Verification of Approximate Hardware Designs with ChiselVerify

Hans Jakob Damsgaard, Aleksandr Ometov, Jari Nurmi
[*] *Electrical Engineering Unit, Tampere University*, Finland, {first.last}@tuni.fi

***Abstract*—Many popular applications show resilience to computational errors. Approximate Computing (AxC) exploits this to reduce their execution time and energy consumption by introducing approximations in software and hardware. Using AxC raises new challenges to ensure that hardware designs satisfy their demands before deployment, which hardware designers address by spending significant efforts on verification flows for their designs. However, there exist no tools for verifying approximate hardware designs, meaning that designers must replicate code to keep track of circuit outputs and subsequently compute relevant error metrics. We aim to solve this issue with a library that abstracts away port sampling and error computations behind a simple interface. With the library, designs can retrieve error metric values and constraint satisfaction results with only a few extra lines of code. We demonstrate these features with code examples and by characterizing a collection of inexact adders and multipliers and an approximate matrix-vector multiplier.**

***Index Terms*—approximate computing, chisel, electronic design automation, hardware verification**

I. Introduction

Historically, computers solved complex but well-defined problems whose outputs were judged on correctness. Moore's law and Dennard scaling meant that the underlying hardware's performance grew to match increasing problem complexities [1]. The end of these trends complicates satisfying compute requirements of popular applications such as Machine Learning (ML), Digital Signal Processing (DSP), and multimedia progressively [2]. This issue is pertinent to energy-constrained devices operating at the Edge of the internet, but also to datacenters where the common solution is to integrate power-hungry application-specific accelerators; something that is infeasible in Edge devices [2]. As the Internet of Things (IoT) is expected to grow in popularity and scale, there is a need to develop and evaluate alternative techniques to improve performance and reduce energy consumption.

The aforementioned algorithms fortunately show resilience to constrained computational errors. Moreover, their outputs are often consumed by humans who evaluate them by their acceptability rather than correctness. This allows for exploiting their error resilience to reduce their execution time or energy consumption [1], [3], [4]. Within the AxC domain, this is done by introducing approximations at various system levels ranging from the *application* layer – in software – through the *architecture* layer down to the *circuit* layer – both being in hardware. Until recently, most research has focused on these layers individually, though expectedly much greater effects can be achieved by optimizing across them [2].

The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation Programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, http://www.apropos-itn.eu/). An early version of this work was presented at the Workshop on Approximate Computing (WAPCO), co-located with the HiPEAC 2023 conference.

In this paper, we maintain a focus on AxC in hardware. We note that unless approximate hardware designs are made to be reconfigurable at run-time, application-layer approximations have one major benefit over them: they can easily be adjusted, even after deployment. Hardware is much more costly to replace and must, therefore, be thoroughly verified [1], [5], for example, through simulation with *constrained-random* techniques [6] or *fuzzing* [7]. Yet, to the best of our knowledge, despite most papers employing similar evaluation strategies and error metrics [2], there exist no tools for performing this on generic hardware designs; prior work [7] being limited to C-based Hardware Description Language (HDL)s.

We target this lack of tools for AxC and propose a library for verifying generic approximate hardware designs. Integrated with `ChiselVerify` [6], it provides access to port sampling and error computation, abstracted away behind a very simple interface. Using our tool, hardware designers can easily write or adapt tests applying a selection of error metrics – including custom ones written by the developer following our library's structure. Our library extends upon and is fully compatible with ChiselVerify's existing constrained-random verification, functional coverage, and bus functional model features.

The rest of the paper is structured as follows: Section II covers necessary background and related work. We present our library and motivate its implementation details with code examples in Section III. Next, we demonstrate the library's functionality by characterizing a collection of approximate adders and multipliers and by verifying an approximate matrix-vector multiplier in Section IV. Finally, Section V concludes the paper and outlines directions for future work.

II. Related Work

Before covering our proposed verification library, we find it necessary to describe its context. Specifically, we outline some common AxC techniques, introduce the Chisel framework, and describe what hardware verification is and what it entails in the AxC scope. For more details, interested readers may refer to [2], [4], [6], [8].

A. Approximate Computing Techniques

In hardware, AxC techniques are distributed across the circuit and architecture levels. At the circuit level, the common ones are voltage over-scaling, inexact arithmetic, and function approximation. Voltage over-scaling can bring vast energy savings but requires optimizing designs at lower-than-Register Transfer Level (RTL). Inexact adder designs include Ripple-Carry Adders (RCAs) with approximate full-adders and Carry-Lookahead Adders (CLAs) with speculative carry prediction [1]. Most inexact multipliers have their partial product trees truncated or approximately compressed [9], [10]. Function approximation aims at substituting complex, time-consuming kernels by, e.g., simple ML models [11]. These are examples of broadly applicable techniques that may require stringent design-time or run-time quality management [2]. We use several inexact arithmetic units for our demonstration later.

