ACED: A Hardware Library for Generating DSP Systems

Angie Wang, Paul Rigge, Adam Izraelevitz, Chick Markley, Jonathan Bachrach, Borivoje Nikolić
University of California, Berkeley
angie.wang|rigge|adamiz|chick|jrb|bora@berkeley.edu

ABSTRACT

Designers translate DSP algorithms into application-specific hardware via primitives composed in various ways for different architectural realizations. Despite sharing underlying algorithms and hardware constructs, designs are often difficult to reuse, leading to redeveloping/reverifying conceptually similar instances. Hardware generators are attractive solutions for effectively balancing fine-grained control of implementation details with simple, retargetable hardware descriptions. This work presents ACED, a hardware library for generating DSP systems. It extends the Chisel hardware construction language and FIRRTL compiler and operates on three principles: zero-cost abstraction, unobtrusive downstream optimization/specialization promoting generator reusability, and unified, portable systems modeling and verification.

CCS CONCEPTS

• **Hardware** → *Digital signal processing*; *Application specific processors*; *Hardware description languages and compilation*; *Software tools for EDA*;

KEYWORDS

DSP, Chisel, FIRRTL, Hardware Generators, Scala, Range Analysis

ACM Reference Format:
Angie Wang, Paul Rigge, Adam Izraelevitz, Chick Markley, Jonathan Bachrach, Borivoje Nikolić. 2018. ACED: A Hardware Library for Generating DSP Systems. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3195981

1 INTRODUCTION

Algorithm development in the application domains of computer vision, big data, sensor fusion, and wireless communication has greatly outpaced our ability to deliver dedicated hardware accelerators that promise orders-of-magnitude energy-efficiency and performance improvements over CPU/GPU realizations. Many systems targeting these applications rely on new algorithmic and architectural solutions, but also on common functions like FFT, matrix multiplication, and digital filtering. Although underlying algorithms and hardware primitives are often reused, architectural realizations must be heavily—and manually—tuned to meet specific performance and deployment platform requirements. In practice, reusing blocks with structural variations increases development and verification time, elucidating the need for reusable and highly parameterized *generators* of hardware instances that support application retargetability and produce optimized RTL.

To address these barriers for developing DSP generators, we present the hardware library ACED: A Chisel Environment for DSP. Built upon the Chisel hardware construction language [1] and its compiler FIRRTL[2], it operates on three key principles:

(1) **Powerful metaprogramming with zero-cost abstractions.** Concisely express algorithm intent, yet still maintain tight control over implementation details.
(2) **Unobtrusive optimization and specialization.** Reuse generator code among all platforms and applications; automatically optimize each generated hardware instance per context.
(3) **Unified, yet portable, modeling, validation, verification, and testing.** Decouple architecture validation from hardware evaluation of quantization effects, etc.; use the same verification criteria across all design tools and abstractions.

To support these principles, ACED introduces the following library and compiler features, which preserve code reusability and enable a single unified validation/verification framework, while still producing highly-optimized hardware instances: (1) operator and data type parameterization, (2) unified systems modeling, and (3) powerful static and trace-based bitwidth optimizations. While successfully used in software-defined/cognitive radio DSP chips, ACED's ability to foster design space exploration at the algorithmic, architectural, and implementation levels is separately evaluated.

2 BACKGROUND

Traditionally, algorithm-to-hardware is done by iteratively applying designer intent across many tools/abstraction boundaries: algorithm and systems models, RTL implementations, & physical design. Expert architects tediously hand-optimize hardware realizations of algorithms, decreasing portability and malleability. For example, many algorithms use floating-point arithmetic, which cannot be synthesized with basic Verilog/VHDL; designers must undergo an error-prone, unintuitive, and manual process to convert to fixed point, track binary point locations, and create bit-level tests.

Mathworks's closed-source, model-based design flow, Simulink, simplifies passing user intent to early stages of design/verification, saving 30% in design time [3] when using pre-existing IP. It supports systems validation/verification with floating-point simulations and uses application-representative test vectors to optimize fixed-point bitwidths. The Simulink environment supports the "Chip in a Day" methodology [4], [5], but only accelerates the design of one instance for one hardware platform, rather than accelerating *generator* design targeting *multiple* platforms. It lacks transparent, programmatic, and extensible abstractions, requiring non-trivial manual effort to support multiple targets (FPGA/ASIC) and data types.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3195981

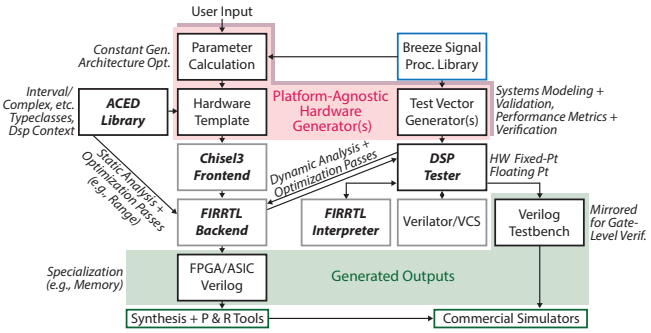

Figure 1: ACED hardware generator design environment.

