IF
type1 type2 STRING= NOT
ABORT“ Non-homogeneous types in set”
THEN
set-exp element-exp AZ^ “ , ” AZ^ type2
2LEAVE ;
```

```
: \u21a6_ ( exp1:$ type1:$ exp2:$ type2:$ -- pair-exp:$ type:$ )
( \u21a6 is the maplet character )
(: VALUE exp1 VALUE type1 VALUE exp2 VALUE type2 :)
exp1 exp2 AZ^ “ \u21a6” AZ^ type1 type2 AZ^ “ PAIR” AZ^
2LEAVE ;
```

```
: }_ ( set-exp:$ type1:$ element-exp:$ type2$ -- set-exp' set-type )
```

```

(: VALUE set-exp VALUE type1 VALUE element-exp VALUE type2 :)
type1 NULL <> IF
  type1 type2 STRING= NOT
  ABORT“ mismatched type in final set element”
THEN
type2 SPACE^ set-exp AZ^ element-exp AZ^ “ , } ” AZ^
type2 “ SET” AZ^
2LEAVE ;

```

As a final example of type checking instances of the domain restriction operation, i.e expressions for the form $S \triangleleft R$. Here, S must be an expression that represents a set, and R an expression whose domain elements are of the same type as S . If the type of R (expressed in postfix) is `T U PAIR SET` then the type of S is `T SET`. The second pass operator \triangleleft must check that each type has the correct form, and that the type of the elements of S is equal to the type of the elements of the domain of R . Here arity checks, scanning right to left, provide the key to dismantling postfix type expressions into their component parts.

6 Conclusions and Future Work

We have described a two pass compilation technique applicable to fairly complex expressions. We believe the technique will extend to the compilation of full operations. The first pass of a compilation produces a parse tree, in which literal values and identifiers, which form the leaves of the parse tree, are tagged with type information. The operations at the nodes of the parse tree match the operations of the expression language, but perform type analysis and compilation functions. The first pass of the compiler is provided by a collection of mutually recursive functions, each of which is designed to compile inputs taken from a specific syntactic category, established in the grammar. The second pass of the compiler is provided by definitions corresponding to operations appearing at the non-leaf nodes of the parse tree.

A Bunch Notation and Grammars

Grammars are usually written in terms of “production rules”. We want to see a grammar as a set of simultaneous equations on strings. We want, also, to consider both non-terminal symbols, E , E_0 etc. and terminal symbols “ \cup ”, “ $+$ ” etc. to be collections of strings, with non terminal symbols generally denoting a plurality of possible strings.

Mathematical descriptions of grammars rely on set theory, but this has some disadvantages. Set theory provides two things simultaneously: collection and packaging: the set $\{1, 3, 5\}$ both collects together the elements 1,3 and 4, and packages them up as a new element. For grammar descriptions we want the power to collect, without being obliged to package, That is we want the ability to deal in plurality as well as elements.

In Eric Hehner’s Bunch theory, a bunch is the contents of a set. This 1, 3, 5 is the bunch that forms the contents of the set $\{1, 3, 5\}$ Collection and packaging become orthogonal concepts. The comma is now an operation rather than syntax. It signifies bunch union. Thus if A and B are bunches then A, B is a bunch consisting of the elements from A and the elements from B .

In our description of grammar, we are dealing with bunches of strings. The main operation is string catenation. We can write it as $s \hat{\ } t$ but we elide the catenation symbol and just write $s t$. Where catenation is applied to bunches of more than one element, it is lifted in an obvious way: if $X = \text{“John, Tom”}$ and $Y = \text{“Jones, Smith”}$ then $X Y = \text{“JohnJones”, “JohnSmith”, “TomJones”, “TomSmith”}$

Relating our use of bunch notation to classical grammar descriptions, comma can be thought of as choice, and juxtaposition as sequencing. Taking an example line from the grammar:

$$A = A \text{ “+” } A_0, A \text{ “-” } A_0, A_0$$

this tells us the bunch S is made up of strings from the three bunches $A \text{ “+” } A_0$, $A \text{ “-” } A_0$ and A_0 . The bunch $A \text{ “+” } A_0$, consists of strings made up of a string from A followed by “ $+$ ” followed by a string from A_0 , and so on.

B Expression Language Syntax

Symbols listed in order of precedence, low precedence symbols first, symbols of equal precedence are enclosed in brackets.

$$\mapsto (\backslash \cup \cap \oplus) (\triangleleft \triangleleft-) (\leftarrow \hat{\ } \triangleright \triangleright- \uparrow \downarrow) (+ -) (* /) \sim$$

Most binary connectives are left associative, the exceptions being \triangleleft and $\triangleleft-$ which are right associative.

Non terminal symbols for the grammar.

L a comma separated list of expressions E an expression λ a lambda expression S an expression representing a set W an expression representing a string or a set F an expression representing a function application A an arithmetic expression N numeric literal $\$$ string literal I an identifier

E_0 expressions from E without \mapsto at the top level.

E_1 expressions from E_0 without $\backslash \cup \cap \oplus$

E_2 expressions from E_1 without $\triangleleft \triangleleft-$

E_3 expressions from E_2 without $\leftarrow \hat{\ } \triangleright \triangleright- \uparrow \downarrow$

E_4 expressions from E without $+ -$

E_5 expressions from E_4 without $* /$

E_6 expressions from E_5 without \sim (unary minus)

S_0 expressions from S without $\setminus \cup \cap \oplus$

S_1 expressions from S_0 without $\triangleleft \triangleleft-$

S_2 expressions from S_1 without $\leftarrow \frown \triangleright \triangleright- \uparrow \downarrow$

W_0 expressions from W without $\setminus \cup \cap \oplus$

W_1 expressions from W_0 without $\triangleleft \triangleleft-$

W_2 expressions from W_1 without $\leftarrow \frown \triangleright \triangleright- \uparrow \downarrow$

A_0 expressions from A without $+ -$

A_1 expressions from A_0 without $* /$

A_2 expressions from A_1 without \sim (unary minus)

B.1 Expression grammar equations

$$\begin{aligned}
E &= E \text{ "}\mapsto\text{" } E_0, E_0 \\
E_0 &= S \text{ "}\setminus\text{" } S_0, S \text{ "}\cup\text{" } S_0, S \text{ "}\cap\text{" } S_0, S \text{ "}\oplus\text{" } S_0, E_1 \\
E_1 &= S_2 \text{ "}\triangleleft\text{" } S_1, S_2 \text{ "}\triangleleft-\text{" } S_1, E_2 \\
E_2 &= S_1 \text{ "}\leftarrow\text{" } E, W_1 \text{ "}\frown\text{" } W_2, S_1 \text{ "}\triangleright\text{" } S_2, S_1 \text{ "}\triangleright-\text{" } S_2, S_1 \text{ "}\uparrow\text{" } A, S_1 \text{ "}\downarrow\text{" } A, E_3 \\
E_3 &= A \text{ "}\text{+}\text{" } A_0, A \text{ "}\text{-}\text{" } A_0, E_4 \\
E_4 &= A_0 \text{ "}\text{*}\text{" } A_1, A_0 \text{ "}\text{/}\text{" } A_1, E_5 \\
E_5 &= \text{ "}\sim\text{" } A_1, E_6 \\
E_6 &= N, \$, I, F \text{ "}\text{(}\text{" } L \text{ "}\text{)"}, \text{ "}\text{'}\text{" } L \text{ "}\text{'}\text{"}, \text{ "}\text{[}\text{" } L \text{ "}\text{]}\text{"}, \text{ "}\text{(}\text{" } E \text{ "}\text{)"}, \text{ "}\text{(}\lambda \text{ "}\text{I}\text{" } E \text{ "}\text{)" }
\end{aligned}$$

$$\begin{aligned}
S &= S \text{ "}\setminus\text{" } S_0, S \text{ "}\cup\text{" } S_0, S \text{ "}\cap\text{" } S_0, S \text{ "}\oplus\text{" } S_0, S_0 \\
S_0 &= S_1 \text{ "}\triangleleft\text{" } S_0, S_1 \text{ "}\triangleleft-\text{" } S_0, S_1 \\
S_1 &= S_1 \text{ "}\leftarrow\text{" } E, S_1 \text{ "}\frown\text{" } S_2, S_1 \text{ "}\triangleright\text{" } S_2, S_1 \text{ "}\triangleright-\text{" } S_2, S_1 \text{ "}\uparrow\text{" } A, S_1 \text{ "}\downarrow\text{" } A, S_2 \\
S_2 &= I, F, \text{ "}\text{'}\text{" } L \text{ "}\text{'}\text{"}, \text{ "}\text{[}\text{" } L \text{ "}\text{]}\text{"}, \text{ "}\text{(}\text{" } S \text{ "}\text{)"}, \lambda
\end{aligned}$$

$$\begin{aligned}
W &= S \text{ "}\setminus\text{" } S_0, S \text{ "}\cup\text{" } S_0, S \text{ "}\cap\text{" } S_0, S \text{ "}\oplus\text{" } S_0, W_0 \\
W_0 &= S_1 \text{ "}\triangleleft\text{" } S_0, S_1 \text{ "}\triangleleft-\text{" } S_0, W_1 \\
W_1 &= S_1 \text{ "}\leftarrow\text{" } E, W_1 \text{ "}\frown\text{" } W_2, S_1 \text{ "}\triangleright\text{" } S_2, S_1 \text{ "}\triangleright-\text{" } S_2, S_1 \text{ "}\uparrow\text{" } A, S_1 \text{ "}\downarrow\text{" } A, W_2 \\
W_2 &= I, F, \text{ "}\text{'}\text{" } L \text{ "}\text{'}\text{"}, \$, \text{ "}\text{(}\text{" } W \text{ "}\text{)"}, \lambda
\end{aligned}$$

$$\begin{aligned}
A &= A \text{ "}\text{+}\text{" } A_0, A \text{ "}\text{-}\text{" } A_0, A_0 \\
A_0 &= A_0 \text{ "}\text{*}\text{" } A_1, A_0 \text{ "}\text{/}\text{" } A_1, A_1 \\
A_1 &= \text{ "}\sim\text{" } A_1, A_2 \\
A_1 &= \text{ "}\sim\text{" } A_1, A_2 \\
A_2 &= I, F, N, \text{ "}\text{(}\text{" } A \text{ "}\text{)" }
\end{aligned}$$

$$\begin{aligned}
\lambda &= \text{ "}\lambda\text{" } I \text{ "}\bullet\text{" } E \\
L &= E, L \text{ "}\text{'}\text{" } E \\
F &= S_2 \text{ "}\text{(}\text{" } L \text{ "}\text{)"}, F \text{ "}\text{(}\text{" } L \text{ "}\text{)" }
\end{aligned}$$

A Look at Gforth Performance

M. Anton Ertl*
TU Wien

Abstract

Gforth used to be an traditional threaded-code system. In the last decade we integrated a number of performance features into Gforth. Several of them were evaluated individually, but an evaluation with a more global perspective has been missing until now. This paper fills this void: We have measured the performance of Gforth releases from 0.5.0 to 0.7.0, on a wide variety of machines, and employing a wide variety of GCC versions for compiling Gforth. We present that data and give explanations for the performance differences.

1 Introduction

Up until and including gforth-0.5.0, Gforth employed quite traditional implementation techniques: Indirect threaded code or, on some architectures, direct threaded code.

Then we added a number of performance-improving techniques, which were released with Gforth 0.6 and Gforth 0.7: Primitive-centric hybrid direct/indirect threaded code [Ert02] was mainly an enabler for further optimizations. Dynamic superinstructions with replication [RS96, PR98, EG03b, EG03a] probably have the most significant effect on performance; these were all present in Gforth 0.6. Static superinstructions were added in Gforth 0.6.2, and static stack caching [EG04, EG05] in Gforth 0.7.0.

Moreover, Gforth-0.7.0 includes a number of changes to make these and other optimizations (in particular, explicit register allocation) more effective: Automatic build tuning, workarounds for GCC bugs, and some architecture-specific improvements.

In this paper, we take an overall look at these changes and their performance effects on various architectures.

Unfortunately, during the same time GCC was also “optimized”, and that often resulted in significantly lower performance for Gforth. We found workarounds for some of these problems, but the question remains how effective they are across GCC

versions and architectures. So in this paper we also look at how Gforth performs when compiled with various GCC versions on various architectures.

2 Setup

2.1 Gforths

We compare four versions of Gforth, with an additional three variants produced by running these versions with an option that turns off a new feature. The Gforth versions and variants we looked at were:

0.5.0 Uses traditional indirect or direct-threaded code. Direct-threaded code is only supported on some architectures, indirect threaded code on all of them.

0.6.1 no dynamic This variant uses primitive-centric hybrid direct/indirect threaded code. It’s still threaded code, but now colon definitions are compiled into a `call` primitive followed by an address, variables are compiled to `lit` followed by the address, etc. I.e., all threaded-code pointers point to primitives. Dynamic superinstructions with replication are disabled in this version (by running Gforth with `--no-dynamic`) in order to make it as close in performance to 0.5.0 as is easily possible, and to allow isolating the effect of that optimization.

0.6.1 This variant enables dynamic superinstructions with replication [RS96, PR98, EG03b, EG03a] on platforms where they are available. This feature works as follows: for a sequence of code without branches, the native code of the primitives is copied to a new place, and these native code fragments are concatenated. The direct threaded code points to these copies of the native code, not the originals. Most of the NEXTs are left away. Only when there is a branch, `call` or `execute` in the threaded code, a NEXT is needed. This feature reduces the number of NEXTs executed and increases the indirect branch prediction accuracy of the remaining NEXTs.

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

Architecture	CPU	Clock rate	
Alpha	21264B	800MHz	8MB L2
AMD64	Opteron 270	2000MHz	1MB L2, like Athlon 64 X2
	Xeon 5450	3000MHz	2 × 6MB L2, like Core 2 Quad
ARM	Xscale IOP 80321	600MHz	
IA32	Pentium 4 (Northwood)	2267MHz	512KB L2
	Athlon MP	2000MHz	512KB L2, like Athlon XP
	Opteron 270	2000MHz	1MB L2, like Athlon 64 X2
	Xeon 5450	3000MHz	2 × 6MB L2, like Core 2 Quad
IA64	Itanium II	900MHz	
PPC	PPC7447A (G4)	1066MHz	512KB L2
	PPC970 (G5)	2000MHz	
PPC64	PPC970 (G5)	2000MHz	

Figure 1: Machines

0.6.2 no superinst This variant has the same performance features as 0.6.1. Static superinstructions, the new performance feature of 0.6.2, are disabled.

0.6.2 This version adds static superinstructions, a platform-independent feature. Static superinstructions essentially combine a sequence of primitives into one primitive. Unlike dynamic superinstructions, which are created at Gforth run-time, static superinstructions are created beforehand and built into the Gforth engine. Gforth 0.6.2 uses 27 and 0.7.0 uses 13 static superinstructions.

0.7.0 simple stack caching This version tests if the explicit register allocation option works, and uses it if it works. Explicit register allocation tells GCC what registers to use for various VM registers (stack pointers etc.). Otherwise GCC often allocates the VM registers in memory, so explicit register allocation can provide a significant speedup on some architectures. Gforth 0.7.0 also contains several other performance improvements that are often somewhat specialized: E.g., it supports indirect branch target alignment for dynamically generated code, providing a speedup on Alpha; there are also performance improvements in mixed-precision division. And a number of architectures have better support in 0.7.0, allowing them to employ dynamic superinstructions.

0.7.0 This variant adds multi-state static stack caching: instead of keeping the number of stack items in registers the same (usually one item in the top-of-stack register) all of the time, the number of stack items in registers can vary to minimize the number of loads from and stores to the stack memory, as well as stack pointer

updates. Most architectures have too few registers available in a way usable with GCC and therefore can use only at most one register. On the PPC and PPC64 architectures we use up to three registers.

All versions of Gforth were compiled without enabling non-default performance features (such as explicit register allocation on versions before Gforth 0.7.0). That is the way that Linux distributors compile Gforth (and most Linux users get Gforth through their distribution rather than building it themselves). On the other hand, most Windows users probably use the binary package built by Bernd Paysan, and that uses non-default build options (in particular `--enable-force-reg` for explicit register allocation) to improve performance. So, the presented results are not representative for typical Windows installations.

A few other features that are not related to performance and are not used for the benchmarks (e.g., the C library interface) were disabled in order to help make the resulting binaries portable. We compiled the four Gforth variants once for each architecture and GCC version, and then ran the resulting binaries on all machines of that architecture.

2.2 Hardware and OS

Figure 1 shows the hardware we used. Several machines were able to run binaries for two architectures. All of these machines were running under various versions of Linux, on various versions of the Debian distribution. All machines had enough RAM to run the benchmarks without swapping.

2.3 Benchmarks

Figure 2 shows the benchmarks we use. These are all application benchmarks of significant size, and

Program	Author	Description
bench-gc 1.0	Anton Ertl	Garbage Collector
brainless 0.0.2	David Kuehling	Chess
cd16sim v11	Brad Eckert	CPU emulator
fcp 1.31-64	Ian Osgood	Chess
lexex	Gerry Jackson	Scanner Generator

Figure 2: Benchmark programs used

hopefully their usage patterns are more representative of other CPU-intensive applications than some of the smaller benchmarks that are often used (and that have quite different behaviour from these and other application benchmarks).

Each benchmark was run three times (on each combination of Gforth variant, GCC version, and machine), and the median of the three results was used further on.

In a few graphs we show results for individual benchmarks, but in most graphs we show an aggregate of all benchmarks. We use the geometric mean for aggregation (with each benchmark having the same weight) [FW86].

