Hardware Verification using Monadic Second-Order Logic

David A. Basin[1] and Nils Klarlund*[2]

[1] Max-Planck-Institut für Informatik,
Im Stadtwald, D-66123 Saarbrücken, Germany
[2] BRICS, Department of Computer Science University of Aarhus,
Ny Munkegade, DK-8000 Aarhus C, Denmark

Abstract. We show how the second-order monadic theory of strings can be used to specify hardware components and their behavior. This logic admits a decision procedure and counter-model generator based on canonical automata for formulas. We have used a system implementing these concepts to verify, or find errors in, a number of circuits proposed in the literature. The techniques we use make it easier to identify regularity in circuits, including those that are parameterized or have parameterized behavioral specifications. Our proofs are semantic and do not require lemmas or induction as would be needed when employing a conventional theory of strings as a recursive data type.

1 Introduction

Properties of functions over finite domains may be established by state space enumeration. Consequently, combinational logic circuits may be viewed as Boolean functions, and the equivalence of circuits may be shown by exhaustive testing—which in turn may be optimized using Boolean decision diagrams (BDDs). But when the domain is infinite, direct enumeration is impossible. For example, although we can apply state space enumeration to prove that a 1-bit adder meets its specification, we cannot do the same for a parameterized n-bit adder.

There are similar problems in verifying temporal properties of a circuit that should hold over all instants of time. Typically such circuits are verified interactively, or semi-automatically using mathematical induction [1, 2, 6, 9, 12, 13, 15]; that is, a time-dependent property $P(n)$ is shown to hold for all instants n by induction over n.

There is an alternative approach to such problems, however: find and analyze a finite characterization of the infinite state space. For example, an n-bit adder may be constructed by chaining together (ripple-carry style) n full adders. The resulting circuit is regular in both the informal and formal sense. If we view the input/output relation of an n-bit adder as words of length n over an alphabet describing inputs and outputs at each position, then the language that is the union of all these languages for $n = 1, 2, \ldots$ can be recognized by a finite automaton and is thus a regular language.

Recently, other groups of researchers, e.g., [7, 16], have presented methods that exploit this kind of regularity. In [16], parameterized circuits are not described as such. Instead, the automaton model is inferred by observing the behavior of circuits for $n = 1, 2 \ldots$ until some technical conditions indicate a fixed point. Alternatively [7, 8] identify classes of parameterized circuits that can be described by recursive BDD tree structures which, for certain classes, correspond to finite automata. While these techniques are novel and can be very effective, they have their drawbacks. In particular, circuits must be encoded directly as automata as opposed to more declaratively by functions or relations that mirror their structure. Similarly, behavioral specifications are encoded instead of being expressed in a more conventional specification language that admits logical connectives, functions, and the like. As a result, circuits and their specifications may be considerably more complex than what is possible in richer languages.

We report here on a logical characterization of regular circuits that generalizes the class of circuits describable by previous methods. We believe it gives a better understanding of when regularity can be exploited to prove automatically the correctness of circuits. Moreover, it allows specification closer to what is possible within richer specification languages.

Our work is based on a second-order monadic theory of strings, M2L(Str), and a system, Mona, that implements a decision procedure and counter-model generator for statements in this logic [10]. M2L(Str) is a very concise, but so far in practice unexplored, way of characterizing regularity. Operators in the logic formalize positions and sets of positions within strings and relate them by means of quantification and logical connectives. These connectives suffice to directly encode Boolean logic and all non-parameterized circuits. Quantification over sets of positions can encode parameterized circuits with regular connectivity like ripple-carry adders. The logic still admits a decision procedure and counter-model generator based

* Basic Research in Computer Science, Centre of the Danish National Research Foundation.

on constructing a canonical automaton for each subformula. As a result, tautology checking is decidable, although in non-elementary time (the lower bound is a stack of exponentials of height proportional to the size of the formula). Thus in principle, we can automatically verify the correctness of circuits relative to specifications that are also expressible in M2L(Str).

Perhaps surprisingly, we show here that verification is possible in practice too. In this report we document the automatic verification of a parameterized Arithmetic Logic Unit (ALU) and a timed flipflop in Mona. Moreover, we show how counter-model generation within M2L(Str) provides a useful tool for debugging and testing incorrect circuits and specifications. Hence, Mona provides not just a tool for circuit validation, but also for circuit development, debugging, and prototyping.

While the applications we present are new, it has been known for over 30 years that monadic second-order theories define regular languages and hence are decidable. The staggering theoretical complexity seemed to preclude practical experiments. And the limited expressibility of M2L(Str) means that not all parameterized circuits or their behavior can be specified. For example, one cannot encode a network whose connectivity is non-regular (e.g., parameterized grids). Furthermore, specifications are limited in the amount of arithmetic that may be formalized; the slightly stronger logic WS1S allows one to express Presburger arithmetic, but little more [17]. Even when arithmetic operations are expressible, a specification in M2L(Str) may be less direct than one in a more expressive logic. As we shall see, arithmetic must be encoded as operations over strings.

Despite these obstacles, we were still able to solve interesting problems. For example, the ALU that we verified (see §3) required only half a CPU minute (on a SPARC station 20) and a couple of megabytes of memory; the largest intermediate automaton generated by Mona in this example contained 154 states.

