An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors

(Extended Abstract, Submitted)

David Cyrluk*, Oliver Möller**, Harald Rueß**

* Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
cyrluk@csl.sri.com

** Fakultät für Informatik
Universität Ulm
D-89069 Ulm, Germany
{moeller,ruess}@ki.informatik.uni-ulm.de

Abstract. In this paper we describe a decision procedure for the core theory of fixed-sized bit-vectors with extraction and composition than can readily be integrated into Shostak's procedure for deciding combinations of theories. Inputs to the solver are unquantified bit-vector equations $t = u$ and the algorithm returns *true* if $t = u$ is valid in the bit-vector theory, *false* if $t = u$ is unsatisfiable, and a system of solved equations otherwise. The time complexity of the solver is $\mathcal{O}(\mid t \mid \cdot \log n + n^2)$, where t is the length of the bit-vector term t and n denotes the number of bits on either side of the equation. Then, the solver for the core bit-vector theory is extended to handle other bit-vector operations like bitwise logical operations, shifting, and arithmetic interpretations of bit-vectors. We develop a BDD-like data-structure called bit-vector BDDs to represent bit-vectors, various operations on bit-vectors, and a solver on bit-vector BDDs. The overall procedure has been integrated with the decision procedures of the PVS prover. The implementation has been tested with typical lemmas from the domain of microprocessor verification. The implementation has also been applied to proofs found in the verification of a commercial microprocessor. By using our decision procedure for bit-vectors we have simplified a number of proofs by eliminating manual proof steps that were previously necessary for reasoning about bit-vectors.

1 Introduction

The advantage of using a theorem prover to verify the correctness of large hardware circuits such as microprocessors is that the user can intelligently decompose and guide the high-level verification task. However, in order to be effective, low-level verification needs to be as automatic as possible. In the PVS verification system this is accomplished through the use of a method due to Shostak [2,7] for combining decision procedures. Currently any proof goal that can be proven by reasoning about equality, arrays, tuples, and linear arithmetic in PVS is proven automatically.

Experience with the verification of a commercial microprocessor [8], and the verification of multipliers [6] has shown that the lack of specialized decision procedures for notions related to bit-vectors is the main impediment to effective automation in theorem proving systems like PVS. This insight forms the starting point of this paper, and we develop an efficient decision procedure for a theory of fixed-sized bit-vectors. Moreover, this decision procedure can readily be incorporated into Shostak's procedure for combinations of theories [7], since our algorithm fulfills the requirements for component theories as stated in [2].

By way of introduction, consider the following true statement in the combined theory of equality and bit-vectors: $(u_{[m]} \otimes v_{[n]})\mathrm{XOR}(\mathrm{x}_{[\mathrm{m}]} \otimes \mathrm{y}_{[\mathrm{n}]}) = 0_{[\mathrm{m+n}]} \supset \mathrm{f}(\mathrm{x}_{[\mathrm{m}]}, \mathrm{y}_{[\mathrm{n}]}) = \mathrm{f}(\mathrm{u}_{[\mathrm{m}]}, \mathrm{v}_{[\mathrm{n}]})$. To prove this by hand we would have to state and introduce several lemmas. One of which is that when the XOR of two bit-vectors is 0 then the two bit-vectors are identical. The user would have to manually apply these lemmas to the hypothesis in our example to conclude that $u_{[m]} = x_{[m]}$ and $v_{[n]} = y_{[n]}$. Once this has been established equality reasoning shows that $f(x_{[m]}, y_{[n]}) = f(u_{[m]}, v_{[n]})$

This example is illustrative of the type of unnecessary reasoning that took place in [8]. The algorithm that we present below takes equations such as the one that appears in the hypothesis in the above example and *solves* for some of the variables in that equation in terms of the remaining variables. In this case the solver would return ($u_{[m]} = x_{[m]}$ and $v_{[n]} = y_{[n]}$. The rest of Shostak's algorithm works roughly by using this *solution* to replace the solved variables with their solved form. We have successfully applied this solver to eliminate manual reasoning about bit-vectors in some of the proofs in [8].

The paper is organized as follows. In Section 2 we present the theory of fixed-sized bit-vectors with composition and extraction as a many-sorted conditional equational theory. Section 3 contains a description of a canonizer and a solver [7] for this bit-vector together with an analysis of their time complexities. In Section 4, we describe how to add boolean bitwise operations to the core solver by a special data-structure called bit-vector BDDs, and in Section 5 we report on some preliminary experiment with an implementation of the bit-vector solver within the PVS system. The paper closes with some final remarks in Section 6.

2 Core Theory of Bit-Vectors

In this section we describe the core equational theory of fixed-sized bit-vectors of length n with composition and extraction of one or several consecutive bits. The length n of bit-vectors is constrained to be a positive natural number, since bit-vectors of length 0 are not permitted, and the bits of a bit-vector of length n are indexed, from left to right, from $n-1$ down to 0. In the following, $n, m, k, \ldots$ denote valid lengths of bit-vectors. The bit-vector theory contains constant bit-vectors $0_{[n]}$ and $1_{[n]}$ of length n, *composition* $t \otimes u$ of bit-vectors t and u, and *extraction* $t^\wedge(i,j)$, where $i, j \in \mathbb{N}$, of $i-j+1$ many bits i through j from bit-vector t. These considerations lead to a many-sorted signature with infinitely many sort symbols $bvec_n$, $n \in \mathbb{N}^+$.

