BASE2: An IR for Binary Numeral Types

Karl F. A. Friebel
Technische Universität Dresden
Dresden, Germany
karl.friebel@tu-dresden.de

Jiahong Bi
Technische Universität Dresden
Dresden, Germany
jiahong.bi@mailbox.tu-dresden.de

Jeronimo Castrillon
Technische Universität Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

ABSTRACT

Custom data types and arbitrary-precision arithmetic are often key for efficient hardware designs on Field Programmable Gate Array (FPGA) platforms. Current end-to-end flows incorporating quantization are not only domain-specific, but also tightly integrated and not repurposable. Abstractions for arbitrary-precision arithmetic are generally vendor-specific, and results are hardly portable across platforms. In this work, we present a new Intermediate Representation (IR), BASE2, to address the programmability issues of custom data types in reconfigurable hardware. We contextualize our proposal in the greater LLVM ecosystem, where we show how existing abstractions can be simplified and unified. We implement BASE2 in Multi-Level Intermediate Representation (MLIR), which allows it to be used in a variety of existing and future target-agnostic frontends. We demonstrate the power of our model by applying it to sample kernels and evaluating the accuracy of the result. For these samples, we achieve interoperability with an existing end-to-end High-Level Synthesis (HLS) flow.

ACM Reference Format:
Karl F. A. Friebel, Jiahong Bi, and Jeronimo Castrillon. 2023. BASE2: An IR for Binary Numeral Types. In *The International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies 2023 (HEART 2023), June 14–16, 2023, Kusatsu, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3597031.3597048

1 INTRODUCTION

In continuous efforts to improve latency, throughput and energy-efficiency of processing pipelines, specialized accelerators have become essential in computing. Typically in the form of *fixed-function accelerators*, such devices execute predefined dataflow programs with high efficiency. Compared to general-purpose hardware, they may offer vastly reduced area and energy footprint, as well as hard real-time performance.

Systems for general-purpose computing can incorporate such accelerators through FPGAs devices, which are examples of *reconfigurable hardware*. FPGAs can achieve very high energy efficiency [16] and meet hard real-time requirements [15], making them attractive also for edge systems. This comes at a cost to programmability, as usually a specialized hardware designer is needed to exploit the FPGA's flexibility profitably.

Good hardware designs rely on pipelining and custom memory hierarchies, but also on specialized data types. For data types, the flexibility of the FPGA allows the designer to implement arbitrary precision arithmetic, which provides a fine-grained trade off compared to traditional Instruction Set Architecture (ISA)-based code generation. The basis for this type substitution is quantization, which is usually constrained by domain-specific considerations. For example, in deep learning, 4-bit quantizations can produce smaller and faster FPGA designs at negligible accuracy losses [5].

Most FPGA designs are generated using specialized compiler tools, with HLS playing the most important role for general applications. These tools limit the hardware designer in their design choices. In general, quantizing a kernel requires to (1) find the quantization function that maps the original variables to quantized values that are representable on the target device, (2) substitute the storage type to the chosen target data type for all occurrences of the original variable type, (3) replace arithmetic on substituted values with quantization-aware implementations (e.g., re-quantize where necessary), and (4) generate target code. If the designer relies on HLS for code generation (4), they are limited by the HLS front-end in terms of usable storage types. Any external tooling capable of (1) must rely on a semantic model of (2) and (3), and may therefore be incompatible, or at least not interoperable, with HLS. Performing (2) and (3) by hand is often a large refactoring task which is severely error prone due to language semantics, for example in C/C++, which is common for HLS.

Currently available end-to-end solutions for quantization are domain-specific. For example, a deep-learning neural network can be trained for quantization by a tool such as Brevitas [2] with little user interaction. This tool performs steps (1) to (3) completely within the neural network abstraction. A specialized synthesis tool called FINN [22] can then be used to generate an FPGA design based on the output. Neither Brevitas nor FINN generalize to non-ML applications, and the IP cores generated in this way cannot be fine-tuned by the user.

Motivation & Contributions

Reconfigurable devices offer new opportunities in mapping algorithms at the cost of increasing the *programmability effort*. With a focus on custom data type support, designers face challenges with respect to **productivity**, since it is harder to implement and maintain their hardware/software solution, **flexibility**, since it is hard to freely choose and combine arbitrary-precision storage types, and **portability**, since it is hard to move to another device and/or toolchain while still achieving a comparable Quality of Result (QoR).

Productivity requires the toolchain to offer an accessible API. Xilinx Vitis ships a C++ library that provides an `ap_fixed` type which allows users to easily express type-safe fixed-point computations. Bambu [6] on the other hand ships and recognizes libsoftfloat [18] functions to synthesize efficient floating-point IP cores. Both (natively) lack the other's abstraction respectively. We define a specification for generalized binary arbitrary-precision arithmetic with a straightforward interface.

Flexibility is limited by the API, sometimes ISA, that implements the storage types. Although parametric, ap_fixed requires a "word-internal" decimal point, i.e., it imposes an arbitrary exponent limit. In addition, it affixes overflow and rounding behavior to the type, making it harder to mix them. Similarly, while libsoftfloat can represent all binary IEEE-754 [9] derivatives, it cannot handle emerging floating-point types such as 8E5M3FN. Our specification generalizes towards fixed-point and floating-point computations with arbitrary parametrizations.

Portability is limited by the IR used. Although generated Hardware Description Language (HDL) is generally portable between devices, higher-level artefacts are generally not. In the world of FPGAs, there is a substantial amount of vendor lock-in that prevents code portability and other interoperation. In our above examples, this is especially evident for the arbitrary-precision compute part of the design. We implement a set of extensible and reusable MLIR [11] dialects that can integrate into end-to-end flows.

