Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behaviour

Jean-Christophe MADRE
Jean-Paul BILLON

Bull Research Center
68, Route de Versailles
78430 Louveciennes FRANCE

Abstract

This paper presents a new method for verifying functionality in the design of VLSI circuits. Our method fits naturally in a methodology based on a Hardware Description Language (HDL). Two programs describe the system under design: (1) its specification and (2) the extracted behaviour from its layout. Verifying the design comes down to proving that these programs are *correct* and *equivalent* with regard to the HDL semantics. We define a process named *Formal Analysis* that permits to prove these properties without setting values to the programs inputs. *Formal Analysis* is based on a new canonical form of Boolean Logic that we name *Typed Shannon's canonical form.* We implemented this method in PRIAM, an efficient circuit prover now used by industrial CPU designers.

1. Introduction

Functional verification is the bottleneck of VLSI circuit design. As the economic constraints impose shorter design delays and the complexity of circuits increases, detecting design errors as soon as possible during the design cycle becomes a critical requirement. Errors detected in hardware cause unacceptable delays and costs in rectification. So, in zero defect methodologies, circuits must actually be completely verified before their physical processing.

One way to make error-free designs would be to use error-free silicon compilers. But experience shows that compilers must be continually improved and debugged in order to use the full power of the technology. So the verification task cannot be avoided. In current methodologies, *simulation* is used to validate the design, but this method cannot guaranty the design correctness.

Simulation consists in exciting a circuit description with 0s and 1s assigned to inputs and comparing its response with the expected one. Thus, the needed time for circuit verification is an exponential function of the number of inputs and is only reasonable for small circuits. Some methods have been proposed to reduce the number of patterns required to prove correctness [1], [2]. However, for complex circuits, this reduction is not suffisant and the correctness can only be statistically approached by simulation. For instance, verifying a mainframe CPU requires more than one year of CPU time with a software simulator.

In this paper, we present an efficient and complete method for formally verifying the functionality of circuits that are not verifiable by simulation. We inserted this method in the already existing Bull's VLSI design methodology. This methodology is based on a HDL called LDS. A circuit under design is described by two LDS programs: (1) its expected behaviour and (2) the description of the circuits just before their physical processing. These programs are respectively named the *specification* and the *realization* of the circuit. The design verification task then comes down to formally proving that *specification* and *realization* programs are *correct* and *equivalent* with regard to -wrt in the following- LDS semantics.

Other methods have been proposed to prove design correctness in the same framework. Odawara et al. [3] present a HDL and a comparison process. Emphasis is laid on the comparison process between boolean expressions in normal form and little is said about programs *reduction* to their normal form. In our method, we use a canonical form of Boolean Logic that makes comparisons trivial. The problem then comes from the reduction process which is not trivial due to the HDL semantics.

We define a process named *Formal Analysis* that permits to verify that LDS programs are *correct* wrt LDS semantics. Through this process we *reduce* them to a canonical form. Comparison between programs is done on their canonical form. *Formal Analysis* is made practically feasible by the use of a new canonical form of Boolean Logic we have defined and named *Typed Shannon's Canonical Form.*

This method has led to the implementation of a tool named PRIAM which is now used by industrial CPU designers. It has already been used to verify VLSI circuit parts. 32 bit datapath operators have been verified in less than a minute and design errors found. A microprocessor control section has also been verified in 30 minutes.

Section 2 presents the VLSI design methodology and in section 3 we show how our formal proof method fits into it. The proof method is fully described in section 4. Section 5 explains some fundamental results about *Typed Shannon's Canonical Form* of Boolean Logic. Section 6 presents some experimental results and our concluding remarks are given in section 7.

2. Formal proof in the Design methodology

This section describes Bull's methodology for VLSI based system design and the way our formal proof method fits into it. This methodology is based on a HDL named LDS [4]. It defines, through the following five main steps, the correct way to design and to verify highly integrated digital systems:

1. The system is described by a hierarchy of *modules*, each one corresponding to a system subpart. Each module is a LDS program composed of two parts:
   - a *structural* part describing the sub-system interface (inputs and outputs ports), its hardware components and the way it is built out of its own sub-parts. Hierarchy leaves are small blocks of circuitry such as data-path operators.
   - a *behavioural* part describing the sub-system functionality. It uses the hardware resources declared in the structural part. Synchronous systems are described at the cycle level. For example, a microprogrammed system is described at the top level by a set of micro-functions. Each of them corresponds to a set of elementary operations that the system can perform during *one* cycle. These micro-functions are then used to write micro-instructions.