High-level synthesis (HLS) tools abstract away low-level implementation details from the algorithm-to-hardware translation process. Xilinx Vivado allows designers to use C, C++, or SystemC to describe algorithms to be compiled to RTL [6]. HLS is useful for designing blocks that require complex memory access scheduling, deep pipelining, and sequential logic. However, many DSP datapaths have high degrees of parallelism, requiring carefully structured SystemC code. Finally, hardware generation from recursive functions is not allowed, except with template metaprogramming.

Other tools also enable designing generators. Genesis2 augments synthesizable SystemVerilog with Perl extensions for generator parameterization [7]. Genesis2 relies on two decoupled languages, increasing the likelihood of generating difficult-to-debug syntax errors. Spiral is a generator specifically targeting DSP hardware [8]. Although powerful, Spiral is designed for static function generation and is not well suited for dynamic or reconfigurable architectures.

Hardware construction/generation languages like Bluespec (closed-source) and Chisel (open-source) enable systems modeling, generator construction, and test environment creation with one underlying functional and object-oriented programming language [9], [1]. Each supports (1) custom-defined number abstractions, (2) compile-time type-checking, (3) type polymorphism and operator overloading for hardware template parameterization, (4) recursive functions to generate hardware, and (5) functional constructs (e.g., *map*, *reduce*, etc.) that enable concisely expressing structured datapaths. Bluespec and Chisel are architecturally transparent, exposing fine-grained implementation details and enabling optimization opportunities that are typically hidden by HLS tools.

3 ACED: A CHISEL ENVIRONMENT FOR DSP

ACED (Fig. 1) provides constructs to capture and propagate designer intent down the hardware abstraction hierarchy without redundant specification. To allow designers to focus on algorithms, it separates algorithmic and implementation abstractions, yet supports platform/application-specific specialization and optimization.

ACED is implemented on top of the open-source Chisel hardware construction language and its open-source compiler, FIRRTL [2]. Because Chisel is written in Scala, existing Scala and Java libraries like the Breeze numerical processing library can be used to generate "golden model" references and constant coefficients for DSP systems [10]. The same underlying language is used by generators, optimizing/specializing compiler passes, and the testing environment, allowing a single set of tests to verify the system. This reduces the likelihood of translation errors and simplifies development and design-space exploration.

The following sections outline three major contributions of this work: operator and data type parameterization, a unified testing environment, and powerful bit-level optimizations.

3.1 Operator and Data Type Parameterization

Type-generic generators increase code reuse; for example, one type-generic FIR filter generator can replace separate complex- and real-type FIR filter generators. Type-generic generators require: (1) operator parameterization, (2) native operator and data types, (3) flexibility to specify new number types, and (4) the ability to parameterize the type used by a Chisel circuit via type generics.

Many DSP operations have implementation details, such as rounding modes (half-up, floor, ceil, & truncate), overflow modes (grow, wrap, & saturate), and depth of pipelining, that affect performance/area trade-offs. Consistently specifying these per-operation properties is tedious and error-prone; ACED provides a *DspContext* to set and override these properties. *DspContext* sets defaults at the top-level module scope and may be overridden hierarchically. If necessary, a single operator's parameters can be overridden.

Powerful native types help achieve functionally correct designs. Chisel's *FixedPoint* type automatically propagates binary points, replacing manual tracking across operations. To enable range-based analyses & optimizations, the native type *Interval* was added.

