Methodical Approximate Hardware Design and Reuse

Amir Yazdanbakhsh Bradley Thwaites Jongse Park Hadi Esmaeilzadeh
Georgia Institute of Technology
{a.yazdanbakhsh, bthwaites, jspark}@gatech.edu hadi@cc.gatech.edu

Abstract

Design and reuse of approximate hardware components—digital circuits that may produce inaccurate results—can potentially lead to significant performance and energy improvements. Many emerging error-resilient applications can exploit such designs provided approximation is applied in a controlled manner. This paper provides the design abstractions and semantics for methodical, modular, and controlled approximate hardware design and reuse. With these abstractions, critical parts of the circuit still carry the strict semantics of traditional hardware design, while flexibility is provided. We discuss these abstractions in the context of synthesizable register transfer level (RTL) design with Verilog. Our framework governs the application of approximation during the synthesis process without involving the designers in the details of approximate synthesis and optimization. Through high-level annotations, our design paradigm provides high-level control over where and to what degree approximation is applied. We believe that our work forms a foundation for practical approximate hardware design and reuse.

1. Introduction

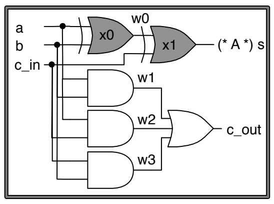
As process technology scales to atomic levels, providing the traditional abstraction of near-perfect accuracy at the circuit level imposes high taxes in terms of performance and energy efficiency [3,7]. Relaxing this abstraction and moving toward a "methodical" approximate hardware design—where parts of the circuit may generate approximate outputs—can potentially unleash considerable benefits in both efficiency and performance. There is in fact an emerging opportunity to avoid such high taxes due to a growing body of prominent applications that are inherently robust to inaccuracies [1,4,6,8,9,11,12,20]. Hardware designers have an opportunity to exploit this property by only providing strict accuracy when and where it is required in the system. However, such a radical departure in digital hardware design requires design abstractions that allow designers to reason about and delineate which part of the hardware system or circuit is "critical" and cannot be approximated. These design abstractions also need to provide an option to the designers to control the error levels as approximation is applied to the different parts of the design. Furthermore, hardware systems implementation relies on modular design practices where the engineers build libraries of modules[1] and compose them to build a more complex hardware system, e.g., a system-on-a-chip (SoC). Further, many complex SoC designs are composed of semiconductor intellectual property cores (IP cores) that are designed and sold by different vendors. In this industrial ecosystem, reusing IP cores is imperative and is a major motivating factor for innovation and entrepreneurship. Generally, hardware systems design cycle has two phases: (1) the "design phase" when engineers design the IP cores and (2) the "reuse phase" when the engineers incorporate the IP cores in a larger system. This paper describes the necessary system design abstractions and semantics that enable "methodical and controlled approximate hardware design, description, and reuse."

In order to incorporate approximation in such a modular design ecosystem while supporting approximation in both phases, our framework provides these four fundamental design abstractions:

1. **[Design Phase]** Design abstractions for delineating which parts of a hardware module can be approximated safely using an *approximation plan* for the module (Section 2).
2. **[Design Phase]** Design abstractions for interfacing approximate and precise hardware modules (Section 3).
3. **[Reuse Phase]** Design abstractions for overriding the approximation plan (Section 4).
4. **[Reuse Phase]** Design abstractions that enable designers to guide the approximate synthesis process without involving them in how approximate synthesis and optimization is applied (Sections 5 and 6).