Definition 1. Let Σ be the signature

$$\begin{aligned}&\langle \{bvec_n \mid n \in \mathbb{N}^+\},\\ &\ \{0_{[n]} \mid n \in \mathbb{N}^+\} \cup \{1_{[n]} \mid n \in \mathbb{N}^+\} \cup\\ &\ \{.\ \otimes_{n,m}\ . \mid n, m \in \mathbb{N}^+\} \cup \{.\ ^\wedge{}_n(i,j) \mid n \in \mathbb{N}^+ \wedge i, j \in \mathbb{N} \wedge n > i \geq j \geq 0\}\rangle\end{aligned}$$

such that for appropriate n, i, and j: $0_{[n]} :\to bvec_n$, $1_{[n]} :\to bvec_n$, $.\ \otimes_{n,m}\ . : bvec_n \times bvec_m \to bvec_{n+m}$, and $.\ ^\wedge{}_n(i,j) : bvec_n \to bvec_{i-j+1}$

The dots to the left and to the right of function symbols indicate the use of infix notation, and extraction $^\wedge{}_n(i,j)$ is assumed to bind stronger than composition $\otimes_{n,m}$. In the following, $x_{[n]}, y_{[m]}, z_{[k]}, \ldots$ denote variables of sort $bvec_n$, $bvec_m$, and $bvec_k$ respectively. The set of well-formed terms is defined in the usual way and $t_{[n]}, u_{[m]}, v_{[k]}, \ldots$ denote bit-vector terms of respective lengths. Subscripts are omitted whenever possible and can be inferred from the context. Moreover, $t \equiv u$ denotes syntactic equality of bit-vectors t and u, $vars(t)$ denotes the set of variables in t.

A bit-vector term t is called *atomic* if it is a variable or a constant $0_{[n]}$ or $1_{[n]}$, and *simple terms* are either atomic or of the form $x_{[n]}{}^\wedge(i,j)$ where $x_{[n]}$ is a variable, and $i \neq n-1$ or $j \neq 0$. Moreover, terms of the form $t_1 \otimes t_2 \otimes \ldots \otimes t_k$ (modulo associativity), where t_i are all *simple*, are referred to as being in *composition normal form*. If, in addition, none of the neighboring simple terms denote the same constant (modulo length) and a simple term of the form $t^\wedge(i,j)$ is not followed by a simple term of the form $t^\wedge(j-1,k)$, then a term in composition normal form is called *maximally connected*.

Definition 2. Let Σ be the bit-vector signature defined in Definition 1; the core theory of bit-vectors is defined by the (conditional) Σ-equalities in Figure 1, and semantic entailment $\models$ in this theory is defined in the usual way.

Note that well-formedness of the bit-vector terms in Figure 1 implies that $n > i \geq j > k \geq 0$ in equation *5)* and $n > i \geq j \geq 0 \wedge i-j \geq k \geq l \geq 0$ in equation *7)* above. Obviously, the bit-vector theory in Definition 2 is consistent, and a possible interpretation of fixed-sized bit-vectors of length n are finite functions with domain $[..n)$ and codomain $\{0, 1\}$.

1)	$(t_{[n]} \otimes u_{[m]})\hat{}(i,j) = u_{[m]}\hat{}(i,j)$	IF $m > i \geq j \geq 0$
2)	$(t_{[n]} \otimes u_{[m]})\hat{}(i,j) = t_{[n]}\hat{}(i-m, j-m)$	IF $m+n > i \geq j \geq m$
3)	$(t_{[n]} \otimes u_{[m]})\hat{}(i,j) = t_{[n]}\hat{}(i-m, 0) \otimes u_{[m]}\hat{}(m-1, j)$	IF $m+n > i \geq m > j$
4)	$t_{[n]}\hat{}(n-1, 0) = t_{[n]}$	
5)	$t_{[n]}\hat{}(i,j) \otimes t_{[n]}\hat{}(j-1, k) = t_{[n]}\hat{}(i,k)$	
6)	$(t_{[n]} \otimes u_{[m]}) \otimes v_{[p]} = t_{[n]} \otimes (u_{[m]} \otimes v_{[p]})$	
7)	$t_{[n]}\hat{}(i,j)\hat{}(k,l) = t_{[n]}\hat{}(k+j, l+j)$	

Fig. 1. Bit-Vector Equations

3 Solving Bit-Vector Equations

Now, we describe a decision procedure for the bit-vector theory as introduced above. This decision procedure can readily be integrated with Shostak's framework for deciding combinations of theories, since it is subdivided into a *canonizer* and a *solver* [7] which satisfy the requirements stated in [2].

3.1 Canonizer

The canonizer $\sigma(t)$ in Appendix A computes the maximally connected composition normal form of t and is a straightforward transliteration of the equalities in Figure 1. In the first phase $\alpha(t)$ this canonizer normalizes a bit-vector term t to an equivalent term in composition normal form (see Section 2). The resulting composition normal form may still contain sub-terms such as $c_{[n]} \otimes c_{[m]}$ or $x_{[2]}\hat{}(1,1) \otimes x_{[2]}\hat{}(0,0)$, which can be further normalized to $c_{[n+m]}$ and $x_{[2]}$ respectively. These kinds of merging are accomplished in the second phase of canonization by the function β (see Appendix A). Altogether, $\sigma(t) ::= \beta(\alpha(t))$ computes the maximally connected composition normal form for a bit-vector term t.

Using this result one can prove that σ fulfills the requirements given in [2] for a canonizer in Shostak's framework.

Theorem 3.

1) An equation $t = u$ in the theory is valid if and only if $\sigma(t) \equiv \sigma(u)$.
2) If t is a term not in the theory, then $\sigma(t) \equiv t$
3) $\sigma(\sigma(t)) \equiv \sigma(t)$
4) If $\sigma(t) \equiv f(t_1, ..., t_n)$ for a term t in the theory then $\sigma(t_i) \equiv t_i$ for $1 \leq i \leq n$.
5) $vars(\sigma(t)) \subseteq vars(t)$.

