SKETCHILOG: Sketching Combinational Circuits

Andrew Becker, David Novo and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
Email: {first.last}@epfl.ch

Abstract—Despite the progress of higher-level languages and tools, Register Transfer Level (RTL) is still by far the dominant input format for high performance digital designs. Experienced designers can directly express their microarchitectural intuitions in RTL. Yet, RTL is terribly verbose, burdened with trivial details, and thus error prone. In this paper, we augment a modern RTL language (Chisel) with new semantic elements to express an imprecise specification: a sketch. We show how, in combination with a naïve, unoptimized, but functionally correct reference, a designer can utilize the language and supporting infrastructure to focus on the key design intuition and omit some of the necessary details. The resulting design is exactly or almost exactly as good as the one the designer could have achieved by spending the time to manually complete the sketch. We show that, even limiting ourselves to combinational circuits, realistic instances of meaningful design problems are solved quickly, saving considerable design and debugging effort.

I. INTRODUCTION

For more than twenty years, designing digital circuits at the Register Transfer Level (RTL) has been one of the key bottlenecks to productivity, and researchers have strived to raise the design abstraction level [2]. Progress in the area of High-Level Synthesis (HLS) has been less steady than originally anticipated, with various generations of tools reaching the market [7] and perhaps only in the last few years achieving some concrete commercial successes. Yet, RTL still offers a designer the most control, and skilled designers’ analytical intuitions about structural circuit optimizations and tradeoffs are usually superior to those achieved by high-level compilers.

We have extended a modern RTL, Chisel [1], and found inspiration from the software world [11], to take a new approach: instead of abstracting away fundamental features of the architecture—as in High Level Synthesis—abstract only those details which a designer is uncertain of. In other words, designers construct their circuits in RTL as usual but leave holes, or explicit indeterminacies, in their designs. SKETCHILOG, our toolchain, reads a regular RTL specification of a desired functionality (typically a trivial unoptimized reference) and an incomplete optimized implementation of the same functionality (a sketch). SKETCHILOG determines whether the holes can be filled so that the functionality of the sketch matches that of the specification under all inputs. If such a substitution exists, SKETCHILOG outputs fully functional Verilog of the completed and fully verified sketch. This effectively relieves designers from coding the most annoying details of an architecture and entirely avoids a major source of maddening and time-consuming bugs. Although we only make the first steps in this direction (for instance, we currently are limited to combinational circuits), we believe that this is a viable path to raise RTL design productivity to new levels. Interested readers can download SKETCHILOG for themselves at http://sketchilog.epfl.ch.

We describe our motivation and formalize the problem to solve in Section II, detail our contribution (including the added RTL constructs and their handling) in Section III along with the corresponding limitations, then finish by reporting the results of running a few examples through our tool.

II. MOTIVATING EXAMPLE AND PROBLEM DEFINITION

Any digital designer knows how to make an efficient two’s-complement ADD/SUB unit. However, suppose for the sake of example that a designer does not remember how exactly to build the unit, but remembers that some voodoo with an adder’s operands can implement a subtracter. Our designer might describe Fig. 1a as a reference and sketch Fig. 1b from fuzzy intuition—an adder signals are some logic function of the adder’s operands can implement a subtracter. However, suppose for the sake of example that a designer does not remember how exactly to build the unit, but remembers that some voodoo with an adder’s operands can implement a subtracter. Our designer might describe Fig. 1a as a reference and sketch Fig. 1b from fuzzy intuition—an adder signals are some logic function of the corresponding ADD/SUB module signals (with carry-in dependent on d).

The core of this sketch can be expressed in SKETCHILOG as shown in Fig. 1c: a simple ripple-carry adder whose inputs are some undetermined function (a black box) of a and b. SKETCHILOG solves the sketch and finds that the values shown in Fig. 1d for the inferred lookup tables force the circuit to match to the reference design. When a solution exists, correct hole values are always found and the resulting design must be functionally correct. If holes are not abused to give excessive architectural freedom, a given solution will usually be very nearly as small and fast as if the designer had no uncertainty at all.

