RTL-Datapath Verification using Integer Linear Programming

Raik Brinkmann
Siemens AG, Corporate Technology
D-81370 Munich, Germany
raik.brinkmann@mchp.siemens.de

Rolf Drechsler
University of Bremen
28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

Abstract

Satisfiability of complex word-level formulas often arises as a problem in formal verification of hardware designs described at the register transfer level (RTL). Even though most designs are described in a hardware description language (HDL), like Verilog or VHDL, usually this problem is solved in the Boolean domain, using Boolean solvers. These engines often show a poor performance for data path verification.

Instead of solving the problem at the bit-level, a method is proposed to transform conjunctions of bitvector equalities and inequalities into sets of integer linear arithmetic constraints. It is shown that it is possible to correctly model the modulo semantics of HDL operators as linear constraints. Integer linear constraint solvers are used as a decision procedure for bitvector arithmetic. In the implementation we focus on verification of arithmetic properties of Verilog-HDL designs. Experimental results show considerable performance advantages over high-end Boolean SAT solver approaches. The speed-up on the benchmarks studied is several orders of magnitude.

1 Introduction

Register transfer level (RTL) hardware description languages (HDLs), such as VHDL and Verilog, are widely used for hardware design. Bitvectors are used to represent variables and their values by RTL-HDLs. Important arithmetic bitvector operators are for instance negation, addition, multiplication, extension, extraction, concatenation, and shifting. These operators are often used to model data paths of hardware designs. Many design automation tools operating on HDLs, including logic verification, test generation, and synthesis, have to solve the satisfiability problem for word-level formulas. Usually this is done by breaking down the problem onto the bit-level. But then datapaths in the RTL often lead to performance problems. Solving the SAT problem directly on the word-level, as opposed to first translating it into the Boolean domain, bears a high potential for optimization for those tools. Solving systems (conjunctions) of equalities and inequalities of arithmetic bitvector terms arises as a sub-problem of word-level satisfiability checking.

Recently, several theories of fixed sized bitvectors along with polynomial complexity bound decision procedures have been proposed [3, 10, 9]. They cannot handle arithmetic operations like addition and scalar multiplication, since deciding arithmetic over bitvectors is NP-hard. Other approaches, like [2], handle addition in their theory.

As an alternative the us of word-level decision diagrams (WLDDs) has been proposed and arithmetic circuits, like multipliers, have successfully been verified, but the integration in an automatic flow turns out to be difficult due to the exponential worst case behavior of the synthesis operations (for more details see [5]).

Cheng et al. [8] propose a hybrid ATPG modular arithmetic constraint solving technique for assertion checking. An arithmetic constraint solver based on the modular number system is used for satisfiability checking of a datapath portion. However this solver is limited to linear constraints arising from adders, subtractors and multipliers with one constant input. Shifters and inequalities are not handled by their modular solver but the possible solutions are heuristically enumerated.

Fallah [7, 6] proposed a hybrid satisfiability approach, HSAT, to generate functional test vectors for RTL designs. This hybrid method generates linear arithmetic constraints (LACs), for arithmetic operators, and conjunctive normal form clauses for Boolean logic. This approach was unified in [18], where both arithmetic word-level operators as well as Boolean parts are linearized yielding a single Integer Linear Constraint Problem (ILP) instance. However [7, 6, 18] model arithmetic operators straightforward, without their usual modulo semantics, which is particularly important as can be seen in the following:

Example 1: Let $A_{[n]}$, $B_{[n]}$ and $C_{[n]}$ be bitvector variables of width n, and $S_{[1]}$ a Boolean variable. (The width of bitvectors and operators is written as subscript.) Let A, B, C and S be linear variables. (Without loss of generality throughout this paper all integer variables are implicitly constrained to be non-negative.) In [7, 18] the following linearization rules were proposed:

- Addition: $C_{[n]} = A_{[n]} +_{[n]} B_{[n]}$ is modeled by the constraint $A + B - C = 0$.
- Comparator: $S_{[1]} = (A_{[n]} < B_{[n]})$ is linearized using the constraint pair $A - B - 2^n * (1 - S) \leq -1$ and $A - B + 2^n * S \geq 0$, with the additional constraints $A \leq 2^n - 1$, $B \leq 2^n - 1$ and $S \leq 1$.