Brainless produces different results on 32-bit and 64-bit systems, and probably would produce different run-times even on a system that was always equally fast in 32-bit and 64-bit mode. Therefore we did not include brainless in the aggregate if we compare 32-bit and 64-bit systems.

2.4 GCC versions

We tried to compile Gforth with as many GCC versions as possible. Fortunately, there is a wide variety of GCC versions available on Debian, and they can be installed simultaneously. In addition, there were some manually installed GCCs available on some architectures.

2.5 Graphs

All graphs are scaled such that the highest-performing system gets speed 1. Also, all graphs are scaled logarithmically.

For graphs where each data point represents a Gforth variant with no reference to a specific compiler, the fastest-performing variant out of those that ran is shown. This should show what the various versions of Gforth are capable of when not hindered by GCC performance bugs.

In some graphs data points are missing, either because building that version of Gforth did not work, or because one of the benchmarks failed (for all of the Gforth compilations under consideration).

If a missing data point lies between two others in a line graph, the line is drawn from the point before to the point after, which is incorrect: It suggests that the performance of the missing point is in

the middle, but actually there was no performance at all for that point; however, trying to make these cases more visible would probably add more confusion than it would help, so we decided against it.

If a missing point is at the start or the end of the line, it is just not shown. In some cases, there is only one point in the line, which is then not shown. Instead you see the label of the “line” to the right of where the point is.

3 Results and Analysis

3.1 Overall performance

Figure 3 shows a performance summary: Each line represents an architecture/machine combination. The points on each line show the performance of different Gforth versions/variants, for each the fastest `gforth-fast` binary that the different compiler versions produced.

Overall, we can see that Gforth performance has improved significantly between 0.5.0 and 0.7.0, e.g., by a factor of more than 3 for IA32 Xeon 5450, and that factor seems pretty typical.

Another overall observation we can make is that we managed to build all Gforth versions on all machines, even on architectures that were not available to us for testing when we released the old versions of Gforth (like ARM or PPC64), or that were not even released when Gforth 0.5.0 was released in 2000, like IA64 (released in 2001) and AMD64 (2003). This shows that Gforth achieves its goal of portability very well.

3.2 Gforth versions

Looking closer, the effect of different changes is different for different architectures:

From 0.5.0 to 0.6.1nd, the threaded code model changed from classical direct or indirect threaded code to primitive-centric direct threaded code. In addition, on IA32 the top-of-stack is no longer kept in a register (without explicit register allocation); registers are scarce on IA32, and without explicit register allocation GCC then spills the stack pointer to memory, causing a significant slowdown compared to not keeping the top-of-stack in a register.

On the IA32 CPUs, switching to primitive-centric direct-threaded code buys a speedup, because it eliminates the cache consistency problems these CPUs have with classical direct threaded code (where code fragments are close to data) [Ert02, Section 3], and which shows up in some of these benchmarks, especially `cd16sim`. Interestingly, the AMD64 versions of Gforth 0.5.0 outperform the IA32 versions on the same machine, even though the AMD64 versions have no architecture-specific

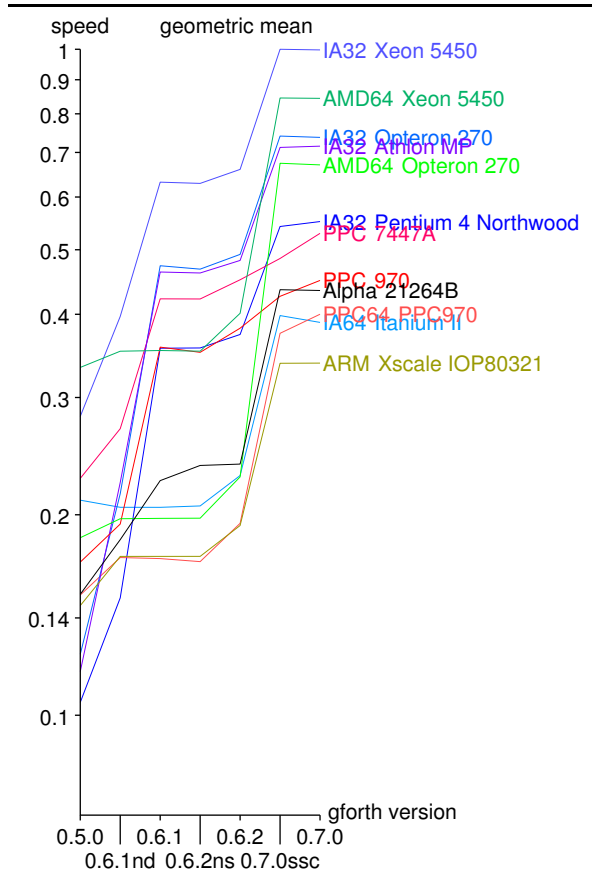


Figure 3: Performance per cycle, geometric mean of benchmarks (without brainless) of the best-compiled versions on all machines

tuning at all. Classical direct threading showed a benefit on the small benchmarks we usually use during development, but obviously these small benchmarks are not representative of large application benchmarks.

Most other machines also show an improvement from going to primitive-centric direct threaded code, because they usually used indirect threaded code in Gforth 0.5.0, and direct-threaded code is faster on most architectures.

From 0.6.1nd to 0.6.1: This enables dynamic superinstructions with replication on several architectures (Alpha, IA32, PPC), and gives large speedups on these machines. On architectures that we did not have available for testing when releasing 0.6.1 (AMD64, ARM, IA64, PPC64), this feature is not supported (it requires architecture-specific code for maintaining cache consistency) and therefore there is no change between 0.6.1nd and 0.6.1 on these architectures.

From 0.6.1 to 0.6.2ns There are no new performance features, so performance should be the same between these variants, and it generally is; we have no good explanation for the speedup on the Alpha 21264B machine.

From 0.6.2ns to 0.6.2 27 static superinstructions were enabled. They buy a small speedup even on systems where dynamic superinstructions work, because the native code for a static superinstruction is optimized compared to the equivalent dynamic superinstructions, which just consists of a concatenation of the code of its parts. Static superinstructions buy a larger speedup on systems where dynamic superinstructions are not supported, because there the static superinstructions also buy a part of the benefit that the dynamic superinstructions give otherwise: fewer NEXTs and better branch prediction. Looking at the individual benchmarks, static superinstructions help most of the benchmarks, but lex is not affected.

From 0.6.2 to 0.7.0ssc there are a number of new performance features, with different effects on different architectures:

Several architectures (AMD64, ARM, IA64, PPC64) became available for testing, and now Gforth supports *dynamic superinstructions with replication* on them; note how AMD64 and PPC64 now catch up to the performance of IA32 and PPC on the same machines.

Automatic tuning: The build script automatically tests whether Gforth works when built with explicit register allocation and/or a C type for double-cell integers, and enables these features if they work (i.e. in the usual case). Explicit register allocation gives significant speedups on IA32 and AMD64.

Branch target alignment inserts padding in the native code such that the targets of branches are aligned to cache line boundaries. This provides a significant speedup on the Alpha; this feature is also implemented for IA32 and AMD64 (but with padding limited to 1 byte), but we have seen little effect there (we also tried more padding).

We also added *workarounds for GCC performance bugs*, resulting in more GCC versions having good performance. This does not show up much in these graphs, which show only the binary from the best-performing GCC, but it is responsible for much of the speedup on PPC: For Gforth 0.6.2, the best-performing GCC for PPC was 2.95, and it performs similarly for Gforth 0.7.0, but there gcc-4.3 performs a little better.

We have also implemented *faster mixed-precision division*, but we do not think that this shows up in these benchmarks.

From 0.7.0ssc to 0.7.0, multiple-state static stack caching is enabled. Unfortunately, on most architectures GCC cannot use more than one register for this purpose; so in addition to always keeping one stack item in a register, Gforth 0.7.0 can now also keep no stack item in a register, and switch between these two states to minimize the work needed. In theory this improves the performance for se-

quences like ! 5, but as we can see, for most architectures (except PPC and PPC64) there is no speedup in application benchmarks.

On PPC and PPC64, GCC can use enough registers for keeping up to 8 stack items in registers, and up to 3 registers are useful [EG05], and that's what Gforth 0.7.0 uses on these architectures; static stack caching provides a speedup then. We suspect that there are also enough registers usable on IA64 and SPARC, but have not tested this.

3.3 Architectures and machines

We can also look at Fig. 3 to compare architectures and machines.

If you look for the best-performing system for running Gforth, the Xeon 5450 performs best per cycle among the machines we tested. In addition, it also has the highest clock rate, so it has the best absolute performance.

Another interesting question is whether to use 64-bit (AMD64, PPC64) or 32-bit (IA32, PPC) binaries of Gforth if you do not need 64-bit cells. In theory there is a speed advantage on AMD64 over IA32, because AMD64 has more registers available; unfortunately GCC makes no productive use of these registers when compiling Gforth; performance disadvantages of the 64-bit versions are the doubled memory requirement for all cells, including the threaded code, resulting in more cache misses; also, on the Xeon 5450 (and Core 2, but not on Opteron/Athlon 64), decoding is a little slower in 64-bit mode. On PPC64, there is no register advantage and no decoding slowdown.

Looking at the results, the 32-bit versions beat the 64-bit versions. There are some differences between the benchmarks here: `cd16sim` and `fcf` show the same performance in both architectures on the Opteron, but on Xeon the 32-bit architecture is a little faster (probably due to the decoding slowdown). For `benchgc` and `lexex`, the slowdown of the 64-bit version is significant (more than a factor of 1.2). This may be caused by the benchmarks doing something differently depending on cell size. E.g., for `benchgc` the cell size may change when and how often garbage collection is called. Or it could be a result of more cache misses.

For the PPC970, there is a slowdown in the 64-bit version even for `cd16sim` and `fcf`. One reason for that could be that we had fewer GCC versions available for PPC64 than for PPC; however, `gcc-4.1` performed well for PPC and was available for PPC64, so we are not very confident that this explanation is correct. Unfortunately, we don't have any other explanation.

Another remarkable thing is how close the performance of the IA32 Opteron is to the IA32 Athlon MP; this confirms that the K8 (Opteron,

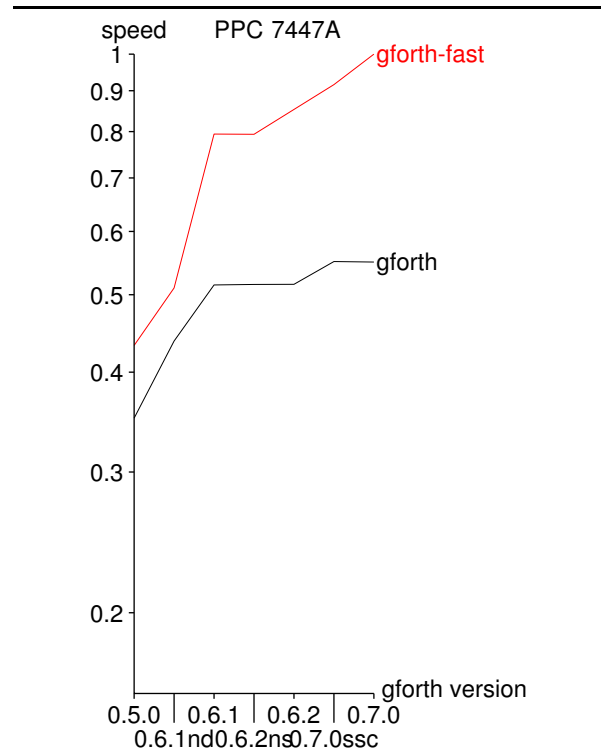


Figure 4: Benchmarking vs. debugging engine

Athlon 64) is really mostly a 64-bit variant of the K7 (Athlon MP, XP).

Another interesting result is that all IA32 and AMD64 machines beat all the others in performance per cycle in Gforth 0.7.0; even the Pentium 4, which has a well-deserved reputation for raising the clock rate at the cost of lower performance per cycle beats all the other architectures.

This is probably due to the indirect branch predictors of these CPUs rather than the architecture itself; and these branch predictors benefit from dynamic superinstructions with replication. Even though dynamic superinstructions reduce the number of executed NEXTs (and thus the number of executed indirect branches) by a factor of more than 3, there are still a lot of indirect branches executed, and they cost a lot unless correctly predicted.

You can see this effect especially well by looking at the PPC7447A line and comparing it to the IA32 lines. In Gforth 0.5.0 and 0.6.1nd, it is the runner-up machine (after the Xeon) in performance-per-cycle, but with the enabling of dynamic superinstruction and replication, it is passed by the Opteron and Athlon MP, and the Pentium 4 also comes close. Finally, it is passed by the Pentium 4 with the enabling of explicit register allocation in Gforth 0.7.0 (PPC has enough registers that GCC performs good register allocation even without explicit register allocation).

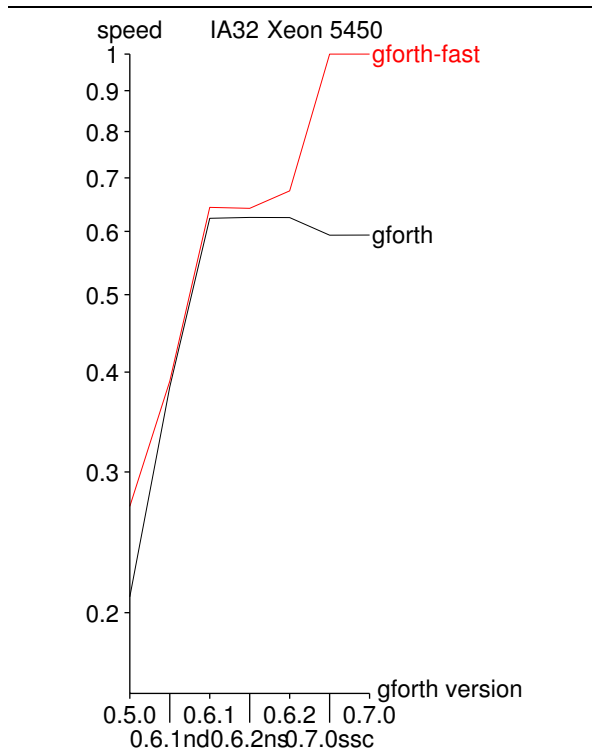


Figure 5: Benchmarking vs. debugging engine

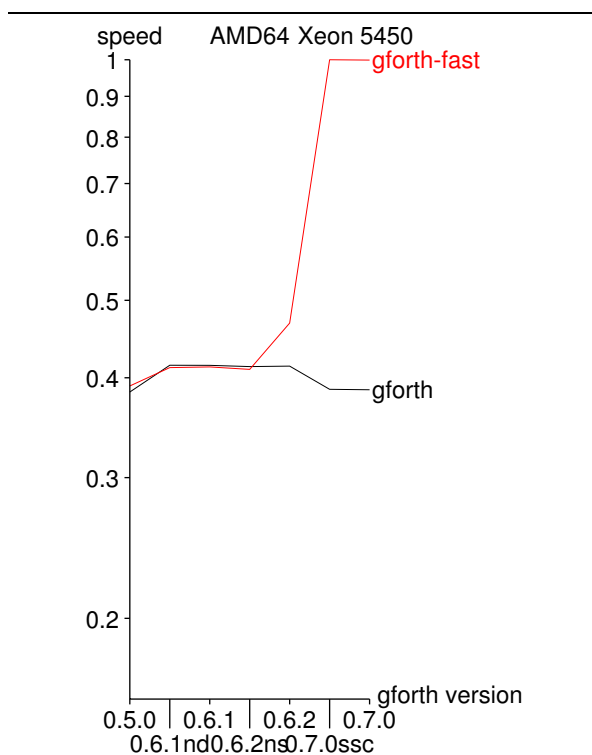


Figure 6: Benchmarking vs. debugging engine

3.4 Gforth-fast vs. gforth

Gforth comes with two engines: the debugging engine `gforth` and the benchmarking engine `gforth-fast`. The debugging engine performs some actions that cost performance, and it disables various performance features to allow better error reporting. How much does this cost, and has it changed over time, and why?

Figure 4 shows the graph for the PPC7447A. Already in Gforth 0.5.0, the debugging engine is slower, because it maintains a copy of the IP and RP virtual machine registers in memory (to allow better error reporting on invalid memory accesses etc.).

Both benefit to a similar amount from switching from indirect-threaded code to primitive-centric direct threaded code in 0.6.1nd; there is also a change in the way that IP is maintained that has no obviously visible effect on the PPC7447A, and is therefore explained later for a machine where the effect is visible.

Gforth-fast benefits a little more from dynamic superinstructions with replication in 0.6.1, probably because before it stalled longer waiting for the branches to resolve (whereas gforth was still busy maintaining IP and RP). There is no change in 0.6.2ns, as expected.

In 0.6.2, gforth-fast gains static superinstructions and a corresponding speedup, whereas the debugging engine does not enable static superinstructions in order to be able to report at which primitive an exception occurred.

Both engines benefit from improvements in 0.7.0ssc (for this machine probably from GCC performance bug workarounds). On this machine gforth-fast profits from the more sophisticated stack caching in 0.7.0, whereas this stack caching is disabled in the debugging engine to support better reporting of stack underflows.

While the graphs for most other machines can be explained in a similar way, there are a few interesting deviations:

Figure 5 shows the graph of the IA32 Xeon. For gforth-0.5.0, IP is maintained in memory by using a global variable for it, which requires loading it at every access. Starting from gforth 0.6, IP is kept in a register, but is stored to memory on every instruction boundary. This eliminates the loads and also guarantees that the in-memory IP always points to a primitive. Apparently the stores alone are very cheap¹, resulting in performance for the debugging engine from 0.6.1nd to 0.6.2ns that is very close to the performance of gforth-fast. On other IA32 machines the performance of the debugging engine is actually slightly higher for these versions, but we

¹Loads alone are also relatively cheap, but round trips through memory are usually expensive.

have no explanation for that.

Gforth 0.7.0 does not automatically tune the debugging engine to use explicit register allocation (to make building Gforth more robust and faster), so in the step from 0.6.2 to 0.7.0ssc we see the speedup from explicit register allocation in gforth-fast, but no speedup in gforth.

The slowdown for the debugging engine from 0.6.2 to 0.7.0ssc is due to workarounds for GCC performance bugs. These workarounds do have a cost; they pay for themselves on many compiler versions, but on the ones that don't need them they still cost.

Figure 6 shows the graph of the AMD64 Xeon. Unlike IA32, we have no classical direct threading with its cache consistency problems and also no spilling of SP, so the performance changes very little from 0.5.0 to 0.6.1nd. In addition, GCC manages to avoid loading IP from memory in 0.5.0 (resulting in code like for 0.6.1nd).

Dynamic superinstructions with replication are disabled in Gforth 0.6 on AMD64, so we see no speedup from that, and a flat line for the debugging engine until 0.6.2. In 0.7.0ssc one would expect dynamic superinstructions with replication to take effect, and they do for gforth-fast, but not for the debugging engine. The reason is that the debugging engine accesses a global variable (the saved IP) in every primitive, and on AMD64 global variables are referenced in a PC-relative way. This makes each primitive non-relocatable, effectively disabling dynamic superinstructions with replication for the debugging engine on AMD64.

3.5 GCC versions

All the graphs until now only showed the performance with the best-performing GCC version. Here we look at how well the different gforth-fast versions perform on different GCC versions on a few different architectures.

Figure 7 shows the graph for the PPC7447A. Gforth 0.5.0 and 0.6.1nd do not perform any optimizations that are broken by newer GCC versions, so their lines are relatively flat. Gforth 0.6.1–0.6.2 gain performance by using dynamic superinstructions with replication and work around GCC performance bugs up to gcc-3.3, but gcc-3.4 (released in 2004, i.e., after Gforth 0.6.2) and later introduced new performance bugs that disable dynamic superinstructions in these versions. Gforth-0.7.0 works around these performance bugs successfully, but in doing so apparently falls pray to a gcc-3.2 performance bug that disables dynamic superinstructions with replication. The GCC version that works best across all Gforth versions is gcc-2.95.

Figure 8 shows the graph for the IA32 Xeon. Again, gcc-2.95 shows the best performance across the board, and is the only compiler that builds

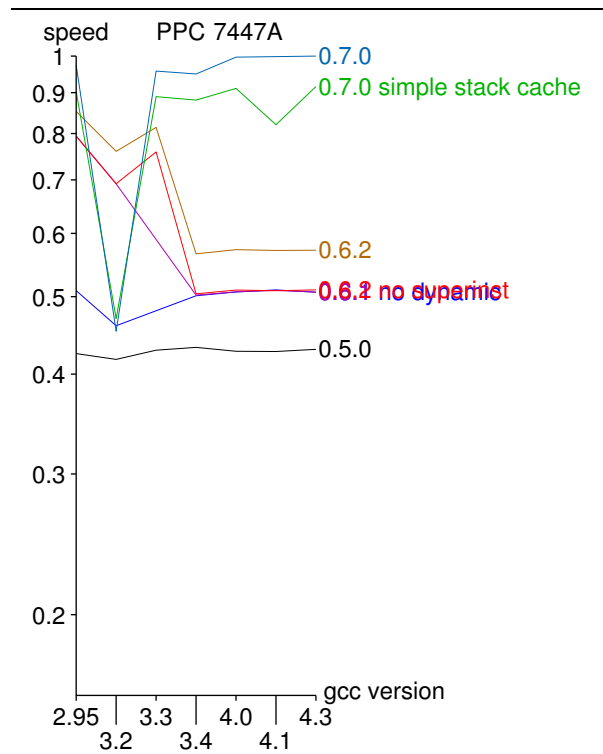


Figure 7: Gforth versions on different GCC versions

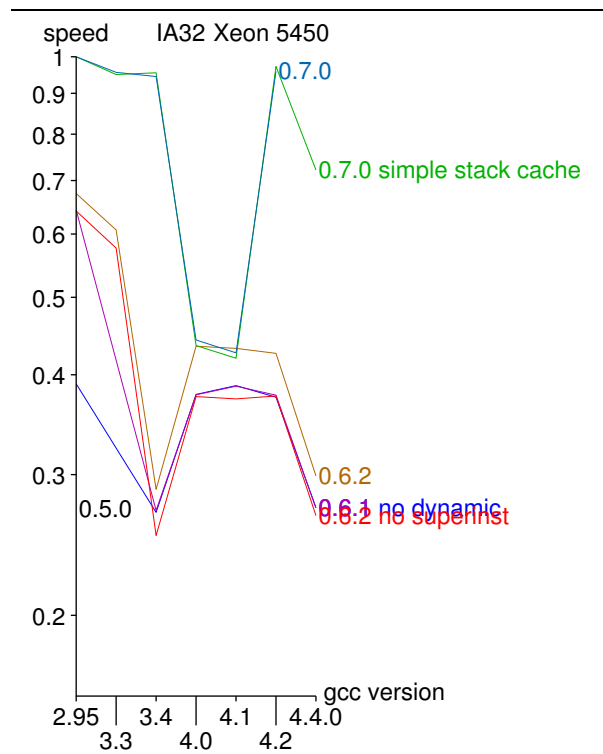


Figure 8: Gforth versions on different GCC versions

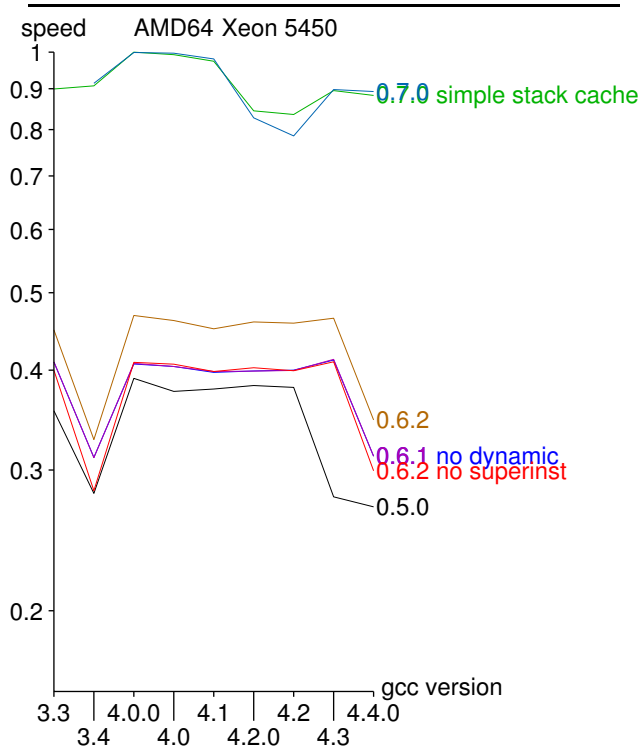


