Floating-Point Types in MLIR: Infrastructure, New Types and Dialect Design

Matthias Springer (NVIDIA)

EuroLLVM 2026 – April 14, 2026

Outline

1. IEEE-754 / OCP floating-point types in LLVM and MLIR
2. MLIR dialect design for HW-specific floating-point types / arithmetics
3. Software emulation of unsupported floating-point types

Floating-Point Formats and Types

Floating-Point Bit Format according to IEEE-754

half (“binary16”): 1 bit sign, 5 bits exponent (bias=15), 10 bits mantissa

single (“binary32”): 1 bit sign, 8 bits exponent (bias=127), 23 bits mantissa

double (“binary64”): 1 bit sign, 11 bits exponent (bias=1023), 52 bits mantissa

$$(-1)^{sign} * 2^{EE...E - bias} * 1.MM...M$$

$$bias = 2^{num_exp_bits - 1} - 1$$

Special numbers:

- Exponent all 0 and zero mantissa: zero (pos/neg)
- Exponent all 1 and zero mantissa: infinite (pos/neg)
- Exponent all 1 and non-zero mantissa: NaN
- Exponent all 0 and non-zero mantissa: denormals

Floating-Point Bit Format according to IEEE-754

half (“binary16”): 1 bit sign, 5 bits exponent (bias=15), 10 bits mantissa

= -0.0

single (“binary32”): 1 bit sign, 8 bits exponent (bias=127), 23 bits mantissa

double (“binary64”): 1 bit sign, 11 bits exponent (bias=1023), 52 bits mantissa

$(-1)^{sign} * 2^{EE...E - bias} * 1.MM...M$

$bias = 2^{num_exp_bits - 1} - 1$

Special numbers:

- **Exponent all 0 and zero mantissa: zero (pos/neg)**
- Exponent all 1 and zero mantissa: infinite (pos/neg)
- Exponent all 1 and non-zero mantissa: NaN
- Exponent all 0 and non-zero mantissa: denormals

Floating-Point Bit Format according to IEEE-754

half (“binary16”): 1 bit sign, 5 bits exponent (bias=15), 10 bits mantissa

1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0

= -inf

single (“binary32”): 1 bit sign, 8 bits exponent (bias=127), 23 bits mantissa

double (“binary64”): 1 bit sign, 11 bits exponent (bias=1023), 52 bits mantissa

$(-1)^{sign} * 2^{EE...E - bias} * 1.MM...M$

$bias = 2^{num_exp_bits - 1} - 1$

Special numbers:

- Exponent all 0 and zero mantissa: zero (pos/neg)
- **Exponent all 1 and zero mantissa: infinite (pos/neg)**
- Exponent all 1 and non-zero mantissa: NaN
- Exponent all 0 and non-zero mantissa: denormals

Floating-Point Bit Format according to IEEE-754

half (“binary16”): 1 bit sign, 5 bits exponent (bias=15), 10 bits mantissa

1	1	1	1	1	1	0	0	0	0	1	0	0	1	0	0

= NaN

single (“binary32”): 1 bit sign, 8 bits exponent (bias=127), 23 bits mantissa

double (“binary64”): 1 bit sign, 11 bits exponent (bias=1023), 52 bits mantissa

$$(-1)^{sign} \; * \; 2^{EE\ldots E \text{ - } bias} \; * \; 1.MM\ldots M$$

$$bias = 2^{num_exp_bits - 1} - 1$$

Special numbers:

- Exponent all 0 and zero mantissa: zero (pos/neg)
- Exponent all 1 and zero mantissa: infinite (pos/neg)
- **Exponent all 1 and non-zero mantissa: NaN**
- Exponent all 0 and non-zero mantissa: denormals

Floating-Point Bit Format according to IEEE-754

half ("binary16"): 1 bit sign, 5 bits exponent (bias=15), 10 bits mantissa

`1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0` = -1 * 2^-14 * 0.25

single ("binary32"): 1 bit sign, 8 bits exponent (bias=127), 23 bits mantissa

double ("binary64"): 1 bit sign, 11 bits exponent (bias=1023), 52 bits mantissa

~~$(-1)^{sign} * 2^{EE...E - bias} * 1.MM...M$~~

$(-1)^{sign} * 2^{1-bias} * 0.MM...M)$

$bias = 2^{num_exp_bits - 1} - 1$

Special numbers:

- Exponent all 0 and zero mantissa: zero (pos/neg)
- Exponent all 1 and zero mantissa: infinite (pos/neg)
- Exponent all 1 and non-zero mantissa: NaN
- **Exponent all 0 and non-zero mantissa: denormals**

Special rules are expensive in hardware!