We provide concrete extensions to the Verilog hardware description language to demonstrate the necessity and effectiveness of these design abstractions. Furthermore, state of the art programming languages for approximation such as EnerJ [21] and Rely [2] require programmers to *manually and explicitly* declare low-level details such as the specific variables and operations that can be approximated. In contrast, we devise concise, intuitive, and high-level semantics that enable hardware designers to rely on an *automatic* synthesis process to discover where and how to apply approximation. Our approximate hardware design semantics lower the restrictions of the typical hardware design and synthesis cycle, which aims to optimize for the worst case conditions. In this realm of approximate hardware design, our abstractions govern the synthesis process, which needs to inevitably incorporate "selectively" relaxed semantics. Through explicit constraints, our system allows designers to fully specify the functional characteristics of their designs with respect to the degree of approximation applied at a high level of abstraction without concern for the details of synthesis and optimization. While prior work has focused on synthesis and optimization of functional units with

[1]In this paper, we refer to a hardware module as a building blocks of a hardware system that can potentially be reused across many different designs.



(a) Full adder design

```
module fa(a, b, c_in, c_out, s);
    input a, b, c_in;
    output c_out;
    (*A*) output s;
    wire w0, w1, w2, w3;

    xor x0(w0, a, b);
    xor x1(s, w0, c_in);

    and u2(w1, a, b);
    and u2(w2, a, c_in);
    and u2(w3, b, c_in);
    or  u4(c_out, w1, w2, w3);
endmodule
```

(b) Approximate full adder in Verilog

Figure 1: Approximation plan for a full adder. Shaded gates can be approximated.

approximate semantics [10, 13, 14, 18, 22–24], our framework enables a modular and methodical approach toward designing and reusing approximate hardware "systems."

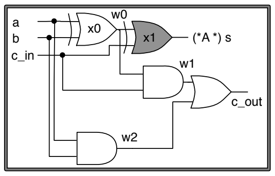
2. Approximation Plan

In this section, we describe how a designer specifies an *approximation plan* for a hardware module. In our framework, an approximation plan *implicitly* identifies which part of the module can be approximated by the synthesis tool. For simplicity, we first describe the approximation plan only within a module, leaving the details of reuse and more complex designs to Sections 4 and 6.

Figure 1a shows a full adder, in which s is the sum of the three inputs, a, b, c_in, and c_out is the carry out. Suppose the designer intends to allow the logic that produces the sum, s, to be approximate while keeping the logic for c_out precise. One option is to allow the designer to *explicitly* mark the XOR gates in 1a as approximate units. However, we find this approach to be burdensome. Instead, we only require the designer to declare the wire s as an approximate signal. Then, the compiler will perform a static analysis and automatically identify the hardware elements that are candidates for approximation. In Figure 1a, as the designer declares s as approximate, the static analysis will identify that the two XOR gates that contribute to s's value are approximable. With this approach, the designer does not need to declare any other wires including a, b, c_in, and w nor any of the XOR gates as approximate. Thus, this abstraction significantly reduces the burden of the designer to analyze and understand complex data flows throughout the circuit. She only intuitively declares a wire as approximate and the static analysis automates the rest.

For backward compatibility, all the wires and units are precise by default. Thus, an unmodified Verilog code will produce the expected results. Therefore, in Figure 1a, the unmarked c_out signal and ANDs, wires, and ORs generating c_out will be precise.

To support this approximate design methodology, we introduce one new language construct to Verilog to allow approximate declarations. This construct, **(*A*)**[2], is an attribute that can be attached to any wire[3] in the design. Figure 1b shows the Verilog implementation of the full adder. Notice that in our framework, there is no notion of approximate inputs. Within a module, the designer does not have control over the precision of the inputs, only how the logic inside the module operates on those inputs.



(a) Full adder design

```
module fa(a, b, c_in, c_out, s);
    input a, b, c_in;
    output c_out;
    (*A*) output s;
    wire w0, w1, w2, w3;

    xor x0(w0, a, b);
    xor x1(s, w0, c_in);

    and u2(w1, c_in, w0);
    and u2(w2, a, b);
    or u2(c_out, w2, w1);
endmodule
```

(b) Approximate full adder in Verilog

Figure 2: Approximation plan for a full adder. Only the one shaded gate can be approximated.