Figure 9: Gforth versions on different GCC versions

Gforth 0.5.0. Gcc-3.3 gratuitously changed the code order, breaking gforth-0.6.1 as a result. We worked around this problem in 0.6.2.

The workarounds in Gforth-0.6.2 for GCC performance bugs work up to gcc-3.3 and then fail. Gcc-3.4 is particularly bad in sharing one indirect branch for all the NEXTs, completely disabling the branch predictor of the CPU (GCC PR15242); that bug also causes the slowdown of 0.6.1nd on gcc-3.4. Gcc-4.0–4.2 fixed this bug, restoring at least a part of the performance, but the PR15242 problem is back in gcc-4.4.0, giving us bad performance again.

Gforth 0.7.0 successfully works around the performance bugs having to do with code ordering and indirect branches in $\text{GCC} \leq 4.3$, but gcc-4.0 and 4.1 spill important virtual machine registers, hurting performance. In addition to resurrecting PR15242, gcc-4.4.0 (released after Gforth-0.7.0) features a new (or worsening) performance bug that makes NEXT longer and slower, resulting in the slowdown shown in the graph. This performance bug uncovered a bug in the implementation of static stack caching in Gforth 0.7.0 (and that bug is responsible for there being no result for 0.7.0 with static stack caching and gcc-4.4.0).

Figure 9 shows the graph for the AMD64 Xeon. Unfortunately, gcc-2.95 is not available for AMD64. Gforth $\leq 0.6.2$ does not use dynamic superinstructions with replication on AMD64 anyway, so the lines for these Gforth versions run mostly in parallel, reflecting the presence of PR15242 in gcc-3.4

and 4.4.0, and their absence in the other version, with one exception: gcc-4.3 exhibits the PR15242 problem for gforth-0.5.0, but not for gforth-0.6.x.

Gforth-0.7.0 successfully works around the GCC bugs that disable dynamic superinstructions with replication. The cause for the performance variations between the gcc-4.x versions seems to be a performance bug that makes NEXT longer (and slower) in varying amounts between these versions.

4 Future work

This work uncovered some performance issues (in particular the unnecessarily long NEXT) that we plan to work around.

In addition, there are some performance ideas that we plan implement, in particular inlining [GE04].

Finally, this performance evaluation should be enhanced by comparing Gforth with other Forth systems. One challenge here is finding a large enough set of application benchmarks that run on all Forth systems.

5 Related work

Instead of working around GCC bugs as we do, one could also fix GCC. Prokopski and Verbrugge [PV08] propose a good method for letting GCC preserve the order of basic blocks and similar assumptions that are helpful for implementing code-copying optimizations like dynamic superinstructions. They don't just disable or restrict optimizations; they record the basic block order at the start and then restore it at the end (if possible), or report an error (if not).

6 Conclusion

The performance of default-compiled Gforth has improved a lot between Gforth 0.5.0 (2000) and 0.7.0 (2008), typically by a factor of 3.

The most significant factor for that performance improvement is the introduction of dynamic superinstructions with replication. While that was relatively easy to implement as a prototype, making it work on a wide range of architectures and GCC versions is a larger effort: First, it requires a small amount of architecture-specific code; more significantly, new GCC versions often break this feature, requiring programming workarounds for these performance bugs. So while this feature was introduced in Gforth 0.6.x, in many practical cases (e.g., various Debian packages) it was disabled in these versions. Gforth 0.7.0 includes a lot of work to make this feature more widely available.

There are also many other performance features, but they often only have a small effect (e.g., static superinstructions) or only on one or a few architectures (e.g., automatic tuning to enable explicit register allocation, which helps a lot on IA32). The combined effect of all these optimizations is quite significant, though.

Another interesting result is that Gforth has proven to be very portable, with even the very old Gforth 0.5.0 running on architectures and being compiled with compilers that did not exist when it was released.

References

- [EG03a] M. Anton Ertl and David Gregg. Implementation issues for superinstructions in Gforth. In *EuroForth 2003 Conference Proceedings*, 2003.
- [EG03b] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.
- [EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004.
- [EG05] M. Anton Ertl and David Gregg. Stack caching in Forth. In *21st EuroForth Conference*, pages 6–15, 2005.
- [Ert02] M. Anton Ertl. Threaded code variations and optimizations (extended version). In *Forth-Tagung 2002*, Garmisch-Partenkirchen, 2002.
- [FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.
- [GE04] David Gregg and M. Anton Ertl. Inlining in Gforth: Early experiences. In *EuroForth 2004 Conference Proceedings*, pages 33–40, 2004.
- [PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.
- [PV08] Gregory B. Prokopski and Clark Verbrugge. Compiler-guaranteed safety in code-copying virtual machines. In *Compiler Construction (CC'08)*, pages 163–177. Springer LNCS 4959, 2008.
- [RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996.

Porting Forth Applications and Libraries

Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England
t: +44 (0)23 8631 441
f: +44 (0)23 8033 9691
e: sfp@mpeforth.com
w: www.mpeforth.com

Abstract

Porting Forth applications between host Forth systems is less difficult than many people assume. It just requires discipline, management, preparation and a minimal ego. This paper discusses how to perform a successful port with reference to several ports I have been involved with. Some guidelines for both producers and consumers are suggested.

Introduction

Successful code tends to have a long lifetime and to evolve over time.

The MPE Forth cross compiler has evolved from a code base originally supplied in 1982. Although hardly a line of code from the original sources has survived unchanged, the code is a direct descendant of the original.

The Candy Construction Project Modelling and Project Control software from Construction Computer Software (CCS) in Cape Town is approaching one million lines of Forth source code dating from the early 1980s. Over that period it has been ported between two CPU architectures (M68000, i386) and four operating systems (HP98xx, DOS16, DOS32, Windows) as well as several Forth hosts from several suppliers.

Hanno Schwalm recently ported his fJACK audio interface to VFX Forth. The code base now runs on iForth and VFX Forth under both Windows, Linux and OSX where the host supports it.

Bernd Paysan is currently porting his Minos/Theseus GUI design tools from the original BigForth host to VFX Forth for Linux.

Brad Eckert provided the FAT file system used with the MPE Forth cross compilers.

The Forth Scientific Library (FSL) has been ported to many different Forth systems and was probably the first example of a significant Forth source code library being widely ported. Its success is in no small measure due to the assumption that library code **should** be ported. The FSL uses the same harness approach as is discussed here.

The issues involved in porting Forth source code are very little different from those in porting source code in any other language. My observation is that standards have helped a great deal in making Forth source code more portable.

Why do the port?

The motivations for porting Forth code depend heavily on whether the code is an application or a library.

Maintainers of applications usually see themselves as tool **users** rather than tool **makers**. Although this line is blurred in large applications, many companies producing applications see the compiler as outside their core competency. Why should a manufacturer of construction planning software or mass spectrometers write Forth compilers? Using a third-party compiler enables them to take advantage of improvements introduced for all users rather than just those required by the application. In turn, if their current Forth system does not support a particular operating system, e.g. Intel OSX, the authors can turn to a Forth host that does support that operating system.

Source code libraries gain by widespread use. You don't reinvent the wheel, you use existing code. You don't waste time learning the details of a web protocol, an audio interface or a GUI, you use existing tools and get your job done faster. Successful libraries influence standards and make other peoples lives easier. Successful use of libraries enables you to do more with your time.

Why not do the port?

The main reason for not doing a port, either from one system or to another, is that you need your code to be smaller or faster. In the desktop world, size is no argument for any Forth application I know of, and both compilers and PC hardware improve over time. The speed/space reason is often advanced in the embedded systems world as it would force a hardware change. In many instances this is a fallacious argument.

The majority of embedded Forth applications are produced in volumes of less than 10,000 units per year. Changing from a 8/16 bit CPU with 60kb of Flash and 4kb of RAM to a much faster 32 bit CPU with 512kb Flash and 64kb of RAM and vastly more peripherals will cost in the range of 1 to 2 dollars/EU/pounds, and greatly extend the lifetime and potential features of the product. The additional hardware cost is easily saved in reduced software development costs.

MPE has had several clients who have stayed with what they know, only to come back five years later saying that they now need to change and that the five years have been very expensive.

When should I port?

Most people port code to another platform when they need to make a step change in the capability of the application.

Programmers of desktop applications may need to move to a new platform or use a feature or library that is unique to a particular Forth host. Embedded systems developers make the change when they run out of memory on an 8/16 bit system or need facilities such as USB, file systems or TCP/IP stacks. Don't even **think** about bank switching – it will cost you a fortune!

Whatever the reasoning, the decision to port must be considered and the porting process managed.

Process

The successful application ports that I have been involved with have all followed a similar process. Library ports to a new host follow the same basic process.

- 1) Preparation – eliminate host specific code and/or move it to a host-specific harness.
- 2) Line in the sand – draw a line in the code. What's below it can be changed between hosts, what's above it cannot be changed.
- 3) Dual build – compile the same code base on the old and the new hosts and retest on both hosts
- 4) Decision time – can you add your new features to the both hosts, or must you abandon one? If you are moving from DOS to Linux or Windows, your objective may well be to abandon DOS.
- 5) New features – only at this stage should you introduce new features.

I have observed an application of 800,000 lines of Forth source code ported in six months using this process.

Preparation

If you rely on host-specific features, they will break your code later. To avoid this, move all such code to a host-specific harness file or directory of files. The harness for your existing host should be fully **documented in the source code**. The harness will become the model for the new hosts.

Code that causes problems includes standard words that have host-specific extensions, e.g. some Forths use range checks in **/STRING** whereas many do not. It is far better to rename this version to something else. A global search/replace on your source tree is much cheaper than days spent chasing bugs. Other nightmares come from words that are common, but have different meanings and semantics on different hosts, e.g. **FOR** and **NEXT**.

Many Forth systems use a vectored I/O model for redirecting **KEY**, **EMIT** and friends to different devices or displays. You will need to find a way to isolate the differences from your application code.

Other sources of error will come from words that effectively split execution into two words, but do not use **:** and **;** to do it. The ANS standard does not permit words to define other words inside themselves and some compilers take advantage of this. Such words will need host-specific hooks into the compiler.

Remove all coded definitions, rewrite them in high level for the harness model. You can always rewrite them later if you have to. Forth assemblers for the same host CPU are very rarely compatible, so remove problems before they occur.

This phase essentially forces you to perform a code review of your source tree. Do not be surprised if you find and correct existing bugs!

Line in the sand

The objective is find a place in your load order above which no code needs to be changed to make the application work on the new host. You will not get right immediately, but is important that the setting of the line is managed and that programmers buy into the idea. There must be a manager of the process.

There's always a big temptation to put conditional compilation into the code above the line. This only indicates that that some piece of code should be changed and the host-specific parts moved into the harness layer. The porting process is a matter of constant negotiation between the partners.

Dual build

The point of the process is to be able to test that the new version runs identically to the one on the old host. You must be able to share data between them.

Do not add new features. This is only a port, you do not want to be debugging new features yet.

At the end of this stage you will have two harnesses – one for the old system, one for the new.

Decision time

If you are porting a library, you have now completed your first port. Your harnesses need to be reviewed. It is in your own interest to reduce the size of the harnesses where possible – it will make future ports easier.

When porting an application, you are probably not going to be able to make all users convert to the new system immediately. Therefore you need to make a decision as to whether to maintain the dual build for a while so that the old host can be extended too, or whether to put the application on the old host on “care and maintenance only”. This decision is essentially a commercial decision.

Where you are porting between (say) Windows and Linux, you will probably have already made, or be in a good position to make, decisions about how to manage the differences between GUIs. Embedded systems developers will be in a position to review speed, space and power budgets.

It is important to (try to) predict what evolution the code will make over the next few years. If you have moved from Windows to Linux, will you want to go to OSX as well? Now that you've moved from a 16 bit CPU to a 32 bit CPU, will you want a file system and a TCP/IP stack? What will be the consequences of these decisions on your harnesses, e.g. for vectored I/O.

New features

Now that you have made your decisions and reviewed the harness code, you can plan your new features.

Application developers who have chosen to abandon the old host will be very tempted to optimise the code base for the new host. Be **very** cautious. The process of building the harness has also contributed to layering the software, which has its own benefits. Where you can profitably take benefit is in removing code that has equivalents in the new host. In embedded systems, the appearance of vectored I/O in the new host may permit considerable simplification.

Library developers should seriously consider doing another port. The effect of another port is to consolidate the harness code. The result of this to make it much easier for third parties to do their own ports, which in turn increases the take-up of the code.

Guidelines

Harness documentation

Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing. Dick Brandon

Yes, this guideline comes first. The harnesses are the specification by example.

The documentation should be in the source code. Programmers don't use Word or OpenOffice, they use UltraEdit or Vi. Stack comments must be accurate, and every word should have at least a one-line description. Literate programming tools for Forth are available. It may take 10% longer to write the code, but you'll save more time during testing and debugging. Once you start documenting your code as just a part of programming, you will see the advantages and apply it to all code.

Put the design notes at the top of each section.

Keep it simple

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. (Brian Kernighan)

The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. (Edsger Dijkstra)

Simple, well factored code is the key to maintainable code. Guru code and clever tricks will bite you later. Identify these, re-factor them, and move the nasty bits to the harness layer. An example comes from the MPE assemblers, which can be switched between opcode-first (prefix) and opcode-last (postfix) modes. This is done using **?PREFIX** to separate the opcode construction code according to when it has to be executed. The original code is a Forth classic by Bob Smith from the 1987 FORML conference.