Low-Precision Floating-Point Types (Examples)

E5M2: 1 bit sign, 5 bits exponent (bias=15), 2 bits mantissa

E4M3FN: 1 bit sign, 4 bits exponent (bias=7), 3 bits mantissa, no inf., 2 NaN patterns

E4M3FNUZ: 1 bit sign, 4 bits exponent (bias=8), 3 bits mantissa, no inf., unsigned zero, single NaN pattern (10000000)

E8M0FNU: 0 bit sign (positive-only), 8 bits exponent (bias=127), 0 bits mantissa, no inf., single NaN pattern (all 1), no denormals

F: Finite, **N**: Special NaN
U: Unsigned, **UZ**: Unsigned Zero
Name does not fully specify the type.
E.g., bias is not encoded.

$$(-1)^{\text{sign}} * 2^{\text{EE...E - bias}} * 1.\text{MM...M}$$

bias = *(depends on the type)*

Special numbers: Support and bit encoding depends on the type.

See OCP 8-bit Floating Point Specification, OCP Microscaling Formats (MX) Specification and HW whitepapers for details.

IEEE-754 Floating-Point Arithmetics

- Special values: `+0.0`, `-0.0`, `sNaN`, `qNaN`, `+inf`, `-inf`, denormals
- There are computation rules for specials values. Some examples:
  - `1.2 / +0.0 = inf`
  - `1.2 / -0.0 = -inf`
  - `+0.0 == -0.0`
  - `+0.0 / +0.0 = nan`
  - `nan ≠ nan`
- Not associative or distributive. E.g.: `a + (b + c) ≠ (a + b) + c`.
- Values can often not be represented exactly and rounding is needed:
  NearestTiesToEven, TowardPositive, TowardNegative, TowardZero, NearestTiesToAway.
- Status: invalid, div by zero, overflow, underflow, inexact

`APFloat`: A Software Implementation of FP Arithmetics

```
APFloat myFloat(2.7f);
llvm::errs() << "float1 = " << myFloat << "\n";
bool losesInfo;
myFloat.convert(APFloat::Float4E2M1FN(), APFloat::rmNearestTiesToEven, &losesInfo);
llvm::errs() << "float2 = " << myFloat << "\n";
APFloat::opStatus status = myFloat.add(myFloat, APFloat::rmNearestTiesToEven);
llvm::errs() << "float2 + float2 = " << myFloat << "\n";

// float1 = 2.70000005
// float2 = 3
// float2 + float2 = 6
```

APFloat: A Software Implementation of FP Arithmetics

normal, inf, nan, zero

unbiased exponent

Mantissa including leading 1

IEEEFloat

- unsigned int sign : 1
- fltCategory category : 3
- int32_t exponent
- union { uint64_t; uint64_t * } significand
- fltSemantics *semantics

+ opStatus **add**(const IEEEFloat &, roundingMode)
+ opStatus **subtract**(const IEEEFloat &, roundingMode)
...
+ void **initFromHalfAPInt**(const APInt &api)
+ void **initFromFloat8E5M2APInt**(const APInt &api)
+ void **initFromFloat8E4M3APInt**(const APInt &api)
...

APFloat

- const fltSemantics *sem
- union(IEEEFloat, DoubleAPFloat) storage

+ opStatus **add**(const APFloat &, roundingMode)
+ opStatus **subtract**(const APFloat &, roundingMode)
+ opStatus **multiply**(const APFloat &, roundingMode)
+ opStatus **divide**(const APFloat &, roundingMode)
+ opStatus **remainder**(const APFloat &)
+ opStatus **mod**(const APFloat &)
+ opStatus **fusedMultiplyAdd**(const APFloat &, const APFloat &roundingMode)
+ opStatus **roundToIntegral**(roundingMode)
+ void **changeSign**()
+ void **copySign**(const APFloat &)
+ APFloat **makeQuiet**()
+ opStatus **convert**(const fltSemantics &, roundingMode, bool *losesInfo)
+ double **convertToDouble**()
+ opStatus **convertToInteger**(APSInt &Result, roundingMode, bool *IsExact)
...