To support specifying new number types and parameterizing Chisel circuits via type generics, ACED abstracts data types and representations using typeclasses, a functional programming pattern that enables type-safe polymorphic code. ACED extends the Scala numeric library Spire [11] with typeclasses for Chisel/ACED types including *UInt*, *SInt*, *FixedPoint*, *Interval*, *DspComplex*, and *DspReal*. Designers can create new number types and use them with other ACED library components [1]. The following illustrates a custom complex number type for use with type-generic generators. First, a *DspComplex* Chisel type is created from a Chisel *Bundle* (like a struct):

```
// T <: Data:Ring says generic type T is a Chisel
     hardware type and implements Ring typeclass
class DspComplex[T<:Data:Ring](val real:T, val imag:T)
     extends Bundle { ... }
```

ACED uses typeclasses for algebraic structures (ring, order, signed, real, etc.). A typeclass *Ring[T]* defines the behavior of addition, multiplication, subtraction, and negation for objects of type *T*; *DspComplex* supports ring operations via a *Ring[DspComplex[T]]* instance. The complex × operator uses three real multipliers:

```
class DspComplexRing[T<:Data:Ring] extends Ring[
     DspComplex[T]] {
  def times(f: DspComplex[T], g: DspComplex[T]) = {
    val p1 = f.real * (g.real + g.imag)
    val p2 = (f.real + f.imag) * g.imag
    val p3 = (f.imag - f.real) * g.real
    return DspComplex.wire(p1 - p2, p1 + p3)
  } ...
}
```

Implementing *DspComplexRing* allows *DspComplex* to be used for any generator with type constraint *T <: Data:Ring*. If such a generator is invoked with an instance of *DspComplex*, all instances of $f * g$ will call this implementation of *times(f, g)*.

[1] Non-2^n FFTs also benefit from mixed-radix number types, where full adders are replaced with custom adders built with simple subtractor + mux modulo operations.

The following direct-form, real-coefficient FIR filter template illustrates how ACED is used to make hardware generators. It directly maps the equation $y[n] = \sum_{k=0}^{N} h[k]x[n-k]$ into hardware.

```
class FIR[T <: Data:Ring:ConvertableTo](genI: => T,
    genO: => T, cfs: Seq[Double]) extends Module {
  // Set module input/output ports
  val io = IO(new FilterIO(genI, genO))
  // Specify pipeline stages and dataflow precision
  val newContext = DspContext.current.copy(numMulPipes=3,
      binaryPoint= Some(14))
  // New scope has newContext parameters
  DspContext.alter(newContext) {
    // Generate register sequence to delay input (taps)
    val tps = cfs.tail.scanLeft(io.in)(
      (in, _) => RegNext(in, init = Ring[T].zero))
    // Make constants from floating-point coefficients
    val cs = cfs.map(c => ConvertableTo[T].fromDouble(c))
    // Create one multiplier per tap/coefficient pair
    val ms = tps.zip(cs).map{case (t,c) => t context_* c}
    // Set output to sum of all multiplier outputs
    io.out :=  ms reduce (_ context_+ _)
  }
}
```

The template takes inputs/outputs of type *Data* that have the typeclasses *Ring* (for add/multiply operations) and *ConvertableTo* (for translating Scala Doubles into the correct Chisel literal type via the *fromDouble* function). The Scala compiler checks that any invocation of the generator satisfies these type constraints. Functional programming constructs represent DSP operations concisely. *DspContext* sets the number of pipeline registers for multiplication and the number of fractional bits used to represent the filter coefficients. This code creates an FIR module specialized for Interval types:

```
new FIR(Interval(range"[-16, 16).2"),
        Interval(range"[?, ?].4"), ...)
```

Likewise, a filter with complex inputs is instantiated with:

```
new FIR(DspComplex(Interval(range"[-16, 16).2")),
        DspComplex(Interval(range"[?, ?].4")), ...)
```

3.2 Unified Systems Modeling and Verification

When verifying systems, the same tests should be propagated across all layers of the design hierarchy, from the systems model down to behavioral and gate level RTL code. Reusable tests prevent design intent from being mistranslated across abstraction and tool boundaries. The ACED library contains a Chisel tester extension that natively supports DSP-specific number representations and error tolerance parameters. ACED's *DspTester* tracks the sequence of user inputs, prompted design outputs, and time steps, mirroring them one-for-one in an automatically generated Verilog testbench that can be run on specialized designs.

It can often be hard to determine if poor DSP system performance is due to quantization error from finite word length or an algorithmic error. Using floating-point data representations virtually eliminates quantization errors and makes it easy to evaluate algorithmic performance and correctness. ACED provides a *DspReal* type that implements non-synthesizable double-precision floating point, simulatable with either the FIRRTL Interpreter or SystemVerilog's Real type. Typeclasses have been implemented for *DspReal*, so floating-point numbers can be used with type-generic generators without any changes to the generator. Once mathematical correctness has been verified, the template can be re-parameterized to use a synthesizable type for implementation.