Many papers have explored integrating AxC into various architectures. Some focus on general-purpose architectures for which approximations include inexact arithmetic, memoization, and unreliable inter-core communication and cache coherence. Regrettably, the adoption of these techniques is hindered by their need for Instruction Set Architecture (ISA) support and limited benefits due to execution flow overheads in modern processors [2]. Other papers focus on application-specific architectures for ML, DSP, and video processing tasks [4]. The applied techniques are similar but often lead to much greater gains [2]. Unlike in the other application domains, work on ML stands out as it frequently applies multiple AxC techniques in software and hardware concurrently [9].

B. Hardware Design with Chisel

Chisel is a hardware *construction* language embedded in Scala [8]. It is an RTL language that simplifies the design process compared with traditional HDLs through abstraction and flexible programming. Being embedded in Scala permits making use of object-oriented and functional programming constructs for hardware construction; a feature yet to be available in the design context in either (System)Verilog or VHDL. Abstracting away the clock and reset signals from single-clock designs brings further simplicity and avoids verbosity. Combined, these features vastly reduce the complexity of designing capable, flexible hardware generators.

In order to ensure compatibility with common synthesis flows, Chisel's compiler backend first lowers designs into an intermediate representation in the FIRRTL language and later into a subset of (System)Verilog [8], [12], [13]. The backend permits legacy designs in (System)Verilog to be included directly as black boxes. Aided by the `ChiselTest` library, this enables simulation of designs using the Treadle engine directly on the intermediate FIRRTL representation or other tools, such as Verilator, on the output (System)Verilog [14]. As other compilers, the backend can be extended with custom optimization or instrumentation passes.

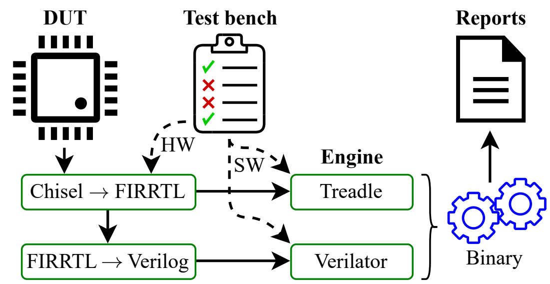

Fig. 1: The dynamic verification flow of a Chisel design using ChiselVerify and either Treadle or Verilator.

C. Hardware Verification

Verification refers to the process of testing a hardware design, a Device Under Test (DUT), prior to tape-out [15]. Dynamic verification is the most common strategy. It involves specifying a *test bench* that manages the DUT during a simulation, providing it with inputs and comparing its outputs against a golden model, as illustrated in Fig. 1. Both Verilog and VHDL support these features and provide a limited set of non-synthesizable constructs for verification. SystemVerilog extends Verilog with several object-oriented programming features that permit users to, e.g., define custom data types, collect coverage information, and generate random values [15]. However, unlike in Chisel, these features can only be used for verification purposes, not for hardware description. The Universal Verification Methodology (UVM) builds atop these features and defines a way of writing composable and reusable test benches in SystemVerilog [16]. UVM suffers from inherent verbosity and a steep learning curve, but its promise of reusability and support for advanced techniques like constrained-random verification [17] and fuzzing [18] means it is set to become the industry standard [6].

Within the Chisel context, dynamic verification is enabled by the `ChiselTest` library [19], which simplifies interfacing with various simulation engines, including Treadle and Verilator [14]. As such, it lets users `poke` values on inputs, `peek` or `expect` (i.e., assert) values on outputs, and advance time by `stepping` the clock. The `ChiselVerify` library extends upon these basic features with support for functional coverage, constrained random verification, bus functional models, and timed assertions [6]. Unfortunately, neither library provides any features for verifying the error characteristics of approximate hardware designs.

While we focus on dynamic verification in this paper, we find it necessary to mention formal techniques that aim at proving, and thus guaranteeing, properties of a module. Contrary to simulation-based verification styles, formal techniques are yet to be well supported for Chisel. Although libraries like Chisel's own `formal` allow for checking simple temporal properties, they require modifications directly in the module's description and, thus, do not work with black boxes [6]. Similarly, recent work remains limited in the languages it supports [20].

TABLE I: Currently implemented error metrics, their types and result types, and their formulas.

