Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Decimal Floating Point Extension for C via decNumber

Jon Grimm
IBM Corporation
jgrimm@us.ibm.com

Abstract

Typical computer systems perform calculations using binary floating point arithmetic. While binary floating-point has been relatively simple and efficient to implement in hardware, most humans are accustomed to performing arithmetic in the decimal system. Unfortunately, conversions (from decimal) to binary float and back, as well as, performing various arithmetic operations on binary floating-point numbers can introduce non-obvious errors and/or unacceptable behavior for some applications. For example, commercial and financial software often demand accurate decimal behaviors for calculations involving monetary values, such as billing or currency conversion.

Recently, the IEEE 754r revision has undertaken efforts to standardize decimal floating-point encodings and arithemetic operations. Consequently, the ISO JTC/SC22/WG14 has started developing (currently Draft 5) "Extensions for the programming language C to support decimal floating point arithmetic."

To examine the emerging C extension and decimal floating point arithmetic (and in absence of real hardware), the decNumber package from Mike Cowlishaw has been integrated into GCC to provide decimal arithmetic operations.

1 Introduction

If one were to multiply the values of 0.70 by 1.05, the typical expected result would be 0.735 and then rounded up to a result of 0.74. However, this calculation as coded in the C program in Table 1, and executed on my test system[1], results in the following:

```
$ ./fp
result = 0.73
```

What is going on here? Instead of the expected[2] result of *0.74*, the program calculates a value of *0.73*.

```
#include <stdio.h>
int main (void)
{
  float x=0.70f,y=1.05f;
  printf ("result = %.2f\n",
          x*y);
  return 0;
}
```

Table 1: Simple C program

Well, contemporary floating-point units implement binary floating-point, which do not nec-

[1] An iBook G4 running Linux

[2] Well, *expected* is not quite accurate, as the program is conforming to the C language standards, but the results are possibly non-obvious to one accustomed to decimal calculations.

essarily convert perfectly to and from the decimal system we are normally accustomed to in day-to-day arithmetic, such as calculating interest on a savings account. In the above case, the calculation results in a value of 0.73, as the binary encoding of the result was slightly *less* that 0.735, with the default rounding-mode then rounding the value to 0.73.

Binary floating-point behavior is inadequate for a variety of applications, and consequently, various software packages have had to implement (or link to) their own support code to handle decimal arithmetic correctly. However, this could increasingly become an issue for accounting and financial applications. A mundane, but relevant example, would be the added volume of calculations required by a per second billing system at a large telecommunications provider. Some commercial workloads have been identified that spend as much as 50% of their time processing decimal data in software [Cowl1].

Accordingly, there is renewed interest in decimal floating-point as a solution for such market needs. There is work in the IEEE floating point standards group to define the encoding and arithmetic for decimal floating-point. Research suggests that hardware implementations should be 1 to 2 orders of magnitude faster than software implmentations [Sch1]. Additionally, there are C and C++ extensions in draft to support the IEEE standards effort.

1.1 Floating Point Standardization

There are obviously many different ways that real numbers can be represented and acted upon by computers. Floating-point representation is a common method to represent numbers, where the number can be represented by a base number multiplied by an exponent. The term *floating* indicates that the scale is not *fixed*, where the scale can be sufficiently represented by the exponent component of the number. While floating-point was a well understood concept, until IEEE 754, there were disturbingly different behaviors seen between different floating-point implementations that made writing software quite problematic and error-prone (and interoperability near impossible)[3].

1.1.1 IEEE 754

IEEE 754, provided a standardized model, encoding, and arithmetic operations for floating-point numbers. The model includes three basic components, a sign, a fraction (or mantissa), and an exponent (radix 2). Additionally, the standard specifies several special values, including infinity and NaN, or Not a Number, obviously useful as the outcome of various arithmetic computations. NaNs, come in two flavors, signalling NaNs, which throw a processor exception, or quiet NaNs. Signed zeros and signed infinities are also representable in IEEE 754.

IEEE 754 describes both basic and extended versions of single-precision and double-precision formats, however, the extended formats are loosely defined so that implmentations using them may be "so different that numerical approximation routines using them could be non-portable." [Hol1] The basic single-precision and double-precision formats require 32 and 64 bits, respectively, for their encodings.