It is frequently useful to integrate analog or non-synthesizable floating-point models together with synthesizable hardware for system evaluation. For example, a designer building a radio baseband may want to study the effects of ADC or FFT quantization error on either a subsequent demodulator's reliability or the entire system performance. One way to do this is to model a radio transmitter and over-the-air channel using library functions in Scala and then feed the model output via the *DspTester* to a simple receive chain consisting of an analog/mixed-signal ADC model followed by parameterized FFT and demodulator blocks that can use either synthesizable or non-synthesizable constructs (Fig. 2). To support this, a *ChiselConvertableFrom* typeclass that implements shims between *DspReals* and synthesizable types has been created.

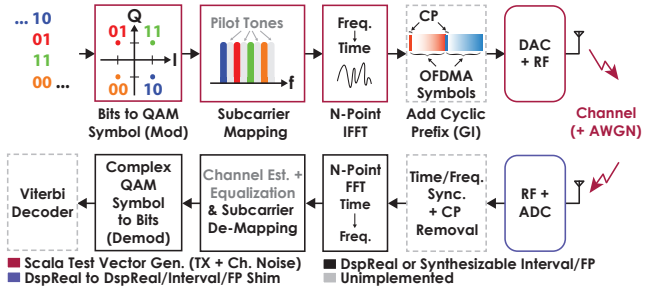

Figure 2: A system model of an OFDM transmitter/receiver pair with an additive white Gaussian noise channel.

One important system metric for radio receivers is the symbol error rate (SER) vs. the input signal-to-noise ratio (SNR). The SER vs. SNR for Fig. 2's systems model with various combinations of synthesizable and non-synthesizable blocks is compared to the theoretical limit in Fig. 3b. This can be used to determine system sensitivity to the performance of individual blocks. Additionally, an FFT generator was evaluated for use in the context of this baseband receiver chain and a separate spectral analysis system. The SQNR achievable for different FFT bitwidths is given in Fig. 3a and used to inform bitwidths of generated FFT instances in practical designs, which included ADCs that were modeled using ACED constructs.

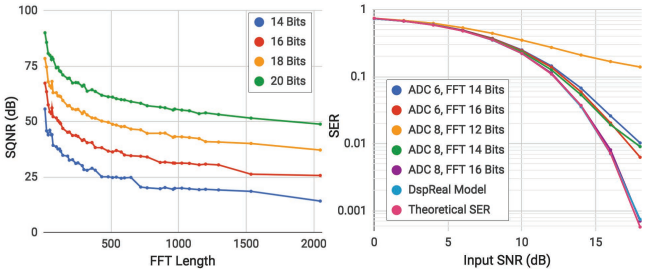

Figure 3: a) SQNR vs. FFT length and b) SER vs. baseband SNR using 16-QAM & a 128-pt FFT. I/Q ADC + FFT output bitwidths indicated.

3.3 Interval-Based Bitwidth Reduction

DSP designers may know module input/output ranges, but often rely on synthesis tools to infer internal node widths. Unfortunately, the tools cannot infer a signal's range from its bitwidth, resulting in suboptimal power and area results. Designer intent can be better captured by directly encoding input ranges into the design, allowing automatic range propagation and bitwidth reductions.

Simple forward/backward range propagation [12], [13], while still conservative, offers improvements without complex symbolic analysis of ranges. Potential ranges are tracked for each operation, and the solver finds the worst case bitwidth. Affine analysis of ranges [14] can cancel correlated terms (e.g., $A - A$ has a range $[0, 0]$), leading to a more compact design at the expense of more complex analysis. These techniques are used in high-level synthesis flows [15], which unfortunately require users to express their designs using non-zero-cost abstractions. The benefits of these optimizations can be offset by the difficulties encountered when porting existing RTL to a new flow and/or application. Dynamic bitwidth analyses [3] can significantly improve upon these methods. They require a thorough set of application-specific test vectors, potentially leading to long runtimes. Without complete coverage, the design's correctness is only statistically guaranteed[2].

Manual bitwidth optimization, especially for *generators*, is tricky, inconvenient, and error-prone. To address this, ACED supports automatic (static) range propagation that aids bitwidth optimization via FIRRTL passes. It also provides dynamic analysis—FIRRTL Interpreter simulation results can automatically direct bitwidth reduction.

3.3.1 Static Interval Optimization: An interval type containing precision, a lower bound, and an upper bound was added to Chisel and FIRRTL. Bounds are closed [], open (), or unknown. Precisions are unknown or an integer. Hardware components like wires, ports, and registers can be declared with these interval types. Additionally, two new primitive operators enable limiting an expression to an interval: *clip* and *wrap* will clip (or wrap) the first input argument's value to the interval of the second argument.

