Computers **2014**, *3*, 69-116; doi:10.3390/computers3030069

OPEN ACCESS
computers
ISSN 2073-431X
www.mdpi.com/journal/computers

Article

An ECMA-55 Minimal BASIC Compiler for x86-64 Linux®

John Gatewood Ham

Burapha University, Faculty of Informatics, 169 Bangsaen Road, Tambon Saensuk, Amphur Muang, Changwat Chonburi 20131, Thailand; E-mail: john@buu.ac.th

Received: 24 July 2014; in revised form: 17 September 2014 / Accepted: 1 October 2014 / Published: 1 October 2014

Abstract: This paper describes a new non-optimizing compiler for the ECMA-55 Minimal BASIC language that generates x86-64 assembler code for use on the x86-64 Linux® [1] 3.x platform. The compiler was implemented in C99 and the generated assembly language is in the AT&T style and is for the GNU assembler. The generated code is stand-alone and does not require any shared libraries to run, since it makes system calls to the Linux® kernel directly. The floating point math uses the Single Instruction Multiple Data (SIMD) instructions and the compiler fully implements all of the floating point exception handling required by the ECMA-55 standard. This compiler is designed to be small, simple, and easy to understand for people who want to study a compiler that actually implements full error checking on floating point on x86-64 CPUs even if those people have little programming experience. The generated assembly code is also designed to be simple to read.

Keywords: BASIC; compiler; AMD64; INTEL64; EM64T; x86-64; assembly

1. Introduction

The Beginner's All-purpose Symbolic Instruction Code (BASIC) language was invented by John G. Kemeny and Thomas E. Kurtz for teaching at Dartmouth College in the early 1960's [2,3]. Its simplicity meant that even non-technical students could learn it quickly and use the language ([4], p. 106). As its popularity grew, the various implementations' dialects of BASIC ceased to be compatible with each other [5–7]. This led to a standardization effort by American National Standards Institute (ANSI) [8,9], which after many compromises became Minimal BASIC. The standardization effort was not successful for a variety of reasons, and most implementations of the language that called themselves BASIC were actually not fully compliant with the Minimal BASIC standard. If for no other reason, the lack of adequate string and file support meant that a strict Minimal BASIC implementation would not be a commercially viable product.

Despite Dijkstra's famous criticisms [10,11], BASIC proved to be a good language for use by normal people who want to learn how to program [12]. The simple type of programming traditionally used for introductory teaching seems to be absent from most schools today. While Microsoft®'s Small Basic [13] was a response to Brin's article, it implements something closer to Pascal than the traditional style of BASIC. Ian Larsen's BASIC-256 [14] is an open source interpreter that was also a response to the article, but it does not really implement the traditional style of BASIC either.

Extensive research failed to reveal any available fully compliant Minimal BASIC implementation, although clearly at least one must have existed which was used to create the National Bureau of Standards (NBS) test suite [15,16]. Most early BASIC implementations were interpreters, even though the language was always intended to be compiled ([4], p. 108; [17], p. 522). Also, no free and open source BASIC compiler generating *native 64 bit assembly code for x86-64 Linux®* was found. Anthony Liguori's GNU Liberty BASIC [18] Compiler Collection compiles to C, not assembly code, requires GNOME and `gcc` to be installed, and does not accept programs with line numbers. Line numbers are a requirement for ECMA-55 Minimal BASIC. Andre Victor's FreeBASIC [19] compiler does not support generating 64 bit programs at all. The commercial TrueBASIC [20] and Liberty BASIC [21] products do not run on or generate code for x86-64 Linux®. In addition, neither comes with source code for the compiler, so they are not useful for people wanting to learn how to build or modify a compiler. While Pure BASIC [22] does generate 64 bit `nasm` assembly code for x86-64 Linux®, it does not support any traditional line-numbered BASIC dialect, but instead is a compiler for a Pascal-derived language similar to the dialect made popular by Microsoft®'s Visual BASIC® product. None of these five compilers is designed for programs written in the Minimal BASIC dialect, although porting such a program to run on TrueBASIC should be possible since their documentation says they still accept line numbered programs. The absence of any working ECMA-55 Minimal BASIC compiler that would run on or generate code for x86-64 Linux® motivated the creation of the compiler described in this paper which compiles Minimal BASIC code into x86-64 assembly code for 64 bit Linux®. Intel® has called the x86-64 instruction set other names including IA-32e, EM64T [23] and Intel® 64 [24] which is the name that company currently uses. AMD has always referred to it as AMD64 [25].

The compiler described in the paper is designed as a simple example compiler. Many less rigorous universities offer a one semester undergraduate compiler course to students with relatively weak programming skills and experience. Complex modern production compilers are large code bases using advanced techniques which include algorithms for optimization that require ability and skill beyond what many beginning computer science students have. This compiler provides a simpler alternative while still generating code for current, widely-available commodity consumer 64 bit hardware. This compiler takes as input a classic language that was especially designed for teaching people outside of computer science to harness the power of a computer. Rather that attempting to use complex algorithms and generating production quality code, this compiler uses simpler algorithms and data structures to generate correct, non-optimized code. It does not support multiple front or back ends, keeping the design simpler and smaller. Compilation is accomplished without using any intermediate representation. That design decision keeps it simple, which is one of the primary goals, but has the side-effect of making advanced data and control flow analysis impossible. As an introductory first compiler, this limitation is acceptable. The goal of keeping things elementary enough for even students with introductory programming skill means that this compiler is not appropriate for use in an advanced course. However, highly-skilled students at academically challenging educational institutions are already well served by several existing advanced open source compilers. This compiler attempts to address the requirements for the low end of academia, where a more approachable, simple, entry-level, non-optimizing, free and open source compiler that undergraduate students with a lower skill level can understand is needed. It achieves this while still targeting modern commodity hardware, supporting floating point, and using an historic computer science teaching language, warts and all, instead of something custom-designed to be easy to compile.

When evaluating this compiler, it is important to keep in mind the target audience for this compiler. The target audience of this compiler is not students at top-tier universities or advanced compiler researchers, but ordinary undergraduate students with relatively poor programming and algorithmic skills who nevertheless need to learn about how compilers work. This compiler is intended to do two things:

(1) Serve as a standards-compliant baseline compiler implementation for the classic ECMA-55 Minimal BASIC dialect that can be used on x86-64 Linux®.
(2) Serve as an example compiler for entry-level undergraduate compiler courses at institutions where the students may possess only modest programming and algorithmic skill.

This compiler is not intended to serve as an example of the state of the art in compiler technology.

The two most well-known C compilers that actually generate x86-64 assembly for Linux® are `gcc` [26] and the `clang`/LLVM [27] toolchain and both of these projects have huge[1], complex code bases and can be considered state-of-the-art examples of modern optimizing production quality compilers. To understand them well enough to modify or extend them requires learning very complex algorithms, learning a non-trivial intermediate representation language, learning a machine description language, and more. The Free Pascal [28] `fpc` compiler can also generate assembly output, but it is also part of a large, complex system. The simpler `tcc` [29] compiler cannot emit assembly code. After the Minimal BASIC compiler was completed, another compiler called `pcc` [30] was found that could generate assembly output. While `pcc` is simpler than `clang`/LLVM and `gcc`, the code is still too difficult to grasp in one semester for many undergraduate students. In short, the well known free and open source production-quality compilers capable of generating x86-64 assembly for Linux® have a very steep learning curve and require a student to master an overwhelming amount of information which is clearly too much for many undergraduate students attending colleges with less-than-stellar academic standards who are just starting to learn about compiler construction. The compiler created by this project, in contrast, is smaller, less complex and yet really is a complete implementation of a programming language supporting full exception handling for floating point. To work with this Minimal BASIC implementation a student still must learn x86-64 assembly, how to implement a deterministic

[1] `gcc` version 4.9.1 has more than 4 million lines of source code, `clang`/LLVM version 3.4.2 has more than 2 million lines of source code.

finite state machine, how to implement top-down recursive-descent parsing, how to make Linux® kernel calls, and how to debug machine code.

2. Experimental Section

2.1. Which Standard To Implement?

The first major problem was that despite being revoked, the *ANSI X3.60-1978 Minimal BASIC* standard is still not freely available. The same is true for ISO's corresponding version, *ISO 6373:1984 Data processing—Programming languages—Minimal BASIC*. Fortunately the Europeans are more reasonable and the European Computer Manufacturers Association (ECMA) standards body allows free access to the *ECMA-55 Minimal BASIC standard* [31], so the compiler described in this paper has been written to process BASIC source code written for that standard.

2.2. Target Language

The next issue was choosing what assembler to target. The Netwide Assembler (`nasm` [32]) works well on Linux® and is actively developed. It uses Intel®'s operand order making it easy to use with Intel®'s documentation. However, it is an optional component for most Linux® distributions and may not be present in a default installation. The GNU `as` assembler [33], however, must be present for GNU `gcc` to generate executables, and `gcc` is included in all major Linux® distributions. GNU's toolchain uses the AT&T assembly dialect which reverses the operands, but is considered the standard on systems that are UNIX®-like. The decision was made to use the GNU `as` assembler because it would avoid a dependency on a possibly non-default `nasm` package.

The next thing to consider was what type of x86-64 assembly to emit. This compiler emits assembly in a style that C compilers call *large model* ([34], p. 34). This model is easier to emit, since it does not use RIP-relative addressing. Instead, full 64 bit addresses are used, together with indirect jumps and calls. When using the *large model*, indirect jumps are necessary because the x86-64 instruction set lacks any way to directly jump (or call) a literal 64 bit address, instead requiring the address to be loaded into a register and then using the value in the register as the jump or call target. This is a design limitation of the instruction set and is unlikely to ever change. Accessing global data also must be done by loading the address in a register and then using a register indirect move.

2.3. Debugging Machine Code on x86-64 Linux®

Part of developing any compiler that generates assembly code is debugging the generated output. Visual inspection can catch obvious problems, but anything subtle will usually only manifest itself at runtime, and traditionally problems at runtime are handled by using a debugger.

Unfortunately there are no full-featured free debuggers for machine-level code available for x86-64 Linux® that even approach the usability of the venerable Borland® Turbo Debugger® [35] for MS-DOS®. This makes debugging machine code quite difficult on that platform. While GNU's `gdb` [36] is a capable debugger, it's curses interface (`gdb-tui`) is painful to use. No GUI interface is included with `gdb`. Various GUI front -ends have been created such as the Data Display Debugger [37] (DDD), but these do not work well for machine code. The `gdb` debugger works well if DWARF [38] debugging code is emitted by the compiler, but is very hard to use on executables that contain no debugging information. This is not to say `gdb` is a bad debugger, it is just a very hard to use debugger for executable machine code that has no source-level debugging information.

The best available tool was Evan Teran's Debugger (`edb` [39]) which in 2013 did not show the SIMD registers or flags at all. In February of 2014 those were added to the register display, but by then most of the debugging of emitted floating point code had already been completed. Because of the difficult debugging environment, a register dumping subroutine was written and calls to it were hand-inserted into generated assembly code to aid in debugging the floating point expression evaluation code.

2.4. Compiler Structure

The compiler described in this paper uses a very traditional structure, with modules for a scanner, a parser, the symbol table, code generation, a line number module, and a runtime library. This compiler essentially has two passes: the first pass scans all the input and creates a token stream, and the second parses the token stream, with code generation as a side-effect. Each of the compiler modules is described in the following sections. See **??** to see how the modules interact.

2.4.1. Line Number Module

Since Minimal BASIC is true to the original BASIC language ([4], p. 109), it is both line-based and it requires each line to have a line number. These features required creation of a line number module. This module tracks all line numbers used. The line number module enables verifying that every branch to a line number with **`IF`**, **`GOTO`**, **`ON .. GOTO`**, or **`GOSUB`** actually references an existing branch target (line number) within the input BASIC program. This verification is done after completion of the parser. To make it possible, the line number module has two lists: a list of all possible jump targets that corresponds to each line number of the input Minimal BASIC program, and a list of all used jump targets from **`IF`**, **`GOTO`**, **`GOSUB`** and **`ON .. GOTO`** statements. After the parse is complete, both lists are fully populated and the verification to ensure every target in the second list exists within the first list can be done. This is done after the parse is completed in order to accommodate forward jumps. Scope of jump targets is also tracked and checked as described in paragraph 2.4.7.4 (**`FOR/NEXT`** loops).

2.4.2. Scanner Module

When deciding the approach to take for creating the scanner, the two possibilities considered were using a scanner generator or writing a hand-coded scanner. While using a scanner generator would make future maintenance easier, it would introduce a dependency on the scanner generator, and would require people studying this compiler to learn yet another tool, and probably complex regular expression syntax as well. Using a hand-coded scanner based on a deterministic finite state machine (DFSM) was simple to implement and had the benefit of avoiding the need of a scanner generator.

First the DFSM diagram to handle all the Minimal BASIC tokens was drawn on paper, and then this was converted to a matrix for inclusion in the C code. However, during development the scanning necessary for handling the rather bizarre rules for **`READ`** from **`DATA`** statements and **`INPUT`** from **`STDIN`** required modifying the scanner several times, and ultimately a second DFSM machine had to be added. This smaller DFSM can be seen in **??**. By using unique state numbers, it was possible to keep the tables for both DFSM implementations in the same array and use the same driver code. The DFSM has two types of accept: a normal accept that consumes the last byte seen, and a special accept that does not consume the last byte seen but instead lets that byte remain unprocessed so that it will be the first byte seen in the search for the next token. The matrix used has 99 states and 45 character classes.

With this compiler, the input Minimal BASIC program source is scanned and converted into a token stream **before** the assembly prologue is emitted, but the parse occurs **after** the prologue is emitted. This is done because the parser emits code in one pass as a side-effect of the parse and any flags used for conditional inclusion or exclusion of assembly support routines must be set before the compiler even begins parsing an ECMA-55 Minimal BASIC dialect source file. The method used in this compiler to determine whether to include support for things like **`READ`**/**`DATA`**, **`INPUT`**, *etc.* is to set flags during the scan when those tokens are found.

This compiler does not attempt to recover from errors. It will exit on the first error encountered. This keeps the design simpler and avoids the infamous cascading error problems exhibited by compilers that do attempt to continue after an error.

2.4.3. Types Of Variables

Minimal BASIC supports three types of variables: scalar string, scalar numeric, and array numeric. In this compiler, the scanner is used to determine the types of variables. Variable names cannot conflict with reserved words due to the language design, so when a variable name is scanned it is possible to know it is a variable name and not possibly a reserved word. String variable names consist of a single upper/case letter followed by a dollar sign (**`'$'`**) and the language only permits scalar string variables. Numeric scalar variables can have names of a single upper-case letter optionally followed by a single Arabic digit. Numeric array variables can have names of a single upper-case letter only, and they must be followed by subscript specification that is within parenthesis. These characteristics of variable names allow the scanner to determine the types of variables. For a variable name which is a single letter, the scanner will assume it is a numeric scalar variable. However, if a token is accepted and it is a left parenthesis, and the previous token is a numeric scalar variable, then the scanner will change that previous token id from numeric scalar to numeric array.

