Normalization at the Arithmetic Bit Level

Markus Wedler, Dominik Stoffel, Wolfgang Kunz
Dept. of Electrical & Computer Eng., University of Kaiserslautern/Germany
email: wedler@eit.uni-kl.de

ABSTRACT

We propose a normalization technique for verifying arithmetic circuits in a bounded model checking environment. Our technique operates on the *arithmetic bit level (ABL) description* of the arithmetic circuit parts and the property. The ABL description can easily be provided by the front-end of an RTL property checker. The proposed normalization greatly simplifies the SAT instances to be solved for arithmetic circuit verification. Our approach has been applied successfully to verify the integer pipeline of an industrial microprocessor with advanced DSP capabilities.

Categories and Subject Descriptors

B6.3 [**Design Aids**]: Verification

General Terms

Algorithms, Verification

Keywords

Property checking, arithmetic bit level normalization, SAT

1. INTRODUCTION

Bounded model checking (BMC) [3] has gained increased significance in Electronic Design Automation (EDA). It is used to verify that a digital circuit design meets the desired behavior. In BMC the design of a sequential circuit is unrolled for a finite number of time frames and augmented with the property under verification. This can be translated into a satisfiability (SAT) problem and is thus handled by standard SAT solvers. These solvers will either give a proof of unsatisfiability or a counter example for the property.

It is well known that SAT solvers have problems when dealing with instances derived from the verification of arithmetic circuits. Hence, although SAT-based property checking can often be applied successfully to the control part of a design, it typically fails on data paths with large arithmetic blocks. One may resort to incomplete techniques like bit-slicing in order to find bugs in arithmetic units. However, they cannot prove the absence of a bug. Especially, it is very likely to miss errors in corner cases.

This motivated the development of automatic bit width reduction techniques like in [8, 9]. However, the resulting model still is often too large to be handled by a SAT solver and there are cases where no reduction is possible.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2005, June 13–17, 2005, Anaheim, California, USA.
Copyright 2005 ACM 1-59593-058-2/05/0006 ...$5.00.

Another idea is to use word-level solvers for integer linear programming (ILP) [4, 7, 14] or constraint logic programming [13]. The problem with word-level solvers is that they do not incorporate the large variety of pruning techniques for the Boolean part of the problem. Therefore, they usually perform poorly on the control part of a design. Moreover, in order to obtain highly optimized implementations designers often implement arithmetic functions at the bit-level so that word-level solvers are not adequate. Thus, a combination of word-level and Boolean solvers has to be developed. This problem is not simple because the different solvers cannot look into each other's non-solution areas. Two promising ways of integrating ILP and SAT have been proposed in [2, 5]. The first uses pseudo-Boolean constraints as clauses in a DPLL style solver and the second uses linear equations as propositions.

In [10] a method for equivalence checking of multipliers based on an arithmetic bit level description of the circuit was proposed. The arithmetic bit level contains partial products and addition networks. To verify that a circuit is a multiplier an arithmetic bit level representation is extracted from the gate netlist. For property checking this extraction is not necessary as the synthesis front-end can easily generate the arithmetic bit level information needed in a proof engine. The method of [11] utilizes the arithmetic bit level information to prune the search space of a standard SAT solver. It can help to prune the search space locally when analyzing individual addition networks. However, it cannot prevent exponential blowup when looking at sophisticated data paths consisting of several arithmetic blocks.

In this paper we suggest a normalization technique at the arithmetic bit level. This technique is based on an arithmetic bit level description of the problem consisting of partial product generators, addition networks and comparators. We will introduce an arithmetic normalization with respect to a set of comparators. After normalization the derived instance is handed over to a SAT solver. Also a hybrid solver like [2, 5] could be used. In fact, the proposed normalization may be very useful in providing additional word-level abstractions as they can be exploited by these solvers.

2. PROPERTY CHECKING FOR ARITHMETIC BLOCKS

In RTL property checking, design and property usually are synthesized into a plain SAT instance that is handled by a standard SAT solver. Unfortunately, SAT solvers get into complexity problems as soon as arithmetic circuitry comes into play.