To resolve all unknown bounds and unknown precisions, we (1) resolve unknown precisions, (2) trim interval bounds to known precision, and (3) resolve all upper and lower bounds. Both (1) and (3) require obtaining variable constraints of the form $w_1 \geq \ldots$, or $w_1 \leq \ldots$, where unknown values are given a variable name. We then ensure monotonicity by checking that no value has both $\geq$ and $\leq$ constraints.

The relationships between bound/precision constraints and their FIRRTL expressions are complex, but a subset of these relationships are summarized in Table 1. To demonstrate, the following example calculates all variable constraints based off of either precisions or bounds:

```
wire x:Interval
wire y:Interval
wire z:Interval
z <- x + y
```

Original

```
wire x:Interval[a, b].c
wire y:Interval[d, e].f
wire z:Interval[g, h].i
z <- x + y
```

With Variables

```
i ≥ max(f,c)
-or-
g ≤ a + d
h ≥ b + e
```

Constraints

After collecting constraints and merging constraints on the same variable, a custom constraint solver uses a forward-backward algorithm to solve the constraints, as described in Fig. 4. Finally, all values specified by variable names are replaced with their solved bounds or precisions. As an example, the following collected constraints undergo forward substitution, backward substitution, and evaluation:

[2]Unrepresentative test vectors can incorrectly add or remove hardware and result in wasted bits or reduced dynamic range.

Algorithm Forward-Backward Substitution

```
procedure SUBSTITUTE(sol: Map[Name,Constraint],c: Constraint)
    if c typeof Variable & sol.has(c.name) then
        c ← sol.get(c.name)
    if c typeof Operator then
        for i in (0 → c.children.length) do
            c.children[i] ← optimize(c.children[i])
            c.children[i] ← substitute(sol, c.children[i])
    return c
procedure FORWARD(con: Map[Name,Constraint])
    sol ← Map[Name, Constraint].empty
    for i in (0 → (con.size − 1)) do          ▷ in insertion order
        (name, c) ← con[i]                    ▷ get ith (key,value)
        sol[name] ← substitute(sol, c)
        sol[name] ← optimize(sol[name])
        sol[name] ← removecycle(name, sol[name])
    return sol
procedure BACKWARD(con: Map[Name,Constraint])
    sol ← Map[Name, Constraint].empty
    for i in ((con.size − 1) → 0) do          ▷ reverse insertion order
        (name, c) ← con[i]                    ▷ get ith (key,value)
        sol[name] ← substitute(sol, c)
        sol[name] ← optimize(sol[name])
    return sol
```

Figure 4: Forward substitution populates a sol with forward-solved variable constraints. It uses the substitution procedure which recursively visits constraint children and substitutes variables with their optimized constraints in sol (if they exist). It also attempts to remove cyclic constraints—failures are reported to the user. Backward substitution iterates through the forward-solved constraints in reverse order, calling substitute and optimizing the result. The solver ignores whether $Name$ is $\geq$ or $\leq$ $Constraint$ in Map, as all constraints are monotonic.

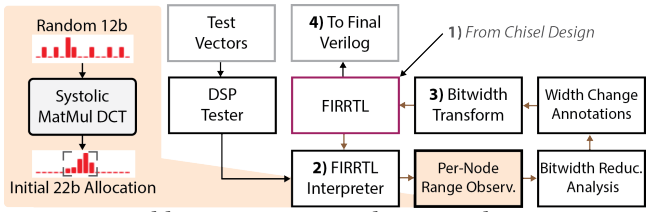

Figure 5: Bitwidth optimization procedure using the FIRRTL Interpreter for dynamic range analysis. Utilization is reported visually to help build design intuition.

```
a ≥ b+c
b ≥ 2
d ≥ b-a
c ≥ 3
```

Collect

```
a ≥ b+c
b ≥ 2
d ≥ 2-(2+c)
c ≥ 3
```

Forward Sub.

```
a ≥ 2+3
b ≥ 2
d ≥ 2-(2+3)
c ≥ 3
```

Backward Sub.

```
a ≥ 5
b ≥ 2
d ≥ -3
c ≥ 3
```

Evaluation

A simple example can demonstrate the savings of ACED range analysis over Chisel's standard bitwidth propagation. The product of three 2-bit unsigned numbers requires 6 bits according to Chisel (Table 1), but since the input ranges can be at worst [0, 3], the worst-case output range is [0, 27], requiring only 5 unsigned bits; the savings are amplified as operations are chained together.