2.4.4. Parser Module

After the scanner was completed, work on the parser began. Again a decision had to be made whether to use a parser generator or write a hand-coded parser. Research revealed that the two most advanced free and open source C compiler toolchains used for Linux® programming, *`clang`*/LLVM and *`gcc`*, both use hand-coded recursive-descent parsers, which indicates that hand-coded recursive-descent parsing is an accepted technique even in state-of-the-art production quality optimizing compilers. For the ECMA-55 Minimal BASIC compiler described in this paper, a hand-coded recursive-descent parser was written. For people who use the source code of this compiler, this avoids any dependencies on a parser generator and the need to learn yet another language for input to a parser generator. Since Minimal BASIC is line-based and not stream-based, and there is no requirement for declarations before code, learning the type of variables might seem tricky. However, as explained previously, the scanner determines variable types before the parse begins.

2.4.5. Numeric Support With Exceptions

While working on the scanner and parser, research on creating the required runtime library and numeric expression evaluation took place in parallel. Because a strictly conforming Minimal BASIC implementation must generate code that handles floating point exceptions [40], but C and C++ generated code from *gcc* and *clang*/LLVM do not, learning how to deal with floating point took a long time. The only book on x86-64 Linux® assembly available [41] simply ignored floating point exceptions altogether. The freely available Intel® documentation [24] was thorough, but lacked any meaningful examples. Internet searches turned up little practical advice, and the required techniques were learned through tedious trial and error.

To complicate issues further, x86-64 processors have two different floating point units (FPUs), the legacy 80×87 and the newer SIMD unit that was created for Multi-Media Extensions (MMX), and later extended for Streaming SIMD Extensions (SSE). Both floating point units were designed to implement the IEEE 754-1985 [42] standard. This SIMD FPU also supports the new Advanced Vector Extensions (AVX) on Haswell and newer Intel® CPUs. Intel® has stopped short of removing the 80×87 instructions, but their documentation strongly encourages use of SIMD math. Since this compiler does not attempt to auto-vectorize any code, either FPU would work. In the end, code generation using SSE instructions was chosen since Intel® promotes them as the replacement for the legacy 80×87 instructions. This choice meant creating the runtime library was more difficult, since the SSE instruction set does not include many of the features that made the 80×87 great, like support for transcendental functions. Because only an old machine with an Intel® Core™ 2 Duo (E4700@2.60 GHz) was available for development, no SIMD features greater than SSSE3 were used.[2] The Conroe series processors do not include support for SSE4.x or newer instructions, which was first made available for desktops in Core™ 2 Duo Wolfdale processors in January of 2008 [43] as part of the Penryn 45mm feature set [44].

An initial attempt was made to write the required mathematical functions (**`SIN()`**, **`COS()`**, **`TAN()`**, **`SQR()`**, **`EXP()`**, **`LOG()`**) using SSE instructions to implement Taylor series, then later minimax polynomial approximations [45–47] were tried, but achieving the required accuracy turned out to be quite tricky [48]. Some papers suggest other algorithms that might have worked better [49], but did not have source code for implementations available for download. The Netlib FDLIBM [50] library provides the functions for free in C form and was considered. However, this Minimal BASIC compiler implementation required SSE assembly code. Hand-optimizing the compiled libm code would be possible in theory, but the many routines in that library are tightly coupled and this was difficult to realize in practice. This prompted the decision to use Naoki Shibata's SIMD Library for Evaluating Elementary Functions [51] (SLEEF [52]) code, which is well tested and is already optimized for SSE. It includes

[2] Initial SSE4 support has now been implemented. See "New Developments" on page 101 for details.

routines that were used as the basis for the following Minimal BASIC runtime library functions: **`SIN()`**, **`COS()`**, **`TAN()`**, **`ATAN()`**, **`LOG()`**, **`EXP()`**, and **`POW()`**. While the SLEEF code was technically written in C, it actually uses SSE intrinsics to generate good SSE code and the resulting assembly was relatively easy to incorporate into the Minimal BASIC compiler's runtime library.

After the difficult experience with the floating point, it was decided that for the runtime library it would be wise to use code for the random number generation that was already known to be good. After some searching, Bob Jenkins' ISAAC-64 [53,54] was found. While ISAAC-64 is only available as C code, converting it to assembler with a C99 compiler and then tweaking the generated code a bit resulted in the required x86-64 assembly runtime library routines. This was the basis for the **`RND`** runtime library function.

During initial testing of the transcendental functions *`printf()`* from GNU C library (glibc [55]) was used. However, to generate static executables that would be independent of the C standard library required that the runtime library include routines for both input and output of floating point numbers. Again, research revealed that writing correct code for either input [56] or output [57,58] is amazingly complex and the best way forward was to use proven, maintained code from David M. Gay [59]. While Mr. Gay's code [60] is freely available, documentation on how to use it is scant, and the default settings did not result in behavior that matched the requirements for Minimal BASIC. It took a lot of experimentation to ultimately discover how to adjust that code to work as required to meet the rules for Minimal BASIC.

Ultimately the numeric conversion code had to be used twice with slightly different modifications, once for use in the runtime library for support of the **`INPUT`** statement in generated executables, and once again for the compiler itself when processing **`DATA`** statements. This was necessary to ensure numeric constants were converted from ASCII to floating point values in the same way by the compiler for **`DATA`** statements (used by **`READ`** statements) and by the runtime for **`INPUT`** statements. Mr. Gay's code is available in C, and a C version was used for processing the **`DATA`** statements, but an x86-64 assembler version was required for the runtime library. Like the SLEEF code and ISAAC-64 code, this was created by compiling with a C99 compiler to assembly, and then modifying the generated assembly code by hand. Changes were required because *`errno`* is a macro in GNU libc which was used by the compiler, but was a normal integer when used in the runtime library. Also, the runtime library has no dynamic memory support, so the use of *`malloc()`* and *`free()`* had to be removed from that version.

2.4.6. Symbol Table Module

The symbol table module is interesting since it was possible to take advantage of the fact that most variables in Minimal BASIC are global (all but those used as single-line function parameters), and the set of possible names of those variables is finite and small. Using that knowledge, it was possible to create a module that pre-loaded entries for all possible global variable names in sorted order. Each entry contained a row with a flags to indicate whether the variable was actually used in the program initially set to false. The flags were updated during compilation. Also, variable names with a single letter from A to Z could be used for either arrays or scalars, so the type of these variables was initially indeterminate, and the type was determined during compilation by examining the context in the scanner as described previously. The ECMA-55 Minimal BASIC standard recommends implementing detection of attempts to read a value from an uninitialized variable, and this compiler does this by using a signaling not-a-number special value (SNaN) for numeric values initially, and a NAK (byte value of 21) as the first byte of strings initially. Those values cannot be legally entered into a Minimal BASIC program, so if they are encountered at runtime then they must be uninitialized variable references.

The compiler generates variables used in the **`FOR`** statement and also for the constants in **`DATA`** statements. This is necessary because the limit and step for the loop are computed only once before the loop begins, and those values are needed to determine when to exit the loop, and how to update to the index variable for each iteration of the loop body. It also must generate constants for string and numeric literal values. These are not in the same symbol tables with the global variables explicitly used in the Minimal BASIC program, but are in separate, dedicated lists. This compiler uses linked lists for this, which for large programs might be a bottleneck in compilation speed. Minimal BASIC is limited to 9999 line programs, so the O(n) search time is not bad in practice because n is small. Tail pointers are used so actually adding entries is O(1). However, the lists are searched first since the compiler performs constant merging for string literals, numeric literals, and **`DATA`** statement values. The entire test suite runs in about 30 seconds on the author's development machine (an ancient 2.6 GHz Intel® Core™ 2 Duo E4700) so linked lists were deemed good enough for this initial implementation of the compiler. The constant merging is done logically in three separate pools, one for numeric literals, one for string literals, and one for **`DATA`** literals. Using three separate pools was easier than using one combined pool, but it does mean that if a program has a string literal "HELLO" and a data item quoted string "HELLO" there will still unfortunately still be two occurrences in the program. Combining the three pools was not done because the **`DATA`** numeric literals are pre-converted to binary form, and all **`DATA`** literals have a type and length that are stored in special blocks, so they do not generate the same assembly, and thus merging between **`DATA`** and non-**`DATA`** literals was not attempted. The effect of this missed optimization was deemed small enough that it was not worth the extra complexity to implement it.

This BASIC must have all variables initialized in order to detect use of uninitialized variables. Initially this was done by explicitly initializing each variable and these were in the .data section. However, this creates a larger executable. Now the compiler places numeric and string variables in the .bss section, which is not actually included in the on-disk executable file. The required initialization is now done automatically when the program begins.

2.4.7. Code Generation Module

The knowledge gained while working on the runtime library support helped when the code generation module was written. However, occasionally there were strange floating point exception errors as a result of assumptions made by the Naoki Shibata's SLEEF code functions in the runtime library that did not match the assumptions made by the code emitted by the code generator. The SLEEF code was designed to be used in C where floating point exceptions are ignored, so the flags were not carefully managed by the SLEEF code. Eventually with the help of some register dumping code, the problems were understood and the necessary adjustments were made.

One fundamental difference between the 80×87 FPU instructions and the newer SSE instructions is that the the 80×87 code uses stack-based math and the SSE code uses register-based math. Taking full advantage of the register-based math would require implementing a complex register allocation and spilling algorithm. To sidestep the need for that due to the extremely short time available to finish the compiler, it was decided that simple RISC-like stack-based math would be used. During implementation, however, the stack-based method turned out to cause severe problems when using the system stack because the AMD64 ABI [34] rules about the stack alignment are very strict and keeping the stack aligned correctly while pushing and popping SSE registers on the system stack was difficult.

To solve this problem, a separate runtime stack dedicated solely to the numeric expression processing code was created. This separate stack solution was simple to implement and works very well. The benefits for the compiler author were that the easy to implement stack-based expression evaluation could be used and unlike the real hardware 80 × 87 stack, the new runtime stack size was not artificially limited in size. Using a dedicated operand stack has the added benefit that even bad stack errors with math operands do not break the system stack used for returns and local variables, easing debugging of the expression code generation. Arithmetic expression code is parsed and operands are pushed as encountered, and the math is done using standard postfix evaluation.

There is a non-trivial runtime cost associated with the stack-based expression evaluation solution, and this was an engineering compromise that traded runtime performance for implementation ease. Rather than open-coding the push and pop actions, assembly language macros were used. Macros were also used for the standard math operations add, subtract, multiply, divide, and power. The macros are used to keep the generated assembly code easy to read. Each arithmetic floating point operation is complicated by the necessary exception handling and if it was emitted directly then reading generated code for expressions would become very tedious. By using macros instead of functions, the actual machine code emitted after assembly has no runtime call/return penalty. The third-party libraries like SLEEF and David Gay's code do not use this private numeric operand stack.

2.4.7.1. Examining The Actual Translation

A traditional compiler is essentially a batch translator from some input language to some output language. To explain the translation results of the compiler described in this paper, the following sections will show snippets of code in the input language (ECMA-55 Minimal BASIC) followed by the corresponding code in the output language (x86-64 instruction set assembly language for the GNU assembler's default AT&T dialect). Interspersed with the code snippets will be commentary describing some of the details of the translation. This format is used to allow the reader to actually see the translation occurring. Readers can view the assembly output together with the ECMA-55 Minimal BASIC input and the correspondence between them can be more easily understood.

2.4.7.2. Arithmetic Expression Evaluation

To understand the arithmetic expression evaluation code that is generated by this compiler, consider this Minimal BASIC program:

```
10 LET A=4
20 LET B=3
30 PRINT A/B+2/3
40 END
```

The lines of generated assembly code for line 30 of the Minimal BASIC program show the use of the private floating point operand stack, and also show the code for dealing with uninitialized variables.

```
.LLINE0030:
        movabsq $.LCURLINENO, %rax
        movq    $30, (%rax)
        floatmem_to_floatreg A,%xmm0,'A'
        pushxmm 0
        floatmem_to_floatreg B,%xmm0,'B'
        pushxmm 0
        binary_divide
        floatlit_to_floatreg .LFLIT0002,%xmm0,2
        pushxmm 0
        floatlit_to_floatreg .LFLIT0001,%xmm0,3
        pushxmm 0
        binary_divide
        binary_add
        popxmm 0
        movabsq $printfloat, %rax
        callq   *%rax
        movabsq $outputbuf, %rax
        callq   *%rax
```

Ideally the compiler would compute the value of 2/3 and avoid emitting the division for that part of the expression, but this non-optimizing compiler does not implement constant folding.

2.4.7.3. **IF** Statement

To support conditional branching, Minimal BASIC has the **IF** statement. The target line number is specified after the **THEN** keyword, and if the logical expression between the **IF** and **THEN** keywords is true, the branch is taken. Otherwise, execution of the program continues with the next line following the **IF** statement in the program. Consider this simple program:

```
10 INPUT X
20 IF X>50 THEN 50
30 PRINT "NOT MORE THAN 50"
40 GOTO 60
50 PRINT "MORE THAN 50"
60 END
```

Minimal BASIC does not support an **ELSE** but this sample program shows that **ELSE** is not actually necessary to implement an either/or branch. The generated assembly code for line 20 is:

```
.LLINE0020:
        movabsq $.LCURLINENO, %rax
        movq    $20, (%rax)
        floatmem_to_floatreg X,%xmm0,'X'
        pushxmm 0
        floatlit_to_floatreg .LFLIT0000,%xmm0,50
        pushxmm 0
        popxmm  1    # rhs expression value
        popxmm  0    # lhs expression value
        comisd  %xmm1, %xmm0
        jp      0f   # LHS and/or RHS was NaN
        jbe     1f
        movabsq $.LLINE0050,%rax
        jmpq    *%rax
0:
        movabsq $.Luninitialized,%rax
        jmpq    *%rax
1:
```

The left-hand expression "**X**" is processed and the result is left on the stack on lines 1178 through 1179. Then the right-hand expression "**50**" is processed on lines 1000 through 1013. The comparison for the greater than logical operator "**>**" occurs on lines 1182 through 1184, and the branch for the case when the logical expression evaluates to false is on line 1186, and the true branch is on lines 1187 through 1188. Lines 1185 and 1189 through 1191 are for error checking.