Name	I/H	A/R	Formula		
Error Distance	I	A	$\mathrm{ED}(a,b)$	$=$	$\lvert b-a\rvert$
Soft Error	I	A	$\mathrm{SE}(a,b)$	$=$	$\mathrm{ED}(a,b)^2$
Relative ED	I	R	$\mathrm{RED}(a,b)$	$=$	$\frac{\mathrm{ED}(a,b)}{b}$
Mean ED	H	A	$\mathrm{MED}(\mathbf{v},\mathbf{u})$	$=$	$\frac{\sum_{i=0}^{\mathrm{len}(\mathbf{v})-1}\mathrm{ED}(\mathbf{v}_i,\mathbf{u}_i)}{\mathrm{len}(\mathbf{v})}$
ED Variance	H	A	$\mathrm{VED}(\mathbf{v},\mathbf{u})$	$=$	$\frac{\sum_{i=0}^{\mathrm{len}(\mathbf{v})-1}(\mathrm{ED}(\mathbf{v}_i,\mathbf{u}_i)-\mathrm{MED}(\mathbf{v},\mathbf{u}))^2}{\mathrm{len}(\mathbf{v})}$
ED Standard Deviation	H	A	$\mathrm{SDED}(\mathbf{v},\mathbf{u})$	$=$	$\sqrt{\mathrm{VED}(\mathbf{v},\mathbf{u})}$
Mean RED	H	R	$\mathrm{MRED}(\mathbf{v},\mathbf{u})$	$=$	$\frac{\sum_{i=0}^{\mathrm{len}(\mathbf{v})-1}\mathrm{RED}(\mathbf{v}_i,\mathbf{u}_i)}{\mathrm{len}(\mathbf{v})}$
Mean SE	H	R	$\mathrm{MSE}(\mathbf{v},\mathbf{u})$	$=$	$\frac{\sum_{i=0}^{\mathrm{len}(\mathbf{v})-1}\mathrm{SE}(\mathbf{v}_i,\mathbf{u}_i)}{\mathrm{len}(\mathbf{v})}$
Root MSE	H	R	$\mathrm{RMSE}(\mathbf{v},\mathbf{u})$	$=$	$\sqrt{\mathrm{MSE}(\mathbf{v},\mathbf{u})}$
Hamming Distance	I	A	$\mathrm{HD}(a,b)$	$=$	$\mathrm{popcount}(a\oplus b)$

Normal letters a and b are scalars. Bold letters $\mathbf{v}$ and $\mathbf{u}$ are vectors. The function $\mathrm{len}(\mathbf{v})$ gives the number of elements in $\mathbf{v}$. All `HistoryBased` metrics assume that $\mathrm{len}(\mathbf{v}) = \mathrm{len}(\mathbf{u})$.

III. Our Verification Library

With the necessary background covered, we turn our attention to our proposed verification library that is written in Scala and targets approximate hardware designs in Chisel or Verilog [8]. Our library is integrated into ChiselVerify and is inspired by its existing functional coverage features, also relying on ChiselTest's testing features [6]. It provides elements, `Watchers`, for watching either pairs of module ports – one being approximate and the other exact – or a standalone approximate port whose exact reference values are computed in software, for subsequent reporting on, or verification of, a collection of popular error `Metrics` (identified in [2]). Our library retains Chisel's and ChiselTest's aims at simplicity through abstraction.

A. Error Metrics

Currently, we have implemented the error metrics listed in Table I. As shown, we associate each metric with two traits. The purpose of this is two-fold: 1) it enforces similar metrics to define common methods (in particular, a `compute` method), and 2) users may find them informative when working with the available metrics. Specifically, we consider `Instantaneous` metrics that are computed on a single sample and its reference in contrast to `HistoryBased` metrics that are computed on sequences of samples and references, both extending a base `Metric` trait. And in parallel, `Absolute` metrics that return non-normalized results versus `Relative` metrics that return normalized or otherwise relative results (e.g., Error Rate (ER)), both extending a `MetricResult` trait. For example, Error Distance (ED) is `Instantaneous Absolute` whereas Mean RED (MRED) is `HistoryBased Relative`. As users may wish to compute `Instantaneous` metrics on sequences of data, we include methods to do so directly in the base trait.

For verification purposes, all metrics take an optional maximum value argument, which may be left out if a metric is used only for tracking. Additionally, all metrics are provided with so-called *companion objects*, allowing users to create them without explicitly calling their constructor and to apply them to data outside a `Watcher`. As an example, consider the companion object for the ED metric in Listing 1. Users wishing to implement custom error metrics can do so by extending either `Instantaneous` or `HistoryBased` and mixing in either `Absolute` or `Relative`. Companion objects are optional.