```
: prefix?      \ -- t/f ; true if in prefix mode
  <prefix> @   ;

2variable aprior

: ?prefix      \ struct -- struct' ; executes previous opcode
  prefix? if
    r>          \ struct ret-addr --
    aprior 2@ 2swap aprior 2! \ exchange with contents of APRIOR
    >r          \ will use previous return address
  endif
;


```

Opcode descriptions often take the form:

```
: opType      \ opcode -- ; --
  create , does> ?prefix ... ;


```

This code makes several assumptions that are dangerous:

- 1) The return address is a single cell on the top of the Forth return stack.
- 2) It performs a flow control which makes assumptions about the return stack depth.
- 3) What follows is effectively a nameless word and the compiler is not told about it.

A better solution that only makes the return stack assumption follows.

```

: (?prefix) \ struct -- struct' ; executes previous opcode
\ This word assumes that the return address is on the top
\ of the return stack and that an xt is inline. It exits
\ the caller.
r> @ \ -- struct xt
prefix? if
  aprior 2@ 2swap aprior 2! \ exchange with contents of APRIOR
endif
execute \ use previous xt
;

: ?prefix \ -- ; finish and start nameless word
\ This word assumes that the return address is on the top of the
\ return stack and that it can compile an xt inline.
postpone (?prefix) here 0 , \ (?PREFIX) gets xt inline
state off smudge
:noname swap ! !csp
; immediate

```

This code still makes the return stack assumption. This is safe on most hosted systems, which is acceptable but still host-dependent. However, it still makes assumptions about the compiler and data alignment. We can fairly easily reduce it to the return stack assumption only.

