Axilog: Language Support for Approximate Hardware Design

Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar
Sindhuja Sethuraman, Kartik Ramkrishnan*, Nishanthi Ravindran*, Rudra Jariwala*, Abbas Rahimi§
Hadi Esmaeilzadeh, Kia Bazargan*
Georgia Institute of Technology, *University of Minnesota, §UC San Diego
axilog@act-lab.org, http://www.act-lab.org/artifacts/axilog

***Abstract*—Relaxing the traditional abstraction of "near-perfect" accuracy in hardware design can lead to significant gains in energy efficiency, area, and performance. To exploit this opportunity, there is a need for design abstractions that can systematically incorporate approximation in hardware design. We introduce Axilog, a set of language annotations, that provides the necessary syntax and semantics for approximate hardware design and reuse in Verilog. Axilog enables the designer to relax the accuracy requirements in certain parts of the design, while keeping the critical parts strictly precise. Axilog is coupled with a Relaxability Inference Analysis that automatically infers the relaxable gates and connections from the designer's annotations. The analysis provides formal safety guarantees that approximation will only affect the parts that the designer intended to approximate, referred to as relaxable elements. Finally, the paper describes a synthesis flow that approximates only the relaxable elements. Axilog enables applying approximation in the synthesis process while abstracting away the details of approximate synthesis from the designer. We evaluate Axilog, its analysis, and the synthesis flow using a diverse set of benchmark designs. The results show that the intuitive nature of the language extensions coupled with the automated analysis enables safe approximation of designs even with thousands of lines of code. Applying our approximate synthesis flow to these designs yields, on average, 54% energy savings and 1.9× area reduction with 10% output quality loss.**

I. Introduction

Emerging applications such as data analytics, machine learning, multimedia, search, and cyber physical systems are inherently approximate and can tolerate imprecision in many parts of their computation. The prevalence of these applications has coincided with diminishing performance and energy returns from traditional CMOS scaling [1,2]. Several pioneering works have shown significant benefits with approximation at the circuit level [2–17]. Most of these techniques focus on optimization of individual functional units and approximate synthesis algorithms, opening avenues for utilizing approximation at the circuit level. However, there is a lack of abstractions that enable designers to methodically control which parts of the circuit can be synthesized approximately while keeping critical elements, such as the control logic, precise. Thus, there is a need for *approximate hardware description languages* for systematic approximate hardware design.

In this work, we introduce Axilog–a set of concise, intuitive, and high-level annotations–that provides the necessary syntax and semantics for approximate hardware design and reuse in Verilog. Axilog enables designers to reason about and delineate which parts of a hardware system or circuit design are critical and cannot be approximated. A key factor in our language formalism is to abstract away the details of approximation while maintaining the designer's oversight in deciding which circuit elements are synthesized approximately. Axilog is also devised with modular reusability as a first order consideration. In general, hardware systems implementation relies on modular design practices where the engineers build libraries of modules and reuse them to build more complex hardware systems. Axilog provides a specific set of annotations to support reusability. Section II elaborates on the Axilog annotations for approximate hardware design and reuse.

TABLE I: Summary of Axilog's language syntax.

Phase	Annotation	Arg	Description
Design	relax	wire, reg, output, inout	Declare an argument as relaxable. Any design element that exclusively affects the argument is safe to approximate.
	relax_local		Similar to relax but the approximation does not cross module boundaries.
	restrict		Any design element that affects the argument is made precise unless explicitly relaxed with another annotation.
	restrict_global		All the design elements affecting the argument are precise.
Reuse	approximate	output, inout	Indicates the output carries relaxed sematics.
	critical	input	Indicates the input is critical and approximate elements cannot drive it.
	bridge	wire, reg	Allow connecting an approximate element to a critical input.

There are a number of approximate software programming languages including EnerJ [18] and Rely [19]. We do not extend EnerJ or Rely's language constructs to Verilog because they require a large number of manual annotations. Instead, we introduce a new set of annotations and couple them with a Relaxability Inference Analysis that automatically infers which circuit elements are relaxable with respect to the designer's annotations. The Relaxability Inference Analysis formally guarantees that approximation will only affect the circuit elements that the designer intended to approximate. Section III details this analysis. In Section IV, we describe an approximate synthesis flow that leverages a commercial synthesis tool (Synopsys Design Compiler) to apply approximation to the parts of the design that are deemed safe to approximate by the analysis.

