The *Libm* Library and Floating-Point Arithmetic in HP-UX for Itanium®

Ren-Cang Li
Peter Markstein
Jon P. Okada
James W. Thomas

June 18, 2002

© Copyright 2002 Hewlett-Packard Company
1 Introduction

The HP-UX *libm* library provides the mathematical library for C, C++, and Fortran 90. The HP-UX *libm* library and compilers for the Itanium processor family (IPF) provide a leading combination of functionality, quality, and performance. They enable transitioning from PA-RISC and porting from other platforms, and they offer modern features that facilitate programming efficiencies that have not been practical heretofore. The all new *libm* implementation for Itanium, introduced last year in HP-UX 11i V1.5, has been upgraded with improved performance and overall quality in V1.6 for Itanium 2. This paper

1) describes the motivations, goals, and high-level development strategies for the *libm* library and the compiler floating-point facilities,

2) characterizes the V1.6 *libm* library in terms of the various dimensions of quality, and

3) discusses programming techniques that exploit the floating-point capabilities of HP-UX on IPF.

2 Libm motivations, goals, and strategies

The HP-UX development team was motivated by the conviction that programmers would be more effective with a full featured, dependable, high performance floating-point facility and that end users would appreciate the ensuing improved robustness and performance in applications. The facility should provide programmers with all the functions they might reasonably expect. The functions should be highly accurate, defect free, and handle special cases predictably according to existing standards. Performance should be the best the hardware can support. Hence the goal for the *libm* library has been to excel in all these quality dimensions:

- accuracy
- correctness (no defects)
- functionality
- special cases
- speed
- standards

Customers haven't yet demanded the same high level of quality from their software development tools as they expect from the underlying hardware, though recent standards, e.g. C99, reflect movement in that direction. Ideally, customers wouldn't have to choose between speed and other quality aspects. But some of the quality goals pull implementation in opposing directions. Meeting strict specifications for accuracy and special cases can prove especially difficult because it generally requires extra CPU cycles, conflicting with the high priority on performance. The development team believed the easier, single-focus approaches -- careful but slow and fast but sloppy - wouldn't satisfy a broad spectrum of customer needs into the future. Instead, the development team undertook to: clarify the behaviors implied by the compile options and pragmas available to the programmer; assure the dependability of the specified behaviors; and, for each of the popular combinations of options and pragmas, provide the best performance possible while still meeting the behavior expectations promised by those controls.

2.1 IPF floating-point features

The IPF architecture is well suited for addressing multiple quality goals, as it provides features specifically designed to enable fast, high quality floating-point code. Instruction level parallelism allows extra computa-
tion (for greater accuracy or special cases) without significantly increasing the latency of the whole routine.
The fully supported 80-bit floating type protects internal calculations in single and double precision routines against (avoidable) overflow, underflow, and loss of accuracy - without requiring extra code or execution
cycles. Similarly, the extra exponent range in the 82-bit floating-point registers protects extended (80-bit) routines against over/underflow. Algorithms utilizing Floating-point Multiply Add (FMA) make for faster elementary function codes. The wide registers and FMA facilitate faster quad (128-bit) elementary functions. Alternate status fields allow libm routines to manage the modes and flags in the Floating-Point Status Register (FPSR), without having to save and restore floating-point state.

2.2 HP-Intel joint algorithm development

In the 1994-1995 timeframe, as part of the joint HP-Intel development of the IPF architecture, a joint floating-point task force developed a set of division and square root routines, since these functions are considered part of the architecture. HP Labs contributed the routines for float, double, and extended precisions, and Intel provided independent verification of correctness of these algorithms. Once the task force activity terminated, HP and Intel independently developed their own mathematical libraries.

In 1999, Intel and HP compared notes on the division and square root algorithms. Each company had improved the original set, and jointly, we discovered even faster, shorter algorithms. Of the dozen algorithms developed by the task force, only one (double precision high throughput division) is still in use today.

HP and Intel developed transcendental function implementation with different objectives. HP initially strove to find algorithms that would give the best performance in a software-pipelined environment, while still adhering to standards and achieving excellent rounding behavior. Intel chose to find routines that would minimize execution time as closed subroutines [6]. HP has since shifted to a mixed strategy for its libm library, finding algorithms that can be software pipelined effectively and that allow short latency closed subroutines. As a result, the HP library will produce identical results whether functions are called conventionally or are in-lined.

2.3 C99 specification

The ISO/IEC C99 standard embodies much of the work from the earlier Numerical C Extensions Group whose charter was to enhance the C language to become attractive for numerical programming. C99 includes a binding for the IEEE 754 standard for floating-point arithmetic (IEC 60559 is the international version). IEEE 754 was implemented in hardware well before it became an official standard in 1985 and now essentially all new floating-point hardware conforms to it. Its standard data formats and predictable results for basic operations have eclipsed the alternative of multiple implementation-specific, mutually inconsistent arithmetics, and are now taken for granted by programmers and end users.

However, IEEE 754 does not include programming language bindings, and the task has proved challenging for language standard committees. Thus, other IEEE 754 features -- including special values (NaNs, infinities, and signed zeros), rounding modes, and exceptions, though supported in hardware, have not been available for practical and portable use. Also, compilers and libraries, focusing on performance priorities, have often undermined the overall predictability and robustness one might hope for based on the rigorousness of the basic operations. C99 at long last makes features of IEEE 754 usable for high-level language programmers through a standard API, and provides mechanisms for overall predictability and robustness with minimal impact on performance.

C99 substantially expands the C89 math library, adds complex arithmetic and functions, adds a facility for type-generic math functions (whose types are determined by the argument types), and provides specification for implementations supporting IEEE 754, with details for IEEE 754 compatible math functions and com-
The libm development team whole-heartedly adopted the C99 specification, not only for the benefits noted above but also because designing its own specification of comparable quality would have been enormously time consuming and still would not have been standard.

The C99 specification in a few cases conflicts with previously documented behavior of HP-UX for PA-RISC systems and with the Unix 95 standard (Unix 200x is adopting C99). For example, the older specifications require mathematical functions to set errno and in a few special cases to return different values. HP-UX math functions for IPF adhere to C99, while providing the +Olibmerrno option to allow users to request errno setting and return values according to the older specifications.