We proceed as follows. In §2 we give an introduction to M2L(Str) and Mona. In §3 we consider verification of parameterized hardware and verify a parameterized adder and ALU. In §4 we consider timed hardware and use Mona to analyze a D-type flipflop. Such memory devices involve feedback, and formal analysis of their properties is surprisingly difficult. We begin by considering a specification given by [6] that was verified by hand. We use Mona to generate a minimal counterexample to this specification and, through further experiments, we develop a correct specification. This demonstrates how M2L(Str) can be used not just as the basis of a yes/no decision procedure, but also as a tool supporting the development and analysis of complex circuits. In the final section we draw conclusions and make further comparisons.

2 The Second-Order Monadic Logic on Strings

We employ a theory M2L(Str), where a closed formula is interpreted relative to a natural number n, called the *length*. A first-order variable p denotes a number i, $0 \leq i < n$, called a *position*. A second-order variable P denotes a subset of $\{0, \ldots, n-1\}$. Alternatively, a second-order variable can be viewed as designating a bit pattern $b_0 \ldots b_{n-1}$ of length n, where b_i is 1 if and only if i belongs to the interpretation of P.

First-order terms are formed from first-order variables, and constructors of the form 0 (denoting the position 0), \$ (denoting the last position $n-1$) and $t \oplus i$ and $t \ominus i$ where t is a first-order term and i is a natural number (denoting $j+i$ mod n and $j-i$ mod n, where j is the interpretation of t). Second-order terms are built from second-order variables, the constants empty and all (which denote $\emptyset$ and $\{0, \ldots, n-1\}$), and are combined using $\cap, \cup$, and $\complement$ (complement relative to $\{0, \ldots, n-1\}$). For t_1 and t_2 first-order terms and S a second-order term, $t_1 \in S$, $t_1 = t_2$, $t_1 < t_2$, $t_1 \leq t_2$, and $t_1 \geq t_2$ are formulas. Formulas are combined by the standard connectives $\neg, \wedge, \vee, \rightarrow$, and $\leftrightarrow$. There are two kinds of quantifiers: $\exists^1$ and $\forall^1$ over first-order variables and $\exists^2$ and $\forall^2$ over second-order variables.

As an example, the formula $\exists^1 p : p \in P$ states that "there exists a position p in P". More complex is

$$0 \in P \wedge \forall^1 p : p < \$ \rightarrow ((p \in P \rightarrow p \oplus 1 \notin P) \wedge (p \notin P \rightarrow p \oplus 1 \in P)),$$

which states that P contains exactly the even positions among $\{0, \ldots, n-1\}$. We also view P as a bit vector and write $P(p)$ for $p \in P$.

A formula ϕ, with free variables, defines a regular language denoting the interpretations that make ϕ true. For example, the formula $\phi \equiv P = \complement\, Q$ defines a language $L(\phi)$ over $\mathbb{B}^2$, where each $(b^1, b^2) \in \mathbb{B}^2$ denotes the membership status of the current position relative to the free second-order variables P and Q. The language is then the set of words describing interpretations of P and Q that make ϕ true. For example, if we represent the letter (b^1, b^2) as $\substack{b^1 \\ b^2}$ then

$$\substack{0110 \\ 1001} \in L(\phi) \quad \text{and} \quad \substack{011 \\ 000} \notin L(\phi).$$

This language represents the set of natural numbers n such that ϕ holds when interpreted over a string of length n.

We can construct an automaton recognizing $L(\phi)$ by standard operations of product, subset construction, and projection. A closed formula ϕ corresponds to an automaton over a one-letter alphabet and is a tautology when the automaton accepts strings of all lengths. For any formula ϕ that is not valid, we can extract from its corresponding automaton a minimal length string defining an interpretation making ϕ invalid. We use this procedure to generate counter-examples to proposed theorems. The automata construction we use is standard (see [17]), however, the implementation is made efficient by using BDDs to compress alphabets, which are exponential in the number of free variables. Implementation details are given in [10].

3 Parameterized Hardware

In this section we specify and verify the correctness of a simple parameterized n-bit ALU given in [14].

3.1 Preliminaries: Boolean Logic

Boolean connectives and quantification over Booleans are not part of M2L(Str), but can easily be encoded. In particular, each Boolean variable b is encoded by a second-order variable B, and occurrences of b in formulas are encoded as $0 \in B$. Quantification over Booleans ($\forall^0$ and $\exists^0$) is encoded using second-order quantification.[1] Non-parameterized circuits are then directly encoded using Boolean quantification and propositional connectives.

In hardware verification, circuits are traditionally represented as functions from inputs to outputs or as relations that hold between the port values. In the functional approach, components are connected by

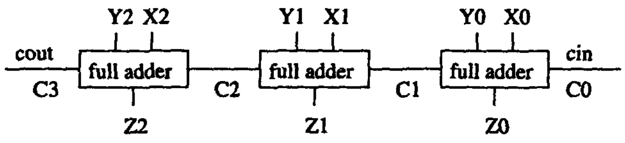


Fig. 1. n-bit adder for $n = 3$

functional nesting. In the relational approach, circuits are viewed as constraints and joined together using conjunction; internal wires are represented by shared variables that are existentially quantified (see [3, 6] for discussions on this). In our work it makes no difference which approach we use, we may even mix them.