```

: (?prefix) \ struct -- struct' ; executes previous opcode
\ This word assumes that the return address is on the top
\ of the return stack and that an xt is inline. It exits
\ the caller.
r> aligned @ \ -- struct xt
prefix? if
  aprior 2@ 2swap aprior 2! \ exchange with contents of APRIOR
endif
execute \ use previous xt
;

: ?prefix \ -- ; finish and start nameless word
\ This word assumes that the return address is on the top of the
\ return stack and that it can compile an xt inline.
postpone (?prefix) align here 0 , \ (?PREFIX) gets xt inline
>r postpone ; :noname r> ! !csp
; immediate

```

What is also interesting about the changes is that the resulting code is more robust and makes assembler macros easier to handle.

Fix bugs first

Fixing a piece of code with two bugs in it is much more difficult than fixing one bug. So fix any bugs as soon as you detect them. Never, ever, leave a bug alone.

Crash early and crash often

When MPE wrote its first Windows Forth back in the days of Windows 3.1, we made many mistakes. A natural consequence was programmers wrote extremely defensive code. When we wrote VFX Forth for Windows, we didn't do that. VFX Forth is brutally intolerant of programming errors, but has good integration with the Windows exception handler. When CCS moved their application to VFX Forth, there were initially complaints that previously working code was crashing. After a month or two it emerged that VFX Forth was revealing bugs that had lurked in the production code for years. When these bugs were fixed, both systems had been improved.

The good thing about a crash is that it's a show-stopper – you have to fix it.

People

The trouble with C++ is that it requires gurus to maintain it. Gurus don't do maintenance. (Anon)

People are part of the design. It's dangerous to forget that. (Anon)

Never attribute to malice that which can be explained by stupidity. (Hanlon's Razor)

Stupidity maintained long enough is a form of malice. (Richard Bos's corollary)

A man who is right every time is not likely to do very much. (Francis Crick).

Porting an application or library is not a competition, it's a collaborative exercise. People skills are an important part of the process. Once your code is shared, you will have to deal with a wide range of people with a wide range of expectations.

Keeping your ego out of the way is just part of the process. None of us is capable of being correct all the time.

Conclusions

Porting a library or application is mainly a matter of discipline and management.

The harness approach is practical and proven over a number of ports.

People and their management are part of the solution.

Acknowledgements

Willem Botha at CCS taught me a great deal about porting code in a disciplined fashion.

Damian Brasher provided a perspective outside the Forth world.

Motivation

SWIG-GForth-Extension

Gerald Wodni

gerald.wodni@gee.at

M. Anton Ertl

anton@mips.complang.tuwien.ac.at

TU Wien

August 24, 2009

- GLForth: OpenGL & SDL where imported by hand
- Octal numbers could be transformed without changing BASE each time
- Nested headers: big projects could easily be converted

39

1 / 12

Outline

- 1 Introduction
Motivation
SWIG
- 2 C
Constants
Types
Functions
- 3 Implementation
Output-Types
Libraries
- 4 Example
- 5 Conclusion
- 6 References

2 / 12

SWIG [1]

3 / 12

- C/C++ Compiler with custom output
- Typically used for C-interfaces to other languages like PHP or Python, but also non-scripting languages like Java or Lua
- Generates C-source
- In our case also Forth-source

4 / 12

Constants

- Integers → "constant"
- Enums are treated as integer constants
- Floats → "fconstant"
- Strings → words (e.g. : TITLE s" SWIG-GForth-Interface" ;)

40

5 / 12

Types

- `off_t` could either be n or d
- As n fits into d , `-m32 gcc`-option is used to create 32/64bit independent code
- SWIG-typemaps used for mapping C-types into Forth
- All pointers are transformed into `a`
- Structs are currently omitted
- Default type if it can't be resolved

6 / 12

Functions

- stackcomments with original parameter names, (sometimes "n n d - n" is not meaningful)
- "forthify" function names (`get_nextItem` becomes `get-next-item`)

7 / 12

Output-Types

- FS
 - Header file is directly converted into a Forth-source
 - Easy to use but requires SWIG
 - Only necessary when an interface to a custom library is needed
 - Platform dependent
- FSI (Independent)
 - Generates C-code, constants are resolved using the compiler on target machine
 - Only needs to be generated once on any platform
 - C-compiler more likely to be installed than SWIG
 - Platform independent

8 / 12

Libraries

- OpenGL
 - First to compile without problems
- stdlibc
 - Already compiled to FSI-Files
 - Requires some human interaction
- others...
 - Creating a collection of FSI files, which could be downloaded and installed easily

41

9 / 12

Example

```
...
#ifdef _SIZEOF_PTHREAD_BARRIERATTR_T
    printf( "%d\constant _SIZEOF_PTHREAD_BARRIERATTR_T\n", (
        long) _SIZEOF_PTHREAD_BARRIERATTR_T );
#endif
#ifdef _ALLOCA_H
    printf( "%d\constant _ALLOCA_H\n", (long) _ALLOCA_H );
#endif
/* <functions> */
printf( "\<functions >\n" );
printf( "\t( -- )\n" );
printf( " c-function\<type-get-mb-cur-max\
    t-ctype-get-mb-cur-max\t\n" );
printf( "\t( --nptr -- )\n" );
printf( " c-function\<tof\<ta -- r\n" );
printf( "\t( --nptr -- )\n" );
printf( " c-function\<toi\<ta -- n\n" );
printf( "\t( --nptr -- )\n" );
printf( " c-function\<tol\<ta -- n\n" );
...

```

10 / 12

Conclusion

- SWIG suitable for outputting other languages than C
- Configuration files could be adopted to other Forth C-interfaces
- FSI collection

11 / 12

References

- 1 SWIG Manual <http://www.swig.org/Doc1.3/Extending.html>. 1995-2008.
- 2 Neal Crook, Anton Ertl, David Kuehling, Bernd Paysan, Jens Wilke. GForth-Manual. 1995-2008.

12 / 12

Motivation

Wrongforth - A reversible Forth

Gerald Wodni
gerald.wodni@gee.at
TU Wien

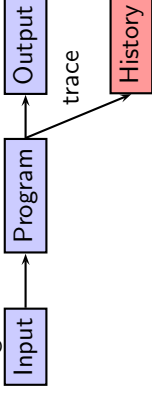
August 24, 2009


- Robert Glück's course - Program Inversion and Reversible Computation [1][2]
- Porting the idea of Janus to another language
- Forth is close to machine level, reversible hardware?

Outline

- 1 Introduction
Motivation
Types of Inversion
- 2 Related Work
RVM-FORTH
Janus
- 3 Forth
Implementation Techniques
Operators
Control Structures
Fibonacci
- 4 Conclusion
- 5 References

Types of Inversion

- Tracing*


```
graph LR; Input[Input] --> Program[Program]; Program --> Output[Output]; Program -- trace --> History[History];
```
- History grows proportionally to length of computation
- Forward programming as usual
- Backtracking, input can only be restored with history
- Stand-Alone†


```
graph LR; Input[Input] --> Program[Program]; Program --> Output[Output];
```
- No History → also applicable in low memory machines
- For-/Backward programming
- Algorithm is inverted:
input can be generated from any output (e.g. writing zip and implicitly getting unzip)

* Bennet:73

† Dijkstra:78, Gries:81

- Guards & Choices
- Reversible primitives (e.g. !_ C!_ +!_)
- Backtracking

```

procedure fib
  if n = 0 then
    x1 += 1
    x2 += 1
  else
    n -= 1
    call fib
    x1 += x2
    x1 <=> x2
  fi x1 = x2

```

Non-destructive updates Swap operator
 Condition-assertion pair



- No destructive assignments (:=)
- Only operators with a reversible counterpart (+ -, * /)
- Each condition has a matching assertion
- Call/Uncall

- Dual Compilation
 - Primitives are compiled with for- & backward semantics
 - Twice the size in memory
- Reversible Computation
 - Primitives switch their semantics based on execution direction
 - Small memory footprint (forward + assertions)
 - No information loss

- Dual Compilation


```
: _+ iffwd POSTPONE + else POSTPONE - then ; immediate
: _- iffwd POSTPONE - else POSTPONE + then ; immediate
```
- Reversible Computation


```
: _1+ forwards? if 1+ else 1- then [resume] ;
: _1- forwards? if 1- else 1+ then [resume] ;
```

Control Structures

- Dual Compilation


```
: _if iffwd POSTPONE if
else POSTPONE drop POSTPONE then
then ; immediate

: _then iffwd POSTPONE drop POSTPONE then
else POSTPONE invert POSTPONE if
then ; immediate

: _else POSTPONE else ; immediate
```

- Reversible Computation


```
: _if POSTPONE if POSTPONE skip
POSTPONE exit POSTPONE exit ; immediate

: _else POSTPONE else POSTPONE skip POSTPONE begin
POSTPONE rskip POSTPONE skip
POSTPONE exit POSTPONE exit ; immediate

: _then
POSTPONE exit POSTPONE exit POSTPONE rskip
POSTPONE until
POSTPONE rskip
POSTPONE then
POSTPONE exit POSTPONE exit POSTPONE rskip
POSTPONE _0= ; immediate
```

Fibonacci

- Dual Compilation


```
\--- forward ---
1 direction !
: fib ( -- ) recursive
n @ 0= _if
x1 1 +=
x2 1 +=
_else
n 1 -=
fib
x1 x2 @ +=
x1 x2 <=>
_endif

\--- backward ---
0 direction !
: fib-1 ( -- ) recursive
x1 @ x2 @ = _then
x1 x2 <=>
x1 x2 @ +=
fib-1
n 1 -=
_endif

x1 @ x2 @ = _then
n @ 0= _if
;
;
```



```
: callfib fibptr [resume] ;
: fib ( n x1 x2 -- n x1 x2 )
beg
  bwdcheck third0= _if
    -1+
    -swap
    -1+
    -swap
    -else
    -rot
    -1-
    -rot
  callfib
  -swap
  -over+
  -then 2dup= fwdcheck
end ;
```

Conclusion

- Dual Compilation
 - As fast as common Forth
 - Easy to implement
 - Works in ANS-Forth
- Reversible Computation
 - Suitable for reversible hardware [2]
 - Literals backwards semantics
 - Manipulates the return-stack
 - Would benefit from changes in the Forth-VM (Direction Register, Paired Branches)
- Both
 - Could save implementation time: Encryption, Compression, ... only need to be implemented once

References

- 1 Tetsuo Yokoyama, Robert Glück. A Reversible Programming Language and its Invertible Self-Interpreter. 2007.
- 2 Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. Reversible Machine Code and Its Abstract Processor Architecture. 2007.
- 3 Bill Stoddart. RVM-FORTH, a Reversible Virtual Machine. 2004.
- 4 Neal Crook, Anton Ertl, David Kuehling, Bernd Paysan, Jens Wilke. GForth-Manual. 1995-2008.

Hardware/Software Co-Design MicroCore

Ulrich Hoffmann
FH Wedel

- What is **MicroCore**?
- architecture
- systems design
- hardware/software-co-design
- future work

What is **MicroCore**?

MicroCore

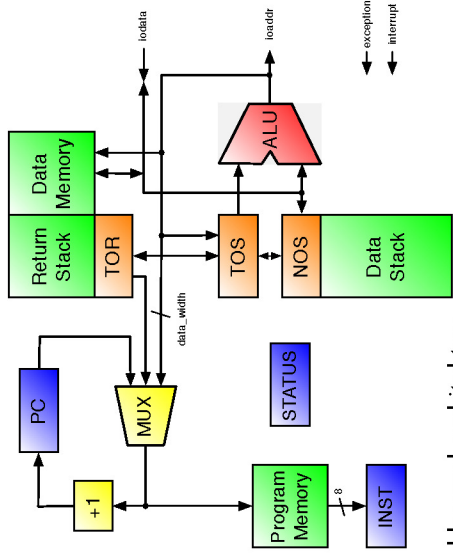
- Softcore-Mikroprozessor/Controller
- Open source in VHDL
- synthesizable for FPGAs (Actel, Altera, Lattice, Xilinx)
- stack based Harvard architecture
- tool chain:
 - Forth cross compiler
 - realtime kernel
 - interactive debugger
 - C compiler in development

What is **MicroCore** used for?

- **MicroCore** invented by Klaus Schleisiek, SEND Offshore GmbH, Hamburg
- Used in ocean bottom seismics
- embedded systems
- device/appliance programming
- made-to-measure hardware/software combination
- system understanding down to the bits
- independence from hardware vendors



MicroCore architecture



- Harvard architecture
- scalable with of data words, L1Terat instruction
- interrupts and exceptions

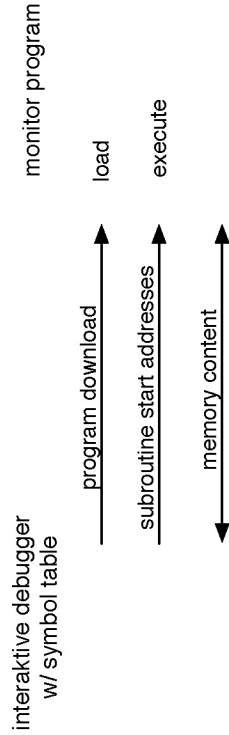
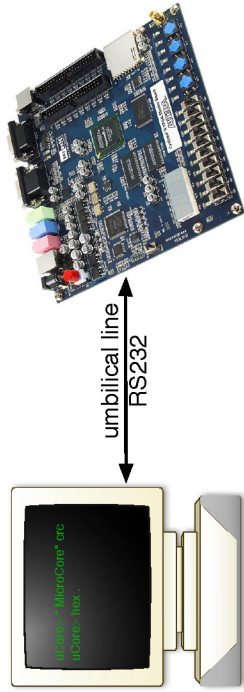
MicroCore instruction set

7	6	5	4	3	2	1	0
\$80	\$40	\$20	\$10	\$8	\$4	\$2	\$1
Lit/Op	Type	Stack	Group				

Code	Name	Action
00	BR	Branches, Calls and Returns
01	ALU	Binary and Unary Operators
10	MEM	Data-Memory and Register access
11	USR	Not used by core, free for user extensions>User instructions / immediate calls

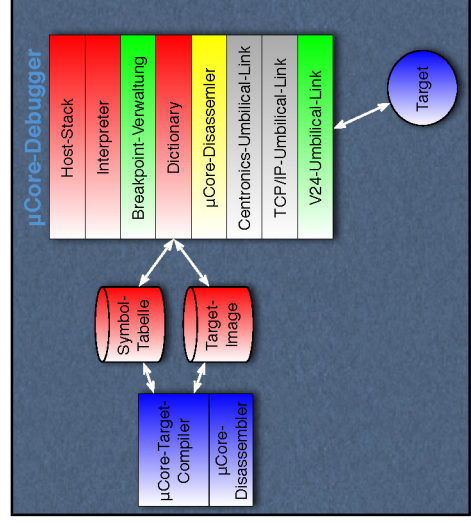
Stack	act	Operation	Forth operators / phrases
none	none	Complex math instructions, see below	SWAP
pop	Stack ->	NOS <op> TOS -> TOS	+ - AND OR XOR NIP
push	NOS <op>	TOS -> TOS -> Stack	2DUP + OVER
both	TOS <op>	TOS -> TOS	0 = * ROR ROL 2/ u2/
	Unary math instructions, see below		

interactive host/target program development



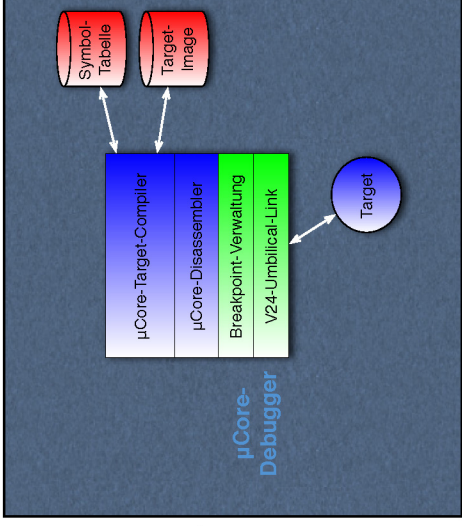
MicroCore Debugger

- old Debugger in C
- Linux dependent (termios)
- C99 w/ local Functions - not really portable
- much of what Forth already has
- much of what the target compiler already has
- no longer required functionality



MicroCore Debugger

- ✓ new Debugger in Forth
- ✓ Gforth - portable
- ✓ currently V24 für Windows, but Code für Linux and Mac in Gforth exists
- ✓ integrated environment
- ✓ incremental compilation



How does it look like?

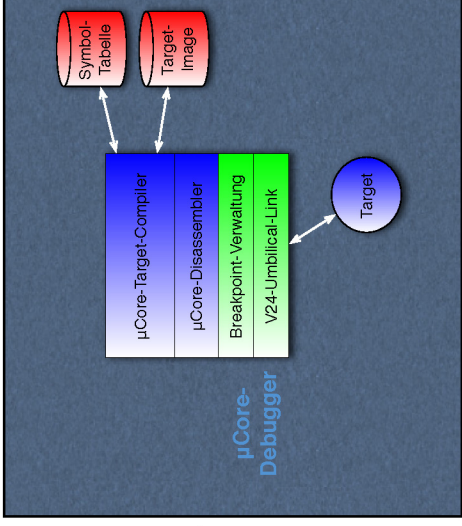
```