2. The highest module in the hierarchy describes the system as it will be seen once realized. This module behavioural part is validated by simulation using patterns generated from the system specifications. For example, the microprogrammed system will be validated by its microprogram. Self-testing and critical sequences of assembly instructions are executed. They validate the microprogram, which itself validates the behavioural description. Each module in the hierarchy is validated in the same way. Once validated, a module M's behavioural description becomes the *specification* S_M of this module hierarchy, i.e. the function that the hardware described in M's structural part must perform.

3. The system is automatically or manually designed and layout is associated to the hierarchy leaves. Layout constitutes the ultimate control point for the designers before circuit processing.

4. A *functional extractor* [5] generates from the layout the behavioural description R_M (*realization*) of each hierarchy module M. Functional extraction and abstraction is done hierarchically through two steps:

   (1) A PROLOG program extracts the functionality of the hierarchy leaves from their layout.

   (2) Consider a module in the hierarchy. First its subparts behaviours are extracted. Then a second program assembles these behaviours according to the module structural part.

5. Here is the functional verification step. Consider a module M in the hierarchy. Through the preceding design steps, it has been associated to two behavioural descriptions, its *specification* S_M and *realization* R_M. Before we introduced PRIAM in the methodology, simulation was used to test the extracted behaviour. R_M was simulated with the same patterns that were used to validate S_M. If no error was detected and comparison of outputs values with the expected ones showed equality in any case, R_M and S_M were declared *equivalent* and the design validated.

The problem is that the simulation based verification cannot be complete because the circuit has a lot of inputs and is complex. Some input pattern causing damage to the circuit may not appear in the test. However, it may happen in the normal use of the circuit and damage it. Proving the design correct thus consists in proving that the programs are *correct* and *equivalent* wrt LDS semantics. Informally, correctness of a program P means that for every pattern that can be assigned to P's inputs, P's execution terminates without error. Equivalence between two *correct* programs P and Q requires that for every pattern that can be assigned to P's and Q's inputs, P's and Q's execution puts each output variable in the same state. The sets of patterns to be considered when proving *correctness* can be different for the *specification* and the *realization*, but they are the same when proving *equivalence*.

3. Correctness and Equivalence of LDS programs

LDS is an imperative language designed to hierarchically describe digital systems structure and behaviour. In this paper, we consider the LDS subpart dedicated to *synchronous systems* description.

LDS has different types of variables to describe hardware structure and behaviour. *Signals*, *booleans*, *registers*, *latches* correspond to hardware components such as memorizing elements and wires. Their semantics were specified from the working mode of their physical counterpart. *Variables* are algorithmic variables used for the sake of clarity and efficiency in the behavioural description. Variables can have more than one bit, e.g. a 32 bit *signal* is considered as 32 *signals* that are grouped together. In this paper we consider, without loss of generality, that the variables are single bit variables. The LDS statements which are relevant to the formal proof process are the following:

1. the simple assignment $var_1 := exp$;

2. the bidirectional assignment $var_1 :=: var_2$;

3. the conditional: IF *exp* THEN <*list_of_statement_1*>;
   ELSE <*list_of_statement_2*>;

where var_1, var_2 are of any of the above defined types, *exp* is an expression built out of variables with the usual boolean operators such as $\wedge$ (logical and), $\vee$ (logical or), arithmetic operators such as +, -, etc...

LDS *denotational semantics* gives a meaning to expressions, statements and programs written in LDS, e.g. it defines the most significant bit in a 32 bit *register* representing a number. Through a set of *semantic rules*, it also determines programs describing well built hardware. Here are the significant semantic rules:

(R1) LDS is a strict language. When an expression is evaluated, every variable occuring in it must have been previously assigned. For example, in the expression $(a \wedge b)$, both a and b must have a value, even if a or b is equal to 0.

(R2) Hardware associated variables (*signal*, *register*, *boolean*) cannot be assigned more than once in a cycle. So they cannot be assigned more than once in a program.

(R3) Algorithmic variables (*variables*) may be assigned more than once in a cycle, so they and can be reassigned in a program.

(R4) In a bidirectional assignment, $var_1 :=: var_2$, one and only one of var_1, var_2 must have a value. After execution of this statement, both variables have the same value. Such statements are generated by the functional extraction process when switches cannot be oriented.