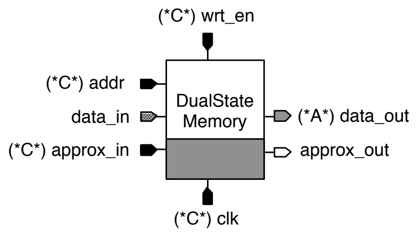
In many cases, the logic which produces an approximate signal may also contribute to a precise signal at some intermediary stage. During static analysis, we maintain the property that any precise signal will not be influenced by approximate logic, providing a *guarantee* of safety in our approximate design paradigm. Figure 2a shows an optimized full adder in which, again, s is an approximate signal while c_out is precise. Since x1 only influences an approximate wire, it is a candidate for approximation. However, x0 generates a signal which propagates to both approximate and precise wires. In this situation, the safety property must be maintained, so x0 must be implemented precisely. Our static analysis will provide this guarantee (Section 3). In Sections 4 and 6, we will provide the abstractions to control quality.

3. Approximate Interface

The ability to reuse components in a modular way is critical to modern industrial hardware systems design. Before we discuss the reuse of approximate modules in a full system, we describe the interface abstractions through which each approximate module communicates with the rest of the system. These abstractions define the external view of the module. The interface of a module consists of its inputs and outputs. Each module must declare which outputs produce approximate results. The default assumption is that if an output is not declared approximate, then it always produces precise results under all circumstances. Therefore, any outputs that have any chance of being influenced by approximation *within the module* must be declared approximate. We use the same **(*A*)**

[2] Verilog 2011 allows specifying attributes for wire, module, ... through the (*ATTRIBUTE*) construct.

[3] In Verilog, the wire, reg, and output keywords can be used to declare a physical wire. Our attribute can be attached to all these keywords.



(a) Approximation interface of a dual-state memory

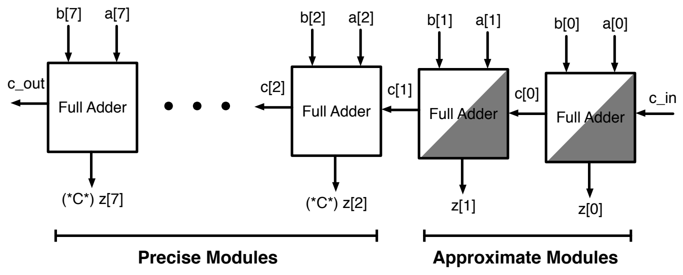
```
module DualStateMemory(
    clk, wrt_en,
    address,
    data_in, approx_in,
    data_out, approx_out);

    (*C*) input clk;
    (*C*) input wrt_en;
    (*C*) input[N-1:0] address;
    input[M-1:0] data_in;
    (*C*) input approx_in;
    (*A*) output[N-1:0] data_out;
    output approx_out;
    ...
endmodule
```

(b) Approximation interface for a dual-state memory unit in Verilog

Figure 3: Approximation interface for a memory. The shaded gate can be approximated.

notation to declare outputs as approximate. At design time, the designer of a module will have no knowledge of whether approximation techniques have been applied to the inputs. However, the designer may want to impose more stringent requirements on certain inputs. Example may include clocks and write-enables which are critical to the functionality of the approximate module when instantiated and reused in a larger hardware system. Therefore, we introduce a new construct, **(*C*)**, which declares an input critical. Semantically, any wire which is influenced by approximation cannot be connected to a critical input. These rules define the approximate interfaces of the module. Figure 3 shows an interface for a simple memory module capable of reading and storing both precise and approximate data. This module either writes to or reads from addr at the rising edge of each clock, depending on the value of wrt_en. Suppose if the value of approx_in is true, then data can be written to an approximate memory cell, otherwise it must be stored in a precise manner. While data_in can carry either precise or approximate data, it could be devastating to the functionality of the module if any of the other inputs have been computed approximately. For example, an error in approx_in could cause important precise data to be written to approximate storage, introducing unacceptable behavior. Thus, the critical inputs are marked **(*C*)** and the module designer is assured that these signals will never be affected by approximate operations. An analogous situation is present in the outputs of the module in Figure 3. The signal approx_out is *not* marked as approximate, indicating that no approximate operations were applied to this output at any point within the scope of