The only non-trivial part of the proof is to show that $t = u$ implies $\sigma(t) \equiv \sigma(u)$. In order to prove this let $t = u$ be a valid equation and presume $\sigma(t) \not\equiv \sigma(u)$. This inequality is of the form $t_1 \otimes t_2 \otimes \ldots \otimes t_n \not\equiv u_1 \otimes u_2 \otimes \ldots \otimes u_m$ where t_i, u_j are simple terms; furthermore, let n_i and m_j respectively denote the lengths of t_i and u_j. Since $\sigma(t) \not\equiv \sigma(u)$, there is a least index i_0 such that $t_{i_0} \not\equiv u_{i_0}$. Proceed by case analysis. First, in case $n_{i_0} = m_{i_0}$, one can easily show that either $t_{i_0} \equiv u_{i_0} \equiv c$ for some same constant or t_{i_0} and u_{i_0} are extractions of the same part from the same variable; this contradicts the assumption $t_{i_0} \not\equiv u_{i_0}$. Second, in case $n_{i_0} \neq m_{i_0}$, we assume without loss of generality $n_{i_0} < m_{i_0}$. Then, t_{i_0+1} can not be combined with t_{i_0}, since $\sigma(t)$ is maximally connected. If u_{i_0} is a constant, then either t_{i_0} or t_{i_0+1} denotes something different. If u_{i_0} is a variable or an extraction term, $t_{i_0} \otimes t_{i_0+1}$ can not build up a prefix of it (for they can not be combined). This yields a contradiction and finishes the proof.

Given a proper data-structure, say abstract syntax trees of bit-vector terms, the function α (Appendix A) visits each subterm a constant number of times, since the topmost bit-vector operator is eliminated in each step. The case analysis takes at most $\mathcal{O}(log\, n)$ time, since it involves only comparison of integers which can be coded with $log\, n$ bits. Thus, α is called at most $\mathcal{O}(\mid t \mid)$ times, and each call takes $\mathcal{O}(log\, n)$ time. In phase β the number of steps equals the number of simple terms; an upper bound of this number is $\mathcal{O}(\mid t \mid)$, and,

```
solve(t = u)  ::=
    t ← σ(t)
    u ← σ(u)
    IF t ≡ u THEN RETURN true ENDIF
    IF vars(t) = ∅ THEN swap(t, u) ENDIF
    IF vars(t) = ∅ THEN RETURN false ENDIF
    {t_1 = u_1, t_2 = u_2, ..., t_m = u_m} ← slice({t, u})
    E ← ⋃_{i=1}^{m} csolve(t_i = u_i)
    IF false ∈ E THEN RETURN false ENDIF
    E_{x_1} ⊎ E_{x_2} ⊎ ... ⊎ E_{x_p} ← E
        /* t.i. {E_{x_i}} is a partition of E where E_{x_i} contains all equations x_i = ... */
    FOREACH i ∈ {1, ..., p} DO E_{x_i} ← slice(E_{x_i}) OD
    FOREACH i ∈ {1, ..., p} DO lazy_constant_propagation(E_{x_i}) OD
    {E_{x_1}, E_{x_2}, ..., E_{x_p}} ← coarsest_slicing({E_{x_1}, E_{x_2}, ..., E_{x_p}})
    FOREACH i ∈ {1, ..., p} DO equality_propagation(E_{x_i}) OD
    RETURN ⋀_{i=1}^{p} (x_i = β(find(E_{x_i})) )
      / * 'find' meaning a composition of representants of each column in E_{x_i} * /
```

Fig. 2. Pseudo-Code for Bit-Vector Solver

as above, each test is performed in $\mathcal{O}(log\, n)$ time. Altogether, this gives the following worst-case complexity for the canonizer in Appendix A.

Theorem 4. *The time complexity of canonization $\sigma(t_{[n]})$ is $\mathcal{O}(|\ t\ | \cdot log\, n)$, where t is the length of the bit-vector t.*

3.2 Solver

A *solver* rewrites any unquantified equality $t = u$ in an equivalent *solved* form $\wedge_i\ x_i = s_i$, where each x_i occurs in none of the s_i. A particular simple approach for solving equations $t = u$ over fixed-sized bit-vectors proceeds by (see [3] for details)

1. replacing any bit-vector variable $x_{[n]}$ with $x_{n-1} \otimes \ldots \otimes x_0$, where x_i are (fresh) variables of sort $bvec_1$,
2. computing the composition normal form of each side,
3. bitwise comparing the corresponding left-hand and right-hand sides of the equations,
4. propagating the resulting equalities by processing the bitwise equalities one-by-one and building up a union-find structure,
5. and finally, replacing the bit-variables with their canonical representatives.

This simple solver, however, can be improved considerably, since, in most cases, it is not necessary to reduce the problem to a bitwise comparison of t and u. In the sequel, we describe the basic ideas of a refined version of the brute force algorithm above by means of an example; pseudocode for the crucial parts of the algorithm can be found in Figure 2.

