Formal Verification of Synchronous Circuits based on String-Functional Semantics: The 7 Paillet Circuits in Boyer-Moore

Alexandre Bronstein & Carolyn L. Talcott*

Computer Science Department
Stanford University, CA 94305
USA

Abstract

Since the beginning of time, the semantics of choice for synchronous circuits has been the finite state machine (FSM). Years of research on FSMs have provided many tools for the design and verification of synchronous hardware. But from a mathematical manipulation point of view, FSMs have several drawbacks, and a new hardware specification style based on the *functional* approach has gained ground recently. Earlier we described a functional semantics for synchronous circuits based on monotonic length-preserving (MLP) functions on finite strings. We have now implemented this theory inside the Boyer-Moore theorem prover, and proved correctness properties for a variety of circuits. In 1985 Paillet presented an interesting sequence of synchronous circuits of increasing "sequential complexity", with *hand*-proofs of their correctness in his P-calculus. Our semantics supports a calculus which extends his. It was therefore very tempting to investigate *mechanical* proofs of those circuits in the Boyer-Moore implementation of our theory. We present the results here.

1. Introduction

Synchronous circuits are still being verified in the industry the "old-fashioned" way: draft, simulate, and iterate, or worse: draft, build, test, and iterate. Fundamentally, we believe this is due to their existing underlying semantics: the finite state machine (FSM). FSMs have been studied for many years, and are the cornerstone of synchronous circuit design in engineering textbooks [2] [21]. Recent research has even shown successful formal verification of sequential circuits based on FSMs [10] [11], although mainly for asynchronous circuits. But in fact, recent formal verification attempts of synchronous circuits based on *theorem proving* also model the circuit at the outermost level as a finite state machine [9] [14].

However, FSMs are inherently operational, and non-compositional. This means that state-checking systems eventually face exponential explosions, and that theorem proving systems based on that model can not easily combine sequential sub-pieces. In response to that, in recent years an alternate specification style based on *functions* or "recursion equations" has appeared

---

*This research was partially supported by Digital Equipment Corp. and by ARPA contract N00039-84-C-0211.

[1] [4] [12] [13] [15] [16] [17] [23] [24] [25] [26] [27]. This offers a great potential for mathematical verification, since functional theories are inherently compositional, and easily amenable to standard mathematical manipulation.

To tap this formal verification potential requires formal semantics. In [8] we described a *functional semantics* for synchronous circuits based on monotonic length-preserving functions on finite strings. The technicalities are inessential for understanding the rest of this paper. The important point is that circuits are mapped to *functions* which we can meaningfully (and easily) compose. As a side-benefit we can just as easily model synchronous circuits with arbitrarily mingled registers and combinationals, such as pipelined designs, as circuits built according to the "standard" finite-state model (one register bank and one big combinational loop). To demonstrate the mechanizability of this theory, we have now implemented it inside the Boyer-Moore theorem prover [5] [7], and proved correctness properties for a variety of circuits.

In [25] Paillet exhibits an interesting sequence of synchronous circuits of increasing "sequential complexity", which involves more subtle aspects than the sheer number of registers or gates. This sequence contains 7 single-clock synchronous circuits, and 1 two-clock circuit. For each of those circuits, he gives a behavioral specification in his P-calculus, which is based on an informal stream-functional semantics (with Z-time), and *hand*-proofs of their correctness. Our semantics supports a calculus which extends his: the MLP-calculus, where "P" in fact denotes the same operator. It was therefore very tempting to use his list as a benchmark, and investigate *mechanical* proofs of those circuits in the Boyer-Moore implementation of our theory. At present, we have successfully proved the 7 single-phase circuits. (The two-phase circuit is handled by an extension of his model which we haven't formalized in our theory yet, and hence can not be treated mechanically in our system.)

One of the more interesting results from this benchmark exercise is the *layered methodology* we have devised to verify these circuits in Boyer-Moore (based on a large collection of circuit-independent theorems and a small Lisp circuit preprocessor) and which has turned out surprisingly versatile. The first 4 Paillet circuits and the 6th do not require any "build-up" lemma before obtaining a proof of the correctness statement. Only the 5th circuit requires one such lemma, which is quite natural for the circuit at hand — expliciting the next-state equations and proving them from the net list, and the 7th requires two: the next-state equations and the macro-cycle invariant.

The organization of this paper essentially follows this introduction. We begin with an *informal* introduction to the idea of history functional semantics to show how such a concept arises naturally in hardware, and describe our particular form of it. After which we compare Paillet's P-calculus with the calculus derived from our semantics. Then we describe our layered methodology for the mechanization of our theory in the Boyer-Moore theorem prover. And finally, we list the Paillet circuits and their corresponding correctness statements in Boyer-Moore, with some miscellaneous technical comments.

2. String-Functional Semantics - An Informal Exposition

Intuitively, the concept of history functional semantics is fairly simple, but it can seem a little foreign when one is used to thinking in terms of state-machines, events, and synchronization, so we first present here the *evolution* of thoughts which leads naturally to it — readers already familiar with the topic should feel free to skip this section.

Consider as a start a *combinational* circuit, i.e. a circuit with no memory (no registers and no feedback loops). Assume that the values which can appear on the wires are binary digits (True and False), then we can identify the circuit with a *boolean function.* This is the cornerstone of circuit design and not surprisingly, is a case of functional semantics! But in fact we can easily move from binary digits to natural numbers for example, and identify more general combinational circuits with functions on these numbers. Abstracting slightly, consider that the values on the wires belong to an arbitrary set: Σ , we can identify a combinational circuit with a function from Σ to Σ .

