APyTypes: Algorithmic Data Types in Python for Efficient Simulation of Finite Word-Length Effects

1st Mikael Henriksson
Dept. of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden
mikael.henriksson@liu.se

1st Theodor Lindberg
Dept. of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden
theodor.lindberg@liu.se

3rd Oscar Gustafsson
Dept. of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden
oscar.gustafsson@liu.se

***Abstract*—A new Python library, APyTypes, suitable for simulating and exploring finite word-length effects is presented. The library supports configurable bit-accurate fixed- and floating-point types of both scalars and multidimensional arrays and uses a C++ backend to accelerate runtime performance. The underlying design principles of the library are introduced and examples show how it can be used. We argue that APyTypes have significant advantages over existing arithmetic libraries, especially from a hardware design perspective. Finally, some directions for further work are outlined.**

1. Introduction

Custom word length bit-accurate fixed- and floating-point data types in high-level languages are useful tools for algorithm and hardware designers. It allows for quick and easy word-length exploration of complex systems without having to resort to hardware description languages (HDLs) and simulators, where development time is longer and more error-prone. High-level programming languages offer better runtime performance compared to digital hardware simulators, making high-level simulations of digital systems more tractable when doing hardware design exploration, especially in the early design phases. With Python being used more frequently in algorithm design, there is a need for custom bit-accurate data type support in Python to bridge the gap between algorithm design and hardware implementation.

Some earlier efforts to implement custom word length bit-accurate fixed-point types in Python have been made. Notable examples include fxpmath [1] and SPFPM [2] which support arbitrary precision fixed-point arithmetic. However, as they are written in pure Python, their performance may be a bottleneck. In addition, gmpy2 [3], a Python binding for the GNU Multiple Precision (GMP) library [4], supports custom floating-point formats similar to the work presented here, though the underlying binary representation is not readily obtainable. Although fxpmath supports arrays, whereas SPFPM and gmpy2 do not, it currently does not support matrix multiplication. It may be convenient to have fixed- and floating-point support in the same library, which these libraries do not.

The Xilinx HLS Arbitrary Precision Types (AP Types) [5] and Siemens EDA Algorithmic C Datatypes (AC Types) [6] are two examples of open-source C++ software libraries that support custom word length bit-accurate arithmetic using fixed- and floating-point numbers. Both AC Types and AP Types are useful tools for word-length exploration and bit-to-bit co-verification of complex digital systems. In the case of AC Types, the fixed-point data types are usable with Siemens High-Level Synthesis (HLS) tools and support a variety of synthesizable digital signal processing (DSP) and machine learning (ML) digital design blocks. However, AC Types and AP Types cannot be efficiently used in an interactive manner as their C++-design cycle require re-compilation every time a word length changes. Hence, they cannot be integrated with Python in a simple way.

In this work, APyTypes, a Python software package for custom word length bit-accurate fixed- and floating-point arithmetic is presented. APyTypes is designed with custom hardware design in mind and tries to accommodate the needs of hardware designers first. APyTypes uses a backend written in C++ to accelerate arithmetic operations and manage memory efficiently. Furthermore, APyTypes include array types of fixed- and floating-point numbers for efficient tensor arithmetic, and it utilizes Python contexts for even more fine-grained control of implementation details relevant to hardware designers.

The primary design goal of APyTypes is to be a hardware-relevant cross-platform Python software library for algorithm and digital hardware designers, leveraging the flexibility of the rich Python ecosystem. The intended library use case is bit-accurate finite word-length effect design exploration of complex digital systems, and to aid users in creating "golden references" for bit-to-bit co-verification of implemented algorithms using custom word lengths.

The first two authors contributed equally to the work.
This work was financed in part by the ELLIIT strategic research environment through the project D3 ACRE – Approximate Computing Reducing Energy, and the Swedish Foundation for Strategic Research through the project Large Intelligent Surfaces – Architecture and Hardware.
The documentation for APyTypes is found at: https://apytypes.github.io/. Install APyTypes using `pip install apytypes`.