`APFloat`: A Software Implementation of FP Arithmetics

fltSemantics
- ExponentType maxExponent - ExponentType minExponent - unsigned precision - unsigned sizeInBits - fltNonfiniteBehavior nonFiniteBehavior - fltNanEncoding nanEncoding - bool hasZero - bool hasSignedRepr - bool hasSignBitInMSB

fltSemantics &**IEEEhalf**()
fltSemantics &**BFloat**()
fltSemantics &**IEEEsingle**()
fltSemantics &**IEEEdouble**()
fltSemantics &**IEEEquad**()
fltSemantics &**PPCDoubleDouble**()
fltSemantics &**PPCDoubleDoubleLegacy**()
fltSemantics &**Float8E5M2**()
fltSemantics &**Float8E5M2FNUZ**()
fltSemantics &**Float8E4M3**()
fltSemantics &**Float8E4M3FN**()
fltSemantics &**Float8E4M3FNUZ**()
fltSemantics &**Float8E4M3B11FNUZ**()
fltSemantics &**Float8E3M4**()
fltSemantics &**FloatTF32**()
fltSemantics &**Float8E8M0FNU**()
fltSemantics &**Float6E3M2FN**()
fltSemantics &**Float6E2M3FN**()
fltSemantics &**Float4E2M1FN**()
fltSemantics &**x87DoubleExtended**()

FP arithmetics (such as `APFloat::add`) is implemented based on `fltSemantics`. In some places, there are hard-coded checks / specialized paths for certain semantics (e.g., `IEEEFloat::initFrom…APInt`).

Floating-Point Types in MLIR

FloatType
+ llvm::fltSemantics &**getSemantics**() + FloatType **scaleElementBitwidth**() + unsigned **getWidth**() + unsigned **getFPManitssaWidth**()

- `APFloat`: Software implementation of a floating-point value.
- MLIR FP Type: Implements the `FloatTypeInterface` (C++ `FloatType`).
- `FloatAttr`: A wrapper around an `APFloat` and a `FloatType`.
  - Parsing/printing of textual IR (such as "`2.7 : f4E2M1FN`") via `APFloat`: first parse a float literal as C++ `double`, then convert it to the type-specific bit pattern via `APFloat`.
- Operations with FP semantics
  - Constant folding based on `APFloat`. E.g., `arith.mulf(a, b) → a*b`.
  - Most operations operate on `FloatType`, as opposed to hard-coded types. E.g.: `arith.addf`, `arith.extf`, `math.cos`, `vector.contract`, `linalg.matmul`, (`llvm.fadd`)

Adding new FP Types to MLIR

1. Define `fltSemantics` in `APFloat.h` / `APFloat.cpp` (should be improved).
2. Define a new MLIR type that implements the `FloatType` interface.
   `YourType::getSemantics()` returns the new `fltSemantics`.

If the new type is "builtin":

3. Define type constraint in `CommonTypeConstraints.td` (e.g., "`def F8E4M3`").
4. Add new token and type parser/printer rule (e.g., `Token::kw_f8E8M0FNU`).
5. Add CAPI and Python bindings.
6. Add type conversion rule in `LLVMTypeConverter::convertFloatType`.