SKETCHILOG translates both the sketch and the specification to flat Boolean functions $S$ and $R$, respectively. Both functions take the same $k$-bit input vector $x$, but the sketched function also takes an additional $m$-bits control parameter $c$, representing the $m$ holes in the sketch. The problem reduces to building a miter from the functions and solving a Quantified Boolean Formula Satisfiability (QBF-SAT) problem:

$$\exists c \in \{0,1\}^m, \forall x \in \{0,1\}^k : R(x) \iff S(x,c).$$
Our goal with SKETCHILOG is to provide useful and intuitive RTL language constructs which help designers focus on architectural intuition instead of nitty-gritty details, and yet can be encoded as a vector of unknown Boolean variables $c$.

### III. IMPLEMENTING SKETCHILOG

We chose Chisel [1] as the base language in which to implement our sketching constructs. Its use of Scala [8] lends it easy extension and customization, and its scripting-like functionality makes sketching more intuitive. Chisel generates regular Verilog code and (solved) sketched designs can be used in standard EDA design flows. Everything we do with Chisel could be done less elegantly inside a VHDL or Verilog parser, though this would likely require a less intuitive syntax.

#### A. The Rules of the Code

On top of the standard Chisel semantics, we provide four intuitive constructs to support uncertainty in designs. Each construct can only be used to provide a value to Chisel data types and never any regular Scala Int type: the left-hand side of each expression below must be a Chisel data type.

\[
\text{x := \texttt{??}(n); // (1).}
\]

This first construct, an uncertain constant (or raw hole) generator, serves as a substitute for a concrete signal value, and represents an $n$-bit constant signal whose value is undefined. This construct is the simplest both to understand and implement. SKETCHILOG infers an additional $n$-bits in the constructed QBF-SAT model’s control vector.

\[
\text{x := \texttt{either choose signal1 or signal2 or signal3; // (2).}
\]

This second construct, a selection operator, allows a designer to express an uncertain choice of signals in a design. SKETCHILOG automatically creates raw holes which represent constant values for the select inputs of multiplexers which choose one of the specified signals.

\[
\text{x := \texttt{my_array}(2 * \texttt{??}(n) - 1); // (3).}
\]

This third construct, an undetermined index operator, allows a designer to express a partially-constrained index or bit in any indexed sequence data structure or Chisel signal type, respectively. It is basically an application of the second construct, selecting among the signals identified through every feasible index into \texttt{my_array} (e.g., 1, 3, 5, etc.). Any out-of-bounds index is silently dropped from consideration—helping designers not to worry about edge cases. A feasible set associated with each hole is computed by static analysis of the index expression similar to classic bitwidth analysis [6].

\[
\text{x = \texttt{BB}(depends, n); // (4).}
\]

This powerful construct, an arbitrary logic function generator, constrains a signal $x$’s value very loosely: only its dependencies and width are provided. Determination of exactly what logic function to implement is left to SKETCHILOG. This creates $2^{\text{depends.width}}$ $n$-bit existentially-quantified inputs in the QBF-SAT model. It must be used cautiously, however, as the number of hole bits grows exponentially with \text{depends.width}. Its misapplication with unreasonably large widths or number of dependencies dramatically affects scalability.

#### B. Hardware Sketching vs. Software Sketching

Solar-Lezama et al. pioneered the sketching concept in a software context with a language called SKETCH [11]. The same group toyed with the idea of sketching hardware [10], but the work provided no clear rationale and remained purely conjecture. We build our hardware flow upon the CEGIS QBF-SAT solver originally designed for software sketching. All other parts of our system are carefully tailored to the hardware design process and are either built from scratch, borrowed from other work with minor modifications (ABC [12]), embraced and extended from other work (Chisel), or heavily modified from their original form (Odin II [4]).