(a) Overriding approximation in an adder

```
module adder(a, b, c_in, c_out, z);
    input[7: 0] a, b;
    input c_in;
    (*A*) output[7: 0] z;
    output c_out;

    (*C*) wire[7: 2] z;
    wire[6:0] c;

    fa u0(a[0], b[0], c_in, c[0], z[0]);
    fa u1(a[1], b[1], c[0], c[1], z[1]);

    fa u2(a[2], b[2], c[1], c[2], z[2]);
    fa u3(a[3], b[3], c[2], c[3], z[3]);
    fa u4(a[4], b[4], c[3], c[4], z[4]);
    fa u5(a[5], b[5], c[4], c[5], z[5]);
    fa u6(a[6], b[6], c[5], c[6], z[6]);
    fa u7(a[7], b[7], c[6], c_out, z[7]);
endmodule
```

(b) Overriding approximation in Verilog for an adder

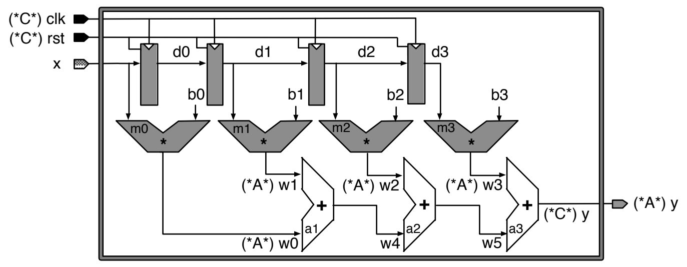
Figure 4: Approximation interface for a memory. The shaded gate is approximate.

the module. Similarly, even though data_out still sometimes holds precise values, it must be marked **(*A*)** because there is a possibility of approximation during its computation.

4. Overriding Approximation and Bridging

Here we focus on the controlled reuse of approximate modules.

Overriding approximation. While the approximation plan defines where approximation is allowed within the module, the system designer must be able to control approximation when instantiating the module in a system. For example, as Figure 4a illustrates, a designer may want to preserve precise semantics for the most significant bits of an adder while allowing approximation in the least significant bits. In this case, the designer needs to override the original full adder approximation plan when instantiating the full adders producing the most significant bits. The mechanism we provide for overriding is to connect a critical wire to the approximate output of the module to be overridden. As Figure 4b shows, we extend the Verilog language to allow redeclaring part of the output vector z as critical using **(*C*)**. Since full adders u7 to u2 are connected to a critical wire, the compiler will not mark them as approximable. In fact, any logic contributing to a critical wire will not be approximated, except in exceptional cases which we describe shortly. Notice that in terms of interfacing the z output is still an approximate output from an outside point of view. Figure 5 describes a more complicated



(a) Overriding with critical wires and approximate wires in a finite impulse response (FIR) filter

```
module fir(clk, rst, x, y);
    (*C*) input clk, rst;
    input[N-1:0] x;
    (*A*) output[M-1:0] y;
    (*C*) wire signed[M-1:0] y;
    parameter b0 = 5;
    parameter b1 = 6;
    parameter b2 = -7;
    parameter b3 = 1;
    reg signed[N-1:0] d0, d1, d2, d3;
    (*A*) wire signed[N-1:0] w0, w1, w2, w3;
    wire signed[N-1:0] w4, w5;

    always @(posedge clk) begin
        if (rst == 1) begin
            d0 <= 0;
            d1 <= 0;
            d2 <= 0;
            d3 <= 0;
        end else begin
            d0 <= x;
            d1 <= d0;
            d2 <= d1;
            d3 <= d2;
        end
    end
    assign w0 = b0 * x;
    assign w1 = b1 * d1;
    assign w2 = b2 * d2;
    assign w3 = b3 * d3;
    assign w4 = w0 + w1;
    assign w5 = w2 + w4;
    assign y  = w3 + w5;
endmodule
```