Section V evaluates Axilog, its analysis, and the synthesis flow using a set of benchmark designs from domains including arithmetic units, signal processing, robotics, machine learning, and image processing. The evaluations use TSMC 45-nm multi-V_t libraries at the slowest PVT corner and show that by setting the quality loss to 5%, our framework achieves, on average, 45% energy savings and 1.8× area reduction. Allowing a quality loss of 10% results in 54% average energy savings and 1.9× area reduction. Further, we evaluate the robustness of our approach across a wide range of temperature variations (ΔT=125°C). Axilog yields these significant benefits while only requiring between 2 and 12 annotations even with complex designs containing up to 22,407 lines of code. These results confirm the effectiveness of Axilog in incorporating approximation in the hardware design cycle.

II. Approximate Hardware Design with Axilog

Our principle objectives for approximate hardware design in Axilog are (1) to carefully craft a small number of Verilog annotations which provide the designer with complete oversight and governance over the approximation; (2) to minimize the number of manual annotations while relying on the Relaxability Inference Analysis to automatically infer the designer's intent for approximation; (3) to relieve the designer from the details of the approximate synthesis process by providing an intuitive separation between approximate design and synthesis

and (4) to support the reuse of Axilog modules across different designs without the need for reimplementation. Furthermore, Axilog is a backward-compatible extension of Verilog. That is, an Axilog code with no annotations is a normal Verilog code and the design carries the traditional semantics of strict accuracy. To this end, Axilog provides two sets of language extensions, one set for the design (Section II-A) and the other for the reuse and interfacing of hardware modules (Section II-B). Table I summarizes the syntax for the design and reuse annotations. The annotations for design dictate which operations and connections are relaxable (safe to approximate) in the module. Henceforth, for brevity, we refer to operations and connections as design elements. The annotations for reuse enable designers to use the annotated approximate modules across various designs without the need for reimplementation. The back-end flow then uses these annotations to determine where in the design to use less costly hardware resources that allow relaxed accuracy (see section III). We provide detailed examples to illustrate how designers are able to appropriately relax or restrict the approximation in hardware modules. Using these examples, we elucidate the interplay between annotations and language constructs for hardware design, such as instantiation, concurrent assignment, and vector declaration. In the examples, we use **background shading** to highlight the relaxable elements inferred by the analysis.

A. *Design Annotations*

Axilog allows each design element to be precise or approximate. The designer's annotations provide the guidelines to identify the design elements that are safe to approximate.

Relaxing Accuracy Requirements By default, all design elements (operations and connections) are precise. The designer can use the relax(arg) statement to *implicitly* approximate a subset of these elements. The variable arg is either a wire, reg, output, or inout. Design elements that *exclusively* affect signals designated by the relax annotation are safe to approximate. The use of relax is illustrated using the following example.

```
module full_adder(a, b, c_in, c_out, s);
    input a, b, c_in; output c_out;
    approximate output s;
    assign s = a ^ b ^ c_in;
    assign c_out = a & b + b & c_in + a & c_in;
    relax(s);
endmodule
```

In this full_adder module, s is the sum of the three inputs, a, b, and c_in. The relax(s) statement shows the designer's intent to relax the accuracy requirement of the design elements that exclusively affect s, while keeping the unannotated c_out (carry out) signal precise. The relax(s) statement implies that the analysis can automatically approximate the XOR operations. Adhering to the designer's intent, the unannotated c_out signal and the logic generating it will not be approximated. Furthermore, since s will carry relaxed semantics, its corresponding output is marked with the approximate annotation. In general any output port that carries approximate semantics needs to be marked with the approximate annotation. The approximate annotation is necessary for reusing modules and will be discussed in Section II-B. With these annotations and the automated analysis, the designer does not need to *individually* declare the inputs (a, b, c_in) or any of the XOR (^) operations as approximate. Thus, while designing approximate hardware modules, this abstraction significantly reduces the burden on the designer to understand and analyze complex data flows within the circuit.

