Precise target floating-point emulation in GDB

Dr. Ulrich Weigand
Senior Technical Staff Member
GNU/Linux Compilers & Toolchain

Date: Sep 8, 2017

Agenda

- **Floating-point formats – background**
- **Floating-point types in DWARF debug info**
- **Floating-point operations in GDB**
- **Implementation status**

A motivating example

- **Print the value of “x” in GDB on powerpc64le-linux:**

```
__float128 x = 1e1000q;
```

- **GDB as of ~ 1 year ago:**

```
(gdb) print x
$1 = 1.32705595564329922141847850965928887e+116
```

- **GDB as of today:**

```
(gdb) print x
$1 = inf
```

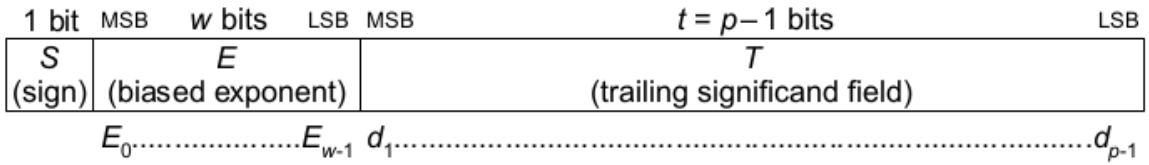
- **GDB including the “Target FP” patch set:**

```
(gdb) print x
$1 = 9.99999999999999999999999999999999957e+999
```


Floating-point formats – background

Floating-point formats (IEEE)

- **IEEE 754 binary interchange format encodings**



  - E == 0 encodes zero (if T == 0) and subnormal numbers (T != 0), where implied d_0 bit is zero
  - E == 2^w-1 encodes infinity (if T == 0) and NaN values
  - Other values of E encode normal numbers, where implied d_0 bit is one
  - Value of (sub)normal numbers defined as: $(-1)^S * 2^{E\text{-bias}} * d_0.d_1d_2...d_{p-1}$
  - Most commonly used variants of the IEEE 754 format include:
    - binary32: 32 bits in storage, precision 24, e_{max}/bias 127
    - binary64: 64 bits in storage, precision 53, e_{max}/bias 1023
    - binary128: 128 bits in storage, precision 113, e_{max}/bias 16383

Floating-point formats (Other)

- **Intel 80-bit extended precision format**
  - Similar in concept to IEEE types, but with some differences
  - 96 bits (i386) or 128 bits (x86_64) in storage, padded from 80 value bits
  - 15-bit exponent field (like IEEE binary128)
  - 64-bit significand field with **no** implicit bit
  - Top two bits of significand define value class, including several non-IEEE types (“denormal”, “pseudo-denormal”, “unnormal”, “pseudo-NaN”, “pseudo-Inf”, …)
- **IBM double double format**
  - 128 bits in storage, interpreted as two double (IEEE binary64) values
  - Special values (0, Inf, NaN) encoded in first double, with second == 0
  - Normal values determined as the sum of the two double values
    - Allows representing the same exponent range as double, and approximate twice the precision
    - Also allows representing other values like $1 + 2^{-1000}$
    - Not equivalent to any IEEE format

Floating-point types

- **Standard C/C++ types**
  - The language standard defines “float”, “double”, and “long double”
  - The standard does **not** define what encoding is used
  - Many platform ABIs use IEEE binary32 as “float” and binary64 as “double”
    - But not all: e.g. on SPU, “float” is 32 bit, but not IEEE binary32
  - No wide-spread standard for “long double”
    - Historically, some platforms used IEEE binary64 as well!
    - Some platforms use IEEE binary128 (e.g. s390)
    - Intel uses the 80-bit extended precision format
    - Power uses the IBM double double format
- **Other data types**
  - GCC defines “__float128” on some platforms (Intel and Power), which is encoded as IEEE binary128
  - TS 18661-3 defines various additional types, including _Float128, which is also encoded as IEEE binary128

Floating-point types in DWARF debug info

Back to the motivating example

- **Why did this happen?**

```
__float128 x = 1e1000q;

(gdb) print x
$1 = 1.3270559556432992214184785096592887e+116
```

- **GDB “thinks” the type of “x” is “long double”!**
  - DWARF debug data cannot distinguish between __float128 and long double in the Power ABI
  - The bit pattern of “x” is interpreted as if it were the pattern of a “long double” (i.e. a pair of doubles)

Back to the motivating example (cont.)