Listing 1 Companion object for the ED metric.

```
case object ED {
  /** Create an unconstrained ED metric */
  def apply(): ED = new ED
  /** Create a constrained ED metric */
  def apply(maxVal: Double): ED = new ED(Some(maxVal))
  /** Create an unconstrained ED metric and apply
      it to the given samples */
  def apply(v1: BigInt, v2: BigInt): Double =
    (new ED).compute(v1, v2)
}
```

B. Sampling

Our library abstracts away sampling and storage using two types of `Watchers`: `Trackers` and `Constraints`. For simplicity, we keep most of the library private, as shown in Fig. 2, providing only the following public end-points to users:

- `track(approxPort, metrics*)` defining a `PlainTracker` and `track(approxPort, exactPort, metrics*)` defining a `PortTracker`. Both `Tracker` types can sample the approximate port relative to either a user-provided reference value or another module port producing exact results and report on the metrics provided, but not verify that their maximum values, if any, are met.
- `constrain(approxPort, metrics*)` defining a `PlainConstraint` and `constrain(approxPort, exactPort, metrics*)` defining a `PortConstraint`. These `Constraint` types extend upon their equivalent `Trackers` by also supporting verification, requiring all given metrics to have defined maximum values.
- `ErrorReporter(watchers*)` is the main class of the library that maintains given `Trackers` and `Constraints`, sampling their ports and reporting their results.

As for the metrics, composing `Trackers` and `Constraints` hierarchically ensures that they share common methods and functionality. Most importantly, all classes inheriting from `Watcher` define a `sample(expected)` method, which `peeks` the approximate port and stores its value with the given expected value, and a `report()` method, which computes the comprised metrics on stored samples and returns the results in a printable `Report`. In addition to these, `PortBased Watchers` define a `sample()` method that `peeks` the exact port as the expected value, and `Constraints` define a `verify()` method that computes the comprised metrics and returns true if they all satisfy their maximum values and false otherwise. For consistency, the `ErrorReporter` defines methods of the same names that each iteratively calls its equivalent in the comprised `Watchers`. Expected values are

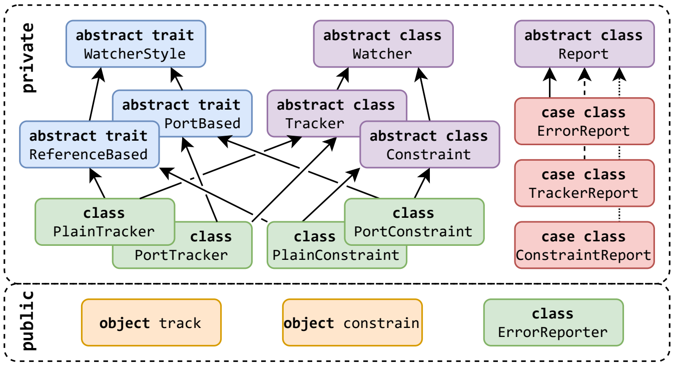

Fig. 2: Overview of the classes and objects in our library, excluding error metrics.

passed to `PlainTrackers` and `PlainConstraints` through the `ErrorReporter`'s `sample(expected)` method.

The library's reporting module implements case classes for `Trackers` and `Constraints` separately and composes their contents together in the generic `ErrorReport` case class. These classes differ in what data they include in their reports: `TrackerReports` state the mean and maximum values of `Instantaneous` metrics and the value of `HistoryBased` metrics. `ConstraintReports` instead state if a metric is satisfied and otherwise reports the maximum value and its index for `Instantaneous` metrics or the value of `HistoryBased` metrics. We include a sample report in Section III-D.

C. Storage

Internally, a `Watcher` must store samples of approximate ports and their expected values. The type of this storage is determined by the input and output types used in the metrics, leading us to carefully consider the available Scala data types for these. We decided on `BigInts` for all inputs, as their flexible size means they work well with arbitrary module IO, and `Doubles` for all outputs, regardless of metric types. In our experience, `Doubles` have sufficient dynamic range and precision to represent the values that the implemented metrics produce. Moreover, consistently using only two data types is well in accordance with our aim for simplicity.

A downside of `BigInts` is their, potentially, large memory footprint. In anticipation of this becoming an issue in large-scale tests, we implement pre-emptive computation and cache the results in the `Watchers`. By default, the sample storage is collapsed into a single data point every 1024 samples, yet this frequency is user-adjustable through the `track` and `constrain` functions. Applying the same strategy to the cache, maintaining sufficient information for later reporting, enables us to hypothetically support infinitely long tests[1]. When collapsing the cache, we rely on the central limit theorem and compute weighted arithmetic and geometric means

[1]We examined memory dumps of some long, crashing tests and found ChiselTest's threaded backend to be the culprit as it stores costly timestamps for each clock step. We avoid these crashes by using the `NoThreadingAnnotation`.