Besides the bit encodings, IEEE 754 also mandates the behaviors of various operations, including the expected math operations of add, subtract, multiply, and divide, but also described several *rounding modes* that can control the rounding behavior, such as rounding towards $+\infty$ or rounding towards $-\infty$.

[3] See [Sev1] for a quite interesting recollection of this standards effort elicited from Professor Kahan.

However, rather than focusing on binary floating-point, let's investigate the details of the decimal floating-point standards proposal.

1.1.2 IEEE 754R and Decimal Floating-Point

Certainly, binary floating-point is sufficient for a variety of applications, including those in the the scientific domain. However, as discussed earlier, commericial and financial applications need behavior much more like a hand-calculator, as opposed to binary floating-point hardware in commericially available computers today.

Currently, the IEEE 754 standard is in the process of being revised. One of the revisions includes standardizing ***decimal*** floating-point [IEEE1].

The concepts noted for IEEE binary floating-point are now extended to decimal floating-point, including signalling and quiet NaNs, infinities, and signed zeros. The components of the number are the familiar basic components of sign bit, significand, and exponent (not suprisingly, radix 10). A decimal floating-point number is expressed as
$(-1)^{sign} * coefficient * 10^{exponent}$.

Currently, three decimal floating-point formats are proposed: decimal32, decimal64, and decimal128, where the numerical suffix to the format name indicates the number of bits needed in the format. For example, the decimal64 format requires 64-bits, which are broken down into the fields [Cowl2] seen in Table 2 as represented in network byte-order.

The *sign* field is a single bit that indicates a negative value.

The 5-bit *combination* field serves a dual purpose. When all of the first four bits are 1, this indicates a non-finite value, such as infinity or NaN. Otherwise, the field represents a combination of two most signficant bits of the exponent and the most signficant *digit* of the coefficient (or significand).

The size of the exponent continuation and coefficient continuation fields depends on the the representation and for decimal64 case are respectively 8 and 50 bits wide.

The *exponent continuation* field contains the least significant bits of the base 10 exponent encoded as an unsigned binary integer. The 2 bits from the combo field may only take on values 0-2 and are prepended to the exponent continuation field. Additionally, a bias (e.g. 398 for a decimal64) is subtracted from the decoded value, to enable a range of both positive and negative exponents. The calculation of the bias and this range of exponent values is dependent on the maximum value of exponent encodable for a given format [Cowl2].

The *coefficient continuation field* contains the remaining bits of the significand that were partially encoded in the combo field. An added (interesting) complication is that the coefficient is neither a binary integer value, nor a binary coded decimal, but instead a new format referred to as a Densely Packed Decimal, or DPD [Cowl4].

With DPD, each 10 bits of a DPD represents 3 compressed decimal digits. The coefficient continuation is appended to the most significant digit provided by the combination field (this MSD may only hold values 0-9). Consequently, a decimal64 number with a 50-bit continuation field, can hold

$$50/10 * 3 + 1 = 16$$

significant digits.

See Table 3 for a summary of the various limits and parameters for the three new formats.

Length(bits)	1	5	8	50
Content	Sign	Combination	Exponent continuation	Coefficient continuation

Table 2: Fields of decimal64

Format	**decimal32**	**decimal64**	**decimal128**
Coefficient length in digits	7	16	32
Maximum Exponent (E_{max})	96	384	6144
Minimum Exponent (E_{min})	-95	-383	-6143

Table 3: Decimal Floating-Point Parameter Summary

One interesting characteristic of the decimal floating-point is that trailing fractional zeros are capable of being encoded, whereas IEEE 754 binary floating-point cannot record such information. On a loosely related subject, decimal floating-point values are not normalized, as again, this destroys potentially relevant information. Additionally, the avoidance of normalization can provide performance and design simplicity benefits [Cowl3].

Besides the rounding-modes specified by IEEE 754, the revision specifies an additional rounding mode for decimal-floating, *Round to Nearest, Ties Away from Zero*, where the value will round to the nearest value, but where equally near, will round towards the larger magnitude (or away from zero).

