

29th EuroForth Conference

September 27-29, 2013

Haus Rissen
Hamburg
Germany

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 29th EuroForth finds us in Hamburg for the second time (after 2010). The two previous EuroForths were held in Vienna, Austria (2011), and in Oxford, England (2012). 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. This year there were three submissions to the refereed track, and one was accepted (33% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 16 submissions, 9 accepts, 56% acceptance rate. Each paper was sent to at least three program committee members for review, and they all produced reviews. Two papers were co-authored by a program committee member, who was not involved in the review of these papers; the reviews of all papers (including the ones co-authored by the PC member) are anonymous to the authors. I thank the authors for their papers and the reviewers and program committee for their service.

Several papers were submitted to the non-refereed track. These proceedings also include these papers and slides for presentations without paper.

Workshops and social events complement the program.

This year's EuroForth is organized by Klaus Schleisiek and Ulrich Hoffmann.

Anton Ertl

Program committee

Sergey N. Baranov, SPIIRAS, Russia

M. Anton Ertl, TU Wien (chair)

David Gregg, Trinity College Dublin

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

Contents

Refereed papers

| | |
|---|---|
| Andrew Read: Optimizing memory access design for a 32 bit FORTH processor | 5 |
|---|---|

Non-refereed papers

| | |
|--|----|
| Nick J. Nelson: Forth Query Language (FQL) — Implementation and Experience | 24 |
| M. Anton Ertl: PAF: A Portable Assembly Language | 30 |
| M. Anton Ertl: Standardize Strings Now! | 39 |
| Sergey Baranov: Forth in Russia | 44 |
| Willi Stricker: Forth Floating Point Word-Set without Floating Point Stack | 51 |

Presentations

| | |
|---|----|
| Ulrich Hoffmann: Forth Literate Programming with IPython notebook . | 58 |
| Gerald Wodni: Forth to .NET Bridge | 62 |
| M. Anton Ertl: Region-Based Memory Allocation | 65 |
| Bernd Paysan: net2o: Application Layer — Browser Components | 68 |

Optimizing memory access design for a 32 bit FORTH processor

Andrew Read

June 2013

andrew81244@outlook.com

Abstract

This paper compares and contrasts two alternative approaches to designing system memory access for a 32 bit FORTH processor. One approach maximizes clock speed whilst the other maximizes instruction throughput. Each approach is found to have advantages and disadvantages. The project's conclusion is that a hybrid memory access design that considers the differing needs of the CPU control unit and datapath is likely to be the optimum performance strategy for a FORTH machine.

1 Introduction

The N.I.G.E. Machine is a complete computer system implemented on an FPGA development board [1]. It comprises a 32 bit softcore processor optimized for the FORTH language, a set of peripheral hardware modules, and FORTH system software. The system is primarily designed to support the rapid prototyping of experimental scientific apparatus. The N.I.G.E. Machine was first presented in a paper at EuroFORTH 2012 [2]. In the conclusions of that paper a number of avenues for further work were suggested. These included improving the bandwidth between the CPU and system memory, implementing an SD-card interface with a FAT file system, and porting the N.I.G.E. Machine to a higher performance FPGA development board.

As of the date of this paper an SD-card interface, native FAT file system, and FORTH File-Access wordset have been successfully implemented and tested. This greatly simplifies the transfer of program and data files between the N.I.G.E. Machine and a PC. The upgrade was relatively straightforward and did not raise substantial new design issues. Porting the N.I.G.E. Machine to a Digilent Atlys development board with a Spartan 6 FPGA is currently underway.

The goal of improving memory access bandwidth was set with the intention of redesigning the connection between the system memory (i.e. FPGA Block RAM, "SRAM"¹) and the softcore CPU. The CPU is a 32 bit processor, but the memory databus between the CPU and system memory is only 8 bits wide in the original N.I.G.E. Machine design (fig. 1). Widening the databus from 8 bits to 32 bits to match the width of the softcore processor might initially seem to be a worthwhile and simple capacity increase at the cost of some further FPGA resources. However upon closer examination this design change actually raises a number of interesting implications

¹In this paper and the N.I.G.E. Machine documentation, Block RAM that is used for system memory is given the term "SRAM".

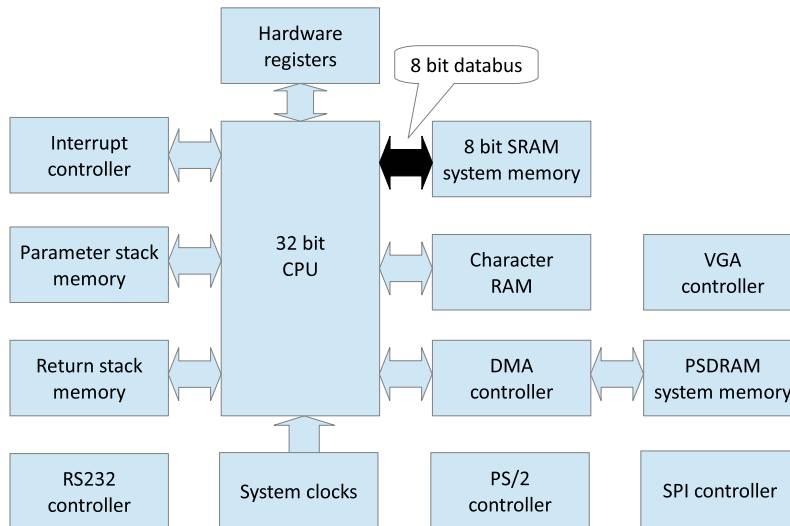


Figure 1: N.I.G.E. Machine system diagram showing the principal components and CPU connections. This is the 8 bit databus configuration (highlighted), as presented at EuroFORTH 2012.

in terms of conflicting requirements for functionality. As a result, the final preference between the 8 bit and 32 bit memory databus configurations requires deeper consideration of the intended application and the specific hardware on which the N.I.G.E. Machine will be deployed.

Using the N.I.G.E. Machine as an example, this paper explains the background and challenges in optimizing memory access for a 32 bit FORTH processor. The problems faced are not new and would be recognizable to any processor designer. The paper seeks to advance the state of the art in softcore FORTH systems by providing a systematic, evidence-based report with detailed implementation information.

The structure of the paper is as follows. First of all the memory access requirements of a FORTH processor are discussed on a qualitative level (section 2). This section also introduces the general tradeoffs between an 8 bit and a 32 bit system memory databus. The 32 bit databus design that was successfully implemented on the N.I.G.E. Machine is described next (section 3). Performance benchmark comparisons between the 8 bit and 32 bit databus configurations of the N.I.G.E. Machine are presented (section 4). In the light of the performance benchmarking results there is a discussion about the tradeoff decisions guiding which implementation should ultimately be preferred (section 5). The conclusion attempts to synthesize the experience with the N.I.G.E. Machine into lessons for FORTH processors in general and suggests avenues for further work (section 6).

All of the N.I.G.E. Machine design files and software are available open source [11].

2 Memory access requirements of a FORTH processor

2.1 Review of prior work

A number of softcores have been designed specifically to execute FORTH [3, 4, 5, 6, 7, 8]. Several aspects of the J1 [3] directly inspired the design of the N.I.G.E. Machine. In most of these examples the focus of the design work has been the CPU itself while memory access requirements have been less of a consideration. There have been some notable exceptions. For example the RTX 2000

includes an on-chip memory page controller that considerably enhances memory access [10], and Klaus Schleisiek's microcore [4] features pre-incrementing and post-incrementing data memory operations (++!, !++, ++@, @++) that are directly supported by hardware. Another approach to improving memory access efficiency is to pack more instructions per unit of data. For example Bernd Paysan's b16 processor[5] packs 3 instructions into a 16 bit word.

Philip Koopman [10], in discussing the characteristics of 16 bit stack machines, makes the point that the fit between the width of the CPU datapath and the FORTH programming model is a key design factor. Koopman observes that an 8 bit CPU is likely be unsatisfactory from a performance standpoint because too much time would be required in synthesizing 16 bit operations, whilst at the time of writing (1989), specifying a 32 bit CPU might be too expensive. The reasonable assumption being made here is that wider datapaths can also access memory across wider databuses, thus increasing processing bandwidth.

Koopman also discusses the limits of memory bandwidth. He makes the point that traditional, register-based processors are very dependent on cache memory. This creates performance bottlenecks that are subject to the hit ratios of various caches, and the organization of the code produced by the compiler. He envisages stack machines offering an alternative approach to memory organization because of their very fast procedure calls. (Procedure calls are fast on stack machines because there is no need to save a register set in system memory). In the stack machine approach, frequently executed code can be stored in on-chip memory, avoiding the requirement for dynamic memory management. The design of N.I.G.E. Machine follows the approach envisioned by Koopman in that code density is very high, and system memory comprises FPGA static RAM (Block RAM) that can be accessed in a single clock cycle.

This paper builds on Koopman's theme of efficient memory access to focus on the specific problem of how best to design the connection between the CPU and system memory on a 32-bit FORTH machine. The first question to answer is, what are the memory access requirements of a 32 bit FORTH processor?

2.2 Impact of system design objectives on memory access requirements

A FORTH processor has some distinct advantages in real-time control applications [9, 10]. The design objectives of the N.I.G.E. Machine reflect its intended role in the real time control of scientific hardware. The principal objectives are listed below and are in turn the main influence on memory access requirements.

Deterministic execution. Avoiding jitter in electronic interfaces is an essential real time requirement for precise control and measurement. This necessitates that the softcore CPU is designed so that any given instruction will always execute in a certain number clock cycles, including instructions that access system memory.

High instruction throughput. High instruction throughput translates directly into higher processing performance and shorter interrupt response times. This is especially important on FPGA softcore processors that operate at lower clock rates than comparable dedicated microcontrollers. Throughput of once instruction per clock cycle is the ideal target.

Maximum code density. The fastest memory resource available to a softcore CPU is FPGA Block RAM. Block RAM also has the advantage over external memory of deterministic access (i.e. guaranteed single clock cycle read/write). However Block RAM resources are typically limited to several tens or hundreds of kilobytes. To maximize the use of Block RAM as program memory, code density needs to be as high as possible. Ideally instructions should be encoded in no more than a single byte.

Flexible memory access. With a 32 bit processor, optimizing the speed and flexibility of memory access requires that CPU instructions are available that read or write memory in byte,

16 bit, and 32 bit formats. Flexibility is further enhanced if even address alignment is not required when accessing 16 bit or 32 bit data in system memory.

2.3 Advantages and limitations of an 8 bit wide databus

Block RAM can be configured in a variety of formats by specifying (with the FPGA design tools) the width (i.e. data size: 8 bits, 16 bits, etc.) and depth (i.e. number of address lines) of the required memory resource. The N.I.G.E. Machine’s softcore is a 32 bit CPU, but the system presented at EuroFORTH 2012 incorporated system memory configured in an 8 bit wide format. Coupling the 8 bit wide Block RAM to the CPU in this design is an 8 bit databus and an address bus that references memory byte-by-byte (fig 1).

This configuration has some advantages: an 8 bit databus naturally facilitates the fetching of single byte instructions, and a byte-by-byte address format avoids misaligned address boundaries. The design of separate CPU instructions that read or write memory in byte, 16 bit, or 32 bit formats is also easily facilitated in this configuration by arranging for the CPU control unit to read or write byte data from/to consecutive memory locations in consecutive CPU clock cycles, as required by the length of the data.

In conjunction with the N.I.G.E. Machine’s three stage pipeline [2], the 8 bit databus configuration provides straightforward memory access and throughput of one instruction per clock cycle for almost all instructions. The important exceptions are those instructions that require access to more than a single byte of memory and which therefore require more than one cycle to execute. These include all of the load literal instructions and the word and longword memory fetch and store instructions.

This impact of this limitation becomes apparent when considering the relative frequency of instructions that comprise the FORTH system software (table 1). The most common instruction, which occurs almost twice as frequently as any other, is LOAD.W (or “#.W”), the instruction to load a 16 bit literal to the stack. The ubiquity of the LOAD.W instruction in the FORTH system software reflects the load/store architecture of a stack machine CPU and the subroutine threaded nature of FORTH (LOAD.W is the instruction used to load a subroutine address prior to a JSR (jump to subroutine)). In the N.I.G.E. Machine instruction format, LOAD.W is a three byte instruction comprising an opcode byte followed by the high and low data bytes in big endian format. It takes three CPU clock cycles to execute with an 8 bit databus configuration. On account of the narrow instruction fetch, the most commonly executed instruction is therefore one of the minority of instructions that have low throughput (less than one instruction per clock cycle). This is a defeat of the optimization maxim, “make the common case fast”, and was the key motivation to consider the design of a wide (32 bit) memory databus.

(Looking ahead, a 32 bit memory databus also allows the development of a new jump-to-subroutine instruction (JSL) that takes a 24 bit immediate literal target.)

| Instruction | Frequency |
|--------------------|------------------|
| LOAD.W | 17.88% |
| JSR | 9.17% |
| RTS and ,RTS | 9.06% |
| LOAD.B | 6.47% |

Table 1: The most used CPU instructions in the FORTH system software in the 8 bit databus configuration of the N.I.G.E. Machine that was presented at EuroFORTH 2012. The most frequently used instruction, LOAD.W, is one of the minority of instructions that have low throughput (less than one instruction per clock cycle).

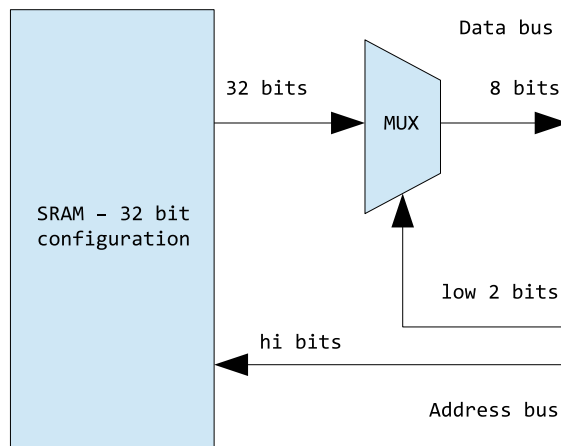


Figure 2: Illustrative scheme for reading individual bytes from 32 bit width RAM. There is no equivalently simple, single step scheme for writing individual bytes to 32 bit width RAM.

2.4 Design complications resulting from a 32 bit wide databus

When Block RAM is configured in longword (32 bit) data format, each memory address references a separate, complete longword. Consecutive memory addresses therefore step through memory in units of 4 bytes at each increment. The first concern in designing a wider databus are the mismatches that arise between the 4 byte wide data format and the width of each instruction (1 byte) and the smallest unit of memory access (also 1 byte). These mismatches have a number of important design implications.

The first implication is that the address bus cannot directly reference memory at the level of individual bytes or 16 bit words. This is at odds with the design requirement that the CPU should be able to read or write to memory either longwords, words or bytes. For memory read instructions this problem can be circumvented by creating a composite address bus whereby the high bits of the composite address bus are matched directly with the Block RAM address bus, and the low 2 bits are used to multiplex individual bytes from within the relevant longword (fig 2).

For write instructions there is, however, no simple solution because when configured in longword format, FPGA Block RAM writes complete longwords without the flexibility to specify only individual words or bytes within them. (Note however that a full review of alternative FPGA device families was not made and this limitation may not apply to higher-end or more recent devices.)

The second implication is the problem of address boundaries. Suppose that the CPU wishes to read 4 consecutive bytes from memory. Two different situations arise (fig. 3). In the first case, the address of the first byte happens to coincide with a longword address within memory. This is aligned access and can be accomplished with a single read instruction to that memory address, with a duration of one clock cycle. However in the second case, the desired 4 bytes may be spread across two consecutive longword addresses in memory. This is non-aligned access and it requires the CPU to execute two read instructions from the two consecutive memory addresses, thus taking two clock cycles. Either that, or non-aligned access must be prohibited by the CPU specification.

Neither of these constraints are attractive. Prohibiting non-aligned memory access, especially at the longword level, decreases the flexibility available to the programmer and the FORTH compiler, and

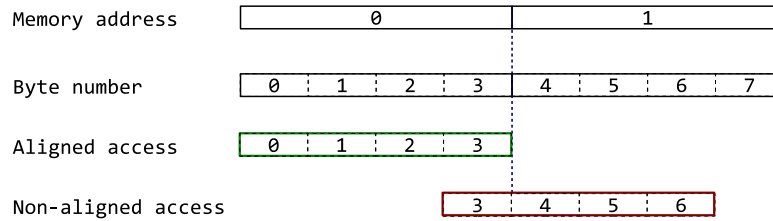


Figure 3: Aligned and non-aligned memory access operations require different treatment.

wastes space in memory. Requiring the CPU to switch between single cycle and two cycle memory access modes depending on address alignment would mean that instruction execution would no longer be deterministic, thus introducing jitter into signals being generated in real time. Fixing the duration all memory read accesses at the worst case of two-cycles would solve the problem of non-deterministic execution, but at the expense of halving the throughput of all load/store instructions.

These issues are a particular concern for FORTH processors for the reasons mentioned earlier. The real-time control applications of a FORTH processor mean that deterministic execution is often sacrosanct. At the same time, FPGA based softcore processors are inevitably clocked a much lower frequencies than general purpose CPU's and so high instruction throughput is essential. FPGA Block RAM is also usually limited to tens or hundreds of kilobytes and FORTH has a natural advantage in the very small code size of its applications compared with other high level languages. For memory efficiency reasons, byte level memory access and byte sized instruction coding is therefore also highly desirable.

Fortunately by leveraging a particular capability of FPGA Block RAM it is possible to design a 32 bit wide memory architecture which circumvents almost all of these constraints. That FPGA Block RAM feature is **dual ported memory access**. It is possible to configure FPGA Block RAM with two independent address and data buses that read or write to separate memory locations in the same clock cycle. (In the Xilinx Spartan 3 FPGA family dual ported Block RAM memory access is a standard feature available at no additional cost, however a full review was not made to determine if that also generally applies to other device families.) The 32 bit N.I.G.E. Machine memory architecture leverages dual ported Block RAM to provide a 32 bit wide memory databus whilst maintaining byte level memory access, deterministic execution and single cycle throughput for most instructions.

3 Design for 32-bit wide system memory access

Figure 4 shows the N.I.G.E. Machine system diagram in 32 bit databus configuration. The design is described in detail below.

3.1 SRAM memory controller

The key component in the N.I.G.E. Machine's 32 bit memory datapath is the SRAM controller that sits between the CPU and dual-ported SRAM, as shown in fig 5. It provides byte, word and longword read/write access to system memory.

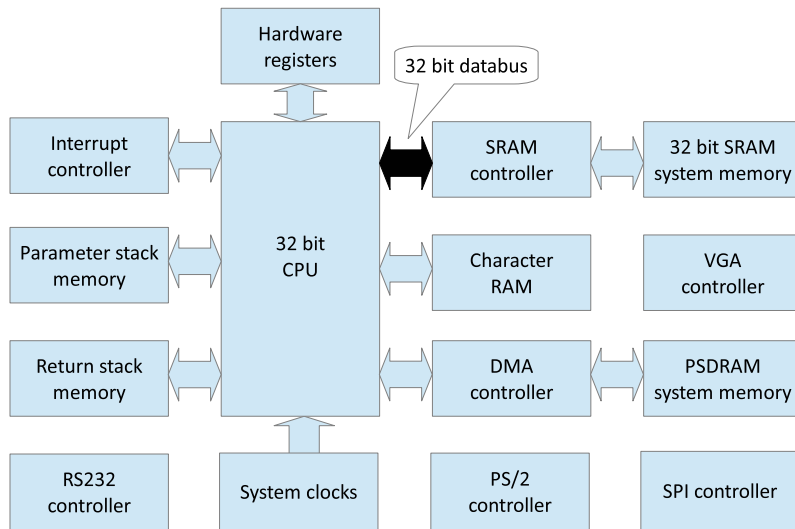


Figure 4: N.I.G.E. Machine system diagram showing the principal components and CPU connections. This is the 32 bit databus configuration, as highlighted.

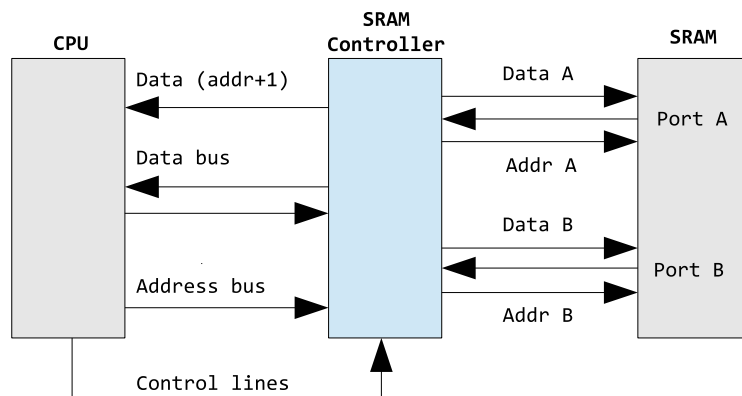


Figure 5: Block diagram of the SRAM memory controller showing the connection to dual ported Block RAM

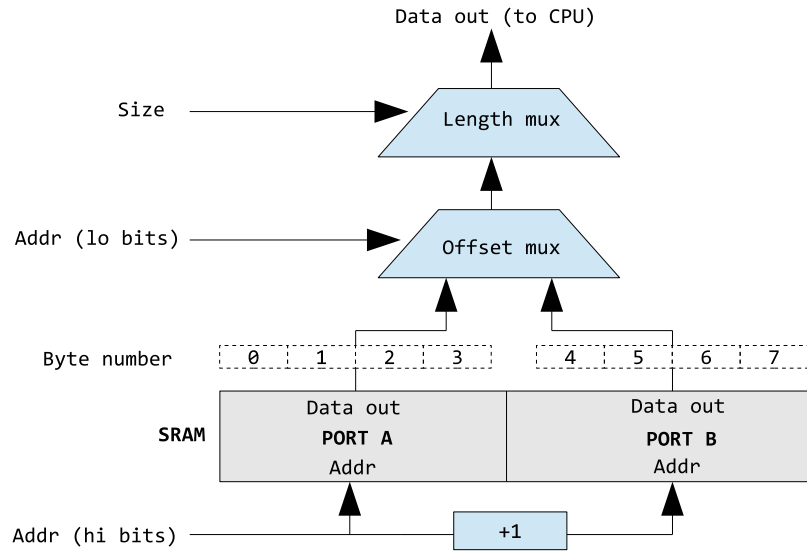


Figure 6: Details of the SRAM controller read functionality. The offset multiplexer select the appropriate longword from the 8 bytes available from SRAM ports A and B. The length multiplexer either passes through the longword or left pads a word or byte output with zero bits, according to the selected size.