To begin with we define gates as functions:

$$\begin{aligned}
not(a) &\equiv \neg a\\
and(a,b) &\equiv a \wedge b\\
or(a,b) &\equiv a \vee b\\
xor(a,b) &\equiv or(and(not(a),b), and(a, not(b)))\\
and3(a,b,c) &\equiv and(and(a,b),c)\\
or3(a,b,c) &\equiv or(or(a,b),c)
\end{aligned}$$

We may then define functions that compute the sum and carry bits for an adder and combine these into a relation specifying a full one-bit adder.

$$\begin{aligned}
sum(a,b,c) &\equiv xor(xor(a,b),c)\\
carry(a,b,c) &\equiv or(and(xor(a,b),c), and(a,b))\\
full\_adder(a,b,out,cin,cout) &\equiv (sum(a,b,cin) \leftrightarrow out) \wedge (carry(a,b,cin) \leftrightarrow cout)
\end{aligned}$$

Consider now an example of a simple theorem in Mona: if Boolean variables x, y, z, cin, and $cout$ fulfill the $full\_adder$ relation, then the outputs (z and $cout$) are uniquely determined given the inputs.

[1] For example the Boolean formula $\forall^0 x, y : \neg(x \wedge \neg y)$ is encoded as the M2L(Str) sentence $\forall^2 X, Y : \neg(0 \in X \wedge \neg(0 \in Y))$. Input to Mona consists of a sequence of definitions and a formula to be proved. The definitions are expanded using the Unix m4 macro processor and emacs macros. To spare the reader from concrete Mona syntax we will take liberties with syntax in what follows, for example manually pretty printing formulas.

$$\begin{array}{l}\forall^0 x, y, cin : \exists^0 z, cout : \mathit{full\_adder}(x, y, z, cin, cout) \\ \quad \wedge \forall^0 z', cout' : (\mathit{full\_adder}(x, y, z', cin, cout') \rightarrow ((z' \leftrightarrow z) \wedge (cout' \leftrightarrow cout)))\end{array}$$

Mona reports that this is a tautology in under 1 CPU second. That is, it reduces this formula to a 1-state automaton that accepts all strings; this indicates that there are no strings, of any length, that encode a counter-model to this formula.

3.2 Correctness of an n-bit Adder

We next turn to parameterized hardware. Figure 1 gives an example of this for $n = 3$. In the general case, an n-bit adder is constructed by wiring together n 1-bit adders where the carry-out of the ith becomes the carry-in of the i+1st. The first and last carry are special cases: the first carry has the value of the carry-in and the last has the value of the carry-out. We can directly specify this using two existentially quantified second-order variables C and D which represent sequences of carry-in and carry-out bits.

$$\begin{aligned} n\_add(X, Y, Z, cin, cout) \equiv \exists^2 C, D : & (\forall^1 p : \mathit{full\_adder}\ (X(p), Y(p), Z(p), C(p), D(p))) \\ & \wedge (\forall^1 p : (p < \$) \rightarrow (D(p) \leftrightarrow C(p \oplus 1))) \\ & \wedge (C(0) \leftrightarrow cin) \wedge (D(\$) \leftrightarrow cout) \end{aligned}$$

Having described an implementation, we now specify its functionality. M2L(Str) is a logic about strings and string positions, and an arithmetic specification must be encoded within this limited language. In particular, we encode addition as an algorithm over strings representing bit-patterns, i.e., binary addition. A simple way to do this is to mimic how addition is computed with pencil and paper. The ith output bit is set if the sum of the ith inputs and carry-in is 1 mod 2, and the ith carry bit is set if at least two of the previous inputs and carry was set. The 0th carry and the final values must be computed as special cases.

$$\begin{aligned} at\_least\_two(a, b, c) &\equiv (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \\ mod\_two(a, b, c, d) &\equiv a \leftrightarrow b \leftrightarrow c \leftrightarrow d \\ add(X, Y, Z, cin, cout) &\equiv \exists^2 C : C(0) \leftrightarrow cin \\ & \qquad \wedge \forall^1 p : mod\_two(X(p), Y(p), C(p), Z(p)) \\ & \qquad \wedge ((p < \$) \rightarrow (C(p \oplus 1) \leftrightarrow at\_least\_two(X(p), Y(p), C(p)))) \\ & \qquad \wedge (cout \leftrightarrow at\_least\_two(X(\$), Y(\$), C(\$))) \end{aligned}$$

We may automatically verify that *add* and *n_add* are equivalent: Mona reports in one CPU second that the following formula is a tautology.

$$\forall^2 X, Y, Z : \forall^0 cin, cout : add(X, Y, Z, cin, cout) \leftrightarrow n\_add(X, Y, Z, cin, cout)$$

Often we are interested in more than one property. With Mona, once preliminary definitions are made, it is easy to test them. For example, also in one CPU second Mona verifies that the n-bit adder computes a unique function from its outputs to its inputs:

$$\begin{array}{l}\forall^2 X, Y : \forall^0 cin : \exists^2 Z : \exists^0 cout : n\_add(X, Y, Z, cin, cout) \\ \quad \wedge \forall^2 Z' : \forall^0 cout' : (n\_add(X, Y, Z', cin, cout') \rightarrow (Z = Z' \wedge (cout \leftrightarrow cout')))\end{array}$$

Alternatively, we have checked (one CPU second) that our specification defines a commutative operation:

$$\forall^2 X, Y, Z : \forall^0 cin, cout : add(X, Y, Z, cin, cout) \leftrightarrow add(Y, X, Z, cin, cout)$$

3.3 Correctness of an n-bit ALU

We now apply our approach to a more complex circuit—a parameterized n-bit ALU. The circuit we analyze is presented in [14]. It is also an interesting theorem for comparison, since it has been recently verified in induction based systems CLAM and PVS (see comparison in §5).