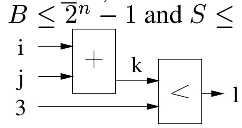

Figure 1. Example RTL

The problem is illustrated by the circuit in fig. 1. With $n = 2$ the equational bitvector representation is $k_{[2]} = i_{[2]} +_{[2]} j_{[2]}$, and $l_{[1]} = (k_{[2]} < 3_{[2]})$. Using the rules above linearization yields the ILP:

$$\begin{aligned} i + j - k &= 0 \\ k - 3 - 2^n * (1 - l) &\leq -1 \\ k - 3 + 2^n * l &\geq 0 \\ i &\leq 3 \\ j &\leq 3 \\ l &\leq 1 \end{aligned}$$

For $i = 3$ and $j = 1$ one would expect $l = 1$, since $i_{[2]} = 3_{[2]}$ and $j_{[2]} = 1_{[2]}$ implies $l_{[1]} = 1_{[1]}$, $3_{[2]} +_{[2]} 1_{[2]} = 0_{[2]}$

and $1_{[1]} = (0_{[2]} < 3_{[2]})$. But the only solution for the ILP is $l = 0$. With an additional constraint $k \leq 3$ the ILP becomes unsatisfiable while in the bitvector domain $l = 1$ still holds. The problem lies in the modeling of the addition, where the modulo semantics is ignored. For this, a straightforward modeling does not work for Verilog-like operators.

We propose a methods for transforming conjunctions of bitvector equalities and inequalities into equivalent conjunctions of equations and disequations in integer linear arithmetic. In particular bitvector terms are translated into linear terms, along with a set of linear arithmetic constraints (LACs). Propositions are directly translated into LACs (but could also be translated into terms, if desired). A conjunction of such constraints is referred to as an Integer Linear Constraint Problem (ILP).
For a given word-level SAT problem an equivalent ILP is generated, which has a solution iff the original bitvector system has a solution. This means that integer solutions can be transformed back into the bitvector domain. These solutions are then also solutions for the word-level SAT problem. The semantics of the word-level operators resembles that of Verilog-HDL. Boolean logic is not modeled because the focus lies on the datapath – however it could be integrated.
In Verilog-HDL arithmetic on bitvectors is always performed modulo and is therefore non-linear. Since a straightforward linearization of word-level arithmetic operators does not work, each bitvector term is linearized, such that it produces exactly the same result (w.r.t. coding integers as bitvectors).
In example 1 the equation $k_{[2]} = i_{[2]} +_{[2]} j_{[2]}$ would have been linearized to $i + j - k - 4\sigma = 0$ with the additional constraints $i + j - 4\sigma \leq 3$ and $\sigma \leq 1$. It is easy to see that this transformation preserves satisfiability and does not introduce more solutions.
Experimental results comparing the proposed technique to high-end SAT solvers show the advantage of the approach. For several examples the speed-up is several orders of magnitude, i.e. for benchmarks the SAT solvers take several minutes, while our techniques finishes the formal verification in less than one CPU second.

2 Bitvector Arithmetic

Bitvectors and the syntax and semantics of a conjunction of bitvector equalities and inequalities are defined.

2.1 Bitvectors

A bitvector $x_{[n]} \in \{0,1\}^n$ of width n is denoted from right to left as $(x_{n-1}, \ldots, x_0)$. The auxiliary functions nat_n and vec_n convert bitvectors into natural numbers and vice versa (such that $vec_n = nat_n^{-1}$). The i-th projection of $(a_{n-1}, \ldots, a_0)$ is denoted as $\pi(a_{[n]})_{n,i}$. For instance $a_{[n]}(i)$ denotes the i-th bit of $a_{[n]}$, and $nat(a_{[n]})$ denotes the integer value of the bitvector $a_{[n]}$ with $nat(a_{[n]}) = \sum_{i=0}^{n-1} 2^i a_{[n]}(i)$. If the width n of a is obvious or irrelevant, a_i is used as shorthand for $a_{[n]}(i)$. The boolean connectives $\bar{\cdot}$ (negation), $\vee$ (or), and $\wedge$ (and) over $\{0,1\}$ are defined as usual.

2.2 Syntax

A bitvector formula is a conjunction of propositions. Propositions are build over BV-terms. The operators involved in BV-terms are a subset of Verilog-RTL. The operators presented here are:

- Propositional: Equality, inequality, less.
- Arithmetic: Addition, summation[1], scalar multiplication[2], negation[3].
- Shift/Slice: Left and right shift, zero extend[4], sign extend[5], concatenation, extraction

Formally, let $t_{[n]}$, $BV_{[n]}$, $CBV_{[n]}$ denote a bitvector term, a bitvector (variable), and a constant bitvector of width n, respectively. Let $m, n, o \in \mathbb{N}$ and $k, p, q, c \in \mathbb{N}_0$. Then the syntax is formally defined in eq. 1, fig. 2. Multiplication could be integrated, along the lines of Fallah [6], using a for example a shift-add scheme.