2.4.7.4. **FOR** and **NEXT** Statements

One statement pair exists in Minimal BASIC that is split across multiple lines, and that is the **FOR/NEXT** pair. Minimal BASIC also has rules to prevent branching that violates a **FOR/NEXT** pair's scope. This was tricky to enforce since a jump from inside a **FOR** is permitted as long as the code flow returns back to inside the same **FOR** loop before (or at) the corresponding **NEXT** statement. A **GOSUB**[3] followed by a **RETURN** should be used instead of a pair of **GOTO** statements for this, but that is not strictly required by the language specification. Also, jumping over the initial **FOR** and then hitting a **NEXT** is an error. Keeping track of the loop index variable is done using a dedicated **FOR** stack. This is also used to ensure that an inner **FOR** does not use any active index variable from an outer **FOR** loop. Consider this simple program:

```
10 FOR I=1 TO 10 STEP 2
20 PRINT I
30 NEXT I
40 END
```

The generated assembly code for line 10 is:

```
1113  .LLINE0010:
1114          movabsq $.LCURLINENO, %rax
1115          movq    $10, (%rax)
1116          # compute expression and push custom FOR loop with index I start value
1117          floatlit_to_floatreg .LFLIT0000,%xmm0,1
1118          pushxmm 0
1119          # compute expression and push custom FOR loop with index I limit value
1120          floatlit_to_floatreg .LFLIT0001,%xmm0,10
1121          pushxmm 0
1122          # compute expression and push custom FOR loop with index I increment value
1123          floatlit_to_floatreg .LFLIT0002,%xmm0,2
1124          pushxmm 0
1125          beginfor I,.LFORINCREMENT0000,.LFORLIMIT0000,.LFORTEST0000,.LFORDONE0000
```

The **FOR** loop has three expressions that are evaluated when the loop begins. These are the start value, the limit value, and the increment value. These are encountered in that order by the parser, so as each is encountered the expression code is generated, which when run leaves the result of the expression on top of the stack. The first thing the beginfor macro does is move those values from the stack to variables. The start value is stored in the normal global loop index numeric variable. The other two values are stored in numeric variables generated by the compiler that are not available to the Minimal BASIC program source code. One is used for the loop limit in the pre-test, and the other is used for the loop increment. The code generator is responsible for creating those two variables and the two required labels. The first

[3] See "**GOSUB** and **RETURN** Statements" on page 85.

label is for the loop pre-test which is the jump target used by the **NEXT** statement after incrementing the loop index. The second label is used in the pre-test as a jump target when terminating the loop. These generated variable names are returned to the parser and stored on the compile-time **FOR** stack so that they can be used later when processing the **NEXT** statement. The generated assembly code for the loop terminating **NEXT** on line 30 is:

```
.LLINE0030:
        movabsq $.LCURLINENO, %rax
        movq    $30, (%rax)
        endfor I,.LFORINCREMENT0000,.LFORTEST0000,.LFORDONE0000
```

As can be seen in the example, the beginfor and endfor macros are used to keep the assembly code easier to read in the main program. The macros for use with 64 bit floats are:

```
        .macro beginfor ndex_var:req,inc_var:req,limit_var:req,tst_lbl:req,done_lbl:req

        .section .rodata,"a",@progbits

        .type .Lbeginfor\@_msg, %object
.Lbeginfor\@_msg:
        .asciz "\ndex_var\()"
        .size .Lbeginfor\@_msg, .-.Lbeginfor\@_msg

        .section .text,"ax",@progbits

        popxmm 0
        comisd  %xmm0, %xmm0
        jnp     0f
        # referenced scalar \inc_var\() was NaN, uninitialized
        movabsq $.Luninitialized_forinc_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
0:
        movabsq $\inc_var\(), %rax
        movsd   %xmm0, (%rax) # store FOR loop with index \ndex_var\() increment value
                              # \inc_var\()
        popxmm 1
        comisd  %xmm1, %xmm1
        jnp     1f
        # referenced scalar \limit_var\() was NaN, uninitialized
        movabsq $.Luninitialized_forlimit_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
1:
        movabsq $\limit_var\(), %rax
        movsd   %xmm1, (%rax) # store FOR loop with index \ndex_var\() limit value in
                              # \limit_var\()
        popxmm 2
        comisd  %xmm2, %xmm2
        jnp     2f
        # referenced scalar \ndex_var\() was NaN, uninitialized
        movabsq $.Luninitialized_forindex_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $.Lbeginfor\@_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
2:
        movabsq $\ndex_var\(), %rax
        movsd   %xmm2, (%rax) # store FOR loop index var \ndex_var\() initial value
\tst_lbl\():                   # FOR loop with index var \ndex_var\() test code starts here
        pxor    %xmm3,%xmm3   # xmm3=0
        movabsq $\ndex_var\(), %rax
        movsd   (%rax),%xmm0  # xmm0=\ndex_var\() (FOR loop index variable)
        comisd  %xmm0, %xmm0
        jnp     3f
        # referenced scalar \ndex_var\() was NaN, uninitialized
        movabsq $.Luninitialized_forindex_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $.Lbeginfor\@_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
3:
        movabsq $\limit_var\(), %rax
        movsd   (%rax),%xmm1  # xmm1=\limit_var\() (FOR loop with index \ndex_var\()
                              # limit value)
        comisd  %xmm1, %xmm1
        jnp     4f
        # referenced scalar \limit_var\() was NaN, uninitialized
        movabsq $.Luninitialized_forlimit_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
4:
        subsd   %xmm1,%xmm0
        # xmm0=\ndex_var\()-\limit_var\() (FOR loop index var \ndex_var\()
        # FOR loop with index \ndex_var\() limit val)
        movabsq $\inc_var\(), %rax
        movsd   (%rax),%xmm2  # xmm2=\inc_var\() (FOR loop with index \ndex_var\()
                              # increment value)
        ucomisd %xmm3,%xmm2
        jae     5f            # if FOR loop with index \ndex_var\() increment \inc_var\()
                              #    is non-negative then goto 0 (positive increment)
                              # else (negative increment)
        movabsq $.LFLT_NEG_MASK, %r15
        movsd   (%r15), %xmm1
        pxor    %xmm1, %xmm0  # xmm0=(-xmm0)
5:
        ucomisd %xmm3,%xmm0
                              # IF (FORINDEX-FORLIMIT)*SGN(FORINCREMENT)<=0 THEN
                              #   GOTO FORLOOPBODY
        jbe     6f            # if ABS(\ndex_var\()-\limit_var\()) <=0 then goto 1
                              # (FOR loop with index \ndex_var\() body)
        movabsq $\done_lbl\(),%rax
        jmpq    *%rax         # else goto \done_lbl\() (exit FOR loop with
                              # index var \ndex_var\())
                              # FOR loop with index var \ndex_var\() test code ends here
6:                            # FOR loop with index var \ndex_var\() body begins here
        .endm

        .macro endfor ndex_var:req,inc_var:req,tst_lbl:req,done_lbl:req

        .section .rodata,"a",@progbits

        .type .Lendfor\@_msg, %object
.Lendfor\@_msg:
        .asciz "\ndex_var\()"
        .size .Lendfor\@_msg, .-.Lendfor\@_msg

        .section .text,"ax",@progbits

                              # FOR loop with index var \ndex_var\() body ends here
                              # FOR loop with index var \ndex_var\() increment starts here
        movabsq $\ndex_var\(), %rax
        movsd   (%rax),%xmm0
        comisd  %xmm0, %xmm0
        jnp     1f
        # referenced scalar \ndex_var\() was NaN, uninitialized
        movabsq $.Luninitialized_forindex_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $.Lendfor\@_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
1:
        pushxmm 0             # push FOR loop index variable \ndex_var\()
        movabsq $\inc_var\(), %rax
        movsd   (%rax),%xmm0
        comisd  %xmm0, %xmm0
        jnp     2f
        # referenced scalar \inc_var\() was NaN, uninitialized
        movabsq $.Luninitialized_forinc_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
2:
        pushxmm 0             # push FOR loop with index \inc_var\()
                              # increment variable \ndex_var\()
        binary_add
        popxmm  0
                              # increment FOR loop index variable \ndex_var\()
        movabsq $\ndex_var\(), %rax      # \ndex_var\() = \ndex_var\() + \inc_var\()
        movsd   %xmm0,(%rax)
                              # FOR loop with index var \ndex_var\() increment ends here
        movabsq $\tst_lbl\(), %rax
        jmpq    *%rax         # goto FOR loop with index var \ndex_var\()
                              # test label \tst_lbl\()
\done_lbl\():
        movabsq $\inc_var\(), %rax
        movabsq $SNaN,%r15
        movq    %r15,(%rax)   # \inc_var\() is now uninitialized again so
                              # illegal jumps into the FOR loop with index var
                              # \ndex_var\() can be detected.
        .endm
```

Note that at the end of the endfor macro, the **FOR** loop index variable (called inc_var in the macro) is set to the signaling NaN value ($SNaN). This ensures that if an attempt is made to jump into the loop after the **NEXT** statement has executed, that jump will generate an error. The compiler must generate slightly different macros when using 32 bit floats, but the macro invocation in the main program remains unchanged.

The compile-time **FOR** stack is managed in the parser during compilation, and does not exist in the generated assembly program. Each entry contains the index variable, the names of the compiler-generated variables generated to hold the increment and limit (which are computed exactly once at the beginning of the loop), the label for the test which is as the top of the loop, the label of the first line after the next code used to exit the loop, and a scope variable. One thing the parser must do is ensure that loops do not overlap. Consider this incorrect code:

```
10 FOR I=1 TO 10 STEP 2
20 FOR J=20 TO 5 STEP -5
30 NEXT I
40 NEXT J
```

The loops may nest, but not overlap. To enforce this the parser will use a scope entry on a dedicated **FOR** stack in the parser. The compiler will emit an appropriate error message if someone attempts to compile this example code, namely “`NEXT I but expecting NEXT J on line 30`”.

The scope variables contain a string that is reminiscent of a REXX (Restructured Extended Executor) stem variable ([61], p. 328), with dots between the scope level names, with the innermost scope last. The outermost scope is obviously ‘`global`’, but it is not stored. The parser module has a variable `current_scope` that contains the current scope level. After line 10 executes, the scope is logically ‘`global.I`’, but stored as ‘`.I`’. After line 20 executes, the scope is ‘`global.I.J`’. When line 30 is reached, the last component of the scope is ‘`J`’ but the variable specified in the **NEXT** is ‘**I**’ and the compiler knows there is an error and it knows the expected value to use in the error message.

Finally, the **FOR** stack is used to detect the case where no **FOR** statement is active but a **NEXT** is encountered. If `current_scope` is the empty string (logically ‘`global`’) and thus the **FOR** stack is empty, **NEXT** is not possible and an error message is printed in the form “`NEXT ? without corresponding FOR on line ####`” where the question mark is replaced by the loop index variable name, and the pound signs are replaced by the line number in the BASIC source program where the bad **NEXT** statement was encountered.

2.4.7.5. **GOTO** Statement

The ECMA-55 Minimal BASIC specifies two different statements using **GOTO**. The first is an unconditional branch and is simply **GOTO** followed by a line number. It is not permitted to use anything but an integer constant representing a line number that exists in the program as a jump target. Consider this Minimal BASIC program:

```
10 GOTO 30
20 PRINT "NEVER HERE"
30 PRINT "DONE"
40 END
```

The corresponding assembly code for line 10 follows:

```
1113  .LLINE0010:
1114          movabsq $.LCURLINENO, %rax
1115          movq    $10, (%rax)
1116          movabsq $.LLINE0030, %rax
1117          jmpq    *%rax
```

Lines 1116 and 1117 implement the **GOTO** using a register indirect jump as required for x86-64 large mode code.

2.4.7.6. **ON** .. **GOTO** Statement

To implement a multi-way branch the **ON** expression **GOTO** sequence is used. The arithmetic expression between the **ON** and **GOTO** keywords is evaluated and rounded to an integer value. If the value is 1, the program will branch to the first line number in the comma-delimited list of jump target line numbers that follow the **GOTO** keyword. If the value is 2, the program will jump to the second branch target, etc. If the value is less than one or greater than the number of specified jump targets then a fatal error occurs. Consider this Minimal BASIC program:

```
10 INPUT X
20 ON X GOTO 100,200,300
30 PRINT "UNREACHABLE"
40 STOP
100 PRINT "CHOICE 1"
110 GOTO 40
200 PRINT "CHOICE 2"
210 GOTO 40
300 PRINT "CHOICE 3"
310 GOTO 40
400 END
```

The corresponding assembly code for line 20 after passing through the peephole optimizer follows:

```
1175  .LLINE0020:
1176          movabsq $.LCURLINENO, %rax
1177          movq    $20, (%rax)
1178          floatmem_to_floatreg X,%xmm0,'X'
1179          cvtsd2sil %xmm0, %eax
1180          stmxcsr .Lnewmxcsr
1181          testl   $FE_INVALID,.Lnewmxcsr
1182          jz      0f
1183          jmp     1f
1184  0:
1185          decl    %eax
1186          cmpl    $2,%eax
1187          ja      1f
1188          jmp     *.LJT0000(,%rax,8)
1189  1:
1190          movabsq $.Longotofail_msg, %rsi
1191          movabsq $.Lprint_error_message, %rax
1192          jmpq    *%rax
1193
1194          .section .rodata,"a",@progbits
1195
1196  .LJT0000:
1197          .quad   .LLINE0100
1198          .quad   .LLINE0200
1199          .quad   .LLINE0300
1200
1201          .section .text,"ax",@progbits
```

The generated assembly code uses a multi-way branch using a jump table. Each element in the jump table is an assembly label for the generated code that corresponds to the Minimal BASIC source line specified in the **ON** .. **GOTO** statement. The assembly line 1199 has the label for the code of the first Minimal BASIC line number specified in the list of jump targets, "**100**", *etc.* Careful checking is done for the low-level conversion of the expression in the **ON** .. **GOTO** statement from a floating point value to an integer value done by cvtsd2sil. A range check is performed on lines 1187 through 1189 with a single test using the technique described in the section on arrays about array bounds checking to ensure that expression values refer to entries in the table. For instance, if the expression on line **20** of the example program evaluated to a value less than one or a value greater than three in this example, that would be detected by the bounds checking.

2.4.7.7. **GOSUB** and **RETURN** Statements

Minimal BASIC supports simple subroutines that are parameter-less and are specified by line numbers instead of by name. This permits coding practices that are considered unsafe today, including multiple entry points, using only global variables to pass parameters, and no support for local variables. However, when the BASIC language was designed it was intended to be a simple alternative to assembly and even today those practices are still considered normal when coding in assembly language.

The **GOSUB** keyword is followed by a jump target line number. When the statement is executed the program saves the line number of the next line after the **GOSUB** statement and then branches to the specified line number. When **RETURN** is executed, the program branches back to that saved line number. **GOSUB** can be be called more than once without a **RETURN**, and this nesting requires the ability to save many return addresses. In this compiler's implementation, the **GOSUB** and **RETURN** sequence uses a runtime private address stack and two assembly macros when generating the assembly code. Using a private address stack allows detecting the case when a **RETURN** is attempted at runtime when no previous **GOSUB** occurred. Macros are used to keep the assembly code easier to read. These are expanded during assembly, and thus do not incur the additional runtime overhead of a function call. For example, consider this simple program:

```
10 LET A=5
20 LET B=10
30 GOSUB 60
40 PRINT C
50 STOP
60 LET C=A+B
70 RETURN
80 END
```

Here is the code for the **GOSUB 60** on line 30:

```
1127  .LLINE0030:
1128          movabsq $.LCURLINENO, %rax
1129          movq    $30, (%rax)
1130          gosub 0f .LLINE0060
1131  0:
```

Lines 1128 and 1129 are used to store the line number in a global variable used by error messages. The gosub macro call on line 1130 has two arguments, the first is the return address, and the second is the jump target. By using a local label 0, the compiler leaves it to the assembler to compute the jump location to push. The work with the address stack is abstracted away in the macro, leaving the code for line 30 easy to read. The code for the **RETURN** on line 70 is also simple:

```
1157  .LLINE0070:
1158          movabsq $.LCURLINENO, %rax
1159          movq    $70, (%rax)
1160          return
```

The BASIC **RETURN** statement is just translated into a call to the return macro on line 1160, again abstracting away the ugly stack management and keeping the generated code easy to read.