ALU specification The ALU is designed to perform 8 arithmetic and 4 logical operations. The 12 functions are selected through 3 "selection" lines s_0, s_1, s_2 and the carry-in *cin* as described in table 1. For example when the s_i are 0 and *cin* is 1 the ALU increments the n-bit input A and places the result in F, producing a carry-out when every bit in F is set.

To specify the functionality of the ALU, we must specify each of these functional sub-units. We begin with abbreviations that define relational versions of the previously defined gates. For example:

$$\mathit{notrel}(a,b) \equiv \mathit{not}(a) \leftrightarrow b$$
$$\mathit{andrel}(a,b,c) \equiv \mathit{and}(a,b) \leftrightarrow c$$

Other relations used are defined analogously.

Selection					
s_2	s_1	s_0	cin	Output	Function
0	0	0	0	$F=A$	Transfer A
0	0	0	1	$F=A+1$	Increment A
0	0	1	0	$F=A+B$	Addition
0	0	1	1	$F=A+B+1$	Addition with carry
0	1	0	0	$F=A-B-1$	Subtract with borrow
0	1	0	1	$F=A-B$	Subtract
0	1	1	0	$F=A-1$	Decrement A
0	1	1	1	$F=A$	Transfer A
1	0	0	X	$F=A\vee B$	OR
1	0	1	X	$F=A\oplus B$	XOR
1	1	0	X	$F=A\wedge B$	AND
1	1	1	X	$F=\overline{A}$	Complement A

Table 1. Function Table for ALU

Using these we may directly define all of the logical and some of the simpler arithmetic functions.

$$\begin{aligned}
\mathit{transfer}(To,From) &\equiv To = From\\
\mathit{compl}(A,F) &\equiv \forall^1 x : \mathit{notrel}(A(x),F(x))\\
OR(A,B,F) &\equiv \forall^1 x : \mathit{orrel}(A(x),B(x),F(x))\\
XOR(A,B,F) &\equiv \forall^1 x : \mathit{xorrel}(A(x),B(x),F(x))\\
AND(A,B,F) &\equiv \forall^1 x : \mathit{andrel}(A(x),B(x),F(x))
\end{aligned}$$

The other function definitions require some basic arithmetic. For b a Boolean encoded by a second-order variable B, define $\mathit{zero}(b)$ to be $B = \mathit{empty}$ and $\mathit{one}(b)$ as $B(0) \wedge \forall^1 p : (p > 0 \rightarrow \neg B(p))$. We can now directly define the remaining arithmetic functions using the previously defined addition operator *add*.

$$\begin{aligned}
\mathit{add\_no\_carry}(A,B,F,cout) &\equiv \exists^0 cin : \mathit{zero}(cin) \wedge \mathit{add}(A,B,F,cin,cout)\\
\mathit{add\_with\_carry}(A,B,F,cout) &\equiv \exists^0 cin : \mathit{one}(cin) \wedge \mathit{add}(A,B,F,cin,cout)\\
\mathit{one\_compl\_add}(A,B,F,cout) &\equiv \exists^0 cin : \exists^2 Comp : \mathit{zero}(cin)\\
&\qquad \wedge \mathit{compl}(B,Comp) \wedge \mathit{add}(A,Comp,F,cin,cout)\\
\mathit{two\_compl\_add}(A,B,F,cout) &\equiv \exists^0 cin : \exists^2 Comp : \mathit{one}(cin)\\
&\qquad \wedge \mathit{compl}(B,Comp) \wedge \mathit{add}(A,Comp,F,cin,cout)\\
\mathit{decrement}(A,F,cout) &\equiv \exists^0 v : \mathit{one}(v) \wedge \mathit{two\_compl\_add}(A,v,F,cout)
\end{aligned}$$

Now, using the following auxiliary definitions

$$\mathit{if}_3(a,b,c,d) \equiv (a \wedge b \wedge c) \rightarrow d$$
$$\mathit{if}_4(a,b,c,d,e) \equiv (a \wedge b \wedge c \wedge d) \rightarrow e$$

we directly encode $\mathit{alu\_spec}(s_0,s_1,s_2,A,B,F,cin,cout)$ by specifying the function table (Table 1) as follows.

$$\begin{aligned}
&\mathit{if}_4(\neg s_2,\neg s_1,\neg s_0,\neg cin,\mathit{transfer}(A,F)) \wedge \mathit{if}_4(\neg s_2,\neg s_1,\neg s_0,cin,\mathit{increment}(A,F,cout))\\
&\wedge\ \mathit{if}_4(\neg s_2,\neg s_1,s_0,\neg cin,\mathit{add\_no\_carry}(A,B,F,cout)) \wedge \mathit{if}_4(\neg s_2,\neg s_1,s_0,cin,\mathit{add\_with\_carry}(A,B,F,cout))\\
&\wedge\ \mathit{if}_4(\neg s_2,s_1,\neg s_0,\neg cin,\mathit{one\_compl\_add}(A,B,F,cout)) \wedge \mathit{if}_4(\neg s_2,s_1,\neg s_0,cin,\mathit{two\_compl\_add}(A,B,F,cout))\\
&\wedge\ \mathit{if}_4(\neg s_2,s_1,s_0,\neg cin,\mathit{decrement}(A,F,cout)) \wedge \mathit{if}_4(\neg s_2,s_1,s_0,cin,\mathit{transfer}(A,F))\\
&\wedge\ \mathit{if}_3(s_2,\neg s_1,\neg s_0,OR(A,B,F)) \wedge \mathit{if}_3(s_2,\neg s_1,s_0,XOR(A,B,F))\\
&\wedge\ \mathit{if}_3(s_2,s_1,\neg s_0,AND(A,B,F)) \wedge \mathit{if}_3(s_2,s_1,s_0,\mathit{compl}(A,F))\ .
\end{aligned}$$