Violations of semantic rules denote design errors. For instance, a *signal* double assignment may denote an implicit loop in the circuit or a possible short circuit; an unassigned *signal* occuring in an expression may correspond to a badly connected wire in the hardware. Extracted behaviours must be carefully checked against violation of the semantic rules because such a violation may result in damaging the circuit.

We say that a LDS program is *correct* wrt LDS semantics when, for every pattern that can be assigned to its inputs, execution terminates (no error is dectected). We then define *equivalence* wrt LDS semantics between two LDS programs: (1) The programs must describe the same circuit, with the same input and output variables, (2) they must be both *correct*, and (3) for every pattern that can be assigned to the programs inputs, execution of the programs puts each output variable in the same state.

The LDS semantic rules are easy to check when performing a numeric simulation. When input variables are assigned numeric values, execution can only put other variables in one of the three following states: unassigned, 0-valued, and 1-valued. Variable assignements in expressions and double assignements are easily detected. Final states comparaison is also immediate. Unfortunately, in order to prove that two programs are *correct* and *equivalent*, one has to execute them on every possible pattern. As the number of such patterns is very large, for instance 2^{67} for a 32 bit adder, one can only test the design on a small part of them.

4. Formal Analysis of LDS programs

Notations.

In the sequel, we will denote boolean variables by lower case letters such as a, b, ..., boolean expressions by upper case letters such as A, B, ..., and LDS variables and expressions by lower *italics* such as *a*, *b*, ...

We describe in this section a process that we name *Formal Analysis*. This process allows to verify that LDS programs are *correct* wrt LDS semantics without setting numeric values to their input variables. Through this process, which is based on a sequential symbolic execution, we also reduce LDS programs to a canonical

form. *Equivalence* proof of programs reduced to their canonical form then comes down to a syntactical equality test.

Verifying LDS semantic rules without setting numeric values to variables leads to some difficulties, as shown in this example:

Input *signal* c, e_1, e_2;
signal v;
IF $c = 1$ THEN $v := e_1$;
IF $c = 0$ THEN $v := e_2$;

In this program, the *signal* v seems to be assigned twice, thus violating the semantic rule R2. But a closer look at the statements shows that the two assignements are conditional and the conditions $(c = 1)$ and $(c = 0)$ are exclusive. So one and only one of these statements (depending on the value of c) is executed during an execution and v is assigned only once.

Formal analysis systems have already been used in hardware verification. When dealing with conditional statements, their classical answer is *case analysis* (also called *forcing*) [6], [7]. It consists in building the tree of all possible paths in the programs and verifying each path separately. For complex designs this tree is very large and, as the number of different paths grows exponentially with the size of the tree, these methods cannot be used.

4.1 Introducing *contexted values*

The way we solve this inefficiency and perform a pure symbolic analysis is by considering that conditional statements create *contexts* of assignement which have to be recorded with the assigned values. We attach to each program variable v a couple:

$$(C_v, V_v)$$

of boolean expressions. We name C_v the *assignment context* of v and V_v the *contexted value* of v. *Formal analysis* of a LDS program then consists in formally computing, using these couples, for each statement in the program: (1) its validity wrt LDS semantics, and (2) the changes produced on the *assignment context* and *contexted value* of the assigned variable.

The key idea [9] is that LDS semantic rules can be checked using *assignment context* and *contexted value*. This is shown on the former example. Executing the first statement (IF $c = 1$ THEN $v := e_1$;) assigns v if and only if the condition $(c = 1)$ is true. No error can occur because c is a program input. *Formal analysis* associates to v the couple:

$$(c,\ c \wedge e_1)$$

where, in the *assigned value*, the *assignment context* is associated to the value in order to form a discrimination net [8]. Then the second statement (IF $c = 0$ THEN $v := e_2$;) is analysed. If no error was detected, it would associate to v the couple:

$$(\neg c,\ c \wedge e_2)$$

Formal analysis verifies that these two couples can be combined without violating the semantic rule R2, and compute the resulting unique couple. First the *assignment contexts* of both couples are combined to check that they cannot be true at the same time. Then *assignment contexts* and *assigned values* are combined to form the final couple:

$$(1, (c \wedge e_1) \vee (\neg c \wedge e_2))$$

4.2 The reduction process

This section describes how Formal analysis of a LDS program is pratically performed. Consider a LDS program with inputs i_1, ..., i_k, outputs o_1, ..., o_m, and local variables l_1, ..., l_n. The program inputs may be constraint. This constraint is a boolean expression that we note A. We consider it as an axiom in the reduction process. For instance, the clock signals ck_1 and ck_2 of a two phase system must not overlap. The corresponding axiom is: $A = \neg(ck_1 \wedge ck_2)$. The *formal analysis* of the program is done on the inputs patterns satisfying the axiom A.