We propose a new intermediate abstraction called BASE2 which provides a target-agnostic way to use arbitrary-precision arithmetic. It is based on a rigorous specification that eliminates architecture-dependent behavior from the concerns of the front-end, while providing as much flexibility in type parameterization as possible. We design BASE2 as a unification of compiler-level arithmetic models that integrates with end-to-end compilation and synthesis flows.

MLIR End-to-end flows have already been demonstrated [21] through the use of Bambu's [6] or Xilinx Vitis's LLVM IR front-ends. In the future, the CIRCT [3] project is poised to bring HLS to MLIR entirely, with some facilities already operable. We demonstrate the value of retargetability by using it to perform design space exploration on the CPU.

2 BACKGROUND

The design of BASE2 rests on four assumptions that are true for the vast majority of current and emerging compute hardware, reconfigurable or not. First and foremost, data is predominantly encoded in binary formats. Second, cyclic integers are the the most fundamental data types. Third, the most common representation of signed integers is two's complement. Finally, arithmetic is either fixed-point or floating-point. This section provides fundamentals of number representations and rounding.

2.1 Binary numbers

Binary numbers are representations of numbers in binary encoding. A binary encoding represents a value as an ordered sequence of bits. These encodings are not necessarily consistent across architectures (e.g., different endianness). However, the majority of platforms share certain encoding properties that make bit manipulations relative to a canonical order (e.g., MSB to LSB) meaningful.

2.1.1 Cyclic binary integers. Virtually all modern systems support cyclic binary integers as their most fundamental data type. These model the cyclic group $\langle \mathbb{Z}_{2^n}, +, \cdot \rangle$, where n is their *bit width*. Encoded as positional binary numerals, they are the *unsigned integers.*

The two's complement representation is an isomorphism for the negative integers onto the cyclic group. This encoding allows negative values to be encoded without a sign symbol, and is the the de-facto standard onon virtually all platforms [10]. Interpreting bits as an integer value thus requires knowing its *signedness.* However, the operations of addition, subtraction and multiplication are implemented using the same circuit, regardless of signedness.

2.1.2 Fixed-point and floating-point arithmetic. All other types of numbers considered by BASE2, and most in practical use, model the rational numbers (and some irrational points). For efficiency, this is achieved using a pair integers z, E such that

$$\mathbb{Q} = \left\{ z \cdot 2^E : z, E \in \mathbb{N} \right\}$$

where z is the *significand* and E is the *binary exponent.*

If the exponent E is implied, i.e., the number is represented solely by z, it is called a *fixed-point* number. If the exponent is explicit, it is a *floating-point* number. These two formats do not differ in their mathematical properties, only in the points they can represent. Since machine representations use finite integers to represent them, different limitations apply.

2.1.3 IEEE-754 binary floating-point. The IEEE-754 [9] standard defines a family of interoperable and ubiquitous floating-point encodings and semantics. The binary family models rational numbers as defined above, with the additional non-finite values of $\pm\infty$ and Not-a-Number (NaN). Both are encoded using reserved exponent values.

The infinities satisfy $\forall q \in \mathbb{Q}. -\infty < q < \infty$, meaning they are ordered. Additionally, it holds that $|\pm\infty| = |\pm\infty|$, meaning that they have a defined magnitude greater than that of any finite number. This makes them a useful extension of the value range that integrates well into rounding schemes and operational semantics.

NaN values have platform-dependent semantics. They are intended to propagate errors, i.e., poison values, where the result of an operation is undefined. IEEE-754 defines both signalling and quiet NaNs, with the intent being a difference in how the hardware reports these as errors. Different implementations of the standard have diverged incompatibly in this regard, leading to the latest standard [9] dropping some wording.

2.1.4 IEEE-754 derivatives. Types such as bfloat16, 8E5M2 and 8E4M3FN [12] are derivatives of the IEEE-754 standard. They adopt a similar encoding, notion of special values, and operational semantics.

At first glance, the bfloat16 format for Tensor Processing Units (TPUs) is a 16-bit binary floating-point type as per IEEE-754. It is an application-specific compromise. It has the same number of exponent bits as binary32, but less significand bits than binary16. The manufacturers explicitly do not grant IEEE-754 compliance, possibly due to minute differences in operational semantics.

The 8E5M2 and 8E4M3FN [12] formats are cousins that have the exact same exponent range. However, 8E4M3FN has an additional bit of significand precision. This is accomplished by removing the infinities from the encoding, thus having two more finite exponent values. As a result, 8E4M3FN is not IEEE-754 compliant.

2.1.5 Other floating-point formats. Other non-derivative floating-point formats are under serious consideration. An important example are Posits [8], which achieve accuracy comparable to IEEE-754 using fewer bits and less silicon area. They feature a run-length encoding using *regime bits* to provide *tapered accuracy* that improves near a magnitude of 1.