Example 5.

$$\underbrace{x_{[3]} \otimes (y_{[4]} \otimes z_{[5]})\hat{}(5,4) \otimes 0_{[4]}}_{t} = \underbrace{((y_{[4]}\hat{}(2,1) \otimes x_{[3]})\hat{}(4,0) \otimes z_{[5]} \otimes x_{[3]})\hat{}(10,2)}_{u}$$

Given an equation $t = u$, the bit-vector solver in Figure 2 first canonizes both sides of the equation to obtain the equation $\sigma(t) = \sigma(u)$ over the maximally connected composition normal forms $\sigma(t)$ and $\sigma(u)$. The equation in Example 5, for example, canonizes to

$$\underbrace{x_{[3]} \otimes y_{[4]}\hat{\ }(0,0) \otimes z_{[5]}\hat{\ }(4,4) \otimes 0_{[4]}}_{\sigma(t)} = \underbrace{x_{[3]} \otimes z_{[5]} \otimes x_{[3]}\hat{\ }(2,2)}_{\sigma(u)}$$

The next step of the algorithm, called *slicing*, computes composition normal forms $t_1 \otimes \ldots \otimes t_m$ and $u_1 \otimes \ldots \otimes u_m$ of $\sigma(u)$ and $\sigma(t)$ respectively, such that each t_i and u_i are of the same length; moreover, it does so by minimizing the number m, called *granulation*, of simple terms on each side. Slicing of the canonized equation above, for example, leads to the following equation.

$$\begin{array}{rcccccccccc} & \underbrace{x_{[3]}}_{t_1} & \otimes & \underbrace{y_{[4]}\hat{\ }(0,0)}_{t_2} & \otimes & \underbrace{z_{[5]}\hat{\ }(4,4)}_{t_3} & \otimes & \underbrace{0_{[3]}}_{t_4} & \otimes & \underbrace{0_{[1]}}_{t_5} \\ = & \underbrace{x_{[3]}}_{u_1} & \otimes & \underbrace{z_{[5]}\hat{\ }(4,4)}_{u_2} & \otimes & \underbrace{z_{[5]}\hat{\ }(3,3)}_{u_3} & \otimes & \underbrace{z_{[5]}\hat{\ }(2,0)}_{u_4} & \otimes & \underbrace{x_{[3]}\hat{\ }(2,2)}_{u_5} \end{array}$$

Obviously, this equation holds if and only if the conjunction of the equations in the set E holds.

$$\begin{array}{rcl} E & ::= & \{x_{[3]} = x_{[3]},\ y_{[4]}\hat{\ }(0,0) = z_{[5]}\hat{\ }(4,4),\ z_{[5]}\hat{\ }(4,4) = z_{[5]}\hat{\ }(3,3), \\ & & \ 0_{[3]} = z_{[5]}\hat{\ }(2,0), \\ & & \ 0_{[1]} = x_{[3]}\hat{\ }(2,2)\} \end{array}$$

Since E contains only equations over simple terms, the problem of solving an equation over arbitrary terms is reduced to solving equations over simple terms by means of the function *csolve* in Appendix B. In our running example, solving equations over simple terms yields the following new equations (possibly containing fresh variables).

$$\begin{array}{rcl} \mathit{csolve}(x_{[3]} = x_{[3]}) & = & \varnothing \\ \mathit{csolve}(y_{[4]}\hat{\ }(0,0) = z_{[5]}(4,4)) & = & \{y = a^{(1)}{}_{[3]} \otimes c^{(1)}{}_{[1]}, z = c^{(1)}{}_{[1]} \otimes a^{(2)}{}_{[4]}\} \\ \mathit{csolve}(z_{[5]}\hat{\ }(4,4) = z_{[5]}(3,3)) & = & \{z = b^{(1)}{}_{[1]} \otimes b^{(1)}{}_{[1]} \otimes a^{(3)}{}_{[3]}\} \\ \mathit{csolve}(0_{[3]} = z_{[5]}\hat{\ }(2,0)) & = & \{z = a^{(4)}{}_{[2]} \otimes 0_{[3]}\} \\ \mathit{csolve}(0_{[1]} = x_{[3]}\hat{\ }(2,2)) & = & \{x = 0_{[1]} \otimes a^{(5)}{}_{[2]}\} \end{array}$$

Thus, a call $\mathit{csolve}(t = u)$ results in a set of equations which

- is empty, if $t \equiv u$,
- contains **false** , if and only if t and u are different constants,
- contains exactly one solved equation of the form $x = s$, with $x \in \mathit{vars}(t = u)$ and s is a simple term (this case occurs whenever x is the only variable in $t = u$), or
- contains exactly two solved equations of the form $x = s_1$, $y = s_2$, with $x, y \in \mathit{vars}(t = u)$ and s_1, s_2.

Solving equations of the form $x\hat{\ }(.,.) = x\hat{\ }(.,.)$ is the only non-trivial case. In this case one has to distinguish between cases where the extracted parts on each side of the equation do or do not overlap. Note that *csolve* introduces fresh variables with the convention that a-variables are known to occur only once in the system of solved equations, b-variables occur at least twice in a single right-hand side, and c-variables occur in exactly two different right-hand sides. This information is used to perform the following steps more efficiently.

In order to perform the next steps it is convenient to rearrange the sets of equations obtained by *csolve* and group together the solved equations for variable x in a so-called *block* E_x. In our running example we get the following three blocks.

$$E_x = \left| \begin{array}{c|c} 0_{[1]} & a^{(5)}{}_{[2]} \end{array} \right| \qquad E_y = \left| \begin{array}{c|c} a^{(1)}{}_{[3]} & c^{(1)}{}_{[1]} \end{array} \right| \qquad E_z = \left| \begin{array}{c|c|c} c^{(1)}{}_{[1]} & a^{(2)\prime}{}_{[1]} & a^{(2)\prime\prime}{}_{[3]} \\ b^{(1)}{}_{[1]} & b^{(1)}{}_{[1]} & a^{(3)}{}_{[3]} \\ a^{(4)\prime}{}_{[1]} & a^{(4)\prime\prime}{}_{[1]} & 0_{[3]} \end{array} \right|$$

In this example, the constant $0_{[3]}$ in the last column of E_z may be propagated by equalizing both ${a^{(4)}}'_{[1]}$ and ${a^{(4)}}''_{[1]}$ with $0_{[3]}$. No further propagation is necessary, since both variables are of kind-a variables. In the general case, however, propagation of constants in one column may trigger further propagations in other columns. Moreover, propagation of constants may result in additional slicings, since block entries may well be compositions. While propagation of constants is an optional step in our algorithm it may be used to detect inconsistencies — i.e. different constants in one column — at the earliest possible stage.