Remark. We do not make any difference between an input pattern and its boolean representation. For instance, the pattern (0, ...,0) assigned to $(i_1, ..,i_n)$ is represented by the expression: $(\neg i_1 \wedge ... \wedge \neg i_n)$.

Formal analysis of a LDS program is performed in the following way:

1. This is the initialization step. Inputs variables are given a symbolyc value. For instance, we attach to i_1 the couple $(1, i_1)$ that is interpreted as: i_1 is assigned and its value is 0 or 1. Local and output variables are considered unassigned before execution of the program. For instance, o_1 is attached the couple (0, 0) which is interpreted as: o_1 is not assigned.

2. A simple unconditional assignement such as $v := exp$ is analysed through the following four steps:

   (S1) Check that the expression *exp* can be evaluated without error. Each variable x occurring in *exp* must be assigned. The semantic rule R1 is respected if and only if whatever the pattern satisfying A is assigned to the program inputs, the execution of the statements preceding this one in the program has assigned x. This means that every pattern satisfying the boolean expression A also satisfies C_x. This is expressed by the formula:

   $$A \Rightarrow C_x.$$

   So we need a tautology checker in Boolean Logic. A lot of them have already been used in the framework of hardware verification. They are based on techniques as different as resolution, paramodulation, and canonical rewriting. We have chosen to built our prover on a new canonical form of Boolean Logic that we have defined and named *Typed Shannon's canonical form*. Proving that a boolean expression is a tautology is done by rewriting it to its canonical form which have to be equal to 1.

   PRIAM reduces the formula $(A \Rightarrow C_x)$ to its canonical form. If it not syntactically equal to 1, then an error message is produced and the error analysed. If it is equal to 1 then we compute the variable x's contexted value: $A \wedge V_x$.

   (S2) LDS expressions such as *exp* are represented in memory by trees defined by the LDS abstract syntax. We use a depth-first traversal of these trees to verify the rule R1. In this way we can compute in a bottom-up manner the canonical form of the boolean expression E associated to *exp*. E is a boolean expression of variables in $\{i_1, ..., i_k\}$.

   (S3) Check that this assignment is correct wrt LDS semantics. If v is a *variable*, its reassignment is allowed and no error can occur. If v is a *signal*, the rule R2 is respected if and only if whatever the pattern satisfying A is set to the program inputs, execution of the program statements preceding the current one has not already assigned v. This means that every pattern satisfying A must not satisfy C_v. This condition is expressed by the formula:

   $$A \Rightarrow \neg C_v.$$

   All input patterns considered in the process satisfy A, so this formula becomes: $C_v = 0$, which means that v has not been assigned by preceding statements. As C_v is kept in its canonical form, correctness is immediately stated.

   (S4) Compute changes in v's *assignment context* and *contexted value*. As the statement is unconditional, it is executed for every pattern that satisfies A. So v's new *assignment context* is:

   A.

   As for v's new *contexted value*, consider first that v is a *signal*. As v has not been assigned by the preceding program statements, we have: $V_v = 0$, and v's new *contexted value* is immediately: E. Now consider that v is a *variable*. Even if v has

already been assigned, it his reassigned because the statement is unconditional, so its new *contexted value* is: E.

3. Conditional statements need more attention. Consider the LDS statement: IF *cond* THEN v := *exp*. Its *formal analysis* is performed through 4 steps:

(C1) Check that *cond* can be evaluated without error. This is done in the same way than in (S1). Through this process, we get a boolean expression that we note COND. A pattern satisfying COND is such that, when it is assigned to the program inputs, the assignment v := *exp* is executed, whatever path has been taken through the preceding program statements.

(C2) Check that the expression *exp* can be evaluated without semantic error. Consider a variable x occurring in it. Consider a pattern that satisfies COND and assign it to the program inputs. Execution of the program statements preceding this one must have assigned x. So this pattern must satisfy C_x. We thus get a formula that looks like the one obtained in (S1):

$$\text{COND} \Rightarrow C_x.$$

PRIAM reduces this boolean expression to its canonical form which must be equal to 1. If no semantic error is detected then we return x's *contexted value*: COND $\wedge$ V_x. In a bottom-up manner, we compute the canonical form of the boolean expression *exp* that we note E.