2.3 Semantics

The interpretation of bitvector terms, propositions and formulas is defined as in Verilog-HDL. Formally an interpretation is a function that, given an assignment to the variables, assigns values of a domain to terms. Formally, given a domain $D = \cup D_n$, $D_n = \{0,1\}^n$. A variable assignment (environment) of variables Var_n of width n is a function $\varphi_n : Var_n \to D_n$ and $\varphi = \cup\varphi_n$. Given an environment φ, the interpretation of a syntactic element s under φ is written as $[\![s]\!]^\varphi$. Given a variable assignment φ the interpretation of bitvector terms and propositions is defined in equation 2, fig. 3. For example the concatenation of two terms $t^1_{[n_1]}$ and $t^2_{[n_2]}$ of width n_1 and n_2, resp., is denoted as $t^1_{[n_1]} \otimes t^2_{[n_2]}$ and for a variable assignment φ the interpretation, i.e. the value of this term, is denoted by $[\![t^1_{[n_1]} \otimes t^2_{[n_2]}]\!]^\varphi$. In this case the value of the concatenation is the bitvector $(a_{n_1-1}, \ldots, a_0, b_{n_2-1}, \ldots, b_0)$, where $(a_{n_1-1}, \ldots, a_0)$ and $(b_{n_2-1}, \ldots, b_0)$ are the values of $t^1_{[n_1]}$ and $t^2_{[n_2]}$ under φ or short $[\![t^1_{[n_1]}]\!]^\varphi$ and $[\![t^2_{[n_2]}]\!]^\varphi$. The interpretation of propositions is either true ($\top$) or false ($\perp$), as usual. For example, two terms are equal iff their interpretations (values under an assignment) are equal.

2.4 Circuit View

A bitvector formula can be viewed as a word-level circuit, i.e. a conjunction of propositions with a variable on the left hand side and a simple term on the right hand side. This view will ease the understanding of the linearization procedure later.
More precisely, a general bitvector formula f is a conjunction of propositions P where each proposition $p \in P$ consists of two terms t_l and t_r, one on the left hand side, one on the right hand side. It can be transformed into a simple bitvector formula by introducing intermediate variables. For t_l, t_r and each of their sub-terms intermediate variables are defined recursively. For each of them an equations representing the operation at the current level is added to the toplevel conjunction.
As an example consider the bitvector formula resulting from the RTL in fig. 4: $(a + b = c + (d << 4)) \wedge (a + b \leq d)$.

[1] Both, summation over a set of terms and addition of two terms are defined to clarify the importance of special linearization rules for each operator, although summation would have been sufficient.

[2] Multiplication with one input constant.

[3] Negation is viewed as arithmetic operation because it is defined on terms.

[4] Zero extend is performed implicitly in Verilog.

[5] Sign extend is needed to model properties, but is not part of Verilog.

$$
\begin{array}{rl}
t_{[n]} ::= & BV_{[n]} \mid CBV_{[n]} \mid (\neg t_{[n]}) \mid (t_{[n]} + t_{[n]}) \mid (\sum_{i=0}^{k-1} t^i_{[n]}) \mid (c * t_{[n]}) \mid (t_{[m]} \otimes t_{[o]})_{|_{n=m+o}} \\
\mid & (t_{[m]}[p,q])_{|_{n=p-q+1, m>p\geq q}} \mid (t_{[n]} \;>> p) \mid (t_{[n]} << p) \mid (t_{[m]}\, \mathbf{ZE}\, p)_{|_{n=p, p\geq m}} \mid (t_{[m]}\, \mathbf{SE}\, p)_{|_{n=p, p\geq m}} \\
proposition ::= & \top \mid \bot \mid t_{[n]} = t_{[n]} \mid t_{[n]} \neq t_{[n]} \mid t_{[n]} \leq t_{[n]} \mid t_{[n]} \geq t_{[n]} \mid t_{[n]} < t_{[n]} \mid t_{[n]} > t_{[n]} \\
formula ::= & proposition \mid (proposition \;\wedge\; formula)
\end{array}
\tag{1}
$$

