Semantics of Floating Point Math in GCC

Placeholder for semantical information on GCC math optimizations. To be completed and renamed. Eventually this should end up in the GCC documentation.

NOTE: This page is in draft status, but feel free to add/modify.

The focus in this page is providing information on what semantics GCC provides for numerical computation. Mostly the information provided is applicable to all supported languages. Sections with more information tailored to specific front ends may be added later.

Here is the summary of the general policy of floating point arithmetic in GCC BOF (2007 GCC Summit) - FP_BOF

Currently GCC supports three main levels of IEEE 754 compliance:

Compliance	Compiler options	Description
Full	`-frounding-math -fsignaling-nans`	Support infinities, `NaN`s, gradual underflow, signed zeros, exception flags and traps, setting rounding modes. Compare with C99's `#pragma STDC FENV ACCESS ON`.
Default		Without any explicit options, GCC assumes round to nearest or even and does not care about signalling `NaN`s. Compare with C99's `#pragma STDC FENV ACCESS OFF`. Also, see note on x86 and m68080.
Relaxed	`-funsafe-math-optimizations`	This mode enables optimizations that allow arbitrary reassociations and transformations with no accuracy guarantees. It also does not try to preserve the sign of zeros.

In addition GCC offers the -ffast-math flag which is a shortcut for several options, presenting the least conforming but fastest math mode. It enables `-fno-trapping-math`, `-funsafe-math-optimizations`, `-ffinite-math-only`, `-fno-errno-math`, `-fno-signaling-nans`, `-fno-rounding-math`, `-fcx-limited-range` and `-fno-signed-zeros`. Each of these flags violates IEEE in a different way. `-ffast-math` also may disable some features of the hardware IEEE implementation such as the support for denormals or flush-to-zero behavior. An example for such a case is x86_64 with it's use of SSE and SSE2 units for floating point math. The flags under -ffast-math can be divided into four catagories according to the different aspect of IEEE they violate; as described in the following table:

Category	**Flags**	**Comments**
Trap handlers and exceptions	-fno-trapping-math, -fno-signaling-nans	IEEE standard recommends that implementations allow trap handlers to handle exceptions like divide by zero and overflow. This flags assumes that no use-visible trap will happen.
Rounding	-fno-rounding-math	IEEE has four rounding modes. This flag assumes that the rounding mode is round to nearest.
Languages and compilers	-funsafe-math-optimizations	Due to roundoff errors the associative law of algebra do not necessary hold for floating point numbers and thus expressions like (x + y) + z are not necessary equal to x + (y + z).
Special quantities (Nans, signed zeros and infinity)	-ffinite-math-only, -fno-signed-zeros	

These flags allow the compiler to assume certain features are not used, without allowing optimizations that affect accuracy. The properties below also depend on the floating-point type. For example, if a target has a single precision floating-point representation that does not support signalling `NaN` values, the property `HONOR_SNANS` would be false regardless of any compiler options and the compiler can perform transformations that would otherwise be invalid.

Property	Flags allowing violation	Comment
`HONOR_SNANS`	`-fno-signaling-nans`	This is default
`HONOR_SIGN_DEPENDENT_ROUNDING`	`-fno-rounding-math`	
`HONOR_NANS`	`-ffinite-math-only`	
`HONOR_INFINITIES`	`-ffinite-math-only`	
`HONOR_SIGNED_ZEROS`	`-fno-signed-zeros`	

Transformations

The table below summarizes the most common transformations that affect floating-point semantics. The table uses (`MODE`) to indicate a typecast to a type with that mode. Variable names that are mode names refer to variables of a type with that mode. Variables starting with a capital C refer to constants known at compile time.

Compliance	Original	Replacement	Violates	Comment
Full	`x / C`	`x * (1.0 / C)`	*none*	When C is a power of two and `1.0 / C` does not overflow.