- **DWARF debug dump for the binary**

```
<1><2d>: Abbrev Number: 2 (DW_TAG_variable)
   <2e>    DW_AT_name          : x
   <30>    DW_AT_decl_file     : 1
   <31>    DW_AT_decl_line     : 2
   <32>    DW_AT_type          : <0x40>
   <36>    DW_AT_external      : 1
   <36>    DW_AT_location      : [...]

<1><40>: Abbrev Number: 3 (DW_TAG_base_type)
   <41>    DW_AT_byte_size     : 16
   <42>    DW_AT_encoding      : 4         (float)
   <43>    DW_AT_name          : [...]: __float128
```

- **The only information provided in DWARF:**
  - __float128 is a “float” type of size 16 bytes

What does the DWARF standard say?

- **DWARF 5 Section 5.1 “Base Type Entries”**
  - A base type is represented by a debugging information entry with the tag DW_TAG_base_type.
  - A base type entry has a DW_AT_encoding attribute describing how the base type is encoded and is to be interpreted.
  - A base type entry has a DW_AT_byte_size attribute or a DW_AT_bit_size attribute whose integer constant value […] is the amount of storage needed to hold a value of the type.
  - For example, the C type int on a machine that uses 32-bit integers is represented by a base type entry with a name attribute whose value is “int”, an encoding attribute whose value is DW_ATE_signed and a byte size attribute whose value is 4.

What does the DWARF standard say? (cont.)

- **DWARF 5 Section 5.1.1 “Base Type Encodings”**
  - A base type entry has a DW_AT_encoding attribute describing how the base type is encoded and is to be interpreted. The value of this attribute is an integer of class constant.
  - Types with binary floating-point encodings (DW_ATE_float, DW_ATE_complex_float and DW_ATE_imaginary_float) are supported in many programming languages and are not discussed further.
  - DW_ATE_decimal_float specifies floating-point representations that have a power-of-ten exponent, such as specified in IEEE 754R.

Consequences of DWARF standard wording

- **Binary floating-point format not specified**
  - DWARF simply relies on platform ABI
- **Different FP types distinguished solely by size**
  - The only distinguishing characteristic provided for by the DWARF standard is the byte size (and optionally the bit size)
  - Assumes that no platform ABI supports two different base floating-point types of the same size
- **This cannot support some current ABIs:**
  - "long double" and "__float128" have the same size but use different floating-point format
  - True on Power, but also on Intel (due to padding!)

What to do about this in GDB?

- **Current “solution” - hard-code type names**

```
const struct floatformat **
ppc_floatformat_for_type (struct gdbarch *gdbarch,
                          const char *name, int len)
{
  if (len == 128 && name)
    if (strcmp (name, "__float128") == 0
        || strcmp (name, "_Float128") == 0
        || strcmp (name, "_Float64x") == 0
        || strcmp (name, "complex _Float128") == 0
        || strcmp (name, "complex _Float64x") == 0)
      return floatformats_ia64_quad;

  return default_floatformat_for_type (gdbarch, name, len);
}
```

- **Implemented as of 2016-09-05**

Moving forward: Proposing a “real” solution

- **Extend the DWARF standard**
  - Add optional attribute to DW_TAG_base_type, which may be present for base types where DW_AT_encoding specifies any binary floating-point encoding
  - Named e.g. “DW_AT_encoding_variant”, with a constant integer value
  - Semantics of the “variant” value remain unspecified by DWARF, thus platform ABI-defined
  - Missing variant (or variant 0) identifies the same format as today, other variant values identify additional formats
- **Implement support for the new attribute**
  - In DWARF producers like GCC
  - In DWARF consumers like GDB

Floating-point operations in GDB

Once again the motivating example

- **Why does this still happen?**

```
__float128 x = 1e1000q;

(gdb) print x
$1 = inf
```

- **GDB cannot handle this particular value!**
  - GDB now correctly interprets “x” as IEEE binary128
  - The pattern is read in and converted to a host floating-point value of type DOUBLEST (i.e. long double)
  - That host value is then printed using host printf
  - But host long double is “IBM double double” – which cannot represent 10^{1000} – value overflows!