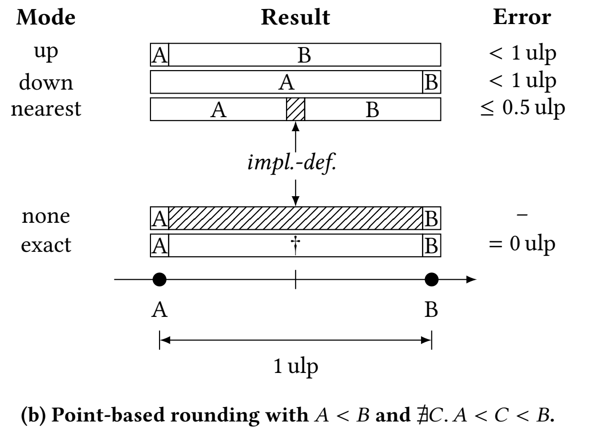
Posits do not fundamentally violate the assumptions of base2: they are binary, and decomposable into a significand and a binary exponent. However, while accomodated by the specification, our implementation does not include parameterizations for them. It is yet unclear whether the common floating-point environment of base2 can fit them.

2.2 Rounding

Rounding is an important aspect of operational semantics. It determines how an exact (mathematical) result is represented by a number type which only has a limited set of points. Rounding is only applied if the result is not exactly representable. The policy applied is usually called a rounding mode, as are illustrated in fig. 1.

Name	Constraint
none	—
exact	$x' = x$
	$\min\lvert x' - x\rvert$, such that
nearest	—
down	$x' \not> x$
up	$x' \not< x$
towards zero	$\lvert x'\rvert \not> \lvert x\rvert$
away from zero	$\lvert x'\rvert \not< \lvert x\rvert$
converge	$\sum_{\infty}(x'_i - x_i) = 0$

(a) Rounding modes defined by constraints on the result x' in relation to input x.



(b) Point-based rounding with $A < B$ and $\nexists C.\, A < C < B$.

Figure 1: Illustration of common rounding modes.

2.2.1 Poison and implementation-defined rounding. In fig. 1, † is used to represent the poison value, which indicates an undefined result. It occurs when a constraint as shown in fig. 1a cannot be satisfied. This is different from *impl.-def.* behavior, which is defined elsewhere, observable, and may be relied on by the user. It is important that a target-agnostic front-end has access to well-defined rounding contracts to meet accuracy goals for its computations.

2.2.2 Unit in the Last Place (ulp) calculus. In fig. 1b, *point-based* rounding is shown. Here, the discrete points (number-like values) representable by the result type are given on an ordered number line, and rounding ranges are shown. The resulting point is chosen by determining into which range the exact input falls. For the zero-relative rounding modes, either up or down is chosen based on the relation to 0. Note that 0 is not necessarily a representable point.

The diagram also shows the definition of the Unit in the Last Place (ulp). This unit is commonly used to bound the rounding error, such as in MPFR [7], which defines correct floating-point operations. For any implementation of an operation, a proof is constructed that bounds the ulp error of the result, also called *ulp calculus*. A result is said to be "correctly rounded" if the rounding error is less than or equal to 0.5 ulp.

2.3 LLVM IR and MLIR

LLVM IR is a widespread and successful Single Static Assignment (SSA) IR, supported also by HLS tools like Xilinx Vitis. Its instructions define a contract that all targets are expected to implement. MLIR is an extensible IR, in which dialects collect attributes, types and operations that form an —ideally self-contained— abstraction. The `llvm` dialect is an incomplete embedding of LLVM IR in MLIR.

Typically, MLIR dialects define lowerings, i.e., conversions to more concrete dialects. In that sense, to an MLIR front-end, operations and dialects are contracts as well. MLIR is unstable, and contracts continue to be refined. Usually, this means a dialect is broken apart into separate responsibilities, resulting in less overall assumptions. One such dialect under current scrutiny is the `arith` dialect. We designed base2 to be a drop-in replacement that solves the underlying issues (e.g., poison semantics) by stronger specification. For now, the MLIR community has already extracted the `index` dialect from `arith` for similar reasons.

2.3.1 Arithmetic in LLVM IR. In LLVM IR, there are parametric signless integer types and a fixed set of pre-defined IEEE-754 derivative floating-point types. The instruction set is designed with cyclic integers in mind, with signedness becoming part of the operation (e.g., `udiv` & `sdiv`). LLVM IR has a built-in poison value system, which operations can opt into for certain overflow scenarios (e.g., `udiv exact`).

Having only signless integers leads to a simplification in code selection and aggressive instruction combination. This is partly due to the absence of (no-op) type cast instructions in typical integer code. A close mapping onto common ISAs is achieved. Additionally, poison values allow for more aggressive compile-time optimizations.

LLVM IR is not extensible, and support for additional types is severely limited. While there are fixed-point intrinsics, such types cannot become first-class citizens, living among separate instructions. Conversely, LLVM IR is forced to mix domain-specific integer arithmetic with pointer and index calculations, which could live under different, more favorable assumptions.

2.3.2 The MLIR `arith` *dialect.* MLIR supports all LLVM IR types as built-ins, which includes signless integers and IEEE-754 derivatives. Additionally, MLIR integers may have a signedness, and new types can be added. While the `llvm` dialect is virtually equivalent to LLVM IR, the `arith` dialect is a model of an arithmetic instruction subset of LLVM IR.

MLIR's `arith` shares the same advantages and disadvantages (except for index computations) as LLVM IR, by design. However, it does not support poison semantics, and does not offer such opt-in behavior. In addition, it does not guarantee a contract for its operations. Although they exist, signed and unsigned integers are not usable in the `arith` dialect.

2.3.3 The MLIR `index` *dialect.* Index computations in MLIR are delegated to a platform-specific integer type called `index` of platform-specific bit width (i.e., a machine address word). It is used widely in the core dialects. The `index` dialect provides constant, casting, comparison and arithmetic operations for this type.

While still implementing a lowering onto `llvm`, the advantage of the `index` dialect is the relaxation of the contracts. Users cannot expect a consistent representation between targets. This is similar to the difference between `std::int64_t` and `std::intptr_t` in C++. Also, any kind of overflow in the `index` dialect can be defined to constitute undefined behavior.