Dialect Design

Dialect Overview from `cuda_tile`'s Point of View

Dialect Overview from `cuda_tile`’s Point of View

tt (Triton) — **vendor-agnostic**

cuda_tile (others…) — **vendor-specific**

arith math vector memref linalg — **vendor-agnostic**

Use arith ops to take advantage of canonicalizations, folders, upstream transformations.

m_neon, arm_sve, arm_sme | nvgpu | x86 | xegpu — **vendor-specific**

Generate nvvm ops to ensure that we utilize the right instructions.

rocdl | arm_neon.intr, arm_sve.intr, arm_sme.intr | nvvm | call_int nsic | xevm — **LLVM type system, vendor-specific**

llvm — **LLVM type system, vendor-agnostic**

LLVM Dialect

- Supported floating-point types according to LLVM IR
  - Builtin LLVM-supported types: `f16`, `bf16`, `f32`, `f64`, `f80`, `f128`
  - LLVM dialect types: `!llvm.ppc_fp128`
  - Other FP types are represented as integers (e.g, `f4E2M1FN` → `i4`). **The concrete type must be encoded in operation/intrinsic name or with extra attributes/operands.**
- Only “simple” arithmetic operations. Intrinsics for everything else.
  - Architecture-agnostic intrinsics. E.g.: `llvm.intr.matrix.multiply`
  - Instructions that map to specific hardware capabilities. Often via `llvm.call_intrinsic`.

```
%0 = llvm.call_intrinsic "llvm.x86.avx512bf16.dpbf16ps.128"(%arg0, %arg1, %arg2)
           : (vector<4xf32>, vector<8xbf16>, vector<8xbf16>) -> vector<4xf32>

nvvm.tcgen05.mma %d_tmem, %a_desc, %b_desc, %idesc, %enable_input_d
    {kind = #nvvm.tcgen05_mma_kind<f8f6f4>, ctaGroup = #nvvm.cta_group<cta_2>}
  : (!llvm.ptr<6>, i64, i64, i32, i1)
```

nvvm: NVIDIA’s Extension of the LLVM Dialect

- 5th gen tensor core (“Blackwell”) → `tcgen05`
- result / accumulator in TMEM (“tensor memory”) → `%d_tmem`
- shared memory descriptor: 14-bit ptr + swizzling info + … → `%a_desc`, `%b_desc`
- instruction descriptor: size M, N, K, exact FP types, etc. → `%idesc`

```
nvvm.tcgen05.mma %d_tmem, %a_desc, %b_desc, %idesc, %enable_input_d
    {kind = #nvvm.tcgen05_mma_kind<f8f6f4>, ctaGroup = #nvvm.cta_group<cta_1>}
  : (!llvm.ptr<6>, i64, i64, i32, i1)
```

Table 39: Various combinations of .kind and shapes

Various Combinations						Shapes Supported		
.kind::*	Has .ws	CTA Group	Sparsity	dtype	atype/btype			
kind::f16	No	1	Dense	.f16	.f16	64xNxK	N = {8, 16, 24, ...	K =
...								
			Sparse					
.kind::f8f6f4	No .ws	1	Dense	.f32 .f16	.e4m3, .e5m2, .e2m3, .e3m2, .e2m1	64xNxK 128xNxK	N = {8, 16, ... 256} steps of 8	K = 32
			Sparse					K = 64
		2	Dense			128xNxK 256xNxK	N = {16, 32, ... 256} steps of 16	K = 32

Corresponds to:

```
tcgen05.mma.cta_group::1.kind::f8f6f4
[%0], %1, %2, %3, PRED_enable_input_d;
```

Op design mirrors PTX:
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#tcgen05-kind-shapes

LLVM Dialect: “Simple” Arithmetics

```
%0 = llvm.fadd %a, %b {fastmathFlags = #llvm.fastmath<none>} : f32
```

- `none`
- `reassoc`
- `nnan`: values are never NaN
- `ninf`: values are never +-inf
- `nsz`: no signed zero (+0.0 == -0.0)
- `arcp`: x/y → x * (1/y)
- `contract`: allow fusing of add+mul → fma (rounding once)
- `afn`: allow approximate implementations
- `fast`: sets all fast math flags

```
%1 = llvm.intr.experimental.constrained.fadd %a, %b towardzero ignore : f32
```

`towardzero`: IEEE-754 rounding mode

Rounding modes constrain the operation, fath-math flags loosen the operation.
You cannot set both at the same time!