Once we introduce memory (or state) in the forms of feedback loops, or registers, things are not so simple. For example, consider an abstract running sum sequential circuit, which accumulates the sum of all the inputs it has seen. It is pictured below, with the square representing a register (initialized with 0) and the circle representing an adder.

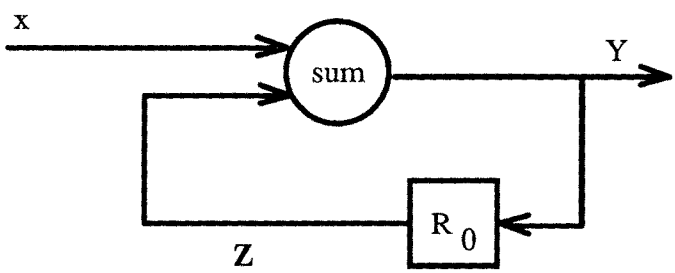

Figure 2-1: Running Sum Circuit

For this example, we have Σ = the set of natural numbers. Assume the first number we present as input is 3, the output is 3. The next number we present is 5, the output is now 8. The next number we present is 5 again, the output is now 13. Clearly, we can no longer identify this circuit with a function on the natural numbers, since it produced a different answer on the same input number.

The solution to this problem is to consider the *history* of all inputs, and the *history* of outputs; in our case: 3.5.5 $\rightarrow$ 3.8.13 . If we ever replay the same sequence of inputs (from the start) then we will get the same sequence of outputs. In other words, a sequential circuit can be identified with a function from sequences of values in Σ (called Σ^*) to sequences of values in Σ. A combinational circuit formerly identified with a function f: $\Sigma \rightarrow \Sigma$ is now identified with the "memory-less" function f* : $\Sigma^* \rightarrow \Sigma^*$ which to the input, say: a.b.c assigns the output: f(a).f(b).f(c) . In contrast, the function which corresponds to our register: R_0, assigns: 0.a.b to

the input string: a.b.c . And our running sum circuit outputs: a.(a+b).(a+b+c) to that same input.

Therefore our conclusion at this point is that a *synchronous circuit* can be identified with a function from Σ^* to Σ^* which we will call a *string-function.* Moreover, such functions have a fundamental property: they are Monotonic with respect to the Prefix relation on strings. To convince oneself of that, assume that on the input string x, the circuit returned the output string y . Now, assume that we add one more value u to x, making it the string: x.u , then the new output string already starts with y, and the circuit will tack on a new value v to y, yielding the output string: y.v . The circuit can not "go back in time", change some of the results it had output on input x, and produce an output string which does not start with y . As an additional remark, these functions produce an output string of the same length as the input, i.e. they are Length-Preserving, hence the name: MLP string-functions.

Next we take a look at how circuits are built. As far as we are concerned here, synchronous circuits are made from two kinds of elements:

- *Combinationals*: elements which do not have memory, or state, and which we have associated above with f* string-functions.
- *Registers/clocked storage*: elements which hold values for one clock period (after which they latch in the input presented to them), and which we have associated above with the R_a string-function. The parameter: a, is the initial value of the register; in the example above it was 0. Note that we use the word "register" in a very narrow sense, which is common in the formal hardware specification literature [16] [19].

Circuits are then built by connecting inputs and outputs of the above components in an almost arbitrary manner. We say "almost" because for a synchronous circuit, every loop in the connection graph should contain at least one register. Otherwise, we get problems of asynchronous latching, oscillations, etc., i.e. not a correct synchronous circuit; see [22] for more details. For our semantics, this restriction: "Every Loop is Clocked" is *not* necessary, and we discuss this in detail in [8] but here it will be easier to keep thinking in terms of such "good" circuits.

If for each element in the circuit we write an equation relating the output to the inputs, then we obtain a new view of our circuit as a system of equations. If there are loops in the circuit, then the system will be recursive. Note that this recursive system of equations is simply an abstract net list! O'Donnell [23] recently defended this hardware specification language ("Recursion Equations") very convincingly. One key argument there is that such a description is isomorphic to a schematic diagram, claimed as "the original Computer Hardware Description Language, and still one of the best", as it combines behavior and structure in a user-friendly fashion.

From a practical standpoint, all this means that once our theory is implemented in a theorem prover, it can potentially take as input simply the output of a CAD workstation!

3. Paillet's P-calculus and our MLP-calculus

Paillet's basic semantics [25] [26] [3] is also history-functional, but the basic objects are infinite streams, in fact infinite in both directions, i.e. isomorphic to Z. The main operator is the P or Passé [Past] operator, which truncates the last character of a stream, such that: (P(x))(t) = x(t-1) . P essentially models a register (when you ignore issues of initial state). Paillet then gives some "rules of combinations" (calculus) about P and stream-functions, which can be loosely summarized as "P and stream-functions commute".

When in [8] we built our (finite) string theory we constructed it from three functions:

- Add: add a character at the end of string; denoted as ".", or "A(x,u)",
- Last: take the last character of a (non-empty) string; denoted as "L(x)",
- Past**: remove the last character of a (non-empty) string; denoted as "P(x)".

When we studied the properties of MLP string-functions in this theory, i.e. tried to derive a calculus, we found (unaware of Paillet's work at the time) that one of the key properties of such functions was that they commute with the P operator (theorem "Reg-MLP" in [8]), modulo some trivial technicalities due to finiteness. In other words, our semantics supports the axioms of the P-calculus, and hence we can interpret, almost literally, P-calculus specifications in our system. There are however two distinctions which must be drawn:

First, Paillet often overloads "x" to mean the stream x or the value x(t), and the reader chooses the right meaning from the context. A mechanical system unfortunately demands more precision, hence our use of the L operator when we refer to the last (or current) value of a string.