Figure 2. Syntax of a Bitvector Logic

$$
\begin{array}{lrl}
\text{Variable:} & [\![x_{[n]}]\!]^\varphi = & \varphi(x_{[n]}) \\
\text{Constant:} & [\![1_{[1]}]\!]^\varphi = & (1) \\
 & [\![0_{[1]}]\!]^\varphi = & (0) \\
 & [\![1c_{[n]}]\!]^\varphi = & (1, c_{n-1}, \ldots, c_0) \text{ with } (c_{n-1}, \ldots, c_0) = [\![c_{[n]}]\!]^\varphi \\
 & [\![0c_{[n]}]\!]^\varphi = & (0, c_{n-1}, \ldots, c_0) \text{ with } (c_{n-1}, \ldots, c_0) = [\![c_{[n]}]\!]^\varphi \\
\text{Negation:} & [\![\neg t_{[n]}]\!]^\varphi = & (\overline{a_{n-1}}, \ldots, \overline{a_0}) \text{ with } (a_{n-1}, \ldots, a_0) = [\![t_{[n]}]\!]^\varphi \\
\text{Concatenation:} & [\![t^1_{[n_1]} \otimes t^2_{[n_2]}]\!]^\varphi = & (a_{n_1-1}, \ldots, a_0, b_{n_2-1}, \ldots, b_0) \text{ with} \\
 & & (a_{n_1-1}, \ldots, a_0) = [\![t^1_{[n_1]}]\!]^\varphi, (b_{n_2-1}, \ldots, b_0) = [\![t^2_{[n_2]}]\!]^\varphi \\
\text{Extraction:} & [\![t_{[m]}[p,q]]\!]^\varphi = & (a_p, \ldots, a_q) \text{ with} \\
 & & (a_{m-1}, \ldots, a_0) = [\![t_{[m]}]\!]^\varphi, 0 \leq q \leq p < m \\
\text{Shift left:} & [\![t_{[n]} \;<<\; p]\!]^\varphi = & [\![t_{[n]}[n-1-p, 0] \otimes 0_{[p]}]\!]^\varphi \\
\text{Shift right:} & [\![t_{[n]} \;>>\; p]\!]^\varphi = & [\![0_{[p]} \otimes t_{[n]}[n-1, p]]\!]^\varphi \\
\text{Zero extend:} & [\![t_{[m]}\, \mathbf{ZE}\, p]\!]^\varphi = & [\![0_{[p-m]} \otimes t_{[m]}]\!]^\varphi \\
\text{Sign Extend:} & [\![t_{[m]}\, \mathbf{SE}\, p]\!]^\varphi = & (a_{p-1}, \ldots, a_0) \text{ with} \\
 & & (b_{m-1}, \ldots, b_0) = [\![t_{[m]}]\!]^\varphi \\
 & & a_i = b_i \forall i \in \{0, \ldots, m-1\} \\
 & & a_i = b_{m-1} \forall i \in \{m, \ldots, p-1\} \\
\text{Addition:} & [\![t^1_{[n]} + t^2_{[n]}]\!]^\varphi = & vec_n(nat_n([\![t^1_{[n]}]\!]^\varphi) + nat_n([\![t^2_{[n]}]\!]^\varphi)) \\
\text{Sum:} & [\![\sum_0^{k-1} \{t^0_{[n]}, \ldots, t^{k-1}_{[n]}\}]\!]^\varphi = & [\![t^0_{[n]} + (t^1_{[n]} + \cdots (t^{k-2}_{[n]} + t^{k-1}_{[n]}) \cdots)]\!]^\varphi \\
\text{Scalar Multiplication:} & [\![c * t_{[n]}]\!]^\varphi = & [\![t^1_{[n]} + t^2_{[n]} + \cdots + t^k_{[n]}]\!]^\varphi \text{ with } t^i_{[n]} = t_{[n]} \text{ if } k > 0 \\
 & = & 0_{[n]} \text{ if } k = 0 \text{ and } 2^n > k \equiv_{2^n} c \\
\text{True:} & [\![\top]\!]^\varphi = & \top \\
\text{False:} & [\![\bot]\!]^\varphi = & \bot \\
\text{Equal:} & [\![t_1^n = t_2^n]\!]^\varphi = & \top \text{ iff } [\![t_1^n]\!]^\varphi = [\![t_2^n]\!]^\varphi \\
\text{Inequal:} & [\![t_1^n \neq t_2^n]\!]^\varphi = & \top \text{ iff } [\![t_1^n]\!]^\varphi \neq [\![t_2^n]\!]^\varphi \\
\text{Greater:} & [\![t_1^n > t_2^n]\!]^\varphi = & \top \text{ iff } nat_n([\![t_1^n]\!]^\varphi) > nat_n([\![t_2^n]\!]^\varphi) \\
\text{Less:} & [\![t_1^n < t_2^n]\!]^\varphi = & \top \text{ iff } nat_n([\![t_1^n]\!]^\varphi) < nat_n([\![t_2^n]\!]^\varphi) \\
\text{Greater of Equal:} & [\![t_1^n \geq t_2^n]\!]^\varphi = & \top \text{ iff } [\![t_1^n = t_2^n]\!]^\varphi \vee [\![t_1^n > t_2^n]\!]^\varphi \\
\text{Less or Equal:} & [\![t_1^n \leq t_2^n]\!]^\varphi = & \top \text{ iff } [\![t_1^n = t_2^n]\!]^\varphi \vee [\![t_1^n < t_2^n]\!]^\varphi
\end{array}
\tag{2}
$$

Figure 3. Semantics of Bitvector Operators and Propositions

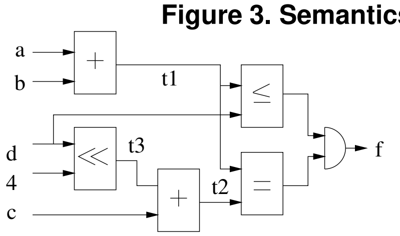

Figure 4. Word-Level Circuit

It will be transformed into:

$$
\begin{array}{cll}
 & t_1 & = t_2 \\
\wedge & t_1 & \leq d \\
\wedge & t_1 & = a + b \\
\wedge & t_2 & = c + t_3 \\
\wedge & t_3 & = d << 4
\end{array}
$$

An intermediate variable is defined for each internal signal of the circuit. This obviously does not change the satisfiability of the formula. Without loss of generality simple bitvector formulas will be referred to as bitvector formulas in the sequel.