2.4.7.8. **DEF** Statement

The only other type of user-defined subroutine available for Minimal BASIC is a numeric function defined by the one-line **DEF** statement. Such a numeric function can optionally have one argument.

Functions created with **`DEF`** that have an argument are the only way to create local variables in Minimal BASIC; all other variables are global. A local variable created in this way will shadow any global that exists with the same name.

To make emitting code for user-defined functions work even though all code generation occurs in one pass as a side-effect of the parse, output is temporarily diverted to a special separate file used to hold the code emitted for user-defined functions. Code for all user-defined functions will be appended to the same temporary file. The contents of the temporary file are later appended to the emitted normal assembly output file after the parse is complete and the entire input BASIC program has been processed.

Since nested functions are not supported, a simple global boolean flag *`in_udf_definition`* exists to signal whether we are currently generating code for a user/defined function or not. This also allows support for a parameter name which may conflict with a global of the same name by having the *`g_nvar()`* and *`g_navar()`* routines in the parser check for the parameter name before attempting to lookup a global numeric variable since the parameter name will shadow a global of the same name. Those two routines are used to emit code for reading the values of scalar and array numeric variables during expression handling and pushing those values onto the dedicated floating point stack.

Consider this Minimal BASIC example program:

```
10 DEF FNA(Y)=Y*Y*Y
20 PRINT FNA(1024)
30 END
```

Here is the code used on line 20 to call **`FNA`** with an argument of **`1024`** and display the answer (after the peephole optimization):

```
1116  .LLINE0020:
1117          movabsq $.LCURLINENO, %rax
1118          movq    $20, (%rax)
1119          floatlit_to_floatreg .LFLIT0000,%xmm0,1024
1120          movabsq $FNA,%rax
1121          callq   *%rax
1122          movabsq $printfloat, %rax
1123          callq   *%rax
1124          movabsq $outputbuf, %rax
1125          callq   *%rax
```

The actual call to **`FNA`** is on lines 1120 and 1121. The function expects the value of the read-only parameter to be passed in the $\%$xmm0 register. The return value of the function will be passed back in the same register. Here is the assembly that is generated in response to the user-defined function **`FNA`** on line 10:

```
6618          .section .data,"aw",@progbits
6619
6620  .LFNA_Y:
6621          .quad SNaN  # 8 bytes, actually double
6622          .size .LFNA_Y, .-.LFNA_Y
6623
6624          .section .text,"ax",@progbits
6625
6626          .type   FNA, @function
6627  FNA:
6628          pushq   %rbp
6629          movq    %rsp, %rbp
6630          movabsq $.LFNA_Y, %r15
6631          movsd   %xmm0, (%r15)      # store function argument Y
6632          floatmem_to_floatreg .LFNA_Y,%xmm0,'Y'
```

```
6633        pushxmm 0
6634        floatmem_to_floatreg .LFNA_Y,%xmm0,'Y'
6635        pushxmm 0
6636        binary_multiply
6637        floatmem_to_floatreg .LFNA_Y,%xmm0,'Y'
6638        pushxmm 0
6639        binary_multiply
6640        popxmm 0     # pop stack into xmm0 for return
6641        movq    %rbp,%rsp
6642        popq    %rbp
6643        retq
6644        .size FNA, .-FNA
```

First, on lines 6630 through 6631, the code for **`FNA`** will copy the argument to a compiler generated variable whose name is created by using the user-defined function name and the parameter name. In this example, that variable is named .LFNA_Y. This also frees up the %xmm0 register used in arithmetic expression evaluation. Then the arithmetic expression that comprises the body of the function is evaluated on lines 6632 through 6639. After the runtime expression evaluation is complete, the function's return value is left on top of the dedicated runtime floating point operand stack. This is then popped into %xmm0 on line 6640 just before the function returns.

2.4.7.9. Arrays

Minimal BASIC has support for simple numeric arrays of one or two dimensions. In the **`DIM`** statement, the dimensions must be specified by integer constants. Also, the standard requires that if an array is specified with a **`DIM`** statement, no references to that array can be made before the **`DIM`** statement. If the **`DIM`** statement is encountered a second time, it is ignored. If no **`DIM`** statement exists but an array reference is used, then the required array is created with an upper bound of 10. The lower bound is zero or one depending on the value specified in the **`OPTION BASE`** statement. If no such statement exists, the lower bound will be zero for all arrays. Once an array has been created, its size cannot be changed.

To ensure the array base limit is set only once, the parser has a global boolean flag *`can_option_base`* that gets set to false after an array reference, **`DIM`** or **`OPTION BASE`** statement is encountered. Another global boolean flag *`base_is_one`* is initialized to false, but set to true if an **`OPTION BASE 1`** is parsed. That flag is used by the code generator to indicate the proper lower limit for the bounds checking code emitted for every array access. The code generated for array access is easy to read. For an example showing this, consider the following Minimal BASIC program which has access of a one-dimensional numeric array:

```
10 LET A(1)=9
20 LET A(A(1)-7)=A(1)*2
30 PRINT A(2)
40 END
```

The generated assembly code for line 20 after passing through the peephole optimizer is:

```
.LLINE0020:
        movabsq $.LCURLINENO, %rax
        movq    $20, (%rax)
        floatlit_to_floatreg .LFLIT0000,%xmm0,1
        array_1D_to_floatreg A,%xmm0,'A',0,10,%r8,%r15
        pushxmm 0
        floatlit_to_floatreg .LFLIT0002,%xmm0,7
        pushxmm 0
        binary_subtract
        floatlit_to_floatreg .LFLIT0000,%xmm0,1
        array_1D_to_floatreg A,%xmm0,'A',0,10,%r8,%r15
        pushxmm 0
        floatlit_to_floatreg .LFLIT0003,%xmm0,2
        pushxmm 0
        binary_multiply
        popxmm 2    #  RHS expression value
        popxmm 0    #  array index expression value
        floatreg_to_array_1D A,%xmm2,%xmm0,0,10,%r8,%r15
```

This example shows two reads and one write to the vector array **A** as part of an assignment in a Minimal BASIC **LET** statement. Array index bounds checking is performed for all three array accesses. This program uses the default setting of zero for the first array subscript. Using **OPTION BASE 1** will make array indices of zero invalid, and will change the generated code for the bounds checking slightly. Note that the bounds checks are done with a single compare, even in the case where the **OPTION BASE 1** has been used. Consider this Minimal BASIC example code which has access of a two-dimensional numeric array:

```
10 OPTION BASE 1
20 LET A(1,1)=9
30 PRINT A(1,1)
40 END
```

Since the bounds checking is done at runtime, code must be generated for that. The generated assembly code for line 30 is:

```
.LLINE0030:
        movabsq $.LCURLINENO, %rax
        movq    $30, (%rax)
        floatlit_to_floatreg .LFLIT0000,%xmm0,1
        pushxmm 0
        floatlit_to_floatreg .LFLIT0000,%xmm0,1
        pushxmm 0
        popxmm  1 # column index
        popxmm  0 # row index
        array_2D_to_floatreg A,%xmm0,%xmm1,'A',1,10,10,%r8,%r9,%r15
        pushxmm 0
        popxmm 0
        movabsq $printfloat, %rax
        callq   *%rax
        movabsq $outputbuf, %rax
        callq   *%rax
```

This example uses a two-dimensional array and the array's first subscript for each dimension is one and not the default zero because of the use of the **OPTION BASE 1** on line 10 of the Minimal BASIC program. The array indices are compared to 9 and not 10 since the subscript values have already been decremented by one in an earlier line of the assembly. This technique of using a single unsigned compare [62] allows generating shorter code with half the conditional branches of naively checking the lower and upper bounds separately. The *clang*/LLVM and *gcc* compilers do not support simple array bounds checking code generation.[4] After completing the implementation in this compiler, the code

[4] Compiling with bounds checking support with those compilers requires using the address sanitizer system.

generated by the Free Pascal compiler (*fpc*) for range-checking of arrays using the *-Cr* option was inspected and it too uses the same technique, but uses subq \$1,%rX instead of decq %rX. That more general code using a subtract makes sense for that compiler since arrays do not start only on zero and one as in Minimal BASIC, and the \$1 can be replaced by the lower bounds of the array specified in the source program, but for Minimal BASIC the simpler decq instruction suffices.

2.4.7.10. Strings

Unlike modern languages, Minimal BASIC uses 7-bit ASCII as the character set for strings. The original language only allowed upper-case letters, but this compiler, as a documented extension, allows lower-case letters in strings and in comments as part of **REM** statements after the upper-case **REM** keyword. This compiler implements strings as zero-terminated strings, often referred to as ASCIIZ strings, and the runtime library includes subroutines for working with those strings called mystrlen, mystrcpy, and mystrcmp which correspond to the well-known C standard library functions *strlen()*, *strcpy()*, and *strcmp()* respectively. Consider this Minimal BASIC program fragment:

```
5 LET S$="HELLO"
10 PRINT S$
```

This example shows assignment of a string literal value on line 5, and then a read of a string variable on line 10. The generated assembly for these lines is:

```
.LLINE0005:
        movabsq $.LCURLINENO, %rax
        movq    $5, (%rax)
        movabsq $.LSLIT0000, %rdi  # %rdi=pointer to 'HELLO'
        pushsaddr
        popsaddr %rsi
        movq    %rsi,%rdi
        movabsq $mystrlen, %rax
        callq   *%rax
        cmpq    $MAX_STRING_BYTES,%rax
        jbe     0f
        movabsq $.Lstring_too_long,%rax
        jmpq    *%rax
0:
        movabsq $S$, %rdi
        movabsq $mystrcpy, %rax
        callq   *%rax
.LLINE0010:
        movabsq $.LCURLINENO, %rax
        movq    $10, (%rax)
        movabsq $S$, %rdi
        cmpb    $NAK,(%rdi)
        jne     0f
        # referenced scalar was uninitialized
        movabsq $.Luninitialized_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        # must create string in reverse order
        xorq    %rdi,%rdi
        movb    $'$',%dil
        shl     $8,%rdi
        movb    $'S',%dil
        shl     $8,%rdi
        movb    $32,%dil
        movabsq $printvarname,%rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
0:
        pushsaddr
```

```
1153          popsaddr %rdi
1154          movabsq $appendbuf, %rax
1155          callq   *%rax
1156          movabsq $outputbuf, %rax
1157          callq   *%rax
```

The code for string assignment calls the mystrlen and mystrcpy subroutines and uses the dedicated string stack which contains the addresses of the string literals. Since the compiler does not support any string operators or string functions, the code seems overly complex. However, once string expressions are implemented (part of "Full BASIC") this will be needed, and it was decided to go ahead and implement this so it does not have to be redesigned later. At least the provided stand-alone, special-purpose peephole optimizer can remove the redundant pushsaddr/popsaddr sequences. On line 1134 of the generated assembly you can see the comparison of the first byte of the string with the ASCII NAK value. If it is not equal, the code jumps to local label 0 and proceeds to append the value stored in the scalar string variable **S$** to the output buffer by calling the appendbuf subroutine, followed finally by calling the outputbuf subroutine to terminate the output buffer and force an actual print to STDOUT. If the first byte of the string is NAK, then an error message is displayed that includes the name of the variable. Lines 1152 and 1153 will be removed when this code is processed by the *peephole* optimizer program[5].

2.4.7.11. **INPUT** Statement

As part of the support for **INPUT**, direct kernel system calls are made. This is done to avoid any dependency on GNU libc. If this compiler is ever ported to another platform, or another operating system using the same hardware, the system calls will have to be changed to match the host operating system. Even 32 bit and 64 bit Linux® on x86 and x86-64 have different system call numbers and mechanisms for the low-level *read()* and *write()* system calls.

The first implementation of the *read()* code used for **INPUT** statements failed when using shell redirection, but the final version works correctly in that case. This was one of many problems detected while using the helpful NBS test suite. Another problem was the fact that input typed as a response to an **INPUT** statement must echo when input is redirected for batch mode execution, but must not echo when input is typed interactively from the keyboard. Code for detection of whether the code is running on a tty or not was added to solve this problem.

The semantic rules for the **INPUT** statement are non-trivial. Essentially, at runtime the program must prompt the user for input and then wait for an entire line of input. After the line input is complete, the program must then scan it into tokens. The tokens that can be valid are:

(1) number (integer or real)
(2) quoted string
(3) unquoted string
(4) comma
(5) end-of-line (newline)

[5] See "Peephole Optimizer" on page 99

The last token must be an end-of-line. The comma serves as the delimiter between input items, except when it is within a quoted string. The number of items must match the number of variables specified in the **`INPUT`** statement. Then the type of each item is checked to ensure it can actually be stored in the corresponding variable specified in the **`INPUT`** statement. A number can always be stored in a scalar string variable or a numeric scalar or array variable. Quoted and unquoted strings can be stored in scalar string variables. But no input item can be stored in its corresponding variable until all type checking for all input items is complete and no type errors were found. Only after all type checking is successful can the input items be copied to the corresponding variables specified in the **`INPUT`** statement. If any error in the number of items or type of items occurs, the program must flush the input buffer and re-prompt the user for the input.

The method to actually generate code to implement those semantics uses two temporary output files for assembly code. It also requires runtime temporary holding hidden variables for each of the items that can be input, since the data cannot be stored in the variables specified in the **`INPUT`** statement until all type checking is successful. Additionally, code for a DFSM scanner is required as part of the runtime support (see **??**). The generation of code in the *`g_inputstmt()`* function of the parser to implement an **`INPUT`** statement works like this:

(1) Initialize the format string to the empty string and open two temporary files.

(2) For each variable in the **`INPUT`** statement do:

  (a) For numeric variables, emit to the first temporary file code that converts the ASCIIZ holding area string for this numeric item to a floating point representation and stores that back to the temporary holding area, and if an overflow occurs during the conversion, then the code must jump back to the start of the statement. Append an 'N' to the format string. Emit to the second temporary file code that copies the floating point value from the temporary holding area to the actual variable.

  (b) For string variables, append an 'S' to the format string. Emit to the second temporary file code that copies the ASCIIZ string value from the temporary holding area to the actual variable.

(3) Add the format string to the string literals list in the symbol table so that it will be emitted together with other string literals after the parsing is finished.

(4) Emit code to the main output file to call a function to print a prompt the user for input, do the line input to a buffer, and process that buffer, performing the scan into tokens left to right. As each number or string token is encountered, if the type is compatible, the data is moved to a hidden temporary storage area. Any time an error occurs, the code must jump back to the start of the statement. The code in this phase verifies the type of each item matches the required type from the format string created earlier. The assembly language routine that does all of this is called doinput, and it will reset the input stack pointer itself before processing input. It will also verify that the number of items input by the user matches the required number of variables, and if it does not, then it will force a jump back to the start of the statement. This check is possible by using the length of the ASCIIZ format string, since there is one byte for each variable.

(5) Rewind both temporary files.

(6) Append contents of the first temporary file to the main output file.

(7) Append contents of the second temporary file to the main output file.
(8) Close and remove both temporary files.