2. The APyTypes Library

APyTypes defines two data types for fixed-point arithmetic, `APyFixed` which is a scalar fixed-point type, and `APyFixedArray` which is a multidimensional array fixed-point type. The fixed-point types support bit-accurate two's complement arbitrary precision arithmetic, currently implemented using mini-GMP [4]. Similarly, APyTypes define bit-accurate scalar and array floating-point types, `APyFloat` and `APyFloatArray`. The floating-point types generalize the convention of the IEEE-754 floating-point standard [7] to custom word lengths, similar to FloPoCo [8]. Currently, the floating-point types are limited to 32 exponent bits and 64 mantissa bits.

All of these four fundamental types can be used in standard arithmetic expressions much like the Python built-in types. Scalars are used like the built-in types `float` and `int`, and array types behave much like the NumPy [9] library array type `numpy.ndarray`. Besides elementary arithmetic operations, some convenience functions are provided: creating objects from built-in floats and strings, casting to floats, comparison operators, shift operators, LaTeX-formatted output, and more. In addition, exact comparisons to native Python types are supported. To obtain an underlying binary representation, the method `to_bits` is used. The APyTypes library also contains varying quantization modes, overflow modes, and Python contexts that give users more fine-grained control over implementation details.

2.1. Fixed-Point Representation

Each `APyFixed` and `APyFixedArray` object has an associated attribute `bits` > 0 that determines the word length of the stored fixed-point number. They also have two attributes `int_bits` and `frac_bits` that determine the location of the binary point. Any two of these three bit attributes can be used when constructing a fixed-point object. All fixed-point arithmetic operations result in a type wide enough to accommodate its result[1]. That way, a hardware designer specifies exactly, using the `cast` method, where and how the word length is reduced in an algorithm. This follows the principle of most libraries, including AC Types, AP Types, and the VHDL Generic Fixed-Point Package [10]. In Table 1, the resulting word length of elementary arithmetic operations is shown. Listing 1 shows creation and arithmetic for `APyFixed`, illustrating resulting word lengths.

2.2. Floating-Point Representation

The implementation of floating-point types follows a generalization of the IEEE 754 standard [7], similar to FloPoCo [8]. A floating-point number is represented using three fields; a sign, a biased exponent, and an integral mantissa. Using this convention, a normalized floating-point number x is represented as

$$x = (-1)^{\texttt{sign}} \times 2^{\texttt{exp}-\texttt{bias}} \times (1 + \texttt{man} \times 2^{-\texttt{man\_bits}}), \quad (1)$$

with an `exp_bits` large exponent, a `man_bits` large mantissa, and a constant `bias`. The bias is given as an optional parameter that can be set to any non-negative value. Special numbers such as subnormals, infinities, and not a number (NaN) are supported and can be detected using Python properties like `is_subnormal`, `is_inf`, and `is_nan`. An `APyFloat` object can be initialized by explicitly setting the five fields in (1) directly, from a bit pattern, or from a native Python floating-point, as shown in Listing 2.

1. Except for division, where an infinite number of bits may be required to represent the result exactly, e.g., $1/3$.

TABLE 1. RESULTING WORD LENGTHS FOR FIXED-POINT OPERATIONS.

Op.	Resulting `int_bits`	Resulting `frac_bits`
$a \pm b$	$\max(a_{\texttt{int\_bits}}, b_{\texttt{int\_bits}}) + 1$	$\max(a_{\texttt{frac\_bits}}, b_{\texttt{frac\_bits}})$
$a \times b$	$a_{\texttt{int\_bits}} + b_{\texttt{int\_bits}}$	$a_{\texttt{frac\_bits}} + b_{\texttt{frac\_bits}}$
$a \,/\, b$	$a_{\texttt{int\_bits}} + b_{\texttt{frac\_bits}} + 1$	$a_{\texttt{frac\_bits}} + b_{\texttt{int\_bits}}$
$-a, \lvert a \rvert$	$a_{\texttt{int\_bits}} + 1$	$a_{\texttt{frac\_bits}}$