2.5 Integer Linear Constraint Problems

Within the framework proposed, bitvector formulas are translated into equivalent integer linear constraint problems (ILPs). An ILP is a conjunction of linear arithmetic constraints (LACs). A LAC has either the form $\sum_{1\leq i\leq n} a_i x_i = c$ (equality constraint) or $\sum_{1\leq i\leq n} a_i x_i \geq c$ (inequality constraint). All a_i and c are integer constants. The variables x_i are implicitly constrained to be non-negative. With $x_0 = 1$ a conjunction of LACs, an ILP, can be written as the following (Presburger) formula[6]:

$$
\begin{array}{rl}
\exists x_1, x_2, \ldots, x_n : & \left(\bigwedge_s 0 = \sum_j c_{sj} x_j\right) \\
 & \wedge \left(\bigwedge_t 0 \leq \sum_j c_{tj} x_j\right), x_0 = 1
\end{array}
\tag{3}
$$

It contains s equality and t inequality LACs over n variables x_j. The constants c_{sj} and c_{tj} are the coefficients of the variable x_j in the s'th and t'th LAC, respectively. A solution of

[6] Note that eq. 3 only involves existential quantification and conjunction, but neither universal quantification, disjunction, nor negation.

an ILP is an assignment to the variables $x_1, x_2, \dots, x_n$ such that all LACs are satisfied. Solving an ILP is equivalent to checking eq. 3 for satisfiability:

Many efficient ILP solvers exist, running in polynomial time for many cases, however solving ILPs in its general form is an NP-complete problem [14].

In the following, we use the Omega test [12], however any ILP solver can be used. There are even automata theoretic approaches, translating ILPs into concurrent number automata (CNA) and checking for non-emptiness of the accepted language [17, 15].

The Omega test first performs preprocessing, including elimination of equality constraints and unbound variables, normalization of the constraints, and checks for directly contradictory or redundant constraints. All of these steps have polynomial time bounds [12]. The core procedure is a polyhedra approach, where the problem is subsequently reduced in dimension, first checking for real solutions and then, if they are present, for integer solutions. It relies on Fourier-Motzkin variable elimination [4], which can be expensive. But the cases where it does appear to be rare in practice [12].

3 Linearization

To use the Omega test, a given (simple) bitvector formula is translated into an ILP. The idea is to convert a bitvector formula f into a conjunction of linear constraints such that the linear constraint problem (see eq. 3) has a solution iff f has a solution. A one-to-one mapping of bitvector variables to linear variables is used, such that the solutions of the bitvector problem can be calculated from the solutions of the ILP. In the following for each operation the corresponding transformation is given. Then the overall algorithm flow becomes very simple, since only the corresponding rule has to be applied.

3.1 Modeling of Modulo in ILP

Since operations on bitvectors are performed modulo, a possible overflow will be discarded. In general, operations on bitvectors are performed modulo $2^{vector\ width}$. Therefore the modulo operator has to be modeled in integer linear arithmetic. Some operations require to discard some of the least significant bits which could be resembled by integer division. Neither modulo nor division is part of integer linear arithmetic.

But using quasi linear constraints they can be modeled as follows: Consider an equality $e = \alpha \ \mathbf{mod} \ m$ with $e \leq m-1$ in the natural domain. This can be written without the modulo operator as $0 \leq \alpha - m\sigma \leq m-1$, $e = \alpha - m\sigma, m = \mathbf{const}$, with no constraints to σ. (This is basically the definition of the modulo operator.) For integer division the procedure is similar: $e = \alpha \ \mathbf{div} \ m$ as $0 \leq \alpha - m\sigma \leq m-1, e = \sigma, m = \mathbf{const}$.

Since in the bitvector domain all variables have a finite domain, σ can be constrained:

$$
\begin{aligned}
a = nat(b_{[y]}) \ \mathbf{mod} \ 2^x \Leftrightarrow \quad & 0 \leq b - 2^x\sigma \\
& \wedge b - 2^x\sigma \leq 2^x - 1 \\
& \wedge a = b - 2^x\sigma \\
& \wedge a \leq 2^x - 1 \\
& \wedge b \leq 2^y - 1 \\
& \wedge \sigma \leq 2^{y-x} - 1 \\
a = nat(b_{[y]}) \ \mathbf{div} \ 2^z \Leftrightarrow \quad & 0 \leq b - 2^z a \leq 2^z - 1 \\
& \wedge a \leq 2^{y-z} - 1 \\
& \wedge b \leq 2^y - 1
\end{aligned}
$$

Linearization techniques are given for single bitvector operators, propositions and formulas. The following notation for the conversion is used:

$$\frac{\textit{bitvector term to be converted}}{\textit{converted integer term along with constraints}}$$

The integer term is build over integer variables. All linear variables are non-negative by definition. This is not a restriction, it just saves some of the writing.