Second, and more fundamentally, Paillet's streams are infinite. It makes many of the calculations simpler, by avoiding such cumbersome "if not empty x ..." or having results as "Y(x)" instead of "a . Y(x)" where "a" is some initial value output by the circuit. However, when initial values are finally needed for the reasoning, they are introduced with an "Init" functional, which "defines an N time scale and a zero time", but this seems artificial. In our case, we carry the weight of initial values from the start. For example, a register is modelled by the function: x $\rightarrow$ a . P(x), where "a" is the initial state of the register. But we need no extra apparatus when reasoning about circuits where initial values "matter".

4. Mechanization of the theory in Boyer-Moore

We had several criteria for a mechanization of our theory: existence (!), soundness, and usability.

Existence clearly was essential, but it was not immediate. Our semantics maps synchronous circuits into string-functions defined by mutual recursion. Many theorem provers have difficulty with such constructs. Boyer-Moore "officially" does not support mutually recursive

---

**Note that in [8] this operation was called All-But-Last. This improved terminology was inspired by Paillet's work.

function definitions [5], but one can use a standard logician's "trick" (found for example in [6]) to enter such definitions. This technique also requires an explicit well-founded relation from which the theorem-prover can prove termination of the definition. For that last purpose, we use a topological ordering of the nodes of the circuit which is well-founded if the circuit has no unclocked loops. Another essential fact which allowed us to enter the theory in Boyer-Moore is that we consider strings as 0-order objects, and hence circuits (string-functions) as 1st-order (in contrast to Paillet for whom P is a functional). This trade-off entails some loss of expressiveness, and we intend to investigate an implementation of our theory in a higher-order system to see the difference.

Soundness: we did not want to just concatenate a long list of axioms (like some HOL proofs sometime look like) and then reduce a theorem to True and claim victory. There are some serious consistency issues in such constructions, which HOL users readily admit. In our case, we do not have a single "Add-axiom" command in Boyer-Moore — the entire theory is built from:

- a "Shell" command for strings defining <Add, Last, Past> with axioms whose soundness is guaranteed by Boyer-Moore,
- "Defn" commands for all new concepts *and* circuit descriptions, whose soundness is checked by the theorem prover before accepting them.
- "Prove-lemma" commands for all other properties.

Usability: Rather than concentrate on one long sequence of theorems which finally prove the correctness of one large circuit, we wanted to develop a base from which one could (more) easily prove correctness for a variety of synchronous circuits. Toward that goal we have developed a *layered* methodology where all our definitions and theorems are separated into distinct classes:

- Circuit-Independent: subdivided into string-theory, arithmetic, string-function specification theory, and standard hardware types.
- Circuit-Dependent: subdivided into Boyer-Moore circuit definition, well-founded ordering proving termination (i.e. no unclocked loops), "general sequentiality" lemmas, and finally specific correctness properties.

Additionally, the first three sub-parts of the circuit-dependent half have been *automated*, i.e. we have a Lisp program which takes the net list of the circuit as input, and generates the right Boyer-Moore code: circuit definition in the right syntax, definition of the right topological ordering for the circuit, loading of the right library files for the combinational components, and instantiations of the general sequentiality lemmas. What we mean by general "sequentiality" lemmas is a list of half-a-dozen fundamental lemmas of the main string operators in relation to the function defined by the circuit (or combinational elements), which we have found useful during the course of our experiments. A couple of examples are given below.

```
; S-fun2 is an arbitrary combinational function (with 2 inputs)

(PROVE-LEMMA A2-LC-S-FUN2 (REWRITE)
    (IMPLIES (NOT (EMPTY X))
             (EQUAL (L (S-FUN2 X Y)) (FUN2 (L X) (L Y)))))

; Sysd is an arbitrary synchronous circuit (with 1 input)

(PROVE-LEMMA A2-PC-SYSD (REWRITE)
    (EQUAL (P (SYSD LN X)) (SYSD LN (P X))))
```

Figure 4-1: Examples of general sequentiality lemmas

This does not make all verifications trivial, by far! But as mentioned in the introduction, it makes the first 4 Paillet circuits and the 6th immediately provable by Boyer-Moore, while the 5th and 7th require only very few (and rather reasonable) intermediate lemmas. Note also that the same base of circuit-independent theorems was used to prove the correctness of the Saxe-Leiserson [20] retimed correlator also with no intermediate lemma, and of a pipelined adder [18] with only a couple of such lemmas.

5. The 7 Paillet Circuits in Boyer-Moore

For each of the circuits, we give first the diagram and specification from [25] and then the net list and specification lemmas in Boyer-Moore. The version of Boyer-Moore we use [7] accepts "hints" with the statement of the lemma to be proved. The hints we give are not shown here, as they sometime get large, and would not be very meaningful without a complete listing of our formalized string theory. Also, we apologize for some of our internal naming conventions: they are intended to make sense quickly when wading through screenfulls of Boyer-Moore output, and not necessarily look aesthetically pleasing in a report. We do not apologize for the annoying *hypotheses* which show up in some of our correctness lemmas but not in the hand-written specifications: painful precision is a sad but true fact of mechanical reasoning systems...