(C3) Check the assignment correctness. If v is a *variable* no semantic error can occur. If v is a *signal*, then things get a little more intricate than for an unconditional assignment. Consider a pattern satisfying COND and assign it to the program inputs. As v cannot be assigned more than once in the program, execution of the preceding program statements must have let x unassigned. This means that this pattern does not satisfy C_x. So the rule R2 is respected if and only if every pattern which satisfies the boolean expression COND does not satisfy C_x. This is expressed by the formula:

$$\text{COND} \Rightarrow (\neg C_v).$$

Using De Morgan's laws, this formula is rewritten to:

$$\text{COND} \wedge C_v = 0.$$

PRIAM reduces the formula (COND $\wedge$ C_v) to its canonical form which must be equal to 0. If not, an error message is produced and the error is analysed.

(C4) Compute changes in v's *assignment context* and *contexted value*. For each pattern that satisfies C_v, v is assigned before the execution of this assignment. For each pattern that satisfies COND, v is assigned by the execution of this assignment. So v's new *assignment context* is:

$$\text{COND} \vee C_v.$$

As for contexted values, first consider that v is a *signal*. As COND $\wedge$ C_v = 0, the same is true for associated *contexted values*, so E $\wedge$ V_v = 0. *Contexted values* can be combined without introducing ambiguity in the discrimination net and v's new *contexted value* is:

$$V_v \vee E.$$

Now consider that v is a *variable*. There are different possible cases to consider. If we have: COND $\wedge$ C_v = 0 then we can combine *contexted values* in the same way than above. Now consider that COND $\wedge$ $C_v \neq 0$. If a pattern satisfies both C_v and COND, then execution of the program with this pattern as inputs initial value reassigns v. If a pattern satisfies C_v but not COND, then the execution of the program with this pattern as inputs initial value does not reassign v. So, v's new *contexted value* is:

$$(\neg\text{COND} \wedge V_v) \vee E.$$

The same mechanism is used to execute nested conditional statements and the boolean expressions associated to embedded conditions are combined by a conjunction.

4.3 Comparing programs

Consider a module *specification* and *realization* we want to prove equivalent with regard to the LDS semantics. Formal analysis verify that they are both *correct*, using each program associated set of input patterns. Through this process, each output variable o is attached a couple (CS_o, VS_o) (respectively (CR_o, VR_o)) describing the state of o after execution of the *specification* (resp. *realization*).

The programs are then proven *equivalent*. if and only if, for every pattern in the *specification* input patterns set, execution of the two programs puts every output variable in the same state. Using *assignment contexts* and *contexted values*, this can be expressed by the following formulas:

(E1) $CS_O \Rightarrow CR_O$ and
(E2) $CS_O \Rightarrow (VS_O \Leftrightarrow VR_O)$

where E1 expresses that whenever a variable gets assigned in the *specification*, it must get assigned in the *realization* and E2 expresses equality of the assigned values. Both conditions are easily verified by PRIAM, which rewrites, for each output variable, the corresponding expressions to their canonical forms which must be equal to 1.

When PRIAM detects a violation of a semantic rule or fails in showing *equivalence* of programs, it generates input patterns that show the error. For instance, consider that some output o is not attached the same *contexted value* for both programs. So VS_O and VR_O are not equivalent under the constraint CS_O. PRIAM can generate all of the input patterns such that VS_O and VR_O do not evaluate to the same value. Some methods, especially [11], use these patterns to locate errors in the hardware.

5. Typed Shannon's canonical form

Section 4 has shown that formally reducing LDS programs to their canonical form requires constant use of boolean expressions reduction. Most of PRIAM speed and power comes from the properties of an original canonical form of boolean logic we have developed and named *Typed Shannon's canonical form.*

Other canonical forms of boolean expressions have already been used in circuit design and verification. W. Buttner and H. Simonis [10] proposed a canonical form built on the boolean operators $\wedge$ (logical and) and $\oplus$ (logical exclusive or). They inserted it in a PROLOG system performing an extended unification (boolean unification). Their experiments in fault diagnosis and test pattern generation [11] have shown the power of the method. But this canonical form lacks *orthogonality*: performing an $\wedge$ (logical and) or an $\vee$ (inclusive or) between two expressions generates a lot of intermediate terms because of the distributivy of the $\wedge$ operator over the $\oplus$ operator.