2.4 Implementation

High-level vs. assembly language

High performance elementary functions are usually hand coded in assembly language, to make use of features in the underlying hardware that are not surfaced in high-level programming languages and to deploy tricks that may be missed in compiler optimized code. The performance gains usually justify the extra development and maintenance costs associated with assembly language code. However, assembly language programming for IPF is especially problematic. Scheduling for IPF entails organizing instructions into groups of instructions that can execute concurrently and that fit into three-instruction bundles within the group. Numerous exceptions to the basic bundling scheme confound the scheduling job. Once instructions are packed in, any change may trigger a massive rearrangement. Different instructions have different latencies, and latency can depend on which instruction will be consuming the result. And, the complicated latency scheme will vary from one implementation of the architecture to the next. An optimal schedule for the first Itanium processor may be significantly less than optimal for Itanium 2. This means that high-performance assembly language codes may need to be repeatedly redone.

In order to avoid the looming development and maintenance costs for assembly language programming, the libm team opted to code in C and to leave the scheduling task to the compiler. To utilize the IPF features designed for, and essential to, high performance mathematical functions, the libm team worked closely with the compiler teams to develop auxiliary C language extensions. Key was inline assembly, which provided almost all the needed IPF instructions in the form of C functions. Other extensions helped with branch prediction, access to IPF’s alternate floating-point status fields, and type determination (for C99 type-generic macros and functions).

Optimization shortcomings (as well as hardware access) can justify coding in assembly language. The push to develop fast libm routines written in high-level language provided valuable feedback to the compiler teams, as testing and tuning uncovered numerous defects and optimization opportunities. Thanks to the responsiveness of the compiler teams, the performance of key functions is approaching, and in some cases equal to, that of hand coded assembly.

Other implementation techniques

In HP-UX 11i V1.6, at +O2 optimization level and higher, the treatment of trigonometric functions allows sine and cosine with the same argument to be evaluated in the same time as only one of these functions, with no special syntax imposed on the user. The libm complex functions, for example, benefit from this efficiency.

The V1.6 compilers are capable of moving seldom used code away from the code body, leading to software piped sequences that are very fast in ordinary cases. At +O3, the inlined sequences for math functions, particularly the trigonometric functions, benefit from this technique. See Tables 7 and 8 in 3.5.
3 Quality dimensions

3.1 Functionality

The HP-UX *libm* library and C and C++ compilers for IPF support four different floating formats and five types. The 11i V1.6 Fortran 90 compiler supports three of the formats.

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Encoding (sign/exponent/implicit bit+explicit bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>real, or real*4</td>
<td>1/8/1+23</td>
</tr>
<tr>
<td>double</td>
<td>real*8, or double precision</td>
<td>1/11/1+52</td>
</tr>
<tr>
<td>extended</td>
<td></td>
<td>1/15/0+64</td>
</tr>
<tr>
<td>quad</td>
<td>real*16</td>
<td>1/15/1+112</td>
</tr>
<tr>
<td>long double</td>
<td>real*16</td>
<td>equivalent to quad</td>
</tr>
</tbody>
</table>

IPF hardware fully supports the float, double, and extended types, with comparable performance for all.

For compatibility with PA-RISC, HP-UX for IPF maps the long double type in the C and C++ programming languages to the quad type. Other vendors’ compilers may map long double to different underlying types. The extended and quad nomenclatures are designed to allow reference to fully specified types, and are recommended over long double wherever the use depends on the particular characteristics (size, precision, range, etc.) of the type.

The extended and quad types amount to extensions to the programming language. They are defined by the inclusion of one or more of the headers *math.h*, *float.h*, *stdlib.h*, and *complex.h*. To avoid conflicts in existing programs that use the names for other purposes, the definitions of the extended and quad types, as well as all the functions and macros involving the types, become visible only if the *-fpwidetypes* compile option is used. (The standard long double nomenclature is always visible.) Illustrating the nomenclature, the *math.h* header contains

```c
float expf(float);
double exp(double);
long double expl(long double);
#ifdef _FPWIDETYPES  // defined by use of the -fpwidetypes option
   extended expw(extended);
   quad expq(quad);
#endif
```

The HP-UX *cmath*, *complex*, and *limits* C++ headers support, through overloading, the 80-bit extended type and almost all the functions available with the C headers. The *libm* library supplies the
underlying implementation for the C++ mathematical functions in these headers.

The HP-UX Fortran 90 compiler for IPF uses *libm* functions for the standard mathematical intrinsics, but does not yet extend the set of intrinsics to all the *libm* functions and does not support the 80-bit type.

The *libm* library for IPF provides the C functions listed in Table 2 below, in all the supported types (except as noted).
### Table 2: Functions