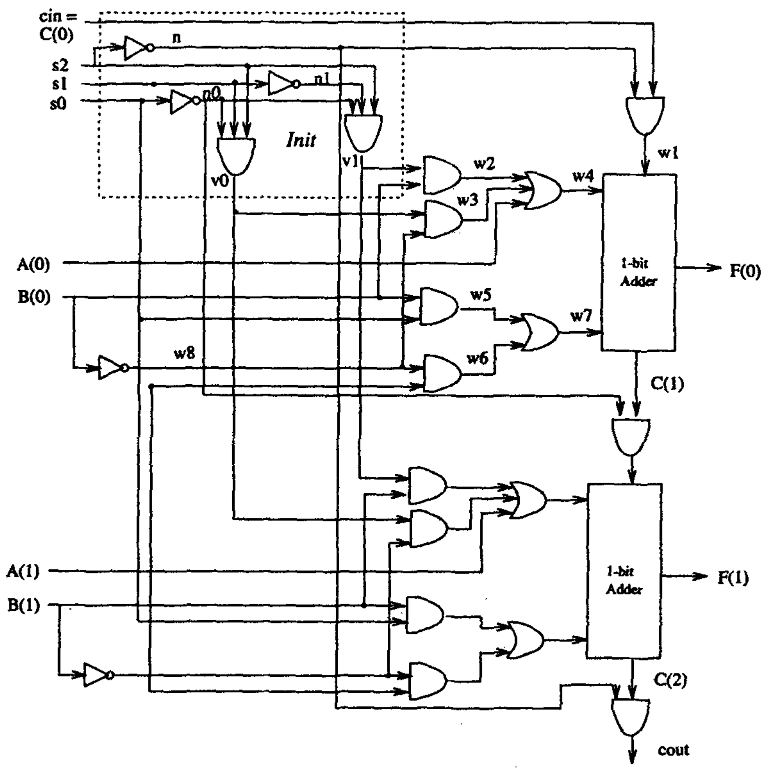


Fig. 2. n-bit ALU ($n = 2$)

ALU implementation The ALU implementation, as specified in [14] is given in Figure 2. The corresponding M2L(Str) formula is encoded analogously to the parameterized adder. The only additional complication is that the description should be subdivided into two parts: an initialization block and a repeating ALU block. The first part, which we call *init* computes negations of the selection wires and conjunctions of them and their negations.

$$\begin{aligned} init(s_0, s_1, s_2, v_0, v_1, n) \equiv \exists^0 n_0,\ n_1 &: notrel(s_0, n_0) \wedge notrel(s_1, n_1) \wedge notrel(s_2, n) \\ &\wedge and3rel(n_0, s_1, s_2, v_0) \wedge and3rel(n_0, n_1, s_2, v_1) \end{aligned}$$

The remainder of the ALU consists of the regular repetition of 1-bit ALU sections. These sections also require the switching wires s_i and the results of the *init* section computed on the wires v_0, v_1, and n.

$$\begin{aligned} one\_alu(a, b, f, cin, cout) \equiv \exists^0 w_1,\ w_2,\ w_3,\ w_4,\ w_5,\ w_6,\ w_7,\ w_8 &: andrel(n, cin, w_1) \wedge andrel(v_1, b, w_2) \\ &\wedge andrel(v_0, w_8, w_3) \wedge or3rel(w_2, w_3, a, w_4) \wedge andrel(b, s_0, w_5) \\ &\wedge andrel(w_8, s_1, w_6) \wedge orrel(w_5, w_6, w_7) \wedge notrel(b, w_8) \\ &\wedge full\_adder(w_4, w_7, f, w_1, cout) \end{aligned}$$

We may now combine the *init* block with ripple-carried 1-bit ALU units to specify the parameterized ALU. The ALU sections are hooked together as were the adder sections in the parameterized adder example.

$$\begin{aligned} n\_alu(s_0, s_1, s_2, A, B, F, cin, cout) \equiv \exists^2 C, D : \exists^0 v_0,\ v_1,\ n &: init(s_0, s_1, s_2, v_0, v_1, n) \\ &\wedge (\forall^1 p : one\_alu(A(p), B(p), F(p), C(p), D(p))) \\ &\wedge (\forall^1 p : (p < \$) \rightarrow (D(p) \leftrightarrow C(p \oplus 1))) \\ &\wedge (C(0) \leftrightarrow cin) \wedge (D(\$) \leftrightarrow cout) \end{aligned}$$

Mona verifies, in 27 seconds, the correctness of the ALU: when the inputs satisfy the circuit relation, they satisfy the ALU specification.

$$\forall^2 A, B, F : \forall^0 s_0, s_1, s_2, cin, cout : n\_alu(s_0, s_1, s_2, A, B, F, cin, cout) \\ \rightarrow alu\_spec(s_0, s_1, s_2, A, B, F, cin, cout)$$

Other properties, such as the functional relation between the inputs and outputs, are also easily checked.