The name of the structure for the temporary holding areas is .Lrt_nput_stack. The pointer used to access elements in this stack is called .Lrt_nput_stack_top. The doinput subroutine uses that area like a stack. However, the other code emitted will access the temporary holding areas directly by treating the stack like a vector instead of popping the items. Consider this sample program line:

```
10 INPUT A,B$,C(1)
```

The generated assembly code for line 10 is:

```
.LLINE0010:
        movabsq $.LCURLINENO, %rax
        movq    $10, (%rax)
        movabsq $.LNFMT0010, %rdi
        movabsq $doinput, %rax
        callq   *%rax
# scalar numeric INPUT A (convert, check for overflow, save in temp slot)
        movabsq $.Lrt_nput_stack, %r15
        movabsq $0,%rax           # slot number 0 on .Lrt_nput_stack
        movabsq $38,%r9           # width of a record on .Lrt_nput_stack
        mulq    %r9
        leaq    1(%r15,%rax),%rdi # 1st arg is ASCIZ buffer
        xorl    %esi, %esi        # 2nd arg is NULL
        movabsq $strtod, %rax     # call code to do actual conversion from ASCIIZ to floating point
        callq   *%rax
        movsd    %xmm0, %xmm1
        # if xmm1 is +Infinity then overflow
        movabsq    $PInf,%r13   # move constant +INF into r13
        subq    $8,%rsp         # allocate scratch 8 byte area
                                # it is a union location for
                                # converting between 64 bit double and
                                # unsigned 64 int
        movq    %r13 ,(%rsp)    # temp = +INF
        movsd   (%rsp),%xmm0    # move +INF into xmm0
        addq    $8,%rsp         # deallocate scratch 8 byte area
        comisd  %xmm1,%xmm0
        jne     0f
        # got overflow
        movabsq $.Loverflow_msg,%rsi
        movabsq $exception_non_fatal,%r13
        callq   *%r13
        movabsq $.LLINE0010,%rax
        jmpq    *%rax
0:
        movabsq $.Lrt_nput_stack, %r15
        movabsq $0, %rax
        movabsq $38, %r9
        mulq    %r9
        movsd   %xmm1,1(%r15,%rax)
# scalar string INPUT B$ (do nothing right now, already in temp slot)
# array numeric INPUT C (convert, check for overflow, save in temp slot)
        movabsq $.Lrt_nput_stack, %r15
        movabsq $2,%rax           # slot number 2 on .Lrt_nput_stack
        movabsq $38,%r9           # width of a record on .Lrt_nput_stack
        mulq    %r9
        leaq    1(%r15,%rax),%rdi # 1st arg is ASCIZ buffer
        xorl    %esi, %esi        # 2nd arg is NULL
        movabsq $strtod, %rax     # call code to do actual conversion from ASCIIZ to floating point
        callq   *%rax
        movsd    %xmm0, %xmm1
        # if xmm1 is +Infinity then overflow
        movabsq    $PInf,%r13   # move constant +INF into r13
        subq    $8,%rsp         # allocate scratch 8 byte area
                                # it is a union location for
                                # converting between 64 bit double and
                                # unsigned 64 int
        movq    %r13 ,(%rsp)    # temp = +INF
```

```
1185          movsd   (%rsp),%xmm0    # move +INF into xmm0
1186          addq    $8,%rsp         # deallocate scratch 8 byte area
1187          comisd  %xmm1,%xmm0
1188          jne     0f
1189          # got overflow
1190          movabsq $.Loverflow_msg,%rsi
1191          movabsq $exception_non_fatal,%r13
1192          callq   *%r13
1193          movabsq $.LLINE0010,%rax
1194          jmpq    *%rax
1195  0:
1196          movabsq $.Lrt_nput_stack, %r15
1197          movabsq $2, %rax
1198          movabsq $38, %r9
1199          mulq    %r9
1200          movsd   %xmm1,1(%r15,%rax)
1201
1202          # INPUT ok, copy from temp locations to real locations
1203
1204   # scalar numeric INPUT A phase 2
1205          movabsq $.Lrt_nput_stack, %r15
1206          movsd   1(%r15),%xmm1
1207          movabsq $A, %r14
1208          movsd   %xmm1, (%r14)
1209   # scalar string INPUT B$ phase 2
1210          movabsq $.Lrt_nput_stack, %r15
1211          movabsq $38, %rax
1212          leaq    1(%r15,%rax),%rsi # 2nd arg is source
1213          movabsq $B$, %rdi  # 1st arg is destination
1214          movabsq $mystrcpy, %rax
1215          callq   *%rax
1216          floatlit_to_floatreg .LFLIT0000,%xmm0,1
1217          pushxmm 0
1218   # array numeric INPUT C phase 2
1219          movabsq $.Lrt_nput_stack, %r15
1220          movabsq $38, %r9
1221          movabsq $2, %rax
1222          mulq    %r9
1223          movsd   1(%r15,%rax),%xmm1
1224          pushxmm 1
1225          popxmm 2    #  RHS expression value
1226          popxmm 0    #  array index expression value
1227          floatreg_to_array_1D C,%xmm2,%xmm0,0,10,%r8,%r15
```

The code to call doinput is on lines 1131 and 1131. Lines 1134 through 1166 process **`A`**, a scalar numeric variable, doing the conversion and overflow check. Lines 1168 through 1200 process **`C(1)`**, an array numeric variable, doing the conversion and overflow check. The copying back phase starts on line 1204. The copy from the temporary hold area .Lrt_nput_stack[0] to **`A`** occurs on lines 1204 through 1208. The copy from the temporary holding area .Lrt_nput_stack[1] to **`B$`** occurs on lines 1209 through 1215. Finally, the copy from the temporary holding area .Lrt_nput_stack[2] to **`C(1)`** occurs on lines 1216 through 1227. The runtime library subroutines for supporting the **`INPUT`** statement were first written in C99, then compiled to assembly for use as part of the runtime library.

2.4.7.12. **`PRINT`** Statement

Like the support for **`INPUT`**, the code for **`PRINT`** does direct kernel system calls to avoid any dependency on GNU libc. The rules for **`PRINT`** statements to display numbers in Minimal BASIC, inherited from the original Dartmouth BASIC design ([4], p. 109), made the implementation of conversion between doubles and ASCIIZ strings non-trivial, so a slightly modified version of David M. Gay's code was used for that conversion. Essentially, numbers that fit in a print zone without using an exponent are printed without an exponent, but numbers that have too many digits to fit are printed with an exponent. If the number is positive, it will have a leading space, otherwise it will have a leading minus sign. When printing, a number will always have a trailing space.

The remaining rules for the **PRINT** statement are fully documented in the standard. In a nutshell, the argument to the **PRINT** statement is a sequence of print items separated by either semicolons or commas. Output is buffered and is output as entire lines. The output line width is 80 columns, and output will wrap to new lines if necessary. There are 5 print zones, each with 15 columns. If a semicolon is used as the separator between items, it means the items should be printed immediately after one another. If a comma is used, the output position is moved to the next print zone, again wrapping if required. There is a special built-in **TAB()** function that takes as an argument a column number, and instructs the runtime output system to move the current output column to the argument specified to tab after rounding it to the nearest integer value. If that column is less than the current output position, the current buffer is printed and cleared, and then the column position is used. If the column position specified is larger than the output width, then the new output position is computed by taking the remainder of integer division of the tab subroutine argument by the output width.

The runtime library code for the **PRINT** support was implemented in C99 first, and then compiled to assembler and used in the runtime library. The appendbuf procedure is used to display a string value. The printfloat procedure is used to display a numeric value. The nextzone procedure is used to move to the next **PRINT** zone to implement the comma separator. The outputbuf procedure is used to force output of the buffer when all of the items in the list have been displayed, except in the case where the final item is followed by a comma or semicolon. The tab procedure is used to implement the **TAB()** function. To see how these low-level library functions are used, consider this trivial program:

```
10 LET P=1.234
20 LET Q$="DEMO"
30 PRINT "00000000011111111112"
40 PRINT "12345678901234567890"
50 PRINT "P is";P,Q$;TAB(5);"FIVE"
60 END
```

Here is the actual output when this program is run:

```
00000000011111111112
12345678901234567890
P is 1.234     DEMO
    FIVE
```

As you can see, the comma moved the output to column 16, the second print zone. The call to **TAB(5)** forced the current line buffer to be output and cleared, and the final string literal **"FIVE"** starts in the fifth column of a new line. Here is the generated code for line 50 after being run through the simple peephole optimizer:

```
.LLINE0050:
        movabsq $.LCURLINENO, %rax
        movq    $50, (%rax)
        movabsq $.LSLIT0003, %rdi  # %rdi=pointer to 'P is'
        movabsq $appendbuf, %rax
        callq   *%rax
```

```
1157            floatmem_to_floatreg P,%xmm0,'P'
1158            movabsq $printfloat, %rax
1159            callq   *%rax
1160            movabsq $nextzone, %rax
1161            callq   *%rax
1162            movabsq $Q$, %rdi
1163            cmpb    $NAK,(%rdi)
1164            jne     0f
1165            # referenced scalar was uninitialized
1166            movabsq $.Luninitialized_msg, %rsi
1167            movabsq $printstring, %rax
1168            callq   *%rax
1169            # must create string in reverse order
1170            xorq    %rdi,%rdi
1171            movb    $'$',%dil
1172            shl     $8,%rdi
1173            movb    $'Q',%dil
1174            shl     $8,%rdi
1175            movb    $32,%dil
1176            movabsq $printvarname,%rax
1177            callq   *%rax
1178            movabsq $badxit, %rax
1179            jmpq    *%rax
1180    0:
1181            movabsq $appendbuf, %rax
1182            callq   *%rax
1183            floatlit_to_floatreg .LFLIT0001,%xmm0,5
1184            cvtsd2si %xmm0,%rdi
1185            movabsq $tab, %rax
1186            callq  *%rax
1187            movabsq $.LSLIT0004, %rdi  # %rdi=pointer to 'FIVE'
1188            movabsq $appendbuf, %rax
1189            callq   *%rax
1190            movabsq $outputbuf, %rax
1191            callq   *%rax
```

This shows output of the initial string literal on lines 1154 through 1156 using the appendbuf procedure. Then, because a semicolon separator was used, output continues in the next column for the numeric value of the **P** variable which is output using the printfloat on lines 1157 through 1159. Since the next separator is a comma, the code moves output to the next output zone using the nextzone procedure on lines 1160 through 1161. The output position then moves to the second zone which starts in column 16. Next the string variable **Q$** is checked to ensure it is initialized on lines 1162 through 1180. If it is, then it is displayed using the appendbuf procedure on lines 1181 through 1182. The next separator is a semicolon, so output will continue with the next available column. The call to move to column 5 with **TAB(5)** is done on lines 1183 through 1186. Since that column is less than the current output column, the currently buffered line will be output, the buffer will be re-initialized, and the output position will only then be moved to column 5 of the now blank output line. The next separator is a semicolon, so, the final string literal **"FIVE"** is output using the appendbuf procedure on lines 1187 through 1189. Since the **PRINT** statement has no trailing separator, the outputbuf procedure is called on lines 1190 through 1191 to print the buffer contents and re-initialize the print system again.

2.4.7.13. **READ**, **DATA**, and **RESTORE** Statements

Minimal BASIC allows embedding literal data into a program which can be conveniently accessed as an alternative to input from STDIN or a file. Comma-delimited literal values are stored in **DATA** statements. If executed directly the statements do nothing and control flow continues to the following statements. All the literals in all the **DATA** statements get placed into one internal list in the order in which they appear in the BASIC program source. To retrieve these values at runtime, a **READ** statement is used. The **READ** has a similar form to the **INPUT** statement, with a list of variables that will receive data immediately following the **READ** keyword. The compiled program keeps a hidden pointer that initially points to the first literal, and is advanced each time a variable is filled with data in a **READ** statement. It is a fatal error to issue a **READ** after all **DATA** elements have been read, so the programmer writing the BASIC program is responsible for keeping track of the number of literals stored in **DATA** statements. However, there is one special BASIC statement, called **RESTORE**, that can be used to reset the hidden internal pointer (.Ldata_item_ptr) used with **READ** back to the beginning of the list of literal values that were specified by **DATA** statements. Consider this example program:

```
10 DATA 5,"HELLO",1,1,2,"HELLO"
20 READ C
30 FOR I=1 TO C
40  READ S$
50  PRINT S$
60 NEXT I
70 RESTORE
80 REM READ IT ALL AT ONCE
90 READ C,A$,B,C,D,E$
100 PRINT A$,B,C,D,E$
110 END
```

This example uses a common BASIC technique of storing the number of data elements as the first literal value in the **DATA** statement. BASIC allows any literal stored in a **DATA** statement to be read into a string variable as a string. On line 70 the pointer is reset, and on line 90 the numeric values are read into numeric variables. While any numeric literal can be read into a string variable, it is a fatal error to read a string literal into a numeric variable. In this example "HELLO" and the value 1 are duplicated in the **DATA** statement. The constant merging will remove the duplicate data blocks, and the list of pointers just has both "HELLO" items point the the same data block and both 1 values point to the same data block. The list of pointers still has 6 pointers, so everything works. The code generation for the **READ** on line 20 is shown here:

```
.LLINE0020:
        movabsq $.LCURLINENO, %rax
        movq    $20, (%rax)
# scalar numeric READ C
        movabsq $.Ldata_item_ptr, %rax
        movq    (%rax), %rax
        movabsq $.Ldata_item_count, %rbx
        movq    (%rbx), %rbx
        cmpq    %rbx, %rax
        jb      0f
        movabsq $.Lout_of_data, %rax
        jmpq    *%rax
0:
        movabsq $.Ldata_items, %rbx     # base of data_items array of struct pointers
        movq    (%rbx,%rax,8),%rax      # load pointer value in slot rax
        cmpb    $2,(%rax)
        je      1f
        movabsq $.Lbad_number_read, %rax
        jmpq    *%rax
1:
        movsd   1(%rax),%xmm0           # ok, load double 2 bytes into that structure into xmm0
```

```
1152          movabsq $C, %rax
1153          movsd   %xmm0,(%rax)
1154          movabsq $.Ldata_item_ptr, %rax
1155          incq    (%rax)                # increment data_item_ptr in memory but not register
1156          movabsq    $PInf,%r15
1157          subq    $8,%rsp         # allocate scratch 8 byte area
1158                                  # it is a union location for
1159                                  # converting between 64 bit double and
1160                                  # unsigned 64 int
1161          movq    %r15 ,(%rsp)    # temp = +INF
1162          movsd   (%rsp),%xmm1    # xmm1 = +INF
1163          ucomisd %xmm1,%xmm0
1164          jne     2f
1165          jmp     3f
1166  2:
1167          movabsq    $NInf,%r15
1168          movq    %r15 ,(%rsp)    # temp = -INF
1169          movsd   (%rsp),%xmm1    # xmm1 = -INF
1170          ucomisd %xmm1,%xmm0
1171          jne     4f
1172  3:
1173          # overflow message
1174          movabsq $.Loverflow_msg,%rsi
1175          movabsq $exception_non_fatal,%r15
1176          callq   *%r15
1177  4:
1178          addq    $8,%rsp         # deallocate scratch 8 byte area
```

Because SSE2 lacks support for the PINSRQ instruction, to move data into the SSE register it must go through memory. To avoid using a global variable for this, on line 1157 a temporary is created on the system stack for the conversion to allow loading positive and negative infinity values used in overflow checking. That temporary storage is de-allocated line line 1178. In the 32 bit narrow mode, the same problem exists due to the lack of support for the PINSRD instruction. The table of pointers (.Ldata_items) used for the data items shows how the data item constant merging works. Each duplicated item from the **DATA** statement points to an already existing data block:

```
7220          .align 16
7221          .type .Ldata_items, @object
7222  .Ldata_items:
7223          .quad .LDATA0000
7224          .quad .LDATA0001
7225          .quad .LDATA0002
7226          .quad .LDATA0002
7227          .quad .LDATA0003
7228          .quad .LDATA0001
7229          .size .Ldata_items, .-.Ldata_items
```

On line 7226 which corresponds to the second numeric value 1, it uses .LDATA0002 so that it points to the data block created for the first instance of the numeric value 1 in the list. Similarly, on line 7228 it uses .LDATA0001 for the second "HELLO", and that is a pointer to the block created for the first "HELLO" that was the second item in the list as shown on line 7224.