<table>
<thead>
<tr>
<th>Function/macro family</th>
<th>C double precision functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>exponentials</td>
<td>exp, exp2, exp10, expm1</td>
</tr>
<tr>
<td>logarithms</td>
<td>log, log2, log10, log1p</td>
</tr>
<tr>
<td>trigonometrics</td>
<td>cos, sin, tan</td>
</tr>
<tr>
<td>inverse trigonometrics</td>
<td>acos, asin, atan, atan2</td>
</tr>
<tr>
<td>trigs with degree arguments</td>
<td>cosd, sind, tand</td>
</tr>
<tr>
<td>inv trigs with degree outputs</td>
<td>acosd, asind, atand, atan2d</td>
</tr>
<tr>
<td>hyperbolics</td>
<td>cosh, sinh, tanh</td>
</tr>
<tr>
<td>inverse hyperbolics</td>
<td>acosh, asinh, atanh</td>
</tr>
<tr>
<td>square &amp; cube roots, reciprocal square root, hypotenuse</td>
<td>sqrt, cbrt, rsqrt, hypot</td>
</tr>
<tr>
<td>power</td>
<td>pow, pow2, powlhn</td>
</tr>
<tr>
<td>financial/growth</td>
<td>compound, annuity</td>
</tr>
<tr>
<td>exponent</td>
<td>logb, ilogb, frexp</td>
</tr>
<tr>
<td>scaling</td>
<td>scalbn, scalbin, scalb, ldexp</td>
</tr>
<tr>
<td>rounding to integral floating</td>
<td>ceil, floor, trunc, round</td>
</tr>
<tr>
<td>IEEE ending to integral floating</td>
<td>rint, nearbyint</td>
</tr>
<tr>
<td>“Fortran” rounding to integer</td>
<td>ilround, llround</td>
</tr>
<tr>
<td>IEEE rounding to integer</td>
<td>lrint, llrint</td>
</tr>
<tr>
<td>integral and fractional parts</td>
<td>modf</td>
</tr>
<tr>
<td>remainder/mod</td>
<td>remainder, remquo, fmod</td>
</tr>
<tr>
<td>error functions</td>
<td>erf, erfc</td>
</tr>
<tr>
<td>gamma functions</td>
<td>lgamma, lgamma_r, tgamma</td>
</tr>
<tr>
<td>next value</td>
<td>nextafter, nexttoward</td>
</tr>
<tr>
<td>absolute value &amp; copy sign</td>
<td>fabs, copysign</td>
</tr>
<tr>
<td>max, min, &amp; positive difference</td>
<td>fmax, fmin, fdim</td>
</tr>
<tr>
<td>Bessel functions of 1st kind</td>
<td>j0, j1, jn</td>
</tr>
<tr>
<td>Bessel functions of 2nd kind</td>
<td>y0, y1, yn</td>
</tr>
<tr>
<td>classification</td>
<td>isnan, isinf, isfinite, isnormal, signbit, fpclassify</td>
</tr>
<tr>
<td>“quiet” comparison</td>
<td>isless, islessequal, isgreater, isgreaterequal, isunordered, islessgreater</td>
</tr>
<tr>
<td>complex exponential &amp; log</td>
<td>cexp, ciog</td>
</tr>
<tr>
<td>complex trigs</td>
<td>ccos, csin, ctan</td>
</tr>
<tr>
<td>complex inverse trigs</td>
<td>cacos, casin, catan</td>
</tr>
<tr>
<td>complex hyperbolics</td>
<td>csinh, csinh, ctanh</td>
</tr>
<tr>
<td>complex inverse hyperbolics</td>
<td>cacosh, casinh, catanh</td>
</tr>
<tr>
<td>complex magnitude &amp; modulus</td>
<td>cabs, carg</td>
</tr>
<tr>
<td>complex square root and power</td>
<td>csqrt, cpow</td>
</tr>
<tr>
<td>complex parts, conjugate, &amp; projection</td>
<td>conj, creal, cimag, cproj</td>
</tr>
<tr>
<td>exception flags</td>
<td>feclearexcept, fetestexcept, feraiseexcept</td>
</tr>
<tr>
<td>opaque exception flag state</td>
<td>fegetexceptflag, fetestexceptflag</td>
</tr>
<tr>
<td>rounding direction modes</td>
<td>fegetround, fetestround</td>
</tr>
<tr>
<td>FP environment as a whole</td>
<td>fegetenv, fetestenv</td>
</tr>
<tr>
<td>hiding exceptions</td>
<td>feholdexcept, feupdateenv</td>
</tr>
<tr>
<td>flush to zero mode</td>
<td>fegetflushzero, fetestflushstozero</td>
</tr>
<tr>
<td>trap enablement</td>
<td>fegettrapenable, fetesttrapenable</td>
</tr>
</tbody>
</table>
The \texttt{rsqrt}, \texttt{pown}, and \texttt{powlln} functions and the \texttt{extended} and \texttt{long double (quad)} complex functions are new for V1.6, as is the C compiler support for the \texttt{extended} and \texttt{long double (quad)} complex and imaginary types.

The \texttt{libm} library in the HP-UX 11i V1.6 release provides only double versions of the Bessel functions.

The API for C mathematical functions in HP-UX for IPF is a major superset of the 11i release for PA-RISC. HP-UX for IPF has complete sets of functions of types \texttt{long double (quad)} and \texttt{extended}, none of which are available in 11i for PA-RISC. It has a complete set of \texttt{float} functions; 11i for PA-RISC has about half of them. It has a few new functions - \texttt{compound}, \texttt{annuity}, \texttt{exp10}, \texttt{fma}, \texttt{scalbln}, \texttt{tgamma}, \texttt{nextrtoward}, \texttt{rsqrt}, \texttt{pown}, \texttt{powlln} - in all floating-point precisions. It has the complex and imaginary functionality specified in C99. 11i C for PA-RISC does not support complex and imaginary types. It provides the C99 \texttt{tgmath.h} header for type-generic mathematical functions, which is not in 11i for PA-RISC.

3.2 Standards

The HP-UX C math library for IPF, as represented by the headers

\begin{verbatim}
complex.h
fenv.h
float.h
math.h
tgmath.h
\end{verbatim}

adheres to the ISO/IEC C99 specification, including Annex F (IEC 60559 floating-point arithmetic) and Annex G (IEC 60559-compatible complex arithmetic). IEC 60559 is the international version of IEEE 754.

The \texttt{libm} library includes all the mathematical functions required by Unix 95. The +\texttt{Olibmerrno} (non-default) option provides versions of the mathematical functions that conform to the C89 and Unix 95 standards. Under +\texttt{Olibmerrno}, \texttt{math.h} functions set \texttt{errno} according to requirements of these standards and to C99's optional \texttt{errno} specification for \texttt{math.h} functions, and they return special-case results according to Unix 95 instead of C99, in the few cases where the standards differ. The upcoming Unix 200x standard aligns with C99.

The ISO C++ standard, which pre-dates C99, is based on C89. The HP-UX aCC product for IPF offers the full functionality of \texttt{libm} through extensions to the C++ standard libraries. The +\texttt{Olibmerrno} option for aCC provides cmath functions that set \texttt{errno}.

Adopting the C99 floating-point specification, HP-UX for IPF offers basic adherence to the IEEE 754 floating-point standard. It provides other features of the standard that historically haven't been practical for serious program use (even though supported in hardware). Also, HP-UX for IPF offers expression evaluation, elementary functions, and complex arithmetic that follow the spirit of the floating-point standard. Notable IEEE 754 related features in HP-UX for IPF include

- well-specified methods for wide expression evaluation, with auxiliary features including C type-generic math functions
- mathematical functions with special-case behavior similar to IEEE basic operations
- C/C++ complex arithmetic and functions with special-case behavior in the spirit of IEEE 754, including imaginary types for C
- correctly rounded binary-decimal conversion between each of the four floating-point formats and up to 36 decimal digits (sufficient to distinguish all quad values)
• C/C++ API for manipulating rounding modes and exception flags
• pragmas and compile options to guarantee reliable use of rounding modes and exception flags with minimal performance impact
• pragmas and compile options to disallow contractions (including FMAs), and C/C++ fma functions for controlled use of FMAs
• C/C++ NAN and INFINITY macros that can be used for compile-time initialization
• I/O for NaNs and infinities
• maintenance of the sign of zero

3.3 Special Cases

The HP-UX libm library for IPF carefully adheres to the detailed specification of special cases for math.h and complex.h functions in Annexes F and G of the C99 standard, which incorporates and extends the IEEE 754 treatment of special cases.

Cases are termed “special” because they don’t have a clearly best result. Both IEEE 754 and C99 specify that very problematic cases, e.g. 0/0, for which any result would be generally misleading, are to return NaN and raise the invalid exception. For other special cases, C99 specifies returning a numerical result, instead of NaN, if that result would be useful in some important applications, especially if the diagnostic value of returning a NaN is dubious. An example is the specification for \(0^0\), which is discussed in [12].

Careful treatment of special cases by the libm library and the compiler make program behavior more predictable, which facilitates writing robust, efficient code that handles special cases gracefully. And it makes debugging easier.