GDB FP operations – basic principles

- **GDB operates in host floating-point arithmetic**
  - Values held in variables of type DOUBLEST
    - Historically a typedef for the largest available standard FP type
    - These days (C++11!) always equals “long double”
  - Only operations on target FP are conversions from and to DOUBLEST
  - All other operations (printing, scanning, conversions, arithmetic, comparisons, …) done in host C++ code using DOUBLEST
- **Advantages**
  - GDB code easy to write, has good performance
- **Disadvantages**
  - Operations on host DOUBLEST may not match target arithmetic
    - e.g. different rounding effects
  - Host DOUBLEST may not be able to hold target values at all
    - e.g. cross-debugging where host “long double” != target “long double”
    - But also native debugging of types larger than “long double”!

Proposal: GDB operations on target FP

- **Proposed new basic principles**
  - GDB now holds floating-point values always as byte buffers in target floating-point format – no more DOUBLEST!
  - A new set of helper routines provide all required basic operations, working on such typed byte buffers
  - Implemented to precisely emulate the target
    - Currently making use of the GNU MPFR library
- **Advantages**
  - All target values representable, same results as target
  - Unifies handling of binary FP and decimal FP in GDB code
- **Disadvantages**
  - GDB now dependent on external library (MPFR)
  - Performance of FP operations is worse

Open issues: Implementation of target FP

- **Proposed solution: GNU MPFR**
  - Existing, well supported / tested implementation
  - Already widely used e.g. in GCC
  - However: new GDB dependency on external library
    - Code requires at least MPFR 3.1
- **Is the dependency issue a problem?**
  - On Linux probably not
    - GDB now requires C++11 to build
    - All major distros with C++11 (gcc >= 4.8) also provide MPFR >= 3.1
  - What about other host platforms?
- **Alternatives?**
  - Make MPFR optional, falling back to host arithmetic if not present
  - Open-code target FP emulation code, e.g. like GCC's real.c
    - But GDB may require more operations than real.c provides
    - Full support may be difficult to implement / maintain

Open issues: Performance

- **Is there significant performance degradation?**
  - MPFR calls of course much slower than host arithmetic
  - But “hot” code paths in GDB (e.g. unwinding, single-stepping) do not perform any FP arithmetic at all!
  - Counter-examples can always be constructed
    - Matrix multiply in a GDB script
    - Software watchpoint condition involving FP ops
- **Alternatives?**
  - Check whether the target FP format matches an available host FP format, and fall back to host arithmetic if so
  - Is this worth the additional code paths / potential complication of test cases?

Implementation status

Proposed implementation

- **Patch set “Target FP”**
  - RFC posted to gdb-patches on 2017-09-05
  - Any feedback welcome!
- **Basic layout of the patch series**
  - 00-02: Some preliminary clean up of binary FP handling.
  - 03: Some preliminary clean up of decimal FP handling.
  - 05-07: Simplify and fix binary FP printing.
  - 08: Remove DOUBLEST from expression parsing and binary FP scanning.
  - 09-12: Introduce target-float.{c,h} and move operations there.
  - 13-14: Remove some remaining uses of DOUBLEST in the code base.
  - 15-17: Final (trival) clean up patches.
  - 18: Remove doublest.{c,h} and dfp.{c,h}.
  - 19: Switch target-float.c to use MPFR if present.
-

Details: FP printing (unformatted)

- **Current status**
  - `print/f` command supposed to work on any data
    - If floating-point type, use it
    - If the same size as some standard FP type, use that
    - Otherwise, convert to DOUBLEST and print
  - `print_floating` makes many assumptions on host and target FP format, e.g. hard-coded number of digits to print
- **New approach**
  - `print_floating` **always** called with a FP type
    - Special cases handled otherwise by the caller
  - Determine properties of the target format algorithmically, e.g. by computing and using the appropriate DECIMAL_DIG value
  - New helper routine to generate string from target format buffer
- **Open issues**
  - Is the current `print/f` behavior actually useful?