C:\Command Prompt - gforth_bsd_unicode.fs
Gforth 0.6.2. Copyright (C) 1996-2003 Free Software Foundation, Inc.
Type 'bye' to exit
ok
boot-image ok
handshake ok
debugger

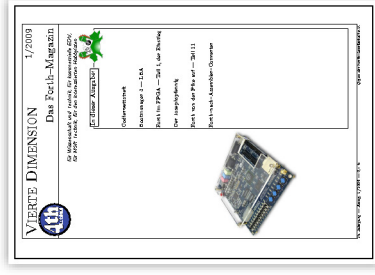
µCore> use bye to return
µCore> trace test
test
000059: 93 00 000003 3
00005B: 10 00005B 0-BRANCH 3 3
00005C: 0C 08 0A 1-
00005F: FF 28 hell CALL
000061: E9 19 delay CALL
000063: 08 19 dunkel CALL
000067: D7 19 delay CALL
000069: F0 09 00005B BRANCH 2 2
00006B: 10 00005B: 10
00006C: 0C 08 0A 1-
00006F: FF 28 hell CALL
000071: E9 19 delay CALL
000073: 08 19 dunkel CALL
000077: D7 19 delay CALL
000079: F0 09 000069 BRANCH 2 2
00007B: 10 00007B: 10
00007C: 0C 08 0A 1-
00007F: FF 28 hell CALL
000081: E9 19 delay CALL
000083: 08 19 dunkel CALL
000087: D7 19 delay CALL
000089: F0 09 000079 BRANCH 2 2
00008B: 10 00008B: 10
  
```

MicroCore Debugger

- ✓ new Debugger in Forth
- ✓ Gforth - portable
- ✓ currently V24 für Windows, but Code für Linux and Mac in Gforth exists
- ✓ integrated environment
- ✓ incremental compilation

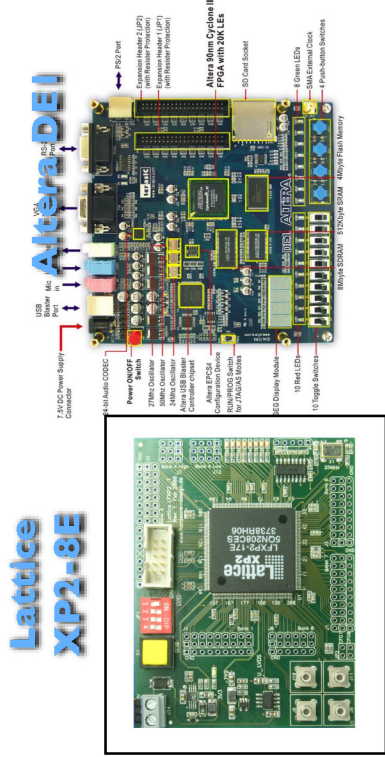


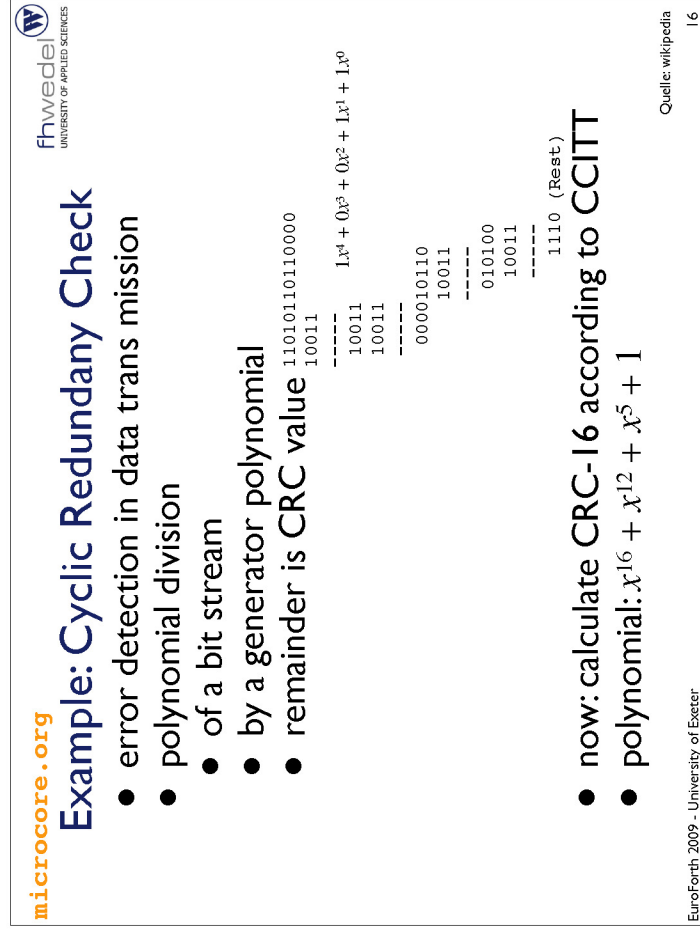
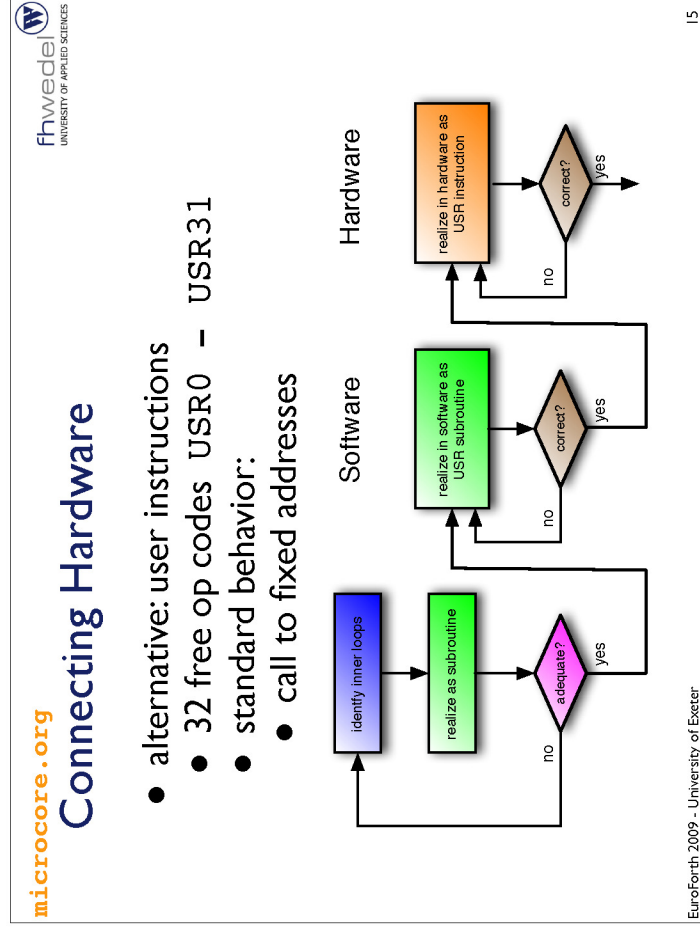
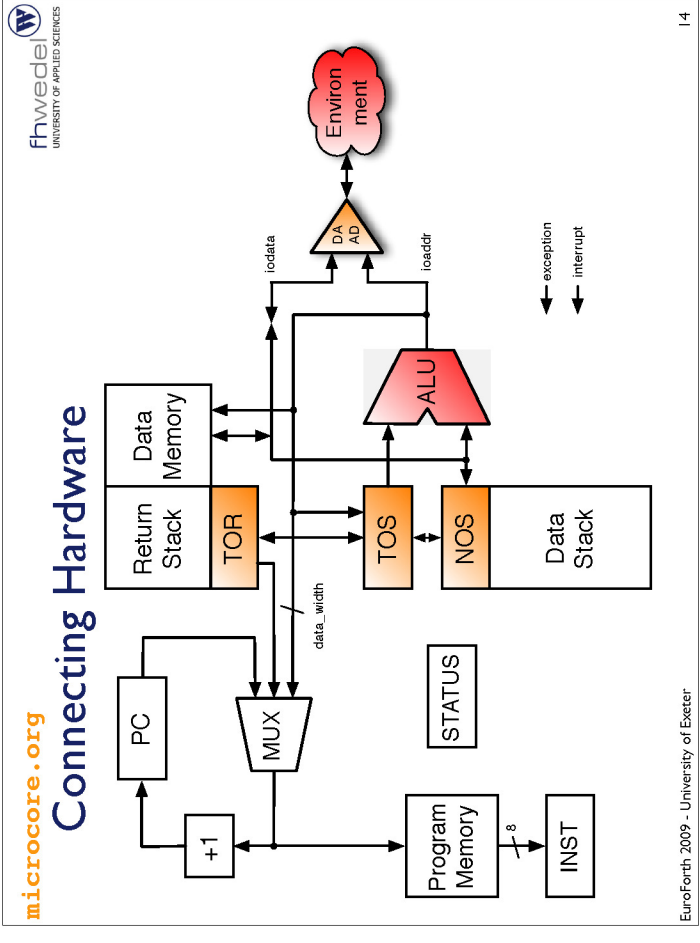
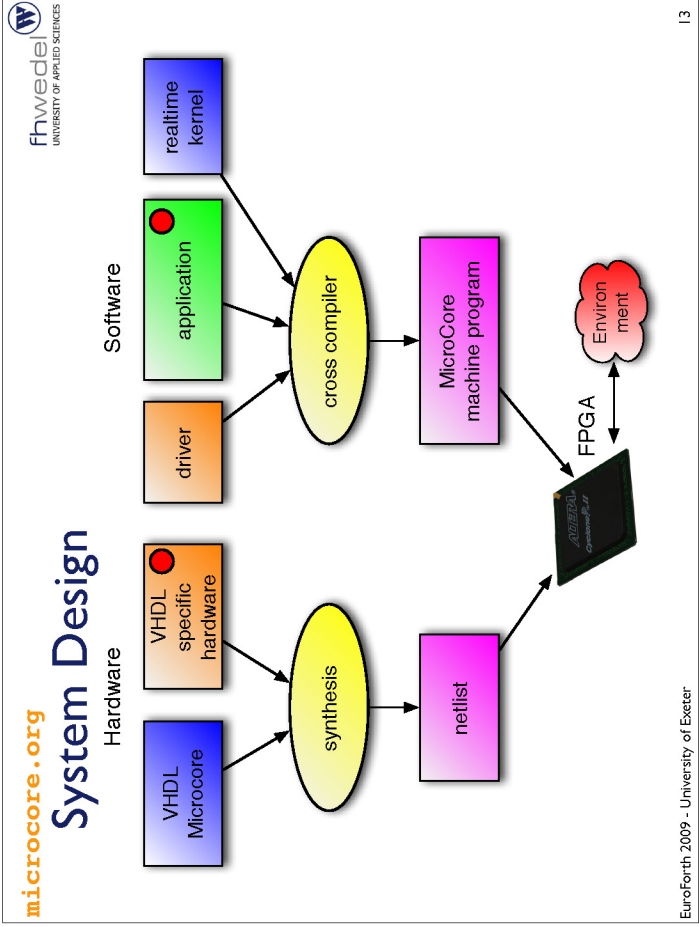
FPGA article series



FPGA-Projekt

currently two target platforms:





Example: Cyclic Redundany Check

- Software realization
- process data byte-wise:

```

unsigned int crc_ccitt_step(unsigned char b
                           unsigned int crc) {
    crc ^= (unsigned char)(crc >> 8) | (crc << 8);
    crc ^= b;
    crc ^= (unsigned char)(crc & 0xff) >> 4;
    crc ^= (crc << 12);
    crc ^= ((crc & 0xff) << 4) << 1;
    return crc;
}