3.1.1 Read functionality

The functionality of the SRAM controller during a memory read is shown in fig. 6. The memory address provided by the CPU points to an individual byte address in memory. The SRAM controller splits this address into two parts: the lowest 2 address bits and the remaining (high) address bits. The high address bits point to the longword address within which the selected byte address lies. The lowest 2 address bit can be interpreted as the offset of the byte address from the zeroth byte of the longword address. The SRAM controller passes the high address bits directly to SRAM port A. It also adds 1 to the high bits address (equivalent to adding 4 to the byte level address) and passes this address to SRAM port B. One clock cycle later, the SRAM read operations occur on both ports simultaneously, and the SRAM controller will have a total of eight contiguous bytes available to it from SRAM ports A and B combined. The offset multiplexer selects four contiguous bytes out of these eight according to the lowest 2 bits of the address specified by the CPU. Finally, the size multiplexer takes a 2 bit control signal from the CPU control unit and selects either a single byte, a single word, or the full longword from the output of the offset multiplexer. In the case of selecting a byte or a longword, the multiplexer shifts the relevant bits to the low end of the output longword and pads the high end with zero.

Table 2 illustrates some worked examples of the SRAM controller read functionality.

3.1.2 Write functionality

The functionality of the SRAM controller during memory write mode is shown in fig. 7. Essentially a write to SRAM now takes place over two cycles, during which time the CPU must hold the address and data constant on the memory bus. In the first cycle, the existing contents of SRAM memory at the relevant addresses are read and multiplexed with the write data from the CPU. Multiplexing takes into account both the memory address offset, and the size of the data being presented by the CPU (longword, word, byte). The result is that the appropriate overlay of the CPU write data

| | CPU address | CPU size request | Port A address | Port B address | Offset | Memory bytes at output |
|----|-------------|------------------|----------------|----------------|--------|------------------------|
| A) | 0 | longword | 0 | 1 | 0 | [00][01][02][03] |
| B) | 0 | word | 0 | 1 | 0 | [-][-][-][00][01] |
| C) | 7 | longword | 1 | 2 | 3 | [07][08][09][10] |
| D) | 7 | byte | 1 | 2 | 3 | [-][-][-][-][07] |

Table 2: Worked examples of SRAM controller read functionality. In case (A) the CPU is reading a longword from memory address 0. The first four bytes in memory appear at the output, in big endian format. In the case (B), the CPU is also reading from memory address zero, but a word. In this case the size multiplexer has shifted the word at memory address zero to the low end of the output databus and filled the high bits with zero (indicated [-] in the table). Cases (C) and (D) illustrate a read from memory address 7. In these cases port A reads longword memory address 1 (byte memory address 4) and port B reads longword memory address 2 (byte memory address 8). The offset of 3 selects a longword starting at third byte on port A.

onto the existing memory contents becomes available at the end of the first cycle. In the second cycle the outputs of the multiplexers are written to SRAM.

As with SRAM read functionality both of the dual SRAM ports are active. The low two bits of the address presented by the CPU form the address offset used by the multiplexers, while the high address bits are used to access two consecutive longwords in SRAM. A single cycle delay on the write enable signal from the CPU defers the SRAM write to the second cycle.

3.1.3 Dual data output

In addition to providing non-aligned (byte addressable) longword access for any given memory address, the SRAM controller is also configured to output the memory contents at the next following address. This databus is labeled “Data (addr + 1)” in fig. 5. The purpose of this data is to expedite the execution of load literal CPU instructions. The format of a load literal instruction is a single instruction byte followed by a longword, word, or byte of data. By making available to the CPU the contents at the next memory address beyond the current instruction, the literal data can be multiplexed directly into the datapath during single cycle execution of a load literal instruction.

3.2 Datapath

Minimal changes were required to the CPU datapath design to accommodate the 32 bit memory databus since the datapath width is already 32 bits.

In the 8 bit databus configuration of the N.I.G.E. Machine, memory read data is made available to the 32 bit datapath on a 32 bit accumulator register that is managed by the control unit finite state machine. The register functions to accumulate the required data byte by byte over the required number of memory read clock cycles. In the 32 bit databus design memory the accumulator is not required because 32 bit data is available directly from the SRAM controller.

The datapath also has direct access to the Block RAM’s that hold the parameter and return stacks. The stack databuses are always 32 bits wide. The memory holding the parameter and return stacks is dual ported and is also available to the CPU over the system memory databus. When Block RAM is configured with a 32 bit databus on one port (in this case for direct stack access) and with an 8 bit databus on the other port (in this case for system memory access in 8 bit databus format) the Xilinx memory layout is little endian by default. The N.I.G.E. Machine is big endian format and hence the 4 bytes of the longword must be reversed when read over the stack databus.

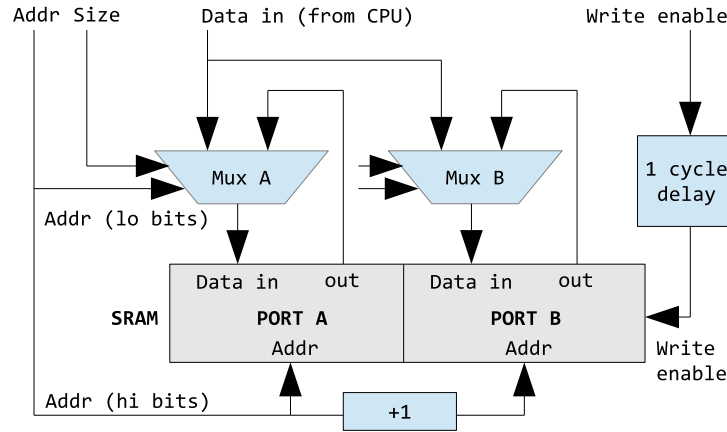


Figure 7: Details of SRAM controller write functionality. A write operation takes place over two cycles. In the first cycle the existing memory contents are read and overlaid at the appropriate position with the data from the CPU. In the second cycle the memory contents are updated.

This is a minor detail that does not affect performance, but the complexity is avoided in the 32 bit databus configuration.

3.3 Control unit

The control unit required a more considerable redesign to accommodate the 32 bit databus and optimize instruction execution.

3.3.1 Program counter logic

The principal impact of the 32 bit memory datapath on the control unit is that the program counter (“PC”) logic needs to be reconfigured to process variable length instructions that execute in a single cycle. In the N.I.G.E. Machine instruction set instructions longer than a single byte only occur when literal data is provided in the second and subsequent bytes (the instruction itself is always fully specified by the initial byte). These include the load literal and branch instructions.

In the 8 bit databus configuration, variable length instructions are executed over several cycles with each successive instruction byte being read from memory in successive cycles. The program counter is therefore hardwired to step in units of a single byte in all circumstances except when a branch or jump occurs. This considerably simplifies the program counter control logic. In the 32 bit datapath format, instructions that are longer than one byte and which include literal data also need to be executed in a single cycle. Therefore the program counter logic must decode the length of each instruction and advance by the relevant number of bytes during a single cycle. This process for PC update is as follows (fig 8).

Firstly the program counter logic must determine the location of the next instruction (i.e. the instruction that will be executed after the currently executing instruction). The control unit is configured so that the currently executing instruction is always found as the byte at the lowest memory address on the 32 bit datapath. (This is the highest order byte of the whole longword on a

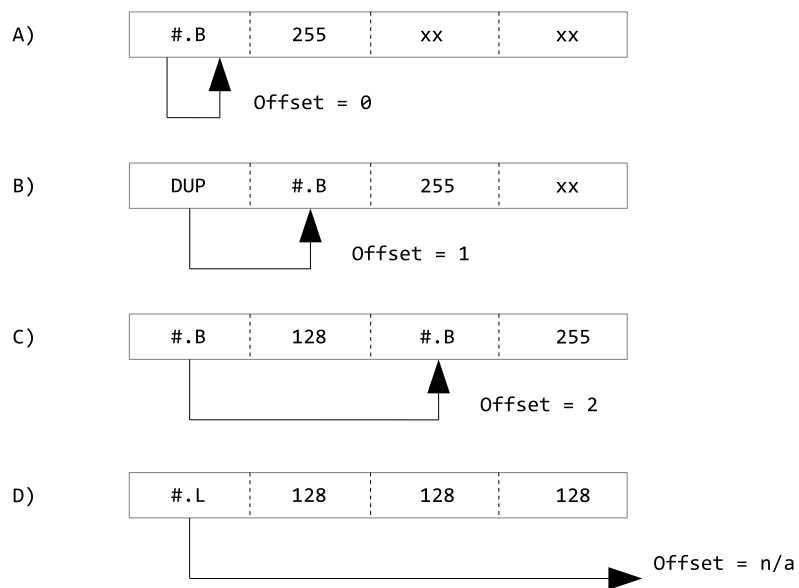


Figure 8: Identification of the next instruction byte on the databus by the program counter logic. In case (A) there is no currently executing instruction and the offset to the next instruction is zero. The next instruction is a load literal byte instruction of length 2. In case (B) the currently executing instruction is one byte in length and the offset to the next instruction is one byte. In case (C) the currently executing instruction is a load literal byte instruction of length 2 bytes, and this is also the offset to the next instruction. Case (D) illustrates that when the currently executing instruction is load literal longword of length 5 the next instruction lies beyond the width of the 4 byte databus and the offset cannot be calculated.

| Component / clock cycle | Cycle #0 | Cycle #1 | Cycle #2 | Cycle #3 | Cycle #4 |
|-------------------------------------|----------|----------|----------|----------|----------|
| Program counter | 0 | 0 | 2 | | |
| Offset | | 0 | | | |
| Next instruction byte | | 53 | | | |
| Length of next instruction | | 2 | | | |
| Instruction byte | | | 53 | | |
| Opcode | | | 53 | | |
| Microcode | | | | 1191 | |
| Datapath combinatorial logic | | | | 255 | |
| Datapath synchronous logic register | | | | | 255 |

Figure 9: Illustration of the execution pipeline for the CPU instruction to load a literal byte with value 255. Executing variable length instructions in a single cycle requires an extra stage in the pipeline (clock cycle #1 in this illustration).

big endian machine such as the N.I.G.E. Machine.) The size of the currently executing instruction is also always known to the control unit finite state machine and made available as an output (labeled as the “offset”, fig 8). The program counter logic refers to the offset to identify which byte of the longword corresponds to the next instruction. For example, the majority of instructions are encoded as single bytes and so the next instruction is the next byte. The load literal byte and word instructions are two or three bytes in length respectively and so the next instruction is two or three bytes ahead respectively. Offsets are not calculated for branch or jump instructions since these require that the PC be diverted rather than incremented.

Once the instruction byte of the next instruction has been identified, that instruction byte is multiplexed to the second stage of the PC logic which determines the instruction length. Finally, in the third stage of the PC logic, the length of the next instruction is added to the current value of the PC, so the the PC will be appropriately updated in the next cycle.

3.3.2 Four stage pipeline

Implementation of the program counter logic is made more complex by the fact that the N.I.G.E. Machine CPU is a pipelined design. As a result the program counter needs to decode the instruction length before, and independently of, the rest of the instruction execution logic. This requires an extra stage at the beginning of the pipeline, which is now 4 stages long as illustrated in fig. 9. The pipeline stages are:

1. “READ INSTRUCTION SIZE”. In the example of fig. 9 the pipeline is being started afresh (following a jump, branch or reboot) and there is no currently executing instruction. The “offset” is therefore zero. During clock cycle #1 the PC logic identifies the next instruction byte according to the scheme described above. In this case 53, corresponding to the load literal byte instruction which is 2 bytes long.
2. “FETCH OPCODE”. On the rising edge of clock cycle #2, SRAM system memory reads the instruction byte at the current PC address and extracts its opcode. A “new” PC address is determined by adding to the PC the instruction size increment calculated in the previous cycle, in this case 2 bytes.
3. “DECODE AND COMPUTE”. On the rising edge of clock cycle #3, SRAM microcode memory within the control unit takes the opcode as a lookup address and returns the corresponding microcode value (1191). During the same clock cycle the combinatorial logic in

| Instruction | Mnemonic | Cycle count (8 bit databus) | Cycle count (32 bit databus) |
|--|---------------------|-----------------------------|------------------------------|
| Load literal byte | LOAD.B (or #.B) | 2 | 1 |
| Load literal word | LOAD.W (or #.W) | 3 | 1 |
| Load literal longword* | LOAD.L (or #.L) | 5 | 2 |
| Branch (conditional or unconditional)* | BEQ / BRA | 3 | 3 |
| Jump to subroutine (address on stack)* | JSR | 2 | 3 |
| Jump to subroutine (literal address)* | JSL | n/a | 3 |
| Return from subroutine* | RTS | 2 | 3 |
| Fetch/store byte in SRAM* | FETCH.B / STORE.B | 2 | 3 |
| Fetch/store word in SRAM* | FETCH.W / STORE.W | 3 | 3 |
| Fetch/store longword in SRAM* | FETCH.L / STORE.L | 5 | 3 |
| Fetch/store in PSDRAM | FETCH.[] / STORE.[] | variable | variable |
| Multiply (signed/unsigned) | MULTS / MULTU | 6 | 6 |
| Divide (signed/unsigned) | DIVS / DIVU | 43/42 | 43/42 |
| ?dup | IFDUP | 2 | 2 |
| All other instructions | | 1 | 1 |

Table 3: Clock cycles per instruction in the N.I.G.E. Machine softcore CPU in both 8 bit and 32 bit databus configurations. Instructions marked * require a restart of the pipeline following their execution. Most, but not all, instructions are faster in the 32 bit configuration.

the datapath is configured according to the microcode value through its control signals. The value of the datapath computation becomes available as the combinatorial output, in this example the literal value loaded is 255.

4. “SAVE”. On the rising edge of clock cycle #4, the output of the datapath in combinatorial logic (i.e. the result of the computation in the previous pipeline stage) is written into the synchronous logic register that holds the value of the top of stack.

3.3.3 Instruction throughput

The number of clock cycles required to execute each instruction in both the 8 bit and 32 bit databus configurations of the N.I.G.E. Machine is scheduled in table 3. The differences in throughput between the two configurations results from two opposing factors. (i) The 32 bit databus configuration reduces the number of clock cycle required to execute instructions that load, fetch, or save access word and longword data in SRAM system memory because of the greater bandwidth. However, (ii) the additional pipeline stage adds an extra cycle to all instructions that require the restart of the pipeline.

The following analysis of instruction throughput speaks from the perspective of the 32 bit datapath configuration and the changes made from the 8 bit format.

- The load literal byte and load literal word instructions now execute in a single cycle. However the load literal longword instruction actually requires two cycles to execute. This is because the length of that instruction (5 bytes) means that whilst it is being executed, the next instruction byte is not visible on the datapath to the PC update logic (fig 8), and hence an extra cycle must be added with no instruction to restart the pipeline.

- Branches also require a restart of the pipeline because of the change to the PC. However the extra cycle this entails is offset by the fact that the whole two byte instruction can be read and decoded in a single cycle. As a result the total execution cycle count is unchanged at 3 cycles.
- The JSR instruction is now one cycle longer due to extra cycle to restart the pipeline. This might imply a considerable performance penalty in executing FORTH code, which is heavily subroutine dependent. However the 32 bit databus configuration permits the inclusion of a new instruction, JSL, that provides a considerable efficiency. This instruction is a “jump to subroutine” with the subroutine address specified as a 24 bit literal value. Previously, the typical FORTH code to execute a subroutine branch comprised (i) `#.W`, to load the subroutine address onto the stack, followed by (ii) JSR. This combination requires a total of 5 cycles. The JSL instruction accomplishes the same result in 3 cycles. However the advantage of 2 cycles on a subroutine call is offset by the fact that an RTS instruction is also one cycle longer due to the lengthened pipeline. The net difference is that subroutine calls are now one cycle faster overall. (As a side note, the introduction of the JSL instruction did not necessitate significant rewriting of the N.I.G.E. Machine system software. The system software is written in assembly language, and the macro assembler implements either style of subroutine call with a macro, “CALL”, appropriate to whichever version of the hardware is being compiled for.)
- Fetches and stores to SRAM system memory of all datasizes now execute in three cycles, compared with two, three, and five cycles for byte, word and longword data previously. In FORTH terms, `C@` and `C!` are slower than before, `W@` and `W!` are unchanged, and `@` and `!` are faster than before.
- Other instructions that do not access SRAM system memory and do not restart the pipeline are unchanged. Fetch and store to the external pseudo-static dynamic RAM (“PSDRAM”) takes place through the PSDRAM controller and timing depends on the arbitration of the bus with other users of PSDRAM memory such as the VGA controller.

In summary, whilst for most instructions the softcore CPU throughput has been increased in the 32 bit datapath design, it is clear that there have been tradeoffs in certain cases. To assess whether the 32 bit datapath configuration should be expected to lead to higher performance overall, it is also necessary to examine the frequency of instruction usage. Table 4 schedule the most frequently used instructions in the N.I.G.E. Machine system software (i.e. the FORTH operating system). Despite the fact that some instructions are slower in the 32 bit datapath configuration, taking into account which instructions are most common, the data suggest that average instruction throughput should be improved in this configuration.

4 Results

4.1 Hardware implementation

Implementation of the 32 bit datapath format on the Nexys 2 FPGA development board proved more challenging than anticipated. Whilst synthesis was satisfactory in the electronic simulator, the new design was not able to complete place and route to meet the timing constraints of a 50MHz clock speed. This was in spite of considerable optimization work with the FPGA design tools. The reason for the slower timing in the 32 bit bus configuration was revealed by analyzing the place and route results and identifying the longest signal path. This signal path is the logic required to operate the variable instruction length program counter logic in the control unit. The steps involved are shown schematically in fig 10. As described in the section discussing the control unit,

| Instruction | Frequency | Clock cycle difference |
|--------------------|------------------|-------------------------------|
| JSL | 10.2% | -2 |
| #.W | 9.8% | -2 |
| RTS and ,RTS | 8.7% | +1 |
| BRA and BEQ | 8.7% | 0 |
| #.B | 7.3% | -1 |
| DUP | 4.8% | 0 |
| DROP | 4.1% | 0 |
| FETCH.L | 3.5% | -2 |
| FALSE | 3.3% | 0 |
| SWAP | 3.3% | 0 |
| OVER | 3.2% | 0 |
| STORE.L | 3.1% | -2 |
| R> | 2.6% | 0 |
| + | 2.6% | 0 |
| FETCH.B | 2.6% | +1 |
| >R | 2.1% | 0 |
| STORE.B | 1.9% | -1 |

Table 4: Relative instruction frequency for the 80% most common instructions in the N.I.G.E. Machine system software (counted by code frequency rather than execution frequency). The clock cycle difference values are the instruction duration difference in moving from the 8 bit to 32 bit databus configuration. Negative values indicate that the 32 bit configuration is faster. The most used instruction, JSL, is 2 cycles faster in the 32 bit databus configuration than the previous equivalent.

| | N.I.G.E. Machine (8 bit databus) | N.I.G.E. Machine (32 bit databus) |
|----------------------------------|---|--|
| Best achievable timing (ns) | 18.22 | 22.97 |
| Equivalent clock frequency (MHz) | 54.89 | 44.84 |

Table 5: Best achievable synthesis and place and route timing for the N.I.G.E. Machine on the Nexys 2 development board with a Xilinx XC3S1200E FPGA. The 32 bit databus configuration fails to make timing for a 50MHz clock speed.

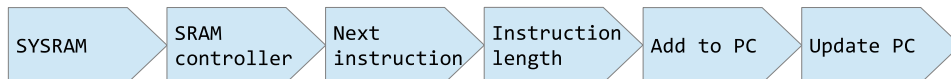


Figure 10: Schematic of the longest signal path in the N.I.G.E. Machine 32 bit databus configuration that is responsible for limiting the best achievable timing to more than 20ns

these steps are necessary if the N.I.G.E. Machine is to execute variable length instructions in a single clock cycle.

Table 5 summarizes the best achievable timing of the N.I.G.E. Machine in both 8 bit and 32 bit databus formats. The best achievable clock frequency in 32 bit format was 44MHz. However it is not possible to operate the N.I.G.E. Machine at arbitrary clock frequencies (say 40MHz), because there are also timing constraints imposed by the peripheral components. In particular the VGA controller should operate at 25MHz or 50MHz on account of to the VGA signal specification, and the system clock needs to be synchronized at a multiple of the VGA clock frequency. (Note that this limitation is a consequence of the design of the VGA controller in the N.I.G.E. Machine rather than a limitation of the Xilinx Spartan 3 FPGA device family or the VGA specification in general.)

For the purpose of comparative benchmarking, the 32 bit databus configuration N.I.G.E. Machine was successfully implemented at 50MHz by reducing the depth of SRAM memory to 4K only. (By reducing memory depth, the number of Block RAM multiplexers is reduced and therefore also the signal time for an SRAM read. The saving was enough to compensate). However this workaround has limited scope, since whilst the benchmarks can be run in under 4K of memory, this restriction in memory size is too severe for a general purpose microcomputer.

4.2 Performance benchmarks

A series of benchmarks were run to compare the performance of the N.I.G.E. Machine in both the 8 bit and 32 bit databus configurations at 50MHz. The benchmarks were based on a number of standard FORTH tests [12], minimally adapted to run in an embedded environment. As a baseline comparison, the benchmarks were also run on a Intel i7 desktop PC at 2.8GHz using VFX FORTH. Tables 6 and 7 show the results.