Listing 2 Test of an approximate adder.

```
"Approximate adder" should "verify with software model" in {
  test(new ApproximateAdder(32, 8)) { dut =>
    val er = new ErrorReporter(
      constrain(dut.io.s, RED(.1), MRED(.025)),
      constrain(dut.io.cout, ER(.01))
    )
    val mask = (BigInt(1) << 32) - 1
    val rng  = new Random(42)
    for (_ <- 0 until 10000) {
      val (a, b) = (BigInt(32, rng), BigInt(32, rng))
      val (cout, s) = (((a + b) >> 32) & 1, (a + b) & mask)
      dut.io.a.poke(a.U)
      dut.io.b.poke(b.U)
      er.sample(Map(dut.io.cout -> cout, dut.io.s -> s))
    }
    er.verify() should be (true)
    println(er.report())
} }
```

Listing 3 Report from the test in Listing 2 reformatted for readability here.

```
======================= Error report =======================
Constraint on port cout has results:
- History-based ER(Some(0.01)) metric is satisfied with
  error 0.0!
Constraint on port s has results:
- Instantaneous RED(Some(0.1)) metric is satisfied with
  maximum error 7.90743868572043E-5!
- History-based MRED(Some(0.025)) metric is satisfied with
  error 2.0970120714029623E-7!
============================================================
```

for `Absolute` and `Relative` metrics, respectively. Naturally, when operating with floating-point data, this approach leads to slight round-off errors, yet we verify that for practical use cases, results are less than 0.5% off a non-cached baseline.

D. Using the Library

Using the library within a test requires instantiating an `ErrorReporter` and passing it any number of `Trackers` and `Constraints` with related error metrics. Subsequently, the registered ports must be sampled by manually calling the `ErrorReporter`'s `sample(expected)` method. At the end of a test, the `report()` and `verify()` methods can be used to produce an error report and verify all the specified `Constraints`. We note that as ChiselTest permits testing only one module at a time, using `PortTrackers` and `PortConstraints` requires defining a top-level module that contains both the approximate and exact modules. This can be done within the test class with little overhead.

As an example, consider the test of a 32-bit adder whose least significant eight sum bits are approximated by carry-free XORs, shown in Listing 2. For illustrative purposes, we use both an `Instantaneous` and a `HistoryBased` metric, include code for random input generation and expected value computation, but leave out the class wrapping the test. The resulting report is shown in Listing 3. Note that the measured ER of 0.0 on the adder's carry output, `cout`, may be an artifact stemming from the limited number of test cases executed.

TABLE II: Measured error and post-synthesis area metrics for the selected exact and approximate adders.

Identifier	Area [LUTs]	Error metrics ER	MED	SDED	MRED
RCA(32)	32	–	–	–	–
CSA(32, 8)	67	–	–	–	–
CLA(32, 8)	56	–	–	–	–
LOA(32, 12)	32	98.4%	1.02e+3	1.02e+3	4.20e−6
LAXA1(32, 12)	26	100%	4.10e+3	1.67e+3	2.01e−5
LAXA2(32, 12)	39	97.9%	1.20e+3	1.17e+3	6.20e−6
LAXA3(32, 12)	45	87.2%	1.02e+3	1.23e+3	3.26e−6
OFLOCA(32, 12, 4)	26	99.8%	6.39e+2	4.96e+2	3.33e−6
OFLOCA(32, 16, 8)	22	99.9%	2.74e+4	8.59e+6	5.71e+0
GeAr(32, 6, 2)	35	41.2%	1.67e+7	1.89e+8	1.25e+1
GeAr(32, 8, 8)	49	0.38%	5.76e+4	1.05e+7	5.97e+0

TABLE III: Measured error and post-synthesis area metrics for the selected exact and approximate unsigned multipliers.

Identifier	Area [LUTs]	Error metrics ER	MED	SDED	MRED
R2M(32)	1347	–	–	–	–
REC(32)	1662	–	–	–	–
R2M(32, RTrunc(2))	1270	100%	2.88e+18	2.36e+18	3.61e+ 0
R2M(32, RTrunc(4))	1184	100%	3.58e+18	2.84e+18	8.16e+ 0
R2M(32, CTrunc(16))	1188	99.9%	2.46e+ 5	9.54e+ 4	1.70e−12
R2M(32, CTrunc(32))	706	100%	3.33e+10	9.06e+ 9	1.48e− 7
R2M(32, ORCmp(16))	1244	99.5%	1.81e+ 5	9.39e+ 4	1.24e−12
R2M(32, ORCmp(32))	968	99.9%	2.90e+10	9.06e+ 9	1.17e− 7
R2M(32, Miscnt(16))	1296	99.9%	1.90e+ 5	9.56e+ 4	1.30e−12
R2M(32, Miscnt(32))	999	100%	2.92e+10	9.27e+ 9	1.20e− 7
REC(32, Approx(16))	1404	92.4%	2.59e+17	7.88e+17	1.18e+ 0

IV. Demonstration

To demonstrate the capabilities of our library, we use it to characterize a collection of inexact adders and multipliers and an approximate constant matrix-vector multiplier that employs the computation coding scheme proposed in [21]. These designs are illustrative examples of circuit and architecture level AxC techniques, respectively, for which we also provide Chisel implementations [22], [23].