3.2 Linearizing Terms

All terms occurring in a simple bitvector formula are simple terms, i.e. they involve just variables, constants and one operator. Therefore linearization of simple terms is sufficient. Each operator in eq. 1, fig. 2 is translated into LACs. A bitvector $a_{[x]}$ and a constant bitvector $c_{[x]}$ can be translated into a linear variable as follows:

$$\frac{a_{[x]}}{a \text{ with } a \leq 2^x - 1} \qquad \frac{c_{[x]}}{\mathbf{nat} \ c_{[x]}}$$

Conversion of negation of a bitvector is also fairly simple.

$$\frac{\neg a_{[x]}}{2^x - 1 - a \text{ with } a \leq 2^x - 1}$$

Addition of two bitvectors can be translated as follows:

$$\frac{a_{[x]} + b_{[x]}}{a + b - 2^x\sigma \text{ with } a + b - 2^x\sigma \leq 2^x - 1 \ \wedge \ \sigma \leq 1}$$

More general, for the sum of at least one bitvector ($n > 0$) it holds:

$$\frac{\sum_{i=1}^{n} a_{[x]_i}}{\sum_{i=1}^{n} a_i - 2^x\sigma \text{ with } \sum_{i=1}^{n} a_i - 2^x\sigma \leq 2^x - 1 \ \wedge \ \sigma \leq n-1}$$

Note that subsequent addition of i terms yields a different linearization result than summation. In the first case i sigma variables with a domain of $\{0, 1\}$ each are generated. Thus there are 2^i possible combinations of values. In the latter case only one sigma variable with a domain of $[0, \dots, i-1]$ is needed. This obviously makes a big difference for large i, and the search space for the ILP is much smaller for summation.

Multiplication with a constant bitvector c is just subsequent addition. For scalar multiplication with $c > 0$ the following rule is applicable:

$$\frac{ca_{[x]}}{ca - 2^x\sigma \text{ with } ca - 2^x\sigma \leq 2^x - 1 \ \wedge \ \sigma \leq c - 1}$$

For extraction, right and left shift the rules are rather complicated. For instance a shift to the right $a_{[x]} \ \mathbf{SR} \ z$ cannot be modeled as $2^{-z}a$, since division and fractions of integers are not part of ILP. Instead extraction and shift operations are modeled using quasi linear constraints for modulo and integer division. Left shift is modeled as a modulo operation followed by a multiplication with a power of 2:

$$\frac{a_{[x]} << z}{\begin{array}{c} k \text{ with } k = 2^z h \ \wedge \ h = a - 2^{x-z}\sigma \ \wedge k \leq 2^z(2^{x-z} - 1) \ \wedge \\ h \leq 2^{x-z} - 1 \ \wedge \ \sigma \leq 2^z - 1 \end{array}}$$

A shift to the right is a division by a power of 2.

$$\frac{a_{[x]} >> z}{k \text{ with } 0 \leq a - 2^z k \leq 2^z - 1 \ \wedge \ k \leq 2^{x-z} - 1}$$

The rule for concatenation of bitvectors is:

$$\frac{a_{[x]} \otimes b_{[y]}}{2^y a + b \text{ with } 2^y a + b \leq 2^{y+x} - 1}$$

The rule for extraction is:

$$\frac{a_{[x]}[i,j]}{k \text{ with } k \leq 2^{i-j+1}-1 \ \wedge\ a - 2^{i+1}\sigma \leq 2^{i+1}-1 \ \wedge \\ a - 2^{i+1}\sigma - 2^j k \leq 2^j - 1 \ \wedge\ \sigma \leq 2^{x-i-1}-1}$$

Sign extend is needed to extend the length of a bitvector with a 2's complement interpretation of it's value. The most significant bit (MSB) of $a_{[y]}$ is $(a \ \mathbf{div}\ 2^{y-1})$ and determines the value of the $(z-y)$ MSBs of $(a_{[y]}\ \mathbf{SE}\ z)$. The integer value representing them is $(2^z - 2^y)$. The value of $(a \ \mathbf{div}\ 2^{y-1})$ is either 0 or 1 and therefore $(2^z - 2^y)(a \ \mathbf{div}\ 2^{y-1})$ is either 0 or $(2^z - 2^y)$. Consequently sign extend is modeled as:

$$\frac{a_{[y]}\ \mathbf{SE}\ z}{(2^z-2^y)d + a \text{ with } 0 \leq a - 2^{y-1}d \leq 2^{y-1}-1 \ \wedge\ d \leq 1}$$