The N.I.G.E. Machine in 32 bit databus format is on average 14% faster in primitive operations and 20% faster in applications than in 8 bit databus format. However the speed increase varies according to the application. Eratosthenes's sieve is only 4% faster while the eight queens problem is 29% faster. This is because not all instructions are faster in the 32 bit datapath format and so the instruction mix of an application is also important.

The N.I.G.E. Machine in 32 bit databus configuration is approximately 150x slower on average than an Intel i7 PC running VFX FORTH, but again the range varies from 120x for random numbers to

| Benchmark | Iterations | N.I.G.E. Machine (8 bit bus) | N.I.G.E. Machine (32 bit bus) | PC i7 VFX FORTH |
|-------------------------|------------|------------------------------------|-------------------------------------|-----------------------|
| Primitives | | ms | ms | ms |
| DO LOOP | 1,000,000 | 260 | 260 | - |
| + | 1,000,000 | 340 | 300 | - |
| * | 1,000,000 | 440 | 420 | - |
| / | 1,000,000 | 1,240 | 1,200 | |
| /MOD | 1,000,000 | 1,240 | 1,200 | 16 |
| */ | 1,000,000 | 1,420 | 1,400 | - |
| Array fill (1000 items) | 1,000,000 | 9,008 | 7,207 | 15 |
| | | 13,948 | 11,987 | 31 |
| Applications | | | | |
| Eratosthenes sieve | 3000 | 19,680 | 18,884 | 110 |
| Fibonacci recursion | 1 | 44,272 | 37,947 | 265 |
| Quick sort | 1,000 | 10,924 | 9,171 | 31 |
| Random numbers | 1,000 | 48,795 | 35,643 | 296 |
| Bubble sort | 100 | 41,089 | 33,387 | 218 |
| Eight queens | 50 | 37,774 | 26,968 | 141 |
| | | 202,534 | 162,000 | 1,046 |

Table 6: Benchmark timing results for the N.I.G.E. Machine in 8 bit and 32 bit datapath configurations, and the same tests run on an Intel i7 PC using VFX FORTH.

almost 300x for quicksort. However the N.I.G.E. Machine was clocked at 50MHz while the PC was clocked at 2.8GHz, a difference of 56x. Allowing for the difference in clock speeds, the N.I.G.E. Machine was only 3x slower on average than the PC.

5 Discussion

At the outset it was expected that the main challenge in widening the memory datapath from 8 to 32 bits would be to design appropriate logic to maintain deterministic execution, instruction throughput, byte-sized instruction format, and flexible memory access. Two major components had to be developed to accomplish these objectives. Firstly, an SRAM controller that leveraged the dual ported Block RAM available on the FPGA. Secondly, an adaption to the control unit to facilitate the execution of variable length instructions in a single clock cycle.

However these components were included at the expense of additional logic levels and a longer signal path. This reduced the maximum achievable clock frequency. Whilst the 32 bit memory databus configuration completes benchmarking tests in approximately 20% less clock cycles than the 8 bit configuration, the maximum clock frequency that can be achieved at implementation is also roughly 20% less (~40 MHz c.f. ~50 MHz)

Perhaps in retrospect this tradeoff should have been anticipated. It is similar to the tradeoff between the RISC (reduced instruction set computing) and CISC (complex instruction set computing) approaches to CPU design, and occurs for similar reasons. RISC designs utilize less logic but can operate at a higher clock speed compared to CISC designs that have more sophisticated instruction set functionality.

| Benchmark | N.I.G.E. | N.I.G.E. |
|---------------------|-------------------------------|-----------------------------|
| | Machine (32 bit) / (8 bit) | Machine (32 bit) / PC i7 |
| | % | multiple |
| Eratosthenes sieve | 96% | 172 |
| Fibonacci recursion | 86% | 143 |
| Quick sort | 84% | 296 |
| Random numbers | 73% | 120 |
| Bubble sort | 81% | 153 |
| Eight queens | 71% | 191 |
| Total | 80% | 155 |

Table 7: Relative benchmark timing results for the 32 bit datapath format N.I.G.E. Machine compared to the 8 bit datapath format N.I.G.E. Machine and a PC i7 running VFX FORTH . The 32 bit datapath configuration is on average ~20% faster than the 8 bit configuration.

One avenue for further consideration might be to review alternative device families to determine whether FPGA’s that incorporate Block RAM with a byte select feature are available, given the utility that would have with a 32 bit databus. However, an overarching aim of the N.I.G.E. Machine is to use only low-cost, ubiquitous hardware and consequently the design preference in general is to work around inherent limitations rather than “up-spec”.

The question is then, which approach is more appropriate for a FORTH softcore such as the N.I.G.E. Machine? A “CISC like”, 32 bit databus with the ability to execute variable length instructions in a single cycle, resulting in instruction execution that completes in few clock cycles. Or a “RISC like”, 8 bit databus matched to the instruction size with fewer layers of logic, resulting in a higher implementable clock frequency?

The answer to this dilemma may be to look more carefully within the CPU at the differing needs of the control unit and the datapath. The datapath within the N.I.G.E. Machine’s softcore CPU is 32 bits wide. Matching the 32 bit datapath to a 32 bit memory databus optimizes execution speed by maximizing data transfer bandwidth. On the other hand, the control unit can operate at a higher clock speed when there is no need to execute variable length instructions in a single cycle. A hybrid design can be envisaged that maintains the 32 bit memory databus matched to the 32 bit datapath, but reverts to a control unit that processes instructions on a byte-by-byte basis. Such a hybrid design might have the following characteristics:

- Maximum clock speed no slower than the 8 bit databus configuration (i.e. 50MHz)
- Fetch/store instructions execute approximately as fast as with the pure 32 bit databus configuration
- The fast JSL (jump to subroutine literal address) instruction is included
- The pipeline could revert to 3 stages, eliminating the extra clock cycle restart penalty of the 4 stage pipeline
- Load literal instructions would have a throughput of less than one instruction per clock cycle

Based on the results discussed above, such a hybrid design is likely to prove a better performer than either the pure 8 bit or 32 bit databus configurations. Development along these lines is an attractive avenue for further work on the N.I.G.E. Machine.

6 Conclusion

This project set out to widen the N.I.G.E. Machine's memory databus from 8 bits to 32 bits. In doing so it was found that neither configuration is absolutely better than the other. The tradeoffs between them concern maximizing clock speed versus maximizing instruction throughput. This result parallels the differences between the RISC and CISC approaches to CPU design. In the case of the N.I.G.E. Machine, a hybrid memory databus that addresses the differing needs of the CPU control unit and datapath is likely to be the optimum performance strategy. Further work will be undertaken on the N.I.G.E. Machine to implement such an approach.

Whilst it is recognized that differing processor designs have differing design tradeoffs at a detailed level, some general conclusions about the strategy for optimizing memory access design for a 32 bit FORTH processor can be drawn from these project results. A FORTH processor is likely to be optimized for the efficient execution of a basic set of stack and memory operations, subject to embedded control objectives such as deterministic execution and high code density. Maximum clock speed is achieved with simple control unit logic. Given these considerations, it is likely desirable to match the width of the memory databus to the control unit to the width of a single instruction (8 bits on the N.I.G.E. Machine). On the other hand, a FORTH processor is a fetch/store architecture and so data bandwidth will be maximized by matching the width of the memory databus to the width of the CPU datapath (32 bits on the N.I.G.E. Machine). The best overall approach is therefore likely to adopt a hybrid databus design, whereby the needs of the CPU control unit and datapath are separately identified and addressed.

The author would like to thank the anonymous academic reviewers for their comments and suggestion, all of which have helped to improve the paper.

References

- [1] The author, <http://www.youtube.com/watch?v=0v-HuVLRoUc>
- [2] The author, "The N.I.G.E. Machine: an FPGA based micro-computer system for prototyping experimental scientific hardware", in *EuroForth*, 2012
- [3] James Bowman , "J1: a small Forth CPU Core for FPGAs" in *EuroForth*, 2010
- [4] K. Schleisiek, "MicroCore," in *EuroForth*, 2001.
- [5] B. Paysan, "b16-small – Less is More," in *EuroForth*, 2004.
- [6] E. Hjrtland and L. Chen, "EP32 - a 32-bit Forth Microprocessor," in Canadian Conference on Electrical and Computer Engineering, pp. 518–521, 2007.
- [7] E. Jennings, "The Novix NC4000 Project," *Computer Language*, vol. 2, no. 10, pp. 37–46, 1985.
- [8] Rible, John, "QS2: RISCing it all," Proceedings of the 1991 FORML Conference, Forth Interest Group, Oakland, CA (1991), pp. 156-159.
- [9] Stephen Pelc, "Programming FORTH", MPE, 2011
- [10] P. J. Koopman, Jr., "Stack computers: the new wave", Halsted Press, 1989
- [11] The author, Github open source repository <https://github.com/Anding/N.I.G.E.-Machine>
- [12] MPE benchmark suite for 32 bit Forth systems, <http://www.mpeforth.com/arena/benchmrk.fth>

EuroForth 2013

Forth Query Language (FQL) - Implementation and Experience

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
4-5 Great Western Court
Ross-on-Wye
Herefordshire
HR9 7XP
UK
Tel. +44 1989 768080
Email njn@micross.co.uk

Abstract

We will demonstrate how a hard problem in SQL becomes easy when combining SQL and Forth using FQL - "Forth Query Language".

1. Introduction

Databases are at the root of many computer applications, and almost all databases are programmed using SQL. In a previous paper (EuroForth 2006) I introduced the concept of a highly efficient method of switching between Forth and SQL, so as to achieve the maximum benefits from both. We have now had several years experience of FQL and can report on its success.

2. Features of SQL

SQL stands for "Structured Query Language" and this is possibly the most deceptive name ever devised by a computer scientist. SQL is very difficult to structure, can do other things besides queries, and is not even a language in the sense that a Forth programmer would understand. It is a "declarative" language, in that one describes the output that one needs, but not the method used for obtaining that output. It cannot be used by itself, and always needs some program written in another language for an interface.

However, for the tasks that it was originally designed for, it is extremely simple and effective. It is only when one tries to stretch it beyond its natural capabilities, that it becomes hard to manage.

3. Features of Forth

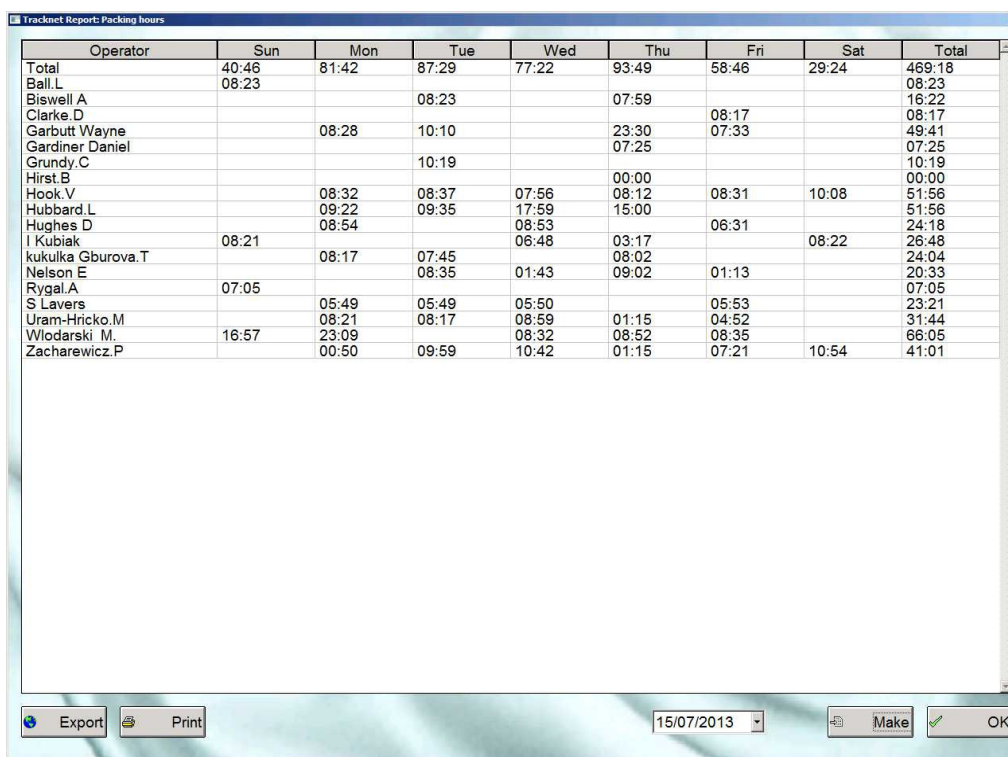
Forth, as we all know, is the outstanding language for creating highly structured and maintainable large programs. However, it is not particularly good at creating large and complex strings, which is what is required to generate SQL queries.

4. A very hard problem in SQL

Although SQL has functional extensions (if for example IF... statements have been added), almost any problem which requires a partly functional solution, is difficult to code in tidy SQL.

Here is an example which is simple to describe, but not so simple to code. Suppose we have a table which records the time when operators start and stop working at a particular work zone. We want to produce a report like this:

We are looking at the total hours spent at this workzone, over a period of a week, and an analysis by day, and by operator.



| Operator | Sun | Mon | Tue | Wed | Thu | Fri | Sat | Total |
|-------------------|-------|-------|-------|-------|-------|-------|-------|--------|
| Total | 40:46 | 81:42 | 87:29 | 77:22 | 93:49 | 58:46 | 29:24 | 469:18 |
| Ball L | 08:23 | | | | | | | 08:23 |
| Biswell A | | | 08:23 | | 07:59 | | | 16:22 |
| Clarke D | | | | | | 08:17 | | 08:17 |
| Garbutt Wayne | | 08:28 | 10:10 | | 23:30 | 07:33 | | 49:41 |
| Gardiner Daniel | | | | | 07:25 | | | 07:25 |
| Grundy C | | | 10:19 | | | | | 10:19 |
| Hirst B | | | | | 00:00 | | | 00:00 |
| Hook V | | 08:32 | 08:37 | 07:56 | 08:12 | 08:31 | 10:08 | 51:56 |
| Hubbard L | | 09:22 | 09:35 | 17:59 | 15:00 | | | 51:56 |
| Hughes D | | 08:54 | | 08:53 | | 06:31 | | 24:18 |
| I Kubiak | 08:21 | | | 06:48 | 03:17 | | 08:22 | 26:48 |
| kukulka Gburowa.T | | 08:17 | 07:45 | | 08:02 | | | 24:04 |
| Nelson E | | | 08:35 | 01:43 | | 01:13 | | 20:33 |
| Rygal.A | 07:05 | | | | | | | 07:05 |
| S Lavers | | 05:49 | 05:49 | 05:50 | | 05:53 | | 23:21 |
| Uram-Hricko.M | | 08:21 | 08:17 | 08:59 | 01:15 | 04:52 | | 31:44 |
| Wlodarski M. | 16:57 | 23:09 | | 08:32 | 08:52 | 08:35 | | 66:05 |
| Zacharewicz.P | | 00:50 | 09:59 | 10:42 | 01:15 | 07:21 | 10:54 | 41:01 |

This appears to be deceptively simple. You just subtract each operator's log on time from his log off time, and add them all up.

Unfortunately, some people work night shifts, and also move to other workzones from time to time. Even so, the algorithm is fairly easy to understand. If we just extract all log on and log off times for each day, we can pair them. Any unpaired times can then be paired with the start and end of the day, as appropriate. Then we can calculate the sum of differences.

It will be seen that while the data extraction, and the sum of differences, are very easy in a declarative language such as SQL, the pairing and above all the treatment of unpaired times, are extremely hard. On the other hand, a functional language such as Forth can easily deal with the pairing.

5. Structuring and factoring a query using FQL

If we could easily switch between SQL and Forth, we could choose which part of a problem to allocate to which language. This is what FQL achieves.

In the above example, we first extract from the table a list of times:

```
: PH-RAWQUERY { st -- parray } \ Get raw data into array
SQL|
SELECT opref, opaction, TIME_TO_SEC(TIME(logsysteme)),
DAYOFWEEK(logsysteme), DATEDIFF(logsysteme,NOW())
FROM operlog
WHERE | CH-LOGONOFF | \ Machine log on or off
AND | st CH-WEEK | \ Monday to Sunday
AND | PH-WORKZONES | \ Sorting workzones
ORDER BY opref,logsysteme
|SQL> CH-INSERTARRAY \ Insert results into array
;
```

The word SQL| starts to create an SQL query, until the next | is encountered, when we switch back to Forth. We then stay in Forth until another | is encountered, when we switch back to SQL. Finally, |SQL> executes the query leaving a pointer to the resulting array. That result is then placed in a temporary table in memory, ready for the pairing operations.

Note the other big advantage of FQL - we have introduced Forth structuring into SQL code, making it much easier to read. Each part of the "WHERE" clause is easy to distinguish. For example, "CH-WEEK" calculates, in Forth, the dates of the previous Sunday and the following Saturday, given any specified date, and inserts these dates into the SQL code as a "BETWEEN" clause.

6. Subquery reuse in FQL

Having carried out our pairing and sums of differences, we can then feed the result into a temporary database table. The final SQL query is still fairly complex:

```
: PH-HOURS { st -- } \ SQL query that lists operator hours at packing area by week
|
| SELECT IFNULL(operator.name,'Total') AS 'Operator name',
| TIME_FORMAT((SEC_TO_TIME(sunday.minutes *60)),('%H:%i')) AS Sun,
| TIME_FORMAT((SEC_TO_TIME(monday.minutes *60)),('%H:%i')) AS Mon,
| TIME_FORMAT((SEC_TO_TIME(tuesday.minutes *60)),('%H:%i')) AS Tue,
| TIME_FORMAT((SEC_TO_TIME(wednesday.minutes*60)),('%H:%i')) AS Wed,
| TIME_FORMAT((SEC_TO_TIME(thursday.minutes *60)),('%H:%i')) AS Thu,
| TIME_FORMAT((SEC_TO_TIME(friday.minutes *60)),('%H:%i')) AS Fri,
| TIME_FORMAT((SEC_TO_TIME(saturday.minutes *60)),('%H:%i')) AS Sat,
| TIME_FORMAT((SEC_TO_TIME(total.minutes *60)),('%H:%i')) AS Tot
| FROM      | st PH-OPERATORS | AS employee
| LEFT JOIN | 0 CH-MINUTES    | AS sunday   ON employee.opref = sunday.opref
| LEFT JOIN | 1 CH-MINUTES    | AS monday   ON employee.opref = monday.opref
| LEFT JOIN | 2 CH-MINUTES    | AS tuesday  ON employee.opref = tuesday.opref
| LEFT JOIN | 3 CH-MINUTES    | AS wednesday ON employee.opref = wednesday.opref
| LEFT JOIN | 4 CH-MINUTES    | AS thursday ON employee.opref = thursday.opref
| LEFT JOIN | 5 CH-MINUTES    | AS friday   ON employee.opref = friday.opref
| LEFT JOIN | 6 CH-MINUTES    | AS saturday ON employee.opref = saturday.opref
| LEFT JOIN | -1 CH-MINUTES   | AS total    ON employee.opref = total.opref
| LEFT JOIN operator
| ON employee.opref = operator.opref
| ORDER BY operator.name
|
| ;
```

This illustrates a further useful feature of FQL. "CH-MINUTES" generates an SQL sub-query which is re-used multiple times. The complete query is in fact rather long and it is not easy to see the structure when written out in full. Expressing it in Forth makes the structure clear.

"CH-MINUTES" takes a parameter which generates a slightly different sub-query each time.

```
: CH-MINUTES { pday -- } \ Inserts the time subquery
|
| ( SELECT IFNULL(opref,0) as opref, SUM(pmins) AS minutes
| FROM sortlog
| WHERE | pday CH-DAY |
| GROUP BY opref WITH ROLLUP
| )
|
| ;
```

Within this word, "CH-DAY" either introduces a "WHERE" clause for each day, or a "WHERE 1=1" type clause, for the totals.

```

: CH-DAY { pday -- } \ Inserts day clause, 0=select all
  | (pday= | pday -1 <> IF
    pday ZFORMAT
  ELSE
    Z"" pday"          \ Select all gives pday=pday
  THEN >SQL | ) |
;

```

Note that we could have introduced two Forth DO...LOOPS for the days of week, which would have made the code more compact. However, I think the readability is probably better without the loops.

7. Making FQL thread safe

Because SQL queries often take an appreciable time to execute, they are usually carried out in separate threads of execution. Furthermore, in an automation application, such as our flagship program "Tracknet", there may be many database operations taking place simultaneously, in different threads. Making FQL thread safe is simply a matter of assigning a private SQL connection handle and query string pad, for each thread.

```

STRUCT _TASKPRIVS
  PADSIZ2   FIELD .OWNPAD2   \ PAD2 and PAD3 for that thread
  PADSIZ3   FIELD .OWNPAD3
  MAXQUERYSIZE FIELD .OWNSQLPAD \ SQL pad for thread
  CELL      FIELD .OWNHSQL   \ Handle of SQL connection
  MAX_PATH  FIELD .OWNCURRPATH \ Current directory for that thread
  CELL      FIELD .VAHDTASK   \ AHD task
  CELL      FIELD .VAHDPROG   \ AHD progress
END-STRUCT

```

```

: SQLPAD ( ---addr ) \ SQL pad owned by a thread
  GET-TASK-PRIVATE .OWNSQLPAD
;

: HTHREADSQL ( ---addr ) \ Handle to SQL connection owned by a thread
  GET-TASK-PRIVATE .OWNHSQL
;

```

8. Comparison of code size

We noted the code lengths, when we converted a particular query from an older technique (see Swialowski, EuroForth 2006) to FQL. The old code required 136 lines, despite the lines being longer and difficult to read. The FQL solution required 89 lines, despite the lines being deliberately shortened to improve reliability. Typically, the old solution required approximately 50% more code than FQL.