(b) Overriding with critical wires and approximate wires in a finite impulse respose (FIR) filter in Verilog

Figure 5: Overriding with critical wires and approximate wires in a filter. The shaded components are approximate.

situation involving a finite impulse response (FIR) filter. Suppose the implementation details require that the adder units be implemented precisely, but the multipliers and registers can be implemented approximately. Since the final output y is affected by approximation in some way within the scope of the FIR module, it must be labeled approximate for interfacing. Without overrides, this would imply that all operations leading to y are candidates for approximation. In order to force a precise implementation for a3, the designer must override its output wire y using **(*C*)**. Now, this override alone would require the entire module to be implemented precisely, but desired situation can be achieved by adding additional overrides. By marking w0, w2, and w4 with **(*A*)**, the precision requirement is overridden for all operations leading to the generation of those wires. In this manner, the designer has flexibility and

```
module sobel(p0, p1, p2, p3, p5, p6, p7, p8, out);
    input[7: 0] p0, p1, p2, p3, p5, p6, p7, p8;
    (*A:PixelError<0.1*) output[7: 0] out;

    wire signed[10: 0] gx, gy;
    wire signed[10: 0] abs_gx, abs_gy;
    wire[10: 0] sum;
    wire[7: 0] out;

    assign gx     = ((p2-p0)+((p5-p3)<<1)+(p8-p6));
    assign gy     = ((p0-p6)+((p1-p7)<<1)+(p2-p8));
    assign abs_gx = (gx[10]? ~gx+1 : gx);
    assign abs_gy = (gy[10]? ~gy+1 : gy);
    assign sum    = (abs_gx+abs_gy);
    assign out    = (|sum[10: 8])?8'hff : sum[7: 0];
endmodule
```

Figure 6: Constraining approximation in sobel filter.

control over the granularity of approximation.

Approximation bridging There is a fundamental difference between a critical wire and a critical input. A critical wire overrides the approximation plan of the logical slice that is driving that wire. A critical input is a mere declaration specifying that no wire carrying approximate semantics can be connected to this input when the module is being reused. The compiler will produce an error when any wire that was affected by approximation is connected to a critical input since it is a clear violation of module designer's intent. This strict interface provides confidence for designers who may not be aware of all implementation details in a reusable lower level module. However, we recognize that there may be cases when the designer trusts an approximate input, but would still like to reuse a module with an input declared as critical. For this situation, we introduce an approximation bridge, indicated with the annotation **(*B*)**, which is used to certify that a signal affected by approximation can be connected to a critical input. Figure 8a illustrates an example in which a dual state memory, which can store both approximate and precise values, has a critical input called addr. Under normal circumstances, it would be crucial for the memory to keep a clean separation between approximate and precise values, which makes addr an obvious choice as a critical input. However, suppose the designer knows ahead of time that only approximate data will ever be stored in this particular instance. In this case, he can create an approximation bridge between the output of the approximate address generator and the addr input. Bridging does not change the precision level of a wire, it only enables a connection. Any output that is affected by the bridged wire will still carry approximate semantics.

5. Approximation Safety Analysis

As mentioned before, the synthesis tool performs a safety analysis on the unapproximated gate-level netlist of the circuit. Figure 9 conceptually illustrates the safety property of our analysis. Each triangle represent the slice of gates that produce each of the outputs. As depicted, the intersection of the slice that produces a precise output, z, and the slice that produces an approximate output must be precise. Furthermore, as Figure 10