3.3.2 Dynamic Interval Optimization: As shown in Fig. 5, to further optimize the bitwidths of a design with a thorough set of test vectors, dynamic interval analysis using the FIRRTL Interpreter can be turned on to automatically trim bitwidths of internal nodes.

Op	Chisel Baseline	Interval Lower Bound	Interval Upper Bound	Precision
add(x, y)	$max(x_w, y_w) + 1$	$x_l + y_l$	$x_h + y_h$	$max(x_p, y_p)$
sub(x, y)	$max(x_w, y_w) + 1$	$x_l - y_h$	$x_h - y_l$	$max(x_p, y_p)$
mul(x, y)	$x_w + y_w$	$min(x_l * y_l, x_l * y_h, x_h * y_l, x_h * y_h)$	$max(x_l * y_l, x_l * y_h, x_h * y_l, x_h * y_h)$	$x_p + y_p$
mux(p, x, y)	$max(x_w, y_w)$	$min(x_l, y_l)$	$max(x_h, y_h)$	$max(x_p, y_p)$
wrap(x, y)	x_w	y_l	y_h	x_p
clip(x, y)	$min(x_w, y_w)$	$max(x_l, y_l)$	$min(x_h, y_h)$	x_p

Table 1: A subset of corresponding constraint expressions for each supported FIRRTL primitive operation. Listed operation arguments are FIRRTL interval-typed expressions (x, y). The argument subscripts refer to the lower bound, upper bound, precision, or width of the expression (e.g., x_l, x_u, x_p, or x_w, respectively). Constraint expressions are: $+, -, *, max, min$, and a constant. Operations $shl, shr, dshl, dshr, bpshl, bpshr$, and $bpset$ are omitted due to space constraints.

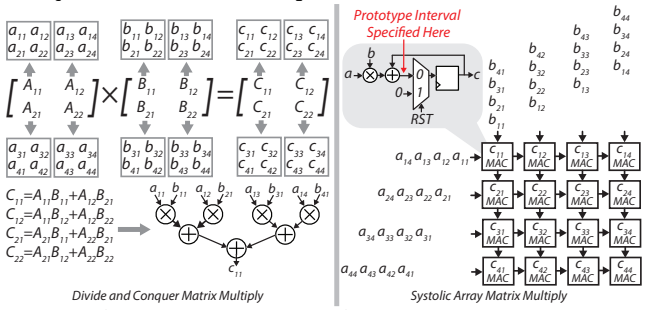

Figure 6: a) Divide + conquer and b) systolic array MatMuls. *Intervals* are automatically propagated in a); prototype *Intervals* from loop unrolling must be supplied in b).

Starting with an unoptimized circuit, the FIRRTL Interpreter captures the smallest and largest values seen at significant (named) internal nodes. It tracks the min./max., mean, and standard deviation at these nodes and displays a histogram of values to give designers a better understanding of data flow through the circuit. As an example, when randomly generated test vectors occupying a 12-bit input range are fed into the circuit in Fig. 5, it is useful to see that the original allocation of 22 bits at the output is far too conservative, and only ~31% of the range is actually utilized.

FIRRTL annotations are generated for downstream optimization passes to: (1) adjust node widths downward based off of their simulated extrema (additional guard bits can be added) or (2) adjust widths to support $x\sigma$ ranges. At the expense of additional logic, saturation operations can be added to protect from undesirable overflow behavior. Finally, bitwidth-optimized Verilog RTL is generated.

4 RESULTS

To evaluate ACED, generators supporting different $\boldsymbol{AB} = \boldsymbol{C}$ matrix multiplication algorithms and architectures (Fig. 6) have been built. Recursive divide-and-conquer MatMul algorithms showcase bitwidth reduction from static and dynamic interval analyses. The associated generators are constructed via a *Matrix* data type supporting *Ring* operations like +, −, and ×. $n \times n$ matrices are recursively partitioned into 4 new block matrices, and either a standard $O(n^3)$ algorithm or Strassen's $O(n^{2.81})$ algorithm [16] is used to create the fully parallel data-flow graph. A hardware template for more typical systolic array-based MatMuls that rely on iterative multiply-accumulate (MAC) operations has also been created. Systolic implementations are much more compact (e.g., the 8-bit 8×8 systolic DCT using Chisel *FixedPoint* has 16.4% of the area of an equivalent DCT implemented using Strassen's algorithm), but require more compute cycles. A prototype *Interval* used for range analysis is specified at the MAC's adder output with the equivalent (unsynthesized) sum of products in unrolled form.