Since zero extend is just a syntactic feature, concatenation of a constant bitvector and a term could be used. However, it is useful to have a special rule for zero extend to save variables and constraints:

$$\frac{a_{[x]}\ \mathbf{ZE}\ z}{a \text{ with } a \leq 2^x - 1}$$

3.3 Linearizing Propositions

Bitvector terms on the left and right hand side of a proposition have the same length. Most of the propositions can easily be translated, since e.g. $a_{[n]} \leq b_{[n]}$ iff $\mathbf{nat}a_{[n]} \leq \mathbf{nat}b_{[n]}$. The same applies for '$\geq$','$<$' and '$>$'. Incorporating rules for those operators is therefore straightforward. For '$\neq$' things are a little bit more complicated, since it is not part of ILP. Although Omega test can handle negation, it is not desirable to keep it, since Omega test cannot deal with it very well [13]. Using the additive inverse[7], it follows that $\delta_{[n]} \neq 0_{[n]}$ iff $\delta_{[n]} \geq 1_{[n]}$. Therefore: $a_{[n]} \neq b_{[n]}$ iff $a_{[n]} = b_{[n]} +_{[n]} \delta_{[n]}$ with $\delta_{[n]} \geq 1_{[n]}$. New variables σ, δ are introduced and $a_{[n]} \neq b_{[n]}$ is modeled as:

$$\frac{a_{[n]} \neq b_{[n]}}{a = b + \delta - 2^n\sigma \text{ with } 1 \leq \delta \ \wedge\ \delta \leq 2^n - 1 \ \wedge\ \sigma \leq 1}$$

3.4 Linearization of Bitvector Formulas

Based on the results above, a whole bitvector formula can now be linearized recursively, such that sub-terms of a term are linearized before the term.A bitvector formula is satisfiable iff the corresponding ILP is satisfiable. A solution of the ILP can be translated into a solution of the bitvector formula using the auxiliary function vec_n.

To formally justify the correctness of the overall procedure, the linearization techniques given above can be formalized as conversion functions, which can not presented here. Roughly the arguments is as follows: A simple bitvector formula f is a conjunction of simple propositions p_i. The satisfiability of the formula f is preserved by a sound modeling of each proposition p_i. A bitvector proposition p_i is satisfiable iff it's ILP p'_i is satisfiable. The ILP f' of a simple bitvector formula f is the conjunction of the ILPs p'_i of propositions p_i, which are in turn conjunctions of ILP constraints.

Furthermore, if the ILP f' is satisfiable and ϕ' is a solution, an environment ϕ of $var(f)$ can be calculated such that it satisfies f, i.e. $[\![f]\!]^{\varphi} = \top$.

[7] It always exists for bitvector addition.

Table 1. Comparison of commercial SAT prover vs. ILP approach

Test case	Hybrid Boolean Prover			ILP (Omega)
	# Input Vars	# Gates	CPU time	CPU time
test_a	35	493	0.70	< 0.01
fir	65	1081	59.17	< 0.01
test_b	35	511	248.27	< 0.01
test_c	35	511	182.83	< 0.01
long5	17	169	0.02	< 0.01
long6	23	283	0.04	0.01
long7	29	397	0.17	< 0.01
long8	35	511	1.55	< 0.01
long9	41	625	4.54	0.01
long10	47	739	20.79	< 0.01
long32	179	3247	1162.39	0.01

Table 2. Comparison of Chaff and ILP approach

Problem	# SAT Vars	# Clauses	# Literals	Chaff	ILP
test_a	530	1480	3452	10.91	< 0.01
fir	1148	3244	–	Abort	< 0.01
long5	196	532	1769	0.78	< 0.01
long6	308	850	1982	11.78	0.01
long7	428	1192	2780	64.43	< 0.01
long8	548	1534	3577	767.07	< 0.01
long9	668	1876	–	Abort	0.01

To translate a formula, we have to linearize each of it's propositions. The order in which the propositions are linearized is irrelevant. If there is term sharing, it is sufficient to linearize each term once. The number of constraints generated for a formula can be calculated easily. Our conversion algorithm has a linear complexity in the number of propositions and terms.

4 Experimental Results

The method described above has been implemented in C++ and has been integrated into an industrial RTL bounded model checking framework (see fig. 5 for the overall flow) developed at the formal verification group at Siemens, Germany. For details on bounded model checking see [1]. All results in the following are measured on a SUN Ultra Sparc 60 and all run times are given in CPU seconds. As ILP solver the Omega test [12] has been used.

In a first series of experiments our technique based on ILP is compared to an industrial high-performance multi-engine SAT solver developed at Siemens, which automatically utilizes different state-of-the-art solvers, like ATPG, BDD and 3-SAT. Each of the test cases shown in the upper half of tab. 1 implements two versions of the same arithmetic function in Verilog-RTL. The property checks for their equivalence.