Details: FP printing (formatted)

- **Current status**
  - `printf "%f", ...` and `printf "%lf", ...` convert to host "double" or "long double" and then use host printf
  - Same handling of special cases as `print/f`
  - DFP handled in a very different way
- **New approach**
  - Convert to **target** double or long double
    - For DPF types, similarly convert to target DPF type
    - Still handle the special cases for binary FP
  - Extend target FP to string helper to support format string
- **Open issues**
  - Usefulness of special cases (even more) questionable

Details: FP scanning / expression parsing

- **Current status**
  - Language parsers scan literal constants via sscanf, convert to DOUBLEST and store as OP_DOUBLE expression node
    - C also supports OP_DECFLOAT, stored as a 16-byte buffer in target format
  - Target **type** to be used is then determined, consulting any suffix if present
- **New approach**
  - Language parsers determine target type **first**
  - Then scan literals (DFP/BFP) into byte buffer in target format appropriate for the type, using a new helper routine
  - New expression node OP_FLOAT replaces OP_DOUBLE and OP_DECFLOAT, always using a target-format buffer
- **Open issues**
  - Pascal / Go parsers currently support “f” / “l” suffixes (including GDB test cases!) although this is not present in the language specs
  - Ada parser uses “sizeof (DOUBLEST)” to determine type to use ...

Details: FP conversions – current status

Everything via DOUBLEST

```
if (code1 == TYPE_CODE_FLT && scalar)
  return value_from_double (to_type, value_as_double (arg2));
else if (code1 == TYPE_CODE_DECFLOAT && scalar)
  {
    enum bfd_endian byte_order = gdbarch_byte_order (get_type_arch (type));
    int dec_len = TYPE_LENGTH (type);
    gdb_byte dec[16];

    if (code2 == TYPE_CODE_FLT)
      decimal_from_floating (arg2, dec, dec_len, byte_order);
    else if (code2 == TYPE_CODE_DECFLOAT)
      decimal_convert (value_contents (arg2), TYPE_LENGTH (type2),
                       byte_order, dec, dec_len, byte_order);
    else
      /* The only option left is an integral type.  */
      decimal_from_integral (arg2, dec, dec_len, byte_order);

    return value_from_decfloat (to_type, dec);
  }
```

DFP very different from BFP

Details: FP conversions – new approach

BFP and DFP

Detect invalid values

Includes BFP<→DFP conversions

Can now create FP value

```
if (is_floating_type (type) && scalar)
  {
    if (is_floating_value (arg2))
      {
        struct value *v = allocate_value (to_type);
        target_float_convert (value_contents (arg2), type2,
                              value_contents_raw (v), type);
        return v;
      }

    /* The only option left is an integral type.  */
    if (TYPE_UNSIGNED (type2))
      return value_from_ulongest (to_type, value_as_long (arg2));
    else
      return value_from_longest (to_type, value_as_long (arg2));
  }
```


Details: Ada fixed-point types

- **Current status**
  - Values of Ada fixed-point types are stored internally as integers; the value must be scaled before use by a rational number (num/den) defined by the type
  - This scaling is currently done in DOUBLEST
- **New approach**
  - Perform all scaling operations in target arithmetic (using normal value_binop / value_cast routines)
- **Open issues**
  - Which target type to use? Currently target long double.

Details: Scripting language interfaces

- **Current status**
  - The scripting languages (Python and Scheme) allow conversion between GDB values and objects in those languages
  - For floating-point values, the scripting language interfaces with external code (like GDB) operate on (host) “double” types
- **New approach**
  - We still convert target FP to/from host “double” where interacting with Python / Scheme
    - This is the only remaining place in GDB to do so
- **Open issues**
  - Is that Python / Scheme interface actually useful? Should we expose a different interface that constructs scripting language objects representing target FP instead?

Next steps

- **DWARF encoding for FP types**
  - Formally propose DWARF extension
  - Implement support in GCC and GDB
- **Target FP emulation**
  - Resolve outstanding open issues
  - Achieve consensus and merge patch series :-)
- **Our motivating example – here's what we want:**

```
__float128 x = 1e1000q;

(gdb) print x
$1 = 9.99999999999999999999999999999957e+999
```

Questions