A. Inexact Arithmetic Units

First, we consider four different adder designs, some of which we provide several configurations of: 1) the well-known *lower-part OR adder* (LOA) that substitutes some least significant full adders by OR gates; 2) an adder we denote by *lower-part approximate adder* (LAXA) that uses inexact full adders [24] instead of OR gates; 3) an extension of the LOA concept with a simple error compensation scheme in the *lower-part constant-OR feedback adder* (OFLOCA) [25]; and 4) the segmented *generic accuracy-configurable adder* (GeAr) with overlapping sub-adders [26]. For each adder, we compute 1,000,000 random sums and track only the sum output, ignoring the adder's carry-out.

Table II lists the results of this characterization and synthesis results achieved for an Artix-7 FPGA with Xilinx Vivado 2022.2. We only report Lookup Table (LUT) utilization as all adders are purely combinational. The listed parameters indicate the width, number of stages, or widths of the adders' sub-elements. Refer to [22] for further specifications. As the LOA, LAXA, and OFLOCA adders approximate the RCA and GeAr resembles a Carry-Select Adder (CSA) or a CLA, we include area metrics for such exact ones too. The results show how different adders achieve different error metrics. The overall trend is that approximating many Least-Significant Bits (LSBs) or reducing carry chain lengths near the Most-Significant Bits (MSBs) greatly increases errors. Consistent with prior work [2], some approximate adders trade off increased area for reduced errors: notably, GeAr has a low ER.

Next, having examined some inexact adders, we consider two different multiplier architectures for unsigned numbers. Firstly, a radix-2 array multiplier that constructs a pyramid-shaped array of partial product bits and *compresses* (adds) them in a tree. We approximate this multiplier by truncating some least significant rows or columns of partial product bits (denoted by RTrunc and CTrunc), by collapsing columns of partial product bits into one (denoted by ORCmp) [27], or by employing inexact compressors (denoted by Miscnt) [28]. Secondly, a recursive multiplier implementing the algorithm of [29], which we approximate by using inexact 2×2-bit multiplier primitives in some least significant sub-products [30]. For each multiplier, we compute 1,000,000 random products and track the product output.

Table III lists the results of this characterization. Again, as all considered multipliers are purely combinational, we only show LUT utilization numbers, including these for the two exact baseline designs. The parameters listed indicate the width and any approximations applied to the multipliers, as outlined above. These results initially show the importance of `Relative` error metrics when comparing approximate designs that exhibit large `Absolute` errors. Secondly, our observation about adders from above also applies to multipliers: truncating too many LSBs or sub-products increases errors. Performing instead some, albeit inexact, compression on these bits can potentially reduce these errors; in line with observations in [28].

B. Approximate Matrix-Vector Multiplier

Finally, we focus on approximate constant matrix-vector multipliers developed specifically for implementation on FPGA. We utilize the computation coding scheme proposed in [21], which *slices* matrices into narrow sub-matrices and iteratively approximates these as products of sparse *matrix factors* whose non-zero elements are restricted to signed powers of two, as illustrated in Fig. 3. The resulting hardware implementation, thus, requires only adders and (constant) shifters, i.e., wires. The scheme is interesting as it enables a controllable balance between quality and hardware costs: the more matrix factors, and thus hardware, spent on approximating each slice, the higher its output quality should be.

In our current implementation [23], we allow for three degrees of freedom when decomposing matrices: 1) the number of *non-trivial* matrix factors p, 2) the number of non-zero elements in matrix factors rows e, and 3) the number of bits used to represent the matrix factor elements n. We choose the width of the slices to be the nearest power of two greater