Scope of approximation. Scope of the relax annotation crosses the boundaries of instantiated modules. The code on the left side of the following example illustrates this characteristic. The relax(x) annotation in the nand_gate module implies that the AND (&) operation in the and_gate module is relaxable. In some cases, the designer might not prefer the approximation to cross the scope of the instantiated modules. For example, the designer might not want the approximation to affect a third-party IP core. Axilog provides the relax_local annotation to limit the scope of approximation and its effects on the logic within the same module in which the annotation is declared.

```
module and_gate(n,a,b);
    input a, b; output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax(x);
endmodule
```

```
module and_gate(n,a,b);
    input a,b; output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax_local(x);
endmodule
```

The code on the right side shows that the relax_local annotation does not affect the semantics of the instantiated and_gate module, a1. In this case, the AND(&) operation in the and_gate module is not relaxable. However the NOT(~) operation which shares the scope of the relax_local annotation is relaxable. The scope of approximation for both relax and relax_local is the module in which they are declared. Relax penetrates the boundary of the module instantiations but relax_local does not. The relax_local and relax annotations can also be applied selectively to certain bits of a vector.

Restricting approximation. In some cases, the designer might want to *explicitly* restrict approximation in certain parts of the design. Axilog provides the restrict(arg) annotation that ensures that any design element that affects the annotated argument (arg) is precise, *unless* a preceding relax or relax_local annotation has made the driving elements relaxable.

```
module and_gate(n, a, b);
    input a,b; output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax(w0)
    restrict(x);
endmodule
```

```
module and_gate(n, a, b);
    input a,b; output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    restrict(w0)
    relax(x);
endmodule
```

The above examples show the interplay between the relax and restrict annotations. On the left side, the designer intends to relax the accuracy of the elements that affect w0 while keeping the ones that affect x precise; hence relax(w0) and restrict(x). With these two declarations, the NOT(~) operation is not approximated but the AND(&) operation will be approximated. Conversely, in the example on the right, the designer relaxes the accuracy of the elements that affect x excluding that which affects w0. The pair of restrict(w0) and relax(x) imply that the NOT operation is approximated while the and_gate and its AND(&) operation remains precise. The restrict annotation crosses the boundary of instantiated modules. In both examples, the output x carries approximate semantics and needs to be annotated with approximate.

Restricting approximation globally. The restrict annotation

does not have precedence over relax. However, there might be cases where the designer intends to override preceding relax annotations. For instance, the designer might intend to reuse a third-party approximate IP core in a precise setting. Certain approximate outputs of the IP core might be used to drive critical signals such as the ones that feed to the controller state machine, write enable of registers, address lines of a memory module, or even clock and reset. These signals are generally critical to the functionality of the circuit and the designers would want to avoid approximating them. To ensure the precision of these signals Axilog provides the restrict_global annotation that has precedence over relax and relax_local. The restrict_global(arg) implies that any design element that affects arg shall not be subject to any approximation. Note that restrict_global penetrates through the boundaries of instantiated modules. The following code snippet illustrates the semantics of the restrict_global annotation.

```
module and_gate(n,a,b);
  input a,b;
  approximate output n;
  assign n = a & b;
  relax(n);
endmodule
```

```
module nand_gate(x, a, b);
  input a, b; output x; wire w0;
  and_gate a1(w0, a, b);
  assign x = ~w0;
  restrict_global(x);
endmodule
```

In the code, restrict_global(x) precedes the relax(n) in the and_gate module. The restrict_global annotation does not allow any form of relaxation to affect the logic that drives x and therefore it is not declared approximate. The rest of this section discusses language annotations, similar to the approximate annotation, that enable reusability in Axilog.

B. Reuse Annotations

This section describes the abstractions that are necessary for reusing approximate modules. Our principle idea for these language abstractions is maximizing the reusability of the approximate modules across designs that may have different accuracy requirements. Axilog's reuse annotations concisely modify the module interface. These annotations declares which outputs carry approximate semantics and which inputs cannot be driven by relaxed wires without explicit annotations.

Outputs carrying approximate semantics. As mentioned, the designers can use annotations to selectively approximate the design elements in a module. These design elements might have a direct or indirect effect on the accuracy of some of the output ports. An approximate module could be given to a different vendor as an IP core. In this case the reusing designer needs to be aware of the accuracy semantics of the input/output ports without delving into the details of the module. To enable the reusing designer to view the port semantics, Axilog requires that all output ports that might be influenced by approximation to be marked as approximate. Below, the code snippets illustrate the necessity of the approximate annotation.

```
module and_gate(n,a,b);
    input a,b;
    approximate output n;
    assign n = a & b;
    relax(n);
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
endmodule
```

```
module and_gate(n,a,b);
    input a, b;
    output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax(x);
endmodule
```

On the left side, output n carries relaxed semantics due to the relax annotation and is therefore declared as an approximate output. Consequently, the a1 instance in the nand_gate module will cause its x output to be relaxed. Therefore, the x marked as an approximate output. On the right side, the x output is explicitly relaxed and x is marked as an approximate output. Relaxing x also implies that the AND operation is relaxable in the a1 instance. However, the and_gate module here does not carry approximate semantics by default. Therefore, the output of the and_gate is not marked as approximate and the approximation is only specific to the a1 instance.