3 BASE2: AN IR FOR BINARY NUMERALS

We designed an SSA IR called BASE2 to address the programmability challenges for arbitrary-precision arithmetic. We implemented this IR as a set of MLIR dialects. The `base2` dialect is a new medium-level abstraction layer for target-agnostic IR producers, such as linear algebra front-ends. One of our explicit design goals is to replace `arith` in all contexts, thus avoiding signless integers and unwanted Undefined Behavior (UB) above the code-selection level.

3.1 Type hierarchy

BASE2 is designed to be minimal but complete with respect to the most common binary number formats. To simplify implementation and extensibility, the requirements placed on its member types are layered. To that extent, BASE2 defines an internal hierarchy of types, as depicted in fig. 2. In summary:

- **Bit sequence types** are represented by bit sequences of a fixed length ω_B.
- **Interpretable types** are bit sequence types with an associated interpretation function Γ_P, i.e., a (reversible) function that maps bit sequences to points.
- **Number types** are interpretable types with an associated rounding function Ω_{N_R} i.e., a function that maps rational numbers to points under a given rounding mode R.
- **Fixed-point types** are number types represented by canonical integers with an associated fixed exponent.
- **Floating-point types** are number types decomposable into an integer significand and exponent of base 2.

These requirements are sufficient to generalize the common operations to all number types. This allows for type- and thus target-independent contracts to be established by the specification. From these requirements, we derive additional tools such as the subtype relation and the type promotion operator (least common supertype).

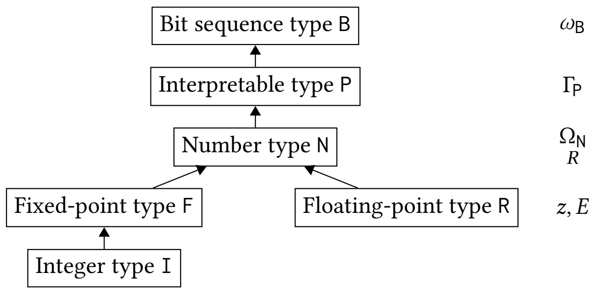


Figure 2: The BASE2 type hierarchy.

The scope of BASE2 is limited to fixed-point and floating-point binary rationals. While the canonical cyclic integer representation fully defines every allowed fixed-point format, the same is not true for floating-point encodings. Fixed-point and integer types are totally specified by BASE2, whereas floating-point formats may exhibit *impl.-def.* behavior.

3.2 Operation definitions

BASE2 provides the common, closed arithmetic operations of addition, subtraction, multiplication, division and remainder. These operations rely on an inherent rounding mode, which constrains the result. We define all of them by applying the type-specific rounding function to the exact mathematical result, which is obtained through interpretation, e.g.

$$\mathtt{add}(a, b, R) = \Omega_{\mathsf{N}_R}\left(\Gamma_\mathsf{N}(a) + \Gamma_\mathsf{N}(b)\right)$$

As another example, the remainder operation disambiguates the "modulo" operator using a rounding mode R via

$$\mathtt{rem}(a, b, R) = \Gamma_\mathsf{N}(a) - \Gamma_\mathsf{N}(b)\mathtt{div}(a, b, R)$$

To remain independent of the encoding, BASE2 does not allow bit manipulation. Some common bit-centric operations can still be expressed in a compatible and portable manner. For example, the (arithmetic) bit shifting operations with $z \in \mathbb{Z}$ and $k, n \in \mathbb{N}$ are given by

$$\mathtt{shl}(z, k) \equiv z \cdot 2^k \mod 2^n$$
$$\mathtt{shr}(z, k) = \lfloor z \cdot 2^{-k} \rfloor$$

Overflow and underflow in BASE2 are covered by the rounding modes. Depending on the requested rounding, the constraints from fig. 1a force a certain saturation behavior as shown in fig. 3a.

Another aspect of BASE2 is the ability to bootstrap rounding, i.e., implement the rounding modes using exact BASE2 operations. This is important as not every platform will implement every rounding mode. For integer saturation, this is trivial. For fractional rounding, we can make use of previous results. Let $z \in \mathbb{Z}$ and $n \in \mathbb{N}$, it follows that

$$\begin{aligned}\Omega_{\mathsf{T}_R}\left(z \cdot 2^{-n}\right) &= \mathtt{div}\left(z', 2^n, \mathrm{down}\right) \\ &= \mathtt{shr}\left(\mathtt{add}\left(z, \epsilon_R(z, n), R\right), n\right)\end{aligned}$$

where ϵ_R is the rounding bias function (cf. fig. 3b).

Mode	**Underflow**	**Overflow**
none	*impl.-def.*	
exact	†	†
nearest	min Y	max Y
down	*impl.-def.*	max Y
up	min Y	*impl.-def.*
towards zero	min Y	max Y
away from zero	*impl.-def.*	
converge	*impl.-def.*	

(a) Saturation on overflow and underflow.

Mode	$\epsilon_R(z, n)$
none	*impl.-def.*
exact	—
nearest	$2^{-n-1} - \begin{cases} 1 & z < 0 \\ 0 & z \geq 0 \end{cases}$
down	0
up	$2^{-n} - 1$
towards zero	$z \geq 0 : \epsilon_{\text{down}}(z, n), z < 0 : \epsilon_{\text{up}}(z, n)$
away from zero	$z \geq 0 : \epsilon_{\text{up}}(z, n), z < 0 : \epsilon_{\text{down}}(z, n)$
converge	*impl.-def.*

(b) Rounding bias for fractional rounding.

