# Floating Point in LLVM: the Good, the Bad, and the Absent

Joshua Cranmer

# What are the LLVM FP semantics?

# IEEE-754*

*Except for

NaN handling, sNaN suppo
denormal support, f
errno, ma
nding mode,

when you need to ask

# IEEE 754 is insufficient

- Not all types are IEEE-754 compliant
- IEEE 754 is underdefined at times (e.g., NaN payload)
- Hardware often deviates from the standard
  - Denormal flushing, lower precision operations
  - Some deviations are extensions, some are not
- Shared, hidden, mostly unused global state bad for optimization
- Users sometime want speed over correctness (-ffast-math)

# Hardware constraints

# Hardware: denormal flushing

- Usually expressed as two independent flags:
  - DAZ: Input denormals are treated as (signed) zero
  - FTZ: Tiny outputs are flushed to (signed) zero
- Note: tiny output isn't the same as denormal output
  - Different hardware has different definitions of "tiny"!
- Some hardware allows denormal flushing on per-type basis
- Some hardware *mandates* denormal flushing
- `-ffast-math` sometimes enables process-wide denormal flushing

# Hardware: x87 FPU

- Only supports `x86_fp80` types in operation
- Supports `load` + `fpext`/`fptrunc` + `store` for `float`, `double`
  - This sequence implies quieting of sNaNs in both directions
- `x86_fp80` is not sufficient to emulate `double` arithmetic
  - Induces double rounding
- Additional exotic bit: precision control
  - If set to the appropriate value, can be used to emulate `double` arithmetic
- Current lowering for x87 implementation is known to be incorrect

# Hardware: FP environment

- IEEE 754 defines rounding modes, exceptions
  - Underflow exception has two possible different implementations
  - Default exception behavior is to set sticky bits
- Some hardware provides extra exceptions
- Most hardware provides optional trapping on exceptions
- Denormal handling bits
- Exotic bits
  - MIPS NAN2008: controls layout of sNaN/qNaN
  - X87 Precision Control: adjusts precision (but not range) of operations
  - Arm Default NaN: disables NaN payload propagation

# Hardware: operations

- IEEE 754 defines a suite of core arithmetic operations
  - +, -, *, /, sqrt, fma, conversions, comparisons nearly universal in hardware
- IEEE 754 also has a suite of additional arithmetic operations
  - Examples: exp, pow, atan
  - Occasionally implemented in hardware (at possibly low precision)
- Non-IEEE 754 functions can sometimes be found in hardware
  - Complex multiply, dot product, etc.
- Low precision operations (reciprocal, rsqrt)
- Static rounding mode operations increasingly prevalent

# LLVM Semantics

# The good parts of floating-point semantics

- LLVM's floating-point types have well-defined formats
- Non-floating-point operations have well-specified behavior
  - load, store, phi, etc. do not change the bit pattern of values
  - fabs, fneg, fcopysign effectively integer operations on the sign bits
- NaN payload behavior now specified (~weak propagation)
- sNaN not guaranteed to be quieted
- *Usually* FP values are precisely that guaranteed by IEEE 754
  - Optimizations can't change precision by default
  - Excess precision (x87 behavior) not allowed

# LLVM and the FP environment

- LLVM assumes the FP environment is unused
  - All control bits are assumed to be default values, UB if not
  - Exception sticky bits are unspecified values
  - Approximately `#pragma STDC FENV_ACCESS OFF` rules
- This allows FP operations to be speculated or removed freely
- The `strictfp` attribute on function overrides this assumption
  - … but what are the semantics of instructions in this?
- Denormal flushing instead uses `denormal-fp-math` attribute
  - … but this isn't consistently correctly set by frontends
  - Can't describe environment changes mid-function

# Constrained intrinsics

- Design goal is to strongly disable optimizations by default
- Replace FP operations in `strictfp` functions
- Has hints for expected rounding mode, exception behavior
- Can’t mix constrained intrinsics and regular FP instructions

```
%val = call half @llvm.experimental.constrained.fadd.f16(
    half %x, half %y,
    metadata !"round.tonearest",
    metadata !"fpexcept.maytrap")
```

# Constrained intrinsics – failed experiment?

- They're experimental… has the experiment failed?
- Requires duplication of every intrinsic
  - We don't have duplicates for target-specific intrinsics
  - Combinatorial explosion when vector predication intrinsics exist
- Also requires duplication of every pattern that can match
- Inflexible for extending to more FP environment bits
- Tentative consensus that we want to use operand bundles

```
%val = call half @llvm.log.f16(half %x)
   ["fpe.except"(i32 0), "fpe.round"(i32 1)]
```

# Math library functions

- Main set of library functions ultimately derived from IEEE 754
- IEEE 754 requires they be correctly rounded; C doesn't
  - In practice, few libraries correctly round
  - Most libraries will return different results for same inputs
- In LLVM, can be C name (`logf`) or LLVM intrinsic (`llvm.log.f32`)
  - Recall that C `long double` maps to potentially 1 of 4 LLVM types
- Intrinsics defined as equivalent to libm without exceptions/errno

# Math library functions – issues

- We happily constant fold intrinsics, which changes results
- Codegen often lowers intrinsics to libm, which may set errno!
- Patterns only match either the C or the LLVM functions
- When are we allowed to apply mathematical identities?
  - Can we say `sin(-x) == -sin(x)`?
  - Can we say `sin(0.0) == 0.0`?
  - Can we say `sin(x) / cos(x) == tan(x)` under `-ffast-math`? Which flags?

# Fast-math

- Mostly specified via orthogonal flags on instructions
  - Except floating-point conversions
- Also via function attributes (mostly used by codegen)
- Slowly working my way through better semantics of existing flags
- We need more flag bits (`reassoc` enables too much)
- Existing optimizations don’t follow rules

# LLVM FP Semantics Scorecard

| Feature | Grade | Issues |
|---|---|---|
| FP types | A | |
| Value semantics | A- | x87 implementation is buggy; silent on noncanonicals |
| Denormal flushing | B- | `denormal-fp-math` unreliable |
| FP environment | C | See slide on constrained intrinsics |
| Math library | C | See slide on math library |
| Fast-math | C- | Incorrect optimizations, insufficient flags, poorly specified |
| Low-precision approximations | D | Not really implemented |
| Static rounding mode operations | F | Not implemented |

# Backup

# LLVM floating point types

- `half`, `float`, `double`, `fp128`: IEEE-754 format types
- `bfloat`: fully describable with IEEE-754 format parameters
- `x86_fp80`: integer bit is explicit instead of implicit
  - But generally required by the other 79 bits to be a particular value!
  - Introduces noncanonical value representations
- `ppc_fp128`: pair of IEEE-754 double values
  - Cannot be directly modeled with IEEE-754 semantics

# Noncanonical values

- Noncanonical value is an alternative representation for a value
- `x86_fp80`, `ppc_fp128`, IEEE 754 decimal FP types have them
- Denormal flushing ~ denormals are noncanonical zeros
- LLVM's semantics say nothing about them
- Best behavior probably like sNaN:
  don't guarantee that regular operations canonicalize values
- Leverage `llvm.canonicalize` for guaranteed canonicalization