Rounding Modes + Fast Math Flags in NVIDIA PTX

- PTX rounding modes and IEEE-754 equivalents
  - `.rn` = roundTiesToEven
  - `.rz` = roundTowardZero
  - `.rm` = roundTowardNegative
  - `.rp` = roundTowardPositive
  - `.rs`: stochastic rounding, not supported by IEEE-754
- Fast math flags
  - `.ftz`: flush denormals to zero
  - `.approx`: approximation, but faster computation
- Slow / fast lowering strategies
  - `-fmad`: compiler flag to fuse multiplication + add into FMA instruction.
  - Fast intrinsic vs. slow `libdevice` library call. E.g., `sin.approx.f32` vs. `__nv_sinf`.

not available in LLVM

coming soon:

```
nvvm.fadd %a, %b
   rounding<rz>
   fast_math<ftz> : f32
```

`arith` / `math` Dialects

- Simple arithmetics operations that (mostly) have a direct lowering to LLVM.
- Can operate an scalars, tensors, vectors. Support for arbitrary FP types.
- Vendor-independent. No support for architecture-specific features (e.g., `ftz`).
- Rich collection of folders + canonicalization patterns (e.g., `x + 0.0 → x`).
- Limited support for rounding modes + fast math flags.
  - Rounding modes + fast-math modeled after the LLVM dialect.
  - “`TODO: In the distant future, this will accept optional attributes for fast math, contraction, rounding mode, and other controls.`”

Dialect Overview from `cuda_tile`'s Point of View

vendor-agnostic

tt (Triton)

vendor-specific

cuda_tile

(others…)

vendor-agnostic

arith

math

vector

memref

linalg

Use arith ops to take advantage of canonicalizations, folders, upstream transformations.

vendor-specific

rm_neon,
arm_sve,
arm_sme

nvgpu

x86

xegpu

Generate nvvm ops to ensure that we utilize the right instructions.

LLVM type system, vendor-specific

llvm

rocdl

arm_neon.intr,
arm_sve.intr,
arm_sme.intr

nvvm

call_intrinsic

xevm

LLVM type system, vendor-agnostic

Loading / Storing of Sub-Byte Types

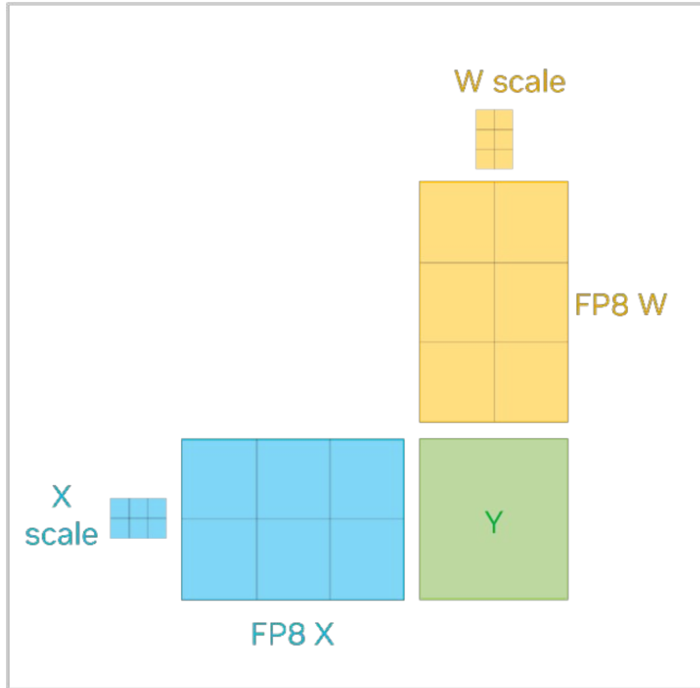
- Most hardware architectures do not support loading / storing less than 1 byte.
- Memory pointers typically point to the beginning of a byte.
- Vectorization or padding may be necessary for correctness and/or lowering.

```
// Can this be lowered? Will it overwrite the second 4 bits in memory?
ptr.store %val, %ptr : f4E2M1FN, !ptr.ptr
```

```
// Padded store.
%val_i4 = arith.bitcast %val : f4E2M1FN to i4
%val_ext = arith.extui %val_i4 : i4 to i8
ptr.store %val_ext, %ptr : i8, !ptr.ptr
```

```
// Vectorized store.
ptr.store %val_vec, %ptr
   : vector<2xf4E2M1FN>, !ptr.ptr
```