5.1 Decision Graphs

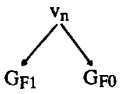
B.M.E. Moret [12] proposed *Decisions Graphs*, which are *directed acyclic graphs* (dag). A boolean expression is represented by a decision graph encoding its truth table. R. E. Bryant [13] used the decision graphs to define a whole set of algorithms for manipulating boolean expressions. These algorithms are now currently used in Boolean Logic based works. He demonstrated the technique on the proof of combinatorial operators, e.g. a 32 bit Arithmetic and Logical Unit was proven correct in 22 minutes on a VAX 780.

Consider a set $\mathcal{V}_n = \{v_1, ..., v_n\}$ of boolean variables and $\mathcal{F}_n$ the set of boolean functions of variables in $\mathcal{V}_n$ (the set of functions from $\{0, 1\}^n$ to $\{0, 1\}$). Consider a function F in $\mathcal{F}_n$. Using Shannon's *expansion theorem* [14], this function can be expressed in terms of a

unique couple of functions (F_1, F_0) of variables in $\{v_1, ..., v_{n-1}\}$:

$$F(v_1, ..., v_n) = (v_n \wedge F_1) \vee (\neg v_n \wedge F_0)$$

where F_1 (respectively F_0) is the function obtained by replacing every occurrence of v_n in F by 1 (resp. 0). In the same way, F_1 and F_0 can be decomposed with respect to v_{n-1} and this recursively defined decomposition constitutes F's *Shannon's canonical decomposition*. It can be represented by the following graph (where G_{F1} and G_{F0} are the graphs respectively associated to F_1 and F_0):

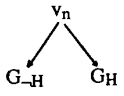


where the left edge corresponds to the case where $v_n = 1$ and the right edge to the case where $v_n = 0$.

Such graphs associated with Shannon's canonical form of boolean functions (or expressions) are binary trees which leaves are 0s or 1s. R. E. Bryant remarked that, in the field of hardware design, boolean functions associated graphs often have a regular structure which allows a compacted representation thanks to subgraphs sharing.

The problem arises with the $\neg$ (logical not) operator. To build $(\neg F)$'s associated graph, one has to build F's graph and then replace all the leaves by their opposite value, i.e. 0s by 1s and 1s by 0s. In the case where both F and $\neg F$ appear in a bigger expression, one has to make a copy of F's graph before modifying its leaves. In addition to the space loss due to copying, this leads to a waste of time in future graph computations due to subgraph sharing loss.

This is shown by the following example. Consider the graph associated to the expression $F = v_n \oplus H$ where H is a function of variables in $\{v_1, ..., v_{n-1}\}$:



where $G_{\neg H}$ and G_H are structurally the same except that their leaves have opposite values.

Our key idea [15] to overcome this inefficiency is to avoid copying and modifying H's graph to obtain $(\neg H)$'s one. This is done by introducing *typed edges* in the graphs, indicating the negation instead of performing it. The former graph thus becomes:

where the minus sign on the left edge indicates the negation to be performed on G_H in order to obtain F's Shannon's canonical decomposition.

In this new kind of graph, the same structure is used to represent an expression and its negation. As no copying is performed, this leads to an important time and space saving in graph computation. The question then is that there are several possible *typed* structures associated to an expression. We show in the next section that there is a *typed* canonical form of boolean expressions (derived from Shannon's form) that we name *Typed Shannon's canonical form*. This typed *canonical form* enables us to associate to any expression a unique *Typed* graph.

5.2 Definitions

Consider the above defined set $\mathcal{F}_n$ of boolean functions of variables in $\mathcal{V}_n = \{v_1, ..., v_n\}$. Using Shannon's way of decomposing functions into ones with less variables, we inductively define, for every $n \geq 0$, a partition of $\mathcal{F}_n$ into *two equipotent sets* $\mathcal{F}_{1n}$ and $\mathcal{F}_{0n}$:

1. $\mathcal{F}_{10} = \{1\}$, $\mathcal{F}_{00} = \{0\}$ where 1 (respectively 0) denotes the function with constant value 1 (resp. 0).
2. If F belongs to $\mathcal{F}_n$, its *Shannon's decomposition* with respect to v_n is: $F = (v_n \wedge F_1) \vee (\neg v_n \wedge F_0)$.
   If F_1 belongs to $\mathcal{F}_{1(n-1)}$ then F *is in* $\mathcal{F}_{1n}$ else F is in $\mathcal{F}_{0n}$.

Functions belonging to $\mathcal{F}_{10}, ..., \mathcal{F}_{1n}$ are said to be *positive*, other ones are said to be *negative*.