Otherwise, IEEE 754R specifies all the expected math operations, such as, add, subtract, multiply, divide, remainder, compare, and negate. Additionally, this proposal specifies the various conversions to and from the various formats, including between the binary and decimal formats.

1.2 C Language Extension

Just as the C language supports float and double types to represent floating-point values, and supports the variety of operations through either the language or through C library functionality, the ISO JTC/SC22/WG14 has started efforts to define the types and operations needed to support decimal floating-point to coincide with the IEEE 754R proposal.

Since the C language predated the IEEE 754 standardization, the C language supports floating-point through a very abstract model, where the lower layer of the model is implementation defined, including the size of data types and their actual encoding. Interestingly, even the radix is implementation specific, though obviously binary floating-point drives most C implementations to radix two implementations. Consequently, creating portable C floating-point software can be difficult. However, the C99 standard now provides support for IEC 559 [C99], where IEC 559 is the ISO standard that adopted IEEE 754 as the binary floating-point standard. In C99, an optional `__STDC_IEC_559__` can indicate an IEC 559 compliant system, so some level of portability seems possible between the prevalent IEEE 754 systems.

Since decimal floating-point formats and operations are being tightly specified to avoid fragmentation and interoperability problems, it is also desirable to more precisely specify the C language bindings for decimal floating-point, as opposed to the abstract model originally

specified for the C language. The extension makes a distinction between generic real types, and decimal real types, where decimal real types are a subclass of the generic real type.

A brief over view of the C language extension proposal follows.

The *C Extension for Decimal Floating-Point* [CExt1] defines three new types to exactly correspond to the decimal floating-point formats as described in IEEE 754R:

- `_Decimal32`
- `_Decimal64`
- `_Decimal128`

Note, however, that the proposed extension (intentionally) does not specify decimal complex and decimal imaginary types.

New constant suffixes of *df*, *dd*, and *dl*, indicate the respective types of `_Decimal32`, `_Decimal64`, and `_Decimal128`, however, such constants may not be used in a hexadecimal-floating-constant.

A new header `<decfloat.h>` defines various macros that specify the decimal floating-point limits and parameters.

Various conversions are specified by the extension, including usual arithmetic conversions and defining the rounding behavior when arithmetic, or in general, conversion cannot be precisely represented. The rounding-modes are similar (though the rounding-mode macro names slightly differ) to those found with C99 IEC 559 support and controlled via **fsetround** and **fgetround** routines. The `FE_DEC_TONEARESTFROM_ZERO` macro represents the IEEE 754 *Round to Nearest, Ties Away from Zero* mode.

The C language operators seem specified to obvious behavior. Additionally, the extension calls out a many math library functions and classification macros that will need to account for decimal floating-point. Of interest, is that most of the functions are allowed to have implementation dependent accuracy, except for **sqrt**, **max**, and **min**. There are two new functions, **quantize**, and **samequantum** which correspond to functionality defined in 754R convert or compare exponents for decimal float values.

Besides the math functions, the C library input/output routines, such as **printf** and **scanf** will need to be updated to handle the new types.

Over time, other languages are expected to incorporate bindings to IEEE 754R decimal floating-point. For example, work has commenced in the the C++ Working Group to investigate how to address DFP, but drafts are not available for viewing as of this writing.

1.3 decNumber Library

The **decNumber** library, authored by Mike Cowlishaw, is a software implementation of the General Decimal Arithmetic Specification [Cowl6] and with a few exceptions implements the decimal floating point encodings and arithmetic of IEEE 754R [Cowl5]. This library provides decimal floating-point support up to a billion digits of precision and 9 digits of exponent. Obviously, this is a bit more than required by the base decimal floating-point types required by IEEE 754R.

The main abstractions provided by decNumber are the decContext and decNumber abstractions. The `decNumber` abstraction represents a generic decimal floating-point value, with its own internal representation of a decimal floating-point value that is efficient (at least

for software) operations. The `decContext` class represents a control abstraction, controlling aspects such as rounding mode, precision, and exception behavior to be applied for a given function invocation.