3.4 Accuracy

The HP-UX libm library for IPF has been carefully crafted to provide as much accuracy as possible commensurate with reasonably high performance. Accuracy for each elementary function is quantified by the largest rounding error (relative to the infinitely precise result) measured in units in the last place (ulp). A one ulp rounding error for float precision signifies a relative rounding error of about \(2^{-23}\) up to (slightly less than) \(2^{-22}\). Correct rounding in IEEE round-to-nearest mode implies a rounding error of at most 0.5ulp.

For the 11i V1.6 release, the libm development team set accuracy goals for commonly used elementary functions under default (IEEE round-to-nearest) rounding mode with +Ofltacc=strict or +Ofltacc=default: to make maximum rounding error no bigger than 0.502 ulp for float, double, and quad (long double) functions and 0.55 ulp for extended functions. These goals have, for the most part, been achieved.

Table 3 below displays bounds for observed errors measured in ulps for various elementary functions in 11i V1.6, with round-to-nearest rounding mode and +Ofltacc=strict or +Ofltacc=default. The measurements reflect the worst observed error for each function in millions of test cases distributed throughout the function domain.
Other *libm* functions, such as `sqrt` and functions that convert to integer formats or integral values, round correctly as specified by either IEEE 754 or the function definitions.

Future releases of the HP-UX libm library for IPF will benefit from algorithmic refinements that will enhance the accuracy of various elementary functions, especially extended ones.

With `+Ofltacc=relaxed`, the 11i V1.6 compiler may invoke faster (see 3.5) and slightly less accurate implementations for math functions. Table 4 below shows accuracy measurements with `+Ofltacc=relaxed`. Some entries - for example the float trigonometrics - are identical to the corresponding ones in Table 3 above because the V1.6 compiler does not select a special version for that precision of the function.

**Table 3: Bounds for Observed Errors in Ulps**

<table>
<thead>
<tr>
<th>Function Family</th>
<th>float</th>
<th>double</th>
<th>extended</th>
<th>quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>exponentials</td>
<td>.50004</td>
<td>.5004</td>
<td>.52</td>
<td>.502</td>
</tr>
<tr>
<td>logarithms</td>
<td>.5002</td>
<td>.501</td>
<td>.505</td>
<td>.5001</td>
</tr>
<tr>
<td>trigonometrics</td>
<td>.50003</td>
<td>.502</td>
<td>.501</td>
<td>.54</td>
</tr>
<tr>
<td>inverse trigonometrics</td>
<td>.501</td>
<td>.503</td>
<td>.7</td>
<td>.5001</td>
</tr>
<tr>
<td>hyperbolics</td>
<td>.5002</td>
<td>.502</td>
<td>.53</td>
<td>.502</td>
</tr>
<tr>
<td>inverse hyperbolics</td>
<td>.500001</td>
<td>.504</td>
<td>2.0</td>
<td>.52</td>
</tr>
<tr>
<td>cube root, hypotenuse</td>
<td>.50002</td>
<td>.501</td>
<td>.502</td>
<td>.5003</td>
</tr>
<tr>
<td>reciprocal square root</td>
<td>.5004</td>
<td>.501</td>
<td>.50001</td>
<td>.50003</td>
</tr>
<tr>
<td>powers</td>
<td>.500001</td>
<td>.5005</td>
<td>.8</td>
<td>.5001</td>
</tr>
<tr>
<td>annuity, compound</td>
<td>.500001</td>
<td>.9</td>
<td>2.5</td>
<td>.7</td>
</tr>
</tbody>
</table>

**Table 4: Bounds for Observed Errors in Ulps with +Ofltacc=relaxed**

<table>
<thead>
<tr>
<th>Function Family</th>
<th>float</th>
<th>double</th>
<th>extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>square root</td>
<td>.9</td>
<td>.50001</td>
<td>.7</td>
</tr>
<tr>
<td>exponentials</td>
<td>1.5</td>
<td>1.8</td>
<td>1.9</td>
</tr>
<tr>
<td>logarithms</td>
<td>.58</td>
<td>.65</td>
<td>2.7</td>
</tr>
<tr>
<td>trigonometrics</td>
<td>.50003</td>
<td>.502</td>
<td>3.5</td>
</tr>
<tr>
<td>inverse trigonometrics</td>
<td>1.0</td>
<td>.8</td>
<td>2.7</td>
</tr>
<tr>
<td>hyperbolics</td>
<td>.5002</td>
<td>1.8</td>
<td>3.4</td>
</tr>
<tr>
<td>cube root</td>
<td>.50002</td>
<td>.501</td>
<td>2.4</td>
</tr>
<tr>
<td>reciprocal square root</td>
<td>.7</td>
<td>1.5</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Faster versions of other functions for use under `+Ofltacc=relaxed` are expected to become available in future releases.
3.5 Speed

Latency

The performance of each function in the HP-UX *libm* library depends in part on its algorithmic complexity, argument and result precision, and the compiler optimization level under which it is called. For optimization levels below +O3 and with +Ofltacc=strict or +Ofltacc=default, most *libm* functions are invoked via call branches to the highly accurate and robust default implementations, which have been optimized for minimum latency for source arguments of usual values (not edge or special cases).

Table 5 below displays the current minimum latency of such calls for the various precisions of common elementary functions in the 11i V1.6 IC43 pre-release. The data reflect the time from each function call to when the result is available to the caller, assuming that the called function has source operand[s] immediately available for its use in the source registers specified by the Application Binary Interface (ABI). Latency units are machine cycles. The tabulated performance levels are reached when the implementations are cache-resident (both instructions and data).

<table>
<thead>
<tr>
<th>Function</th>
<th>float</th>
<th>double</th>
<th>extended</th>
<th>quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>fabs</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>sqrt</td>
<td>30</td>
<td>41</td>
<td>45</td>
<td>110</td>
</tr>
<tr>
<td>exp</td>
<td>36</td>
<td>44</td>
<td>58</td>
<td>269</td>
</tr>
<tr>
<td>log</td>
<td>33</td>
<td>36</td>
<td>65</td>
<td>221</td>
</tr>
<tr>
<td>cos</td>
<td>47</td>
<td>53</td>
<td>77</td>
<td>294</td>
</tr>
<tr>
<td>sin</td>
<td>47</td>
<td>53</td>
<td>81</td>
<td>321</td>
</tr>
<tr>
<td>tan</td>
<td>71</td>
<td>81</td>
<td>118</td>
<td>314</td>
</tr>
<tr>
<td>atan</td>
<td>49</td>
<td>62</td>
<td>74</td>
<td>321</td>
</tr>
<tr>
<td>atan2</td>
<td>62</td>
<td>72</td>
<td>88</td>
<td>331</td>
</tr>
<tr>
<td>cosh</td>
<td>52</td>
<td>49</td>
<td>73</td>
<td>244</td>
</tr>
<tr>
<td>sinh</td>
<td>49</td>
<td>61</td>
<td>69</td>
<td>271</td>
</tr>
<tr>
<td>tanh</td>
<td>60</td>
<td>80</td>
<td>98</td>
<td>361</td>
</tr>
<tr>
<td>rsqrt</td>
<td>33</td>
<td>38</td>
<td>50</td>
<td>115</td>
</tr>
<tr>
<td>pow</td>
<td>64</td>
<td>78</td>
<td>105</td>
<td>644</td>
</tr>
</tbody>
</table>