5.3 Fundamental results

Lemma 1.
Consider $n \geq 0$ and F in $\mathcal{F}_n$. F Shannon's decomposition with respect to v_n is: $F = (v_n \wedge F_1) \vee (\neg v_n \wedge F_0)$. Then:
$F = \neg((v_n \wedge \neg F_1) \vee (\neg v_n \wedge \neg F_0))$

Proof.
Using De Morgan's laws, we get :
$\neg F = (v_n \wedge \neg F_1) \vee (\neg v_n \wedge \neg F_0) \vee (\neg F_1 \wedge \neg F_0)$.
By the consensus rule, we can eliminate the third term (because of redundancy) and obtain:
$\neg F = (v_n \wedge \neg F_1) \vee (\neg v_n \wedge \neg F_0)$.
Now we have $F = \neg(\neg F)$ which gives the result.

Lemma 2.
Consider $n \geq 0$ and F in $\mathcal{F}_n$. Then: F is *positive* if and only if $\neg F$ is *negative*. This proves that for every n, $\mathcal{F}_{1n}$ and $\mathcal{F}_{0n}$ are equipotent sets.

Proof.
The proof is done by induction on n. The result is immediate for $n = 0$ (by definition). For $n > 0$, the proof is trivially done using lemma 1.

Theorem.
Consider $n \geq 0$ and a function F in $\mathcal{F}_n$. There is a unique couple (H_1, H_0) of functions *in* $\mathcal{F}_{n-1}$ such that H_1 is *positive* and F's Shannon's decomposition with respect to v_n is:
1. $F = (v_n \wedge H_1) \vee (\neg v_n \wedge H_0)$ if F is *positive*.
2. $F = \neg((v_n \wedge H_1) \vee (\neg v_n \wedge H_0))$ if F is *negative*.

The couple (H_1, H_0) is called the *Positively Typed* Shannon's decomposition of F with respect to v_n.

Proof.
F's Shannon's decomposition with respect to v_n is:
$F = (v_n \wedge F_1) \vee (\neg v_n \wedge F_0)$.
If F is *positive*, then the result is immediate by definition, H_1 is F_1 and H_0 is F_0.
If F is *negative*, then F_1 is also *negative*. Using the lemma 1, we get:
$F = \neg((v_n \wedge \neg F_1) \vee (\neg v_n \wedge \neg F_0))$.
Using the lemma 2, $\neg F_1$ is *positive*, we get $H_1 = \neg F_1$ and $H_0 = \neg F_0$.

This theorem shows that *Typed Shannon's decomposition* is a canonical form of functions of an ordered set of boolean variables $\{v_1, ..., v_n\}$. This form can be represented by graphs with *typed* edges as shown in the previous example. Introducing such *typed* edges in Bryant's graphs reduces both their size and the needed time to compute them. In particular, negation is performed in *null* time and we do not need a total graph sharing system such as Bryant's one. Subgraph sharing is natural and is not broken by logical operations.

6. Experimental results

PRIAM is written in an object oriented language named CEYX which is built on a LISP dialect called Le_Lisp. The results presented here were run on a SPS9 RIDGE/62.

6.1 Datapath combinatorial operators

Firstly, in order to compare *typed* decision graphs with decision graphs, we ran PRIAM on a set of ALUS taken from previously designed CPUs. We used the best order on the variables (commands, then most significant bits, then less significant bits). *Realizations* were extracted from gate level descriptions. The results were the following:

Operands size	Reduction time	Graphs Size
8 bits	17 s	646 nodes
16 bits	38 s	1358 nodes
32 bits	85 s	2782 nodes
64 bits	221 s	5630 nodes

There are two points worth mentioning about this example. First point is that attempts made at compacting the graphs showed that

they are naturally computed minimal in size. Second point is that needed CPU time and memory size needed for formal proof is proportional to the operands size. Doubling this size multiply CPU time by 2.3 approximately and graphs size by 2.1 .

We also verified combinational operators of under design CPUs. The *realizations* of these operators were extracted from their layout. A 32 bits adder with a sophisticated arborescent carry propagation had a 2 lines *specification* and a 700 lines *realization*, with dozens of local variables. *Formal analysis* was performed in 20 seconds. The equivalence test found a very local design error. This error had been unseen by simulation and was visible only on the 15th output. Final graphs were very small (350 nodes). A format decoder with 120 lines *specification* and a 2000 lines *realization* was also verified in 30 seconds and a design error found.