Listing 1. Initialization and operations using `APyFixed`.

```
import APyFixed from apytypes import QuantizationMode

a = APyFixed(7, bits=5, int_bits=2)  # 0.875 = 7/2**(5-2)
b = APyFixed.from_float(3.5, int_bits=4, frac_bits=1)
c = a + b  # APyFixed(35, bits=8, int_bits=5)  =  4.375
d = a - b  # APyFixed(235, bits=8, int_bits=5) = -2.625
e = a * b  # APyFixed(49, bits=10, int_bits=6) =  3.0625
f = a / b  # APyFixed(32, bits=11, int_bits=4) =  0.25

# Cast variable to different format (h = 1.125)
g = APyFixed.from_str("1.11829576904296875",
                      int_bits=2, frac_bits=18)
h = g.cast(frac_bits=4, quantization=QuantizationMode.RND)
```

Listing 2. Initialization options for `APyFloat`.

```
from apytypes import APyFloat

# Three identical floating-point values (1.75)
a = APyFloat(sign=0, exp=15, man=3, exp_bits=5, man_bits=2)
b = APyFloat.from_bits(0b0_01111_11, exp_bits=5, man_bits=2)
c = APyFloat.from_float(1.75, exp_bits=5, man_bits=2)
```

Unlike `APyFixed`, and as is common for floating-point, arithmetic results involving `APyFloat` inherit the word length of its operands. Conversions between different `APyFloat` formats are performed using the `cast` method with the target exponent and mantissa bit widths specified, and an optional parameter for bias and quantization mode.

APyTypes supports operations with operands representing different floating-point formats. Using mixed floating-point arithmetic with `APyFloat`, the resulting exponent and mantissa bit widths equal the largest of each of its operands, as demonstrated in Listing 3. Smaller formats are thus conveniently extended while quantizing larger formats is performed explicitly through the `cast` method.

Listing 3. Operations using `APyFloat`.

```
from apytypes import APyFloat
from apytypes import QuantizationMode

a = APyFloat.from_float(9.625, exp_bits=4, man_bits=6)
b = APyFloat.from_float(2.125, exp_bits=4, man_bits=6)
c = APyFloat.from_float(-2.25, exp_bits=3, man_bits=8)

# Word length is kept for same input format
# APyFloat(sign=0, exp=10, man=30, exp_bits=4, man_bits=6)
d = a + b  # 11.75

# Word length is increased for mixed input formats
# APyFloat(sign=1, exp=11, man=90, exp_bits=4, man_bits=8)
e = a * c  # -21.625

# APyFloat(sign=1, exp=11, man=22, exp_bits=4, man_bits=6)
mode = QuantizationMode.TIES_ZERO
f = e.cast(man_bits=6, quantization=mode) # -21.5
```

2.3. Array Types

The library array types `APyFixedArray` and `APyFloatArray` are used to hold multiple scalars of the same word length in one contiguous memory layout. These types make an effort to perform redundant arithmetic, like linear algebra operations, efficiently. Like arrays in NumPy [9], APyTypes arrays have an associated `shape` that determines the legality and mode of operation for standard linear algebra operations between arrays.

It is possible to convert an APyTypes array to and from a NumPy array of floating-points, using `to_numpy` and `from_float` respectively. This makes it convenient to use `APyFixedArray` and `APyFloatArray` with other standard Python library tools. For example, it is possible to plot the data of `APyFixedArray` and `APyFloatArray` using Matplotlib 3.6 or higher [11] or draw random data from NumPy random variables. This is shown in Listing 4.