Note that the latency of 6 shown in Table 5 for *fabs* includes a 2-cycle overhead for the call. The compiler normally replaces a call to *fabs* with an inlined IPF *fmerge* instruction (provided the *math.h* header is included), so the latency for *fabs* is actually only 4.

With +Ofltacc=relaxed, the 11i V1.6 compiler may invoke faster and slightly less accurate (see 3.4) implementations for math functions. Table 6 shows latency measurements with +Ofltacc=relaxed.
Throughput with inlining

The compiler may inline the implementations of common elementary functions into the caller's code, provided the header that declares the function is included, thus enabling opportunities for performance optimization to exploit the parallelism of the IPF architecture. At all optimization levels, the 11i V1.6 compilers inline very simple functions, e.g. \texttt{fabs} and \texttt{copysign}; at +O2, they inline square root functions; and at +O3 they inline several more functions, including ones shown in Tables 7 and 8 below. If the inlined \textit{libm} function lies within a loop construct, the compiler may be able to unroll and modulo-schedule the loop body, thusaffording much greater throughput of function results than that achievable by call branches to closed function implementations. For example, an inlined \texttt{expf} function in a loop is capable of delivering a result nearly every 5 cycles on the average, as opposed to every 36 cycles if the closed routine is called. Similarly, an inlined \texttt{sqrtf} can approach throughput of a result every 5 cycles and \texttt{sqrtf} with +Ofltacc=relaxed can approach throughput of a result every 3 cycles.

In Tables 7 and 8 below, the throughput numbers (cycles per function output) are obtained from executing a loop that evaluates the function on an input vector of length 1024 and stores the outputs to another vector of the same length. The entries of the input vector are selected in such a way that any of the rare cases, e.g. invalid input arguments, or arguments at which the function overflows or underflows, or arguments of huge magnitudes for trigonometric functions, are excluded. The same loop is executed a few, say 20, times and the minimum number of cycles of a particular loop execution is taken and divided by 1024, the vector length, to arrive at cycles per function output - the throughput number. This number may not be achieved when data are not cache residents.

<table>
<thead>
<tr>
<th>Function</th>
<th>float</th>
<th>double</th>
<th>extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td>22</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>exp</td>
<td>36</td>
<td>39</td>
<td>44</td>
</tr>
<tr>
<td>log</td>
<td>30</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>cos</td>
<td>47</td>
<td>53</td>
<td>57</td>
</tr>
<tr>
<td>sin</td>
<td>47</td>
<td>53</td>
<td>57</td>
</tr>
<tr>
<td>tan</td>
<td>71</td>
<td>81</td>
<td>85</td>
</tr>
<tr>
<td>cosh</td>
<td>52</td>
<td>50</td>
<td>52</td>
</tr>
<tr>
<td>sinh</td>
<td>49</td>
<td>48</td>
<td>61</td>
</tr>
<tr>
<td>tanh</td>
<td>60</td>
<td>69</td>
<td>89</td>
</tr>
<tr>
<td>rsqrt</td>
<td>23</td>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>
It is anticipated that future releases of the HP-UX *libm* library for IPF will perform even better due to both ongoing performance tuning of implementation algorithms and advances in compiler technology.

4 Programming Techniques

The floating-point facilities of HP-UX on IPF support programming techniques for simpler, more robust code. Describing code as more robust means that it delivers useful results for a broader range of inputs. Results from floating-point calculations may not be useful because the input is too close to where the problem is mathematically ill-posed. A general rule applicable in various problem areas is

\[
\text{error} \sim \frac{\text{precision roundoff}}{\text{distance from ill-posed}}
\]

Proportionality to the precision roundoff implies that using greater precision in the calculation reduces the error.

Calculations may fail because of intermediate overflow or underflow, though the desired result would be
within range. In these cases, overflow and underflow might be avoided if intermediate calculations use
greater exponent range.

4.1 Wide types

Using a wider type for internal calculations is often an easy and effective strategy for improving robustness.
HP-UX on IPF offers two wide types, extended and quad, each with more range and precision than the
float (single) and double types typically used for input data and results. The extended type has 11
more bits of precision than double, which generally reduces the number of failures due to precision round-
off by a factor of 1000. The quad type has still 49 more bits of precision than extended. Both
extended and quad provide 4 more exponent bits than double, increasing the exponent range by a fac-
tor of 16, which is usually sufficient to eliminate overflow and underflow in calculations on float or dou-
ble inputs. The extended type is particularly attractive in this context because IPF's extended
arithmetic is as fast as float or double, and extended elementary functions in HP-UX are roughly 0.7
times as fast as corresponding double ones. Even quad, whose elementary functions average about 0.25
times the speed of extended ones, might be considered for performance sensitive applications that need
more precision than extended for a small number of operations and functions.

Wide floating types are a notable advantage for IPF over most RISC systems. RISC systems typically have
no hardware support for types wider than double. They may provide the IEEE quad type implemented in
software, or they may provide a non-IEEE, double-double type, using two doubles to represent the head and
tail of a value. The double-double type is problematic because its exponent range is no wider than double,
its precision depends on the value being represented, and regular treatment for edge cases is difficult. On the
other hand, IPF implementations have full hardware support for the extended type, and have features that
can be used to implement substantially faster IEEE quad functions than RISC systems can offer.

HP-UX C/C++ on IPF provide several features that facilitate using the wide types. First, they support
nomenclature for the various programming constructs involving extended and quad. For example,

```
cc -fpwidetypes

#include <stdio.h>
#include <math.h>
#include <float.h>
printf(“%hLe\n”, logw(EXT_MAX)); //print max extended value
printf(“%lLe\n”, logq(QUAD_MAX)); //print max quad value
```
illustrates C I/O specifiers, function suffixes, and macros for the extended and quad types. Note that use
of the nomenclature requires the -fpwidetypes option.

