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

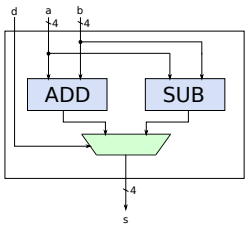
Any digital designer knows how to make an efficient two's-complement ADD/SUB unit. However, suppose for the sake of example that a designer does not remember *how exactly* to build the unit, but remembers that some voodoo with an adder's operands can implement a subtracter. Our designer might describe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

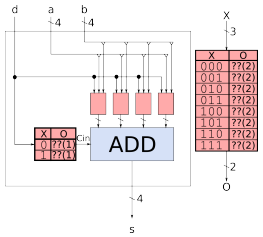
SKETCHILOG translates both the sketch and the specification to flat Boolean functions $\mathcal{S}$ and $\mathcal{R}$, respectively. Both functions take the same k-bit input vector x, but the sketched function also takes an additional m-bits *control* parameter c, representing the m holes in the sketch.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

$$\exists c \in \{0,1\}^m, \forall x \in \{0,1\}^k : \mathcal{R}(x) \Leftrightarrow \mathcal{S}(x,c).$$

978-3-9815370-2-4/DATE14/©2014 EDAA



(a) A simple reference ADD/SUB unit.

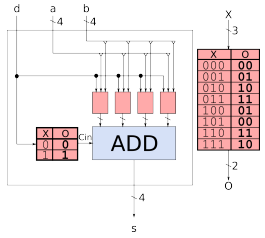


(b) A naïve sketch of an optimized ADD/SUB unit.

```
carries(0) := BB(io.d, 1);
for(i <- 0 until 4){
  val fadd = new full_adder;
  val unkn = io.d##io.a(i)##io.b(i);
  val bb = BB(unkn, 2);

  fadd.io.a := bb(0);
  fadd.io.b := bb(1);
  fadd.io.cin := carries(i);
  sum_sigs(i) := fadd.io.s;
  carries(i+1) := fadd.io.cout;
}
```

(c) SKETCHILOG code for (b).



(d) The solved sketch from (b).

Fig. 1: A naïve sketch of an ADD/SUB unit. The solution (d) immediately reminds an inexperimented designer that the adder should be fed with signal a unmodified and with carry-in and b signals conditionally inverted upon the value of d.

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.

```
x := ??(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 undetermined. 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.

```
x := 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.

```
x := my_array(2 * ??(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 `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].

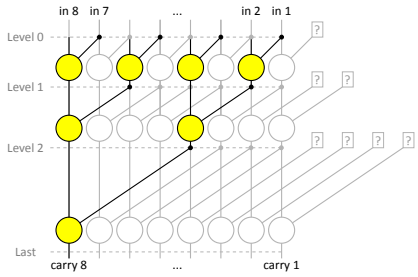
```
x = BB(depends, n); //(4).
```

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 $2^{depends.width}$ 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 $depends.width$. 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) The adder structure.

```
for (k <- 0 until levels) {
  for (i <- 0 to width) {
    val GP = new carryOp;
    GP.io.left := c(k)(i);
    GP.io.right:= either choose c(k)(i - pow2(k)) or ??(2);
    c(k+1)(i) := GP.io.out; } }
```

(b) The simple code to build the structure using SKETCHILOG.

Fig. 2: A Kogge-Stone Adder. With SKETCHILOG, the designer focuses on the intuition of creating a binary tree of `carryOp` cells for each output and essentially ignores trivial but annoying boundary conditions.

are always known precisely while, in SKETCH, assumptions are made to constrain the size of potential hole assignments. Secondly, software SKETCH allows the end user to reference a raw hole nearly anywhere in the code. Instead, we provide the set of constructs detailed in Section III-A to encapsulate holes and thus prevent the user from misusing them in ways that are possible in SKETCH. For example, software SKETCH code can contain a hole in place of a loop bound, resulting in potentially enormous models as the loop is unrolled. Such uncertainties in circuit structure cannot happen with SKETCHILOG.

C. The Limitations of SKETCHILOG

SKETCHILOG is necessarily limited in scope, however, in two key regards. First, only combinational circuits are currently supported. Second, the difficulty of the QBF-SAT problem limits the feasible problem size, and solver performance is highly instance-specific. Minor changes in a designer's sketch might have a dramatic effect on solution time. We believe these limitations do not fatally detract from the value of SKETCHILOG. While a limitation to combinational circuits seems severe, it still covers many use cases, and simple pipelined circuits are functionally isomorphic to combinational models. This makes SKETCHILOG applicable to many arithmetic circuit generators, which are often some of the most tricky circuits to get right.

IV. EXPERIMENTS

This section demonstrates our tool through simple but conceptually representative use-cases. For clarity, we have selected simple architectures which are described in any basic course in computer arithmetic, even if they are readily available in synthesis libraries—the purpose is to illustrate the simplicity of the approach and how SKETCHILOG could even benefit library writers themselves.

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:

$$C_i = GP_i \star GP_{i-1} \star \ldots \star GP_1 \star C_0 \tag{1}$$

The ripple-carry implementation is an easy reference for SKETCHILOG but it is faster to compute all carry signals independently: they can be computed fully in parallel as binary trees of $\star$ 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 `width` is not a power of two, the largest tree is itself incomplete. Fig. 2b shows the actual code needed in SKETCHILOG to generate the correct hardware, using two of our SKETCHILOG-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 $\left[\frac{1}{4}, 1\right)$, 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:

```
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, SKETCHILOG 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

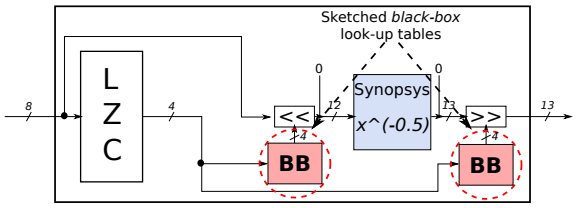


Fig. 3: Complex adaptation of an IP component. The intuition is that the shifters and leading zero detector will help to scale the input into the component's domain and to correctly re-scale the output. The exact control logic is left to our tool.

Experiment	Width	Hole Bits	Unopt. Time (s)	Opt. Time (s)	AIG Depth	AIG Nodes
KoggeStone	16	427	3.732	3.799	12	229
Hybrid_max1	16	509	1.681	1.450	12	217
Hybrid_max2	16	368	0.697	0.901	13	196
BrentKung	16	334	0.842	1.048	16	160
RippleCarry	16	0	n/a	n/a	32	131
KoggeStone	23	713	22.414	24.383	12	379
Hybrid_max1	23	902	6.544	8.374	13	345
Hybrid_max2	23	570	3.434	3.112	15	296
BrentKung	23	522	2.884	3.339	18	237
RippleCarry	23	0	n/a	n/a	46	187
KoggeStone	32	992	93.620	85.064	14	545
Hybrid_max1	32	1754	27.984	40.008	14	529
Hybrid_max2	32	1166	19.772	19.871	15	488
BrentKung	32	720	10.558	12.256	20	333
RippleCarry	32	0	n/a	n/a	64	259
Expanded_InvSqrt	8/13	96	1.131	0.928	370	4093
Raw_InvSqrt	8/12	0	n/a	n/a	371	4002
ConstDivision	32	40	26.867	7.960	84	1152
DW_Const_Divider	32	0	n/a	n/a	255	2007
ConstDivision	64	73	3373.790	333.083	164	4400
DW_Const_Divider	64	0	n/a	n/a	529	6291

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.

is a passable approximation, but is not exact. The designer's intuition is again simple: perhaps there are some integers x, y, and z, such that $(i \cdot x + y) \gg z \equiv \frac{i}{65535}$. In other words, maybe some magic fudge factors make a simple affine approximation exact. Such values do exist in this case, and SKETCHILOG finds a correct design significantly smaller than a naïve DesignWare divider with a constant operand.

V. EXPERIMENTAL RESULTS

We sketched the circuits described in Section IV. We also sketched a few other adders (a Brent-Kung and *hybrid* prefix adders, which lie between the Kogge-Stone and the Brent-Kung). `RippleCarry` is a non-sketched ripple-carry adder. `Expanded_InvSqrt` is the Section IV-B example; `Raw_InvSqrt` is the IP used inside. `DW_Const_Divider` is a DesignWare divider with a constant divisor. We run SKETCHILOG both with and without a sketch pre-optimization pass, where the sketch circuit model is first run through logic synthesis in ABC. This pass is usually ineffective or counterproductive, but opens the door for improved heuristics and optimization strategies which only enhance scalability. The resultant circuit's AIG depth and size are shown after ABC simplifies it with structural hashing and SAT sweeping.

Our data show that for most experiments, the solver runtime is low enough to enable SKETCHILOG's use as a real design aid. The adder experiments in particular show that our framework is scalable enough to be used as part of a standard design flow, at least for some important circuits. The inverse square-root example demonstrates that the described sketching constructs require very little overhead in the final solved circuit.

VI. CONCLUSIONS

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: SKETCHILOG 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), SKETCHILOG 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, SKETCHILOG 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.

REFERENCES

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Proceedings of the 49th Design Automation Conference*, San Francisco, Calif., Jun. 2012, pp. 1212–1221.

[2] R. Camposano, "From behavior to structure: High-Level Synthesis," *IEEE Design and Test of Computers*, vol. 7, no. 5, pp. 8–19, Oct. 1990.

[3] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, Calif.: Morgan Kaufmann, 2004.

[4] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II - An Open-source Verilog HDL Synthesis tool for CAD Research," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 149–156.

[5] C. Lomont, "Fast inverse square root," 2003. [Online]. Available: http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf

[6] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, "Bitwidth cognizant architecture synthesis of custom hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-20, no. 11, pp. 1355–71, Nov. 2001.

[7] G. Martin and G. Smith, "High-Level Synthesis: Past, present, and future," *IEEE Design and Test of Computers*, vol. 26, no. 4, pp. 18–24, Jul.–Aug. 2009.

[8] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*, 2nd ed. Walnut Creek, Calif.: Artima, 2010.

[9] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Design*, 2nd ed. New York: Oxford University Press, 2010.

[10] A. Raabe and R. Bodík, "Synthesizing hardware from sketches," in *DAC*. ACM, 2009, pp. 623–624.

[11] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 404–415.

[12] B. L. Synthesis and V. Group. (2005, December) ABC: A system for sequential synthesis and verification. [Online]. Available: http://www-cad.eecs.berkeley.edu/~alanmi/abc