Besides the `decNumber` abstraction, the library also provides classes that directly correspond to the base decimal floating-point formats, realized as modules which implement `decimal32`, `decimal64`, and `decimal128` formats. These formats are useful for storage and interoperability, but must be converted into decNumber format for the wide variety of operations that one wishes to execute. So for one to add a decimal32 with a decimal128 encoded value one would need to first convert each to a decNumber, as seen in Table 4.

The library itself is split into various *modules* which map to C source files representing the core abstractions. For example, `decNumber.c` and `decNumber.h` represent the decNumber abstraction. Consequently, navigating the source code is very easy.

The more interesting interfaces live in `decNumber.h`. A very important macro in this file, is that of `DECNUMDIGITS`, which eventually is responsible for defining the storage size used by the decNumber internal representation. One may manually set this hook, or a default value will be set, for example, when including the concrete representation header files (e.g. `decimal128.h`), but take note that the order of including these header files is important to make this work right.

The are a large number of public interfaces declared in decNumber.h, each prefixed by a *decNumber*. A quick survey of some of the functions in this file include, **Abs**, **Add**, **Compare**, **Divide**, **Max**, **Min**, **Normalize**, **Quantize**, **SquareRoot**, **Subtract**, **ToIntegral**, **Trim**, and **Zero**.

Overall, the decNumber library contains a very robust set of operations with which to manipulate decimal floating-point numbers. Written in portable ANSI C, the library provides strong foundation for implementing a software based implementation of decimal floating-point support within GCC.

2 GCC

After that whirlwind overview of various decimal floating-point it is finally time to look at GCC.

To analyze and investigate issues involved with decimal floating-point support, we[4] decided to use Mike Cowlishaw's decNumber library as a foundation for implementing the DFP C extension in GCC. There are several several long term issues with actual integration into the GCC source code. Some issues are easily addressed, such as coding style. Others, are intentionally ignored, such as GCC specific memory management, for the purposes of purely exploring DFP and the proposed C extension. Certainly, these issues will need addressed for long term maintenance, but this situation can be remedied as time and priorities dictate.

A primary goal is to develop an initial implementation with which to root out issues with the draft C extension, as well as, discover any changes needed within the GCC infrastructure to support said extension. This GCC work can then be supplemented with changes in related tools, such as the debugger and C library, and we also plan to modify GDB and GLIBC to handle the proposed C extension.

This effort to develop the decimal floating-point extension is being done in the *dfp-branch*

[4] Currently, Ben Elliston, Dwayne McConnell, and myself

```
decimal128 add32to128(decimal32 d32,
                      decimal128 d128)
{
  decContext set;
  decNumber a, b;

  decContextDefault(&set,
                    DEC_INIT_DECIMAL128);
  set.traps = 0;
  decimal32ToNumber(d32, &a);
  decimal128ToNumber(d128, &b);

  /* Result into a. */
  decNumberAdd(&a, &a, &b);

  /* Convert back into decimal128.*/
  decimal128FromNumber(&d128, &a,
                       &set);
  return d128;
}
```

Table 4: Simple decNumber code

off of GCC's CVS and the branch is routinely merged with changes from CVS HEAD.

One of the added benefits to implementing decimal floating-point, is that the activity really provides a broad survey of the GCC landscape. The activity stretches from the C front-end all the way to the backend emulation of decimal floating-point operations, along the way, interacting with the various internal machinery of GCC.

2.1 The C Front-end

The changes needed in the C front-end are not at all invasive to implement.

The `libcpp` library was changed to identify the new constant suffixes of *df*, *dd*, and *dl*, and appropriately classify values as having decimal floating-point characterstics. A new `CPP_N_DFLOAT` classifies the number as a decimal float, while the already existing `CPP_N_SMALL`, `CPP_N_MEDIUM`, and `CPP_N_LARGE` respectively flag the 4-byte, 8-byte, and 16-byte sized decimal floating types. Accordingly, the C front-end uses this information to assign an appropriate type to the numeric input. This is also where hexidecimal representation for decimal float constants are rejected.

Obviously, the 3 new decimal float type names need to be understood by the front-end. Luckily, the C front-end makes this quite simple. The c-parser contains a structure of reserved names and various characterstics about the name. The new type names are now added into this structure (and currently marked as a GCC extension feature). The C front-end machinery uses this structure to parse out the types and correlate the name to an internal RID, or Reserved ID, value. Another important task by the front-end is to convert decimal float con-

stant strings to the GCC internal representation, REAL_VALUE_TYPE.