A *coarsest slicing* is a transformation of a set of equations of the form $x = t$, where t is in composition normal form, such that the cross-references between the terms in composition normal form on the right hand sides are resolved. More precisely, if a fresh variable c of kind C is split up into several parts $c', c'', \ldots$ in one equation (and possibly into parts $\hat{c}, \hat{\hat{c}}, \ldots$ in another one) these split-ups are sliced with each other, thus increasing the number of splinters of c, but computing what is the coarsest granulation possible at this point of the propagation.

In our example, this operation leaves the blocks untouched; given the blocks $E_v = \mid\ c' \ \mid\ c'' \ \mid\ a\ \mid$ and $E_w = \mid\ c\ \mid$, however, E_w gets updated to $\mid\ c'\ \mid\ c''\ \mid$.

Finally, the *propagation of equalities* step transforms all blocks to the coarsest slicing, so all references between them can be made explicit. The principle thereto is very much the same as in the brute force solver above. However, it is applied on (hopefully) vast parts of the variables instead of tiny bits. Note also that it is not necessary to check the consistency with the constants, since any such conflict has already been detected in the *propagation of constants* step.

Applying *propagation of constants*, *coarsest slicing*, and *propagation of equalities* to the equalities of our running examples we obtain the following solved form for the equation in Example 5 :

$$
\begin{aligned}
solve(E) \quad = \quad \{\ x_{[3]} &= 0_{[1]} \otimes {a^{(5)}}_{[2]}, \\
y_{[4]} &= {a^{(1)}}_{[3]} \otimes {c^{(1)}}_{[1]}, \\
z_{[5]} &= {b^{(1)}}_{[1]} \otimes {b^{(1)}}_{[1]} \otimes 0_{[3]} \quad \}
\end{aligned}
$$

Altogether, it can be shown that *solve* in Figure 2 is indeed a correct and complete solver for the given bit-vector theory.

Theorem 6. $\vdash t = u$ *if and only if* $solve(t = u) = \mathbf{true}$.

Moreover, the bit-vector decision procedure *solve* can be readily used in Shostak's framework for deciding combinations of theories, since it fulfills, besides Theorem 6, Shostak's requirements for individual solvers as stated in [2].

Theorem 7. *Let* $E \ ::= \ solve(e)$*; then:*

a) $E \in \{\mathbf{true}\ , \mathbf{false}\ \}$ *or* $E \equiv \bigwedge_i (x_i = t_i)$.
b) If $vars(e) = \varnothing$ *then* $E \in \{\mathbf{true}\ , \mathbf{false}\ \}$.
c) If $E \equiv \bigwedge_i (x_i = t_i)$ *then the following holds:*

1. $x_i \in \mathrm{vars}(e)$
2. *for all* i, j*:* $\ x_i \notin \mathrm{vars}(t_j)$
3. *for all* $i \neq j$*:* $\ x_i \neq x_j$
4. *for all* i*:* $\ \sigma(t_i) = t_i$.

Finally, we analyze the time complexity of the solver in Figure 2 in terms of the maximum $\mid t \mid$ of the lengths of t and u, and the number n of bits of the bit-vectors on either side of the equation $t = u$. Obviously, *slicing* can be computed in linear time and each call to *csolve* takes at most logarithmic time. It is a bit tricky, however, to determine the complexity of the propagations; but in a worst-case estimation one can say that: First, *lazy_constant_propagation* results either in a speed-up or "wastes" only linear time by unsuccessfully searching for constants to propagate. Second, slicing can be done in linear time by introducing, for example, a boolean vector of length n for each variable to denote the positions where splits have been introduced while processing the *coarsest slicing* routine. Third, propagation of equalities between the simple terms takes at most $\mathcal{O}(n^2)$ time, since there are at most $\mathcal{O}(n)$ such simple terms and equality within columns is propagated by browsing each column and computing canonical representatives for each entry; using specialized *union-find* structures this takes linear time. This analysis, together with the complexity of the canonizer (see Theorem 4) yields the following result.

Theorem 8. *The time complexity of* $solve(t_{[n]} = u_{[n]})$ *is* $\mathcal{O}(\mid t \mid \cdot \log n + n^2)$.

4 Bit-Vector BDDs

To this point we have described a solver for the *core* theory of bit-vectors with only the operations of composition and extraction. We now describe how to add the boolean bitwise operations, and call this new theory the *extended* theory of bit-vectors.

The two basic requirements that we must satisfy when adding the boolean operations is canonicity and solvability. As will be seen *binary decision diagrams* (BDDs) [1] over bit-vectors satisfy both these criteria.