In the following we use the Infineon property language called interval temporal logic (ITL) [12] to present a typical example of a property to be proven for an arithmetic processor pipeline. Table 1 specifies that the result of a multiplication is presented at some internal signal *ip_res* of a design after

four cycles if no reset has occurred and certain assumptions about the environment are valid. Note that the state of the design at time t is only constrained by the assumptions explicitly stated by the verification engineer, i.e., the bounded model checker starts at an arbitrary state and checks the implication *assumptions→commitments* in a bounded time interval. Properties of this kind are used frequently to verify

```
theorem mulXXX;
  freeze: op1_at_t=op1@t, op2_at_t=op2@t;
  assume:
    within[t,t+4]:no_reset;
    within[t,t+4]:environment_assumptions;
    at t:command(mulXXX,op1,op2);
  prove:
    at t+4:ip_res=command_res(mulXXX,op1_at_t,op2_at_t);
end theorem;
```

Table 1: Sketch of an arithmetic property

the correct implementation of the instruction set of a processor. For an arithmetic instruction one can specify the result by an arithmetic expression *command_res(operands)*.

It is clear that in general the unrolled circuit and the synthesized expression are structurally dissimilar. Moreover, the unrolled circuit may still contain a significant amount of control logic. Therefore, a general methodology needs to address two problems. First, it has to avoid that the SAT solver reasons over the entire globally reconvergent network created by expression and design. Second, it has to deal with the control parts embedded in the unrolled circuit.

The first issue is addressed in Sections 3 and 4. We will develop a normalization process that transforms both, property and design, in a way such that structural similarity is created and used for globally simplifying the SAT instance. The second issue is addressed in Section 5 where it is discussed how the embedded control parts are handled.

3. ABL DESCRIPTION

In this section we will define an arithmetic bit level description (ABL) of a circuit. An ABL models three kinds of objects: partial products, addition networks and comparators. Therefore, we first define what these basic objects are and then compose them into a graph. Sometimes we want to map bit vectors onto the unsigned integer they represent and use the notion $\mathbb{Z}(x) := \sum_{i=0..n} 2^i x_i$. Furthermore, *mod* denotes the modulo operation on $\mathbb{Z}$, i.e., $a \ mod \ b := k$ where k is the smallest $k \geq 0$ such that there is an $m \in \mathbb{Z}$ with $a - bm = k$.

An addition network calculates a weighted sum $\sum_{l \in A} w_l l$ over a set of boolean variables A where the weights are (signed) integers. Formally, we define:

Definition 1. *An addition network N is a 4-tuple (A, r, w, c) where:*

- *A is a set of boolean variables with $A = A_0 \cup \cdots \cup A_n$,*
- *$r = (r_0, \ldots, r_n)$ is a vector of boolean variables,*
- *$w = (w_0, \ldots, w_n)$ is a vector of mappings $w_i : A_i \to \mathbb{Z}$,*
- *$c = (c_0, \ldots, c_n)$ is a vector of integers $c_i \in \mathbb{Z}$.*

We call A the set of addends of the addition network and $n+1$ the number of columns of N. $a \in A_i$ is called an addend of column i of the network. For each addend a, $w_i(a)$ is called its weight. r_i is called the result of column i. c_i is called the constant offset of column i.

The result r of N is constrained by the addends through the following equation:

$$\mathbb{Z}(r) = (\sum_{i=0..n} 2^i (c_i + \sum_{a \in A_i} w_i(a)a)) \ mod \ 2^{n+1}. \quad (1)$$

Note that $A_i \cap A_k = \emptyset$ does not necessarily hold, i.e., an addend can be added to different columns of the addition network. Moreover, every addition network can be identified with a boolean function that is a composition of halfadders. This is easy to see for addition networks without negative weights. Fortunately, we can eliminate negative weights in an addition network. Suppose we have $w_i(a) < 0$ for an addend a with $i < n$. By redefining the weight of the addend a as $w'_i(a) = -w_i(a)$ and $w'_{i+1}(a) = w_{i+1}(a) + w_i(a)$, we obtain an equivalent addition network where the negative weight is moved to column $i+1$. In this way, we can move all negative weights to column n. For this column we can redefine $w'_n(a) = -w_n(a)$ to obtain an equivalent network with positive weights. To prove this we consider Equation 1 again. With $x + 2^n w_n(a) - (x - 2^n w_n(a)) = 2^{n+1} w_n(a)$ it follows that $(x - 2^n w_n(a)) \bmod 2^{n+1} = (x + 2^n w_n(a)) \bmod 2^{n+1}$, i.e., the right hand side of Equation 1 is unchanged.

Negative weights will be used to handle signed arithmetic operations. But since they can be removed in the shown way they do not need special consideration in the normalization technique to be developed.

As an example for an addition network we model the piece of circuitry in Figure 1 as follows:

- $A_0 = \{x_0\}$, $A_1 = \{x_1, x_2\}$, $A_2 = \{x_3\}$, $A_3 = \emptyset$,
- $r = (y_0, y_1, y_2, y_3)$,
- $w_i(x) = 1$ for all $x \in A_i$ and all i,
- $c = (0, 0, 0, 0)$.