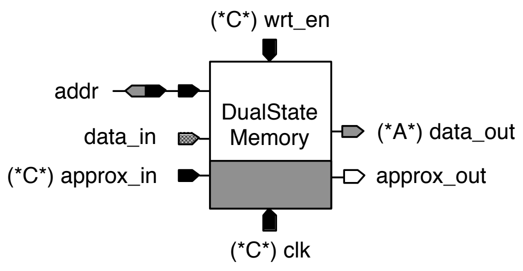
```
module sobel(p0, p1, p2, p3, p5, p6, p7, p8, out);
    input[7: 0] p0, p1, p2, p3, p5, p6, p7, p8;
    output[7: 0] out;

    (*A*) wire signed[10: 0] gx, gy;
    (*A*) wire signed[10: 0] abs_gx, abs_gy;
    (*A*) wire[10: 0] sum;
    (*A*) wire[7: 0] out;

    assign gx = ((p2-p0)+((p5-p3)<<1)+(p8-p6));
    assign gy = ((p0-p6)+((p1-p7)<<1)+(p2-p8));
    assign abs_gx = (gx[10]? ~gx+1 : gx);
    assign abs_gy = (gy[10]? ~gy+1 : gy);
    assign sum = (abs_gx+abs_gy);
    assign out = (|sum[10: 8])?8'hff : sum[7: 0];

endmodule
```

Figure 7: Approximate implementation of sobel filter by EnerJ [21]-like extensions to Verilog. With EnerJ-like model the designer must explicitly declare all the wires that are safe to approximate.



(a) Critical bridge in a memory design

```
module Mem(...);
    ...
    DualStateMemory u0(.clk(clk), .wrt_en(wrt_en),
                       .address( (*B*) addr),
                       .data_in(data_in),
                       .approx_in(approx_in),
                       .data_in(data_in),
                       .data_out(data_out),
                       .approx_out(approx_out));
    ...
endmodule
```

(b) Overriding approximation in Verilog in a memory design

Figure 8: Critical bridge is used to connect an approximate signal to a critical input in a dual-state memory. The shaded parts are approximate.

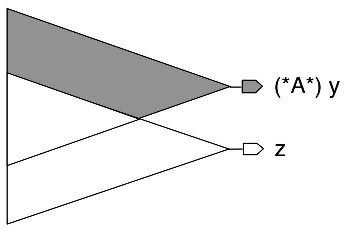


Figure 9: The compiler will perform a bi-directional safety analysis and mark hardware component approximable that do not contribute to any precise signals. Here, output z is precise while output y is approximate. The shared logic that contributes to both outputs is kept precise to guarantee safety and ensure that output z always carries precise semantics.

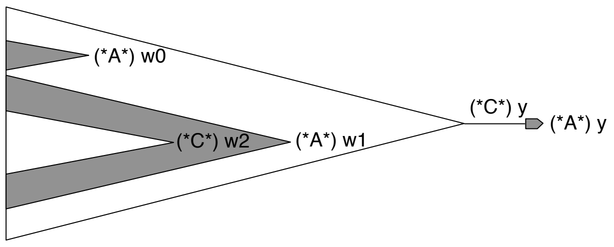


Figure 10: Multiple overriding as conceptually depicted here. Since approximate circuit components contribute to the value y, it is an approximate output.

shows, the safety analysis must ensure that the slices that produce overriding critical wires are precise.

To perform the static safety analysis, we first find the wires that must always be precise. Algorithm 1 presents the algorithm to find these precise wires. In this algorithm, for each precise output or critical wire, we mark precise the wires connected to the inputs of the gate driving that output or wire. We repeat this step for the newly marked precise wires until we reach an input or an approximate wire. After marking all the wires that must be precise, any remaining wires will be marked approximate. The last step is to mark any gate that drives an approximate wire as a safe candidate to be approximated.

This static analysis only provides safety guarantees and does not deal with quality. In the following sections, we will describe the design abstractions that enable designers to express quality requirements. The quality requirements will guide the synthesis process to only select a subset of the safe-to-approximate gates for approximation.

6. Constraining Approximation

The ability of the designer to express quality of result requirements, thereby constraining approximation, is imperative at both design and reuse time. Where and how a module will be reused in a higher level design is not necessarily known during its design. Furthermore, accuracy requirements can be polymorphous depending on the exact implementation context, yet reusability is a critical feature of many modules. Finally, the designer should be able to control approximate synthesis and optimization under all such circumstances without being burdened by the details of this process. Towards this end, we describe an abstraction and the corresponding semantics by which acceptable error bounds may be applied to any approximate element within the system.