4 Timed Hardware

We now consider circuits with timed specifications and feedback. A good example is the standard implementation of a D-type flipflop as shown in Figure 3. Although this circuit looks simple, understanding and demonstrating its correctness is a difficult task. A thorough and very well-written analysis of this flipflop is given by Hanna and Daeche in [9].[2] They used Veritas, a theorem prover based on a higher-order logic, to give a comprehensive analysis using a partial description of waveforms. Their analysis is complex, and it took an experienced user a week to construct the proof.

Our starting point is a simpler model of this circuit proposed by Gordon in [6]. He used a discrete representation of time and assumed each gate had a delay of one time unit. The proof that the circuit meets its specification, which he notes "is fairly complicated" was done by hand only. The flipflop and Gordon's specification are easily encoded in Mona with Gordon's choice of timing parameters. To our surprise, our system generated the following counterexample (in only 9 seconds).

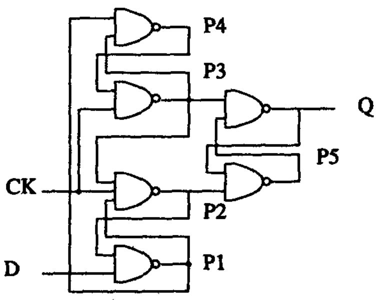


Fig. 3. D-Type Flipflop

D	11010100	CK	10110010
Q	10101111	$P1$	00010101
$P2$	11111111	$P3$	10100111
$P4$	11111101	$P5$	10101000

One sees that the D signal changes from 1 to 0 at the same time (instant 2) as the CK signal goes high. Therefore the value 0 is incorrectly propagated to the output gate Q. The D value is held constant for two time units before the rise of the clock as specified by [6], but the counter-example shows that the value must also be stable at the moment the clock rises.

Analysis of this failed proof attempt led us to discover that the theorem as stated in [6] does not hold without additional assumptions; in particular, that the circuit must not oscillate to begin with and that inputs D and CK are further constrained so as to prevent the circuit from becoming unstable.

To help us reason about events and intervals, we use the following predicates:

[2] Hanna and Daeche challenge the reader (page 193):

"It turns out, on analysis, that the *modus operandi* of this circuit is far from simple: in fact, it is unusually complex, and (so the authors found) difficult to understand intuitively. (If, like most people, you find this remark difficult to accept at face value, read the rest of this account, then set it aside, and attempt, within (say) one working day, to come up with a carefully justified account of 'how' the proposed implementation is intended to function...)"

- the value of F is stable in $[t_1, t_2]$:
  $stable(t_1, t_2, F) \equiv \forall^t t : t_1 \leq t \leq t_2 \rightarrow (F(t) \leftrightarrow F(t_1))$
- t_2 is the first instant after t_1 when F becomes high:
  $next(t_1, t_2, F) \equiv t_1 < t_2 \wedge F(t_2) \wedge (\forall^t t : t_1 < t < t_2 \rightarrow \neg(F(t))$
- F rises at t:
  $rise(t, F) \equiv t > 0 \wedge (\neg F(t \ominus 1) \wedge F(t))$
- F falls at t:
  $fall(t, F) \equiv t > 0 \wedge (F(t \ominus 1) \wedge \neg F(t))$

Also, we use the higher-order predicate (also just a macro) $eqpred(Out, Q, Ival)$, where Out is a second-order term, Q is a predicate $Q(p, I)$ with p a first-order variable and I a second-order variable. The predicate $eqpred(Out, Q, Ival)$ holds if and only if for all p, $Out(p) \leftrightarrow Q(p, Ival)$. Thus Out is the result of evaluating Q according to $Ival$. For example, $eqpred(P, rise, CK)$ holds if and only if P is the set of time instants for which the clock goes high.

4.1 The circuit

The temporal behavior of a unit-delay nand-gate with inputs I_1 and I_2 and output O is described by

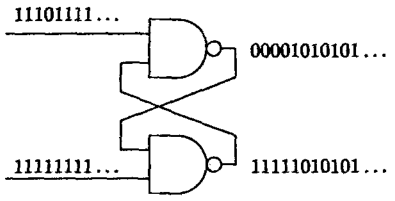
$$nand(I_1, I_2, O) \equiv \forall^t t : t < \$ \rightarrow O(t \oplus 1) \leftrightarrow \neg(I_1(t) \wedge I_2(t)).$$

If we call the corresponding predicate for three inputs $nand3(I_1, I_2, I_3, O)$, then the flipflop in Figure 3 is described by

$$\begin{aligned} dtype\_imp \equiv\ & nand(P_2, D, P_1) \wedge nand3(P_3, CK, P_1, P_2) \wedge nand(P_4, CK, P_3) \wedge \\ & nand(P_1, P_3, P_4) \wedge nand(P_3, P_5, Q) \wedge nand(Q, P_2, P_5). \end{aligned}$$

4.2 Stability analysis

In our model a simple flipflop may begin to oscillate due to a single negative spike:



One condition for the circuit to be stable is that the inputs do not change for a period of length *input_stable_time*. We define

$$\begin{aligned} input\_stable(t) \equiv\ & t + input\_stable\_time - 1 \leq \$ \\ & \wedge\ stable(t, t + input\_stable\_time - 1, D) \\ & \wedge\ stable(t, t + input\_stable\_time - 1, CK) \end{aligned}$$

to denote that inputs are stable for a period of length *input_stable_time*.[3] For our purposes, we regard the circuit as stable if both flipflops connected to the inputs are stable, i.e.,

$$\begin{aligned} circuit\_stable(t) \equiv\ & t + circuit\_stable\_time - 1 \leq \$ \\ & \wedge\ stable(t, t + circuit\_stable\_time - 1, P_1) \\ & \wedge\ stable(t, t + circuit\_stable\_time - 1, P_2) \\ & \wedge\ stable(t, t + circuit\_stable\_time - 1, P_3). \end{aligned}$$

(Note that P_4 need not be restricted for the stability analysis.)