A bit-vector BDD of size n is a BDD with bit-vector variables of size n as the internal nodes and the constant bit-vectors $1_{[n]}$ and $0_{[n]}$ as the terminals. The intended meaning of such a bit-vector BDD is the conjunction of the constraints that the n BDDs impose on the n individual bits of the bit-vector variables. The meaning, for example, of the bit-vector BDD,

$$\mathbf{ite}(x_{[3]},\, \mathbf{ite}(y_{[3]},\, 1_{[3]},\, 0_{[3]}),\, 0_{[3]}) \tag{1}$$

where $\mathbf{ite}(.,\, .,\, .)$ is the common *if–then–else* conditional, is given by:

$$\begin{array}{l}\mathbf{ite}(x_{[3]}{}^\wedge(2,2),\, \mathbf{ite}(y_{[3]}{}^\wedge(2,2),\, 1_{[1]},\, 0_{[1]}),\, 0_{[1]})\wedge\\ \mathbf{ite}(x_{[3]}{}^\wedge(1,1),\, \mathbf{ite}(y_{[3]}{}^\wedge(1,1),\, 1_{[1]},\, 0_{[1]}),\, 0_{[1]})\wedge\\ \mathbf{ite}(x_{[3]}{}^\wedge(0,0),\, \mathbf{ite}(y_{[3]}{}^\wedge(0,0),\, 1_{[1]},\, 0_{[1]}),\, 0_{[1]})\end{array}$$

Now, consider the additional constraint $x_{[3]}{}^\wedge(0,0) = 1_{[1]}$. In this case, the example bit-vector BDD in (1) specializes to

$$\mathbf{ite}(x_{[3]}{}^\wedge(2,2),\, \mathbf{ite}(y_{[3]}{}^\wedge(2,2),\, 1_{[1]},\, 0_{[1]}),\, 0_{[1]}) \otimes \mathbf{ite}(y_{[3]}{}^\wedge(0,0),\, 1_{[1]},\, 0_{[0]})$$

Thus, the use of bit-vector-BDDs permits maintaining the paradigm of largest chunks possible that has already guided the development of the efficient solver for the core theory.

The canonicity of BDDs immediately provides a canonical form for bit-vector BDDs. Extractions and composition distribute over bit-vector BDDs so the normal form for the core theory can still be used for extended bit-vectors. Thus, the new normal form is composition of bit-vector BDDs whose variables are either bit-vector variables or extractions of bit-vector variables; in the following we assume given a function γ that computes this new normal form.

Bit-wise operations over bit-vectors are represented by canonical bit-vector BDDs. For example, bit-wise conjunction "*AND*" and right-shift "*RSH*" are represented in this extended theory using the following correspondences.

$$\begin{array}{r}BDD^1{}_{[n]}\ AND\ BDD^2{}_{[n]} = \gamma(\mathbf{ite}(BDD^1{}_{[n]},\, \mathbf{ite}(BDD^2{}_{[n]},\, 1_{[n]},\, 0_{[n]}),\, 0_{[n]}))\\ RSH(BDD_{[n]}) = \gamma(BDD_{[n]}{}^\wedge(0,0)) \otimes \gamma(BDD_{[n]}{}^\wedge(n-1,1))\end{array}$$

Now we have collected all the ingredients to describe a solver on bit-vector BDDs. Consider the BDD B:

$$\mathbf{ite}(P, B_P, B_{\bar{P}}). \tag{2}$$

where B_P and $B_{\bar{P}}$ respectively are the positive and negative cofactors of B with respect to variable P. We now describe a procedure for solving Equation 2 for the propositional variable P in terms of the variables in the rest of the BDD. The procedure will if necessary successively solve for the remaining variables lower in the BDD variable ordering. Equation 2 can be rewritten as follows:

$$(\neg(P \wedge \neg B_P) \wedge \neg(\neg P \wedge \neg B_{\bar{P}})) \wedge (B_P \vee B_{\bar{P}}). \tag{3}$$

This is equivalent to:

$$\big(P = \mathbf{ite}(B_P, \mathbf{ite}(B_{\bar{P}}, \delta, \mathbf{true}), \mathbf{false})\big) \wedge \tag{4}$$

$$(B_P \vee B_{\bar{P}}), \tag{5}$$

where δ is a newly generated variable that indicates that, when both B_P and $B_{\bar{P}}$ are true, BDD 2 imposes no constraint on the truth value of P. Equation 4 gives a solution for P in terms of variables lower in the BDD variable ordering. Equation 5 does not contain P and can be recursively solved for variables lower in the ordering than P. By successively solving the Equations 5 that are generated, a triangular system of equations is produced. By back substitution we then generate a completely solved system.

Consider, for example, solving the following propositional equation.

$$p \wedge q = \neg p. \tag{6}$$

We first generate the equivalent BDD, $\mathbf{ite}(p, \mathbf{ite}(q, \mathbf{false}, \mathbf{true}), \mathbf{false})$, and then solve for p and then q. Using our procedure we generate the equation for p:

$$p = \mathbf{ite}(\mathbf{ite}(q, \mathbf{false}, \mathbf{true}), \mathbf{ite}(\mathbf{false}, \delta, \mathbf{true}), \mathbf{false}) \tag{7}$$

which simplifies to

$$p = \mathbf{ite}(q, \mathbf{false}, \mathbf{true}). \tag{8}$$

The procedure also generates the following constraint on q:

$$\mathbf{ite}(q, \mathbf{false}, \mathbf{true}). \tag{9}$$

This constraint is then recursively solved to produce:

$$q = \mathbf{false} \tag{10}$$

Back substituting this solution for q into Equation (8) gives us the solution for p:

$$p = \mathbf{true}. \tag{11}$$

Equations 11 and 10 are our final solved form.

Given this solver on bit-vector BDDs, the core solver developed in Section 3 can easily be extended by, first, enabling the function *csolve* to accept bit-vector BDD arguments and, second, modifying the propagation of equalities step. While the complexity of solving the core theory of bit-vectors was analyzed above to be polynomial, the bit-vector theories containing bit-wise operations are *NP*- and *coNP*-hard, and, therefore, are not expected to be solvable in polynomial time. However, in the processor verification examples that we have looked at, the amount of bit-wise manipulations are quite limited and we do not expect the manipulations described in this section to dominate the complete solver.