Listing 4. Matrix multiplication using `APyFixedArray` with random data drawn from NumPy, plotted using Matplotlib.

```
from apytypes import APyFixedArray
import matplotlib.pyplot as plt
import numpy as np

# Fixed-point matrix (100 x 100) of normal distributed data
An = np.random.normal(1, 2, size=(100, 100))
A = APyFixedArray.from_float(An, bits=10, int_bits=3)

# Fixed-point vector of uniformly distributed random data
bn = np.random.uniform(0, 1, size=100),
b = APyFixedArray.from_float(bn, int_bits=4, frac_bits=5)

# Fixed-point matrix-vector multiplication
# Resulting word length: int_bits=14, frac_bits=12
c = A @ b.T

# Plot resulting APyFixedArray using Matplotlib
plt.plot(c)
```

The word length of array-type arithmetic scales in a similar fashion to its scalar counterpart. Importantly, for matrix multiplication with `APyFixedArray`, the word length grows logarithmically with matrix size and not linearly.

2.4. Accumulator Contexts

By using APyTypes, a user has full control of the word length between arithmetic operations through static word-length inference and explicit word-length casting. As hardware designers are typically also interested in the word lengths used inside of standard algebraic operations, APyTypes supports hardware-aware contexts that give even more control to its users.

The dedicated accumulator context classes make it possible to control the size of accumulators used when performing inner products or matrix multiplications. Continuing the code of Listing 4, a user can control the matrix multiplication-accumulator word lengths using the code in Listing 5.

Listing 5. Matrix-vector multiplication using an accumulator context.

```
from apytypes import APyFixedAccumulatorContext

# Perform matrix-vector multiplication using full precision
c = A @ b.T  # Accumulator size: int_bits=14, frac_bits=12

# Perform matrix-vector multiplication using a narrow
# accumulator with fixed-point rounding (ties towards
# infinity) after each scalar multiplication
m = QuantizationMode.RND
with APyFixedAccumulatorContext(frac_bits=9, quantization=m):
    d = A @ b.T
```

There is a similar context which is used to control the word length of floating-point intermediate results.

2.5. Quantization

One of the most important aspects of finite number arithmetic is how quantization is performed. As the need for quantization is application-depended, this for a hardware designer often entails spending time implementing a set of quantization modes and running HDL simulations for evaluation. APyTypes accelerates this process by supporting several different quantization modes and having an API that can readily be parametrized. Table 2 shows the currently supported quantization modes. As noted, there are often aliases to support the ”common” name used, which sometimes differ between fixed-point and floating-point arithmetic.

In floating-point arithmetic, the result will be quantized using the currently selected quantization mode. The default mode `TIES_EVEN` can be changed using the static function `set_float_quantization_mode` or, preferably, by using what APyTypes refers to as a quantization context. Within a quantization context, the current quantization mode is changed temporarily for a specific section of the code, and it will automatically be reverted when the context is exited, as shown in Listing 6. Quantization contexts can be also nested, allowing for a high degree of freedom in simulations where multiple quantization modes are used.

For stochastic rounding, APyTypes automatically generate a pseudo-random seed when loaded, but the random number engine of APyTypes can also be seeded by the user via the Python API as shown in Listing 7. This makes it possible to create deterministic runs and facilitates analysis.

TABLE 2. SUPPORTED QUANTIZATION MODES, QUANTIZATIONMODE.

Quantization mode	Name (Alias)
Round to nearest, ties to even	RND_CONV (TIES_EVEN)
Round to nearest, ties to odd	RND_CONV_ODD (TIES_ODD)
Round to nearest, ties to away	RND_INF (TIES_AWAY)
Round to nearest, ties to zero	RND_ZERO (TIES_ZERO)
Round to nearest, ties to positive infinity	RND (TIES_POS)
Round to nearest, ties to negative infinity	RND_MIN_INF (TIES_NEG)
Round towards positive infinity	TRN_INF (TO_POS)
Round towards negative infinity	TRN (TO_NEG)
Round away from zero	TRN_AWAY (TO_AWAY)
Round towards zero	TRN_ZERO (TO_ZERO)
Magnitude truncation (add sign-bit)	TRN_MAG
Jamming (von Neumann rounding)	JAM
Unbiased jamming	JAM_UNBIASED
Stochastic rounding with equal probability	STOCH_EQUAL
Stochastic rounding with weighted probability	STOCH_WEIGHTED