Our language attributes not only provide abstraction for *safe* approximate hardware design but also the necessary semantics for *high-level quality control*. We introduce a modified **(*A*)** attribute in the form of **(*A: f() < ε*)**, in which ε is the upper bound for the error level described by a user-provided function **f()**. However, the synthesis process may not be able to apply approximation properly and prove meaningful guarantees with no information about the input profile. Full knowledge about the probability of each input, along with an approximation plan, would allow the synthesis process to perform approximate optimizations on the system while guaranteeing with complete certainty that the constraints will be satisfied. Such knowledge is not attainable in practice due to the unpredictable nature of input data sets. Therefore, we allow the designer to provide representative inputs for profiling, along with a global confidence metric which should be satisfied by the synthesis tool. These representative inputs enable the synthesis tool to provide statistical quality guarantees. As the representative input data become more comprehensive, the synthesis tool can apply approximate optimizations more aggressively with a higher degree of confidence. These high-level abstractions for expressing quality requirements provide the following benefits. (1) Hardware designers do not explicitly decide where or how to apply approximate synthesis optimizations. (2) Theses modules provide better reusability since the same module implementation can be reused across many system that may require different level of precision. The synthesis and layout tools have jurisdiction over how different approximation techniques are applied while requiring only a single implementation from the designer. Quality constraints can be easily adjusted or introduced to meet the accuracy needs of various systems, providing a powerful reuse mechanism for hardware modules, which is particularly compelling for IP designers.

Algorithm 1 Backward slicing to find precise wires.

```
Inputs:   $K$: Circuit
          $\Theta$: Set of precise outputs
          $\Psi$: Set of critical wire overrides
          $\Upsilon$: Set of approximate wires overrides
Output:   $\Re$: Set of precise wires

  Initialize $\Re \leftarrow \emptyset$
  Initialize $Q \leftarrow \emptyset$
  for each $w_i \in (\Theta \cup \Psi)$ do
    enqueue($Q$, $w_i$)
  end for
  while ($Q \neq \emptyset$) do
    $w_i \leftarrow$ dequeue($Q$)
    $\Phi \leftarrow$ In $K$, find input wires of the gate that drives $w_i$
    for each $w_j \in \Phi$ do
      if ($w_j \notin \Upsilon$ and $w_j \notin \Re$) then
        $\Re \leftarrow \Re \cup w_j$
        enqueue($Q$, $w_j$)
      end if
    end for
  end while
```

7. Approximate Synthesis Process

In our framework, we envision a synthesis tool that 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, our safety analysis—part of which is presented in Algorithm 1—will identify the safe-to-approximate subset of the gates with regards to the designer annotations. The safety criteria is that this subset of gates does not contribute in any ways to precise outputs. It is in the next step that the synthesis tool incorporates the error bounds and the quality requirements. Considering the error constraints, the synthesis tool may choose a subset of the safe-to-approximate gates to be approximated, but it is illegal to apply such optimizations to the precise gates. The synthesis tool has the liberty to apply gate substitution, gate elimination, logic restructuring, voltage over-scaling, or any other optimizations as it deems prudent. The computational problem of selecting a subset of the safe-to-approximate gates and the corresponding approximation technique can be formulated as a constrained optimization problem. The objective is to minimize a combination of error, delay, and energy. In future work, we will provide a generalized framework for this constrained optimization problem. Nevertheless, our design abstractions enable the designer to guide the synthesis process with high-level annotation while delegating the implementation and application of approximation to an automated procedure.

8. Related Work

A growing body of research shows the applicability and significant benefits of approximation [5, 6, 8, 9, 19]. However, prior research has not explored extending hardware description languages for systematic, and reusable approximate hardware design. Our work is at the intersection of approximate language design and approximate hardware synthesis techniques.