5 Experiments

The bit-vector decision procedures described above have been implemented and integrated with the decision procedures of the PVS [5] proof system. Most of the examples we have dealt with so far have been extracted from the verification of the AAMP5 [8], an industrial-strength microprocessor. Figure 3 lists a collection of representative lemmas automatically proven using our bit-vector decision procedures together with the run-times (im milli-seconds) of the bit-vector decision procedures. Note that these timings do not include the preprocessing step of PVS formulas and the run-time of the other decision procedures.

In addition to the above lemmas we redid some proofs that had previously required manual reasoning. In these proofs we were able to eliminate the manual proof steps. We were also able to turn off much of the bit-vector rewriting that took place as that was replaced with the more efficient bit-vector decision procedures.

#	Lemma	Run-Time (in *msec*)
1.	$fill_{[32]}(0)\ XOR\ x_{[32]}\ =\ x_{[32]}$	7
2.	$(NOT(x_{[32]}))^\wedge(15,0)\ =\ NOT(x_{[32]}{}^\wedge(15,0))$	14
3.	$(x_{[32]}\ XOR\ y_{[32]})^\wedge(15,0)\ =\ x_{[32]}{}^\wedge(15,0)\ XOR\ y_{[32]}{}^\wedge(15,0)$	26
4.	$(fill_{[12]}{}^\wedge(0) \otimes nat2bv_{[4]}(1))\ =\ nat2bv_{[16]}(1)$	1
5.	$(x_{[32]}{}^\wedge(31,16)^\wedge(7,0) \otimes x_{[32]}{}^\wedge(15,0))\ =\ x_{[32]}{}^\wedge(23,0)$	2
6.	$((fill_{[16]}(nat2bv_{[10]}(192)^\wedge(9)) \otimes fill_{[8]}(nat2bv_{[10]}(192)^\wedge(8))) \otimes nat2bv_{[10]}(192)^\wedge(7,0)$ $OR\ fill_{[16]}(x_{[16]}{}^\wedge 15) \otimes x_{[16]})^\wedge(7,0)$ $=\ fill_{[2]}(1) \otimes x_{[16]}{}^\wedge(5,0)$	65
7.	$(fill_{[16]}(nat2bv_{[10]}(511)^\wedge(9)) \otimes fill_{[8]}(nat2bv_{[10]}(511)^\wedge(8)) \otimes nat2bv_{[10]}(511)^\wedge(7,0))$ $=\ (fill_{[16]}(0) \otimes fill_{[16]}(1))$	22
8.	$(bv2nat(fill_{[16]}(x_{[16]}{}^\wedge(15)) \otimes x_{[16]})\ =\ 0) \Leftrightarrow (bv2nat(x_{[16]}) = 0)$	3500

Fig. 3. Bit-Vector Lemmas

In order to apply our decision procedures to a larger number of lemmas from the AAMP5 verification, we had to make some extensions to support further bit-vector operations. The operation $nat2bv_{[n]}(m)$ is used in the lemmas in Figure 3 to generate a bit-vector of length n with unsigned interpretation m, $bv2nat$ is an interpreted function that computes the unsigned interpretation for bit-vectors, $fill_{[n]}(b)$ is a bit-vector of length n containing the bit b at every position, and $t^\wedge(i)$ extracts the i-th bit from bit-vector t. The extensions of the solver to support these bit-vector operations are all straightforward. Equations of the form $bv2nat(x_{[n]})\ =\ bv2nat(y_{[m]})$, for example, can be solved by padding 0's to the left of the shorter bit-vector argument until the lengths of x and y are equal and by solving the bit-vector equation over the resulting arguments.

Most of the examples we have tried so far have been proven automatically in a fraction of a second by the bit-vector decision procedures. Even more interestingly, the run-time performance of the decision procedures is in many cases *independent* of the width of the data-paths. The processing of Lemma 3 in Figure 3, for example, results in bit-vector BDDS with variables of the form $z_{[32]}{}^\wedge(15,0)$ and no further splits are necessary.

6 Conclusions

The main achievement of this paper is the development of an efficient decision procedure for the fundamental theory of fixed-sized bit-vectors with composition and extraction that can readily be integrated with Shostak's algorithm for combining decision procedures. We have successfully applied this procedure to proofs and lemmas that arise in the verification of a commercial microprocessor. The decision procedure obviated manual proof effort that was previously necessary.

To the best of our knowledge this is the first time that a specialized, efficient decision procedure for this bit-vector theory has been developed. Clearly, more experiments with our bit-vector decision procedures are needed to demonstrate the practical gain over the simple approach of proving bit-vector equivalences by reduction to bit-wise comparisons. Further work includes extensions of the bit-vector decision procedures to deal with arbitrary-sized bit-vectors and extraction positions possibly containing variables. For this case, however, we can not expect to have a polynomial solver for the core theory, since, using a reduction from 3SAT, it can be shown that solvability is *NP*-complete in this case [4]. On the other hand, using these extensions, one could apply a combination of decision procedures — including the one for the bit-vector theory — to prove properties about fixed- but arbitrary-sized bit-vectors.

Acknowledgments: Thanks to J. Skakkebaek for many useful comments, M.K. Srivas for supplying interesting test examples, and the second author expresses his gratitude to F.W. von Henke and J. Rushby for supporting a fruitful visit at SRI International, Menlo Park.