Hardware instances are generated for 8×8 MatMuls, and *Interval* ranges at fixed input bitwidths are swept. As an example, the smallest *Interval* needing 8 bits is $[-65, 65)$—where the MSB is relatively underutilized, and savings accumulate more quickly—while the largest (full-width *Interval*) is $[-128, 128)$. For DCT hardware, the MatMuls are configured so matrix $\boldsymbol{A}$ contains constant DCT coefficients, and 1-D DCTs are calculated for the columns of $\boldsymbol{B}$. The generated instances are synthesized with a commercial 16nm FinFET technology at 500MHz[3] to observe how FIRRTL range optimizations affect the QoR of the final design.

Static range analyses promise the most benefit over baseline *FixedPoint* designs relying on width inference for small input bitwidths (9% smaller area for full-width 4-bit input *Intervals* with DCT using Strassen's algorithm); the number of bits trimmed makes up a larger fraction of bits in the Chisel baseline design. Additionally, the DCT examples show that static range analysis performs better with specific input information, e.g., that elements of $\boldsymbol{A}$ are fixed values (Fig. 7). However, even when only full-width input *Intervals* are used, range analysis provides some area savings. The algorithms are easily compared: Fig. 8 shows that range analysis benefits the standard divide-and-conquer MatMul less, but the standard algorithm is more area efficient than Strassen's algorithm for 8×8 matrices. Unfortunately, the synthesized area is larger for a systolic DCT implementation with 16-bit inputs when the full-width *Interval* type is used (vs. *FixedPoint*); bitwidth trimming using range analysis removes some design regularity that synthesis tools use for deep optimizations across PEs. Finally, ACED static range optimization times are compared with Chisel width inference times in Table 2. Recursive implementations using range analyses require significantly longer optimization and constant propagation times.

User intent is embedded into the design via dynamic range analysis. Test vectors spanning the input range are generated from a uniform random distribution, and bitwidth optimization is performed, leading to significant area savings. In particular, properly capturing the system context in simulation (e.g., when the inputs are first low-pass filtered) leads to greater dynamic analysis savings—up to 38% in Fig. 7.

To evaluate ACED, a similar Strassen MatMul has been implemented using MATLAB HDL Coder and C++ with Vivado HLS. Synthesis results are summarized in Table 3. It should be noted that the Vivado HLS results required extensive refactoring and heavy use of pragmas—without which the designs had poor throughput and

[3] SS process corner, 0.72V supply, 125°C, and cworst_CCworst RC corner with Cadence Genus

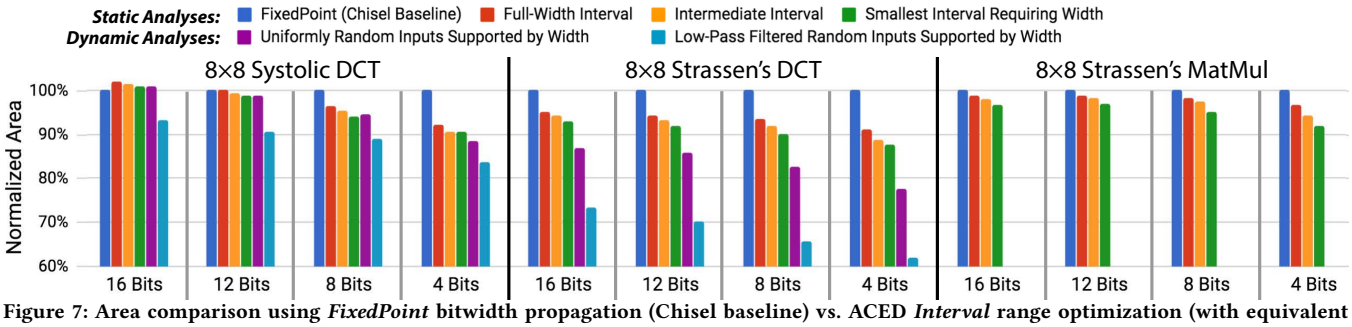

Figure 7: Area comparison using *FixedPoint* bitwidth propagation (Chisel baseline) vs. ACED *Interval* range optimization (with equivalent input bitwidths). Static and dynamic range analyses can automatically reduce area down to 62%, with zero (static) or minimal (dynamic) performance penalties using representative test vectors and input conditions. Constant coefficients are built into DCT-specific MatMul circuits. Area is calculated post-synthesis and includes the cell + estimated net area.

8 × 8 Design	16b Fix.	16b Intvl.
Systolic DCT	1.361	2.597
Strass. DCT	4.004	19.020
Strass. MatMul	4.065	17.104

Table 2: Static range optimization times (s) for designs using 16-bit *Fixed-Point* and full-width *Interval* inputs.