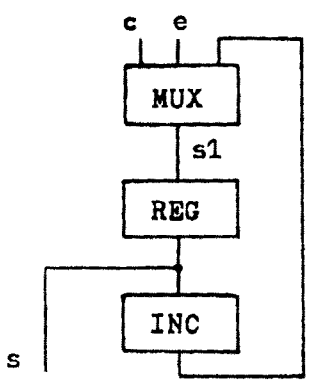

Figure 5-1: Paillet #1: Barrow-Gordon counter

s = COMPT(c,e) = si P c alors P e sinon INCN P s .

```
(setq sysd '(sy-COUNT (xreset xload)
(Ymux  S Mux xreset xload Yinc)
(Yreg  R 0 YMux)
(Yinc  S Incn Yreg)))

(prove-lemma count-paillet-correct-L ()
(implies (and (not (empty (P xreset))) (not (empty (P xload))))
         (equal (L (sy-count 'Yreg xreset xload))
                (IF (L (P xreset))
                    (L (P xload))
                    (incn (L (P (sy-count 'Yreg xreset xload))))))))
```

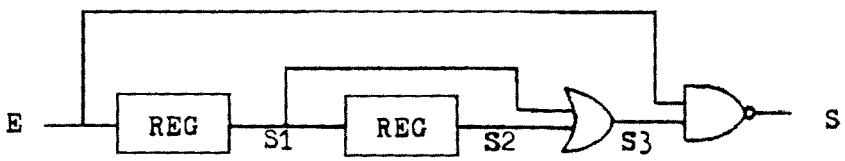

Figure 5-2: Paillet #2: simple BCD code checker

Un code DCB correct est un mot <b0 b1 b2 b3> tel que b0=0 ou b1=b2=0. Ceci peut s'écrire: non b0 ou non b1 & non b2 = 1 . S = f(E) = non E ou non P E & non P2 E .

```
(setq sysd '(sy-BCD (x)
(Y0 R 'r0 x)
(Y1 R 'r1 Y0)
(Y2 S bor Y0 Y1)
(Yout S bnand x Y2)))

; BCD-bits defines a correct binary coded decimal,
; b0 is most-significant.

(defn BCD-bits (b0 b1 b2 b3)
(or (equal b0 0)
    (and (equal b1 0)
         (equal b2 0))))

(defn BCD-Spec (x)
(BCD-bits (L x) ; most recent = most significant
          (L (P x))
          (L (P (P x)))
          (L (P (P (P x))))))

(prove-lemma BCD-correct2 ()
(implies (not (empty (P (P (P x)))))
         (equal (bibo (L (sy-bcd 'Yout x)))
                (BCD-Spec x))))

; Note: bibo converts: numeric bit -> boolean
```

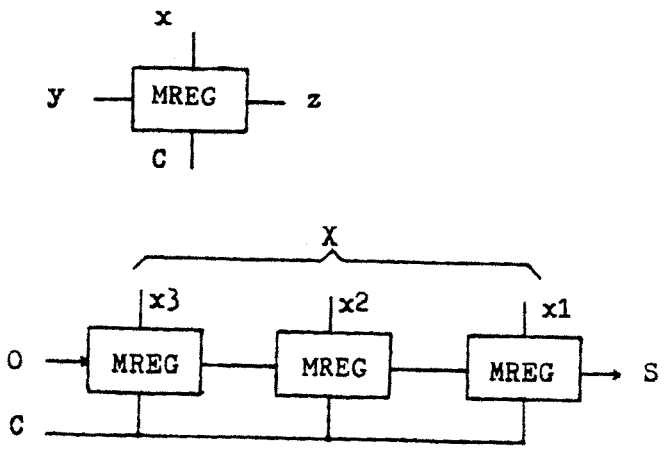

Figure 5-3: Paillet #3: parallel-load, serial-read register

z = MREG(C,x,y) = REG(MUX(C,x,y)) .

S = SERIAL(C,X) = P C -> P x1 ; P2 C -> P2 x2 ; P3 C -> P3 x3 ; 0 .

```
(setq sysd '(sy-SERIAL (xC x1 x2 x3)
(YC0 S const 0 xC)
(YM3 S mux xC x3 YC0)   (Y3 R 'a3 YM3) ; arbitrary initial value
(YM2 S mux xC x2 Y3)    (Y2 R 'a2 YM2)
(YM1 S mux xC x1 Y2)    (Y1 R 'a1 YM1)))

; Here we give a full string-function as spec:

(defn SERIAL-Spec (xC x1 x2 x3)
(I 'a1   ; "I" inserts a char at the front of a string.
  (S-IF (P xC)
        (P x1)
        (I 'a2 (S-IF (P (P xC))
                     (P (P x2))
                     (I 'a3 (S-IF (P (P (P xC)))
                                  (P (P (P x3)))
                                  (S-CONST 0 (P (P (P xC))))))))))))

(prove-lemma serial-correct (rewrite)
(implies (not (empty (P (P (P xC)))))
         (equal (sy-serial 'Y1 xC x1 x2 x3)
                (SERIAL-Spec xC x1 x2 x3))))
```

The next example is a 2-step serial Adder, attributed to Gordon. It is interesting because it introduces Paillet's Déroulement [Unrolling] operator, about which we would like to make the following comments: It seems essentially to be an unfolding of the recursive definition. As such it seems to be a proof tool, rather than a specification tool. It can be linguistically eliminated simply by realizing that Dn Y = ... means if length x = n then Y(x) = ... We have obtained these corresponding lemmas mechanically. But this yields rather *weak* specifications, where the induction (i.e. the fact that the entire circuit is re-initialized correctly) is "left to the

reader". We have therefore created more complete specifications, and proved them.