The data block for a numeric item actually contains both string and numeric representations, as shown here for the first **DATA** item in the example, "**5**", from the Minimal BASIC source line number **90**:

```
7182  .LDATA0000S:        # string represenatation
7183          .asciz "5"
7184  .LDATA0000:
7185          .byte 2    # Type is: UQSTR
7186          .double  5  # numeric representation
7187          .byte 2    # length of ASCIIZ string pointed to by .LDATA0000S including NULL byte
7188          .quad .LDATA0000S   # pointer to string representation
7189          .size .LDATA0000, .-.LDATA0000
7190          .type .LDATA0000, @object
```

This avoids any need for conversion for numeric data items at runtime for the generated executable. The 2 on line 7185 represents the unquoted string type, which can be read into either a numeric or a string variable. The 2 on line 7187 is the length of the ASCIIZ string representation including the terminating NULL byte.

The other type of data block is for a quoted string:

```
7191  .LDATA0001S:        # string represenatation
7192          .asciz "HELLO"
7193  .LDATA0001:
7194          .byte 0    # Type is: QSTR
7195          .byte 6    # length of ASCIIZ string pointed to by .LDATA0001S including NULL byte
7196          .quad .LDATA0001S   # pointer to string representation
7197          .size .LDATA0001, .-.LDATA0001
7198          .type .LDATA0001, @object
```

The 0 on line 7194 represents the quoted string type, and the 6 on line 7195 is the length of the ASCIIZ string representation including the terminating NULL byte. Note that for type 0, no floating point version exists, because a quoted string cannot be read into a numeric variable.

2.5. Peephole Optimizer

In addition to the main compiler, a separate, stand-alone, special-purpose peephole optimizer [63] called `peephole` was implemented to remove redundant push/pop sequences on the dedicated runtime floating point and string address operand stacks. For the numeric expression evaluation code, pushxmm 0 followed immediately on the next line by popxmm 0 is removed. For the similar string expression evaluation code, pushsaddr followed by popsaddr %rdi sequences are removed. This trivial optimization pass works on the assembly code and thus allows the main compiler to generate easy to understand expression code, but still allows improving the actual executable that results. The optimizer is primitive and the window size is only two lines, but it works well enough to remove unnecessary assembly code macro sequences where something is pushed from a register onto one of the special-purpose operand stacks and then immediately popped back into exactly the same register.

In **??** the unoptimized code emitted from the compiler is in the left-hand column, and the corresponding code after passing through the special-purpose stand-alone `peephole` optimizer is in the right-hand column. The difference between the code before and after is the removal of the superfluous push/pop sequence. Those removed commands are macro invocations, not single instructions, so their removal has a more significant effect on performance and executable code size than it might at first appear.

3. Results and Discussion

3.1. Testing The Compiler

Since no other ECMA-55 Minimal BASIC compiler exists for x86-64 Linux® , no performance comparisons with alternative implementations are possible. However, the National Bureau of Standards (NBS) created a Minimal BASIC test suite and parts of it were available online. For the parts that were not, the author was able to obtain paper copies from Mr. Emmanuel Roche. The test suite contains 208 programs and running these helped with debugging of the compiler. After compiler development was feature-complete, six tests still did not pass. Several of the failures were a result of negative constants being interpreted differently with ECMA-55's grammar than with the ANSI grammar for which the tests were designed[6]. One test had a string longer than the maximum 18 bytes[7]. One test was for overflow at $\pm(\pi/2)$ in the **`TAN()`** function, but the SLEEF tangent function implementation never overflows in that case[8]. Two tests used uninitialized variables, which ECMA-55 recommends should be detected and flagged as errors, but ANSI may not care[9]. Those six tests were fixed, and a test harness was created using GNU `bash` shell in combination with `file`, `diff`, and GNU `make`. This allowed easily running the entire test suite every time a change was made, ensuring no regressions would occur. In retrospect it may have been better to implement the automatic test system earlier so bugs could have been noticed immediately instead of much later in the project.

One issue with using the NBS test suite is that it was written for 32 bit processors, not 64 bit processors. This means that the floating point constants do not have as many digits or cover as wide a range as is really possible for a machine using a 64 bit FPU. The initial version of the compiler was designed to generate output that would match the NBS test suite, but later a 'wide' mode was added that generates wider output possible when using 64 bit floating point values. Wide mode uses a 132 column output buffer and the five print zones use 25 columns each in that case. For wide mode, numeric output that uses exponents has three digit exponents instead of two digit exponents, and up to 16 significant digits instead of only 7. Since Minimal BASIC is a dead standard, and the NBS no longer exists, there will never be a 64 bit version of that test suite and conformance for the 64 bit case can never be definitively verified.

Once the compiler was stable and the NBS test suite passed, special care was taken to test the compiler itself using Valgrind's Memcheck [64] to verify there were no dynamic memory errors. The `clang`/LLVM address sanitation [65] implementation was used to verify there were no out of bounds array accesses. Also all integer types were converted to use the C99 fixed-width integer types in `inttypes.h` ([66], pp. 198–200) after some 32 versus 64 bit bugs were found in the compiler while testing with integer values too large to fit in 32 bits. Since the NBS tests were written for implementations without 64 bit support, these bugs were not seen while running the NBS Minimal BASIC test suite.

3.2. Late Developments

The author learned of the `pcc` compiler after the Minimal BASIC compiler was completed. It is another open source C compiler that can generate assembly code. It is directly descended from Stephen C. Johnson's 1970's-era compiler, but now has C99 and x86-64 support. The `pcc` compiler was used as an alternative to `gcc` by the BSD community, which prefers non-GPL licensed tools, before the recent dramatic rise in popularity of the LLVM toolchain. The assembly output of the `pcc` compiler is actually much easier to read than the output of `gcc` or `clang`/LLVM, and might have provided an easier method

[6] NBS Test programs 26 & 38.
[7] NBS Test program 100.
[8] NBS Test program 129.
[9] NBS Test programs 134 & 136.

of generating runtime assembly language routines from C99 source code. As expected, the ECMA-55 Minimal BASIC compiler functions correctly when compiled with `pcc`.

While writing this document the author learned that the Intel® Xeon Phi™, while touted as x86-64 compatible, actually cannot operate on XMM registers and lacks the CMOV instruction ([67], p. 659). Code was added to the program prologue emitted by the code generator to verify that SSE2, SSE3, SSSE3, and CMOV features are available and if they are not to exit with an error message before even attempting to do anything else. Now that Intel has implemented a CPU with an "extended subset" once, it may happen again, so these new extra checks are necessary as a safety measure.

In response to reviewer comments on drafts of this document, the code generation for arithmetic expressions was modified to use macros to abstract away the repetitive error checking when loading values from and storing values to memory. The actual machine code after expanding those macros is equivalent to the previous versions of code the compiler generated, and still verifies values are initialized and valid. These changes resulted in much easier to read assembly code.

In September of 2014 the author of this paper received a new machine with a Haswell refresh processor, which finally allowed adding SSE4 features to the compiler described in this paper. This was done by adding a `-4` switch to the compiler to trigger use of the PINSRQ and PEXTRQ instructions for 64bit floating point data transfer. These instructions avoid the need to use a temporary stack variable to move data between general-purpose registers and SIMD registers. Performance measurements showed that no discernible difference in runtime between the two methods exists on a machine with an Intel® Core™ i7-4790 CPU. However, the assembly code using SSE4 is easier to read and understand. The corresponding PINSRD and PEXTRD are used for 32 bit floating point data transfer. Since the compiler was designed and implemented before the new machine was available, the examples in this document were created without using the `-4` switch.

3.3. Teaching with the MinimalBASIC Compiler

Many textbooks are available which discuss creating compilers, but few actually target real, modern CPUs. Often compiler courses emphasize creating a compiler for a 'nice' input language generated for some 'ideal' CPU, or they target CPUs that have reasonable instruction sets but are not widely used for new general-purpose machines today, such as MIPS64. Some books detail a full production compiler, which is too complex for most students in an introductory course [68]. Other books are highly theoretical [69]. Real CPUs still in use today tend to be ugly, non-orthogonal, sensitive to memory alignment, *etc*. Intel's x86-64 has few general-purpose registers and only one instruction in 64 bit mode that can actually take a 64 bit literal, yet despite being truly awful to program in assembly, x86-64 is the dominant 64 bit instruction set in use today. Once students learn to generate code for the worst case x86-64 instruction set, any other stack-based CPU will be easy for them.

This compiler is *intentionally* much simpler than production compilers, yet accepts as input a simple, line-based, historical computer language and targets a real CPU for an operating system kernel that is in use on many servers today. It avoids the complexity inherent in a retargetable compiler by just compiling one input language for one target, and generating unoptimized code directly as a side-effect of the parse. No explicit abstract syntax tree or intermediate language is used, and no graph analysis is used. This makes it a good first compiler for students with no previous compiler experience to examine, since it works, the input and output are not idealized languages, and it targets the widely-used and ugly x86-64 instruction set for systems running a modern Linux 3.x kernel. The compiler itself is written in C99, which is available for almost any platform, and keeps things simple by avoiding object-oriented programming or any explicit graph representation of the parse tree.

One concern some instructors may have is that the design of this compiler does not allow for any data or control flow analysis and thus makes implementing almost any optimization impossible. This is a feature, not a fault, since this compiler is designed for teaching students with weak algorithmic and programming skills, not researchers or excellent students attending elite institutions. If the instructor's students are truly ready to learn about optimization, then they are ready for one of the many open source optimizing compilers like *`clang`*/LLVM, *`gcc`*, *`pcc`*, *`fpc`*, *etc*. The compiler featured in this paper is designed to teach people sustainable compiler skills that may not be optimal, but that are definitely practical, easy to remember, easy to use, implementation language agnostic, and good enough in practice for common small problems like processing configuration files, writing small expression evaluators for interactive applications, *etc*.

The compiler described in this paper is easily used for student projects. Examples of projects of varying difficulties for students would include:

- Add support for **`AND`**, **`OR`**, and **`NOT`** logical operators.
- Adding a **`WHILE`**/**`ENDWHILE`** loop statement pair.
- Modifying the symbol table, runtime library, and scanner by adding support for longer variable names.
- Adding runtime support, updating symbol table support, and changing the grammar for full string support from ECMA-116.
- Changing the code generator by targeting another operating system using the same CPU, for example Apple®'s OS X®.
- Improving the compiler by replacing linked lists with more scalable alternatives.

These tasks would have students begin with a known working compiler and then they would use techniques learned in class to change the parser, symbol table management, code generation, *etc*. For undergraduate compiler courses usually there is only one semester available and the students often lack experience programming non-trivial projects. There is rarely enough time to start from scratch implementing a compiler, but by using this already-debugged simple compiler as a starting point, many techniques can be explored. While many of the methods used in this compiler are not appropriate for a production compiler, an optimizing production compiler is not appropriate for average students at schools with modest academic standards for their first exposure to compiler technology. By avoiding the use of a compiler kit, scanner generator, or parser generator, the amount of black-box programming is minimized. The skills learned can be used with any iterative, procedural language on any platform.

This compiler does not support optimization at all, and could not be easily modified to support it. The design decision to keep things simple has the disappointing side-effect that using it to teach about optimization is not practical. However, beginners first learning about compilers should have as their primary goal generating a correct, verifiable translation. The assembly output of a fully optimizing compiler that re-arranges code, removes dead code, *etc.* is not trivially easy to match up to the input program. Students who have completed their study of this compiler will be ready to move on to the next step, which would be studying one of the many available FOSS optimizing production-quality compilers, should they wish to continue their study of compiler technology. Consider the case of a Formula 1 driver. Most of them do not begin their career driving a Forumula 1 car in competition. Instead, they build their skills driving simpler, lower performance vehicles. Most recommendations suggest starting very young with kart racing. Yet for compiler study, some people expect new students to start at the highest level, a production, optimizing compiler. This compiler is intended to fill the gap where a student wants to learn about compilers but has no experience with them. Is it better to have students learn 100% of a simple compiler or a tiny percentage of some complex compiler? Ultimately, the answer to that question is a matter of opinion. The author of this compiler and paper posits that a **complete**, usable understanding of a simple system is better than a vague understanding of giant project *when the goal is to learn about compilers*, but freely admits that it is doubtful that this position can ever be definitively proved or disproved. This compiler provides a viable alternative for exposing new students to compiler technology, and is not intended as a replacement for more rigorous instruction for higher-level study, but instead as an additional introductory first step to ease students into the compiler writing discipline a bit more gently than is usually done today.

Since Algol-derived languages tend to be stream-based, one might reasonably ask the question "Why is the capability of processing a line-based language still relevant today?" It is still relevant because many languages which remain in use today are line-based, including Bourne-style shells, nroff, COBOL and REXX. The FORTRAN and python languages are both still actively developed, and they are also line-based languages. The input for the common `make` project build tool, and the assemblers used for low-level programming are line-based. Even the C language preprocessor `cpp` is line-based. As in the case of the x86-64 CPU, if students develop the skills necessary to handle the syntax of line-based languages, then working with the syntax of stream-based languages will be easy for them.

3.4. Future Work

This section details several areas where improvements could be made to this compiler. This includes adding more BASIC language features, improving code generation, and improvements that could be made now that current hardware is available for development and testing. Implementing a Minimal BASIC compiler has been a vital first step in resurrecting *traditional* BASIC as a compiled language for a modern Linux® platform. The term 'traditional BASIC' in this case means a BASIC that requires line numbers for every statement, and not the completely incompatible, numberless, Pascal-like language made popular by Microsoft® with its Visual BASIC® product line that shares little besides the name with the original BASIC language. The feature set of Minimal BASIC is not complete enough for general-purpose programming. Implementing just some of Full BASIC's features would result in a BASIC dialect capable of string and file processing, yielding a BASIC that is suitable for doing most entry-level, general-purpose, text-mode, single-threaded, non-networked programming.

3.4.1. Full BASIC

The most useful logical next step in the development of the compiler is to enhance it to support features from ANSI's "Full BASIC" [70]. The ANSI version of the Minimal BASIC specification, the *ANSI X3.113-1987 "Programming Languages Full BASIC"* specification, despite being revoked, remains non-free. Fortunately the similar ECMA-116 BASIC-1 BASIC standard [71] is freely available.