GCC has also been modified to emit builtin defines to be used by an implementations of `decfloat.h` macros. These builtin defines are quite trivial to implement,

However, unlike the corresponding regular `float.h` macros the `decfloat.h` versions should be identical across implementations (as the C extension is tightly coupled to the IEEE 754 decimal floating-point), so might not *really* be needed. However, the builtins are not difficult to implement, so remain here for flexibility.

Other issues handled in the front-end include rejecting invalid combinations of type-specifiers, such as a `short _Decimal64` and disallowing complex decimal float types. Additionally, this layer provides various warnings for conditions such as unsafe type conversions. However, the **-Wformat** I/O format checking is an example of one diagnostic not implemented yet.

2.2 Middle-end: Internal Representation

The middle-end is where things start to get interesting.

GCC generically represents floating-point values via the REAL_VALUE_TYPE. The internal representation stores real values in it own encoding within the following structure:

Additionally, there are a set of interfaces exported from `real.h` that can be used by the components of GCC to operate on REAL_VALUE_TYPE structures, including the ability to do compile-time arithmetic, comparisons, and conversions.

This generic representation, is eventually encoded into to a format appropriate for the underlying hardware (most often IEEE 754). Target formats are represented by `struct real_format` and include function hooks to encode to the target format and decode back to the internal format.

An important issue is how to represent a decimal floating-point value internally, as converting into the current internal binary format is not really acceptable, as this incurs the same inexact encoding issues that decimal floating-point is trying to avoid in the first place. Creating a new structure, was not acceptable as this would duplicate all sorts of code, whereas much of the GCC source would not care whether this was really a decimal floating point, instead just needing the various abstract operations to succeed. Two new files, `dfp.h` and `dfp.c`, hide most of the gory details between mapping between GCC needs and the decNumber library.

A bit was stolen from the `uexp` field of the REAL_VALUE_TYPE to define a new `decimal` bitfield, to differentiate between a regular (binary) real and the new decimal internal representation. For example, this field is used to override the behavior of operations executed against REAL_VALUE_TYPEs, such as comparison or arithmetic.

The current implementation stores a decimal128 encoded value as the internal representation of a decimal float. Now, as mentioned in section 1.3, the decNumber library has its own internal format for most of the interesting operations. Consequently, this means that in implmenting the various *real* interfaces, one must first convert out of decimal128 format into a `decNumber` [5] At a later stage of development, it could be investigated whether worthwhile to increase the size of a REAL_VALUE_TYPE large enough to hold a `decNumber` structure in hopes of speeding up compilation.

[5] Of course, encoding to decimal128 target format becomes a no-op!

```
/* Maximum exponent for _Decimal128 type. */
builtin_define_with_int_value ("__DEC128_MAX_EXP__", 6144);
/* Maximum representable _Decimal32. */
builtin_define_with_value ("__DEC32_MAX__", "9.999999E96DF", 0);
```

```
/* From real.h */
struct real_value GTY(())
{
  /* Use the same underlying type for all bit-fields, so as to make
     sure they're packed together, otherwise REAL_VALUE_TYPE_SIZE will
     be miscomputed.  */
  unsigned int /* ENUM_BITFIELD (real_value_class) */ cl : 2;
  unsigned int decimal : 1;
  unsigned int sign : 1;
  unsigned int signalling : 1;
  unsigned int canonical : 1;
  unsigned int uexp : EXP_BITS;
  unsigned long sig[SIGSZ];
};

#define REAL_VALUE_TYPE struct real_value
```

Currently, the overriding of `real` behavior relies upon checking the `decimal` field and calling out to a decimal float specific function from within the `real.c` provided functions. Rather than cluttering `real.c` with such callouts, the current set of exported interfaces and macros could be turned into wrappers that call the appropriate routine in either `real.c` or `dfp.c` without affecting callers. At the bottom of `real.h`, there appear to be macros, for example `REAL_CONVERT`, that could be used as such wrappers. However, these macros do not appear to consistently used across the GCC source code.