Critical inputs. At design time, the designer of a module may have no knowledge of the circumstances in which the module will be used. The designer may want to prevent approximation to affect certain inputs, which are critical to the functionality of the circuit. To mark these input ports, Axilog provides critical annotation. Wires that carry approximate semantics cannot drive the critical inputs without designer's explicit permission at the time of reuse.

```
module multiplexer(select, x0, x1, z);
    critical input select;
    input x0, x1; output z;
    assign z = (s == 1) ? x1 : x0;
endmodule
```

In this example, the select input of the multiplexer is declared as critical to prevent approximation to affect it.

Bridging approximate modules to critical inputs. As of yet, Axilog does not allow any wire that is affected by approximation to drive a critical input. However, we recognize that there may be cases when the reusing designer entrusts critical input with an approximate driver. For such situations, Axilog provides an annotation called bridge, which shows designer's explicit intent to drive a critical input by an approximate signal and certifies this connectivity. The example below shows the use of the bridge annotation.

```
module top(x0, x1, z);
    input x0, x1;
    approximate output z; wire s;
    and a1(s, x0, x1);
    relax(s); bridge(s);
    multiplexer m1(s, x0, x1, z);
endmodule
```

In this code, the designer annotation relaxes the logic driving s that is connected to a critical input select of multiplexer. This connectivity therefore requires designer's consent. The bridge(s) annotation certifies the connectivity of approximated signal s to the select critical input of the m1 instance of the multiplexer module.

In summary, the semantics of the relax and restrict annotations provides abstractions for designing approximate hardware modules while enabling Axilog to provide *formal guarantees* of safety that the approximation will only be restricted to the design elements that are specifically selected by the designer. Moreover, the approximate output, critical input, and bridge annotations enable reusability of the modules across different designs. In addition to the modularity, the design and reuse annotations altogether enable *approximation polymorphism* in hardware design. That is, with Axilog, the modules with approximate semantics can be used in a precise manner without reimplementation and conversely precise modules can be instantiated with approximate semantics. These abstractions provide a natural extension to the current practices of hardware design and enable the designer to apply approximation with full control without adding substantial overhead to the conventional hardware design and verification cycle.

III. Relaxability Inference Analysis

After the designer provides annotations, the compiler needs to perform a static analysis to find the approximate and precise design elements in accordance with these annotations. This section presents the *Relaxability Inference Analysis*, a static analysis that identifies these relaxable gates and connections. To simplify the implementation, we first translate the RTL Verilog design to primitive gates, while maintaining the module boundaries. We then apply the *Relaxability Inference Analysis* at the gate level. The *Relaxability Inference Analysis* is a backward slicing algorithm that starts from the annotated wires and iteratively traverses the circuit to identify which wires must carry precise semantics. Subtracting the set of precise wires from all the wires in the circuit yields the relaxable set of wires. The gates that immediately drive these relaxable wires are the ones that the synthesis can potentially approximate. Algorithm 1 illustrates the procedure that identifies the precise wires.

Inputs: $\mathbb{K}$: Circuit-under analysis $\mathbb{M}$: Set of all the modules within the circuit
$\mathbb{R}$: Set of all the globally restricted wires
Output: $\mathbb{P}$: Set of precise wires

```
Initialize ℙ ← ∅
for each m_i ∈ 𝕄 do
    I: Set of all the inputs ports in m_i        A: Set of all the relaxed wires in m_i
    LA: Set of all the locally relaxed wires in m_i
    Sink: Set of all the restricted wires in m_i ∪ Set of unannotated output ports
    UW: Set of wires driven by modules that are instantiated within m_i
    //Phase1: This loop identifies the m_i module's local precise wires (w_i)
    Initialize N ← ∅
    while (Sink ≠ ∅) do
        w_i ← dequeue(Sink)
        if (w_i ∉ I and w_i ∉ (A ∪ LA)) then
            if (w_i ∈ UW) then
                N.append(w_i)
            else
                ℙ.append(w_i)
            end if
            enqueue(Sink, for all the input wires of the gate that w_i in m_i)
        end if
    end while
    //Phase2: This loop identifies the relaxed wires (w_j) that are driven by the
    m_j submodules; the m_j submodules are the instantiated modules in m_i
    for (w_j ∈ UW) do
        if (w_j ∉ N and w_j drives wire ∈ A) then
            m_j = module driving the wire w_j
            m_j.A.append(w_j)
        end if
    end for
end for
//Phase3: This loop identifies the precise wires (w_k) that are globally restricted
while (ℝ ≠ ∅) do
    w_k ← dequeue(ℝ)
    ℙ.append(w_k)
    ℝ.append(Set of all the input wires of the gate that is driving w_k)
end while
```

Algorithm 1: Backward flow analysis for finding precise wires.