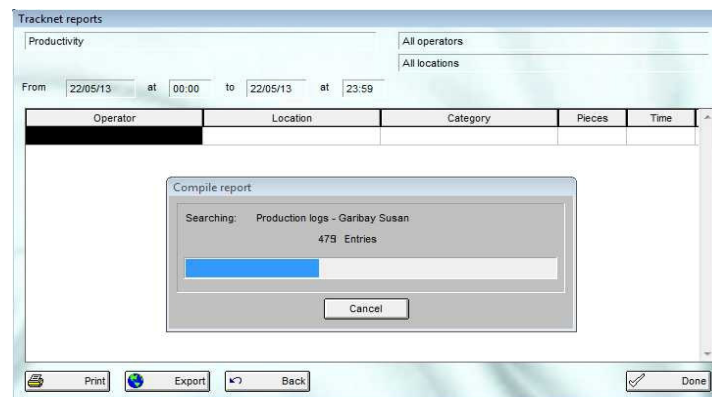
9. Comparison of performance

Very significant performance improvements can sometimes be achieved by using FQL instead of pure SQL. The most dramatic improvements are observed when the SQL code contained extensive use of procedural "enhancements" for which SQL was never originally designed.

Extreme example: Operator efficiency query.

| | |
|----------|------|
| Pure SQL | 86s |
| FQL | 4.5s |

A further advantage is that, because the SQL is broken up into lots of smaller queries, it is possible to support a plausible "progress" bar and cancel button, for time consuming operations.



10. Conclusion

By combining the best features of both SQL and Forth, FQL creates code that is faster, more compact, more readable and easier to maintain.

NJN

September 2013

References:

1. The Nearly Invisible Database or ForthQL
N.J. Nelson, EuroForth 2006
2. Database access for illiterate programmers
K.B.Swiatlowski, EuroForth 2006

PAF: A portable assembly language*

M. Anton Ertl[†]
TU Wien

Abstract

A portable assembly language provides access to machine-level features like memory addresses, machine words, code addresses, and modulo arithmetics, like assembly language, but abstracts away differences between architectures like the assembly language syntax, instruction encoding, register set size, and addressing modes. Forth already satisfies a number of the characteristics of a portable assembly language, and is therefore a good basis. This paper presents PAF, a portable assembly language based on Forth, and specifically discusses language features that other portable assembly languages do not have, and their benefits; it also discusses the differences from Forth. The main innovations of PAF are: tags indicate the control flow for indirect branches and calls; and PAF has two kinds of calls and definitions: the ABI ones follow the platform’s calling convention and are useful for interfacing to the outside world, while the PAF ones allow tail-call elimination and are useful for implementing general control structures.

1 Introduction

Traditionally compilers have produced the assembly language for the various target architectures, and interpreters were written in assembly language. The disadvantage of this approach is that it requires retargetting for every new architecture. As a result, many such compilers and interpreters target only one or few architectures, and ports to new architectures often take quite a while.¹

*An slightly shorter version of this paper appears at KPS 2013; I recommend the present version.

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

¹E.g., AMD64 CPUs became available in 2003; The *lina* interpreter for AMD64 became available in 2008, the *iForth* compiler became available for AMD64 in 2009 (and 32-bit releases were stopped at the same time), and other significant Forth compilers like *SwiftForth*, *VFX*, and *bigForth* still do not offer 64-bit support in 2013. By contrast, the *Gforth* interpreter which uses a portable assembly language, was available there right from the start (thanks to our portable assembly language being there from the start), and we verified that by building and testing *Gforth* on an AMD64 system in August 2003.

Portable assembly languages promise to solve this problem: The source language compiler (the *front end*) compiles to (or the interpreter is written in) the portable assembly language, and the compiler or interpreter will work on a variety of architectures without extra effort. Of course the portable assembly language implementation has to be targeted for these architectures, but that effort can be reused (and possibly the cost shared) by several compilers/interpreters.

In this paper we present a new portable assembly language, PAF (for “Portable Assembly Forth”). There have been a number of languages that been designed and/or used as portable assembly languages (Section 2), so why introduce a new one?

1.1 Contributions

An issue that a number of portable assembly languages have had is that they require the code to be organized in functions that follow the standard calling convention (ABI) of the platform, which usually prevents tail-call optimization. PAF provides ABI calls and definitions for interfacing with the rest of the world, but also PAF calls and definitions, which (unlike ABI calls) can be tail-call-optimized and can therefore be used as universal control flow primitives [Ste77] (see Section 3.10 and 3.11).

Another problem is that indirect branches and calls have a high cost, because the compiler has to assume that every branch/call can reach any entry point. PAF introduces tags to specify which branches/calls can reach which entry points (see Section 3.9 and 3.10).

The most significant difference between PAF and Forth is that PAF contains restrictions that ensure that the stack depth is always statically determinable, so stack items can be mapped to registers (Section 3.3 and 3.9). It is interesting that these restrictions are relatively minor and don’t affect much Forth code; it’s also interesting to see an example of Forth code that is affected (see Section 5).

2 Previous Work

This section discusses existing portable assembly languages, their features and deficiencies and why we feel the need for a new one.

2.1 C

C and its dialects, like GNU C, have been used as a portable assembly language in many systems: It is the prevalent language for writing interpreters (e.g., Python, Ruby, Gforth) and run-time systems; C has also been used as target language for compilers: (e.g., the original C++ compiler `cfront`, and one of the code generation options of GHC).

However, the C standard specifies a large number of “undefined behaviours”, including things that one expects to behave predictably in a portable assembly language, e.g., signed integer overflow. In earlier times this was not a problem, because the C compilers still did what the programmer intended. Unfortunately, a trend in recent years among C compiler writers has been to “optimize” programs in such a way that it miscompiles (as in “not what the programmer intended”) code that earlier compiler versions used to compile as intended. While it is usually possible to find workarounds for such a problem, the next compiler version often produces new problems, and with all these workarounds the direct relation from language feature to machine feature is lost.

Another problem of C (and probably a reason why it is not used as often as compiler target language as for interpreters) is that its control flow is quite inflexible: Code is divided into C functions, that can be called and from which control flow can return; the only other way to change control flow across functions is `longjmp()`.

Varargs in combination with other language features have led to calling conventions where the caller is responsible for removing the arguments from the stack. This makes it impossible to implement guaranteed tail-call optimization, which would be necessary to use C calls as a general control flow primitive [Ste77].

As a result, any control flow that does not fit the C model, such as unlimited tail calls, backtracking, coroutines, and even exceptions is hard to map to C efficiently.

2.2 LLVM

LLVM is an intermediate representation for compilers with several front ends, optimization passes and back ends [LA04].

Unfortunately, it shares many of the problems of C: In particular, you have to divide the code into functions that follow some calling convention, restricting the kind of control flow that is possible. To work around this problem, it is possible to add your own calling convention, but that is not easy.²

²Usenet message <KYGdnTH8PMpM7MnZ2dnUVZ_j-dnZ2d@supernews.com>

LLVM was also promised to be a useful intermediate representation for JIT compilers, but reportedly its code generation is too slow for most JIT compiler uses.

LLVM supports fewer targets than C. Given that it also seems to share many of the disadvantages of C, it does not appear to be an attractive portable assembly language to me, despite the buzz it has generated.

2.3 C--

C-- [JRR99] has been designed as portable assembly language. Many considerations went into its design, and it appears to be well-designed, if a little too complex for my taste, but the project appears to be stagnant as a general portable assembly language, and it seems to have become an internal component of GHC (called `Cmm` there).

While C-- does not appear to be an option as portable assembly language for use in practical projects at the moment, looking at its design for inspiration is a good idea.

2.4 Vcode and GNU Lightning

Vcode [Eng96] is a library that provides a low-level interface for generating native code quickly (10 executed instructions for generating one instruction) and portably. It was part of a research project and has not been released widely, but it inspired GNU Lightning, a production system.

The demands of extremely fast code generation mean that GNU Lightning cannot perform any register allocation on its own. Therefore the front end has to perform the register allocation. It also does not perform instruction selection; each Lightning instruction is translated to at least one native instruction.

GNU Lightning also divides the code into functions that follow the standard calling convention, and one can call functions according to the calling convention. However, it is also possible to implement your own calling conventions and other control flow, because the front end is in control of register allocation, but (from reading the manual) it is not clear if this can be integrated with the stack handling by GNU Lightning and if one can use the processor’s call instruction for your own calling convention.

It is possible to use better code generation technology with the GNU Lightning interface, and also to provide ways to use the processor’s call and return instructions for your own calling convention.

With these changes, wouldn’t the GNU Lightning interface be the perfect portable assembly language? It would certainly satisfy the basic requirements of a portable assembly language, but as a

replacement for a language like C, it misses conveniences like register allocation.

3 Portable Assembly Forth (PAF)

3.1 Goals

- Portability: Works on several different architectures
- Direct relation between language feature and machine feature, i.e., if you look at a piece of PAF code, you can predict what the machine code will look like.

However, the relation between PAF and the machine is not as direct as for GNU Lightning: There is register allocation and instruction selection, there may be instruction scheduling, and code replication. Instruction selection and instruction scheduling make better code possible (at the cost of slower compilation); register allocation interacts with these phases, and leaving it to the clients would require duplicated work in the clients, as register allocation is not really language-specific.

- Capabilities of the (user-mode part of the) machine can be expressed in PAF. However, this goal is moderated by the needs of clients and by the portability goal. I.e., PAF will at first only have language features that compilers and interpreters are likely to need (features can be added when clients need them); and machine features of particular architectures that cannot be abstracted into a language feature that can be implemented reasonably on all the intended target machines will not be supported, either.

3.2 Target machines

While a portable assembly language can abstract away some of the differences between architectures, there are differences that are too difficult to bridge, and would lead PAF too far away from the idea of a direct correspondence between language feature and architectural feature, so here we define the class of machines that we target with PAF:

PAF targets general-purpose computer architectures, i.e., the architectures that have been designed as compiler targets, such as AMD64, ARM, IA-32, IA-64, MIPS, PowerPC, SPARC.

Memory on the target machines is byte-addressed with a flat address space; e.g., DSPs with separate X and Y address spaces are not target machines. The target machines use modulo (wrap-around) arithmetics and signed numbers are represented in 2s-complement representation.

The target machines have a uniform register set for integers and addresses (not, e.g., accumulators with different size than address registers), and possibly separate (but internally also uniform) floating point registers.

3.3 Forth and PAF

Forth's low-level features are quite close to assembly language; e.g., like in assembly language, neither the compiler nor the run-time system maintains a type system, and the language differentiates between different operations based on name, not based on type; e.g., Forth has < for signed comparison and U< for unsigned comparison of cells (machine words), just like MIPS has `slt` and `sltu`, and Alpha has `cmplt` and `cmpult`.

Therefore Forth is a good basis for a portable assembly language. However, there are features that are problematic in this context: In particular, in Forth the stack depth is not necessarily statically determined (unlike in the JVM), even though in nearly all Forth code the stack depth is actually statically determined (known to the programmer, but not always the Forth system). So we change these language features for PAF.

A number of higher-level features of Forth are beyond the goal of a portable assembly language, so PAF does not support them.

On the other hand, there are a few things that are missing in standard Forth that have to be added to PAF, such as words for accessing 16-bit quantities in memory.

3.4 Example

The following example shows two definitions written in PAF:

```

                                \ cml %edx,%eax
: max                            \ jle L28
  2dup >? if                    \ ret
  drop exit endif              \ L28:
nip exit ;                      \ movl %edx,%eax
                                \ ret
abi:xx- printmax {: n1 n2 -- :}
  "max(%ld,%ld)=%ld\n\0" drop
  n1 n2 2dup max abi.printf.xxxx-
  exit ;

\ Call from C:
\ main() { printmax(3,5); return 0; }
```

The first, `max`, looks almost like conventional Forth code, and corresponding assembly language code for IA-32 is shown in comments to the right. `max` does not have a fixed calling convention; the PAF compiler can set a calling convention that is appropriate for `max` and its callers (e.g., it can be

tail-called). Since `max` does not follow the platform's calling convention, it cannot be called from, e.g., C code.

The second definition, `printmax`, follows the standard ABI of the platform (as indicated by using an `abi:` defining word. The `xx-` in `abi:xx-`³ shows that `printmax` expects and consumes two cells from the data stack and 0 floats from the FP stack and produces 0 cells and 0 floats; a C prototype for this definition could be `void printmax(long, long)`. `Printmax` calls `max`, and the compiler can choose the calling interface between the call and `max`; it calls `printf` using the standard calling convention with the call `abi.printf.xxxx-`, where the `xxxx-` indicates that four cells are passed as integer/address parameters and the return value of `printf` is ignored.

Locals are used in `printmax` but can be used in every definition. Exiting from the definitions is explicit.

3.5 Registers

Several language features correspond to real machine registers: Stack items, locals, and values.

Stack items (elements) are useful for relatively short-lived data and (unlike locals) can be used for passing arguments and return values. There is no stack pointer and memory area specific to the stack, it's just an abstraction used by the compiler. Stack manipulation words like `DUP` or `SWAP` just modify the data flow and there is no machine code that directly corresponds to them (indirect consequences may be, e.g., move instructions at control flow joins).

Locals live within a definition and are a convenience: Local variables of the source language can be mapped directly to PAF's locals without needing register allocation or stack management in the front end. If a source local needs to be distributed across several PAF definitions (e.g., because a control structure of the source language is mapped to a PAF (tail) call), the local can be defined in each of these definitions, and the constants are passed on the stack across calls; this is not as convenient as one might like, but seems to be a good compromise.

Values are global (thread-local) variables whose address cannot be taken, so they can be stored in registers.

If stack items and locals don't fit in the registers, they are stored in a stack that is not visible to PAF

³This paper assumes the use of a recognizer feature in the Forth system to process parameterized names; the conventional Forth way would be to use a parsing word, in this case, e.g., `abi: xx- printmax`.

code; this stack stores items from the data and FP stack, locals, and return addresses, so this does not correspond to the memory representation of, e.g., the data stack.⁴

If values don't fit in the registers, they are stored in global/thread-local memory.

3.6 Memory

The words `c@ uw@ ul@ (addr -- u)` load unsigned 8/16/32-bit values from memory, while `sc@ w@ l@ (addr -- n)` load signed 8/16/32-bit values from memory; `@ (addr -- w)` loads a cell (32-bit or 64-bit, depending on the machine) from memory; `sf@ df@ (addr -- r)` load 32/64-bit floating-point values from memory. `c! w! l! ! (x addr --)` and `sf! df! (r addr --)` store stack items to memory.

3.7 Arithmetics

The usual Forth words `+` `-` `*` `negate` and `or` `invert` `lshift` `rshift` correspond to the arithmetic and logic instructions present in every machine. There are also additional words like `/ m*` `um*` `um/mod` `sm/rem` that correspond to instructions on some machines, and have to be synthesized from other instructions on other machines.

3.8 Comparison

The words `=?` `<?` `u<?` `f=?` `f<?` etc. compare two stack items and return 0 for false and 1 for true. They correspond to the Forth words `=` `<` `u<` `f=` `f<` etc., with the difference that the Forth words return `-1` (all-bits-set) for true. A number of machines have instructions that produce 0 or 1 (MIPS, Alpha, IA-32, AMD64), while for others it is as easy to produce 0 or 1 as to produce 0 or `-1`, so "0 or 1" is more in line with the goal of the direct relation to the machine feature. An implementation of a 0-or-`-1` language like Forth would use a sequence like `<? negate` for which good code can be generated easily.⁵

3.9 Control flow inside definitions

The standard Forth words `begin` `again` `until` `ahead` `if` `then` `cs-roll` are available in PAF

⁴Some languages have local variables whose address can be taken; it may be a good idea to provide a way to store them in this stack eventually, but for now such variables have to be stored elsewhere. The interaction of such a feature with, e.g., tail calls has to be considered first.

⁵Conversely, one might also decide to have `<` etc. instead of `<?` in PAF, and let the compiler handle the mismatch to some machines, but that would be somewhat against the spirit of a portable assembly language.

and are useful for building structured control flow, such as `if ... then ... elseif ... then ... else ... end`.

While one can construct any control flow with these words [Bad90], if you want to implement labels and gotos, it's easier to use labels and gotos. Therefore, PAF (unlike Forth) provides that, too: `L:name` defines a label and `goto:name` jumps to it.

PAF also supports indirect gotos: `'name/tag` produces the address of label `name`, and `goto/tag` jumps to a label passed on the stack. The tag indicates which gotos can jump to which labels; a PAF program must not jump to a label address generated with a different tag. E.g., a C compiler targeting PAF could use a separate tag for each `switch` statement and the labels occurring there.

These tags are useful for register allocation. One can use different tags when taking the address of the same label several times, and this may result in different label addresses, with the code at each target address matched to the gotos that use that tag (i.e., several entry points for the same PAF label).

Whichever method of control flow you use, on a control flow join the statically determined stack depth has to be the same on all joining control flows. This ensures that the PAF compiler can always determine the stack depth and can map stack items to registers even across control flow. This is a restriction compared to Forth, but most Forth code conforms with this restriction. Breaking this rule is detected and reported as error by the PAF compiler.

So the tags have another benefit in connection with the stack-depth rule: The static stack depth for a given tag must be the same (for all labels and all gotos), but they can be different for different tags. If there were no tags, all labels and gotos in a definition would have to have the same stack depth.

3.10 PAF Definitions and PAF calls

A definition where the compiler is free to determine the calling interface is defined in the classical Forth way:

```
: name ... exit ;
```

The end of the definition does not produce an implicit return (unlike Forth), so you have to return explicitly with `exit`.

You call such a definition by writing its name, i.e., the traditional Forth way. You can explicitly tail-call such a definition with `jump:name`; this can be written explicitly, in the spirit of having a portable assembly language. Optimizing implicit tail calls is not hard, so the PAF compiler may do it, too.

We can take the address of a definition with `'name:tag`, call it with `exec.tag` and tail-call it

with `jump.tag`. The tags indicate which calls can call which definitions.

The stack effects of all definitions whose address is taken with the same tag have to be compatible. I.e., there must be one stack effect that describes all of them; e.g., `(x x -- x)` is a valid stack effect of both `+` and `drop` (although the minimal stack effect of `drop` is `(x --)`), so `+` and `drop` have compatible stack effects.

The use of tags here has two purposes: It informs the PAF compiler about the control flow; and it also informs it about the stack effect of the indirect call (while a Forth compiler usually has to assume that `execute` can call anything, and have any stack effect). Or conversely, in connection with the stack-depth rule: Tags allow different stack effects for indirectly called definitions with different tags; without tags, all indirectly called definitions would have to have the same stack effect.

3.11 ABI definitions and ABI calls

We need to specify the stack effect explicitly as signature of an ABI definition or call. The syntax for such a signature is `[xr]*-[xr]*`, where `x` indicates a cell (machine word/integer/address) argument, and `r` a floating-point argument; the letters before the `-` indicate parameters, and the letters afterwards the results. The division into `x` and `r` reflects the division into general-purpose registers and floating-point registers on real machines, and the role these registers play in many calling conventions.

A definition conforming to the calling convention is defined with `abi:sig name`. `Sig` specifies the stack effect, and indicates the correspondence between ABI parameters and PAF stack items. This signature is not quite redundant, e.g., consider the difference between the following definitions:

```
abi:x-x id exit ;
abi:- noop exit ;
```

These definitions differ only in the signature, yet they behave differently: `id` returns its argument, `noop` doesn't, and with ABI calling conventions, there is usually a difference between these behaviours.

You can call to an ABI-conforming function with `abi.name.sig`, where `name` is the name of the function (which may be a PAF definition or a function written in a different language and dynamically or statically linked with the PAF program). The signature specifies how many and which types of stack items to pass to the called functions, and what type of return value (if any) to push on the stack.

Putting the signature on every call may be a bit repetitive for human programmers, but PAF is mainly intended as an intermediate language, and

an advantage of this scheme is that different calls to the same function (e.g., `printf`) can have different stack effects.

You can take the address of an ABI function with `abi_name` and call it with `abi-exec.sig`. There are no tail calls to ABI functions, because we cannot guarantee that tail calls can be optimized in all calling conventions.

Unlike PAF definitions, for ABI functions there is no point in tagging these function addresses, because the call always uses the ABI calling convention (whereas the compiler is free to determine the calling interface for PAF calls). The signature in indirect ABI calls has the same significance as in direct ABI calls.

3.12 Definitions and Calls Discussion

Why have two kinds of definitions and two kinds of calls?

The PAF definitions and calls allow to implement various control structures such as backtracking through tail calls [Ste77]. They also allow the compiler to use flexible and possibly more efficient calling interfaces than the ABI calling convention.

On the other hand, the ABI counterparts allow interfacing with other languages and using dynamically or statically linked binary libraries, including callbacks, and using PAF to build such libraries (e.g., as plug-ins).

3.13 Exceptions

It is possible to build non-local control-flow such as exceptions with tail-calls, but it is often more convenient to let a PAF definition correspond to a source language function/method/procedure (no need to spread locals across several definitions). Exceptions are a common non-local control-flow construct, so PAF includes them.

4 Non-Features

This section discusses various features that PAF does not have and why.

4.1 Garbage collection

A number of virtual machines, e.g., the Java VM, support garbage collection. However, this feature significantly restricts what can be done. In particular, the data representations are restricted, and one cannot implement “unmanaged” languages or use a different data representation for a garbage collected language (e.g., the Java VM representation is quite different from how most Prolog or Lisp systems represent their data).

Even C--, which is intended as a portable assembly language for garbage collected languages does not implement garbage collection itself, but leaves it to the higher-level language, because that leaves the full freedom on how to implement data and garbage collection to the higher-level language [JRR99].

4.2 Types

PAF does not perform type checking during compilation, nor at run-time; also, there is no overloading of several operations on the same operator based on types. This is consistent with the descent from Forth, and non-portable assembly languages have the same approach.

In contrast, in C-- the compiler knows about data types and uses that knowledge for overloading resolution. The disadvantage of such approaches is that it complicates the C-- compiler without making life easier for the front end compiler, which has to know exactly anyway whether it wants to perform, say, signed or unsigned comparison.