The HP-UX C/C++ compile option

```
-fpeval=float|double|extended
```
causes evaluation of narrower precision binary operations and floating constants to the type specified in the
option. Under the default -fpeval=float, binary operations and floating constants are evaluated to their
semantic type. Under the option -fpeval=double, binary operations and floating constants of type float
are evaluated to double. Under the option -fpeval=extended, binary operations and floating con-
stants of type float or double are evaluated to extended. The evaluation methods for
-fpeval=float and -fpeval=double adhere to the C99 specification (corresponding to
FLT_EVAL_METHOD = 0 and FLT_EVAL_METHOD = 1, respectively). The evaluation method for
-fpeval=extended follows the C99 specification for evaluation to long double, except that evalua-
tion is to extended instead of long double.

The irregular use of wide types by some compilers (e.g. coercing wide representations when registers are “spilled”) has undeservedly blemished the reputation of wide evaluation. HP-UX wide evaluation is based on the well-defined methods in C99. Results are just as predictable with wide evaluation as with any other method.

The math.h header defines the C99 types float_t and double_t, which are the types used for evaluating float and double operations and floating constants. For example, by default float_t is float and double_t is double, but under -fpeval=extended both are extended.

Also, HP-UX provides the C99 type-generic functions. When the tgmath.h header is included, an unsuffixed math function takes on a type determined by its arguments. The mechanism supports all floating types. This feature is similar to C++ overloading and Fortran intrinsics, but restricted to system library functions of floating type. For example

```c
cc -fpwidetypes

#include <tgmath.h>
extended x;
float complex z;
log(x); /* equivalent to logw(x) */
log(z); /* equivalent to clogf(z) */
```

**Example**

This example illustrates the use of wide types and their supporting features to improve robustness. The problem is to calculate \(\log(ab + cd)\) where double is sufficient to represent the inputs \(a, b, c,\) and \(d,\) and \(ab + cd\) is known analytically to be non-negative. Here's a straightforward implementation using double:

```c
#include <math.h>
double a, b, c, d, res;
double s;
...
s = a*b + c*d;
res = log(s);
```

However, this code is problematic. First, a \(\log\) function is necessarily unstable where \(a*b + c*d\) is inexact and near 1. For example, the calculation loses all significant bits if \(a*b = 1\) and \(|c*d| < 2^{-53}\), because then \(a*b + c*d\) yields 1 and \(\log\) of 1 returns 0, whereas \(\log(1 + x)\) is approximately \(x\) when \(x\) is small.

Also, \(a*b + c*d\) can lose all significant bits from cancellation if \(a*b\) and \(c*d\) are nearly negatives of each other and one or both are inexact. And, the computation of \(a*b + c*d\) can overflow or underflow the double format, though the true value of \(\log(ab + cd)\) is safely in the double range.

Simply compiling with -fpeval=extended makes the code somewhat more robust. The effect is that \(a*b + c*d\) is computed in extended, retaining 11 extra bits in the products, which greatly reduces the number of possible inputs that would cause harmful cancellation. However, the assignment to \(s\) still yields a double value. The \(\log\) arguments near 1 aren't represented with any extra precision. And, the extended sum \(a*b + c*d\) can overflow or underflow when coerced to the range of double.
The next version addresses all three robustness problems.

```
cc -fpwidenotypes -fpeval=extended ...

#include <tgmath.h>

double a, b, c, d, res;
double_t s;
...
  s = a*b + c*d;
  res = log(s);
```

Here the `-fpeval=extended` option causes `double_t` to be defined as `extended` and the inclusion of `tgmath.h` causes the reference to `log` to be type generic. Thus `s` is computed fully in `extended`, the `extended` `log` function is used, and coercion to `double` occurs only in the final assignment to `double`. With this version, the likelihood of encountering each of the two precision problems has been reduced by a factor of roughly 1000, and the risk of intermediate over/underflow has been completely eliminated.

Note that writing

```
res = log(a*b + c*d);
```

doesn’t suffice for fully wide calculation, because the semantic type of the argument, not its evaluation format, determines the type of the generic `log` function. (This specification was chosen for C99 so that enabling wide evaluation would not automatically incur the extra performance cost of wider library functions, even though calling wide functions corresponding to the evaluation format for the argument would have made wide evaluation a more effective tool for improving robustness.)

### 4.2 Tradeoff between performance and floating-point behavior

HP-UX compilers and `libm` library are designed to provide the best performance possible while delivering a very high level of overall quality, suitable for a broad spectrum of programs. Overall quality entails predictability, accuracy, and carefully treated special cases, all of which can affect the correctness and robustness of a program. Often highest quality comes with at least some performance penalty. However, many important applications need all the speed they can get and are tolerant of less rigorous floating-point behavior, while others need even stricter behavior than is guaranteed by default (which is designed for most programs). To accommodate different needs of different applications, HP-UX provides controls in the form of compile options and pragmas to allow programmers to make intelligible tradeoffs between speed and floating-point behavior if they are not satisfied with the default setting. The controls are of three sorts:

- general controls for optimization
- controls for special floating-point functionality
- controls to trade-off the strictness of floating-point behavior for speed

#### 4.2.1 General controls

General optimization controls include optimization level options (e.g. `+O3`), profiling, link options (e.g. `-Bprotected`), user assertions (e.g. `+Onoptrs_to Globals`), among others. Many of these options improve performance of floating-point code. The `+O2` option is very effective, as at that optimization level the compiler optimizes most `libm` functions (the ones without side effects) like built-in floating-point operators and it inlines square root functions. At the `+O3` optimization level the compiler inlines several key `libm`
functions, including logarithm, exponential, and trigonometric functions, which can result in dramatic speedup where the function call appears in a loop that the compiler can modulo schedule (see 3.5).

The general optimization controls have no negative effect on documented floating-point behavior. Hence, these options can be chosen based on other considerations, e.g. compile time, without concern about sacrificing correctness or robustness of floating-point code.

4.2.2 Controls for special floating-point functionality

Using floating-point modes and flags

Application code that runs in non-default floating-point modes (for example, a non-default rounding direction) or tests exception flags must either be compiled with the +Ofenvaccess option or else be under the effect of a

    #pragma STDC FENV_ACCESS ON

directive (specified in C99), for reliable behavior. Without one of these controls the compiler might undermine expected behavior by moving or removing code, for example moving the multiply in

    #include <fenv.h>
    ...
    fesetround(FE_UPWARD);
    a = b*c;

to before the fesetround call, or even to compile time if the compiler can determine the values of b and c.

For best performance, #pragma STDC FENV_ACCESS ON should be placed in the smallest blocks that enclose the code needing protection.