Approximate programming languages. EnerJ [21] provides a set of type qualifier to enable programmers to explicitly declare *all* the approximate variables in the program. As Figure 7, if we had extended EnerJ's model to Verilog, the designer would have been required to manually declared all the wires that are approximate. With our abstractions as Figure 8 illustrates, the designer usually marks the approximate outputs and the safety analysis automatically identifies which wires and modules are safe-to-approximate. Furthermore, EnerJ does not provide any semantic for specifying accuracy requirements or acceptable error bounds. Rely [2] requires the programmer to explicitly and manually mark both variables and operations as approximate. However, it provides semantics for verifying whether these annotations will satisfy programer specified accuracy requirements. Or work aims to automate the process of selecting where to apply approximation and yet provide statistical guarantees.

Approximate circuit design and synthesis. In [10, 14, 22, 25] alternative and less accurate implementation specific hardware blocks such as adders and multipliers are proposed. Miao et al. [18] provides many Pareto-optimal designs alternatives for adder and allows the designer to choose a variant based on her energy, performance, and accuracy requirement.

While these ad-hoc approaches show significant promise for approximation in the circuit level, they do not provide any methodical and general approach for approximate hardware design. However, Salsa [24], [16], [15] [17] propose a systematic approach to automatically apply approximate synthesis techniques, generally gate pruning and timing speculation. Salsa and [17] incorporate user-defines error constraints in their synthesis techniques. While all these synthesis techniques provide significant improvements, they do not focus on providing hardware description semantics for methodical approximate hardware design and reuse. In fact, our framework can benefit and leverage from all these techniques. Overall, through extensions to Verilog, we propose a systematic design flow and its required abstractions to enable approximate hardware design and reuse in larger scale hardware systems. Our framework enables approximate synthesis techniques to systematically be a part of the SoC design cycle.

9. Conclusion

While approximate circuits have been shown to provide significant energy and performance benefits, there is a clear need for design flows and abstractions that enable larger scale approximate hardware design and reuse. In this paper, we proposed design abstractions that enable system designers to implicitly declare which parts of the design can be safely approximated. All these abstractions are presented as concrete extensions to the mainstream Verilog hardware descriptions language. Further, our framework allows designers to override the approximation plan of an already designed approximate module at reuse time and to explicitly control the quality tradeoffs. We provide a static analysis that infers where to safely apply approximation without violating safety guarantees. Through this automatic analysis, we strike a balance between designer involvement and automation of approximation in hardware systems design. The flexible and automatic nature of our framework provides a less arduous environment compared to a mere extension of existing approximate programming models for hardware design. We believe that our work forms a foundation for widespread and methodical approximate hardware design and reuse.

References

[1] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs," in *ISCA*, 2013.

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

[3] L. N. Chakrapani, P. Korkmaz, B. E. Akgul, and K. V. Palem, "Probabilistic system-on-a-chip architectures," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2007.

[4] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *DAC*, 2013.

[5] H. Cho, L. Leem, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2012.

[6] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," *in ISCA*, 2010.

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

[8] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ACM SIGARCH Computer Architecture News*, 2012.

[9] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[10] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: imprecise adders for low-power approximate computing," in *ISLPED*, 2011.

[11] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *ETS*, 2013.

[12] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *ISLPED*, 1999.

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

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

[15] A. Lingamneni, K. K. Muntimadugu, C. Enz, R. M. Karp, K. V. Palem, and C. Piguet, "Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling," in *Computing Frontiers*, 2012.

[16] Y. Liu, R. Ye, F. Yuan, R. Kumar, and Q. Xu, "On logic synthesis for timing speculation," in *ICCAD*, 2012.

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

[18] J. Miao, K. He, A. Gerstlauer, and M. Orshansky, "Modeling and synthesis of quality-energy optimal approximate adders," in *ICCAD*, 2012.

[19] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE*, 2010.

[20] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: self-tuning approximation for graphics engines," in *MICRO*, 2013.

[21] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," *ACM SIGPLAN Notices*, 2011.

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

[23] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *DATE*, 2013.

[24] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: systematic logic synthesis of approximate circuits," in *DAC*, 2012.

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