For example, 'fir' has two equivalent outputs, 't1' and 't2' which implement a shift/add function, commonly used in digital filters:

```
module fir(a,b,c,d,e,f);
   input [12:0] a;   input [12:0] b;
   input [12:0] c;   input [12:0] d;
   input [12:0] e;   input [12:0] f;
   wire [12:0]  t1;   wire [12:0]  t2;
   assign t1 = a + (b << 2'd1) + (c << 2'd2)
                 + (d << 3'd3) + (e << 3'd4)
                 + (f << 2'd2);
   assign t2 = a+4'd2*(b+4'd2*(c
                       + 4'd2*(d+3'd2*e)))
                  +4'd4*f;
endmodule
```

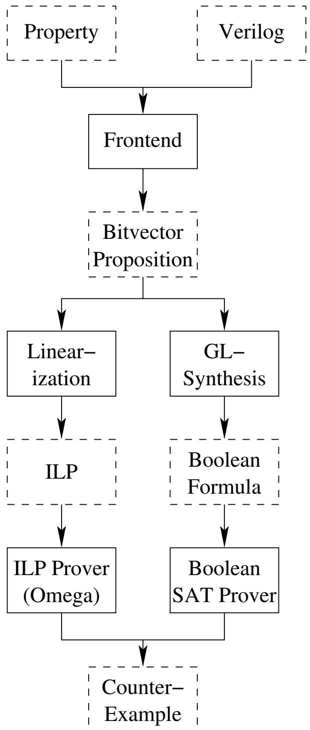

Figure 5. RTL Bounded-MC Framework

The examples differ in structure as well as in the arithmetic operators occurring. Although the examples appear to be small – counted in the number of gates – they are difficult to prove in the Boolean domain.
To show the scalability of approach, the examples 'long5' through 'long10' are the same design, scaled in the width of the variables from 5 to 10 bit. The limits of the Boolean packages were reached at 10 bit. The Omega test did not show any growth in run time until it failed at 33 bit[8] because of variable width limitations. As can easily be seen, the ILP solver obtains significant speed-ups compared to the high-performance multi-engine solver.
Finally, a comparison to a pure SAT solver is given. In the Boolean domain, Chaff [11] is by far the fastest available public domain SAT solver as a recent study has shown [16]. The results are given in tab. 2. As can be seen, in this case the ratio even becomes worse. For larger instances Chaff was not able to find the result within the given time and memory bounds.
In summary, a tremendous speed-up can be obtained by solving the verification problem on the word-level instead of the bit-level. While the bit-level problems turn out to be very hard, the ILP solver never took more than a hundredth of a second to terminate.

5 Conclusion

A method to check satisfiability of bitvector formulas has been presented. The bitvector operators have a modulo semantics as found in HDLs, like Verilog. Bitvector formulas are translated into the integer domain, such that counterexamples for the resulting ILP-SAT problem can directly be translated into counterexamples for the original problem. There are no false positives or false negatives. The procedure has been integrated into an industrial RTL bounded model checking framework. The Omega test is used as ILP solver. The experimental results show that using this technique, considerable performance gains over Boolean provers can be achieved.

[8] Due to the limitations in the word size of the ILP prover, we had to limit variable width to 16 to 32 bit. These limitations can be overcome using integers of arbitrary length, instead of machine data types, within Omega test or using techniques proposed by Fallah [6].

References

[1] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *Proceedings of DAC'99*, pages 317–320, 1999.
[2] J. R. L. Clark W. Barrett, David L. Dill. A decision procedure for bit-vector arithmetic. In *Proceedings of DAC'98*, pages 522–527, 1998.
[3] D. Cyrluk, H. Rueß, and O. Möller. An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors. In *Proceedings of 9th CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 60–71, 1997.
[4] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin Elimination and Its Dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.
[5] R. Drechsler. *Formal Verification of Circuits*. Kluwer Academic Publisher, 2000.
[6] F. Fallah. *Coverage Directed Validation of Hardware Models*. PhD thesis, MIT, 1999.
[7] F. Fallah, S. Devadas, and K. Keutzer. Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. In *Proceedings of 35th DAC-98*, pages 528–533, 1998.
[8] C.-Y. Huang and K.-T. Cheng. Assertion Checking by Combined Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques. In *Proceedings of DAC'00*, pages 118–123, 2000.
[9] P. Johannsen. BooStER: Speeding Up RTL Property Checking of Digital Designs by Word-Level Abstraction. In *Proceedings of CAV'01 Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 373–376, 2001.
[10] M. O. Möller and H. Rueß. Solving Bit-Vector Equations. In *Proceedings of 2nd FMCAD'98*, volume 1522 of *Lecture Notes in Computer Science*, pages 36–48, 1998.
[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC'01*, pages 530–535, 2001.
[12] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the ACM*, number 8, pages 102–114, 1992.
[13] W. Pugh and D. Wonnacott. Experiences with constraint-based array dependence analysis. Technical report, University of Maryland, College Park, MD, 1994.
[14] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1998.
[15] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Proceedings of 10th CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 280–292, 1998.
[16] M. N. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. In *Proceedings of DAC'01*, pages 226–231, 2001.
[17] P. Wolper and B. Boigelot. An Automata-Theoretic Approach to Presburger Arithmetic Constraints. In *Proceedings of the SAS'95*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32. Springer-Verlag, 1995.
[18] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: A Unified Approach to RTL Satisfiability. In *Proceedings of DATE'01 Conference*, pages 398–402, 2001.