This procedure is a *backward-flow* analysis that operates in three phases: (1) **The first phase** starts by identifying a set of *sink* wires. The sink wires are either unannotated outputs or wires that are *explicitly* annotated with restrict. The procedure identifies the gates that are driving the sink wires and adds their input wires to the precise set if they are not explicitly annotated as relaxed. The algorithm repeats this step for the newly added wires until it reaches an input or an explicitly relaxed wire. However, this phase is only limited to the scope of the module-under-analysis; (2) **In the second phase**, the algorithm identifies the relaxed outputs of the instantiated submodules. Due to the semantic differences between relax and relax_local, the output of a submodule will be considered relaxed if the following two conditions are satisfied. (a) The output drives another explicitly relaxed wire, which is not inferred due to a relax_local annotation; and (b) the output is not driving a wire already identified as precise. The algorithm automatically annotates these qualifying outputs as relaxed. The analysis repeats these two phases for all the instantiated submodules. For correct functionality of this analysis, all the module instantiations are distinct entities in the set $\mathbb{M}$ and are ordered hierarchically; (3) **In the final phase**, the algorithm marks any wire that affects a globally restricted wire as precise. This final phase allows the restrict_global to override any other annotations in the design.

Finally, the Relaxability Inference Analysis–part of which is presented in Algorithm 1–identifies the safe-to-approximate subset of the gates and wires with regards to the designer annotations. An approximation-aware synthesis tool can then generate an optimized netlist, with the approximation applied to only the safe-to-approximate circuit elements.

Axilog's language semantics and the Relaxability Inference Analysis are independent of the approximate synthesis. That is, Axilog abstracts away the details of the approximate synthesis and relieves the designer from its specifics. Axilog can be used with virtually any approximate synthesis tool.

IV. Approximate Synthesis

In our framework, the synthesis tool first takes in the annotated Verilog source code and produces a gate-level netlist without employing any approximate optimizations. However, the synthesis tool preserves the approximate annotations. Then, the Relaxability Inference Analysis identifies the safe-to-approximate subset of the gates and wires with regards to the designer annotations. In the next step, the synthesis tool applies approximate synthesis and optimization techniques *only* to the safe-to-approximate circuit elements. The tool has the liberty to apply any approximate optimization technique including gate substitution, gate elimination, logic restructuring, voltage over-scaling, and timing speculation as it deems prudent. The objective is to minimize a combination of error, delay, energy, and area considering final quality requirements. Figure 1 shows one such approximate synthesis technique. Our synthesis technique uses commercial tools to selectively relax timing requirements on safe-to-approximate paths of the circuit. As shown in Figure 1a, we first use Synopsys Design Compiler to synthesize the design with no approximation. We perform a multi-objective optimization targeting the highest frequency while minimizing power and area. We will refer to the resulting netlist as the baseline netlist and its frequency as the baseline frequency. We account for variability by using Synopsys PrimeTimeVX which, given timing constraints, provides the probability of timing violations due to variations. In case of violation, the synthesis process is repeated by adjusting timing constraints until PrimeTimeVX confirms no violations.

Second, as shown in Figure 1b, we selectively relax the timing constraints and provide more slack on the safe-to-approximate paths. For the precise paths, the timing constraints are set to the most strict level (the baseline frequency). We then extract the post-synthesis gate delay information in Standard Delay Format (SDF) and perform gate-level timing simulations with a set of input datasets. We use the baseline frequency for the timing simulations even though some of the safe-to-approximate paths are synthesized with more timing slack. The timing simulations yield a set of output values that may incur quality loss since the approximated paths in the circuit may not generate the correct output at the baseline frequency. We then measure the quality loss and if the quality loss is more than designer's requirements, we tighten the timing constraints