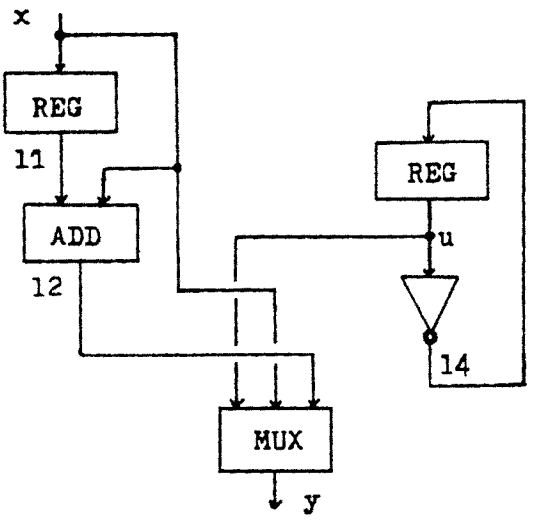

Figure 5-4: Paillet #4: 2-step serial adder

$y = u \text{ -> } x \,;\, P\,x + x\,.\quad u = P\ non\ u\,.\quad D1\ y\ \{1\} = P\,x + x\,.$

```
(setq sysd '(sy-SADDER (x)
(Y1 R 'a0 x)        (Y2 S plus Y1 x)      (Y4 S bnot Ydone)
(Ydone R 1 Y4)      (Yout S bmux Ydone x Y2)))

; we can do the actual Deroulement:

(prove-lemma sadder-Paillet ()
(implies (equal (len x) 2)
         (equal (L (sy-sadder 'Yout x))
                (plus (L (P x)) (L x)))))

; or either of these general specs:

(prove-lemma sadder-correct-1 (rewrite)
(implies (not (empty (P x)))
 (equal (L (sy-sadder 'Yout x))
        (if (equal (L (sy-sadder 'Ydone x)) 1)
            (L x)
            (plus (L (P x)) (L x) )))))

(prove-lemma sadder-correct-3 (rewrite)
(implies (and (not (empty x))
              (equal (remainder (len x) 2) 0))
         (equal (L (sy-sadder 'Yout x))    (plus (L (P x)) (L x)))))
```

Example 5 is a BCD code checker but which keeps track of how many bits it has seen (modulo 4) and hence with looped-registers. Here again, Paillet uses his Déroulement operator, and the same comments as above apply. We have obtained mechanical proofs of these

unfoldings, but these are not very satisfying. Instead we show the "complete" correctness lemmas (with Boyer-Moore verified induction).

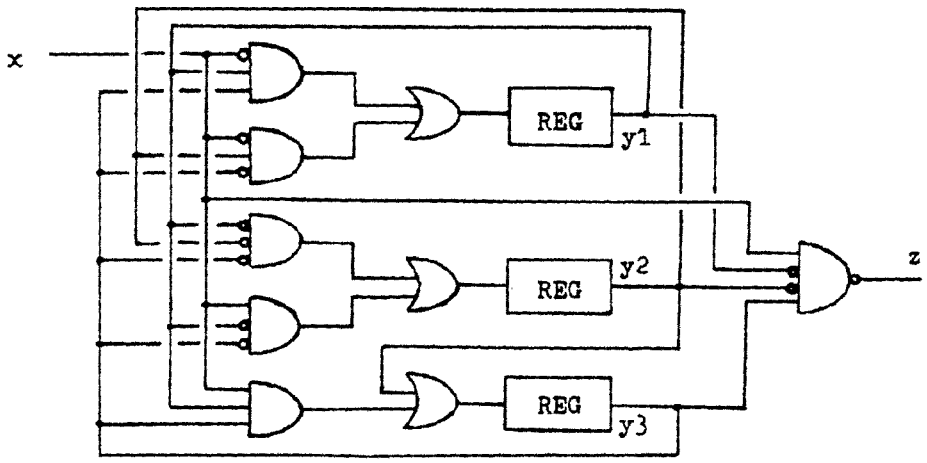

Figure 5-5: Paillet #5: BCD code checker

y1 = non Px & Py1 & Py3 ou non Px & Py2 & non Py3 ,

y2 = non Py1 & non Py2 & non Py3 ou Px & non Py1 & non Py3 ,

y3 = Py2 ou Px & Py1 & Py3 ,

z = non(x & non y1 & non y2 & y3) .

d0 Q = <0,0,0> , d0 Z = 1 , d1 Q = <0,1,0> , d1 Z = 1

d2 Q = <non P,P,1> , d2 Z = 1

d3 Q = <non P & non P2, 0, P2 ou P & non P2 > , d3 Z = non x ou non P x & non P2 x .