The decNumber library was heavily relied upon to provide the internal arithmetic, comparisons, and other fundamental operations. A `decNumberNegate` macro was added to provide simpler implementation of the internal *negate* operation.

Besides the various arithmetic operations mentioned, the `dfp.c` module implements various conversion routines, including routines to convert between various precisions of both decimal and binary floating-point types. One inefficiecy in the conversions between binary and decimal representations, is that the values are converted first to an intermediate string format and then converted back down to the needed type. This implementation choice was due to the convenience of string conversion utility functions already available in GCC.

In GCC's tree representation of a program, types have corresponding type nodes, and as such, there are new type nodes to represent the 3 new decimal float types. Additionally, each type will map need to map to to an RTL based machine mode. New RTL machine modes of *SDmode*, *DDmode*, and *TDmode*, represent the IEEE 754R 4-byte, 8-byte, and 16-byte decimal floating-point types.

One (possibly very) troublesome issue with the middle-end, is a wide-spread use of the `TYPE_PRECISION` macro in ways that don't consider that there may be more than one `REAL_TYPE` that has an identical result and that the underlying type might not actually be binary. The `TYPE_PRECISION` macro indicates the "number of bits used in the representation." [GCC1] For example, the `TYPE_PRECISION` of a `float_type_node` may be identical to the type node of a `_Decimal32`. A search on the GCC sources shows many comparisons and consequent actions that need investigating to determine whether safe, though a large number of them are strictly with non-real types that will not cause problems. It may be possible to instrument the `TYPE_PRECISION` macro to assist in rooting out problems. To date, this issue has not been addressed sufficiently (though various instances have been fixed-up as discoverd to be causing problems.)

2.3 Back-end: Software Emulation

While soft-dfp support should be possible across most, if not all, architectures, there is still some back-end specific efforts to make it work. As such, the new decimal float machine modes are defined by the backend and the work described herein is implemented on the rs6000 backend.

As GCC already provides examples of floating-point emulation, the same patterns were followed to plug in soft-dfp support. The `dfp-bit.c` module implements functions that are linked into `libgcc` to provide the actual implmentation of the operations needed to implement the runtime emulation of decimal float support. Unsuprisingly, there are arithmetic operations, comparison operations, as well as, conversion operations, and are backed by decNumber under the covers.

Another back-end responsibility is how to pass and return these types. This has not yet been implemented, but passing of arguments seems to be working by luck, possibly falling into logic that decides how to pass floats of sizes already handled by the rs6000 back-end.

Additionally, the back-end will need to be taught how to move these new decimal float types around. The current plan is to create a new file, `decimal.md` to hold the new patterns. The move patterns, as well as, the ABI issues described, could be implmented with GPRs and/or memory, or could even use FPRs, if available for a target.

Currently, there is no method to pass a rounding-mode down into `libgcc` and consequently, the current IEEE 754 binary floating-point software emulation only supports a default rounding mode. While the decNumber library implements the various IEEE 754R rounding-modes, and this situation might be addressable for soft-dfp, this issue is likely to be deferred for later exploration.

2.4 Generating Tests from dectest

Complementing the decNumber library, the **dectest** testsuite [Cowl7] defines and provides a language independent testsuite for the General Decimal Arithmetic. The testcases are described in high level directives, operations, conditions. The dectest testsuite is available at `http://www2.hursley.ibm.com/decimal/dectest.zip`

For example, a simple dectest testcase might look like:

```
addx361 add 0E+50 10000E+1 ->
1.0000E+5
```

A clever idea from Ben Elliston is to autogenerate C language testcases from the dectest test-

suite. The General Decimal Arithmetic specification is much broader than the needs of IEEE 754R base decimal floating-point needs, so only some of the tests are relevant. An example autogenerated test follows:

```
#include <assert.h>
#include <fenv.h>
#include <math.h>
int main ()
{
  union {
    _Decimal32 d;
    unsigned char bytes[4];
  } u;

/*  FIXME: Set the
           rounding mode.  */
/*  assert (fsetround
    (FE_DEC_TONEAREST) == 0); */

  u.d = 0E+50df + 10000E+1df;
  return (!(u.d == 1.0000E+5df));
}
```

To enable this testsuite, one first needs to download the testsuite, then export a `DECTEST` environment variable pointing to the location of the unzipped dectest sources. Running the GCC testsuite will execute the testsuite, or one can explicitly test via command of `make check-gcc RUNTESTFLAGS=''dectest.exp''`.