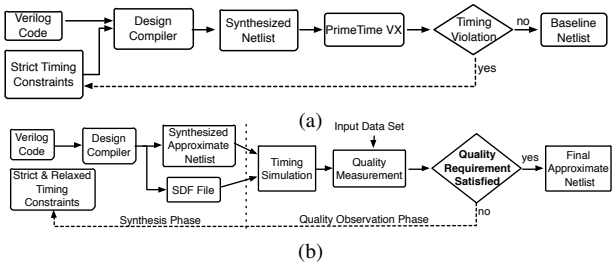


Fig. 1: Synthesis flow for (a) baseline and (b) approximate circuits.

on the safe-to-approximate paths. We repeat this step until the designer quality requirements are satisfied. This methodology has a potential to reduce energy and area by utilizing slower and smaller gates in the safe to approximate paths in which we use relaxed timing constraints.

V. Evaluation

To evaluate the effectiveness of Axilog, we annotate several benchmark designs and apply our Relaxability Inference Analysis and synthesis flow.

Benchmarks and Code Annotation. Table II lists the design benchmarks implemented in Verilog. We use Axilog annotations to judiciously relax some of the circuit elements. The benchmarks span a wide range of domains including arithmetic units, signal processing, robotics, machine learning, and image processing. Table II also includes the input datasets, application-specific quality metrics, number of lines, and number of Axilog annotations for design and reuse.

Axilog annotations. We annotated the benchmarks with the Axilog extensions. The designs were either downloaded from open-source IP providers or developed without any initial annotations. After development, we analyzed the source Verilog codes to identify relaxable parts. The last two columns of Table II show the number of design and reuse annotations for each benchmark. The number of annotations range from 2 for Brent-Kung with 352 lines to 12 for InverseK with 22,407 lines. The Axilog annotation coupled with the Relaxability Inference Analysis has enabled us to only use a handful of annotations to effectively approximate designs that are implemented with thousands of lines of Verilog.

The relaxable parts are more common in datapath of the benchmarks designs rather than their control logic. For example, K-means involves a significant number of multiplies and additions before the calculated result can be written in a memory module. We used the relax annotations to declare these arithmetic operations approximable; however, we used restrict to ensure the precision of all the control signals. For smaller benchmarks, such as Brent-Kung, Kogge-Stone and Wallace Tree, only a subset of the least significant output bits were annotated to limit the quality loss. To be able to reuse some of the design, we also annotated the benchmarks with reuse annotations. The number of this type of annotation are listed in the last column of Table II. For example, the add_sub signal that selects the addition and subtraction operation for an ALU is annotated with the critical reuse annotation. Overall, one graduate student was able to annotate all the benchmarks within two days without being involved in their design. The intuitive nature of the Axilog extensions makes annotating straightforward.

Application-specific quality metrics. Table II shows the application-specific error metrics to evaluate the quality loss due to approximation. Using application-specific quality metrics is commensurate with prior work on approximate computing and language design [18, 19]. In all cases, we compare the

TABLE II: Benchmarks, input datasets, and error metrics.

Benchmark Name	Domain	Input Data Set	Quality Metric	# Lines	# Annotations Design	# Annotations Reuse
Brent-Kung (32-bit adder)	Arithmetic Computation	1,000,000 32-bit integers	Avg Relative Error	352	1	1
FIR (8-bit FIR filter)	Signal Processing	1,000,000 8-bit integers	Avg Relative Error	113	6	5
ForwardK (forward kinematics for 2-joint arm)	Robotics	1,000,000 32-bit fixed-point values	Avg Relative Error	18,282	5	4
InverseK (inverse kinematics for 2-joint arm)	Robotics	1,000,000 32-bit fixed-point values	Avg Relative Error	22,407	8	4
K-means (K-means clustering)	Machine Learning	1024x1024-pixel color image	Image Diff	10,985	7	3
Kogge-Stone (32-bit adder)	Arithmetic Computation	1,000,000 32-bit integers	Avg Relative Error	353	1	1
Wallace Tree (32-bit Multiplier)	Arithmetic Computation	1,000,000 32-bit integers	Avg Relative Error	13,928	5	3
Neural Network (feedforward neural network)	Machine Learning	1024x1024-pixel color image	Image Diff	21,053	4	3
Sobel (sobel edge dectector)	Image Processing	1024x1024-pixel color image	Image Diff	143	6	3