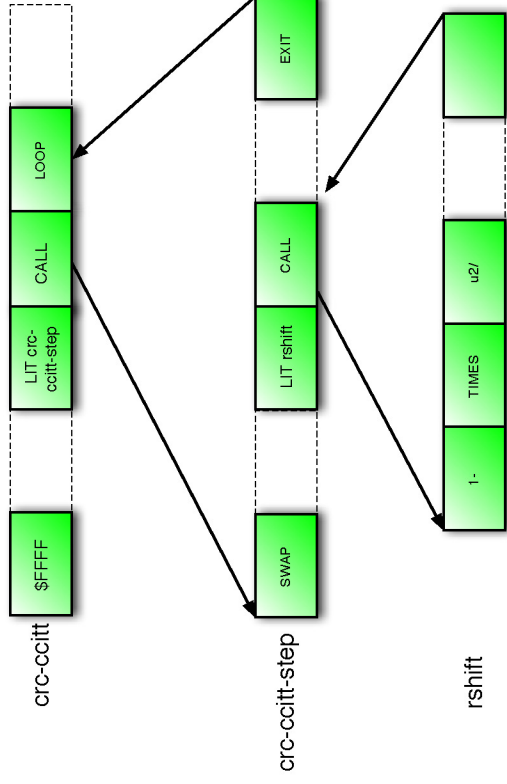
unsigned int crc_ccitt(unsigned char* data
                       int len) {
    unsigned int crc = 0xFFFF;
    for (int i=0; i<len; i++) {
        crc = crc_ccitt_step(data[i], crc);
    }
    return crc;
}
    
```

Forth

```

: lshift ( x n -- x' ) | - times 2* ;
: rshift ( x n -- x' ) | - times u2/ ;
: crc-ccitt-step ( crc b -- crc' )
  swap dup 8 lshift
  swap 8 rshift or xor
  dup $FF and 4 rshift xor
  dup 12 lshift xor
  dup $FF and 5 lshift xor
  $FFFF and
  ;
: crc-ccitt ( addr len -- crc )
  $FFFF rot rot
  bounds ?DO | @ | @ crc-ccitt-step
  LOOP
  ;
    
```

Call Structure

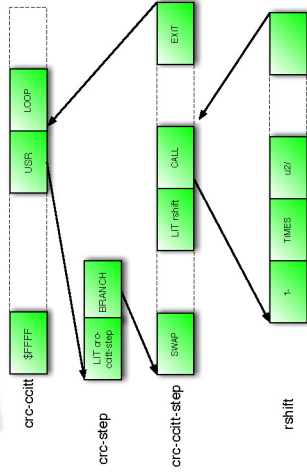


Software Realization by a USR-Instruction

```

5 USR: crc-step [ ] crc-ccitt-step nop branch ;USR
: crc-ccitt ( addr len -- crc )
  $FFFF rot rot
  bounds ?DO | @ | @ crc-step
  LOOP
  ;
    
```

Forth



Hardware Realization as a USR-Instruction

```

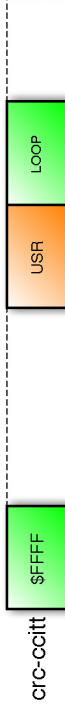
function crc_ccitt_step
(d: std_logic_vector(7 downto 0);
 crc: std_logic_vector(15 downto 0))
return std_logic_vector is
variable result: std_logic_vector(15 downto 0);
begin
    result(0) := d(4) xor d(0) xor crc(8) xor crc(12);
    result(1) := d(5) xor d(1) xor crc(9) xor crc(13);
    result(2) := d(6) xor d(2) xor crc(10) xor crc(14);
    result(3) := d(7) xor d(3) xor crc(11) xor crc(15);
    result(4) := d(4) xor crc(12);
    result(5) := d(5) xor d(4) xor d(0) xor crc(8) xor crc(12) xor crc(13);
    result(6) := d(6) xor d(5) xor d(1) xor crc(9) xor crc(13) xor crc(14);
    result(7) := d(7) xor d(6) xor d(2) xor crc(10) xor crc(14) xor crc(15);
    result(8) := d(7) xor d(3) xor crc(6) xor crc(11) xor crc(15);
    result(9) := d(4) xor crc(1) xor crc(12);
    result(10) := d(5) xor crc(2) xor crc(13);
    result(11) := d(6) xor crc(3) xor crc(14);
    result(12) := d(7) xor d(4) xor d(0) xor crc(4) xor crc(8) xor crc(12) xor crc(15);
    result(13) := d(5) xor d(1) xor crc(5) xor crc(9) xor crc(13);
    result(14) := d(6) xor d(2) xor crc(6) xor crc(10) xor crc(14);
    result(15) := d(7) xor d(3) xor crc(7) xor crc(11) xor crc(15);
return result;
end crc_ccitt_step;
    
```

Hardware Realization as a USB-Instruction

```
VHDL
CASE i_type IS
WHEN OTHERS => -- op_USB
CASE i_USB IS
WHEN "00110" => -- USB 6
-- push_stack("0000000000000000000000001111");
pop_stack
-- tos_new <= multiply(tos_reg,nos_reg);
tos_new <= crc_ccitt_step(tos_reg,nos_reg);
WHEN OTHERS =>
IF inst=INT_OP THEN
push_stack(tos_reg);
tos_new <= status;
END IF;
END CASE;
END CASE;
```

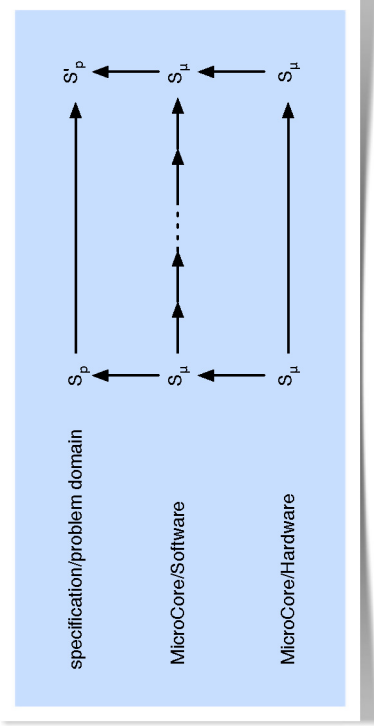
- Data stack effekt
- ▲ Return stack effekt
- ▲ Sequencer

```
5 USB: crc-step ;USB
: crc-ccitt ( addr len -- crc )
$FFFF rot rot
bounds ?DO 1 @ crc-step
LOOP
Forth
```



Proof obligations

- In what areas do you need to think about the problem?

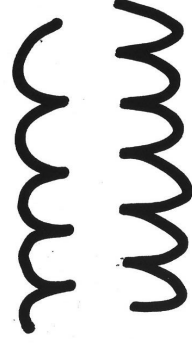


Future

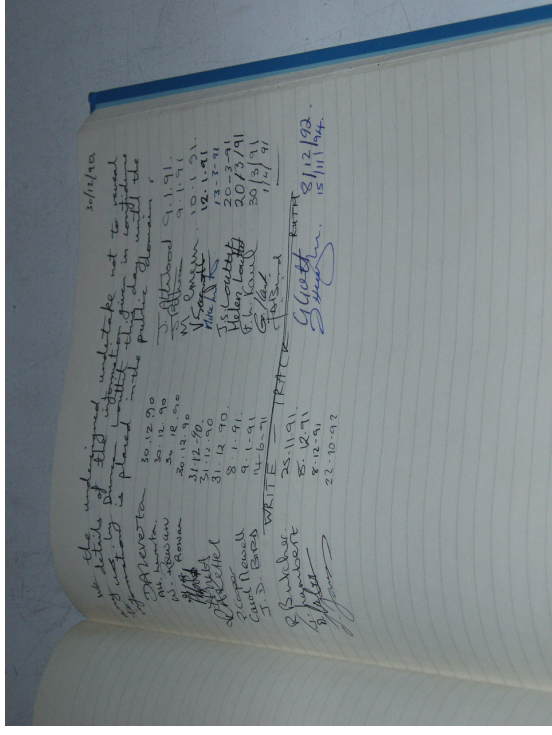
- Other candidates for realization in hardware:
 - Multiplication
 - MD5
 - RSA
 - digital filters
 - FFT
- **MicroCore**
 - Experiments in computer architecture
 - Overflow detection already present
 - Safe program execution
 - Tagged memory for type information

Writetrack: an inventor's
experience

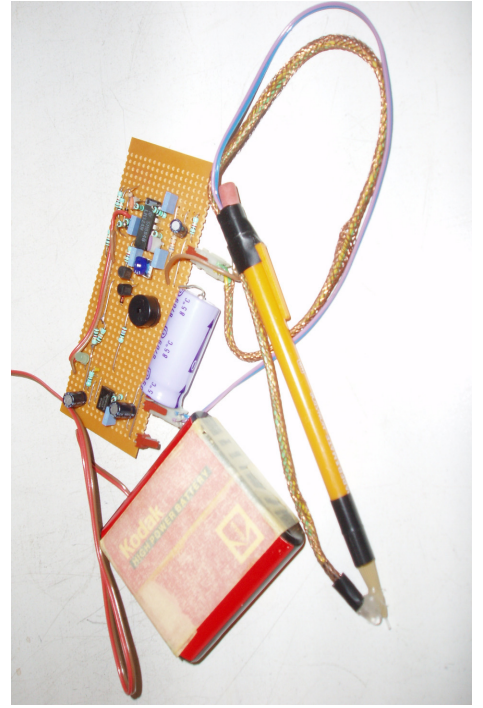
1990: the birth of an idea.



Patent agents and protection.



As it was:



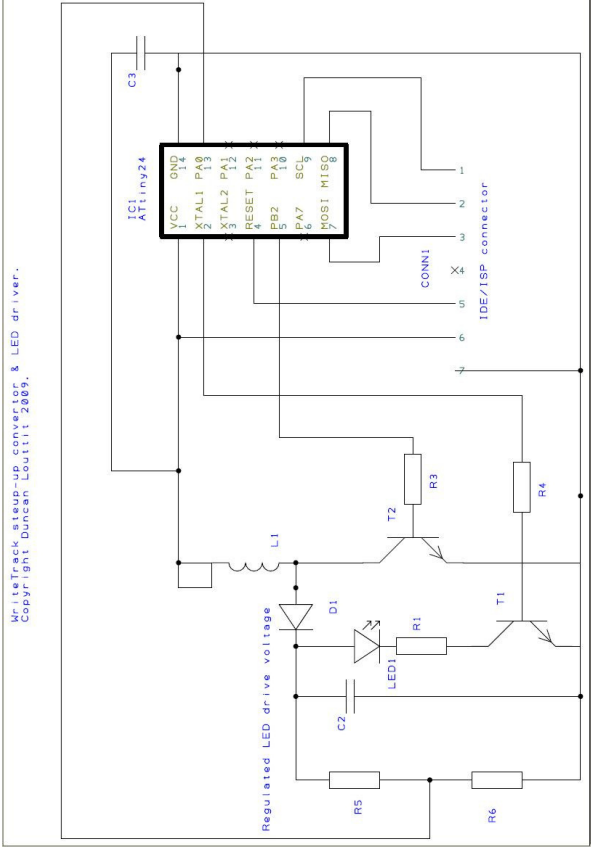
“Project X”

2006: at last some time!



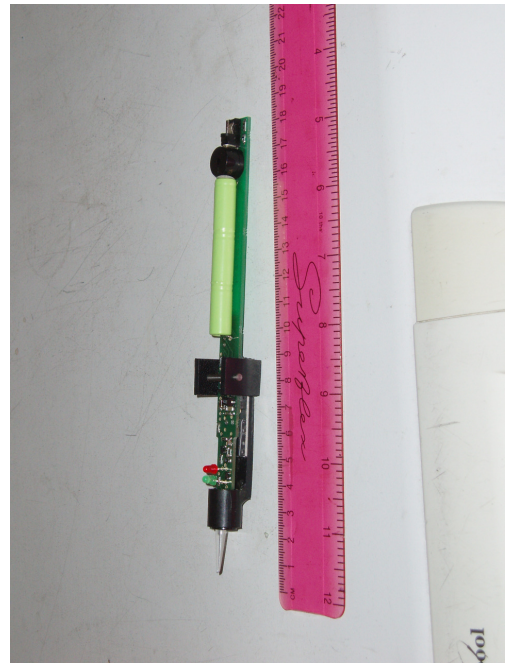
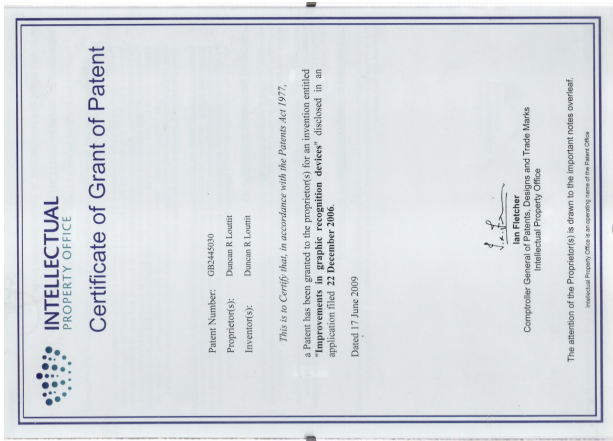
What's it got to do with FORTH?

WhileTrack:steup-up_converter & LED driver.
Copyright: Danzen, 2009.



2009: Grant and commercialisation.

And James?



Get Rid of Accumulated Cruft

Rebuild from Scratch: Internet 2.0
Things I like to do during my sabbatical

Bernd Paysan

EuroForth 2009, Exeter

- The Internet is finally starting to show its age — a lot of things don't work as well as they should
- At the same time, dependency increases
- IPv6 solves some of the problems, but creates others, and ignores many
- Too complex, not following its own specifications (RFCs) in parts, too many protocols
- Postel problem: If you are generous in what you accept, produced rubbish will increase over time
- Solution: Throw it away and rebuild from scratch

Outline

- 1 Motivation
 - 30 Years of Accumulated Cruft, and Still Accumulating Requirements
- 2 Topology
 - Packet Header
 - Flow Control
 - 1:n Connections
 - Legacy
- 3 Abstraction
- 4 Security

Scalability Must work well with low and high bandwidths, loose and tightly coupled systems, few and many hosts connected together over short to far distances.

Easy to implement Must work with a minimum of effort, must allow small and cheap devices to connect. One idea is to replace “busses” like USB and firewire with cheap LAN links.

Security Users want authentication and authorization, but also anonymity and privacy. Firewalls and similar gatekeepers (load balancers, etc.) are common.

Media capable This requires real-time capabilities, pre-allocated bandwidth and other QoS features, end-to-end.

Transparency Must be able to work together with other networks (especially Internet 1.0).

Requirements

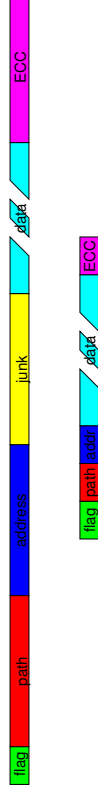
- The Internet is connected by routers, which route every packet
- Most Internet traffic is connection oriented; even DNS usually goes to a local cache server
- IPv6 increases the routing tables even more

Physical Route

- Take first n bits of target address and select destination
- Shift target address by n
- Insert bit-reversed source into address field

- Routing server resolves name → destination. Routing service is a cache, distributed data base like DNS.
- Routing happens once per host and destination, and is cached
- Net interconnects and end nodes separated — no problem of abusing end nodes
- Return path not spoofable, and shared with destination

	Size
Flags	2
Path	2/8
Address	2/8
Junk	0/8
Data	32/128/512/2k
ECC	L1 dependent



TCP/IP

- End node flow control
- Slow start (scalability problem)
- Randomly dropped packets in case of overload
- Retransmits and fake or arbitrary delayed acknowledges cause problems
- Remember: End node is possibly hostile!

Network flow control

- Keep statistics
- Send "jam" messages back to worst offenders
- Notify early about available resources
- Topological fairness, not per-connection fairness

1:n Connections

Multicasting

- Route is a tree, not a path
- Use table to select multiple destinations
- Hosts joining a multicast search upstream switch and add address to table
- Multicasting is a scarce resource — use when appropriate
- No regulation needed

Broadcasting

- Use bitmap to select destinations
- Small end user routers may use CAM^a
- Broadcasts use global/region numbers: Regulation needed

^aContent Addressed Memory

Legacy

- Internet 2.0 over Ethernet Put packets into ordinary Ethernet frames — needs jumbo frames for 2k packets
- Internet 2.0 over IP Put packets into UDP datagrams, use VoIP/P2P techniques to route through NAT
- IP over Internet 2.0 Similar to MPLS:¹ Build tunnels per destination group
- Access to Content Protocol translation (all kind of different implementations possible)

¹Multi Layer Protocol Switching

Abstraction

Avoid unnecessary abstraction

Distributed Shared Memory Packet transfer data blocks from one node to another, on this level, it's just addresses and data

Active Messages Separate data and "metadata." Metadata is really code — coded in a (sandboxed) stack-VM that suits the abstraction model, and allows expanding it. Protocol hierarchy build on a common foundation of commands

Files with Attributes Most Internet protocols deal with files of sorts. Files often have attributes (sender/receiver and subjects on E-Mails, data type, date, name, possibly references in Hypertext files). Add properties to files, and allow querying for those.

Abstraction II

Caching, P2P, Clouds Distributed file system: Use a cryptographic hash ("URI") to identify documents and look them up where you can get them cheapest — neighbour computer, cloud cohosting, P2P network.

Text Formatting Wikis show: HTML too complicated. Use wiki-style formatting plus separated style sheets for layout.

Single Frontend With consolidated protocols, the browser should do everything neatly. Add AJAX-like capabilities directly to the data model the browser understands.