Block-Scaled Matmul in High/Low-Level Dialects



High-level:

- `tosa.matmul_t_block_scaled`
- `cuda_tile.mmaf_scaled`

High-level building blocks

- `linalg.matmul`
- `arith.scaling_extf` / `arith.scaling_truncf`
- `mx_tensor.quantize` (see [tpp-mlir])

Low-level:

- `nvvm.tcgen05.mma.block_scale`
- `rocdl.mfma.scale.*`
- `amdgpu.scaled_wmma`

Details: NVIDIA Developer Blog: Per-Tensor and Per-Block Scaling Strategies for Effective FP8 Training

Matmul in cuda_tile

```
%res = cuda_tile.mmaf          %A, %B, %C
    : tile<128x64xf8E5M2>,   tile<64x256xf8E5M2>,   tile<128x256xf32>
```

Block-Scaled Matmul in `cuda_tile`

```
%res = cuda_tile.mmaf_scaled %A, %B, %C, %sfA, %sfB
    : tile<128x64xf8E5M2>,   tile<64x256xf8E5M2>,   tile<128x256xf32>,
      tile<128x2xf8E8M0FNU>, tile<2x256xf8E8M0FNU>
```

every 32 “k” values share a scaling factor

Block-Scaled Matmul in `cuda_tile`

```
%sfA_3d   = reshape %sfA : tile<128x2xf8E8M0FNU> -> tile<128x2x1xf8E8M0FNU>
%sfA_bc   = broadcast %sfA_3d : tile<128x2x1xf8E8M0FNU> -> tile<128x2x32xf8E8M0FNU>
%sfA_flat = reshape %sfA_bc : tile<128x2x32xf8E8M0FNU> -> tile<128x64xf8E8M0FNU>
%A_f16    = ftof %A : tile<128x64xf8E5M2> -> tile<128x64xf16>
%sfA_f16  = ftof %sfA_flat : tile<128x64xf8E8M0FNU> -> tile<128x64xf16>
%A_scaled = mulf %A_f16, %sfA_f16 : tile<128x64xf16>
// (Same for B.)

%r = mmaf %A_scaled, %B_scaled, %C : tile<128x64xf16>, tile<64x256xf16>, tile<128x256xf32>
```

Block-Scaled Matmul in `cuda_tile`

```
%sfA_3d    = reshape %sfA : tile<128x2xf8E8M0FNU> -> tile<128x2x1xf8E8M0FNU>
%sfA_bc    = broadcast %sfA_3d : tile<128x2x1xf8E8M0FNU> -> tile<128x2x32xf8E8M0FNU>
%sfA_flat = reshape %sfA_bc : tile<128x2x32xf8E8M0FNU> -> tile<128x64xf8E8M0FNU>
%A_f16     = ftof %A : tile<128x64xf8E5M2> -> tile<128x64xf16>
%sfA_f16   = ftof %sfA_flat : tile<128x64xf8E8M0FNU> -> tile<128x64xf16>
%A_scaled = mulf %A_f16, %sfA_f16 : tile<128x64xf16>
// (Same for B.)


%r = mmaf %A_scaled, %B_scaled, %C : tile<128x64xf16>, tile<64x256xf16>, tile<128x256xf32>
```

Block-Scaled Matmul in cuda_tile / arith

```
%sfA_3d   = reshape %sfA : tile<128x2xf8E8M0FNU> -> tile<128x2x1xf8E8M0FNU>
%sfA_bc   = broadcast %sfA_3d : tile<128x2x1xf8E8M0FNU> -> tile<128x2x32xf8E8M0FNU>
%sfA_flat = reshape %sfA_bc : tile<128x2x32xf8E8M0FNU> -> tile<128x64xf8E8M0FNU>

%A_scaled = arith.scaling_extf %A, %sfA_flat
    : tile<128x64xf8E5M2>, tile<128x64xf8E8M0FNU> to tile<128x64xf16>
// (Same for B.)

%r = mmaf %A_scaled, %B_scaled, %C : tile<128x64xf16>, tile<64x256xf16>, tile<128x256xf32>
```