[3] We here use + instead of $\oplus$. The formula $t < t' + 3$ holds if + is interpreted in the usual sense without "wrap-around." We need the conjunct "$t + input\_stable\_time - 1 \leq \$$" to prevent t from lying too close to the end.

Stability preservation of the circuit can be expressed informally as: if the circuit is stable at some t_s and if the inputs are held stable at t_i then there is $t'_s \geq t_i$ such that the circuit is stable at t. Thus, we define

$$\begin{aligned} \textit{stability\_preserved} \equiv (\exists^1 t_s : \textit{circuit\_stable}(t_s)) \rightarrow \\ (\forall^1 t_i : \textit{input\_stable}(t_i) \wedge \text{“}t_i \textit{ not too close to } \$\text{”} \rightarrow \\ \exists t'_s : t'_s \geq t_i \wedge \textit{circuit\_stable}(t'_s)) . \end{aligned}$$

Here "t_i *not too close to* $" is a condition that is necessary since formulas are interpreted over finite sequences only. We have chosen it to be simply *true*. But in general, some constant must be chosen so that the quantification $\exists^1 t'_s$ succeeds before "time runs out," i.e. before the finite segment of time that the logic is interpreted over ends.

4.3 Input requirements

Stability is not preserved unless the inputs are restrained. The clock signal must not form a negative spike of duration less than *min_clock_low* or a positive spike of duration less than *min_clock*. The D signal must be stable for at least *setup* units before CK rises. We define these conditions as

$$\begin{aligned} \textit{input\_requirements} \equiv \forall^1 t : (\textit{fall}(t, CK) \rightarrow \textit{stable}(t, \textit{min\_clock\_low} - 1, CK)) \wedge \\ (\textit{rise}(t, CK) \rightarrow \textit{stable}(t, \textit{min\_clock} - 1, CK)) \wedge \\ (\textit{rise}(t, CK) \rightarrow \textit{stable}(t - \textit{setup}, t, D) . \end{aligned}$$

Now, with the choices

min_clock_low	2
setup	2
circuit_stable_time	2
input_stable_time	4

Mona proves the implication

$$\textit{dtype\_imp} \wedge \textit{input\_requirements} \rightarrow \textit{stability\_preserved}$$

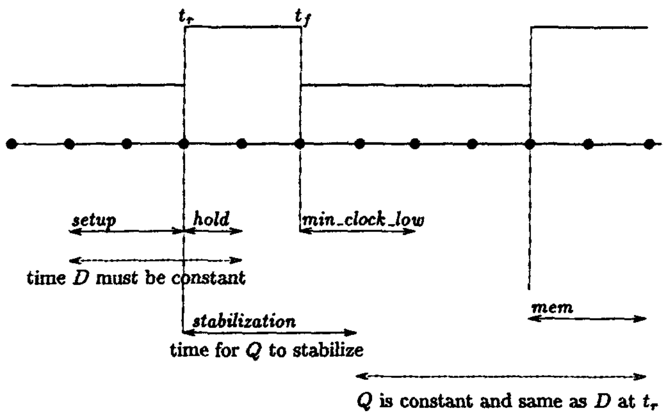
in about 15 seconds. These constants are optimal. For example, if we lower *setup* to 1, then the counter-example

$$\begin{array}{llll} t_i & = 001000 & P_1 & = 001111 \\ t_s & = 100000 & P_2 & = 111010 \\ D & = 100000 & P_3 & = 111010 \\ CK & = 001111 & P_4 & = \text{x}11010 \\ Q & = 101010 & P_5 & = 101010 \end{array}$$

is produced. Here we have made t_s and t_i free variables so that Mona can generate a counter-example that identifies the exact spot of trouble.[4] One clearly sees how the failure of maintaining the D signal stable before the rise of the clock results in oscillations despite that the inputs are later kept stable.

The essential D-flipflop behavior is as depicted below: if the circuit is stable at t_0 and the clock rises at t_r, then falls at t_f (after at least *min_clock* units from t_r), and then rises again at t'_r (after at least *min_clock_low* units from t_f), then the value of D at t_r appears at Q at time t_r + *stabilization* and remains there until time t'_r + *mem* provided that the D value is held constant in the period from t_r − *setup* until t_r + *hold*. This complicated set of circumstances is best illustrated in a diagram:

[4] Note that t_i and t_s are first-order position variables. These are actually encoded in Mona as second-order variables ranging over singleton sets. E.g., t_i and t_s point to the 2nd and 0th position respectively.



t_r
t_f
setup
hold
min_clock_low
time D must be constant
stabilization
time for Q to stabilize
mem
Q is constant and same as D at t_r

Formally, we express these conditions as:

$$\begin{aligned} dtype \equiv \forall\, t_0, t_r, t_f, t'_r :\ & \\ & (circuit\_stable(t_0) \wedge t_0 < t_r \wedge rise(t_r, CK) \\ & \wedge (\exists^2 P : eqpred(P, rise, CK) \wedge next(t_r, t'_r, P)) \\ & \wedge (\exists^2 P : eqpred(P, fall, CK) \wedge next(t_r, t_f, P))) \rightarrow \\ & (stable(t_r + stabilization, t'_r + mem, Q) \\ & \wedge Q(t_r + stabilization) \leftrightarrow D(t_r)) \end{aligned}$$

With the additional choices

min_clock	2
hold	1
mem	1

the implication

$$dtype\_imp \ \wedge\ input\_requirements \rightarrow dtype$$

is verified in about 15 seconds. Also, experiments show that these values cannot be lowered.