Controlling contractions

Application code that is not tolerant of contractions (which the compiler may synthesize by default) should be compiled with the +Oftacc=strict option or be under the effect of a

    #pragma STDC FP_CONTRACT OFF

directive (specified in C99). Contractions refer to multiple language-level operations combined into just one machine instruction. For example, in

    float r;
    double x, y, z;
    ...
    r = x*y + z;

the multiply, add, and conversion from double to float can be contracted into one IPF fma.s instruction (the .s completer indicates rounding to single). Contractions are acceptable in most applications, and generally even improve accuracy. However, they can break codes that depend on operations being rounded to specific range and precision, and rarely even ones that don’t. In the example of log(ab + cd) in 4.1, use of FMA in the calculation of a*b + c*d can yield a negative value, even though ab + cd > 0 mathematically, and thus cause an invalid exception and NaN result from the log.
On the other hand, controlled use of FMA can enable the design of much more efficient codes for some problems. FMA offers the obvious advantage of computing two operations in the time of one and with just one rounding error. Also, it is particularly effective for capturing extra bits of a calculation, as illustrated in the next example.

IPF’s \texttt{fma} instruction meets the specification for the \texttt{fma} function in C99. The \texttt{fma} function, together with the facility to disable compiler syntheses of FMA, provide the control needed for algorithm development. The compiler may generate an Floating-point Multiply Subtract instruction for \texttt{fma}(x, y, -p).

\textbf{Example}

This example uses FMA to compute $e^{xy}$, efficiently and accurately, without use of wider types. Consider the straightforward program

\begin{verbatim}
 extended x, y, r;
 r = expw(x*y);
\end{verbatim}

The relative error in the exponential function is proportional to the absolute error in the argument. The rounded results of $x*y$ can be in error by as much as $2^{-50}$. (The largest value for which \texttt{expw} won’t overflow is about $2^{14}$.) Thus, the low order 14 bits of \texttt{expw} may be corrupted because of the rounding error in $x*y$.

The product $xy$ can be written as the exact sum $high + low$, where $high$ is the computed $x*y$ and $low$ is the error, which an FMA can determine exactly: $\texttt{fmaw}(x, y, -x*y)$.

As

$$e^{low} = 1 + low + low^2/2 + \ldots, \text{ where } |low| < 2^{-50}$$

we have the approximation

$$e^{xy} = e^{high+low} = e^{high} \cdot e^{low} = e^{high} \cdot (1 + low) = e^{high} \cdot e^{high}$$

Computation of $e^{high}$ produces at most just slightly more than .5 ulp error, the multiply-add produces at most .5 ulp error, and so the following code calculates $e^{xy}$ with just slightly more than 1 ulp error.

\begin{verbatim}
 #include <math.h>
 extended x, y, r, high, low;
 high = x*y;
 low = fmaw(x, y, -high);
 rt = expw(high);
 r = fmaw(rt, low, rt);
\end{verbatim}

\textbf{Using errno}

With HP-UX for IPF, codes that depend on \texttt{math.h} functions setting \texttt{errno} must be compiled with the \texttt{+Olibmerrno} option. (See section 3.2.) Use of the \texttt{+Olibmerrno} option degrades performance of \texttt{libm} functions that may as a result of the option set \texttt{errno} or return results that differ from the C99 specification.
The IEEE exception flag mechanism functionally supersedes the `errno` mechanism for mathematical functions. Four exception flags -- invalid, divide-by-zero, overflow, and underflow -- provide finer identification of the special cases that `errno` classifies as domain or range errors. Each exception has its own flag, which is not overwritten when a different exception occurs, whereas the `errno` value is overwritten on various “error” conditions, including ones unrelated to floating-point. Exceptions are raised consistently by both mathematical functions and also built-in floating-point operations, whereas only library functions set `errno`. The hardware exceptions flags can be accessed with less restriction on optimization than can the global `errno` variable. The `+O[no]libmerrno` option does not affect `errno` setting by non-math functions.

### 4.2.3 Controls for tradeoff between speed and floating-point behavior

Other controls, principally the `+Ofltacc=strict|default|limited|relaxed` compile option, are designed specifically to allow tradeoff between performance and floating-point behavior. Such options should be considered for performance hungry applications that are known to be tolerant of a less rigorous floating-point model or can be thoroughly tested.

#### Value-changing optimization

`+Ofltacc=default` (the default) disallows optimizations that might change result values from what the language specifies, except that it allows contractions (including FMA synthesis). This option balances speed and behavior considerations and is suitable for most applications.

`+Ofltacc=strict` disallows all optimizations that might change result values, including contractions (notably FMA synthesis). This option provides the most predictable floating-point behavior, but loses the performance benefits of contractions, which can be considerable for some applications.

`+Ofltacc=limited` is equivalent to the default except that infinities, NaNs, and the sign of zero may not be treated as specified. This option might provide some minor speedup for applications that do not depend on special values. Note that the specified treatment of special values may improve robustness and facilitate debugging even if the program is not specifically coded with special values in mind.

`+Ofltacc=relaxed` allows code transformations based on mathematical identities for real numbers, even if the transformations may change result values. With this option, the compiler may invoke slightly less accurate versions of mathematical functions (see 3.4 and 3.5) and may inline math function sequences that yield slightly different result values than the closed call counterparts. This option delivers the best performance.

#### Flush-to-zero vs. gradual underflow

The `+FPD` option (for a compile command that produces an executable) causes program startup to install flush-to-zero underflow mode, instead of the default gradual underflow mode. In flush-to-zero mode, floating-point instructions return zero in lieu of non-zero results with magnitude less than $2^{emin}$. The IEEE 754 floating-point standard prescribes gradual underflow as a way of filling the extraordinarily wide gap in floating-point numbers around zero which one sees with flush-to-zero:
Gradual underflow:

\[\begin{array}{cccccc}
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
-2^{emin+1} & -2^{emin} & 0 & 2^{emin} & 2^{emin+1} & \ldots
\end{array}\]

Flush-to-zero:

\[\begin{array}{cccccc}
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
-2^{emin+1} & -2^{emin} & 0 & 2^{emin} & 2^{emin+1} & \ldots
\end{array}\]

(Imagine \(2^{\text{precision}-1}\) points between consecutive powers of 2.) Gradual underflow assures that \(x = y\) if and only if \(x - y = 0\) (for finite \(x\) and \(y\)). With flush-to-zero, code like

\[
\text{if } (x != y) \text{ r = a / (x - y);}
\]

might raise a divide-by-zero or invalid exception. Some codes are more robust when run with gradual underflow. If tiny values will not be encountered or not be significant in the calculation or logic, then flush-to-zero is fine.

Use of flush-to-zero mode might significantly improve performance of single precision calculations; it is much less likely to speed up calculations in wider precision.