One may wonder about the “absence” of some operations in PAF; e.g., there is `<? U<?`, but only `=? + - *`. The reason is that, on the two’s-complement machines that PAF targets, these operations are the same for signed and unsigned numbers.

4.3 Debugger

Quite a bit of effort in C-- is devoted to supporting the standard debugger. For now there are no plans to make such an effort for PAF. C became a successful portable assembly language even though it has very little debugger support for languages that use it as intermediate language.

4.4 SIMD

Supporting SIMD instruction set extensions such as SSE, AVX, AltiVec etc. is not planned, mainly because few higher-level languages need such features. They can be added later if there is demand.

5 PAF vs. Forth

The restrictions on stack handling in PAF provide new insights into Forth, and we take a closer look at that in this section.

5.1 Effect on implementation

PAF has restrictions and features that allow the compiler to statically determine the stack depth. As a consequence, in PAF there is no need to implement the stacks in memory, with a stack pointer for each stack (data stack and return stack for cells, floating-point stack for floating-point values).

```

\ Forth
: selector ( offset -- )
  create ,
does> ( ... o -- ... )
  @ over @ + @ execute ;

1 cells selector foo
2 cells selector bar

\ PAF
: foo ( ... o -- ... )
  dup @ 1 cells + @ jump.foo ;
: bar ( ... o -- ... )
  dup @ 2 cells + @ jump.bar ;

```

Figure 1: Defining method selectors in Forth and in PAF (simplified)

In contrast, Forth needs to have a separate memory area and stack pointer for each stack, and while stack items can be kept in registers for most of the code, there are some words (in particular, `execute`) and code patterns (unbalanced stack effects on control flow joins), that force stack items into memory and usually also force stack pointer updates.

This property of Forth is avoided in PAF by requiring balanced stack effects on control flow joins (see Section 3.9), and by replacing `execute` with `exec.tag` (see Section 3.10); all definition addresses returned for a particular tag are required to have compatible stack effects, so `exec.tag` has a statically determined stack effect.

5.2 Effect on Programs

The effect on real programs is relatively small: most Forth code has balanced stack effects for control flow anyway, and most occurrences of `'` and `execute` can be converted to their tagged variants, because programmers keep the stack depth statically determinable in order to keep the code understandable.

However, there are cases where the restrictions are not so easy to comply with. E.g., object-oriented packages in Forth use `execute` for words with arbitrary stack effects. Programs using these words have a statically determined stack effect, too, but it is only there at a higher level; e.g., if you use a separate tag (and a separate `exec.tag`) for each method selector, typical uses comply with the restriction, but in most object-oriented packages there is only one `execute`.

Figure 1 shows code for this example: the Forth variant defines a defining word `selector`, and the selectors are then defined with this defining word; in contrast, the PAF variant defines the selectors directly (and pretty repetitively), each with its own tag.

If you want to define a defining word for method selectors like you usually do in Forth, the tag would have to be passed around as a define-time parameter between the involved defining words. This support for higher-level programming is not required inside PAF (there we leave such meta-programming to the higher-level language), but if we want to transfer the tag idea back to Forth, we would have to add such things.

5.3 Compiling Forth to PAF

Translating Forth code that is not PAF code into PAF code can be instructive.

As an example, we use another variant of the selector code above⁶:

```

: do-selector ( .. obj m-off -- .. )
  over @ + @ execute ;

: foo ( .. obj -- .. )
  1 cells do-method ;

: bar ( -- )
  1 2 my-obj foo . ;

```

This is not PAF because of the `execute`, which can have an arbitrary stack effect. We translate this `execute` into a PAF `jump` with tag `forth`; we decide that the PAF calling convention for xts with that tag is `(--)`. I.e., any Forth stack effects have to be translated into accesses to an explicitly implemented memory stack in PAF. The stack pointer of the data stack is implemented as a `value sp`.

`Do-selector` itself only needs to store the stack item `obj` into this explicit stack, but the direct and indirect callers of `do-selector` usually have to access this explicit stack as well. In our example, `bar` has to push two items on the explicit stack and pop one item from the explicit stack:

```

0 value sp

: do-method
  over sp cell- tuck ! to sp
  swap @ + @ jump.forth ;

: foo
  1 cells jump:do-method ;

: bar
  sp cell- 1 over !
  cell- 2 over !
  to sp
  my-obj foo
  sp dup @ swap cell+ to sp
  jump:. ;

```

One would have to implement the floating-point stack in the same way.

Some people would like to extend standard Forth with return-address manipulation. One can also do a translation from such an extended Forth to PAF, and it shows how expensive that feature can be. Looking just at the `do-method` part of the example above:

```
0 value sp
0 value rp

: thunk1
  exit ;

: do-method
  over sp cell- tuck ! to sp
  swap @ + @
  rp cell- to rp 'thunk1:forth rp !
  exec.forth rp cell+ to rp
  jump:thunk1 ;
```

The return stack pointer has to be made explicit (as `rp`). Instead of translating the `execute` into an indirect tail call (`jump.forth`), we have to first store the return address `'thunk1:forth` on the explicit return stack, then use an indirect non-tail call `exec.forth`, then drop the return address from the explicit return stack, and then continue with the rest of the definition (`thunk1`), which just returns in this case.

6 Related work

We have discussed C, LLVM, C--, and Vcode/GNU Lightning in Section 2.

There are projects that are similar to PAF in using a restricted or modified form of a higher-level language as portable assembler:

- The Python system PyPy uses a restricted form of Python called RPython as low-level intermediate language [AACM07].
- `Asm.js`⁷ is a subset of JavaScript that is so restricted that it can serve as portable assembly language.
- PreScheme is a low-level subset of Scheme used as intermediate language for implementing Scheme48 [KR94].

In all these cases the base language is much higher-level than Forth, and it is much more of a stretch to create a low-level subset than for Forth..

⁶This variant defines a selector as a colon definition instead of with `does>`; for presentation purposes we leave the defining word `selector` away and define the selector `foo` directly instead of with `selector foo`.

⁷<http://asmjs.org/>

Machine Forth (which evolved into `colorForth`) is a simple variant of Forth created by Chuck Moore, the inventor of Forth. It closely corresponds to the instructions on his Forth CPUs, but he also wrote an implementation for IA-32 that creates native code. The IA-32 compiler is very simple, basically just expanding the words into short machine code sequences.⁸ It does not map stack items beyond the top-of-stack to registers, yet the generated code is relatively compact; this reflects the fact that machine Forth is close to the machine, including IA-32.

7 Conclusion

PAF is a subset/dialect of Forth that is intended as a portable assembly language. The main contributions of PAF are:

- *Tags* indicate which indirect branches can reach which labels and which indirect calls can call which definitions. Compared to general indirect branches and calls, this gives more freedom to the front end's stack usage and to the PAF compiler's register allocator. Tags need less implementation effort and produce better results than trying to achieve the same result through program analysis.
- Definitions and calls are split into those conforming to the ABI/calling convention of the platform, and others for which the compiler can use any calling interface (and different ones for different sets of callers and callees). This allows tail-call optimization (unlike ABI calling conventions), which in turn means that we can use the calls as a primitive for arbitrary control structures (e.g., coroutines).
- Restrictions (compared to Forth) on the use of stack items make it possible to have a static relation between stack items and registers for all programs, and avoid the need for a separate stack pointer and memory area for each stack. This highlights which Forth features are expensive and where they are used.

References

- [AACM07] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In Pascal Costanza and Robert Hirschfeld, editors, *DLS*, pages 53–64. ACM, 2007.

⁸<http://www.colorforth.com/forth.html>

- [Bad90] Wil Baden. Virtual rheology. In *FORML'90 Proceedings*, 1990.
- [Eng96] Dawson R. Engler. vCODE: A re-targetable, extensible, very fast dynamic code generation system. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 160–170, 1996.
- [JRR99] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, September 1999.
- [KR94] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.
- [LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, 2004.
- [Ste77] Guy Lewis Steele Jr. Debunking the “expensive procedure call” myth or procedure call implementations considered harmful or lambda: The ultimate goto. AI Memo 443, MIT AI Lab, October 1977.

Standardize Strings Now!

M. Anton Ertl*
TU Wien

Abstract

This paper looks at the issues in string words: what operations may be required, various design options, and why this has led to the current state of standardization of string operations that is insufficient in the eyes of many.

1 Introduction

Despite the presence of a string wordset in Forth-94, there are frequent complaints about lack of string support in Forth, and many Forth programmers design their own string library to counter this lack.

2 String operations

This section looks at the string operations present in the language AWK, which is designed for string handling, which gives us an idea of what things string words should be capable of.

AWK is a language that is designed for processing text files, extracting data from them, and outputting the data in some different format. Below we describe GNU AWK (gawk), which offers some features that other AWK variants do not have.

AWK has some language-level capabilities: It splits a file into lines/records (based on a record separator regexp), splitting a line/record into fields (based on a field separator regexp, or a field regexp); it matches lines/records with regexps and uses that to select an action to perform; the action can access the fields through the $\$n$ syntax. AWK also allows easy string concatenation by juxtaposing the two strings, and it supports strings as array indexes.

AWK also provides a number of string functions, which can be divided into several categories:

sorting `asort`, `asorti`

substitution within strings `gsub`, `gsub`, `sub`
replace patterns in arbitrary strings, `sprintf`
constructs a string from a template.

conversion `strtonum`, `sprintf`

searching `index`, `match`

information length

splitting `pat``split`, `split`

substrings `substr`

case conversion `tolower`, `toupper`

3 Design issues

This section discusses the design issues of string words in Forth.

3.1 Desirable Properties

Ease of use One property we would like strings to have is that programming with them is as easy as programming with single or double numbers, without such encumbrances as explicitly managing buffers (including avoiding buffer overflows).

Integration Another nice property is that existing words are useful for dealing with strings. E.g., we can use `2dup 2swap 2over` to handle `c-addr u` type string descriptors on the stack, `2@ 2!` for storing them, and arithmetic words for computing substrings.

As we will see, these two properties are somewhat at odds with each other.

3.2 Allocation

Manual buffer management

Who allocates string buffers, and who frees them?

This issue comes up when generating new strings, such as string concatenation, and is probably the primary issue why we have not found a consensus on a string wordset that includes words for generating new strings (not even concatenation).

One approach is that the word that produces the new string allocates it, e.g.

```
\ s+ ( c-a1 u1 c-a2 u2 -- c-a3 u3 )  
dir s" /" file s+ s+ r/o open-file throw
```

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

The usage looks cute, but it does not free the strings, and therefore is a memory leak. With proper freeing it is no longer so cute:

```
dir s" /" file s+ over >r s+ r> free throw
over >r r/o open-file throw r> free throw
```

This is one reason for disliking this approach, but a stronger one for a significant subset of the Forth community is the use of `allocate`-style allocation itself.

Embedded systems Forths do not necessarily support `allocate`, and even if they have it, one may not want to use it, because of fragmentation or performance concerns. On the other hand, just like embedded users can avoid `allocate` even though it is standardized, they can just as well avoid string creation words that `allocate`, and create strings in the way they do now. One probably won't use Forth as a scripting language on these embedded systems anyway.

Instead of allocating the string buffer in the creating word, one can pass a buffer to the word. This approach is used in `read-line` and `substitute`, and a variant of `s+` with this kind of interface looks as follows:

```
\ s+ ( c-a1 u1 c-a2 u2 c-a3 u3 -- c-a3 u4 n)
create buf1 200 chars allot
create buf2 200 chars allot
dir s" /" buf1 200 s+ 0< abort" buf short"
file buf2 200 s+ 0< abort" buf short"
r/o open-file throw
```

This does not appear attractive, either. A major problem with this approach is that it is possible to provide a too-small buffer, and in general (not for `s+`, but, e.g., for `substitute`), it is hard to know in advance how large the target buffer should be.

Automatic reclamation

So we want to avoid the problems of passing a pre-allocated buffer as well as the problems of having to free the buffers. Many other languages do this by using garbage collection. We can do that, too, and there is a garbage collector for Forth (written in standard Forth). With garbage collection, we can use the original `s+` usage example.

Requiring garbage collection as part of a string wordset is probably not going to find consensus, however. Garbage collection has a number of disadvantages: It is more complex to implement than explicit deallocation; it is most easily implemented in a stop-the-world fashion, and that does not combine well with real-time systems or multi-threading.

There has been a lot of work on making garbage collection compatible with real-time requirements and multi-threading, but the implementation cost is

significant. Also, most (all?) of this work assumes that the compiler and run-time system knows what is an address and what is not; this is generally not possible in Forth.

A practical problem with garbage collection is that, in general, garbage collection has to scan all the data memory, the stacks, and the locals to see which strings are still referenced.

This need can be reduced by always using special words to deal with strings, to keep track of string references. E.g., one might declare all memory storage for string descriptors explicitly, thus avoiding the need to scan all data memory (for dynamically allocated memory, one needs to untrack the memory in some way).

Furthermore, we could have a separate string stack with separate string stack operations, and `str@` and `str!` instructions for accessing string descriptors in memory. This approach has a low integration, though.

If the Forth system knows all the string descriptors, there are additional ways for automatic reclamation: In particular, we can use reference counting (since strings don't contain pointers, the cycle problem of general reference counting cannot occur).

Or, as a variant of that, we can use the following simple string buffer management strategy that ensures that every string only has one reference: copy the string when we copy the descriptor and free the string when we `drop` or `overwrite` the descriptor (this is inspired by Henry Baker's article on linear logic [Bak94]).

Region-based memory management

A manual reclamation method that is more convenient than `allocate/free` is region-based memory management. The program can create several regions, allocate memory in these regions, and finally free all the memory allocated in a region at once.

You typically collect data into a region if it all becomes garbage and should be freed at (mostly) the same time. E.g., in a compiler you might have a region for stuff that is relevant for a basic block and can be freed after you are done with the block, a region for stuff that is relevant for a colon definition, etc.

One nice feature of regions is that it allows the programmer to decide whether he wants to live with more not-yet-freed garbage or whether he wants to invest more programming effort and have finer-grained regions for less not-yet-freed garbage (up to having the same programming effort as `allocate/free`).

The following example shows a fine-grained use (each of the two memory allocations has a separate region), with the region passed explicitly as a parameter on the stack:


```
\ s+ ( c-a1 u1 c-a2 u2 region-id -- c-a3 u3 )
: make-path
  {: dir-a dir-u file-a file-u outer --
   path-a path-u :}
  new-region {: tmp :}
  dir-a dir-u s" /" tmp s+
  file-a file-u outer s+
  tmp free-region ;
```

Here we have two regions: `outer`, and `tmp`. We pass the id of the target region to `s+`, and once we are done with the strings in `tmp`, we free the region.

One problem with this approach is that we have to pass a region-id to any word that returns allocated memory, which causes stack juggling (avoided above by the use of locals); and that additional parameter is needed for every word that generates a string. Instead of passing the region-id explicitly, it can be passed through an implicit *current region* through a context wrapper [Ert11].

Another problem with the example above is that it is not any simpler than explicit deallocation. That's because it does exactly the same thing, and deallocates the intermediate result as soon as possible.

Here is an example where the programmer chooses to let the intermediate result hang around longer, in exchange for easier programming. E.g., if we let the the intermediate result live as long as the final result, and pass the current region implicitly, we can program `make-path` in the ease-of-use way:

```
\ s+ ( c-a1 u1 c-a2 u2 -- c-a3 u3 )
: make-path
  {: dir-a dir-u file-a file-u --
   path-a path-u :}
  dir-a dir-u s" /" file-a file-u s+ s+ ;
: open-path ( dir-a dir-u file-a file-u -- )
  new-region dup >r
  ['] make-path with-region
  r/o open-file throw
  r> free-region ;
```

The region management happen at an outer level.

Regions are an interesting idea, but have not made a big impact outside Forth; I guess most go for garbage collection if they want anything more automatic than explicit deallocation. However, given all the problems of general garbage collection, regions may be the way to go for Forth.

One widely available implementation of regions are `glibc`'s `obstacks` (which offer the additional convenience that every region can be treated as a stack).

3.3 String representation

The favoured string representation in standard Forth is `c-addr u`. It allows representing strings

of any length with any content, and you can produce arbitrary substrings without needing to copy the string to a new buffer. The disadvantage of this representation is that it takes two cells on the stack, and dealing with several strings at once can therefore be cumbersome.

The other common string representation in standard Forth is the counted string: The on-stack representation is the address of the count byte; the count byte is followed by the characters of the string. The advantage of this representation is that it needs only one cell on the stack. But it can only represent strings with up to 255 chars, and any substring operation needs to create a new string buffer. Converting from counted to `c-addr u` is easy (`count`), but the other direction is cumbersome. Some people have suggested using cell counts instead of byte counts to get rid of the length limitation.

Some people have proposed using zero-terminated strings (as in C). The on-stack representation is the address of the first character. It can represent strings of arbitrary length that don't contain a NUL char. Substring operations usually need to create a new string buffer (unless the substring is just the tail of the input string). The main advantage is that this string representation makes interfacing to some C functions easier; note that C offers `c-addr u`-compatible versions of many functions in order to be able to deal with arbitrary text; e.g., there is `fputs()` for zero-terminated strings and `fwrite()` for `c-addr u` strings.

If we go with a separate string stack and an in-memory string representation that is only accessed through string words, strings become an abstract data type, and the implementer has a choice of internal string representations. Such a representation may include such things as a reference count.

3.4 Regular expressions

Many scripting languages support searching within strings for a pattern; this is used for selecting among strings, for splitting strings into parts (with the pattern used either as separator or to specify the parts), or for replacing the patterns with replacement strings. The common practice for specifying patterns is regular expressions (regexps); there are some variations of regular expressions, and the Perl 5 variant is probably the most popular one.

All of the uses mentioned above can be implemented with the following regular expression primitive:

```
search-regexp ( c-a1 u1 c-a2 u2 --
  c-a1 u3 c-a4 u4 c-a5 u5 true | false )
```

Search for regexp `c-a2 u2` in string `c-a1 u1`; if the regexp is found, `c-a1 u3` is the substring before the first match, `c-a4 u4` is the first match, and `c-a5 u5` is the rest of the string, and the TOS is true; otherwise return false.

If you use the same regexp several times, it can be more efficient to compile the regular expression into a more readily executed form once, and then use that form repeatedly. An interface for that would be:

```
:regexp ( c-a2 u2 "name" -- )
```

Compile regular expression `c-a2 u2`, define *name* to perform the action below:

```
name execution: ( c-a1 u1 --
  c-a1 u3 c-a4 u4 c-a5 u5 true | false )
```

Search for regexp `c-a2 u2` in string `c-a1 u1`; if the regexp is found, `c-a1 u3` is the substring before the first match, `c-a4 u4` is the first match, and `c-a5 u5` is the rest of the string, and the TOS is true; otherwise return false.

3.5 Implicit parameters

The `c-addr u` representation leads to words with a lot of stack parameters, e.g., `compare`, `search` and `search-regexp`. This is often cumbersome to work with, and one may want to use some of the techniques for reducing stack depth [Ert11]. In particular, we can use implicit parameters and context-wrappers to get rid of one input and/or output string.

The obvious implicit input parameter is the parse area (`source`), and we can use the context-wrapper `execute-parsing (addr u xt --)` to put an input string in the parse area; then we need parsing variants of the words that have too many input strings. E.g., we could have a parsing variant of `search-regexp`:

```
parse-regexp ( c-a2 u2 --
  c-a1 u3 c-a4 u4 true | false )
```

Search for the regexp `c-a2 u2` in the parse area. If a match is found, `c-a4 u4` is the address of the match, and `c-a1 u3` is the string that was skipped before the match was found. The next parse starts right behind the matching string.

For string results, the implicit output parameter is the user output device; i.e., `type` is the implicit-output variant of `move`. The context-wrapper is `>string-execute (xt -- c-a u)`.

As an example, here we have a program that replaces all the occurrences of natural numbers with `<num>`, passing both input and output parameters through a context wrapper.

```
: repl-num1 ( -- )
  begin
    s" [0-9]+" parse-regexp while
      2swap type 2drop ." <num>"
  repeat
    0 parse type ;

: repl-num2 ( c-a u -- )
  ['] repl-num1 execute-parsing ;

: repl-num ( c-a1 u1 -- c-a2 u2 )
  ['] repl-num2 >string-execute ;
```

This code would be a bit tighter with quotations.

4 Conclusion

There are a number of partly conflicting requirements for string packages, in particular

- Ease of use
- Integration with the rest of Forth
- No garbage collection

The various approaches to these problems have led to a large variety of string packages, that cannot be reconciled. Yet, extending the string capabilities of Forth is a much-requested (and, in my case, often-used) feature, so we should standardize additional string capabilities at some point, although the new words will be in parallel to what various string packages offer and ideally make them redundant.

When I started working on this paper, it was unclear to me what the right approach is. Now, it seems to me that the solution is to continue in the direction that the standard string wordset has gone, and add to that:

- Use `c-addr u` as on-stack string representation.
- Add words that create new strings by allocating space for them (e.g., `>string-execute`).
- To make memory reclamation easier, add a region-based memory allocation mechanism (useful not just for strings).

- To reduce the stack depth, use implicit parameters with context-wrappers such as `execute-parsing` and `>string-execute`.
- Add a word or several for matching regular expressions.

References

- [Bak94] Henry Baker. Linear logic and permutation stacks — the Forth shall be first. *ACM Computer Architecture News*, 22(1):34–43, March 1994.
- [Ert11] M. Anton Ertl. Ways to reduce the stack depth. In *27th EuroForth Conference*, pages 36–41, 2011.