Listing 6. Using APyFloatQuantizationContext.

```
from apytypes import QuantizationMode
from apytypes import APyFloatQuantizationContext

# Addition using the default round to nearest, ties even
c = a + b

# Addition using round towards zero
with APyFloatQuantizationContext(QuantizationMode.TO_ZERO):
    d = a + b

# Subtraction using the default round to nearest, ties even
e = a - b
```

2.6. Jupyter Notebook Integration

LaTeX-formatted output is available in enabled environments, Jupyter Notebooks and Spyder to mention two. In such environments, the representation of a fixed-point number could look like

$$\frac{35}{2^3} = 4.375, \tag{2}$$

and a floating-point number could be displayed as

$$\left(1 + \frac{9}{2^4}\right) 2^{18-15} = 25 \times 2^{-1} = 12.5. \tag{3}$$

3. Future Plans

Future development plans for APyTypes include explicit support for unsigned numbers. Configuring floating-point numbers to, e.g., not support subnormal numbers or infinities is planned, as well as the simplified FloPoCo floating-point format [8]. Support for generating test and verification data in suitable formats should be added. A larger effort is to generate code suitable for HLS tools.

Listing 7. Seeding the quantization context.

```
# Calculation with stochastic weighted quantization
# and an initial seed of 0x1234
seed = 0x1234
mode = QuantizationMode.STOCH_WEIGHTED
with APyFloatQuantizationContext(mode, seed):
    e = a + b
```

4. Conclusions

APyTypes is a new Python software library for simulation of finite word-length effects, and bit-to-bit co-verification of digital systems. Developed with hardware designers in mind, it supports bit-accurate fully custom fixed- and floating-point arithmetic of both scalars and multidimensional array types. By moving bit-accurate hardware simulations from HDL and digital simulators to a high-level Python library using a C++ backend, APyTypes has the potential to greatly accelerate both development time and simulation run-time of complex digital systems, without compromising the intricate details offered by HDLs. By leveraging Python contexts, APyTypes give hardware designers fine-grained control of arithmetic operations using the introduced accumulator- and quantization contexts. Finally, APyTypes offer a variety of quantization modes, both classical ones and stochastic ones, that can be paired with the contexts for easier exploration of the effect caused by different quantization modes.

References

[1] F. Alcaraz, "fxpmath," 2024. [Online]. Available: https://github.com/francof2a/fxpmath

[2] R. Penney, "Simple Python fixed-point module (SPFPM)," 2023. [Online]. Available: https://github.com/rwpenney/spfpm

[3] gmpy2 authors, "gmpy2," 2024. [Online]. Available: https://github.com/aleaxit/gmpy

[4] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library, version 6.3.0*, 2023. [Online]. Available: http://gmplib.org/

[5] Xilinx, "HLS arbitrary precision types library," 2019. [Online]. Available: https://github.com/Xilinx/HLS_arbitrary_Precision_Types

[6] Siemens EDA, "Algorithmic C (AC) datatypes," 2022. [Online]. Available: https://github.com/hlslibs/ac_types

[7] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[8] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Des. Test. Comput.*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[9] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.

[10] "IEEE standard for VHDL language reference manual," *IEEE Std 1076-2019*, pp. 1–673, 2019.

[11] T. A. Caswell, A. Lee, M. Droettboom, E. S. de Andrade, T. Hoffmann, J. Klymak, J. Hunter, E. Firing, D. Stansby, N. Varoquaux, J. H. Nielsen, B. Root, R. May, P. Elson, J. K. Seppänen, D. Dale, J.-J. Lee, D. McDougall, A. Straw, P. Hobson, hannah, O. Gustafsson, G. Lucas, C. Gohlke, A. F. Vincent, T. S. Yu, E. Ma, S. Silvester, C. Moad, and N. Kniazev, "matplotlib/matplotlib: Rel: v3.6.0," Sep. 2022.