**Limited range for complex operations**

The compile option **+Ocxlimitedrange** and the C99 directive

```c
#pragma STDC CX_LIMITED_RANGE ON
```

allow trading off certain behaviors of the complex multiply and divide operations and `cabs` functions for performance. The HP-UX complex multiply and divide operations and `cabs` functions are designed to avoid spurious over/underflow and underflow and to honor the infinity properties specified in C99, e.g.

\[
f\text{inite nonzero} / \text{finite} = \text{zero}
\]

(See section 4.3.) Straightforward implementations using the usual textbook formulas

\[
(a+bi)(c+di) = (ac-bd) + (bc+ad)i
\]

\[
(a+bi)/(c+di) = ((ac+bd) + (bc-ad)i) / (c^2 + d^2)
\]

\[
|a+bi| = \sqrt{a^2 + b^2}
\]

are prone to intermediate over/underflow and tend to return NaN results rather than potentially useful values. The \(a^2 + b^2\) in the formula for the complex absolute value overflows if the exponent of either \(a\) or \(b\) is more than half the maximum exponent for the evaluation type. Using the division formula, \((1+0i) / (\infty+\infty i)\) yields NaN+NaN \(i\), whereas a zero would be better.

The **+Ocxlimitedrange** compile option and the **CX_LIMITED_RANGE** pragma allow the compiler to deploy faster code for complex multiply, divide, and absolute value, provided only that it be at least as robust as code using the textbook formulas. With HP-UX 11i V1.6, these controls can significantly speed up application code that contains complex multiply, divide, or absolute value in modulo scheduled loops. The per-
formance benefits are less obvious outside of modulo scheduled loops because the implementation uses just one IPF fclass instruction to determine whether extra computation is needed for the infinity properties. In HP-UX 11i V1.6, these performance controls don't cause intermediate over/underflow for float and double complex, because the implementation uses wide evaluation internally.

Package options

HP-UX offers package options that imply various specific performance options.

The +Ofast (or -fast) option implies the +Ofltacc=relaxed and +FPD options, among others not specific to floating point.

The +Oaggressive option implies the +Ofltacc=relaxed option, among others not specific to floating point.

4.3 Complex arithmetic in C

C99 introduces complex arithmetic into the C language, specifying:

- complex types: float complex, double complex, long double complex (HP-UX also provides extended complex and quad complex)
- imaginary types (stored like corresponding real types): float imaginary, double imaginary, long double imaginary (HP also provides extended imaginary and quad imaginary)
- an imaginary unit: I
- efficient promotion rules for binary operations containing a mix of real, complex, and imaginary operands
- infinity properties for basic operations, and a related performance pragma (see 4.2.3)
- a header complex.h declaring complex functions (see 3.2)
- special cases in the style of IEEE 754

C99 promotion rules for binary operations do not require that real or imaginary operands be converted to complex. For example, the expression

\[ a(x + y i) + b i \]

is evaluated as

\[ (a x + a y i) + b i = ax + (ay + b) i \]

rather than as

\[ (a + 0i)(x + yi) + (0 + bi) = ... = ((ax - 0y) + 0) + ((ay + 0x) + b)i. \]

An optimizer might remove operations involving unwanted zeros introduced by more traditional promotion rules; however, such optimizations can yield unexpected result values, e.g. 0y -- 0 is problematic because 0y is NaN if y is infinite or NaN and is -0 if y is negative and finite.
The infinity properties are the following: For \( z \) nonzero and finite,
\[
\begin{align*}
\infty \ast z &= \infty, \\
\infty \ast \infty &= \infty, \\
\infty / z &= \infty, \\
\infty / 0 &= \infty, \\
z / \infty &= 0, \\
0 / \infty &= 0, \\
z / 0 &= \infty.
\end{align*}
\]

A complex value is regarded as an infinity if either the real or the imaginary part or both are infinite (this includes a complex value for which one part is infinite and the other part is a NaN). This specification preserves the information that the magnitude is infinite, even when the direction isn't meaningful. The \texttt{cabs} function is consistent with the definition of complex infinities in that \texttt{cabs} of any complex infinity, e.g. \texttt{cabs(NAN + INFINITY*I)}, returns positive infinity.

HP-UX for IPF supports the C99 specification, including the portion in Annex G (IEC 60559-compatible complex arithmetic). C99 features allow for more natural and efficient codes than is possible with more traditional complex arithmetic facilities. The following simple example illustrates.

**Example**

Let \( z = x + y \, i \), where \( x \) and \( y \) are real, compute
\[
w = (z - i) / (z + 2i) \text{ if } (z - i) / (z + 2i) \text{ lies inside the unit circle; } \\
w = 0 \text{ otherwise.}
\]

The code might contain
\[
#include <complex.h>
\double x, y;
\double complex z, w;
... \\
z = x + y*I;
w = (z - I) / (z + 2*I);
if (cabs(w) > 1) w = 0;
\]

A complex value is constructed from the real variables \( x \) and \( y \) by means of the natural notation \( x + y \, i \), which the compiler implements without the use of actual floating-point instructions. The quotient \( (z - I) / (z + 2*I) \) too is expressed in a mathematical style notation. Because \( I \) has imaginary type and the promotion rules do not require real and imaginary operands to be converted to complex, the numerator \( z - I \) and denominator \( z + 2*I \) each is evaluated with just one real floating-point instruction. In contrast, more traditional promotion rules require that the product \( 2*I \) be promoted to \( (2+0i)(0+i) \), implying four real multiplies and two adds. Thus C99 features for complex arithmetic allow natural, mathematical style notation and entail built-in execution time efficiency.

C99 infinity properties guarantee that evaluation of the quotient \( (z - I) / (z + 2*I) \) at the pole \( z = -2i \) will yield a complex infinity. Without the infinity property, \texttt{nonzero / zero = infinite}, the quotient in this case might yield \texttt{NaN}+\texttt{NaN} \( i \) (and would, if computed with the ordinary division formula), and then \texttt{cabs(w)} > 1 would be false (and would raise the invalid exception), and the result from the code above would be incorrect. Without the infinity property, extra code would have to be added to test for \( z = -2i \) and handle that case separately. Thus C99 features for complex arithmetic facilitate use of simpler algorithms.
Bibliography

1. ANSI X3.159-1989, Programming Language C (C89).


   http://developer.intel.com/technology/itj/q41999/articles/art_5.htm

   http://docs.hp.com/hpux/onlinedocs/B3901-90003/B3901-90003.html

8. HP web page: Intel® Itanium® processor family architecture.


    http://www.intel.com/design/itanium/manuals/index.htm

    http://anubis.dkuug.dk/jtc1/sc22/wg14/


    http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf