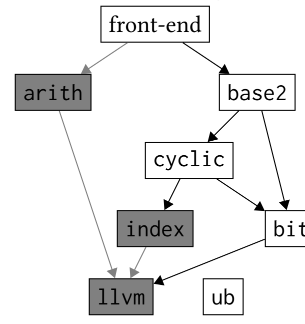
Figure 3: Implementation of the rounding modes.

	Existing dialects
`arith`	Signless integer types of arbitrary bit width. Some binary floating-point types. Arithmetic with underdefined overflow and rounding.
`index`	An integer type suitable for address computation. Its values are ordered and support arithmetic. Overflow is explicitly undefined.
	`base2-mlir` dialects
`bit`	Manipulation of ordered bit sequences.
`cyclic`	Parametric unsigned and signed cyclic integers.
`base2`	Parametric fixed-point and floating-point arithmetic.

(a) Dialect contracts.

`ub.`	`#poison, poison, freeze`
`bit.`	`#bits, #dense_bits, constant, cast, cmp, select` `and, or, xor, shl, shr, count, clz, ctz`
`cyclic.`	`constant, trunc, ext, cmp, min, max, shl, shr` `add, sub, mul, div, rem`
`cyclic.checked.`	`trunc, shl, add, sub, mul, div`
`base2.`	`!fixed, !ieee754, cast, cmp, min, max, add` `sub, mul, div, rem`
`base2.fixed.`	`add, sub, mul, div, rem`
`base2.float.`	`ilogbn, scalbn`

(b) Attributes, types and operations in `base2-mlir`. Incomplete features are in gray.



(c) Dialect lowerings in `base2-mlir`. MLIR core features in gray.

Figure 4: Breakdown of the `base2-mlir` MLIR project.

3.3 MLIR implementation

Our MLIR implementation `base2-mlir`[1] is the reference implementation of the BASE2 specification[2]. It provides target-agnostic front-ends with contracts for fixed-point and floating-point arithmetic on parametric types. While fixed-point types are covered in their entirety, the MLIR implementation is extensible via interfaces to add further formats, especially for floating-point. The current implementation is incomplete with respect to our unification goals for a common floating-point environment.

3.3.1 Contracts. The new `base2` dialect is the highest level of abstraction provided by our implementation, and is intended to completely replace `arith`. It depends on cyclic integers, which are modeled by the new `cyclic` dialect. Bitwise operations are provided by the new `bit` dialect. Undefined behavior is modeled using the new `ub` dialect, which we hope will be provided by the MLIR core in the future. All new dialects together with `index` combine to form a lowering path to `llvm` (see fig. 4c). The IR elements offered by the new dialects are listed in fig. 4b. The contract each dialect offers in that scenario is summarized in fig. 4a.

3.3.2 Interfaces. `base2-mlir` is designed for extensibility and reusability. The implementation relies heavily on MLIR interfaces and type & attribute specializations. This greatly reduces the effort required to add compliant types, even when they are outside the user's control. For example, the `FixedPointSemantics` and `IEEE754Semantics` interfaces seamlessly integrate the built-in types into `base2-mlir`. Support for handling scalar and container constants of bit sequence types is provided by our `BitSequenceAttr` and `DenseBitSequencesAttr` of the `bit` dialect. This trivializes the implementation of constant folding that would otherwise be hard to implement for new binary encodings.

3.3.3 End-to-end usage. For CPU targets, `base2-mlir` is directed at typical users which expect a reasonably efficient and functional lowering pipeline to `llvm`. For reconfigurable targets, `base2-mlir` is directed at advanced users that are in posession of either vendor target dialects or pluggable HLS. An end-to-end flow requires exiting the `base2` lowering flow at an interoperable level, for example via LLVM IR or CIRCT HLS.

As users of Bambu HLS [6], we exit at the `llvm` level, lowering fixed-point entirely, and implementing `!base2.ieee754` via `libsoftfloat` calls. We achieve this using a trivial, new `softfloat`

[1] https://github.com/KFAFSP/base2-mlir
[2] https://github.com/KFAFSP/base2-spec

dialect, mapping library operations 1 : 1, which `base2` can be trivially lowered to, and is in turn trivially lowered to `func` and thus `llvm`.

4 EVALUATION

To test our `base2-mlir` implementation, we performed two experiments. In section 4.1, we use a simple test kernel to show how arbitrary-precision types in BASE2 can be used. In section 4.2, we look at an excerpt of a more complex real-world application, which exhibits promising opportunities for fixed-point arithmetic.

4.1 Interpolation operator

In our first experiment, we use a simple MLIR kernel from the linear algebra domain. We use the tensorial isotropic interpolation kernel from [19], which is given by