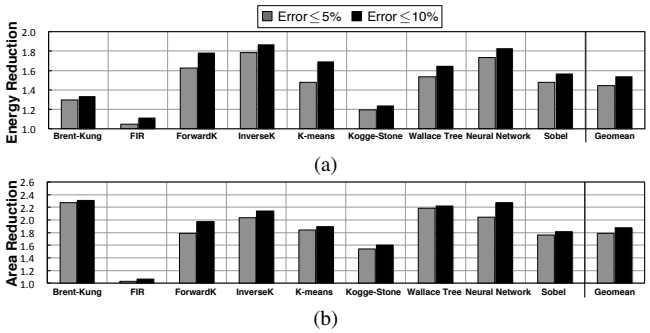
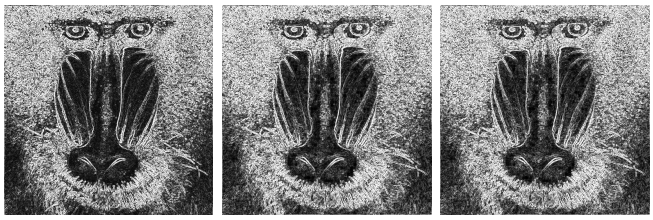


Fig. 2: Reductions in (a) energy and (b) area when the quality degradation limit is set to 5% and 10% in the synthesis flow.

output of the original baseline application to the output of the approximated design. For the benchmarks which generate numeric outputs, including brent-kung adder, FIR filter, forward kinematics, inverse kinematics, kogge-stone adder, and wallace tree multiplier, we measure the average relative error. For the neural network, kmeans clustering, and sobel edge detection applications, which produce images, we use the average root-mean-square image difference.

Tools and experimental setup. We use Synopsys Design Compiler (G-2012.06-SP5) and Synopsys PrimeTime (F-2011.06-SP3-2) for synthesis and energy analysis, respectively. We use Cadence NC-Verilog (11.10-s062) for timing simulation with SDF back annotations extracted from various operating corners. We use the TSMC 45-nm multi-V_t standard cells libraries and the primary results are reported for the slowest PVT corner (SS, 0.81V, 0°C).

Experimental results. Figure 2 illustrates the energy savings (2a) and area reduction (2b) when the quality loss limit is set to 5% and 10% in our synthesis flow. The baseline is synthesis with no approximation. With the 5% limit, our framework achieves, on average, 45% energy and 1.8× area reduction, respectively. When the quality loss limit is set to 10%, the average gains grow to 54% energy reduction and 1.9× area reduction. The Axilog annotations force the control logic in these benchmarks to be precise. Therefore, the benchmarks such as InverseK, Wallace Tree, Neural Network, and Sobel–that have a larger datapath–provide a larger scope for approximation and are usually the ones that see larger benefits. The structure of the circuit also affects the potential benefits. For instance, Brent-Kung and Kogge-Stone adders benefit differently from approximation due to the structural differences in their logic trees. The FIR benchmark shows the smallest energy savings since it is a relatively small design which does not provide many opportunities for approximation. Nevertheless, FIR still achieves 11% energy savings and 7% area reduction with 10% quality loss. This result suggests



(a) 0% Quality Loss (b) 5% Quality Loss (c) 10% Quality Loss

Fig. 3: Visual depiction of the output quality degradation with approximate synthesis for the Sobel application.

TABLE III: The energy reduction when the quality degradation limit is set to 10% for two different PVT corners. Here, we consider temperature variations.

PVT Corners	Brent-Kung	FIR	ForwardK	InverseK	K-means	Kogge-Stone	Wallace Tree	Neural Network	Sobel	Geomean
(SS, 0.81V, 0°C)	34%	11%	78%	87%	69%	24%	65%	83%	57%	54%
(SS, 0.81V, 125°C)	32%	7%	72%	79%	65%	21%	63%	72%	41%	48%

that even designs with limited opportunities for approximation can benefit significantly from the precisely targeted relaxation that Axilog provides. We evaluate the effectiveness of our technique in the presence of temperature variations for a full industrial range of 0°C to 125°C. We measured the impact of temperature fluctuations on the energy benefits for the same relaxed designs. Table III compares the energy benefits at the lower and higher temperatures (the quality loss limit is set to 10%). In this range of temperature variations, the average energy benefits ranges from 54% (at 0°C) to 48% (at 125°C). These results confirm the robustness of our framework that yields significant benefits even when temperature varies.

We visually examine the output of the Sobel application,which generates an image. Figure 3 displays the output with 0% (no approximation), 5%, and 10% quality degradation. Interestingly, even 10% quality loss is nearly indiscernible to the eye. Nevertheless, for the 10% error level approximate synthesis provides 57% energy saving and 1.82× area reduction.