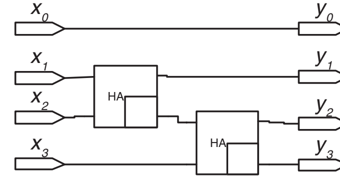

Figure 1: Addition network of a 2-bit multiplier

The addition network of Figure 1 produces carries in columns 1 and 2. Formally we define:

Definition 2. *An addition network generates carries in column k if there is an assignment of the addends such that*

$$s_k := (\sum_{i=0..k} 2^i (c_i + \sum_{a \in A_i} w_i(a)a)) \geq 2^{k+1} \ or \ s_k \leq -2^{k+1}.$$

Next, we would like to model partial product generators. A partial product generator models bitwise multiplication of two vectors of boolean variables.

Definition 3. *A partial product generator P is a triple (o_1, o_2, p) where $o_1 = (o_{1,0}, \ldots, o_{1,n})$, $o_2 = (o_{2,0}, \ldots, o_{2,m})$ and $p = (p_0, \ldots, p_{n*m})$ are vectors of distinct boolean variables. The $o_{i,k}$ are called bit-level operands, the p_i are called partial products. o_1 and o_2 are called word-level operands. The partial products are constrained by*

$$p_{i+nk} = o_{1,i} \cdot o_{2,k}.$$

Combining addition networks and partial product generators we can model the arithmetic bit vector functions $+$ and $*$. Since weights may be negative, we can also model subtraction and signed multiplication. Subtraction of two bit vectors $(a_k, \ldots, a_0)$ and $(b_k, \ldots, b_0)$ is modeled by an addition network with $A_i = \{a_i, b_i\}$, $w_i(a_i) = 1$, $w_i(b_i) = -1$ and $c_i = 0$. For signed multiplication of bit vectors $(a_k, \ldots, a_0)$ and $(b_l, \ldots, b_0)$ we subtract the partial products $a_k b_i (i < l)$ and $a_j b_l (j < k)$ from the appropriate columns in the addition network, i.e., $w_{k+i}(a_k b_i) = -1 (i < l)$ and $w_{j+l}(a_j b_l) = -1 (i < l)$.
In the following, this framework is extended by a comparator that represents bit vector comparisons like $=$ and $\neq$. During the normalization process we will exploit the invariant of these comparison operations that addition of any value x to both operands of a comparator does not change the result. Note that due to possible overflows of the addition this invariant does not hold for $<, >, \leq$ and $\geq$. However, under certain restrictions on addition networks in the fanin of such a comparison normalization can be performed in the same manner as it will be introduced for $=$ and $\neq$ in the following.

Definition 4. *A comparator C is a 4-tuple (c_1, c_2, o, f) where c_1 and c_2 are vectors of boolean variables $c_1 = (c_{1,0}, \ldots, c_{1,n})$ and $c_2 = (c_{2,0}, \ldots, c_{2,n})$, o is a boolean variable, and f is a mapping $f : B^n \times B^n \rightarrow B$ such that $f(x, y) = f(x + z, y + z)$ for all $z \in B^n$ where $+$ denotes bit-vector addition. The c_i are called operands, o is called the result and f is called the comparison function. o is constrained by the operands by the boolean function*

$$o = f(c_1, c_2).$$

From the basic blocks described above we can now compose our arithmetic bit level description by the following definition.

Definition 5. *Let $\mathcal{N}$ be a set of addition networks, $\mathcal{P}$ be a set of partial product generators and $\mathcal{C}$ be a set of comparators. An arithmetic bit level description (ABL) is a directed acyclic graph $G = (V, E)$, where V is the union $V = \mathcal{N} \cup \mathcal{P} \cup \mathcal{C}$.*

With this definition we have not yet defined the topology of the graph. For example, it allows edges between addition networks that have no variable in common. Before defining the topology, we define the terms *fanin* and *fanout* variable for the components of the ABL.

Definition 6. *A boolean variable v is called* fanin *variable of an addition network N iff v is an addend of N. v is called* fanin *variable of a partial product generator P iff v is an operand of P. v is called* fanin *variable of a comparator C iff v is an operand of C. v is called* fanout *variable of an addition network N iff v is a result of N. v is called* fanout *variable of a partial product generator P iff v is a partial product of P.*

A correct topology for the ABL is derived from the fanin/fanout relationship between the components as follows:

Definition 7. *An ABL has a correct topology iff for all $(x, y) \in E$ there is a boolean variable l such that l is fanin variable of y and l is fanout variable of x.*

In the following, all ABLs are supposed to have correct topology. Note that edges (c, a) or (c, p) with $a \in \mathcal{N}, p \in \mathcal{P}$ and $c \in \mathcal{C}$ are prohibited by Definition 7. Every node of an ABL represents a multi-output boolean function and the entire ABL corresponds to the composition of these functions. Hence, we can establish an equivalence relation on the set of ABLs.