$$v = (S \otimes S \otimes S \otimes u)^{axbycz}{}_{xyz}$$

We replace the scalar type with `base2.ieee754` and `base2.fixed`, and change all uses of `arith` operations to the `base2` dialect. We compile the kernel in both a standard `f64` (11 exponent bits, 52 significand bits) and the modified version (cf. fig. 5). Using a C++ test adapter, we invoke these kernels for randomized data and compare the relative errors.

```
#contr1_trait = {
    indexing_maps = [
        affine_map<(a,b,c,x,y,z) -> (a,x)>,
        affine_map<(a,b,c,x,y,z) -> (b,y)>,
        affine_map<(a,b,c,x,y,z) -> (c,z)>,
        affine_map<(a,b,c,x,y,z) -> (x,y,z)>,
        affine_map<(a,b,c,x,y,z) -> (x,y,z)>
    ],
    iterator_types = [
        "reduction", "reduction", "reduction",
        "parallel", "parallel", "parallel"
    ]
}
!scalar = !binary.ieee754<53, 11>
!S_type = tensor<11x11x!scalar>
!u_type = tensor<11x11x11x!scalar>

%t = linalg.generic #contr1_trait
        ins(%S, %S, %S, %u : !S_type, !S_type, !S_type, !u_type)
        outs(%zeros : !u_type) {
    ^bb0(%Sax : !scalar, %Sby : !scalar, %Scz : !scalar, %uxyz : !scalar, %accu : !scalar):
        %1 = binary.mul %Sax, %Sby : !scalar
        %2 = binary.mul %1, %Scz : !scalar
        %3 = binary.mul %2, %uxyz : !scalar
        %4 = binary.add %accu, %3 : !scalar
        linalg.yield %4 : !scalar
} -> !u_type
```

Figure 5: The interpolation kernel in `linalg` with `base2`.

In both the floating-point and the fixed-point version, the kernel is fed data from a seeded random number generator. Using uniform integer distributions, this generator produces IEEE-754 `binary64` floating-point values using at most p significand bits and ld E exponent bits. This parameterization allows us to mock different example datasets and compare the implementation accuracy. In the floating-point version, our design space is two-dimensional: we can control both the number of significand and exponent bits. In the fixed-point version, we assume a total of 64 bits and control the number of fractional bits (negative exponent) only.

Figure 6 shows the results of this test. The top row shows the floating-point results, the bottom row the fixed-point results. The leftmost plots clearly show how the error shrinks with increased significand precision, reaching 0 for sufficiently large values. In theory, we would expect a diagonal boundary where `frac_bits` $= p$. Because the kernel is heavy on multiplications, the error is dominated by a rightward gradient. The center plot subtracts the mean gradient to reveal the expected top-left to bottom-right diagonal gradient. The fixed-point version is shifted right compared to the floating-point version, as negative exponents consume additional significand bits.

The rightmost plots show the impact of dataset exponent range on the results. For the floating-point case, a critical diagonal exists, beyond which out-of-range exponents are rounded to $\pm\infty$, leading to infinite errors. This plot only compares finite errors to show the fine gradient between the regions. For the fixed-point case, a roughly triangular shape centered at p is visible. This is due to an increase in exponent range requiring more significand bits whilst decreasing the resolution compared to the floating-point case.

4.1.1 *End-to-end flow integration.*

This kernel has previously been evaluated in [21] as part of an end-to-end flow for FPGA acceleration. In that work, quantization using integer data types was also considered. However, custom floating-point implementations were not part of the evaluation.

The flow presented in [21] also permits Bambu [6] HLS for kernel synthesis. This open-source HLS tool provides a custom IP core generator for arbitrary-precision floating-point arithmetic. It accomplishes this by substituting calls to the `libsoftfloat` library. For this experiment, we added a `base2` to `softfloat` lowering in MLIR, which implements `!ieee754` arithmetic using this library, allowing us to plug our kernels directly into the existing end-to-end flow.

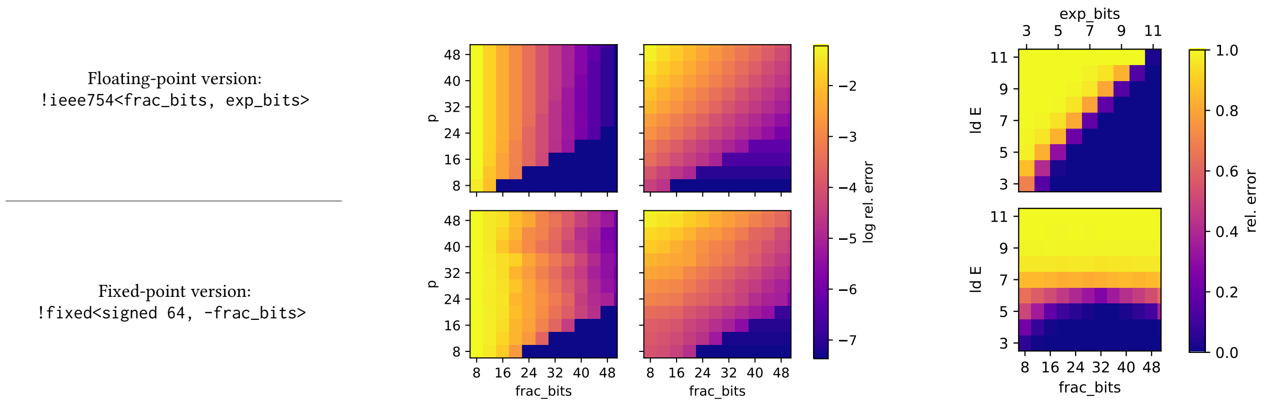
4.2 RRTMGP

For our second experiment, we are looking at part of the RTE-RRTMGP [13] kernels. The Rapid Radiative Transfer Model for GCM solvers (Parallelized), is a modernized implementation of the successful RRTM [14]. Its ancestor is widely used in Global Climate Models (GCMs), such as WRF [20]. Its main function is to compute radiative fluxes.

In this experiment, we consider only the major absorbers optical depth integral. This is broken down into a normalization step and a numerical integration. Normalization adjusts the provided pressure, temperature and gas mixing profiles for a reference atmosphere. The resulting integer part is used for lookups into tabulated absorption coefficients. The optical depth integral is then computed at quadrature points by multi-linear interpolation of the table entries using the fractional part. We test the kernel using a real-world example profile and the most up-to-date constant dataset[3].