Default	`0.0 - x`	`-x`	`HONOR_SIGNED_ZEROS`	`0.0 - 0.0 = +0.0`, not `-0.0`, unless rounding to -Inf
	`x - 0.0`	`x`	`HONOR_SIGNED_ZEROS`	`0.0 - 0.0 = -0.0`, when rounding to -Inf
	`0.0 / x`	`0.0`	`HONOR_SIGNED_ZEROS` and `HONOR_NANS`	
	`x / 1.0`	`x`	`HONOR_SNANS`	
	`x / -1.0`	`-x`	`HONOR_SNANS`	
	`-(a / b)`	`a / -b`	`HONOR_SIGN_DEPENDENT_ROUNDING`	
	`-(a / b)`	`-a / b`	`HONOR_SIGN_DEPENDENT_ROUNDING`	
Relaxed	`(SF) ((DF) xf)`	`(SF) xf`	*any*	
	`-(a - b)`	`b - a`		Should be `!HONOR_SIGNED_ZEROS && !HONOR_SIGN_DEPENDEND_ROUNDING`?
	`x - x`	`0.0`		Should be `!HONOR_NANS`?
	`-(a + b)`	`(-a) - b`		
	`-(a + b)`	`(-b) - a`		
	`x / C`	`x * (1.0 / C)`		
	`(a * c) + (b * c)`	`(a + b) * c`		
	`(a + b) * c`	`(a * c) + (b * c)`		
	`a * (b / c)`	`(a * b) / c`		May change under/overflow behavior
	`a * (b * c)`	`(a * b) * c`		
	`a + (b + c)`	`(a + b) + c`		Will typically change error of summation.
	`x + C1 == C2`	`x == C2 - C1`		Different rounding makes equality test useless
	`x + C1 != C2`	`x != C2 - C1`		Different rounding makes inequality test useless

Further floating-point related flags

`-fsingle-precision-constant` causes floating-point constants to be loaded in single precision even when this is not exact. This avoids promoting operations on single precision variables to double precision like in `x + 1.0/3.0`. Note that this also uses single precision constants in operations on double precision variables. This can improve performance due to less memory traffic.

`-fcx-limited-range` causes the range reduction step to be omitted when performing complex division. This uses `a / b = ((ar*br + ai*bi)/t) + i((ai*br - ar*bi)/t)` with `t = br*br + bi*bi` and might not work well on arbitrary ranges of the inputs.

`-fno-errno-math` disables setting of the `errno` variable as required by C89/C99 on calling math library routines. For Fortran this is the default.

Note on built-in math functions

While traditionally all mathematical functions would reside in an external library (such as `libm`), GCC may replace calls to certain functions by calls to built-in versions or may evaluate them at compile time (using MPFR) in some circumstances. Functions are only evaluated at compile time if the rounding mode is known to be round to even (`-fno-rounding-math`), or if the result is exact and does not depend on the rounding mode. Compile-time evaluation always rounds correctly, even for transcendental functions. However, as library functions may not be correctly rounded for all arguments, function results may differ depending on wether they are evaluated at compile time or at run time.

Built-in functions have names such as `__builtin_sqrt`. What built-in functions are used by GCC depends on the mode. By default, built-in functions are only used when they provide the semantics guaranteed by the C99 standard. However, with `-funsafe-math-optimizations`, substituted functions may have less precision or be restricted to a smaller domain.

Use the option `-fno-builtin-somefun` to avoid `somefun` to be replaced by a built-in version. The option `-fno-builtin` prevents all autmatic use of built-in functions, including non-mathematical ones.

Note on x86 and m68080 floating-point math

For legacy x86 processors without SSE2 support, and for m68080 processors, GCC is only able to fully comply with IEEE 754 semantics for the IEEE double extended (`long double`) type. Operations on IEEE double precision and IEEE single precision values are performed using double extended precision. In order to have these operations rounded correctly, GCC would have to save the FPU control and status words, enable rounding to 24 or 53 mantissa bits and then restore the FPU state. This would be far too expensive.

The extra intermediate precision and range may cause flags not be set or traps not be raised. Also, for double precision, double rounding may affect the final results. Whether or not intermediate results are rounded to double precision or extended precision depends on optimizations being able to keep values in floating-point registers. The option `-ffloat-store` prevents GCC from storing floating-point results in registers. While this avoids the indeterministic behavior just described (at great cost), it does not prevent accuracy loss due to double rounding.

On more modern x86 processors that support SSE2, specifying the compiler options `-mfpmath=sse -msse2` ensures all `float` and `double` operations are performed in SSE registers and correctly rounded. These options do *not* affect the ABI and should therefore be used whenever possible for predictable numerical results. For x86 targets that may not support SSE2, and for m68080 processors, use `long double` where exact rounding is required and explicitly convert to `float` or `double` when necessary. Using `long double` prevents loss of 80-bit accuracy when values must be spilled to memory. See x87note for further details.

For more information on mixing x87 and SSE math, see Math_Optimization_Flags.

Known Math Bugs

- Implicit use of extended precision on x86 when using the x87 FPU, see x87note
- `-fsignaling-nans` is still experimental and may not disable all optimizations that affect signaling NaN behavior.