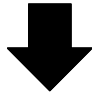
Emulation of Unsupported FP Types

Emulation of Unsupported FP Types

1. Software emulation via runtime library:
   `--convert-arith-to-apfloat`
2. Software emulation via IR expansion:
   `--arith-expand`,
   `--math-expand-ops`
3. Upcasting to supported FP type:
   `--arith-emulate-unsupported-floats`,
   `--math-extend-to-supported-types`

Software Emulation via Runtime Library

```
// RUN: mlir-opt %s --convert-arith-to-apfloat

func.func @addf(%arg0: f4E2M1FN, %arg1: f4E2M1FN) -> f4E2M1FN {
  %0 = arith.addf %arg0, %arg1 : f4E2M1FN
  return %0 : f4E2M1FN
}
```



all floats are passed as i64

```
func.func private @_mlir_apfloat_add(i32, i64, i64) -> i64
```

```
MLIR_APFLOAT_WRAPPERS_EXPORT int64_t _mlir_apfloat_add(
    int32_t semantics, uint64_t a, uint64_t b) {
  const llvm::fltSemantics &sem = llvm::APFloatBase::EnumToSemantics(
      static_cast<llvm::APFloatBase::Semantics>(semantics));
  unsigned bitWidth = llvm::APFloatBase::semanticsSizeInBits(sem);
  llvm::APFloat lhs(sem, llvm::APInt(bitWidth, a));
  llvm::APFloat rhs(sem, llvm::APInt(bitWidth, b));
  lhs.add(rhs);
  return lhs.bitcastToAPInt().getZExtValue();
}
```

```
func.func @addf(%arg0: f4E2M1FN, %arg1: f4E2M1FN) -> f4E2M1FN {
  %0 = arith.bitcast %arg0 : f4E2M1FN to i4
  %1 = arith.extui %0 : i4 to i64
  %2 = arith.bitcast %arg1 : f4E2M1FN to i4
  %3 = arith.extui %2 : i4 to i64
  %c18_i32 = arith.constant 18 : i32
  %4 = call @_mlir_apfloat_add(%c18_i32, %1, %3) : (i32, i64, i64) -> i64
  %5 = arith.trunci %4 : i64 to i4
  %6 = arith.bitcast %5 : i4 to f4E2M1FN
  return %6 : f4E2M1FN
}
```

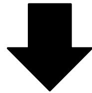
bit-cast to i4 and extend to i64

f4E2M1FN is element number 18 in enum APFloatBase::Semantics

truncate to i4 and bit-cast back to f4E2M1FN

Software Emulation via Runtime Library

```
// RUN: mlir-opt %s --convert-arith-to-apfloat

func.func @extf(%arg0: f4E2M1FN) -> f32 {
  %0 = arith.extf : f4E2M1FN to f32
  return %0 : f32
}
```

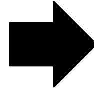


```
func.func private @_mlir_apfloat_convert(i32, i32, i64) -> i64

func.func @extf(%arg0: f4E2M1FN) -> f32 {
  %0 = arith.bitcast %arg0 : f4E2M1FN to i4
  %1 = arith.extui %0 : i4 to i64
  %c18_i32 = arith.constant 18 : i32
  %c2_i32 = arith.constant 2 : i32
  %2 = call @_mlir_apfloat_convert(%c18_i32, %c2_i32, %1) : (i32, i32, i64) -> i64
  %3 = arith.trunci %2 : i64 to i32
  %4 = arith.bitcast %3 : i32 to f32
  return %4 : f32
 }
```

`f4E2M1FN` is element number 18, `f32` is element number 2 in enum `APFloatBase::Semantics`

Software Emulation via IR Expansion

```
// RUN: mlir-opt %s \
// RUN:      -arith-expand="include-f4e2m1=true"

func.func @extf(%arg0: f4E2M1FN) -> f32 {
  %0 = arith.extf : f4E2M1FN to f32
  return %0 : f32
}
```