The normalization step is particularly interesting because its results have a constrained range and significantly lower precision than `binary64` permits. However, the reference implementation uses IEEE-754 `binary64` everywhere. Specifically, the interpolation fractions are an attractive target for a fixed-point replacement. In addition, a re-quantization of the lookup tables could potentially

[3]Code and data can be obtained at https://github.com/earth-system-radiation/rte-rrtmgp/tree/main/rrtmgp



(a) versus significand precision p
$\mathrm{ld}\,E$ = **exp_bits** = 6

(b) versus exponent range $\mathrm{ld}\,E$
p = **frac_bits** = 32

Figure 6: Error in the interpolation kernel for varying test parameters.

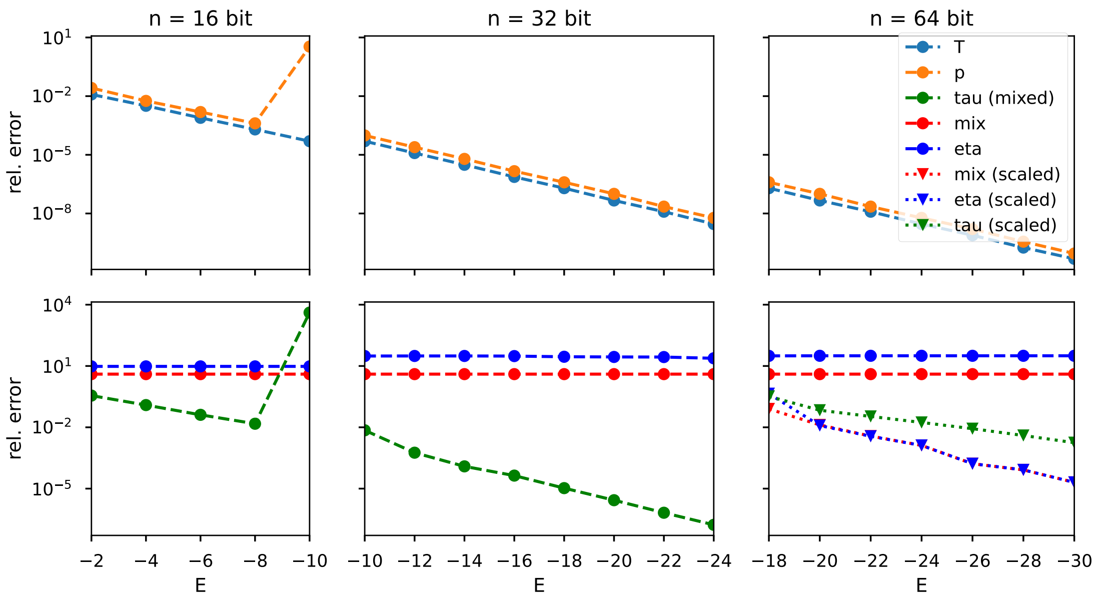


Figure 7: Error in the RRTMGP kernel for varying fixed-point types.

yield huge benefits for an FPGA implementation due to area reduction. In this experiment, we look at how custom fixed-point types impact the interpolation accuracy.

We have tested a variety of 16, 32 and 64 bit fixed-point formats. Figure 7 shows the results of our tests, plotting relative errors against the fixed exponent (i.e., number of fractional bits). The top diagrams show that the linearly interpolated temperature and pressure converge quickly to low errors. The outlier for $n = 16, E = -10$ stems from the fact that the remaining integer bits can no longer represent the pressure range. This confirms our hypothesis that these fractions can be faithfully represented using narrow fixed-point types.

Unfortunately, the same cannot be said for the variables that incorporate gas concentrations. The lower diagrams show the unacceptable performance of our chosen fixed-point formats for these computations. However, this is likely due to scaling issues, with the gas concentrations being in the range of 10^{12}–10^{22} molec./cm^2. In a proof of concept, for the 64 bit case, we have pre-scaled the values and tables by $2^{61} \approx 10^{18}$ to show the impact.

In practice, absolute values can be replaced by mixing ratios in the normalizaton, and different type parameterizations can be used for different variables. In our experiment, we have mixed the 16 and 32 bit formats for interpolation fractions with `binary64` for gas concentrations. The tau (mixed) plot shows their impact on the error of the overall result. In the 64 bit case, we have used the prescaled values to compute tau (mixed) entirely within fixed-point. Clearly, other quantizations or a tightly-consrtained floating-point representation would be more advantageous in this case.

5 CONCLUSION AND OUTLOOK

We presented BASE2, a new target agnostic IR that extends the MLIR framework with (1) portable parametric custom types and generalizable operational semantics, and (2) built-in Poison semantics. We believe BASE2 to be an important generalization within MLIR, especially for hardware design and HLS flows. We demonstrated how BASE2 could be used to generate code from kernels without code modifications to both CPU and FPGA implementations. This enables steering hardware/software co-design with deterministic, simulated execution. We also showed how BASE2 integrates into third-party pipelines to increase the flexibility of existing HLS flows.

BASE2 does not perform quantization, i.e., reason about quantization functions. However, it provides a storage type implementation which was badly needed, in particular by machine learning compiler developers in MLIR. We have begun refactoring the MLIR `tosa` ISA [1] for deep learning by extracting a `StorageTypeInterface`. We plan to demonstrate a drop-in integration of BASE2 into an existing machine learning front-end this way.