References

1. R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
2. D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's Decision Procedure for Combination of Theories. In M. A. McRobbie and J. K. Slaney, editors, *Proc. of CADE'96*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.
3. D. Cyrluk, O. Möller, and H. Rueß. An Efficient Decision Procedure for a Theory of Fixed-Sized Bitvectors with Composition and Extraction. Technical report, Universität Ulm, D-89069 Ulm, Oberer Eselsberg, December 1996.
4. O. Möller. Some Notes on Bit-Vectors. See: `http://www.informatik.uni-ulm.de/ki/Personen/moeller/bitvector-e.html`, 1996.
5. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
6. H. Rueß. Hierarchical Verification of Two-Dimensional High-Speed Multiplication in PVS: A Case Study. In M.K. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, November 1996.
7. R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31(1):1–12, January 1984.
8. M.K. Srivas and S.P. Miller. Formal Verification of the AAMP5 Microprocessor. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

A Canonizer

$$
\begin{array}{lll}
\alpha(s) & ::= & \textsc{cases}\ s\ \textsc{of} \\
& & \quad t \otimes u \rightarrow \alpha(t) \otimes \alpha(u), \\
& & \quad t_{[n]}{}^\wedge(n-1,0) \rightarrow \alpha(t_{[n]}), \\
& & \quad t^\wedge(i,j)^\wedge(k,l) \rightarrow \alpha(t^\wedge(k+j,l+j)), \\
& & \quad (t_{[n]} \otimes u_{[m]})^\wedge(i,j) \rightarrow \ \textsc{if}\ m > i\ \textsc{then}\ \alpha(u_{[n]}{}^\wedge(i,j)) \\
& & \quad\quad \textsc{elseif}\ j \geq m\ \textsc{then}\ \alpha(t_{[n]}{}^\wedge(i-m,j-m)) \\
& & \quad\quad \textsc{else}\ \alpha(t_{[n]}{}^\wedge(i-m,0)) \otimes \alpha(u_{[m]}{}^\wedge(m-1,j)) \\
& & \quad\quad \textsc{endif}\ , \\
& & \quad \textsc{otherwise} \quad s \\
& & \textsc{endcases}
\end{array}
$$

$$
\begin{array}{lll}
\beta(s) & ::= & \textsc{cases}\ s\ \textsc{of} \\
& & \quad c_{[n]} \otimes c_{[m]} \otimes u \rightarrow \beta(c_{[n+m]} \otimes u), \\
& & \quad x_{[n]}{}^\wedge(i,j) \otimes x_{[n]}{}^\wedge(j-1,k) \otimes u \rightarrow \beta(x_{[n]}{}^\wedge(i,k) \otimes u), \\
& & \quad x_{[n]}{}^\wedge(i,j) \otimes u \rightarrow x_{[n]}{}^\wedge(i,j) \otimes \beta(u), \\
& & \quad \textsc{otherwise} \quad s \\
& & \textsc{endcases}
\end{array}
$$

$$\sigma(t) ::= \beta(\alpha(t))$$

B Solving Equations over Simple Terms

This appendix is not part of the main submission but includes pseudo-code for a crucial part of the bit-vector solver that is referred to in the paper only by means of an example.

$csolve(t, u)$ ::=

IF $t \equiv u$ THEN RETURN $\emptyset$ ENDIF
IF $vars(t) = \emptyset$ THEN $swap(t, u)$ ENDIF
IF $vars(t) = \emptyset$ THEN RETURN **false** ENDIF
IF $vars(u) = \emptyset$ THEN RETURN $\{var(t) = u\}$ ENDIF
IF $vars(t) = vars(u)$
THEN /* both terms are necessarily extractions */
CASE
both parts do not overlap
$fresh(a^{(1)}), fresh(a^{(2)}), fresh(a^{(3)}), fresh(b^{(1)})$
RETURN $\{\mathrm{var}^{\star\star}(t) = a^{(1)} \otimes b^{(1)} \otimes a^{(2)} \otimes b^{(1)} \otimes a^{(3)}\}$
$\frac{1}{2} \cdot length(t) > length(overlapping\ part)$
$fresh(a^{(1)});\ fresh(a^{(2)});\ fresh(b^{(1)});\ fresh(b^{(2)})$
RETURN $\{\mathrm{var}(t) = a^{(1)} \otimes b^{(1)} \otimes b^{(2)} \otimes b^{(1)} \otimes b^{(2)} \otimes \mathrm{freshb1} \otimes a^{(4)}\}$
$length(overlapping\ part) \geq \frac{1}{2} \cdot length(t)$
$\Gamma \leftarrow leftmost\_position(t, u) - rightmost\_position(t, u) + 1$
$\Delta \leftarrow \mid left\_position(t) - left\_position(u) \mid$
$\Lambda \leftarrow \Gamma$ MOD Δ
IF $\Lambda = 0$ THEN $fresh(a^{(1)});\ fresh(a^{(2)});\ fresh(b^{(1)} : bvec[\Delta])$
RETURN $\{\mathrm{var}(t) = a^{(1)} \otimes b^{(1)} \otimes ... \otimes b^{(1)} a^{(2)}\}$
ELSE $fresh(a^{(1)});\ fresh(a^{(2)});$
$fresh(b^{(1)} : bvec[\Lambda]);\ fresh(b^{(2)} : bvec[\Delta - \Lambda])$
RETURN $\{\mathrm{var}(t) = a^{(1)} \otimes b^{(1)} \otimes b^{(2)} \otimes ...$
$\otimes b^{(1)} \otimes b^{(2)} \otimes b^{(1)} \otimes a^{(2)}\}$
ENDIF
ENDCASES
ELSE $fresh(a^{(1)});\ fresh(a^{(2)});\ fresh(a^{(3)});\ fresh(a^{(4)});\ fresh(c^{(1)});$
RETURN$\{var(t) = a^{(1)} \otimes c^{(1)} \otimes a^{(2)}, \mathrm{var(u)} = a^{(3)} \otimes c^{(1)} \otimes a^{(4)}\}$
ENDIF

This article was processed using the LaTeX macro package with LLNCS style