Updating the compiler to support Full BASIC could be done incrementally by adding different parts of the Full BASIC feature set individually. The first piece to add would be full support for string processing. While this was available in 1966 for Dartmouth BASIC ([17], p. 528), unfortunately it was not included in the Minimal BASIC standard. Other pieces to add include support for files, matrix math operations with **`MAT`**, the new additional built-in functions, and **`DECIMAL`** math. Another area of changes would be the changes to support **`ELSE`**, compound conditionals, **`DO`** loops, **`CASE`** statements, multi-line functions, multi-line subroutines, the **`CHAIN`** feature, **`PRINT USING`**, *etc*. BASIC-2 requires support for a third type of floating point, **`FIXED`**. Since the x86-64 instruction set no longer supports BCD (binary coded decimal) math, implementing the required default **`OPTION ARITHMETIC DECIMAL`** math mode would require updating the runtime library to supported emulated decimal math.

3.4.2. Code Generation Improvements

The Minimal BASIC compiler described in this paper emits unoptimized *large model* code. Since no BASIC program that can fit in the 9999 line limit of Minimal BASIC would actually require 64 bit jumps or calls, a more ideal but also more complex solution would be to use *small model* code.

This compiler avoids the complexity of register allocation by using a dedicated floating point stack at a non-negligible runtime cost. Rewriting the expression evaluation code to use registers directly should result in higher quality code generation and improved performance of generated executables, but would probably also increase compilation time.

The string functions could be improved to check for a maximum length like the standard *`strnlen()`*, *`strncpy()`*, and *`strncmp()`* functions. In addition, no attempt was made to reorder instructions to accommodate the CPU pipeline as suggested in Intel®'s optimization manuals [72]. Actually modifying the compiler to do that kind of instruction scheduling would improve runtime performance, but only at a considerable cost in compiler complexity.

More assembly macros could be created to make the generated code sequences for **`INPUT`** and **`READ`** statements easier to read. It would also be possible to use macros for abstracting away the string error checking in string expressions in a fashion similar to the recent changes made to arithmetic expression processing. While such macro changes would not improve performance or add features, they would definitely make reading the generated code easier for novices, and less tedious for the more adept.

3.4.3. Adding Debugging Support

Another feature that would be nice to have is generation of DWARF format debugging information. This would allow normal use of the GNU *`gdb`* debugger on compiled Minimal BASIC programs. DWARF format debugging data is complex to add to code generation, and there is no specific BASIC language support in the DWARF standard, so adding DWARF support to this compiler would certainly be a challenge.

3.4.4. Adding AVX Support

Even though this compiler does not attempt any vectorization, using AVX instruction support would be beneficial for future development because it includes a compiler-friendly three operand format for SIMD math operations ([73], p. 2) and has relaxed alignment requirements ([74], p. 6) for data access. For a compiler writer this makes implementing register-based expression evaluation much easier compared to x86-64 processors without AVX support. For the Minimal BASIC compiler described in this paper, using the three operand AVX math features will make it much easier to update the compiler to use register-based numeric expression evaluation. The SLEEF library already can use AVX instructions and can provide drop-in support for the necessary elementary functions.

4. Conclusions

4.1. Results

The Minimal BASIC compiler this paper describes is a complete and faithful implementation of a simple, non-optimizing compiler to assembly language for a standard traditional BASIC dialect. Since it is a free and open source implementation, anyone can use parts of it as a base to create their own custom compiler. The compiler is also simple enough that undergraduate students with a meager set of programming skills can study, understand, and modify it. The compiler is intended to be a compiler good for teaching issues often not covered in beginning courses, including code generation for an actual mainstream 64 bit CPU, floating point exception handling, and dealing with line-based languages. By refraining from using scanner and parser generators and complete compiler kits, students can actually learn how the algorithms work instead of relying on black-box tools. Every respectable compiler textbook covers DFSM scanners and recursive-descent parsers, and this compiler uses those well-known, time-tested, proven techniques.

The aim of this project was to create a compiler that conforms 100% to the ECMA-55 Minimal BASIC standard. Correctness was the primary concern, and a secondary objective was simplicity so that even mediocre students can easily understand how it works. The target was x86-64 assembly language in the AT&T dialect used by GNU binutils' `as` assembler, and one requirement was to make the resulting executables stand-alone so they would not depend on any external shared libraries. The time allotted for development was one year. All of the design goals were met in roughly nine months.

The compiler passed the majority of the NBS test suite without problems, and the few failures have been analyzed and it was determined that the remaining cases were test case errors for the ECMA-55 standard (although they may be valid for the paywalled ANSI standard) of Minimal BASIC. All required exception handling for the floating point math was implemented for expression evaluation. The recommended optional uninitialized variable detection was also implemented. The generated static executables run well without any dependency on external libraries like libc or libm. This feature means that compiled Minimal BASIC programs are easy to distribute as a single static executable. The total number of lines used for the compiler and runtime is less than 25 thousand lines, and only a C99 compiler (`gcc`, `clang`/LLVM, and `pcc` all will work), the GNU `as` assembler, and the GNU `ld` linker are actually *required* to build it.

Since it has already been stated that this compiler eschews advanced techniques in order to maintain simplicity, one obvious question to consider is "While this compiler is definitely new, does it really provide anything that can be considered novel?" First I will reiterate the novel teaching and learning approach it is designed to both enable and encourage, which is somewhat subjective. Then I will list the unique technical features which in contrast are verifiable facts.

This compiler enables an innovative instructional approach for introducing people to the world of compilers. Most compiler instruction is designed for advanced students and researchers. Many people have the belief that compilers need to be complex, production quality, optimizing implementations in order to be relevant. That elitist attitude keeps the number of people studying compilers much smaller than it should be. Compiler courses tend to gain a reputation as GPA killers, and in many programs the compiler course is an elective, so the number of people who actually enroll is small. This compiler allows younger, less experienced programmers to safely begin studying compilers earlier in their careers, and succeed doing it. This implementation is much less exclusive than full production quality implementations, since anybody who can learn simple C, basic assembly language, and can use arrays and linked lists has all the necessary skills to work with this code base. Compiler writing can now finally cease to be the exclusive domain of advanced computer scientists. When more people start studying compilers sooner, they will realize that a pragmatic and simple approach allows them to create a working compiler that runs on and targets a modern platform. Students will see that learning how an entire simple compiler works is something a normal programmer can achieve even early in their career. This will allow them to change from fearing compiler technology to realizing that it's just one more area of computer science to consider. At many institutions teachers are faced with the hard choice of either demanding genius if they want a student to understand the entire compiler, or giving up and having the students modify something they have no hope of understanding. Since most students are not geniuses, most programs just teach some small percentage of the entire compiler. The compiler described in this paper is designed for teachers and students who want a complete, working traditional compiler to machine code that they can fully comprehend. This lowers the entry barrier to the study of compiler technology to not only allow more people to get involved, but also to allow more people to actually understand the entire process, from scanning the input to actual code generation, with a concrete implementation. In short, this compiler enables beginners to learn about compilers as just another area of computer science equivalent with operating systems, databases, networking, etc. instead of as an advanced topic only for the chosen few. Of course people intending to pursue compiler research or a commercial career with compilers would have to learn much more, but this compiler provides a solid foundation for the basic concepts that ordinary students at average schools can fully master, in contrast to just studying a small part of some huge, complex compiler.

What technological features does this non-optimizing compiler implementation provide that make it stand out from other alternatives?

(1) This compiler includes a complete runtime library, so the behavior of generated executables remains uniform even if the system libraries are different on different Linux distributions.

(2) The generated executables are statically linked and have no embedded paths, so they can be moved both within a filesystem and between machines and they still work, only requiring a Linux 3.X x86-64 kernel.
(3) This is the only fully-compliant implementation of ECMA-55 Minimal BASIC targeting x86-64 Linux.
(4) This is the the first and only 64 bit version of a fully compliant ECMA-55 Minimal BASIC for any platform.
(5) This compiler generates floating point code with full exception checking, something that `gcc`, `clang`/`LLVM`, `pcc`, and `tcc` cannot even optionally do.

No other FOSS compiler targeting x86-64 Linux® exists that accepts a traditional line-numbered BASIC, uses 64-bit floating point code, and generates assembly. The only commercial BASIC compiler available for that platform, Pure BASIC [22], does not support any traditional line-numbered BASIC dialect. The compiler described in this paper is not merely a trivial enhancement or simple add-on to some existing compiler toolchain. This compiler is actually an all-new, from scratch implementation. This results in a compiler that is small and only includes what is required. In fact, the compiler itself, together with the runtime library which it embeds, can be distributed as a single static executable of only 740KB when linked against musl [75] libc. The peephole optimizer adds another 22KB. A complete binary package with documentation is only 790KB.

4.2. Performance of Generated Executables

The compiler is already fast enough to compile hundreds of programs in less than a minute even on obsolete hardware. But what about the generated executables? Runtime performance of the generated executable programs is fast enough for the kinds of non-production work that can be done in ECMA-55 Minimal BASIC. To prove this, the `DUMBSINE.BAS` program was created that does hundreds of thousands of calculations of the `sin()` function using a naive Taylor series expansion and then compares the results to the built-in **`SIN()`** function. On an Intel® Core™ 2 Duo CPU (E4700@2.60 GHz) the compiled ECMA-55 Minimal BASIC program runs in about 44.5 seconds. A *roughly* equivalent C99 program `dumbsine.c` takes about 19.0 seconds when compiled without optimization. The C99 program attempts to do some exception checking so that the comparison is at least approximately fair. The ECMA-55 Minimal BASIC program is about 42.6% of the speed of the C99 version. The ECMA-55 Minimal BASIC compiler makes no attempt at any instruction scheduling, and that, combined with the stack-based expression evaluation, is more than likely responsible for much of the difference in runtime even when comparing against unoptimized output of a C99 compiler. The code for both programs was meant to exercise floating point math functions, not to be a serious attempt at calculating values for the `sin()` function. The source code for the programs discussed here is included in the appendix. The standard Whetstone [76] benchmark for floating point performance could not be ported since it requires integer support which is not available in ECMA-55 Minimal BASIC.

4.3. Availability

The compiler and included runtime are open source, and both read-only Mercurial version control repository access and standard `tar` files compressed with `xz` are freely available and can be downloaded from the following SourceForge project URL: http://sourceforge.net/projects/buraphakit/.

The compiler itself is licensed under the GNU General Public License Version 2 [77] only, and the third party runtime modules have a variety of free and open source licenses noted in the accompanying documentation.

Acknowledgments

The author would like to thank Mr. Emmanuel Roche for providing copies of several of the NBS Minimal BASIC test suite programs.

Author Contributions

John Gatewood Ham wrote this entire document himself.

A. Benchmark Programs

Reviewers of a draft of this paper requested some performance information. Since no other ECMA-55 Minimal BASIC compiler exists for the x86-64 Linux® platform, one program was written in ECMA-55 Minimal BASIC and then another corresponding program was written in C99. The programs are not quite equivalent since by default C99 has no floating point exception checking and that had to be added, but it was not practically feasible to exactly match the semantics of the original ECMA-55 Minimal BASIC program.

A.1. ECMA-55 Minimal BASIC **`DUMBSINE.BAS`** *Program Source*

This program computes values for the sine function using a Taylor series expansion and compares the results to the built-in **`SIN()`** function. This is done repeatedly for various angles in order to benchmark the performance of the code generated by the compiler.

```
10 REM THIS IS A PROGRAM DESIGNED TO BURN A LOT OF CPU
20 REM TO GIVE A ROUGH MEASURE OF RUNTIME PERFORMANCE.
30 REM Z IS ARRAY TO HOLD RESULTS FOR SINE(0.0001) THROUGH SINE(PI/8)
40 REM IN STEPS OF 0.001 FOR 1000 VALUES, PI=3.141592653589793
50 DIM Z(4000)
60 DEF FNP=3.141592653989793
65 FOR K=1 TO 1000
70 FOR X=0.0001 TO FNP/8 STEP .0001
80 LET A=X
90 GOSUB 1000
100 LET Z(1000*X)=R
110 NEXT X
120 REM PRINT RESULTS
130 FOR X=0.0001 TO FNP/8 STEP .0001
140 PRINT "MY SIN(";X;") IS";Z(1000*X);"ACTUAL IS";SIN(X);
141 PRINT "DIFF";ABS(SIN(X)-Z(1000*X))
150 NEXT X
155 NEXT K
160 STOP
1000 REM THIS IS A SINE SUBROUTINE.
1010 REM THE ANGLE IN RADIANS SHOULD BE IN GLOBAL VARIABLE
1020 REM A, AND THE SIN(A) WILL BE RETURNED IN GLOBAL VARIABLE
1030 REM R.  A TAYLOR SERIES EXPANSION WITH 5 TERMS IS USED.
1040 LET R=0
1045 REM S IS SIGN FLAG, AND 0 MEANS SUBTRACT, 1 MEANS ADD
1050 LET S=0
1060 FOR I=1 TO 5 STEP 1
```

```
1070 REM T IS ONE TERM
1080 LET T=1
1090 REM N IS THE NUMERATOR A^I
1100 LET N=1
1110 FOR J=1 TO I*2-1 STEP 1
1120 LET N=N*A
1130 NEXT J
1140 REM D IS THE DENOMINATOR I!
1150 LET D=1
1160 FOR J=1 TO I*2-1 STEP 1
1170 LET D=D*J
1180 NEXT J
1190 LET T=N/D
1200 IF S=1 THEN 1240
1210 LET R=R+T
1220 LET S=1
1230 GOTO 1260
1240 LET R=R-T
1250 LET S=0
1260 NEXT I
1270 RETURN
1300 END
```