3 Summary

As is probably apparent, much of the remaining work (and yet to be discovered work) is in the the back-end. Simple arithmetic testcases seem to work, including cases involving constant-folding optimizing away the arithmetic. Another currently unimplemented feature is NaN support[6].

There are still several thorny issues and/or unknowns to deal with, but hopefully, a few of the concerns will have been handled in the next couple months even.

Besides the compiler implementation, there is also work underway to make complimentary changes to the GDB debugger and the GLIBC library. There is a proposal to DWARF Standards Committe to add support for the new types specified by the C extension.

To date, the C extension for decimal floating-point provides a reasonable mapping of IEEE 754R decimal floating-point into the C language. Decimal floating-point math is hoped to be simpler to program for and hopefully hardware implementations will follow someday to address the performance issues driving IEEE 754R decimal floating-point standardization.

Overall, GCC is fantastic playground to test and explore the C extension for decimal floating-point. GCC is uniquely positioned to propagate this extension to a wide variety of OS and processor platforms. Hopefully, this initial effort can serve as a foundation for decimal floating-point support in GCC and welcome ideas for improvements to the implementation by the audience [7].

References

[Sch1] M.J. Schulte, N. Lindberg, and A. Laxminarain, *Performance Evaluation of Decimal Floating-Point Arithmetic*, IBM 6th Annual Austin Center for Advanced Studies

[6] Need to check whether this is implemented on regular soft-float even.

[7] Old (lame) joke: *There are 10 kinds of people in this world, those who know binary and those who don't.*— Unknown Author

[Cowl1] M. Cowlishaw, *Decimal Floating Point: Algorism for Computers*, Proceedings of the 16th IEEE Symposium on Computer Arithmetic. Available at `http://www2.hursley.ibm.com/decimal/IEEE-cowlishaw-arith16.pdf`

[Hol1] S. Hollasch, *IEEE Standard 754 Floating Point Numbers*, Available at `http://stevehollasch.com/cgindex/coding/ieeefloat.html`

[IEEE1] IEEE 754R Working Group, *Draft Standard for Floating-Point Arithmetic P754/D0.10.9*, Available at `http://754r.ucbtest.org/drafts/754r.html`

[Sev1] C. Severance, *An Interview with the Old Man of Floating-Point*, Available at `http://cch.loria.fr/documentationj/IEEE754/wkahan/754story.html`

[Cowl2] M. Cowlishaw, *Decimal Encoding Specification Strawman 4d*, Available at `http://www2.hursley.ibm.com/decimal/decbits.html`

[Cowl3] M. Cowlishaw, *Decimal Arithmetic FAQ (Frequently Asked Questions)*, Available at `http://www2.hursley.ibm.com/decimal/decifaq.html`

[Cowl4] M. Cowlishaw, *A Summary of Densely Packed Decimal encoding*, Available at `http://www2.hursley.ibm.com/decimal/DPDecimal.html`

[CExt1] JTC-1/SC22/WG14, *ISO/IEC DTR 24732, Working Draft 5, Programming languages, their environments and system interfaces - Extension for the programming language C to support decimal floating-point arithmetic*, Available at `http://www.open-std.org/jtc1/sc22/wg14/wwww/docs/n1107.htm`

[C99] ISO/IEC, *Programming Languages - C*, ANSI/ISO/IEC 9899-1999

[Cowl5] M. Cowlishaw, *The decNumber C library*, Available at `http://www2.hursley.ibm.com/decimal/decnumber.pdf`

[Cowl6] M. Cowlishaw, *General Decimal Arithmetic specification*, Available at `http://www2.hursley.ibm.com/decimal/decarith.pdf`

[GCC1] Free Software Foundation, *GCC Internals Manual*, Available at `http://gcc.gnu.org/onlinedocs/gccint/`

[Cowl7] M. Cowlishaw, *General Decimal Arithmetic Testcases*, Available at `http://www2.hursley.ibm.com/decimal/dectest.pdf`