d4 Q(x) = <0,0,0>

```
(setq sysd '(sy-BCDS (x)
(Y01 S not x)              (Y02 S not x)           (Y03 S not YR3)
(Y04 S not YR1)            (Y05 S not YR2)         (Y06 S not YR3)
(Y07 S not YR1)            (Y08 S not YR3)
(Y11 S and3 Y01 YR1 YR3)   (Y12 S and3 Y02 YR2 Y03)
(Y13 S and3 Y04 Y05 Y06)   (Y14 S and3 x Y07 Y08)
(Y15 S and3 x YR1 YR3)     (Y21 S or Y11 Y12)      (Y22 S or Y13 Y14)
(Y23 S or YR2 Y15)         (YR1 R F Y21)           (YR2 R F Y22)
(YR3 R F Y23)              (Y31 S not YR1)         (Y32 S not YR2)
(Y41 S and4 x Y31 Y32 YR3)                         (Yout S not Y41)))

; SY-B2 has the next-state equations.

(defn SY-B2 (ln x)
(IF (EQUAL LN 'YR1) (IF (EMPTY X) (E)
 (I F (S-OR (S-AND3 (S-NOT (P x))
                    (SY-B2 'YR1 (P x))
                    (SY-B2 'YR3 (P x)))
            (S-AND3 (S-NOT (P x))
                    (SY-B2 'YR2 (P x))
```

```
                    (S-NOT (SY-B2 'YR3 (P x)))))))
(IF (EQUAL LN 'YR2) (IF (EMPTY X) (E)
 (I F (S-OR (S-AND3 (S-NOT (SY-B2 'YR1 (P x)))
                    (S-NOT (SY-B2 'YR2 (P x)))
                    (S-NOT (SY-B2 'YR3 (P x))))
            (S-AND3 (P x)
                    (S-NOT (SY-B2 'YR1 (P x)))
                    (S-NOT (SY-B2 'YR3 (P x)))))))
(IF (EQUAL LN 'YR3) (IF (EMPTY X) (E)
 (I F (S-OR (SY-B2 'YR2 (P x))
            (S-AND3 (P x)
                    (SY-B2 'YR1 (P x))
                    (SY-B2 'YR3 (P x))))))
(SFIX X)))))

(prove-lemma BCDS-is-B2 (rewrite) ;proved from net list
(and (equal (SY-bcdS 'YR1 x) (SY-B2 'YR1 x))
     (equal (SY-bcdS 'YR2 x) (SY-B2 'YR2 x))
     (equal (SY-bcdS 'YR3 x) (SY-B2 'YR3 x))))

(prove-lemma BCDS-Paillet-R-correct (rewrite)
(implies (and (not (empty x)) (S-boolp x))
(and
 (equal (L (SY-B2 'YR1 x))
  (if (equal (remainder (len x) 4) 1) F
  (if (equal (remainder (len x) 4) 2) F
  (if (equal (remainder (len x) 4) 3) (not (L (P x)))
       (and (not (L (P x))) (not (L (P (P x)))))))))
 (equal (L (SY-B2 'YR2 x))
  (if (equal (remainder (len x) 4) 1) F
  (if (equal (remainder (len x) 4) 2) T
  (if (equal (remainder (len x) 4) 3) (L (P x))
       F ))))
 (equal (L (SY-B2 'YR3 x))
  (if (equal (remainder (len x) 4) 1) F
  (if (equal (remainder (len x) 4) 2) F
  (if (equal (remainder (len x) 4) 3) T
       (or (L (P (P x))) (and (L (P x)) (not (L (P (P x))))))))))))

(prove-lemma BCDS-Paillet-Yout-correct (rewrite)
(implies (and (not (empty x)) (S-boolp x))
(equal (L (SY-BCDS 'Yout x))
       (if (equal (remainder (len x) 4) 0)
           (BCD-Lbits (L x) (L (P x)) (L (P (P x))) (L (P (P (P x)))))
           T))))
```

Example 6 is a (simple) handshake receiver part. The interesting aspect here is that Paillet proves an "Initialization property": that some correct input sequence correctly initializes the circuit, as well as standard correctness properties. We have obtained both: the standard correctness properties in the usual way, with a circuit initialized with the right values (0's in the registers), and the initialization property by giving Boyer-Moore an altered definition of the circuit *explicitely parametrized* on the (arbitrary) initial values of the registers. We then proved that the right input sequence yields the right values in the registers.

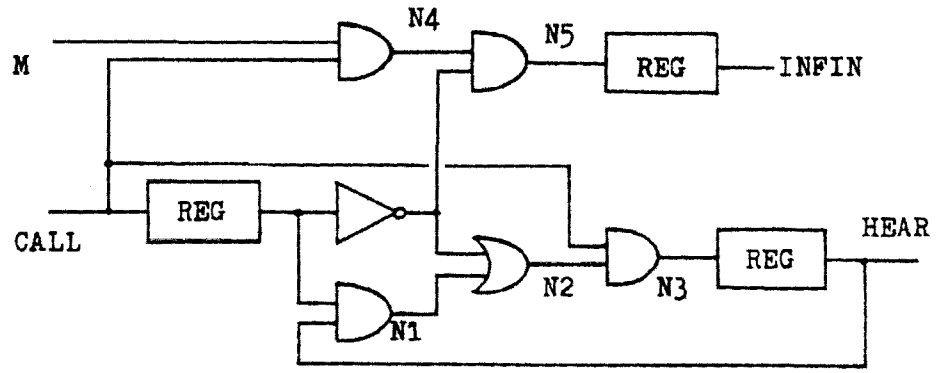

Figure 5-6: Paillet #6: simple handshake receiver

Soit encore (P CALL <--> HEAR) = 1 , donc HEAR = P CALL .
INFIN = P M & P CALL & non P2 CALL .
La condition CALL(t0)=0 est une condition d'Initialisation tout à fait "acceptable" vis-à-vis de l'utilisation du circuit.

```
(setq sysd '(sy-HANDREC (Xcall Xm)
(Yrcall R F Xcall)  (Y0 S not Yrcall)    (Y1 S and Yrcall Yhear)
(Y2 S or Y0 Y1)     (Y3 S and Xcall Y2)  (Y4 S and Xcall Xm)
(Y5 S and Y4 Y0)    (Yhear R F Y3)       (Yinfin R F Y5)))

(prove-lemma correct-handrec-hear-Paillet ()
(implies (and (not (empty (P Xcall))) (S-boolp Xcall))
         (equal (L (sy-HANDREC 'Yhear Xcall Xm)) (L (P Xcall)))))

; or:
(prove-lemma correct-handrec-hear-S (rewrite)
(implies (S-boolp Xcall)
(equal (sy-HANDREC 'Yhear Xcall Xm)
       (if (empty Xcall) (e) (I F (P Xcall))))))

(prove-lemma correct-handrec-infin-Paillet (rewrite)
(implies (and (not (empty (P (P Xcall))))
              (not (empty (P Xm))))
(equal (L (sy-HANDREC 'Yinfin Xcall Xm))
       (and (L (P Xcall)) (L (P Xm)) (not (L (P (P Xcall)))))))))

(prove-lemma correct-handrec-Init ()
(implies (and (equal Xcall_i (A (A (e) F) u1))
              (equal Xm_i (A (A (e) u2) u3)))
(and (equal (L (SY-HANDREC-i a1 a2 a3 'Yrcall Xcall_i Xm_i)) F)
     (equal (L (SY-HANDREC-i a1 a2 a3 'Yhear Xcall_i Xm_i)) F)
     (equal (L (SY-HANDREC-i a1 a2 a3 'Yinfin Xcall_i Xm_i)) F))))
```