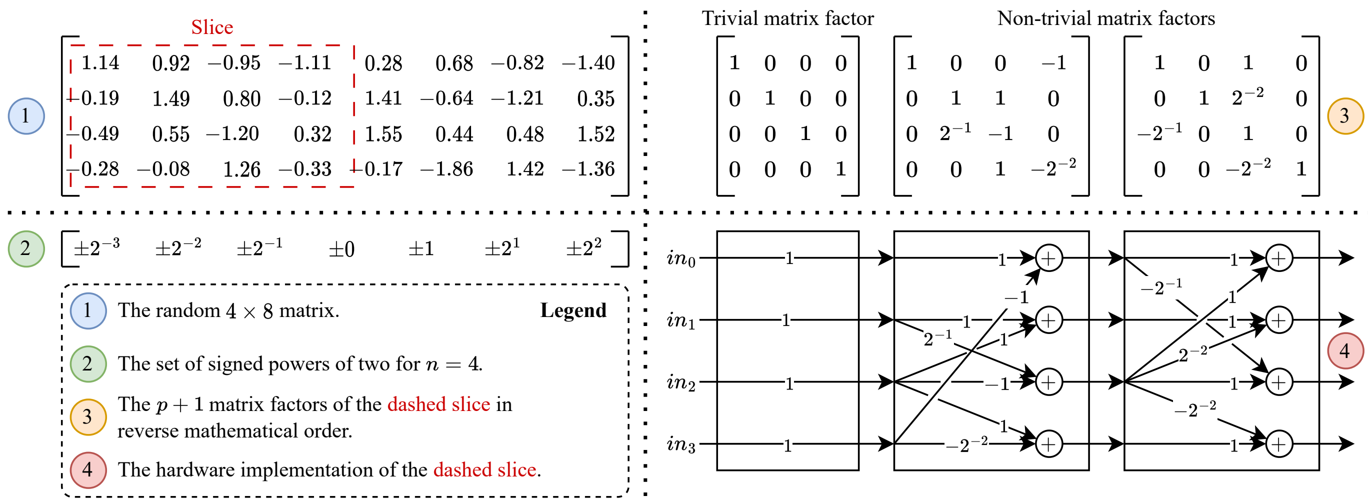

Fig. 3: Illustration of the computation coding scheme applied to a random 4×8 matrix.

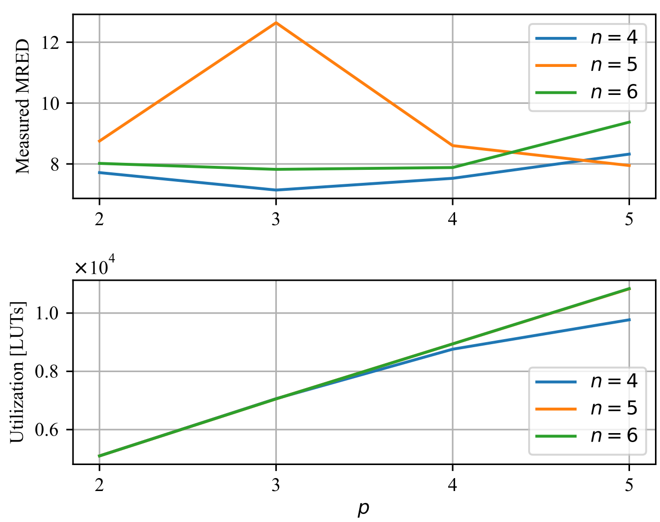

Fig. 4: Results from verifying different decompositions of the same random 16×16 matrix. Note, the designs with $n = 5$ and $n = 6$ have nearly identical LUT utilization.

than e or at most 8 as this balances algorithm run-time and hardware utilization well. For our experiments, we decompose the same random 16×16 matrix while varying p from 2 to 5 and n from 4 to 6, holding e a constant at 2. For each of these configurations, we compute 1,000,000 random matrix-vector products and track all product outputs.

Rather than tabulate the results of this experiment, we plot each design's geometric mean MRED and LUT utilization in Fig. 4. We notice that, contrary to our expectations outlined above, the measured geometric mean MRED does not appear inversely proportional to p and n, despite these designs clearly consuming more LUTs. We believe this may be due to our decomposition algorithm [23] being a slightly erroneous interpretation of the original [21] and leave unraveling this issue as future work.

V. Conclusion and Future Work

In this paper, we have described a library for verifying approximate hardware designs within a `chiseltest` test bench. The library abstracts away details of sampling outputs and computing error metrics with a simple, yet flexible interface and its well-defined internals allow for easily integrating new metrics. Moreover, we have demonstrated our tool by characterizing a collection of inexact adders and multipliers as well as an approximate matrix-vector multiplication unit from the literature.

We are currently working on extending our library with error metrics for sequence- and image-like data types used in DSP and image/video processing contexts, like (peak) signal-to-noise ratio and structural similarity [31]. Their support requires being able to potentially assemble signals temporally with many port samples. Once implemented, we plan to move beyond the scope of simulation-based verification that may require infeasible or unreasonable efforts to be exhaustive even for simple designs, including 32-bit adders like the ones used in Section IV. We will explore formal verification approaches as a potential solution to this, as highlighted in related work [1], [5], [32].

Source Access

The proposed verification library is available in open source at https://github.com/chiselverify/chiselverify. Code snippets needed to reproduce the results of Section IV are available at https://github.com/hansemandse/wapco.