The main difference (other than the domain of application) between software sketching as presented by Solar-Lezama et al. [11] and our SKETCHILOG hardware design framework lays in the generation of the QBF-SAT problem. Firstly, the software SKETCH framework needs to build the Boolean circuit models used to solve the QBF-SAT problem from an imperative C-like language by a sort of high-level synthesis. This inherits the difficulties of HLS: the generated models are often more complex than required, leading to increased solution times. In contrast, in our framework the Boolean circuit model is the actual sketch, which is directly constructed by the designer. As part of the model itself, our hole bit-widths
A. Prefix Adders

The problem of adding two binary numbers as quickly as possible reduces to the problem of computing the carry signals $C_i$ (represented in the form of a generate and propagate signal pair) for all bit positions $i$ [3], [9]. The computation of the carry signals can be posed in the form of a series of associative but noncommutative operations:

$$C_i = GP_1 \ast GP_{i-1} \ast \ldots \ast GP_1 \ast C_0$$

The ripple-carry implementation is an easy reference for SKETCHLOG but it is faster to compute all carry signals independently: they can be computed fully in parallel as binary trees of $\ast$ operators, resulting in a Kogge-Stone Adder represented in Fig. 2. Even such a simple structure requires careful attention to detail in the code: instantiating a complete binary tree is not possible for many $i$ and if $\text{width}$ is not a power of two, the largest tree is itself incomplete. Fig. 2b shows the actual code needed in SKETCHLOG to generate the correct hardware, using two of our SKETCHLOG-specific Chisel syntax extensions. Note the design is not obfuscated by clumsy boundary tests: the designer simply says “connect regularly if you can, or else find a suitable constant”.

B. Sketching to Enable Design Re-Use

Suppose a designer would like to use a library component, like a Synopsys DesignWare inverse square-root unit. Unfortunately, that IP component requires the input to be in the range $[\frac{1}{4}, 1)$, a restriction not adapted to the domain required by the designer. The designer would rather create an adaptation interface than reimplement the unit from scratch. Elementary algebra suggests a variable shift at the input and output of the unit. Intuitively, there must be a correlation between the magnitude of the input and the scaling factors. Unfortunately, finding the exact relations is tricky and error prone. Instead, the designer can construct a general architecture with just their intuition (see Fig. 3) and these lines:

```chisel
val pre_shift_amt = BB(zero_count, 4);
val post_shift_amt = BB(zero_count, 4);
```

These lines specify that the shift amounts depend somehow on the signal `zero_count` and are 4 bits wide. When run with an extra sketched adjustment for the border cases against a trivial infinite-precision look-up table reference, SKETCHLOG finds the correct implementation—and automatically infers essential but trivial details, like that the input shift amount must be even to re-scale the output without loss of precision.

C. Strength Reduction of a Constant Divider

Our final example shows the case, common in arithmetic circuits, of finite precision operations implemented by simpler operators with so-called magic numbers. One well-known example is the inverse square-root approximation found in, among other places, the Quake III video game source code [5].

In our example, a designer wants to devise an efficient implementation of a fixed-point constant division unit with a near-power-of-two divisor (e.g., 65,535). A simple right shift
is a DesignWare divider with a constant divisor. We run hybrid Raw_InvSqrt is the IP used inside. Brent-Kung). 

is a naive DesignWare divider with a constant operand. Such values do exist in this case, and words, maybe some magic fudge factors make a simple affine approximation exact, but is not exact. The designer’s pass, total number of hole bits, and critical path delay (AIG // The exact control logic is left to our tool. // is that the shifters and leading zero detector will help to scale // the input into the component’s domain and to correctly re-scale // the output. The exact control logic is left to our tool. // Fig. 3: Complex adaptation of an IP component. The intuition // finds a correct design significantly smaller than // KETCHI black-box look-up tables // $x^{(-0.5)}\gg \equiv \begin{array}{c} \text{LZC} \\
\text{Sketched black-box look-up tables} \\
\text{Synopsys}
\end{array}$