Forth in Russia

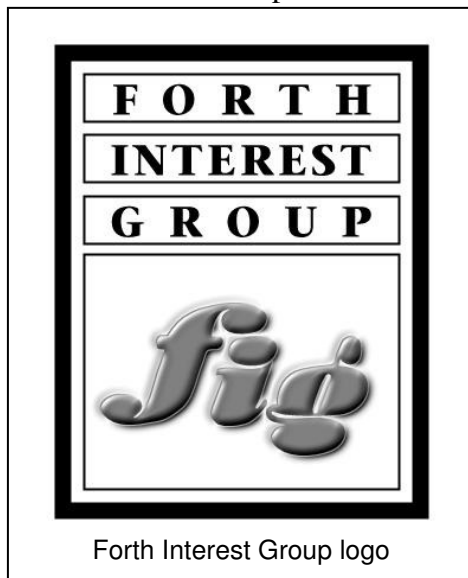
Sergey Baranov

St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences
(SPIIRAS)

SNBaranov@gmail.com

This paper provides an extended version of a presentation made by the author at the International Conference “Development of Computing and Software in the States of the Former Soviet Union and Russia” SoRuCom-2011, held at the Yaroslav the Wise Novgorod State University (Velikiy Novgorod, Russia) on 12-16 September 2011 [27]. The purpose is to highlight the major milestones of Forth expansion in Russia since its early days, the current status of the Russia Forth community, and derive certain lessons learnt with an outlook into the future.

Forth became known in the USSR since the end of 1970-ies. After its appearance in the US – the first official publication dates to 1974 – this language and associated technique of



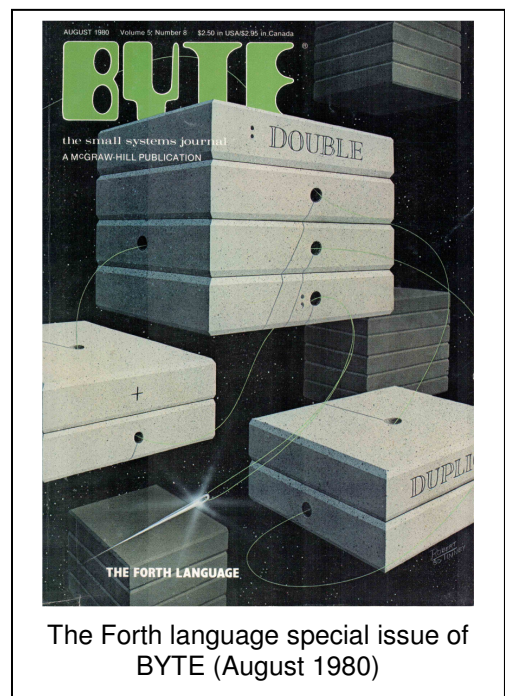
Forth Interest Group logo

programming quickly won acknowledgement as a fast and efficient means for creating meaningful applications for microprocessor machinery, where efficiency of memory footprint and small program size were often vital. The Forth Interest Group [2] was soon founded, which is active still now, with the purpose to standardize this language and make it popular among software developers.

At that time a rapid growth of microprocessors was in place, a widespread Soviet one was K580IK80 – a sound copy of Intel 8080. At the Computer Center of the Leningrad State University (now the St. Petersburg State University) a team was established with an assignment to develop software for a new Soviet microcomputer with this processor under a contract with customers from industry. The team was headed by Associated Prof. Boris Katsev, PhD; he was a well-known specialist in computing machinery, with talent, authority and organizing skills. Shortly before, he joined the University faculty after terminating his career in computer industry. Prof. Katsev staffed the team with young researchers and engineers of the Computer Center and faculty, the author being among them.

Contracts with leading Moscow industrial institutions NITSEVT and NIISCHEMASH for developing software of computer terminals with K580IK80 as the core processor were won for the University through Prof. Katsev's efforts. The mentioned institutions just started to develop such terminals for manufacturing for the whole Soviet Union. At that time the major computer facility of the Computer Center was a new mainframe ES 1030 (an analog of IBM/360) and old original Soviet ones M-220 and M-222, ending their life cycle. To test and debug software for K580IK80 the team decided to develop a cross-system for the K580 family, which included an assembler and a byte-code emulator.

PL/I was selected as an instrumental language for developing the cross-assembler, development took over a half-year. The resulting source code seemed to be enormous at that time (over 1000 lines in PL/I). All tasks running in parallel partitions of the IBM/360 compatible have to be shut down in order to provide the PL/I compiler with all available memory (less than 512K bytes at that time) for compilation of the cross-compiler code in one extended partition.



The Forth language special issue of
BYTE (August 1980)

Just at that time the team came across a copy of an article in the Dr. Dobb's Journal with a listing of an Intel 8080 assembler in Forth which took only 54 lines of text, one third of which being a table with recognizable mnemonics of Intel 8080 assembler instructions. Especially striking was the authors' claim that this was a complete assembler encompassing all modes and features of the Intel 8080 instruction set! The team spent considerable effort to clarify and understand how the assembler was done (with the CREATE-DOES> constructs); however, as soon as we got it, the power and beauty of this approach was greatly appreciated. As there were no other texts on Forth available at that time, the challenge was in grasping how this suite of Forth words worked as expected from just this listing. Only much later we accessed a special Forth issue of BYTE in 1980 [3] with many bright samples of how Forth may be used in various cases.

The most remarkable feature of Forth is its mechanism of introducing in a very elegant manner new basic language constructs, which render specifics of the task under consideration in the most appropriate way. Actually, Forth proposed a meta-language mechanism which allowed for creating new abstract data types along with their implementation at any level of abstraction, up to the machine code level. The latter allowed for reaching the maximal speed of code execution which was so important for many applications. For example, the fundamental notions of a variable and a constant may be quite elegantly introduced in just one line of text as:

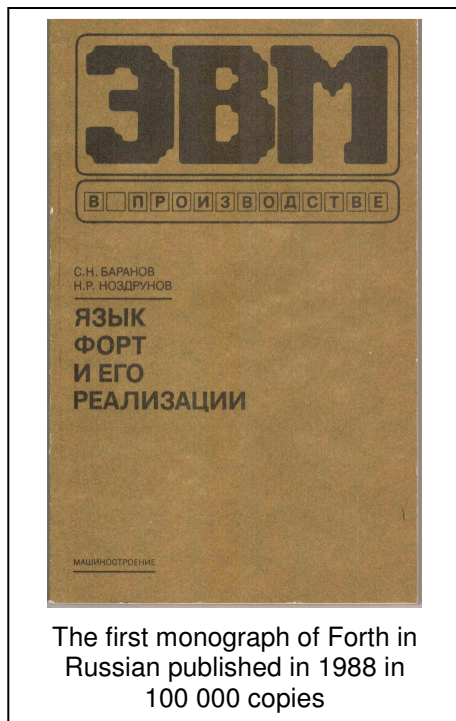
```
: VARIABLE CREATE 0 , DOES> ; : CONSTANT CREATE , DOES> @ ;
```

Having acquired that even fundamental control flow structures – branches and loops – may be so easily and simply expressed in the Forth core, we started to create quite exotic and unusual (at that time) control structures and experiment with them: switches, backtracking, exceptions handling and throwing, etc., based on the idea of a vector code field and manipulations with the return address and self-modifications of the running code.

The team immediately started to develop an implementation of Forth for ES 1030 (an IBM/360 compatible) – the only available computer at that time. An implementation in assembler was done remarkably fast, it was then used to bootstrap a Forth system from its baseline written in Forth; this source code was later published as an appendix to the first monograph on Forth in Russian [4], specifically aimed at industrial applications. At the same time another implementation of Forth was developed for the terminal ES-7970 with the K580 microprocessor [5], along with a number of utility applications for it.

The next phase started with implementing rather complicated projects in Forth. One of them was incorporated in the PhD thesis of Vyacheslav Kirillin “An Instrumental System for Developing Language Means of Microprocessor Machinery”, proved at the Leningrad State University in 1985. In particular, it contained descriptions of portable compilers from Pascal and Basic into K580, which worked on the terminal ES-7970 on top of its Forth system. Further popularization of Forth was contributed by Prof. Joseph Romanovsky at the Mathematical Department of the Leningrad State University [6]. A computer class was organized in form of a series of over 20 terminals connected to a powerful (at that time) ES mainframe (an IBM/370 compatible), where students could study Forth and experiment with it, surfing through an on-line Forth manual developed by Igor Agamirzian, Sergey Baranov, Vyacheslav Kirillin, and Nikolay Nozdrunov. While working at the terminal, students could create their own Forth programs, run them, and observe the results in parallel with reading manual sections. After studying Forth, it was much easier for students to learn PostScript and other interpretative programming languages provisioned at the curriculum.

At that period only the names of relevant technical journals were known in Russia: BYTE, Datamation, Dr. Dobb's Journal, and Forth Dimensions. We learnt about annual Forth conferences, held at the Rochester University, N.Y., by Institute for Applied Forth Research, Inc. However, to get access to these journals or to attend this conference seemed to be unrealistic. We



The first monograph of Forth in Russian published in 1988 in 100 000 copies

learnt that since 1983 the Institute published the Journal of Forth Applications and Research, that SIGForth (Special Interest Group) on Forth was established within the Association for Computing Machinery (ACM) with its periodicals SIGForth Bulletin and Newsletters, while in Europe annual euroForth conferences were regularly held by industrial companies and R&D institutions interested in Forth. A colleague of us, Alexander Sakharov, who worked at that time at the Library of the Academy of Sciences of the USSR, managed to provide within several years a subscription to the Journal of Forth Applications and Research, thus making this journal available to the interested people in Leningrad at least for reading at this open public library.

We found and contacted a team developing a dedicated Forth-processor [7] at the Institute of Cybernetics of the Estonian Academy of Sciences. The project was financed within a special contract with industry. As the leading engineer of the team Alexander Astanovsky came from Leningrad, we soon established common relations and interests of this group with our team in the Leningrad State University. Due to the efforts of these Estonian colleagues, a number of our R&D papers appeared in the collections of articles “Programming for Microprocessor Machinery” regularly published by the Institute of Cybernetics in Tallinn. In 1982 a Soviet conference on Forth was organized by Matti Tombak at the Tartu State University in Tartu, Estonia.

Approximately in 1980 within the framework of the Commission on Technology of the State Committee on Science and Technology headed by Prof. Igor Velbitsky of the Institute of Cybernetics of the Ukrainian Academy of

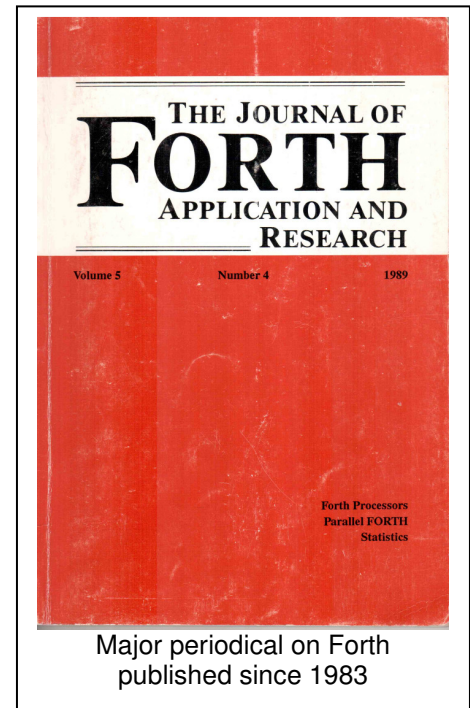
Sciences, a Working Group on Microprocessor Machinery was formed, headed by Dr. Raivo Raud of the Institute for Cybernetics of the Estonian Academy of Sciences. In this Working Group a Forth division was quickly established with active participation and contributions from Vsevolod Kotlyarov, Sergey Baranov, Grigory Pogosiants, and Alexander Liberov.

Appearance of the already mentioned monograph [4], which incorporated accumulated experience of developing Forth systems, became a noticeably milestone on the Forth path in the USSR. Its first print was made in 50 000 copies; however, after receiving many requests from practitioners, the public house Mashinostroyenie (Machine-building) made another print of additional 50 000 copies, which turned out to be a rare case in its publishing practice!

A number of Soviet Forth systems, based on the standards fig-Forth and Forth-83, had already been known by that time: Forth-SM (S.Katsev, I.Shendrikov), Forth-Tartu (R.Viainaste, A.Yuurik), Forth-K580 (V.Kirillin, A.Klubovitch, N.Nozdrunov), Forth-ES (S.Baranov), Forth-Iskra-226 (G.Lezin), Forth-M6000 (V.Patryshev), Forth-BESM-6 (I.Agamirzian), Forth-Elbrus (A.Soloviev), Forth-Agat (A.Trofimov), to name just a few, which clearly demonstrates a great interest to this language and the respective programming technique among the Soviet software community. In the mentioned monograph, principles of Forth were systematically explained and demonstrated with the source code of a Forth core for the IBM/360 instruction set. Later, with appearance of personal computers, Forth systems for IBM PC and compatibles under MS-DOS were developed and widely deployed: Astro-Forth (I.Agamirzian) and Beta-Forth (S.Baranov).

In parallel with implementations of Forth as per se, various dialects of the language were developed, mainly for control applications. One of them – the Comfort system – was developed by a team at the St. Petersburg State Polytechnic Institute (V.Kotlyarov, N.Morozov, A.Pitko, S.Kireyev) for two families of 16-bit microcomputers Elektronika S5 and Elektronika 60, which were successfully employed in industrial control systems of various classes. In the Leningrad Construction Bureau of the state company “Svetlana” a chip for Forth and Comfort was developed and manufactured.

At the peak of “perestroika” (re-building) in the USSR in April 1988, through enormous efforts of Boris Katsev and Nikolay Nozdrunov the coop “Forth-Info” was established and registered in Leningrad, at that time it was one of the first coops in the area of programming and computing machinery. Employees of the laboratory of system programming of the Department of



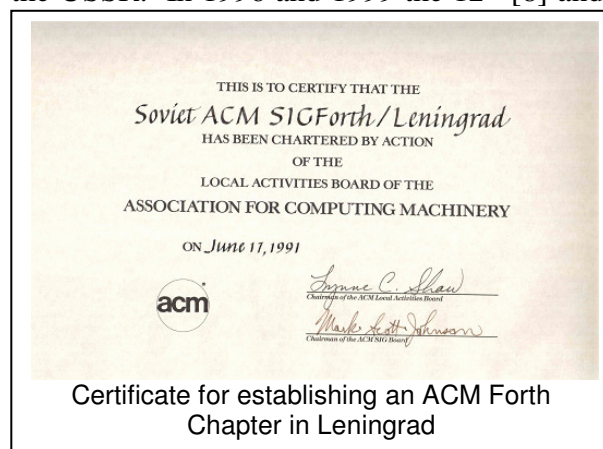
mathematics and mechanics of the Leningrad State University with experience in Forth formed its core. Their direct task was creating and further developing of new programming techniques based on Forth. A noticeable outcome of the coop first three years was developing the 16-bit microprocessor Dofin-1610 for real-time control systems. The processor turned out to be the fastest one in its class of 16-bit processors in the USSR at that time. It displayed performance 50 times higher than its closest analog i8086 and was produced in small parties at the state company “Integral” in Minsk, Belorussia, mainly thanks to Prof.Katsev’s connections in the industry. Later the coop was transformed into an innovative and technology company “TechnoForth”.

It so happened that the monograph [4] caught attention of Forth activists in the USA and the author received an invitation to come to the Rochester Forth Conference in July 1989. There were many technical hurdles in organizing this trip, which was handled through the Presidium of the USSR Academy of Sciences in Moscow; however, all was over thanks to the efforts and good will of the American colleagues. Within several following years thanks to established and strengthened connections, Soviet Forth-people could participate in the annual Forth conferences in the USA and Europe with their contributions and presentations of their accomplishments, exchanging experience and new ideas in this domain with the world “Forth elite”. In 1992 with support from the coop “Forth-Info” a group of students from the Leningrad State University was brought to Rochester, and after that they were invited as interns for summer practice in several US companies interested in Forth. That was genially new experience in this transition period of the Russian history! Companies in US and Europe, including MMS, MPE, Delta-T, Silicon Composers, and others demonstrated tangible interest to the “Russian experience” and established partnership with Soviet organizations and specialists.



Participants of Rochester conference in 1989

At that time conferences on Forth and its applications started to be regularly held in the USSR as part of activities of the Working Group on Microprocessor Machinery. Some of them were sponsored by the coop “Forth-Info” and some were held under financing from the state budget. Return visits of Director of the Forth Institute in Rochester Larry Forsley and Prof. Nicholas Solntseff of the McMaster University in Hamilton, Canada, took place. Their itineraries included visiting institutions in Leningrad, Moscow, and Novosibirsk. In 1991 the first issue of the Russian journal “Forth in Research and Development” under the aegis of the Leningrad State University; unfortunately with no continuation because of financial reasons due to the collapse of the USSR. In 1996 and 1999 the 12th [8] and 15th [9] annual conferences EuroFORTH were held in St. Petersburg, Russia, in the hotel “Rus”, organized by SPIIRAS with leading contribution from S.Baranov, I.Podnozova, E.Ignashkina, M.Kolodin, and M.Gassanenکو. In 1996 Michael Gassanenکو proved his PhD thesis “Mechanisms of Code Execution in Open Extendible Systems Based on Threaded Code” and then continued his research in this direction [10].



Certificate for establishing an ACM Forth Chapter in Leningrad

In 1991 a local ACM chapter on Forth was created in Leningrad, which worked for several years. Due to these activities, the chapter received subscriptions of all ACM periodicals (over 30 titles) that were deposited at the Library

of the Academy of Sciences with free access to the general public. This helped the Library to maintain the completeness of its repository when the state funding for purchasing technical literary nearly stopped because of the USSR collapse.

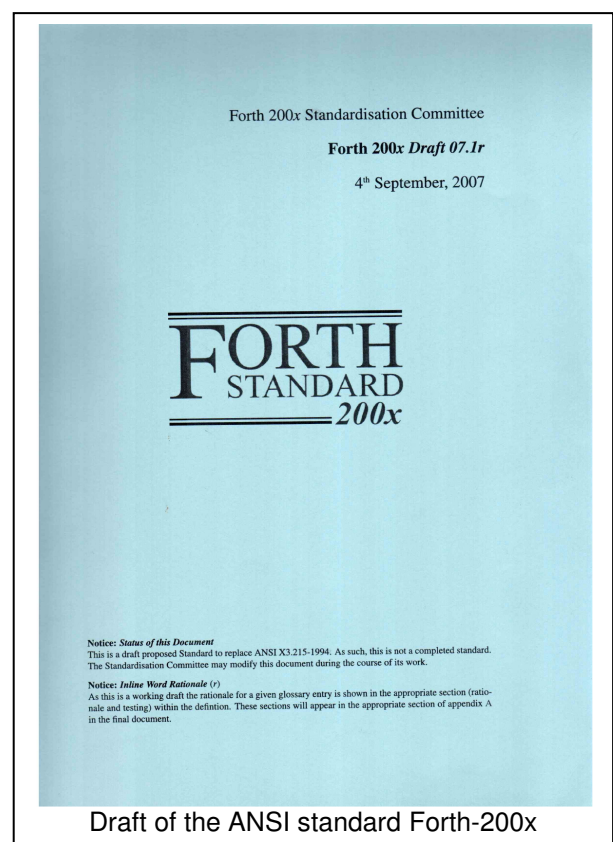
Later on, translations of books on Forth and its practical applications started to appear in Russia, such as [11], [12], [13], [14]. They stimulated further R&D in this area in Russia. For example, after learning about hardware implementation of cellular automata, which T.Toffoli and N.Margolus worked with at MIT [12], S.Baranov developed its software implementation of a cellular automata machine on top of his Beta-Forth system. Due to thorough programming of the main kernel, the resulting implementation Beta-CAM [16] displayed acceptable performance at the very first IBM PC with the processor i8080, which caused a surprise of those specialists when they saw its demonstration, inviting the author to visit MIT after the Rochester conference of 1991. Similarly, A.Kutuzov developed an expert system IBM PC based on the approach of C.Townsend [13], the system was later used to teach students at the St. Petersburg State Polytechnic University.

The second Russian monograph on Forth was that by V.Diakonov [15] published in 1992 at 30 000 copies, in 1993 followed by 50 000 copies of the monograph by Yu.Semyonov [17] where experience of the Institute for Theoretical and Experimental Physics in this area in this area was described. The valuable feature of this monograph was its appendices with the source code of a Forth interpreter for the processor Intel 80286 in the macro assembler MASM, implementing the standard fig-Forth, and a number of applications in Forth.

At the end of 1990-ies Andrey Cherezov [18] implemented his SP-Forth which is still in use in a number of Russian developments. A dedicated site of the Russian Forth community was established and is being supported [19], it stimulates further R&D and offers new ideas and solutions based on Forth.

Fallout of the USSR and transition of the former Soviet Union states to market economy made corrections to the status of Forth in Russia. The number of Forth enthusiasts and Forth addicts decreased, as the lion's share of programmers became employed in software companies with C/C++ and Java as their major instrumental languages. It's noteworthy that Java uses the same two-level structure of its code as Forth does: the source text is first translated into an intermediate representation (byte-code), which is interpreted by a Java virtual machine at run time. The Forth strength which allowed it to quickly fill-out its proper niche in software development for microprocessor machinery – direct access to all processor resources – at the same time turned out to be its vulnerability with respect to software security and safety. The Forth ideology – *everything allowed!* – opens doors to unwanted self-modifications of the executable code with penetration of software viruses and reduces code portability to other platforms. Java partially resolves this issue forbidding direct execution of machine instructions at the language level by protecting its assembler kernel from direct access from running applications via a complicated mechanism of their certification, while Forth can only rely on self-discipline and good will of the programmers.

Nevertheless, Forth developments continued to go on in Russia and in the rest of the world. Appearance of the Forth ANSI standard [20] which replaced the previous de facto standard Forth-83, became a strong argument for accepting Forth by the software industry. The new standard resulted from enduring efforts of the Technical Committee X3J14 which eliminated previous limitations of 16-bit address space and introduced the necessary ordering in the structure of the language. Since the beginning of 2000-ies



the standard is being revisited by a team of enthusiasts with support from companies which continue to use this language. Annual EuroForth conferences became the forum for their meetings, including the 29th one EuroForth 2013 in Hamburg [21].