Definition 8. *Two ABLs A, B are equivalent iff they represent the same boolean function.*

4. NORMALIZATION

In this section we define a normal form for an ABL and describe an algorithm to normalize a given ABL. Before we go into details, we would like to motivate our ideas with a small example. Suppose we have a circuit for evaluating the polynomial $p = ax^2 + bx + c$ in two clock cycles using a multiply/accumulate unit that calculates $o = xy + z$. Unrolling this circuit leads to the expression $p = (ax + b)x + c$. At the word level one could use the distributive law to normalize both expressions to $p = ax^2 + bx + c$.
The same kind of normalization can be achieved at the bit level. Let us suppose that all the operands a, b, c and x are unsigned two-bit numbers.
Then, we have
$p = (2a_1 + a_0)(2x_1 + x_0)^2 + (2b_1 + b_0)(2x_1 + x_0) + (2c_1 + c_0)$.
We use the distributive law to normalize this expression:

$$p = 8a_1x_1 + 8a_1x_0x_1 + 4a_0x_1 + 4a_0x_0x_1 + 4b_1x_1$$
$$+ 2a_1x_0 + 2b_0x_1 + b_0x_0 + 2b_1x_0 + 2c_1 + a_0x_0 + c_0.$$

Unfortunately, in practice we cannot use term representations like this due to those parts in the design where the designer realizes an arithmetic function by instantiating bit-level arithmetic units like halfadders or fulladders. We refer to these blocks as hard-coded arithmetics. They are used by designers to obtain more sophisticated implementations. In these implementations intermediate results cannot be described at the word-level and word-level signals are broken down to the bit-level using slice operators. Generating the above representation is not practical in this scenario. However, the intuitive view of normalization translates to our notion of ABLs in the following way:

Definition 9. *An ABL $(\mathcal{N} \cup \mathcal{P} \cup \mathcal{C}, E)$ is in* **normal form** *iff E does not contain any edge (N, P) with $N \in \mathcal{N}$ and $P \in \mathcal{P}$.*

In other words, no addition network feeds a partial product generator.

Definition 10. *An ABL $(\mathcal{N} \cup \mathcal{P} \cup \mathcal{C}, E)$ in normal form is* **reduced** *iff the following conditions hold:*

a) For all $N_1, N_2 \in \mathcal{N}$ for which there is a $C \in \mathcal{C}$ such that $(N_1, C) \in E$ and $(N_2, C) \in E$, N_1 and N_2 do not have any common addends.

b) There is no edge $(N_1, N_2) \in E$ with $N_1, N_2 \in \mathcal{N}$.

ABLs in reduced normal form do not cascade addition networks and simplify comparisons by subtraction of common addends. Note that the reduced normal form is not canonical. For example, one could add a constant offset of 2 to the uppermost column of any addition network and obtain an equivalent network. Theorem 1 states that every boolean function can be expressed by an ABL in reduced normal form. This implies that a reduced normal form for an ABL always exists.

THEOREM 1. *For every boolean function there is an equivalent ABL in reduced normal form.*

Proof: Let f be a boolean function. f can be expressed in (positive) Reed-Muller normal form, i.e., $f = \sum \prod x_i$, where $\sum$ denotes exclusive OR and $\prod$ denotes conjunction. Hence all products $\prod x_i$ can be expressed by cascaded 1×1 partial product generators. The entire sum can be represented by an addition network with one column. The products are the addends of this column with weight 1. ■

From the proof of Theorem 1 one could immediately derive a normalization algorithm for ABLs. However, calculating the Reed-Muller normal form is not efficient. Furthermore,in the ABLs, calculated in the proof, all addition networks consist of a single column only. This eliminates the great benefit of addition networks that they implicitly represent carries between their columns. The rest of this section therefore describes an efficient heuristic algorithm that determines a reduced normal form directly on a given ABL.

DEFINITION 11. *Two addition networks N_1 and N_2 are merged into the addition network N_3, iff the ABLs $(V, E) = (\{N_3\}, \emptyset)$ and $(V', E') = (\{N_1, N_2\}, \{(N_1, N_2)\})$ are equivalent.*

In other words, we can replace N_1 and N_2 by N_3.