Example 7 is the Gordon multiplier. One point worth mentioning here is that the circuit is specified at a relatively high level of abstraction with sub-components such as multiplexers, zero-tests, and adders, and lines carrying abstract natural numbers. Our theory (and

mechanization) handles this in *exactly* the same way as boolean gates and wires. That's one of the payoffs of the fairly abstract work in [8]. To be fair, this circuit also brings to light one of the technical drawbacks of our theory: the fact that we work in a domain of strings of equal length. In all the previous circuits, we had not needed to mention this hypothesis explicitely. In this one however we must, and this entails a computational, as well as aesthetic, cost.

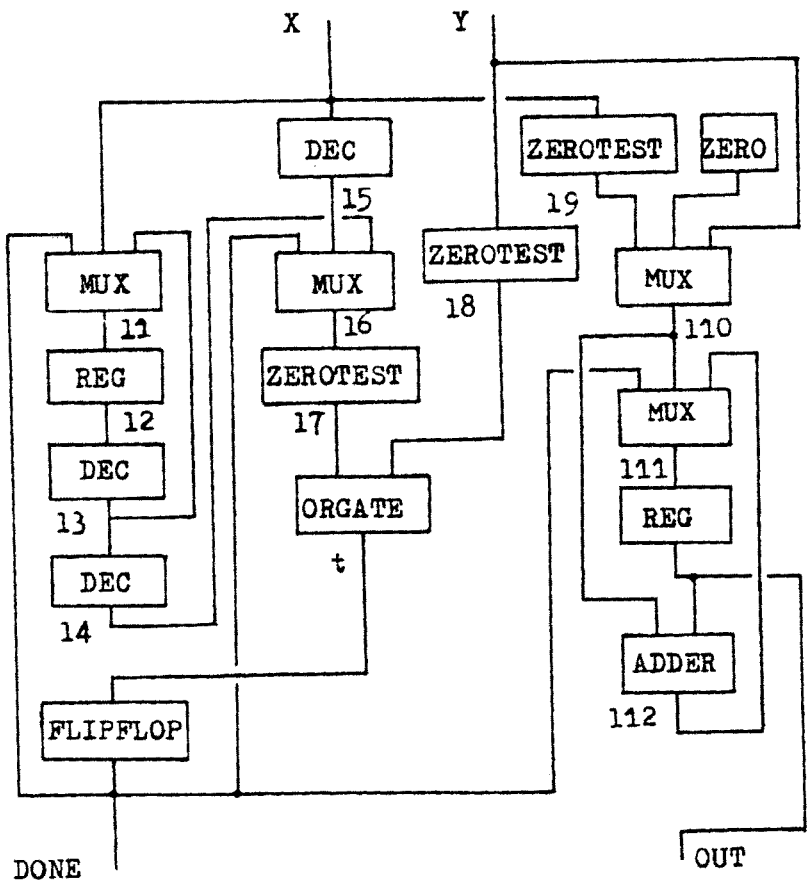

Figure 5-7: Paillet #7: Gordon multiplier

*(i) On peut déjà régler le cas trivial a=0 ou b=0: ce qui donne bien OUT =a * b .*
*(ii) Il reste à demontrer que le circuit implémente bien la multiplication, dans le cas a<>0 et b<>0 . La valeur de OUT est bien a * b .*

```
(setq sysd '(sy-MULTADD (Xx Xy)
(Y1 S mux Ydone Xx Y3)   (YN R 0 Y1)      (Y3 S dec YN)      (Y4 S dec Y3)
(Y5 S dec Xx)            (Y6 S mux Ydone Y5 Y4)            (Y7 S eq0 Y6)
(Yt S or Y7 Y8)          (Ydone R T Yt)   (Y8 S eq0 Xy)     (Y9 S eq0 Xx)
(Y0 S const 0 Xx)        (Y10 S mux Y9 Y0 Xy)  (Y11 S mux Ydone Y10 Y12)
(Y12 S plus Y10 Yout)    (Yout R 0 Y11)))

(prove-lemma MULTADD-CORRECT-CASE-0 (rewrite)
(implies (and (numberp Xa) (numberp Xb)
              (or (zerop Xa) (zerop Xb))
              (equal Xx (S-constL Xa 2))
              (equal Xy (S-constL Xb 2)))
  (and (equal (L (sy-multadd 'Yout Xx Xy)) (times Xb Xa))
       (equal (L (sy-multadd 'Ydone Xx Xy)) T))))

(prove-lemma MULTADD-CORRECT (rewrite)
(implies (and (numberp Xa) (numberp Xb)
              (not (zerop Xa)) (not (zerop Xb))
              (equal Xx (S-constL Xa (add1 Xa)))
              (equal Xy (S-constL Xb (add1 Xa))))
  (and (equal (L (sy-multadd 'Yout Xx Xy)) (times Xb Xa))
       (equal (L (sy-multadd 'Ydone Xx Xy)) T))))
```

6. Conclusions

From a practical standpoint, once the the bottom layers of the theory were sufficiently developed, with the right lemmas, induction schemes, etc., much of the time was spent on writing out a precise enough *specification*, rather than fighting the theorem prover. We believe this will be true in general: obtaining a completely formal, and yet meaningful, specification will be the hard part of mechanical verification.