One of the projects aimed at implementing the complete ANSI standard for IBM PC under MS Windows was carried out in 1994-1995 under a contract with Motorola, Inc. by the joint-stock company IBS created at SPIIRAS and headed by S.Baranov at that time. V.Kirillin, A.Klubovitch, and D.Preobrazhensky who participated in this project, later became authoritative specialists in Java and its implementations. The project curator from Motorola was A.Sakharov; before that he left for US and was employed by Motorola.

At the end of 1980-ies S.Baranov developed a Forth-based technology of porting large size legacy programs to other platforms, which included automated building of a compiler of the source language from its grammar representation in formal regular expressions considered as texts in Forth [22]. With this technology a known system of symbolic computations SAC-2 written in a special algorithm description language ALSDES was successfully ported to IBM PC from CDC-6000 and IBM/360 mainframes. Due to additional strong type control, included into the porting technology, 2 errors in SAC-2 algorithms were found which went unnoticed for many year of using this system on mainframes. Based on the results of this and other adjacent research works, in 1990 S.Baranov proved his Doc.Sci thesis “A Forth-Based Technology for Porting and Implementing Large Computer Algebra Packages”.

Application of the Forth technology in school informatics may be also considered as an important accomplishment. In the beginning of 1990-ies an implementation of full Logo for the Russian school PC “Elektronika UKNTS” [23] was carried out under a contract with the St. Petersburg branch of the Institute for New Technologies. This product was used for some time in a number of schools in St. Petersburg and Moscow before a massive migration of IBM PCs occurred. Due to Forth features all computer graphics of Elektronika UKNTS worked remarkably fast, in spite of limited processor performance and small memory size. To minimize memory requirements of this implementation, a special mechanism of detaching the finished software product from its instrumental Forth system and making it a minimized stand-alone one [24] was developed. Starting from the main Forth word of the application (similar to the function `main` in C), only words from the Forth core were selected, which were referenced to from this word in the process of automatic construction of their transitive closure. Moreover, if dynamic search in the Forth vocabulary was not anticipated, then the vocabulary entries were deprived of their headers, making the entries “truncated” and consisting on the code field and parameter field only. Thus, the resulting applications became remarkably small – just 8K for a complete implementation of the Logo language.

Using the same approach, a version of the school micro PC Elektronika 31 was created at the already mentioned Construction Bureau of the state company “Svetlana”, the operating system and a programming system in Basic being implemented in the Forth dialect Comfort.

Interesting ideas in the same area of interpretative programming languages were developed by A.Baehrs in Novosibirsk. He proposed a notion of the “working mix” [25] which became a conceptual basis for developing software for the work station “Mramor”. His PhD thesis presented for viva in 1993 was unanimously qualified as a Doc.Sci work by examiners and members of the dissertation council, so his Doc.Sci viva took place in Moscow in the next year of 1994. On an advice from A.Baehrs and with his support a detailed analysis of the Forth phenomenon was made and published [26].

Summarizing, one can say that for nearly 40 years Forth continues to exist and attract talented programmers with its options “to do everything” with high quality, little effort, and quite fast. In spite of reduced share of implementations for embedded applications, it continues to find its champions and supporters and allows them to succeed in their developments in the current market environment.

References

- [1] Moore C.H. FORTH: A New Way to Program a Mini-Computer. – Astronomy and Astrophysics Supplement, 1974, vol.5. – P.497-511
- [2] Forth Interest Group – <http://www.forth.org/index.html>
- [3] BYTE, Vol.5, No 8, August, 1980

- [4] Baranov S.N., Nozdrunov N.R. The Forth Language and its Implementations. – Leningrad.: Mashinostroyeniye, 1988. – 156 p. (In Russian)
- [5] Baranov S.N., Kirillin V.A., Nozdrunov N.R. Implementation of Forth for the Display Terminal ES-7970. – In the collection: “Programming of Microprocessor Machinery”. Tallinn: Institute for Cybernetics, 1984. – P. 41-49 (In Russian)
- [6] Burago A.Yu., Kirillin V.A., Romanovsky J.V. Forth – a Language for Microprocessors. Leningrad.: Znaniye, 1989. – 36 p. (In Russian)
- [7] Astanovsky A.G., Lomunov V.N. A Processor Oriented to Forth. In the collection: “Programming of Microprocessor Machinery”. Tallinn: Institute for Cybernetics, 1984. – P. 50-67 (In Russian)
- [8] EuroForth 1996. – <http://www.forth.org/bournemouth/euro/index.html>
- [9] EuroForth 1999. – <http://www.forth.org.ru/~mlg/ef99/EF99repo.html>
- [10] Gassanenko M.L. A One-Stack Implementation of Backtracking for Forth. – Proceedings of SPIIRAS. 2002. Issue 1. Vol. 1. St. Petersburg.: Nauka, 2002. – P. 211–223 (In Russian)
- [11] Brodie L. Starting Forth. An Introduction to the Forth Language and Operating System for Beginners and Professionals. – Moscow: Finance and Statistics, 1990. – 352 p. (In Russian)
- [12] Toffoli T., Margolus N. Cellular Automata Machines: A New Environment for Modeling. – Moscow: Mir, 1991. – 280 p. (In Russian)
- [13] Townsend C., Feucht D. Designing and Programming Personal Expert Systems. – Moscow: Finances and Statistics, 1990. – 314 p. In Russian)
- [14] Kelly M., Spies N. Forth: A Text and Reference. – Moscow: Radio and Telecom, 1993. – 320 p. (In Russian)
- [15] Diakonov V.P. Forth Systems for PC Programming. – Moscow: Nauka, 1992. – 352 p. (In Russian)
- [16] Baranov S.N. Cellular Automata on a PC. Priroda, 1992, №9. – P.17-23. (In Russian)
- [17] Semyonov Yu.A. Programming in Forth. – Moscow: Radio and Telecom, 1991. – 241 p. (In Russian)
- [18] Site of Andrey Cherezov – <http://www.enet.ru/win/cherezov/> (In Russian)
- [19] Site of the Russian Forth-community – <http://www.forth.org.ru/news/> (In Russian)
- [20] American National Standard for Information Systems. Programming Languages. Forth. – <http://www.openfirmware.info/data/docs/dpans94.pdf> – 210 p.
- [21] EuroForth 2013. – <http://www.complang.tuwien.ac.at/anton/euroforth/ef13/>
- [22] Baranov S.N. Implementation of the MINISAC System for Symbolic Computations in Forth. – In collection of papers: Mathematical Methods of Constructing and Analysis of Algorithms Leningrad: Nauka, 1990. – P.3-15 (In Russian)
- [23] Baranov S.N., Preobrazhensky D.S. Logo in Forth. – Prolog, 3(5), 1993. – P.6-10 (In Russian)
- [24] Baranov S.N., Software Product Alienation in Beta-Forth. – In book: Problems of Software Engineering. St. Petersburg: SPIIRAS, 1992. – P.139-147 (In Russian)
- [25] Baehrs A.A. On Object-Oriented Aspects and Organization of the Architecture of Software Systems. – In collected articles “Actual Problems of Software Engineering”. Leningrad, LIAN, 1989. – P.4-15. (In Russian)
- [26] Baranov S.N., Kolodin M.Yu. The Forth Phenomenon. – In book: System Informatics, issue 4, Novosibirsk: ISP, 1995. – P.193-271 (In Russian)
- [27] SoRuCom 2011 – <http://sorucom.novgorod.ru/>

About the Author



Sergey Baranov graduated with honor the Leningrad State University in 1972, worked at this university, at SPIIRAS, Motorola, St.Petersburg State Polytechnic University; PhD since 1978, Doc.Sci since 1991, Prof. since 1993. Currently works at SPIIRAS as a Chief Research Associate, teaches students at 3 major St. Petersburg Universities, and performs consulting at Motorola Mobility LLC. Major scientific interests are software engineering, compilers, analysis and verification of software specifications, formal methods, and symbolic computations.

Forth Floating Point Word-Set without Floating Point Stack

Willi Stricker

Springe, Germany,

September, 20, 2012

Preface

Originally computers work with integer numbers only. That is as well true for the Forth programming language with its basic words UM*, UM/MOD and further *, /, MOD, */ and */MOD.

For practical use this integer arithmetic is very restricted and there were lots of attempts to implement floating point arithmetic in computers as well. But in contrast to integer numbers there are lots of choices how to represent floating point numbers. Due to this problem the IEEE standards committee established the IEEE standard for floating point numbers.

The main properties are:

The floating point number is separated in 3 parts: sign, significant and (transformed) exponent.

The significant with suppressed 1 (first bit) (normalized with a special denormalized treatment for very small numbers).

Especially they standardised two formats: single precision (32 bits wide) and double precision (64 bits wide) and additionally a special format for internal representation (80 bits wide).

Software versus hardware

At first the floating point arithmetic was programmed by software, using different formats. Later, hardware components were offered to avoid extensive programming and execution time, especially since the IEEE-format was established. The special floating point hardware is either an I/O-port or a so called „coprocessor“. Now the coprocessors are mostly incorporated in standard microprocessors.

The main features of floating point coprocessors are:

- Use of a special coprocessor interface with special instructions
- The internal representation of floating point numbers is 80 bits (IEEE special format)
- It has its own internal stack for floating point numbers
- The access of the operands needs special programming steps from „outside“ via the floating point stack.

Nowadays most „better“ microprocessors have an integrated floating point coprocessor. On the other hand especially for smaller systems without coprocessor it is sometimes necessary to use floating point numbers and arithmetic and that must be programmed by software.

Consequence:

There are two choices:

- Using the floating point coprocessor directly, that means access of the operands by the floating point stack with special use of control instructions and control registers,
- Using floating point arithmetic without floating point stack with either floating point created by software (no coprocessor) or using the coprocessor indirectly by hiding the hardware by software adaption.

In the Forth 2012 standard both choices are offered, implementation dependant.

Floating point arithmetic without floating point stack in Forth

That means the standard Forth parameter stack is used for all numbers in the same manner. For a 32 bit system: Integer and single precision floating point numbers have 32 bits, double integer and double precision floating point numbers have 64 bits

For example adding two numbers
with

```
VARIABLE INT1 VARIABLE INT2 VARIABLE INT3
VARIABLE FLO1 VARIABLE FLO2 VARIABLE FLO3
```

integer addition:

```
INT1 @ INT2 @ + INT3 !
```

floating:

```
FLO1 @ FLO2 @ F+ FLO3 !
```

Remark

There is no need for special treatment of fetching, storing or alignment of floating point numbers. Obviously the work with these numbers is much more easy to handle.

Problem: Double precision floating point numbers

The floating point numbers used in a floating point stack are always in 80 bit special format. Consequently there is no special difference necessary for single or double precision because all operations are done on the floating point stack. Only for input, output and storage it is necessary to convert the numbers into the used format.

In contrast the numbers on the parameter stack have to be in the working format. So all operations must be executed in that format.

Conclusion:

Like for integer numbers all operators must be present in single and double format (F+ and DF+ etc.).

For example adding two double precision numbers
with

```
2VARIABLE INT4 2VARIABLE INT5 2VARIABLE INT6
```

```
2VARIABLE FLO4 2VARIABLE FLO5 2VARIABLE FLO6
```

integer addition:

```
INT4 2@ INT5 2@ D+ INT6 2!
```

floating addition:

```
FLO4 2@ FLO5 2@ DF+ FLO6 2!
```

Double precision input

In Forth2012 floating point numbers have no distinguishing mark to indicate the input as single or double precision. For integers in Forth there is the mark dot „.“ to indicate the double precision input anywhere inside the digits.

Floating point numbers in Forth have a dot already as a position for the fractional (decimal) point. So other than Forth2012 standard **it is suggested to add another dot at the end of the input stream to indicate double precision floating point inputs.**

Examples:

single floating: 123.4E7 45E-6 -332E

double floating: 123.4E7. 45E-6. -332E.

Double precision output:

As stated above **outputs need a special word, suggested „D.“** The output is just printed with the maximum count of possible digits.

Examples:

input stream: 123.4E7

Result: FS. 1.234000E9

input stream: 123.4E7.

Result: DF. 1.2340000000000000E9

Conclusion

Appendix A shows the whole Forth2012 floating-point word set in a table with remarks about the use with and without floating point stack and recommended words for double precision.

Appendix B shows the main Forth words for a floating point word set written in high level Forth to indicate that it is just a pretty simple and short program. It needs only about 2 kByte for a 32 bit system!

Appendix A

Comparison of floating point words with and without floating point stack with single and double precision:

Remarks:

*) Instructions necessary for double precision floating point arithmetic without floating point stack but not defined in Forth 2012

***) one of them maybe defined as standard, presumably FROUND

| | | |
|--|--|--|
| FORTH 2012 with Floating stack single and double precision | FORTH 2012 without floating stack single precision | FORTH 2012 without floating stack double precision |
|--|--|--|

Format alignments

| | | |
|------------------|-----------------|-----------------|
| FALIGN | <i>Not used</i> | |
| FALIGNED | <i>Not used</i> | |
| DFALIGN | | <i>Not used</i> |
| DFALIGNED | | <i>Not used</i> |
| SFALIGN | <i>Not used</i> | |
| SFALIGNED | <i>Not used</i> | |
| FLOAT+ | <i>Not used</i> | |
| FLOATS | <i>Not used</i> | |
| FFIELD | <i>Not used</i> | |
| SFLOAT+ | <i>Not used</i> | |
| SFLOATS | <i>Not used</i> | |
| SFFIELD | <i>Not used</i> | |
| DFLOAT+ | | <i>Not used</i> |
| DFLOATS | | <i>Not used</i> |
| DFFIELD | | <i>Not used</i> |

Declarations

| | | |
|------------------|----------------------|-----------------------|
| FCONSTANT | Use CONSTANT instead | Use 2CONSTANT instead |
| FVARIABLE | Use VARIABLE instead | Use 2VARIABLE instead |
| FLITERAL | Use LITERAL instead | Use 2LITERAL instead |
| REPRESENT | <i>Not used</i> | |
| FVALUE | <i>Not used</i> | |

Memory access

| | | |
|------------|----------------------|-----------------------|
| F@ | <i>Use @ instead</i> | |
| F! | <i>Use ! instead</i> | |
| DF@ | | <i>Use 2@ instead</i> |
| DF! | | <i>Use 2! instead</i> |

Special control

| | | |
|---------------|--------------------------|--|
| FLOOR | <i>Not implemented</i> | |
| FROUND | <i>e.g. standard **)</i> | |

| | | |
|----------------------|--------------------------|--|
| FTRUNC | <i>e.g. standard **)</i> | |
| F~ | <i>Not used</i> | |
| PRECISION | <i>Not used</i> | |
| SET-PRECISION | <i>Not used</i> | |

Stack operations

| | | |
|---------------|--------------------------|--------------------------|
| FDEPTH | <i>Use DEPTH instead</i> | |
| FDROP | <i>Use DROP instead</i> | <i>Use 2DROP instead</i> |
| FDUP | <i>Use DUP instead</i> | <i>Use 2DUP instead</i> |
| FOVER | <i>Use OVER instead</i> | <i>Use 2OVER instead</i> |
| FROT | <i>Use ROT instead</i> | <i>Use 2ROT instead</i> |
| FSWAP | <i>Use SWAP instead</i> | <i>Use 2SWAP instead</i> |

Numerical output

| | | |
|------------|------------|---------------|
| F. | F. | |
| FE. | FE. | |
| FS. | FS. | DF. *) |

Format conversion

| | | |
|------------------|-------------------|-------------------|
| >FLOAT | >FLOAT | |
| F>D | F>D | DF>D*) |
| F>S | F>S | |
| D>F | D>F | D>DF *) |
| S>F | S>F | |
| | F>DF *) | DF>F *) |

Standard arithmetic functions

| | | |
|----------------|----------------|--------------------|
| FNEGATE | FNEGATE | DFNEGATE *) |
| F+ | F+ | DF+ *) |
| F- | F- | DF- *) |
| F* | F* | DF* *) |
| F/ | F/ | DF/ *) |
| FABS | FABS | DFABS *) |

Comparison

| | | |
|---------------|---------------|-------------------|
| FMIN | FMIN | DFMIN *) |
| FMAX | FMAX | DFMAX *) |
| F0< | F0< | DF0< *) |
| F0= | F0= | DF0= *) |
| F< | F< | DF< *) |

Extended functions

| | | |
|-------------|-------------|-----------------|
| FSIN | FSIN | DFSIN *) |
|-------------|-------------|-----------------|

| FEXP | FEXP | DFEXP *) |
|-----------|-----------|--------------|
| etc | etc | etc *) |

Appendix B

Forth programs for a short floating point word set without floating point stack with single precision

Forth programs for the words FNEGATE, F+, F-, F*, F/, S>F, F>S written in high level forth

Explanations:

An intermediate format is defined for the three components, called „FX“ (s e m), that separates the components:

s = sign: msb. 0 = positive, 1 = negative,

e = exponent: signed exponent that is a retransformed transformed exponent ($e = e_t - 7F$),

m = mantissa: significant with open bit and normalized,

f = floating point number according to IEEE (single precision, 32 bits wide),

d = single integer number (32 bits wide)

```
***** floating kernel for 32 bit processor *****
*****
\ ---- conversion auxiliaries -----
:   FX>F      ( m e s -- f )
[ HEX ] \ -----
80000000 AND >R      \ sign
7F + >R              \ exponent
DUP 0= IF R> DROP 0 >R THEN \ test 0 mantissa
R@ 1 <
IF                  \ denormalized ?
  R> 1- >R BEGIN U2/ R> 1+ DUP >R 0= UNTIL \ adjust exponent
THEN
R@ FE > IF R> DROP FF >R DROP 0 THEN \ adjust infinity
007FFFFF AND
R> 17 SHIFTL OR
R> OR
; DECIMAL

:   F>FX      ( f -> m e s )
[ HEX ] \ -----
DUP 80000000 AND >R      \ sign
DUP 7F800000 AND 17 SHIFTR 7F - >R \ exponent
7FFFFFFF AND
R@ -7F >
IF 800000 OR
ELSE DUP 0= NOT
  IF 2* BEGIN DUP 800000 AND 0= WHILE 2* R> 1- >R REPEAT THEN
THEN
R> R>
; DECIMAL

\ ---- fixpoint <-> floating point conversion -----
:   S>F      ( d -- f )
\ -----
DUP 0=
IF
  -127 0
ELSE
  DUP [ HEX ] 80000000 [ DECIMAL ] AND >R \ sign
  DUP 0< IF NEGATE THEN
  31 >R BEGIN DUP 0< NOT WHILE 2* R> 1- >R REPEAT \ normleft
```

```

      8 SHIFTR R>          \ mantissa, exponent
      R>                  \ sign
      THEN                \ m e s = fx
      FX>F                \ convert to f
;

:      F>S      ( f -- d )
\ -----
F>FX          \ m e s
>R           \ sign
23 - DUP 7 > \ exponent, 8-1 because of sign bit
IF          \ infinit )
  DROP DROP R> [ HEX ] IF 80000000 ELSE 7FFFFFFF THEN
  [ DECIMAL ]
ELSE
  DUP 0>
  IF SHIFTL ELSE NEGATE SHIFTR THEN
  R> IF NEGATE THEN
THEN ;

\ ---- floating point arithmetic -----

:      FNEGATE  ( f -- fneg )
[ HEX ] \ -----
80000000 XOR          \ toggle sign bit
; DECIMAL

:      F+      ( f1 f2 -- fsum )
[ HEX ] \ -----
>R F>FX R> F>FX          \ convert operands to fx
>R 2 ROLL >R            \ save signs,
2 ROLL OVER OVER MAX >R SWAP - DUP >R 0< \ exponent, exp-diff
IF      SWAP R> NEGATE SHIFTR SWAP \ normalize m1 if necess
ELSE   R> SHIFTR \ normalize m2 if necess
THEN
R> R> SWAP >R IF SWAP NEGATE SWAP THEN \ negate m1 if necess
R> R> SWAP >R IF NEGATE THEN \ negate m2 if necess
DUP 80000000 AND DUP R> SWAP >R >R \ add m1 and m2, sign
IF NEGATE THEN          \ negate mantissa if necess
DUP 0= NOT
IF                      \ mantissa not zero ?
  DUP 0FFFFFFF. U>
  IF
    U2/ R> 1+ >R
  ELSE
    \ normalize
    BEGIN DUP 800000 AND 0= WHILE 2* R> 1- >R REPEAT
  THEN
THEN
R> R>          \ add exponent and sign
FX>F          \ convert to f
; DECIMAL

:      F-      ( f1 f2 -- fdiff )
\ -----
FNEGATE F+
;

:      F*      ( f1 f2 -- fprod )
[ HEX ] \ -----
>R F>FX R> F>FX          \ convert operands to fx
3 ROLL XOR >R           \ sign
2 ROLL + >R             \ exponent
UM*                    \ mantissa 64 bits
FFFF AND WSWAP SWAP FFFF0000 AND WSWAP OR \ reduce to 32 bits

```



```

DUP DUP 80 AND IF 100 + THEN \ round
DUP 0<
IF SWAP DROP R> 1+ >R \ normalize
ELSE DROP 2* DUP 80 AND IF 100 + THEN
THEN \ normalize and round
8 SHIFTR \ normalize to 24 bits
R> R> \ add exponent and sign
FX>F \ convert to f
; DECIMAL

```

```

: F/ ( f1 f2 -- fquot )
[ HEX ] \ -----
>R F>FX R> F>FX \ convert operands to fx
3 ROLL XOR >R \ sign
OVER 0= IF DROP DROP 800000 -96 THEN \ check for zero
ROT SWAP - >R \ exponent
>R 0 SWAP R> UM/MOD2 \ mantissa division
0> IF U2/ 80000000 OR ELSE R> 1- >R THEN \ normalize
8 SHIFTR \ normalize to 24 bits
SWAP DROP \ drop remainder
R> R> \ add exponent and sign
FX->F \ convert to f
; DECIMAL

```

Auxiliary instructions used in the above programs

```

: UM/MOD2 ( d1 dh d - mod q1 qh ) \ unsigned divide with double quotient result
>R 0 R@ UM/MOD R> SWAP >R UM/MOD R> ;