5 Statistics, Comparison, and Conclusions

The case studies presented here were carried out over two weeks using the Mona system. This time included adding extensions to the system to generate counter-models and debugging of specification errors caused by the primitive front end supported by the current implementation.

It is instructive to compare our proofs with previous proofs of these circuits. The parameterized adder and ALU were also studied by Cantu using the Edinburgh Clam System [5] and by Cyrluk *et. al.* using PVS [4]. CLAM is a system that generates proofs by induction for a higher-order logic (a constructive type theory). Cantu's development took two weeks and the proof is constructed automatically (in about 6 minutes) by CLAM. His specification shares some similarities to ours, but differs in several important respects. He specified the circuit as a recursive function while we specified it as a non-recursive relation. Both are valid representation techniques, but note that we cannot write explicit recursive functions in M2L(Str). On the other hand, if Cantu had specified a recursively defined relation, the system he uses would have been unable to construct a proof.[5]

The ALU theorem was also verified using PVS. PVS is a semi-interactive theorem prover that features built-in simplifiers and decision procedures; for example BDDs are used for propositional reasoning. Users can control proof construction by writing proof strategies (similar to tactics in the LCF sense). In [4] the adder and the ALU are verified using the induction, normalization, and BDD features of PVS. The

[5] To the best of our knowledge, all systems automating proof by mathematical induction are geared towards reasoning about recursively specified functions, but not recursively specified relations. Indeed, some provers used for hardware verification, such as NQTHM, are so biased towards functions that they cannot represent hardware specified relationally (e.g., they lack existential quantification).

formalization of these circuits is similar to Cantu's. Verification by induction of the parameterized adder is stated to last approximately 2 minutes (as opposed to our time of one second) although their proof of the ALU required only three times as much time (90 seconds versus our 27 seconds).

Reasoning about temporal properties of circuits like flip-flops can be carried out in systems based on different varieties of temporal logic. We believe, however, that the ability to refer directly to instants of time — instead of being restrained by a particular set of temporal modalities (which can be directly defined in Mona) — is a particular advantage when the behavior is as complex as that of the circuit studied.

As discussed in §4, verification of flipflops has been laboriously carried out in theorem provers based on higher-order logic and here the use of Mona brings real advantages. Of course, unlike interactive proof, our approach is inherently limited by the combinatorial explosion that follows when the number of variables or gates become bigger. However, being a subset of higher-order logic, our method is particularly suited for integration into traditional theorem proving systems and would supplement such systems in the same way as integration of BDD and model checking procedures [4, 11].

Acknowledgements

The first author was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. Responsibility for the content lies with the authors. The authors thank Alan Bundy, Harald Ganzinger, Tom Melham, and Natarajan Shankar for their feedback on a draft of this paper.

References

1. David Basin and Peter DelVecchio. Verification of combinational logic in Nuprl. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989. Springer-Verlag.
2. Albert Camilleri, Mike Gordon, and Tom Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Science Publishers B. V. (North-Holland), 1987.
3. Albert John Camilleri. *Executing Behavioural Definitions in Higher Order Logic*. PhD thesis, University of Cambridge, 1988.
4. Cyrluk D., S. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In *Second International Conference On Theorem Proving In Circuit Deisgn: Theory, Practice and Experience*, Bad Herrenalb, Germany, September 1994.
5. 1994 Francisco Cantu, Edinburgh DAI. Personal Communication.
6. Michael J. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North-Holland, 1986.
7. A. Gupta and Allen L. Fisher. Parametric circuit representation using inductive boolean functions. In C. Courcoubetis, editor, *Computer Aided Verification, CAV' 93, LNCS 697*, pages 15–26. Springer Verlag, 1993.
8. A. Gupta and Allen L. Fisher. Tradeoffs in canonical sequential function representations. In *Proceedings of the IEEE International Conference on Computer Design*, October 1994.
9. F.K. Hanna and N. Daeche. Specification and verification using higher-order logic: a case study. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 179–213. Elsevier Science Publishers B.V., 1986.
10. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. Technical Report RS-95-21, BRICS, Department of Computer Science, University of Aarhus, 1995.
11. Hardi Hungar. Combining model checking and theorem proving to verify parallel processes. In C. Courcoubetis, editor, *Computer Aided Verification, CAV' 93, LNCS 697*, pages 154–165. Springer Verlag, 1993.
12. Warren Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.
13. Warren J. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89–129. Elsevier Science Publishers B. V. (North-Holland), 1987.
14. M. Morris Mano. *Digital logic and computer design*. Prentice-Hall, Inc., 1979.
15. T. F. Melham. Using recursive types of reasoning about hardware in higher order logic. In *International Working Conference on The Fusion of Hardware Design and Verification*, pages 26–49, July 1988.
16. J.-K. Rho. and F. Somenzi. Automatic generation of network invariants for the verification of iterative sequential networks. In C. Courcoubetis, editor, *Computer Aided Verification, CAV' 93, LNCS 697*, pages 123–137. Springer Verlag, 1993.
17. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.