These results suggest that Axilog can achieve significant savings while preserving the application functionality. This tradeoff is attainable because the high-level language annotations and design abstractions allow the designer to target approximation where it is most effective without compromising the critical parts of the computation. Furthermore, the synthesis tunes the approximate parts of the circuit within the quality constraints specified by the designer. Axilog thereby achieves a balance between quality and efficiency which is advantageous for the specific application.

VI. Related Work

A growing body of research shows the applicability and significant benefits of approximation [2–16]. However, prior research has not explored extending hardware description languages for systematic and reusable approximate hardware design. Below, we discuss the most related works.

Approximate programming languages. EnerJ [18] provides a set of type qualifier to manually annotate all the approximate variables in the program. If we had extended EnerJ's model to Verilog, the designer would have had to manually annotate all approximate wires/regs. Rely [19] asks for manually marking both approximate variables and operations, which requires more annotations. The work in [20] proposes language extension to the OpenMP software programming language that allows programmers to manually specify approximable regions of code. With our abstractions, the designer marks a few wires/regs and then the analysis automatically infers which other connections and gates are safe to approximate.

Approximate circuit design and synthesis. Prior work proposes imprecise implementations of custom instructions [17] and specific hardware blocks [4, 5, 7–10]. The work in [6, 11–16] propose algorithms for approximate synthesis that leverages gate pruning, timing speculation, or voltage overscaling. While all these synthesis techniques provide significant improvements, they do not focus on providing hardware description language semantics for methodical approximate hardware design and reuse. In fact, our framework can benefit and leverage all these synthesis techniques.

VII. Conclusion

Axilog provides a less arduous framework compared to a mere extension of existing approximate programming models for hardware design. Axilog's automated analysis enables the designers to approximate hardware without delving deeper into the intricacies of synthesis and optimization. Furthermore, all the abstractions presented in this paper are concrete extensions to the mainstream Verilog HDL providing designers with backward compatibility. We evaluated Axilog, its automated Relaxability Inference Analysis, and the presented approximate synthesis and demonstrate 54% average energy savings and 1.9× area reduction with merely 2 to 12 annotations per benchmark. These results confirm that Axilog is a methodical step toward practical approximate hardware design and reuse.

VIII. Acknowledgement

This work was supported in part by a Qualcomm Innovation Fellowship, Semiconductor Research Corporation contract #2014-EP-2577, and a gift from Google.

References

[1] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[2] L. N. Chakrapani, P. Korkmaz, B. E. Akgul, and K. V. Palem, "Probabilistic system-on-a-chip architectures," *in TODAES*, 2007.

[3] H. Cho, L. Leem, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," *in TCAD*, 2012.

[4] V. Gupta *et al.*, "IMPACT: imprecise adders for low-power approximate computing," in *ISLPED*, 2011.

[5] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *ICCAD*, 2013.

[6] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *DATE*, 2010.

[7] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSI*, 2011.

[8] A. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012.

[9] D. Mohapatra *et al.*, "Design of voltage-scalable meta-functions for approximate computing," in *DATE*, 2011.

[10] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, 2004.

[11] S. Venkataramani *et al.*, "Salsa: systematic logic synthesis of approximate circuits," in *DAC*, 2012.

[12] K. Nepal, Y. Li, R. Bahar, and S. Reda, "Abacus: a technique for automated behavioral synthesis of approximate computing circuits," in *DATE*, 2014.

[13] Y. Liu *et al.*, "On logic synthesis for timing speculation," in *ICCAD*, 2012.

[14] A. Lingamneni *et al.*, "Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling," in *CF*, 2012.

[15] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD*, 2013.

[16] S. Ramasubramanian *et al.*, "Relax-and-retime: A methodology for energy-efficient recovery based design," in *DAC*, 2013.

[17] M. Kamal, A. Ghasemazar, A. Afzali-Kusha, and M. Pedram, "Improving efficiency of extensible processors by using approximate custom instructions," in *DATE*, 2014.

[18] A. Sampson *et al.*, "EnerJ: Approximate data types for safe and general low-power computation," *PLDI*, 2011.

[19] M. Carbin, S. Misailovic, and M. Rinard, "Verifying quantitative reliability of programs that execute on unreliable hardware," 2013.

[20] A. Rahimi, A. Marongiu, R. Gupta, and L. Benini, "A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters," in *CODES+ISSS*, 2013.