In the long run, we believe BASE2 and related efforts can extend to full math support for arbitrary-precision expressions. This would allow for the generation of correct, target-independent math libraries along the vision of [7]. A BASE2-enabled compiler could be the first step to resolve existing accuracy issues, such as in `libc` [4].

In terms of backend integration, we plan to further demonstrate the interoperability BASE2, with other downstream flows. Future perspectives on integration with HLS tend towards pulling as many design decision into the MLIR dialect stack. The CIRCT project is already mature enough that simple kernels, such as our interpolation use case, can be synthesized entirely within MLIR. Aside from CIRCT, we will demonstrate the boostrapping capabilities of BASE2's existing LLVM IR lowering for porting kernels between Vitis and Bambu HLS within the EVEREST project [17].

ACKNOWLEDGMENTS

This work is partially funded by the EU Horizon 2020 Programme under grant agreement No 957269 (EVEREST). We would like to thank the ML compiler team at AMD Cologne for all the ideas and feedback they provided us, especially through collaboration in the open-source MLIR community.

REFERENCES

[1] 2022. TOSA 0.50.0 Specification. https://www.mlplatform.org/tosa/tosa_spec.html
[2] 2023. Brevitas. Xilinx. https://github.com/Xilinx/brevitas
[3] 2023. "CIRCT" / Circuit IR Compilers and Tools. LLVM. https://github.com/llvm/circt
[4] 2023. GNU libc: Errors in Math Functions. https://www.gnu.org/software/libc/manual/2.37/html_node/Errors-in-Math-Functions.html
[5] Quentin Ducasse, Pascal Cotret, Loïc Lagadec, and Robert Stewart. 2021. Benchmarking Quantized Neural Networks on FPGAs with FINN. https://doi.org/10.48550/arXiv.2102.01341 arXiv:arXiv:2102.01341
[6] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *ACM/IEEE Design Automation Conference (DAC)*. https://doi.org/10.1109/DAC18074.2021.9586110
[7] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2 (jun 2007), 13–es. https://doi.org/10.1145/1236463.1236468
[8] John L. Gustafson and Isaac T. Yonemoto. 2017. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (April 2017), 71–86. https://doi.org/10.14529/jsfi170206
[9] IEEE Std 754-2019 2019. *IEEE Standard for Floating-Point Arithmetic (Revision of IEEE 754-2008).* Standard. Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/IEEESTD.2019.8766229
[10] ISO JTC1/SC22/WG21 P0907R4 2018. *Signed Integers are Two's Complement.* Published Proposal. International Organization for Standardization. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0907r4.html
[11] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).
[12] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. https://doi.org/10.48550/ARXIV.2209.05433
[13] Eli Mlawer. 2014. *RRTMGP: A High-Performance Broadband Radiation Code for the Next Decade.* Technical Report. ATMOSPHERIC AND ENVIRONMENTAL RESEARCH INC LEXINGTON MA.
[14] Eli J Mlawer, Steven J Taubman, Patrick D Brown, Michael J Iacono, and Shepard A Clough. 1997. Radiative transfer for inhomogeneous atmospheres: RRTM, a validated correlated-k model for the longwave. *Journal of Geophysical Research: Atmospheres* 102, D14 (1997), 16663–16682.
[15] Luciano Musa. 2008. FPGAS in high energy physics experiments at CERN. In *2008 International Conference on Field Programmable Logic and Applications*. 2–2. https://doi.org/10.1109/FPL.2008.4629896
[16] Tan Nguyen, Samuel Williams, Marco Siracusa, Colin MacLean, Douglas Doerfler, and Nicholas J. Wright. 2020. The Performance and Energy Efficiency Potential of FPGAs in Scientific Computing. In *IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 8–19. https://doi.org/10.1109/PMBS51919.2020.00007
[17] Christian Pilato, Stanislav Bohm, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, Vojtech Cima, Radim Cmar, Dionysios Diamantopoulos, Fabrizio Ferrandi, Jan Martinovic, Gianluca Palermo, Michele Paolino, Antonio Parodi, Lorenzo Pittaluga, Daniel Raho, Francesco Regazzoni, Katerina Slaninova, and Christoph Hagleitner. 2021. EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms. In *IEEE/EDAA Design, Automation & Test in Europe Conference (DATE)*. https://doi.org/10.23919/DATE51398.2021.9473940
[18] UC Berkeley Architecture Research. 2023. Berkeley SoftFloat Release 3e. https://github.com/ucb-bar/berkeley-softfloat-3
[19] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. 2018. CFDlang: High-level Code Generation for High-order Methods in Fluid Dynamics. In *Proceedings of the 3rd International Workshop on Real World Domain Specific Languages (RWDSL 2018)* (Vienna, Austria) *(RWDSL2018)*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3183895.3183900
[20] William C Skamarock, Joseph B Klemp, Jimy Dudhia, David O Gill, Zhiquan Liu, Judith Berner, Wei Wang, Jordan G Powers, Michael G Duda, Dale M Barker, et al. 2019. A description of the advanced research WRF model version 4. *National Center for Atmospheric Research: Boulder, CO, USA* 145, 145 (2019), 550.
[21] Stephanie Soldavini, Karl Friebel, Mattia Tibaldi, Gerald Hempel, Jeronimo Castrillon, and Christian Pilato. 2023. Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics. *ACM Trans. Reconfigurable Technol. Syst.* 16, 2, Article 21 (mar 2023), 34 pages. https://doi.org/10.1145/3563553
[22] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–74. https://doi.org/10.1145/3020078.3021744 arXiv:1612.07119 [cs]