```
Open connection
s'' http' protocol s'' foo.com'' host
<addr> <len> data-window
<addr'> <len'> command-window
open-port
```

```
Get hypertext document
:? expand-preload BEGIN dup list-len >r
dup s'' preload'' get-attribute-list append
uniquify-list dup list-len r> = UNTIL ;
[[ s'' bar/url.wiki'' ]] expand-preload
' send-mime-file map-list
```

- Encryption to avoid eavesdropping
- Authentication and Authorization for access control
- Key exchange and trust network (PKI)
- Anonymity if necessary (onion routing)

- This is pure vaporware now
- Path to reality: Reference implementation, RFC, IETF discussion

 Bernd Paysan
Internet 2.0
<http://www.jwdt.com/~paysan/internet-2.0.html>

Forth Type Checker

by

Jürgen Pfitzenmaier
pfitzen@web.de

Abstract

Forth engine <---> Type Checker

Installation & next version

monolithic --> library + ocaml runtime
pforth

Reading input

from file --- from keyboard

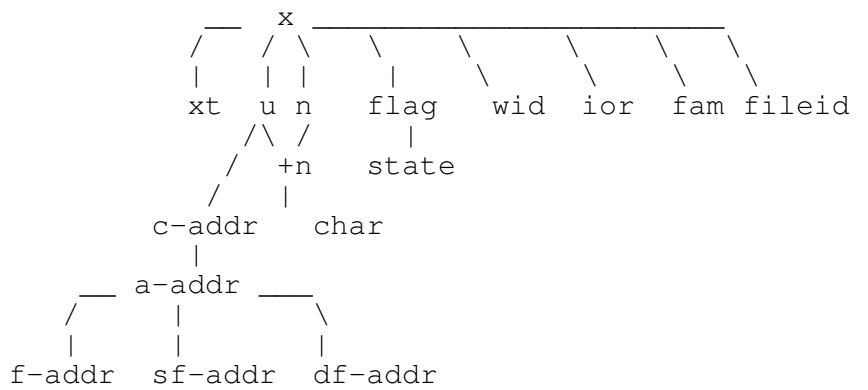
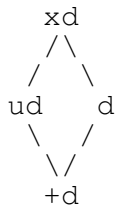
automatic testing

Declaring Types

much like in the current standard

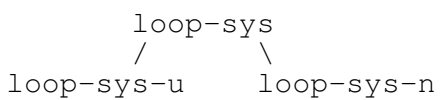
```
\T Execution: (u -- x n) \  
\T Compilation: (x -- ) Runtime: ( -- x)  
  
\T Assume: Execution: (x -- x)  
  
\T Cast: Execution: (n -- ) (F: -- r)
```

Standard Types



```
colon-sys  do-sys  
case-sys   of-sys  
orig       dest
```

Finer types for loops



Exit as Forward Connection

nest-sys like-nest-sys

How to empty the Stack

i*x j*x k*x empty

Special Types

state recurse

Reserved for future use

obj class frame

Full Declaration prevents internal shifting

Compilation: (--) Runtime: (x -- x)

is NOT equal to

Runtime: (x -- x)

Delay type check for one token

```
RESTRICT
IMMEDIATE
IMMEDIATE RESTRICT
```

Example 1

```
: bad1 \T ( u -- u)
   -1 +
; \ end of bad1

: bad1b \T ( u -- u)
   -1 +
; \ end of bad1b

: good1 \T ( n -- n)
   -1 +
; \ end of good1
```

Example 2

Constant as a simple constraint

```
: good2 \T ( -- n)
   4
; \ end of good2

: good2b \T ( -- 4)
   4
; \ end of good2b
```

Example 3

IF ELSE THEN and dependent types

```
: bad3 \T (x true -- x x) | (x false -- char.addr)
  IF
    DUP DUP
  ELSE
    PAD
  THEN
; \ end of bad3

: bad3b \T (x true -- x x x) | (x false -- char.addr)
  IF
    DUP DUP
  ELSE
    PAD
  THEN
; \ end of bad3b

: good3 \T (x true -- x x x) | (false -- char.addr)
  IF
    DUP DUP
  ELSE
    PAD
  THEN
; \ end of good3

: good3b \T (x_{1} true -- x_{1} x_{1} x_{1}) | (false -- char.addr)
  IF
    DUP DUP
  ELSE
    PAD
  THEN
; \ end of good3b
```

Example 4

Casting Types

```
: FLOATSTACK? \T ( -- flag)
  S" FLOATING-STACK" environment?
  \T Cast: (false -- false) | (i*x true -- n true)
  if 0<>
  else false
  then
;

```

Example 5

Floating Point Stack

```
: float1 [ floatstack? ] [IF] \T (F: r r -- r)
                        [ELSE] \T (r r -- r)
                        [THEN]
      F+
; \ end of float1
```

```
: local1 \T Cast: ( n n -- n n ) ; DOKU warum cast vor local-def
      { m n -- } \T ( n n -- n n )
      n dup m * dup n *
; \ end of local1

: local2 \T Cast: ( n n -- n n )
      { v1 v2 | l1 l2 -- } \T ( n n -- )
      v1 . v2 . cr
      v1 v2 + -> l1
      l1 . l2 . cr
; \ end of local2
```

```
: exit1 \T (false -- +n) | (true -- +n +n)
      IF
          1
          THEN
          2
; \ end of exit1

: exit2 \T (false -- +n) | (true -- +n)
      IF
          1 EXIT
          THEN
          2
; \ end of exit2
```

Example with ABORT and CASE

```
: POSTPONE \T ( -- ) ; CORE 6.1.2033
  bl word find
  CASE
    0 OF ." Postpone could not find " count type cr ABORT ENDOF
    1 OF compile, ENDOF \ immediate
    -1 OF (compile) ENDOF \ normal
  ENDCASE
; immediaterestrict
```

Problem with matching types

```
: abort2 \T (+n -- n)
  CASE
    1 OF 1 ENDOF
    2 OF 2 2 ENDOF
  ENDCASE
; \ end of abort2

: abort3 \T ( empty -- ) (R: empty -- ) \
  \T | ( k*x -- -1) (R: k*x -- )
  -1 throw
; \ end of abort3
```

RECURSE in action

```
: recl \T (n -- n)
  DUP 2 > IF
    DUP 1 - * RECURSE
  THEN
; \ end of recl
```

Using CREATE and DOES>

```
: 2VARIABLE \T ( -- ) nameExecution: ( -- d.a-addr ) ; DOUBLE 8.6.1.0440
  create 0 , 0 , \T AllowRebind: ( -- d.a-addr)
; \ end of 2variable
```

```
: CONSTANT \T (x_{1} -- ) nameExecution: ( -- x_{1}) ; CORE 6.1.0950
  \ XXX im ANS fehlt der index 1
  CREATE , \T AllowRebind: (x -- )
DOES> \T ( x_{1}.a-addr -- x_{1} )
  @
; \ end of constant
```

```
: loop1 \T (n --)
  5 do cr loop
; \ end of loop1

: loop2 \T (n --)
  5 do J 4 = if LEAVE then loop
; \ end of loop2
```

Dirty Failures

```
: dirtyloop \T (n --)
  5 do if LEAVE then loop
; \ end of dirtyloop

: dirty \T ( -- )
  R> DROP
; \ end of dirty

: ?EXIT \T Compilation: ( -- ) Runtime: (x --) Runtime: (R: nest-sys --) \
  \T | Compilation: ( -- ) Runtime: (x -- )
  postpone IF postpone EXIT postpone THEN
; immediate
```

Microcore Status Report

Klaus Schleisiek
 SEND Off-Shore Electronics GmbH
 Hamburg
 ks@send.de

Debugger in Gforth

- Cross compiler
- VHDL backend
- Interactive debugger with 2 wire umbilical
- Program memory change without processor state change
- Breakpoints
- interactive tracing

Overview

- Debugger in Gforth
- Objects & Methods
- VHDL code amalgamation
- Instruction overview
- Un-interruptible sequences
- Optimization
- uCore 2

Objects & Methods

```

Class
inherit
New (Object)
:: (Attribute)
[] (Reference), is
Array
Class Point Point definitions
Cell :: X
Cell :: Y Point seal
M: @ ( point -- x y ) dup Self X@ Self Y @ ;
Forth definitions
  
```




alu.vhd
 dstack.vhd
 rstack.vhd
 sequencer.vhd
 uBus.vhd
 uCore.vhd ==> uCore.vhd



```
00 mults      -0 +          +0 2dup +      00 invert
00 divs       -0 +c        +0 2dup +c    00 2*
              -0 -         +0 2dup -      00 2/
00 crcs       -0 swap-    +0 2dup swap-  00 u2/
00 dro1       -0 and      +0 2dup and   00 ror
00 dro2       -0 or       +0 2dup or    00 rol
00 +st (2 cyc) -0 xor     +0 2dup xor   00 0=
00 swap       -0 nip      +0 2dup over  00 time?
```



```
00 nop        -0 branch    +0 dup      -0 drop
00 rot        ?0 ?dup-branch ?0 ?dup     -0 pack
00 -rot       -0 s-branch  +0 tuck     +- call
00 0<        -0 ns-branch  +0 under    -0 times
00 less?     -0 z-branch   +0 unpack   -? z-exit
0- rdrop     -0 nz-branch  +0 ovfl?    -? nz-exit
0- ldivs     -0 no-branch  +0 carry?   -? next
0- exit      -0 nc-branch  +0 I        -- iret
```



```
-0 status!    -0 ST          +0 LD        +0 status@
-0 st_set     -0 1 + ST     +0 1 + LD    +0 r@
+- >r         -0 2 + ST     +0 2 + LD    +- r>
-0 lst        -0 3 + ST     +0 3 + LD    +0 lld
-0 rsp!       -0 4 - ST     +0 4 - LD    +0 rsp@
-0 dsp!       -0 3 - ST     +0 3 - LD    +0 dsp@
-0 tst        -0 2 - ST     +0 2 - LD    +0 tld
              -0 1 - ST     +0 1 - LD
```

USER group

```
-- 0divs     +- -divs
++ int
0+ exc
0? ?ovfl
0+ break
```



```
+st ( n addr -- addr )
: +! +st drop ;
only 1st cycle is new and its Forth equivalent is
: (+st ( n addr -- n+ addr )
  >r r@ + r> ;
2nd cycle executes a store instruction
```

An un-interruptible sequence !



```
: < - less? ;
: > swap - less? ;
: u> swap - drop carry? 0= ;
: u< - drop carry? 0= ;
```



```
set_st ( mask -- )
1 Constant #carry
#carry set_st
sets carry
#carry invert set_st
resets carry
```



```
<word> CALL ; => <word> BRANCH
>r ; => BRANCH
?dup IF => ?dup_BRANCH
0= IF => 0<>BRANCH
0< IF => NS-BRANCH
0< 0= IF => S-BRANCH
carry? IF => NC-BRANCH
ovfl? IF => NO-BRANCH
```

Optimizations



```

over over +      => 2dup_+
swap -          => swap-
swap over      => tuck
over swap      => under
<n> + LD       => +n_LD
    
```

uCore 2



Bit#	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CALL	1						15-bit abs. Address									
LIT	0	1					14-bit Lit									
BRANCH	0	0	1	1			12-bit signed branch offset									
?BRANCH	0	0	1	0	1		11-bit signed branch offset									
NEXT	0	0	1	0	0		11-bit signed branch offset									
MEM	0	0	0	1	1	pop	push	9-bit signed post-increment								
ZPAGE	0	0	0	1	0	pop	push	1	8-bit absolute address							
[RSP]	0	0	0	1	0	pop	push	0	1	7-bit offset						
[TASK]	0	0	0	1	0	pop	push	0	0	7-bit offset						
ALU	0	0	0	0	1	pop	push	exit	256 Operations							
REG	0	0	0	0	0	pop	push	exit	8-bit signed address							