References

[1] Q. Xu *et al.*, "Approximate Computing: A Survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.
[2] H. J. Damsgaard *et al.*, "Approximation Opportunities in Edge Computing Hardware: A Systematic Literature Review," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–49, 2022.
[3] A. Ometov and J. Nurmi, "Towards Approximate Computing for Achieving Energy vs. Accuracy Trade-offs," in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 632–635.

[4] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[5] S. Venkataramani *et al.*, "Approximate Computing and the Quest for Computing Efficiency," in *Proceedings of 52nd Design Automation Conference (DAC)*. ACM, 2015, pp. 1–6.

[6] A. Dobis *et al.*, "Verification of Chisel Hardware Designs with ChiselVerify," *Microprocessors and Microsystems*, vol. 96, p. 104737, 2023.

[7] K. Yoshisue *et al.*, "Dynamic Verification of Approximate Computing Circuits using Coverage-based Grey-box Fuzzing," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2021, pp. 1–7.

[8] J. Bachrach *et al.*, "Chisel: Constructing Hardware in a SCALA Embedded Language," in *Proceedings of 49th Design Automation Conference (DAC)*. ACM, 2012, pp. 1212–1221.

[9] G. Zervakis *et al.*, "Approximate Computing for ML: State-of-the-art, Challenges and Visions," in *Proceedings of 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2021, pp. 189–196.

[10] Y. Wu *et al.*, "A Survey on Approximate Multiplier Designs for Energy Efficiency: From Algorithms to Circuits," *ACM Transactions on Design Automation of Electronic Systems*, 2023, just Accepted.

[11] H. Esmaeilzadeh *et al.*, "Neural Acceleration for General-Purpose Approximate Programs," in *Proceedings of 45th International Symposium on Microarchitecture (MICRO)*. IEEE, 2012, pp. 449–460.

[12] C. Lattner *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[13] S. Eldridge *et al.*, "MLIR as Hardware Compiler Infrastructure," in *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[14] Veripool, "Verilator," 2023. [Online]. Available: https://www.veripool.org/verilator/

[15] C. Spear, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer Science & Business Media, 2008.

[16] A. S. Initiative, "Universal Verification Methodology (UVM) 1.2 User's Guide," 2015. [Online]. Available: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

[17] A. Dobis *et al.*, "Towards Functional Coverage-driven Fuzzing for Chisel Designs," in *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[18] T. Trippel *et al.*, "Fuzzing Hardware like Software," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3237–3254.

[19] R. Lin *et al.*, "chiseltest," https://github.com/ucb-bar/chiseltest, 2022.

[20] J. Kumar *et al.*, "Formal Verification of Integer Multiplier Circuits using Binary Decision Diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 4, pp. 1365–1378, 2022.

[21] A. Lehnert *et al.*, "Most Resource Efficient Matrix Vector Multiplication on FPGAs," *IEEE Access*, vol. 11, pp. 3881–3898, 2023.

[22] H. J. Damsgaard, "approx: A Library of Approximate Arithmetic Units in Chisel," https://github.com/aproposorg/approx, 2022.

[23] ——, "cmvm: Hardware-Optimized Approximate Matrix-Vector Multiplication," https://github.com/aproposorg/cmvm, 2023.

[24] Z. Yang *et al.*, "Approximate XOR/XNOR-based Adders for Inexact Computing," in *Proceedings of 13th International Conference on Nanotechnology (IEEE-NANO)*. IEEE, 2013, pp. 690–693.

[25] A. Dalloo, "Enhance the Segmentation Principle in Approximate Computing," in *Proceedings of International Conference on Circuits and Systems in Digital Enterprise Technology (ICCSDET)*. IEEE, 2018, pp. 1–7.

[26] M. Shafique *et al.*, "A Low Latency Generic Accuracy Configurable Adder," in *Proceedings of 52nd Design Automation Conference (DAC)*, 2015, pp. 1–6.

[27] T. Yang *et al.*, "Design of a Low-Power and Small-Area Approximate Multiplier using First the Approximate and Then the Accurate Compression Method," in *Proceedings of Great Lakes Symposium on VLSI (GLSVLSI)*, 2019, pp. 39–44.

[28] A. Momeni *et al.*, "Design and Analysis of Approximate Compressors for Multiplication," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 984–994, 2014.

[29] A. N. Danysh and E. E. Swartzlander, "A Recursive Fast Multiplier," in *Proceedings of Asilomar Conference on Signals, Systems and Computers*, vol. 1. IEEE, 1998, pp. 197–201.

[30] P. Kulkarni *et al.*, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," in *Proceedings of 24th Internatioal Conference on VLSI Design*, 2011, pp. 346–351.

[31] Z. Wang *et al.*, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[32] K. Palem and A. Lingamneni, "Ten Years of Building Broken Chips: The Physics and Engineering of Inexact Computing," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2s, pp. 1–23, 2013.