### TABLE I: Experimental results detailing instance bitwidth, CEGIS solver runtime both with and without an optimization pass, total number of hole bits, and critical path delay (AIG depth) and area (AIG size) for the resulting completed design.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Width</th>
<th>Hole Bits</th>
<th>Unopt. Time (s)</th>
<th>Opt. Time (s)</th>
<th>AIG Depth</th>
<th>AIG Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>KoggeStone</td>
<td>16</td>
<td>427</td>
<td>3.732</td>
<td>3.799</td>
<td>12</td>
<td>229</td>
</tr>
<tr>
<td>Hybrid_max1</td>
<td>16</td>
<td>509</td>
<td>1.681</td>
<td>1.450</td>
<td>12</td>
<td>217</td>
</tr>
<tr>
<td>Hybrid_max2</td>
<td>16</td>
<td>368</td>
<td>0.697</td>
<td>0.901</td>
<td>13</td>
<td>196</td>
</tr>
<tr>
<td>BrentKung</td>
<td>16</td>
<td>334</td>
<td>0.842</td>
<td>1.048</td>
<td>16</td>
<td>160</td>
</tr>
<tr>
<td>RippleCarry</td>
<td>16</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>32</td>
<td>131</td>
</tr>
<tr>
<td>KoggeStone</td>
<td>23</td>
<td>713</td>
<td>22.414</td>
<td>24.383</td>
<td>12</td>
<td>379</td>
</tr>
<tr>
<td>Hybrid_max1</td>
<td>23</td>
<td>902</td>
<td>6.544</td>
<td>8.374</td>
<td>13</td>
<td>345</td>
</tr>
<tr>
<td>Hybrid_max2</td>
<td>23</td>
<td>570</td>
<td>3.434</td>
<td>3.112</td>
<td>15</td>
<td>296</td>
</tr>
<tr>
<td>BrentKung</td>
<td>23</td>
<td>522</td>
<td>2.884</td>
<td>3.339</td>
<td>18</td>
<td>237</td>
</tr>
<tr>
<td>RippleCarry</td>
<td>23</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>46</td>
<td>187</td>
</tr>
<tr>
<td>KoggeStone</td>
<td>32</td>
<td>992</td>
<td>93.620</td>
<td>85.064</td>
<td>14</td>
<td>545</td>
</tr>
<tr>
<td>Hybrid_max1</td>
<td>32</td>
<td>1754</td>
<td>27.984</td>
<td>40.008</td>
<td>14</td>
<td>529</td>
</tr>
<tr>
<td>Hybrid_max2</td>
<td>32</td>
<td>1166</td>
<td>19.772</td>
<td>19.871</td>
<td>15</td>
<td>488</td>
</tr>
<tr>
<td>BrentKung</td>
<td>32</td>
<td>720</td>
<td>10.558</td>
<td>12.256</td>
<td>18</td>
<td>333</td>
</tr>
<tr>
<td>RippleCarry</td>
<td>32</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>64</td>
<td>259</td>
</tr>
<tr>
<td>Expanded_InvSqrt</td>
<td>8/13</td>
<td>96</td>
<td>1.131</td>
<td>0.928</td>
<td>370</td>
<td>4093</td>
</tr>
<tr>
<td>Raw_InvSqrt</td>
<td>8/12</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>371</td>
<td>4002</td>
</tr>
<tr>
<td>ConstDivision</td>
<td>32</td>
<td>40</td>
<td>26.867</td>
<td>7.960</td>
<td>84</td>
<td>1152</td>
</tr>
<tr>
<td>DW_Const_Divider</td>
<td>32</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>255</td>
<td>2007</td>
</tr>
<tr>
<td>ConstDivision</td>
<td>64</td>
<td>73</td>
<td>3373.790</td>
<td>333.083</td>
<td>164</td>
<td>4400</td>
</tr>
<tr>
<td>DW_Const_Divider</td>
<td>64</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
<td>529</td>
<td>6291</td>
</tr>
</tbody>
</table>

Contrary to common expectations a decade or two ago, it appears RTL design is here to stay. RTL may be complemented by higher level abstractions, but likely will not supplanted. We have made first attempts at simplifying RTL design by allowing designers some indeterminacy in designs. Despite the simplicity of our examples, the benefits of sketching circuits are clear: SKETCHLOG removes the burden of those small details which often cause errors, and are most annoying to get exactly right. Since a reference version is available (of any quality—hence naturally simple to write and debug), SKETCHLOG not only takes the dirty work from the designer but also guarantees, that the resulting design is correct. If there is no way to fill in the holes and obtain a working circuit, SKETCHLOG immediately reports so. Although in some domains, like digital arithmetic, the tool is already able to produce practical results, it remains ripe for further exploration, extension, and improvement.

**VI. CONCLUSIONS**

### REFERENCES