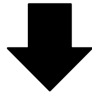


```
func.func @extf(%arg0: f4E2M1FN) -> f32 {
  %c-8_i4 = arith.constant -8 : i4
  %c-2147483648_i32 = arith.constant -2147483648 : i32
  %c1056964608_i32 = arith.constant 1056964608 : i32
  %c1073741824_i32 = arith.constant 1073741824 : i32
  %c0_i32 = arith.constant 0 : i32
  %c20_i32 = arith.constant 20 : i32
  %c7_i4 = arith.constant 7 : i4
  %c4_i4 = arith.constant 4 : i4
  %c2_i4 = arith.constant 2 : i4
  %c1_i4 = arith.constant 1 : i4
  %c0_i4 = arith.constant 0 : i4
  %0 = arith.bitcast %arg0 : f4E2M1FN to i4
  %1 = arith.andi %0, %c7_i4 : i4
  %2 = arith.shli %1, %c2_i4 : i4
  %3 = arith.cmpi eq, %1, %c1_i4 : i4
  %4 = arith.select %3, %c0_i4, %2 : i4
  %5 = arith.extui %4 : i4 to i32
  %6 = arith.shli %5, %c20_i32 : i32
  %7 = arith.cmpi uge, %1, %c4_i4 : i4
  %8 = arith.select %7, %c1073741824_i32, %c1056964608_i32 : i32
  %9 = arith.cmpi eq, %1, %c0_i4 : i4
  %10 = arith.select %9, %c0_i32, %8 : i32
  %11 = arith.cmpi uge, %0, %c-8_i4 : i4
  %12 = arith.select %11, %c-2147483648_i32, %c0_i32 : i32
  %13 = arith.addi %6, %10 : i32
  %14 = arith.addi %13, %12 : i32
  %15 = arith.bitcast %14 : i32 to f32
  return %15 : f32
}
```

Only a few operations / types are supported today.

Software Emulation via Upcasting

```
// RUN: mlir-opt %s --arith-emulate-unsupported-floats

func.func @addf(%arg0: f4E2M1FN, %arg1: f4E2M1FN) -> f4E2M1FN {
  %0 = arith.addf %arg0, %arg1 : f4E2M1FN
  return %0 : f4E2M1FN
}
```



```
func.func @addf(%arg0: f4E2M1FN, %arg1: f4E2M1FN) -> f4E2M1FN {
  %0 = arith.extf %arg1 fastmath<contract> : f4E2M1FN to f32
  %1 = arith.extf %arg0 fastmath<contract> : f4E2M1FN to f32
  %2 = arith.addf %1, %0 : f32
  %3 = arith.truncf %2 fastmath<contract> : f32 to f4E2M1FN
  return %3 : f4E2M1FN
}
```

`extf`, `truncf` ops remain
(and can be emulated via IR expansion)

Summary

Summary

- MLIR floating-point types implement the `FloatType` **interface**.
- Foldings/canonicalizations based on `APFloat` **software implementation**.
    - Future work: Modularize the infrastructure to make it extensible without having to patch LLVM.
- Vendor-independent dialects: `arith`, `math`, `tosa`, `llvm`, …
    - Open question: With increasing vendor-specific types / features, shared dialects become harder to use. How to avoid duplication (e.g., each vendor having their own `arith` copy)?
- Vendor-dependent dialects: `cuda_tile`, `nvvm`, `amdgpu`, `arm_sve`, `x86`, …
- Software emulation of FP types is useful for various reasons (correctness checking, forward/backward compatibility across architectures, …) and implementations have been contributed by various vendors.

Questions?

`APFloat`
`FloatTypeInterface`
IEEE-754
OCP Microscaling Formats (MX)
Rounding Modes
Stochastic Rounding
Fast Math Flags
Denormals and Flush-to-zero
Approximate Implementation
`arith` / `math` Dialect
cuda_tile Dialect
`llvm` / `nvvm` Dialect
LLVM intrinsics

Tensor Cores
PTX
Block-scaled Matrix Multiplication
`cuda_tile.mmaf_scaled`
`arith.scaling_extf`
Loading/storing Sub-byte Values
FP Software Emulation
`-convert-arith-to-apfloat` (CPU impl.)
`-arith-expand-ops` (arith impl.)
`-arith-emulate-unsupported-floats` (upcasting)

The CUDA Tile IR team (and other compiler teams at NVIDIA) are hiring.