More fundamentally, we started our work in synchronous circuit verification by an extensive development of the semantics and mathematics [8], so as to base our mechanical verification on sound foundations. The resulting string-functional semantics are intended to model easily general synchronous circuits with arbitrarily sprinkled registers and combinationals (such as pipelined or systolic designs) within a *functional* setting. We have now mechanized our theory in Boyer-Moore, and developed a methodology from which we can easily prove the correctness of the 7 Paillet circuits, and several others, with very strong soundness guarantees. We believe this supports the appropriateness of functional semantics for synchronous circuits, and the feasibility of theorem proving for their formal verification.

References

1. Amblard, Paul; Caspi, Paul; Halbwachs, Nicolas. Describing and Reasoning about Circuit Behaviour by Means of Time Functions . In *7th Int'l. Conf. on Computer Hardware Design Languages, pp. 39-48*, 1985.

2. Booth, Taylor L.. *Sequential Machines and Automata Theory* . John Wiley & Sons, New York, 1967.

3. Borrione, D.; Paillet, J.L.; Pierre, L. Formal Verification of CASCADE Descriptions . In *Int'l. Conf. on Fusion of Hardware Design and Verification, pp. 183-208*, 1988.

4. Boute, Raymond T. An Introduction to System Semantics . In *Embedded Systems (LNCS 284), pp. 91-107*, 1987.

5. Boyer, Robert S.; Moore, J Strother. *A Computational Logic* . Academic Press, New York, 1979.

6. Boyer, Robert S.; Moore, J Strother. A Mechanical Proof of the Unsolvability of the Halting Problem . ICSCA-CMP-28,Institute for Computing Science, Univ. of Texas at Austin, TX 78712, 1982.

7. Boyer, Robert S.; Moore, J Strother. The User's Manual for A Computational Logic . TR 18, CLInc, 1717 W. 6th St., Suite 290, Austin TX 78703, 1988.

8. Bronstein, Alexandre; Talcott, Carolyn L. String-Functional Semantics for Formal Verification of Synchronous Circuits . STAN-CS-88-1210, Computer Science Dept., Stanford University, CA 94305, 1988.

9. Cohn, A. A Proof of Correctness of the VIPER Microprocessor: The First Level . In *VLSI Specification, Verification and Synthesis, pp. 27-71*, 1988.

10. Dill, David L. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* Ph.D. Th., CMU-CS-88-119,Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh PA 15213, 1988.

11. Dill, D.L.; Clarke, E.M. "Automatic Verification of Asynchronous Circuits Using Temporal Logic ". *IEE Proceedings, v. 133, pt. E, no. 5, pp. 276-282* (1986).

12. Halbwachs, N.; Longchamp, A.; Pilaud, D. Describing and Designing Circuits by means of a Synchronous Declarative Language . In *From HDL Descriptions to Guaranteed Correct Circuit Designs, pp. 255-268*, 1987.

13. Hobley, K.M.; Thompson, B.C.; Tucker, J.V. Specification and Verification of Synchronous Concurrent Algorithms: A Case Study of a Convolution Algorithm. In *Int'l. Conf. on Fusion of Hardware Design and Verification, pp. 342-369*, 1988.

14. Hunt, Warren A. Jr. *FM8501: A Verified Microprocessor* . Ph.D. Th., TR 47, Institute for Computing Science, Univ. of Texas at Austin, TX 78712, 1985.

15. Johnson, Steven D. *Synthesis of Digital Designs from Recursion Equations* . Ph.D. Th., Indiana University, The MIT Press, Cambridge, Massachusetts, 1983.

16. Johnson, Steven D. Applicative Programming and Digital Design . In *11th Symp. on Principles of Programming Languages, Salt Lake City, pp. 218-227*, 1984.

17. Kloos, Carlos Delgado. *Semantics of Digital Circuits* . Ph.D. Th., Lecture Notes in Computer Science 285, Springer-Verlag, 1987.

18. Kogge, Peter M.. *The Architecture of Pipelined Computers* . Hemisphere Publishing, New York, 1981.

19. Leiserson, Charles E.; Saxe, James B. "Optimizing Synchronous Systems ". *Journal of VLSI and Computer Systems, v. 1, no. 1, pp. 41-67* (1983).

20. Leiserson, Charles E.; Saxe, James B. Retiming Synchronous Circuitry . TR 13, Digital Systems Research Center, 130 Lytton Av, Palo Alto CA 94301, 1986.

21. McCluskey, Edward J.. *Logic Design Principles* . Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1986.

22. Mead, Carver A.; Conway, Lynn A.. *Introduction to VLSI Systems* . Addison-Wesley, Reading, Massachusetts, 1980.

23. O'Donnell, John T. Hardware Description with Recursion Equations . In *8th Int'l. Conf. on Computer Hardware Description Languages, pp. 363-382*, 1987.

24. O'Donnell, J. HYDRA: Hardware Description in a Functional Language Using Recursion Equations and High Order Combining Forms. In *Int'l. Conf. on Fusion of Hardware Design and Verification, pp. 305-324*, 1988.

25. Paillet, J.L. Un Modèle de Fonctions Séquentielles pour la Vérification Formelle de Systèmes Digitaux. Rep. 546, IMAG-ARTEMIS, Grenoble, France, 1985.

26. Paillet, J.L. A Functional Model for Descriptions and Specifications of Digital Devices . In *From HDL Descriptions to Guaranteed Correct Circuit Designs, pp. 21-42*, 1987.

27. Sheeran, Mary. *μFP - an Algebraic VLSI Design Language* . Ph.D. Th., PRG-39, Oxford Univ. Computing Lab, 8-11 Keble Rd. Oxford OX1 3QD, England, 1983.