```

```

: WSWAP ( w0:w1 - w1:w0 ) \ swap operand halves
DUP 16 LSHIFT SWAP 16 RSHIFT OR ;

```

Literate Programming

- Mix prose text and program fragments
- Explain, why you are doing things in your programs
- Present in an educational order

Forth Literate Programming with IPython notebook

Ulrich Hoffman <uh@fh-wedel.de>

IPython & IPython Notebook

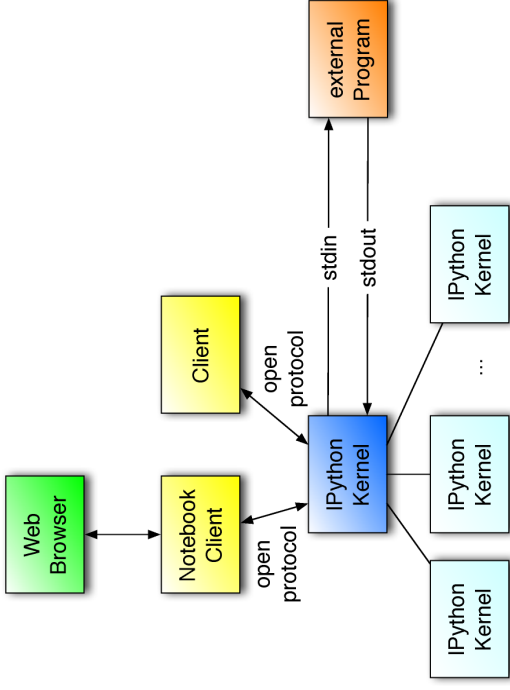
- Literate Programming
 - IPython and IPython notebook
 - Connecting Forth to IPython notebook
 - Demo
 - next
- Authoring system based on *cells*
 - Markdown Text
 - Python program fragments

Overview

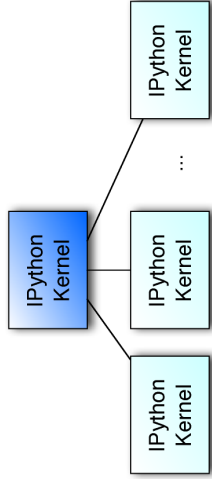
IPython & IPython Notebook - architecture



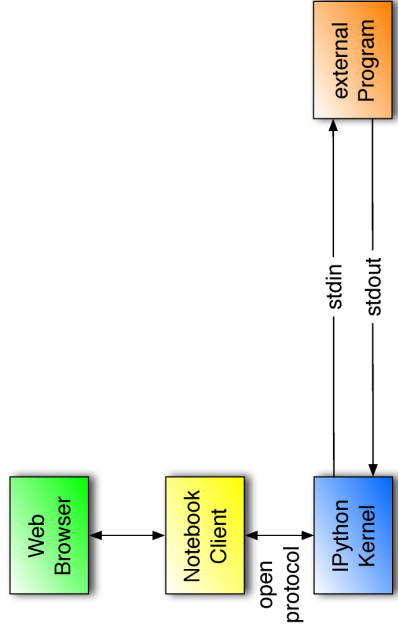
IPython & IPython Notebook - architecture



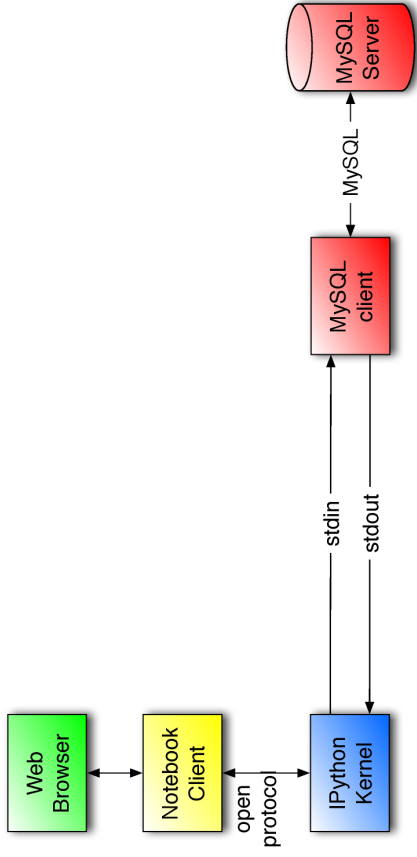
IPython & IPython Notebook - architecture



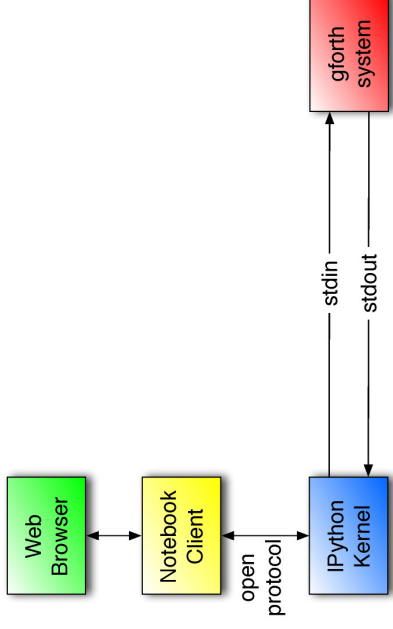
IPython & IPython Notebook - architecture



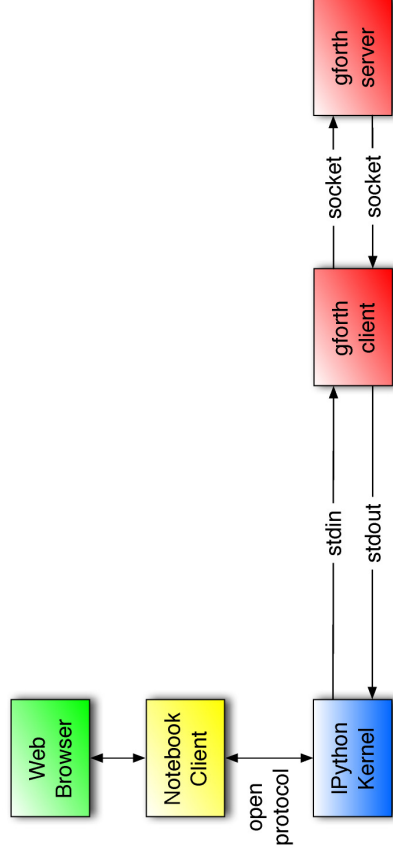
IPython & IPython Notebook - architecture



Connect Forth to IPython notebook



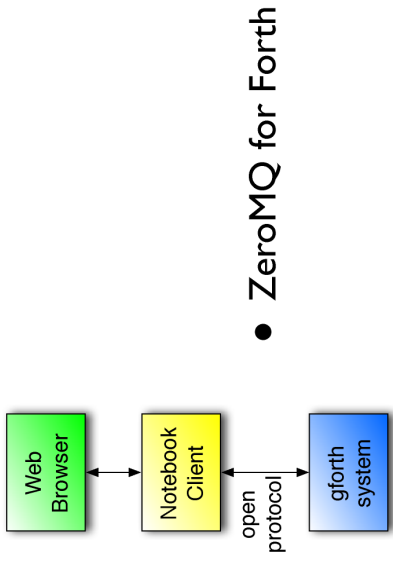
Connect Forth to IPython notebook



- Well, a Forth system is also just a program...

Demo

Directly connect the IPython client to Forth



next

- Use IPython as a Literal Terminal for Launchpad...
- Directly connect the IPython client to Forth

Fork me on GitHub

- <https://github.com/uho/forth-notebooks>

Forth to .NET Bridge

Gerald Wodni

EuroForth 2013

- interpretation mode only
- simple type conversion: `>o o>s`
- primitives by reflection of classes:
class math { public int add(int a, int b) { return a+b; } }
- load assemblies at runtime
- inspect types and activae (instance) them

Forth-System in .NET

Stacks

Number *n* and *d*

Float

Bool

String

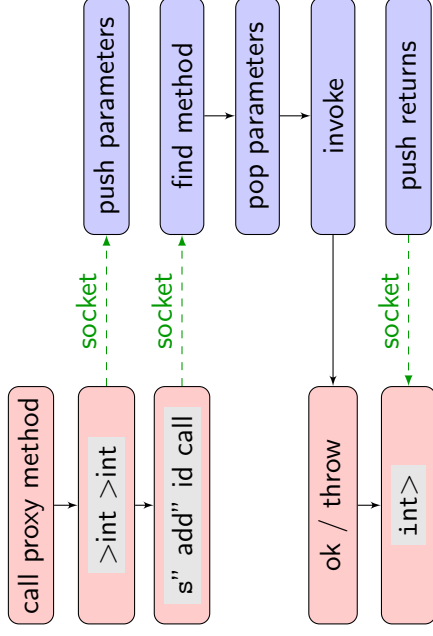
Type allow casting, `t` " System.Windows.Forms.Form"

Object "data" stack, no meaningful operations

- 2 TCP-connections per thread (ok+exceptions, events)
- Netstring based `2:s" ,5:hello;`
- Stack interface `>int, int>, >string, string>`

Protocol

Calling a function



Overloading

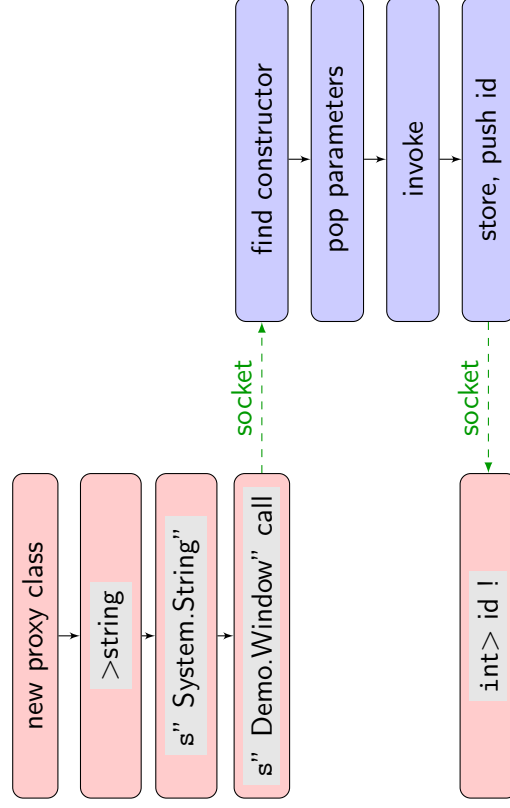
```

class math {
    public int add( int a, int b ) {
        return a+b;
    }
    public float add( float a, float b, float c ) {
        return a+b+c;
    }
}
  
```

• Distinguish between signatures

- add:n-n
- add:r-r-r

Instance Class



Overloading

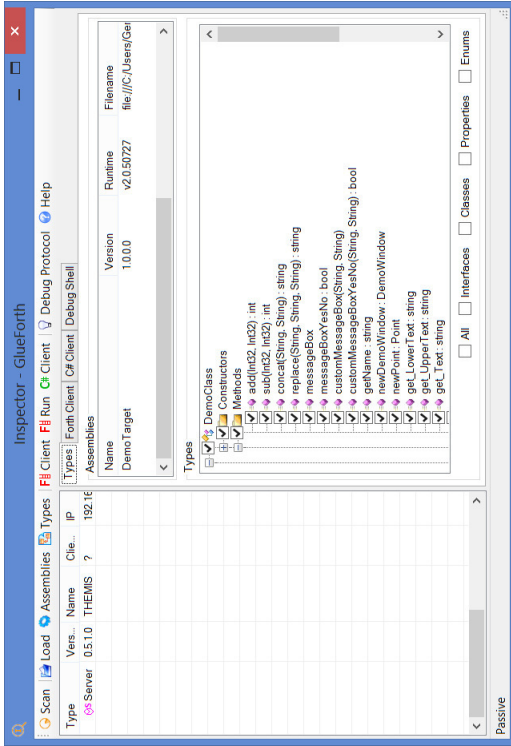
```

class math {
    public int add(params int[] numbers ) {
        int sum = 0;
        foreach( int number in numbers )
            sum += number;
        return sum;
    }
}
  
```

• Distinguish between signatures

- add:o

Demo



Future Work

- Construct Classes on the fly in .NET
- coexistence of COM and .NET:
 - switch calling convention
 - wrap COM into .NET class

Region-based Memory Allocation

M. Anton Ertl
TU Wien

`allot`

- can only reclaim in LIFO manner

`allocate/free`

- `free` after the last reference is consumed
- error prone:
 - dangling reference (`free` too early)
 - memory leaks (`forgot to free`)
- various workarounds
 - restrict programming
 - may cost performance (e.g. extra copies)

Problem: Memory management

How to reclaim no longer needed memory?

Garbage collection

- Convenient, but
- Complex, particularly with:
Real-time requirements
multicores
little type information (Forth)
- Forth garbage collection library since 1999

Region-based memory allocation

```
new-region ( -- region-id )  
region-alloc ( u region-id -- addr )  
free-region ( region-id -- )
```

Uses

- Separate regions for things that die at the same time
- E.g., in compiler:
region for the block
region for the definition
- In web service: Region for the HTTP request

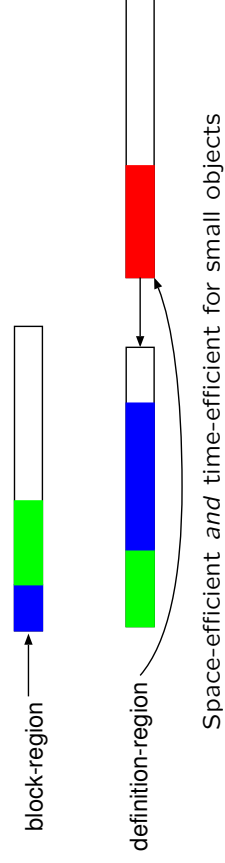
Reference counting

- Cyclic data structures
- slow
- Special dup, drop, ! etc. for addresses

Using regions

- Programmer control:
- Fewer regions: more convenient
- More regions: less dead wood
- Start out with few, add more if necessary

Implementation



Alternative interface

```
new-region ( -- region-id )
free-region ( region-id -- )
do-region ( xt -- ) \ xt ( region -- )
with-region ( region-id xt -- ) \ xt ( -- )
allocate ( u -- addr ior )
free ( addr -- ior ) \ does nothing
```

[: ['] word-using-allocate with-region use-result ;] do-region

Library words using allocate are usable with regions

Conclusion

- More convenient than free
- Compatible with multicores and real-time
- Why have regions not taken over the world?
Forth: interface issues
other languages: garbage collection won

net2o: Application Layer

Browser Components

Bernd Paysan

EuroForth 2013, Hamburg

Overview

Motivation

Sidetracking: What Changed Due to Snowden?

Requirements

A Few Demos

Slideshow

Videos

Text

Outlook

Purpose?

People want to share information (*share means copy*)

- Texts, photos, videos, music
- longer, structured documents
- Real-time media (chat, videos, video conferences)
- collaborative gaming

Things I want to show

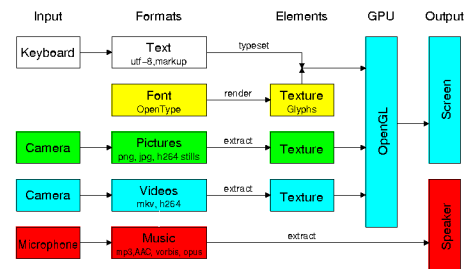
- In 2011 I did a presentation with the same title — back then, this was complete vaporware: the plan.
- Now there are components which need to be put together
- and there's a concept how to do that
- needs to work on PCs and mobile platforms like Android, which are sometimes "a bit strange".

Sidetracking: Impact of Snowden Leaks

- Encryption now uses Keccak (SHA-3) as primitive. Universal crypto primitive, faster than Wurstkessel at same level of security, chosen in an open competition.
- ECDHE for connection setup in a way that doesn't reveal identities (metadata!)
- Made sure the random numbers use entropy of the system, but not directly system random numbers
- Secure internet more important than ever!

Formats and IO

How to display things



Why OpenGL?

OpenGL can do everything

OpenGL renders:

- 1 Triangles, lines, points — simple components
- 2 Textures and gradients
- 3 and uses shader programs — the most powerful thing in OpenGL from 2.0.

Real requirement: visualization of *any* data. OpenGL can do that.

How to connect the media?

Lemma: every glue logic will become Turing complete

- currently used glue: HTML+CSS+JavaScript
- containers with Flash, Java, ActiveX, PDF, Google's NaCl...
- conclusion: use a powerful tool right from start!
- browser: run-time and development tool for applications

Security

Lemma: every sufficiently complex format can be exploited

Java's approach to secure the language from the inside can be seen as a failure. Java is now malware entry door number 1.

Sandbox

- sandbox the process that interprets network apps
- funnel network connections through a proxy — a shared memory module for net2o is missing
- encryption (key access!) outside the sandbox
- „same-origin“-policies don't work in a P2P cloud

Slideshow

I use the slide-how for this presentation

Fader

```
: fade { n1 n2 f: delta-time -- } n1 n2 = ?EXIT
ftime { f: startt }
BEGIN ftime startt f- delta-time f/ fdup 1e f< WHILE
<draw-slide
1e blend n1 draw-slide
( time ) blend n2 draw-slide
draw-slide> REPEAT
<draw-slide 1e blend n2 draw-slide draw-slide>
fdrop ;
```

Slideshow 2

Even more effects

Hslide

```
: hslide { n1 n2 f: delta-time -- } n1 n2 = ?EXIT
ftime { f: startt }
BEGIN ftime startt f- delta-time f/ fdup 1e f< WHILE
<draw-slide
pi f* fcos 1e f-
[ pi f2/ fnegate ] FLiteral f* fcos 1e f-
fdup n1 n2 > IF fnegate THEN xshift n1 draw-slide
2e f+ n1 n2 > IF fnegate THEN xshift n2 draw-slide
draw-slide> REPEAT
<draw-slide n2 draw-slide draw-slide>
fdrop ;
```



Approach and Problems

libSOIL: Simple API to load images

libjpeg and libpng have a very complicated AP
Other option: libSOIL:

libSOIL load texture

```
: >texture ( addr w h -- )
2 pick >r rgba-texture wrap nearest r> free throw ;
: mem>texture ( addr u -- addr w h )
over >r 0 0 0 { w~ w w~ h w~ ch# }
w h ch# SOIL_LOAD_RGBA SOIL_load_image_from_memory
r> free throw w @ h @ 2dup 2>r >texture 2r> ;
: load-texture ( addr u -- w h )
open-fpath-file throw 2drop slurp-fid mem>texture ;
```



Onion-Programming

Looks big from the outside



Onion-Programming

Use of martial tools recommended



Onion-Programming

Onion "all the way down"



Videos

OpenMAX AL

- Android uses OpenMAX AL as video framework — similar to gstreamer, but slightly different...
- renders video into a texture, but can also record videos from the camera
- input: MPEG transport stream
- C++-like C API (vtable implemented as function pointer struct)
- only half-hearted implemented, needs Java via JNI, can't handle resizes
- four languages for video player: Forth, C, Java, OpenGL shader language



JNI declarations

MediaPlayer

```
jni-class: android/media/MediaPlayer

jni-new: new-MediaPlayer ()V
jni-method: prepare prepare ()V
jni-method: start start ()V
jni-method: setSurface setSurface (Landroid/view/Surface;)V
jni-method: setVolume setVolume (FF)V
```



JNI declarations II

SurfaceTexture

```
jni-class: android/graphics/SurfaceTexture

jni-new: new-SurfaceTexture (I)V
jni-method: updateTexImage updateTexImage ()V
jni-method: getTimestamp getTimestamp ()J
jni-method: setDefaultBufferSize setDefaultBufferSize (II)V
jni-method: getTransformMatrix getTransformMatrix ([F)V
```



JNI calls

get timestamp

```
: get-deltat ( -- f )
media-sft >o getTimestamp o> d>f 1e-9 f*
first-timestamp f@ f- ;
```

Java-Calls integrate seamless into Mini-OO2 (Mini-OO2 with current object)



MTS? All videos today are MKV!

"Matroska" sounds like onion programming, too...

Container — what for?

- Usual explanation: several files too difficult to handle. IMHO, directories with multiple files are better than containers.
- Videos and audio stored as single frames and short packets
- Timestamps for synchronized playback
- Index for random access



Matroska interpreter

Binary XML format



Solution: read MKV, convert to MTS

- Matroska parser uses a hash table for the tags
- each tag has an associated Mini OOF2 method
- different classes for different purposes: dump for inspection, MTS converter class

Fonts rendering

Freetype-Gl renders OpenType fonts into OpenGL-Textures



- OpenType is state of the art
- we render textures, so the vector font needs to go into a texture
- FreeType-Gl uses a texture as glyph cache
- 1 glyph: 2 triangles

Text Render Demo Code



Fonts and Texts

```
48e FConstant fontsize#
atlas "/system/fonts/DroidSans.ttf\0" drop
fontsize# texture_font_new Value font1
atlas "/system/fonts/DroidSansFallback.ttf\0" drop
fontsize# texture_font_new Value font2
Variable text1$ "Dös isch a Tägscht." text1$ $!
Variable text2$ "这是一个文本：我爱彭秀清。" text2$ $!
```

Text Render Demo Code II



Fonts und Texte

```
: glyph-demo ( -- ) hidekb
1 level# +! BEGIN
<render
0. penxy 2!
font1 to font text1$ $@ render-string
-100e penxy sf! -60e penxy sfloat+ sf!
font2 to font text2$ $@ render-string
render>
sync >looper
level# @ 0= UNTIL ;
```

Outlook

This presentation has been rendered with \LaTeX Beamer...



- The next presentation should be rendered with MINOS2
- Texts, videos, and images should be get with net2o, shouldn't be on the device
- Typesetting engine with boxes and glues, line breaking and hyphenation missing
- a lot less classes than MINOS — but more objects
- add a zbox for vertical layering
- integrate animations
- combine the GLSL programs into one program?

Literature&Links



 BERND PAYSAN

<http://fossil.net2o.de/net2o/>

 BERND PAYSAN

<http://fossil.net2o.de/minos2/>