Large differences in specification and realization programs size are due to the functional extraction process. As it is based on pattern recognition, logical functions are locally extracted. In case of the 32 adder, global boolean expressions cannot be produced because of their complexity, so internal signals are kept as local variables in the extracted program. In the case of the decoder, transistors used as switches creates a lot of conditional assignments that cannot be reduced without formal analysis !.

6.1 µsyc control section.

µsyc [16] is a microprogrammable microprocessor designed to execute the fundamental operations in a relational database system. Its *specification* is a set of 150 micro-functions used to write its microprogram. From this specification we produced the control section *specification* (2500 lines). It is composed of the micro-sequencer and the 69 bits microword decoding part which generates 150 datapath control signals. Its *realization* was obtained by extraction from the gate level description. Errors were found during both *specification* and *realization* formal analysis. Needed time to perform *formal analysis* and *equivalence* test is about 30 minutes.

7. Conclusion

In this paper, we have presented an original approach to VLSI circuit design verification. Two ideas underly our formal proof method.

The first idea is to use an improved form of decision graphs to compute boolean expressions. Though our canonical form presents the same limits than Bryant's one [13], it is much more efficient. Simple heuristics are used to order the input variables. Some kinds of boolean expressions cannot be represented in a compacted way by decision graphs, for instance combinational multiplier expressions. We are working on methods to deal with such boolean expressions.

The second idea is to use the canonical form as the basis of a HDL semantic checker. This idea seems to be a good alternative to case analysis when dealing with conditional statements. Our *formal analysis* process check LDS programs in one pass and do not need any backtracking mechanism. Other HDL semantics can also be checked. For instance µsyc's description was not written in LDS.

Hardware descriptions considered in this paper are at the cycle level. However, the method can deal with sequential descriptions as well. The LDS semantic rules are extended and PRIAM can take them into account. Sequential descriptions correspond to automatons and we use *typed cyclic* graphs instead of simple *typed* decision graphs to represent them. We thus need to modify our current algorithms to compute such cyclic graphs and compare them.

Acknowledgment

The authors wish to thanks Miss Laugée and Miss Kerhervé for the suggestions and careful reading of the manuscript and the referees for their constructive criticism.

References

[1] R. Wei, A.L Sangiovanni-Vincentelli
PROTEUS: A Logic Verification System for Combinational Circuits, 1986 International Test Conference

[2] H. T. Ma et al.
Logic Verification Algorithms and their Parallel Implementation, 24th Design Automation Conference,1987

[3] G. Odawara et al.
A Logic Verifier Based on Boolean Comparison, 24th Design Automation Conference, 1986

[4] M. J. Bellosta, D. Jaillet, P. Mertens
LDS reference manual. BULL S.A. May 1987

[5] J. Y. Murzin
FAON: A Functional Abstractor of Netlist. Séminaire de Programmation Logique. Lannion, May 1986 (in french)

[6] H.G. Barrow
VERIFY: A Program for Proving Correctness of Digital Hardware Designs. Artificial Intelligence No 24, 1984

[7] M. S. Chandrasekhar, J.P. Privitera, K. W. Conradt
Application of Term Rewriting Techniques to Hardware Design Verification. 24th ACM/IEE Design Automation Conference 1987

[8] E. Charniak, C. K. Riesbeck, D. V. McDermott
Artificial Intelligence Programming, LEA Publishers, 1980

[9] J. C. Madre, J. P. Billon
Formal Proof of Combinatorial Circuits.
Bull Research Report No 87022. August 1987

[10] W. Buttner, H. Simonis
Embedding Boolean Expressions into Logic Programming
Journal of Symbolic Logic. January 1987

[11] H. Simonis, M. Dincbas
Using Logic Programming for Fault Diagnosis in Digital Circuits.
ECRC TR-LP-18. December 1986

[12] B. M. E. Moret
Decision Trees and Diagrams.
Computing surveys, Vol 14, No 14. December 1982

[13] R. E. Bryant
Graph-based Algorithms for Boolean Function Manipulation.
IEEE Transactions on Computers. Vol C35 No8, August 1986

[14] A. Macfarlane
Proceedings of the American Association for the Advancement of Science. Vol 39, p57, 1890

[15] J. P. Billon
Perfect Normal Forms for Discrete Functions
Bull Research Report No 87019, June 1987

[16] Michel Couprie et al.
µsyc, un co-processeur pour la machine parallele DDC, IVème journées Bases de Données Avancées, Bénodet, May 1988