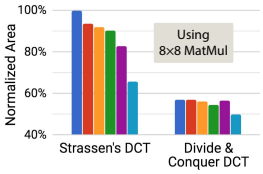

Figure 8: Relative areas for 8-bit divide + conquer & Strassen's MatMul DCTs.

MatMul Generator Design	Slice LUTs	DSPs
MATLAB HDL Coder (2017B)	41528	0
C++ Vivado HLS (2017.4)	18364	196
ACED (Smallest Full-Width *Interval*)	22924	328

Table 3: Synthesis results for ACED, HLS, and MATLAB HDL Coder (2017B) outputs using Vivado 2017.4. 8-bit 8x8 Strassen's matrix multiplies were synthesized with 10.0ns 50% duty cycle clocks for the ZC706 FPGA. MATLAB used dynamic optimization with uniform random matrices as inputs. ACED used static range optimization.

high resource utilization—to achieve reasonable QoR. The Chisel designs were generated for ASICs and reused here without any tuning.

5 CONCLUSION

Extensibility is important to a useful library. ACED operates at several different levels of abstraction, from high level type-generic generators to low-level bitwidth optimizations, but these abstractions are kept separate to facilitate extensibility. For example, static bitwidth optimization and *Intervals* can be used on any design (e.g., a processor) and do not make assumptions about being run with the DSP-centric types.

We have shown how the ACED hardware library can be used for generating DSP systems. It supports zero-cost abstraction and high-level data types, "free" optimization and specialization via open-source compiler passes, and a unified systems modeling framework. *Interval* bitwidth reduction is analyzed closely, and the ACED library has been successfully used to create DSP hardware blocks including FFTs, CORDICs, and FIR filters for various tapeouts.

It is important to emphasize that the contribution of our paper is not just the particular bitwidth optimization techniques, but rather their implementation in an open compiler framework and a discussion of the value of these techniques in the context of writing generators. Other optimization techniques can be added to the compiler chain in a way that is transparent to the designer of a top level generator.

ACKNOWLEDGEMENTS

This work was funded by the DARPA CRAFT program (HR0011-16-C-0052), NSF-GRFP (DGE-1106400), BWRC, and ADEPT (Intel iSTC on Agile Design). The authors thank Chris Yarp, Jim Lawson, Stephen Twigg, Richard Lin, and Stevo Bailey for their support.

REFERENCES

[1] J. Bachrach, *et al.* 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proc. IEEE/EDAC/ACM DAC*. 1212–1221.
[2] A. Izraelevitz, *et al.* 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *Proc. IEEE/ACM ICCAD*. 209–216.
[3] A. Gai. 2006. Model-based design with MATLAB, Simulink, and Altera DSP builder. ftp://ftp.altera.com. (2006).
[4] W. R. Davis, *et al.* 2001. A design environment for high throughput, low power dedicated signal processing systems. In *Proc. IEEE CICC*. 545–548.
[5] D. Marković, *et al.* 2007. ASIC design and verification in an FPGA environment. In *Proc. IEEE CICC*. 737–740.
[6] 2017. Ultrafast high-level productivity design methodology guide. https://www.xilinx.com. (Dec 2017).
[7] O. Shacham, *et al.* 2012. Avoiding game over: Bringing design to the next level. In *Proc. IEEE/EDAC/ACM DAC*. 623–629.
[8] M. Püschel, *et al.* 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (Feb 2005), 232–275.
[9] R. Nikhil. 2004. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Proc. ACM/IEEE MEMOCODE*. 69–70.
[10] 2018. Breeze. https://github.com/scalanlp/breeze. (2018).
[11] E. Osheim, *et al.* 2018. Spire. https://github.com/non/spire. (2018).
[12] H. Keding, *et al.* 1998. FRIDGE: A fixed-point design and simulation environment. In *Proc. IEEE DATE*. 429–435.
[13] M. Stephenson, *et al.* 2000. Bitwidth analysis with application to silicon compilation. In *Proc. ACM SIGPLAN PLDI*. 108–120.
[14] D.-U. Lee, *et al.* 2005. MiniBit: Bit-width optimization via affine arithmetic. In *Proc. IEEE/ACM DAC*. 837–840.
[15] C. C. Wang, *et al.* 2011. An automated fixed-point optimization tool in MATLAB xsg/syndsp environment. *ISRN Signal Processing* (2011).
[16] V. Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13, 4 (Aug 1969), 354–356.