A.2. C99 `dumbsine.c` Program Source

This is an attempt to port the **`DUMBSINE.BAS`** program to C99. It is roughly equivalent except that it will of course use proper register-based arithmetic expression evaluation.

```
/*
 gcc -Wall -Wextra -pedantic -std=c99 -D_GNU_SOURCE -O0 \
   -m64 -march=core2 -mtune=core2 -frounding-math -mno-recip \
   -mcmodel=large -mno-recip -fno-associative-math -fmerge-constants \
   -fno-inline -fno-builtin dumbsine.c -o dumbsine -lm -static
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fenv.h>

static double Z[4000], A, D, I, J, K, N, R, S, T, X;
static int set_excepts;

static double FNP(void) { return 3.1415926535989793; }

static void mysin(void) {
  double temp;

  R=0.0; S=0.0; I=1.0;
  while (1) {
    if (I>5.0) break;
    set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
    if (set_excepts & FE_INVALID) goto fail;
    if (set_excepts & FE_OVERFLOW) {
      fprintf(stderr,"FE_OVERFLOW exception\n");
      I=INFINITY;
    }
    T=1.0; N=1.0; J=1.0;
    while (1) {
      temp=I*2.0;
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      temp=temp-1.0;
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      if (J>temp) break;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        J=INFINITY;
      }
      if (set_excepts & FE_UNDERFLOW) {
        fprintf(stderr,"FE_UNDERFLOW exception\n");
        J=0.0;
      }
      N=N*A;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        N=(A<0?-INFINITY:INFINITY);
      }
      if (set_excepts & FE_UNDERFLOW) {
        fprintf(stderr,"FE_UNDERFLOW exception\n");
        N=0.0;
      }
      temp=J+1.0;
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      J=temp;
    }
    D=1.0; J=1.0;
    while (1) {
      temp=I*2.0;
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      temp=temp-1.0;
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      if (J>temp) break;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        J=INFINITY;
      }
      if (set_excepts & FE_UNDERFLOW) {
        fprintf(stderr,"FE_UNDERFLOW exception\n");
        J=0.0;
      }
      D=D*J;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        D=(J<0?-INFINITY:INFINITY);
      }
      if (set_excepts & FE_UNDERFLOW) {
        fprintf(stderr,"FE_UNDERFLOW exception\n");
        D=0.0;
      }
      temp=J+1.0;
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      J=temp;
    }
    T=N/D;
    set_excepts = fetestexcept(FE_INVALID | FE_UNDERFLOW | FE_DIVBYZERO);
    if (set_excepts & FE_INVALID) goto fail;
    if (set_excepts & FE_UNDERFLOW) {
      fprintf(stderr,"FE_UNDERFLOW exception\n");
      T=0.0;
    }
    if (set_excepts & FE_DIVBYZERO) {
      fprintf(stderr,"FE_DIVBYZERO exception\n");
      goto fail2;
    }
    if (S==0) {
      R=R+T;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        R=INFINITY;
      }
      S=1.0;
    } else {
      R=R-T;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        R=-INFINITY;
      }
      S=0.0;
    }
    temp=I+1.0;
    if (set_excepts & FE_INVALID) goto fail;
    if (set_excepts & FE_OVERFLOW) {
      fprintf(stderr,"FE_OVERFLOW exception\n");
      temp=INFINITY;
    }
    I=temp;
  }
  return;
fail2:
  fprintf(stderr,"FE_DIVBYZERO exception\n");
  goto gameover;
fail:
  fprintf(stderr,"FE_INVALID exception\n");
gameover:
  abort();
}

int main(void) {
  double temp;

//#pragma STDC FENV_ACCESS ON     // No support for this on Fedora 20

  // initialize global data
  for (X=0;X<4000;X++) Z[(unsigned int)round(X)]=NAN;
  A = NAN; D = NAN; I = NAN; J = NAN; K = NAN;
  N = NAN; R = NAN; S = NAN; T = NAN; X = NAN;
  set_excepts = 0;

  feclearexcept(FE_INVALID|FE_OVERFLOW);

  K=1.0;
  while (1) {
    if (K>1000.0) break;
    X=0.0001;
    while (1) {
      temp=FNP();
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      temp=temp/8.0;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      if (X>FNP()/8.0) break;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        X=INFINITY;
      }
      A=X;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        A=(X<0?-INFINITY:INFINITY);
      }
      mysin();
      Z[(unsigned int)round(1000.0*X)]=R;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        Z[(unsigned int)round(1000.0*X)]=(R<0?-INFINITY:INFINITY);
      }
      temp=X+0.0001;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      X=temp;
    }
    X=0.0001;
    while (1) {
      temp=FNP();
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      temp=temp/8.0;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      if (X>temp) break;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        X=INFINITY;
      }
      printf("MY SIN( %17.15lg ) IS %17.15lg ACTUAL IS %17.15lg",
        X,Z[(unsigned int)round(1000.0*X)],sin(X));
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        abort();
      }
      printf(" DIFF %17.15lg\n",fabs(sin(X)-Z[(unsigned int)round(1000.0*X)]));
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        abort();
      }
      temp=X+0.0001;
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) goto fail;
      if (set_excepts & FE_OVERFLOW) {
        fprintf(stderr,"FE_OVERFLOW exception\n");
        temp=INFINITY;
      }
      X=temp;
    }
    temp=K+1.0;
    set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
    if (set_excepts & FE_INVALID) goto fail;
    if (set_excepts & FE_OVERFLOW) {
      fprintf(stderr,"FE_OVERFLOW exception\n");
      temp=INFINITY;
    }
    K=temp;
  }
  return EXIT_SUCCESS;
fail:
  fprintf(stderr,"FE_INVALID exception\n");
  return EXIT_FAILURE;
}
```

Conflicts of Interest

The author declares no conflict of interest.

References

1. Torvalds, L. (Maintainer). The Linux® Kernel. Available online: http://www.kernel.org/ (accessed on 24 July 2014).

2. BASIC A Manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System, 1964.
3. Kemeny, J.G.; Kurtz, T.E.; Cochran, D. *BASIC A Manual for BASIC, the Elementary Algebraic Language Designed for Use with the Dartmouth Time Sharing System*, 4th ed.; Dartmouth College Compoutation Center: Hanover, NH, USA, 1968.
4. Kurtz, T.E. BASIC. *ACM SIGPLAN Not.* **1978**, *13*(8), 103–118.
5. Isaacs, G.L. *Interdialect Translatability of the BASIC Programming Language*; Research and Development Division, American College Testing Program, 1972.
6. Lientz, B.P. A Comparative Evaluation of Versions of BASIC. *Commun. ACM* **1976**, *19*, 175–181.
7. Luehrmann, A.W.; Garland, S.J. Graphics in the BASIC language. *ACM SIGGRAPH Comput. Graph.* **1974**, *8*, 1–8.
8. Anderson, R.E. ANSI BASIC The proposed standard. In Proceedings of the ACM'82 Conference; Association for Computing Machinery: New York, NY, USA, 1982; p. 213.
9. Kurtz, T.E.; Garland, S.J. Toward standardization of BASIC. *ACM SIGCUE Outlook* **1971**, *5*, 221–222.
10. Dijkstra, E.W. How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective*; Springer-Verlag: New York, USA, 1982; pp. 129–131.
11. Dijkstra, E.W. The threats to computing science. circulated privately.
12. Brin, D. Why Johnny Can't Code. Salon. Sep. 14, 2006. Available online: http://www.salon.com/2006/09/14/basic_2/ (accessed on 24 July 2014).
13. Raji, V. Microsoft™ Small Basic. Available online: http://smallbasic.com/ (acessed on 24 July 2014).
14. Larsen, I. BASIC-256. Available online: http://www.basic256.org/ (accessed on 24 July 2014).
15. Cugini, J.V.; Bowden, J.S.; Skall, M.W. *NBS Minimal BASIC Test Programs—Version 2, User's Manual, Volume 1—Documentation*; Government Printing: Washington, DC, USA, 1980.
16. Cugini, J.V.; Bowden, J.S.; Skall, M.W. *NBS Minimal BASIC Test Programs—Version 2, User's Manual, Volume 2—Source Listings and Sample Output*; Government Printing: Washington, DC, USA, 1980.
17. Kurtz, T.E. Basic. In *History of Programming Languages*; ACM: New York, NY, USA, 1981; pp. 515–537.
18. Liguori, A. GLBCC, The GNU/Liberty Compiler Collection. Available online: http://lbpp.sourceforge.net/ (accessed on 24 July 2014).
19. Victor, A. FreeBASIC Programming Language. Available online: http://www.freebasic.net/ (accessed on 24 July 2014).
20. Kemeny, J.G.; Kurtz, T.E. TrueBASIC. Available online: http://www.truebasic.com/faq#11 (accessed on 24 July 2014).
21. Gundel, C. Liberty BASIC. Available online: http://www.libertybasic.com/faq.html#macOrLinux (accessed on 24 July 2014).
22. Laboureur, F.; Harter, T. PureBasic. Available online: http://www.purebasic.com/ (accessed on 24 July 2014).

23. Intel® Extended Memory 64 Technology Software Developer's Guide, Revision 1.1, 2004.
24. Intel®64 and IA-32 Architectures Software Developer's Manual, 2014. Order Number: 325462-050U.
25. AMD64 Architecture Programmer's Manual Volume 1: Application Programming, 2012; Publication No. 24592, Revision 3.19.
26. GCC, The GNU Compiler Collection. Available online: http://gcc.gnu.org/ (accessed on 24 July 2014).
27. The LLVM Compiler Infrastructure. Available online: http://llvm.org/ (accessed on 24 July 2014).
28. Free Pascal. Available online: http://www.freepascal.org/ (accessed on 24 July 2014).
29. Preud'homme, T. (Maintainer). The Tiny C Compiler. Available online: http://bellard.org/tcc/ (accessed on 24 July 2014).
30. Portable C Compiler. Available online: http://pcc.ludd.ltu.se/ (accessed on 24 July 2014).
31. STANDARD ECMA-55 Minimal BASIC; Eurpean Computer Manufacturers Association: Geneva, Switzerland, 1978.
32. Anvin, H.P. (Maintainer). Netwide Assembler. Available online: http://nasm.us/ (accessed on 24 July 2014).
33. Clifton, N. (Head Maintainer). GNU Binutils. Available online: http://www.gnu.org/software/binutils/ (accessed on 24 July 2014).
34. Matz, M.; Hubička, J.; Jaeger, A.; Mitchell, M. *System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.6*, 2013.
35. Turbo Debugger® Version 2.0 User's Guide, 1990.
36. Alves, P.; Brobecker, J.; Evans, D.; Kratochvil, J.; Tromey, T.; Zaretskii, E. (Maintainers). GDB: The GNU Project Debugger. Available online: http://www.gnu.org/software/gdb/ (accessed on 24 July 2014).
37. Saha, S. (Maintainer). Data Display Debugger. Available online: http://www.gnu.org/software/ddd/ (accessed on 24 July 2014).
38. DWARF Debugging Standard. Available online: http://www.dwarfstd.org/ (accessed on 24 July 2014).
39. Teran, E. edb debugger. Available online: https://code.google.com/p/edb-debugger/ (accessed on 24 July 2014).
40. Hauser, J.R. Handling Floating-point Exceptions in Numeric Programs. *ACM Trans. Program. Lang. Syst.* **1996**, *18*, 139–174.
41. Seyfarth, R. *Introduction to 64 Bit Intel Assembly Language Programming for Linux*, 2nd ed.; CreateSpace Independent Publishing Platform: North Charleston, SC, USA, 2012.
42. IEEE Standard for Floating-Point Arithmetic. *Technical Report, Microprocessor Standards Committee of the IEEE Computer Society*; IEEE Computer Society: New York, NY, USA, 1985.
43. List of Intel Core 2 microprocessors. Available online: http://en.wikipedia.org/wiki/List_of_Intel_Core_2_microprocessors (accessed on 24 July 2014).
44. Intel® SSE4 Programming Reference, 2007. Reference Number: D91561-001.
45. Denman, H.H. Minimax Polynomial Approximation. *Math. Comput.* **1966**, *20*, 257–265.

46. Hart, J.F.; Cheney, E.W.; Lawson, C.L.; Maehly, H.J.; Mesztenyi, C.K.; Rice, J.R.; Thacher, H.G., Jr.; Witzgall, C. *Computer Approximations*; SIAM Series in Applied Mathematics, R. E.; Krieger Pub. Co.: Malabar, FL, USA, 1978.
47. Green, R. Faster Math Functions. In Proceedings of Game Developers Conference; UBM Tech: New York, NY, USA, 2003.
48. Goldberg, D. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.* **1991**, *23*, 5–48.
49. Brent, R.P. Fast algorithms for high-precision computation of elementary functions. In Proceedings of 7th Conference on Real Numbers and Computers (RNC 7), 10-12 July 2006, Nancy, France; pp. 7–8.
50. Freely Distributable LIBM C math library verison 5.3. Available online: http://www.validlab.com/software/ (accessed on 24 July 2014).
51. Shibata, N. Efficient evaluation methods of elementary functions suitable for SIMD computation. *Comput. Sci.-Res. Dev.* **2010**, *25*, 25–32.
52. Shibata, N. SLEEF version 2.80. Available online: http://shibatch.sourceforge.net/ (accessed on 24 July 2014).
53. Robert J. Jenkins, J. ISAAC. In Proceedings of the Third International Workshop on Fast Software Encryption; Springer-Verlag: London, UK, 1996; pp. 41–49.
54. Robert J. Jenkins, J. ISAAC: A fast cryptographic random number generator. Available online: http://burtleburtle.net/bob/rand/isaacafa.html (accessed on 24 July 2014).
55. Arnold, R.S.; Brown, M.; Eggert, P.; Jelinek, J.; Kuvyrkov, M.; McGrath, R.; Myers, J.; O'Donell, C.; Oliva, A.; Schwab, A. (Maintainers). The GNU C Library. Available online: http://www.gnu.org/software/libc/ (accessed on 24 July 2014).
56. Clinger, W.D. How to Read Floating Point Numbers Accurately. *SIGPLAN Not.* **1990**, *25*, 92–101.
57. Guy L. Steele, J.; White, J.L. How to Print Floating-point Numbers Accurately. In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation; Association of Computer Machinery: New York, NY, USA, 1990; pp. 112–126.
58. Burger, R.G.; Dybvig, K.R. Printing Floating-point Numbers Quickly and Accurately. In Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation; Association of Computer Machinery: New York, NY, USA, 1996; pp. 108–116.
59. Gay, D.M. Correctly rounded binary-decimal and decimal-binary conversions. *Numerical Analysis Manuscript* 90-10, 1990.
60. Gay, D.M. `dtoa.c` & `g_fmt.c`. Available online: http://netlib.org/fp/ (accessed on 24 July 2014).
61. Cowlishaw, M.F. The Design of the REXX Language. *SIGPLAN Not.* **1987**, *22*, 26–35.
62. Henry S. Warren, J. *Hacker's Delight, Second Edition*; Addison-Wesley/Pearson Education, Inc.: Courier in Westford, MA, USA, 2013.
63. McKeeman, W.M. Peephole Optimization. *Commun. ACM* **1965**, *8*, 443–444.
64. Valgrind's Memcheck. Available online: http://www.valgrind.org/ (accessed on 24 July 2014).

65. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. In Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12); USENIX Association: Berkeley, CA, USA, 2012.
66. ISO/IEC G14/N1124 Comittee Draft of ISO/IEC 9899:TC2, 2005.
67. Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual, 2012. Reference Number: 327364-001.
68. Fraser, C.W.; Hanson, D.R. *A Retargetable C Compiler: Design and Implementation*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1995.
69. Aho, A.V.; Sethi, R.; Ullman, J.D. *Compilers Principles, Techniques, and Tools*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1986.
70. Guntheroth, K. The New ANSI BASIC Standard. *SIGPLAN Not.* **1983**, *18*, 50–59.
71. STANDARD ECMA-116 BASIC. *Eurpean Computer Manufacturers Association*: Geneva, Switzerland, 1986.
72. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2014. Order Number: 248966-029.
73. Intel® Avanced Vector Extensions Programming Reference, 2011. Reference Number: 319433-011.
74. Firasta, N.; Buxton, M.; Jinbo, P.; Nasri, K.; Kuo, S. Intel® AVX: New Frontiers in Performance Improvements and Energy Efficiency, 2008 (White Paper).
75. Felker, R. (Maintainer). The musl C standard library. Available online: http://www.musl-libc.org/ (accessed on 24 July 2014).
76. Curnow, H.J.; Wichmann, B.A. A synthetic benchmark. *Comput. J.* **1976**, *19*, 43–49.
77. GNU General Public License, 1991. Available online: http://www.gnu.org/licenses/gpl-2.0.html (accessed on 24 July 2014).