LEMMA 1. *Let N_1, N_2 be addition networks in an ABL with $(N_1, N_2) \in E$. Further let N_1 not generate any carries in the uppermost column. N_1 can be merged with N_2 if one of the following conditions holds:*

- *A column $k \geq 0$ of N_1 exists such that for each column $j \geq k$ the result r_j is an addend of column $j - k$ of N_2.*
- *A column $k \geq 0$ of N_2 exists such that for each column i of N_1 the result r_i is an addend of column $i + k$ of N_2.*

A column i of an addition network N_1 can be merged with a column j of an addition network N_2 if the results of i is an addend of j.

Two addition networks can be merged in linear time in the number of addends. This is done by adding all the addends of N_1 to the appropriate addend sets of N_2 and redefining the weight functions and constant offsets accordingly.

DEFINITION 12. *Let P_1, P_2 and $P3$ be partial product generators. P_1 and P_2 are merged into P_3 iff for all partial products of P_1 and P_2 an equivalent partial product of P_3 exists.*

LEMMA 2. *Let P_1, P_2 be partial product generators in an ABL. P_1 and P_2 can be merged if they have a common word-level operand.*

Merging of partial product generators means concatenating the non-common operands. This can be done in linear time in the number of operands. If the operands are sorted one can check the condition of the above lemma in linear time. Hence the complexity of the merge routine can be bounded by $O(n \log n)$.

DEFINITION 13. *Let P be a partial product generator and N be an addition network with $(N, P) \in E$. P can be distributed through N iff there is a partial product generator P' and addition networks $N'_1 \dots N'_k$ that add up the partial products of P' such that replacing N and P by $N'_1 \dots N'_k$ and P' yields an equivalent ABL.*

LEMMA 3. *Let P be a partial product generator and N be an addition network with $(N, P) \in E$ and $(N, X) \in E \Rightarrow X = P$. P can be distributed through N if one of the operand sets of P equals the set of results of N.*

Note that the conditions of Lemma 3 can always be fulfilled by duplicating the addition network (in the case of fanout) and extending the partial product generator such that all results of the addition network are operands on P.

Figure 2 visualizes the distribution of a 2×1 partial product generator through a halfadder. Distributing partial product generators replicates the addition network k times if k is the size of the second operand of the partial product generator. Note that this is a potential source of memory blowup. In practice it turns out that the new addition networks can usually be merged to some other addition network immediately.

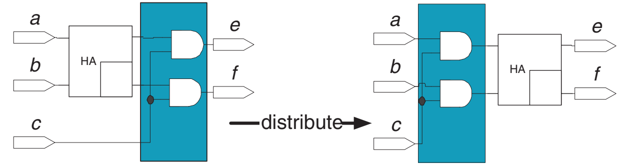

Figure 2: Distribution of partial products

With these two operations we are now able to move all partial product generators in front of all addition networks, i.e., we can normalize and reduce an ABL. This is done by the following process:

1. merge addition networks
2. merge partial products
3. distribute partial products
4. merge addition networks
5. determine equivalent partial products
6. remove common addends from each pair of addition networks feeding a common comparator

In fact, for normalization step 3 is sufficient. All the other steps are added to ensure that the resulting ABL is also reduced.

5. HANDLING CONTROL LOGIC

In this section we describe how to eliminate control logic separating arithmetic blocks in a design.

5.1 Constant propagation

For properties like the one sketched in Table 1, the frontend of a bounded model checker augments the unrolled circuit with logic creating an output for both, assumptions and property. The assumptions are set to 1 when proving the property. In many cases these constants will propagate deep into the circuit. For example, the assumption that a specific multiplication instruction is executed will specify a lot of internal control signals connecting the decoder and the integer pipeline of a microprocessor. Unfortunately, propagation is not complete. Hence, in real-life problems after constant propagation there always remains some logic that separates the arithmetic parts from each other. However, it turns out that this logic is transparent, i.e., a SAT algorithm can easily prove that equivalences between the outputs of one arithmetic block and the inputs of some other arithmetic block exist or that certain signals are constant. These equivalences and constants are then used for logic substitutions which eliminate the remaining logic.

5.2 Handling difficult cases by abstraction

There are practically relevant properties where the commitment is not a pure arithmetic expression. These properties contain a small portion of control. An example for this kind of problem is multiplication with saturation. Saturation means that the output of a circuit is set to a specific constant when an arithmetic result exceeds a certain threshold value.

In these cases, we can use normalization to obtain a word level abstraction for the arithmetic part of the design. Figure 3 illustrates a situation where normalization is useful for word-level abstraction.

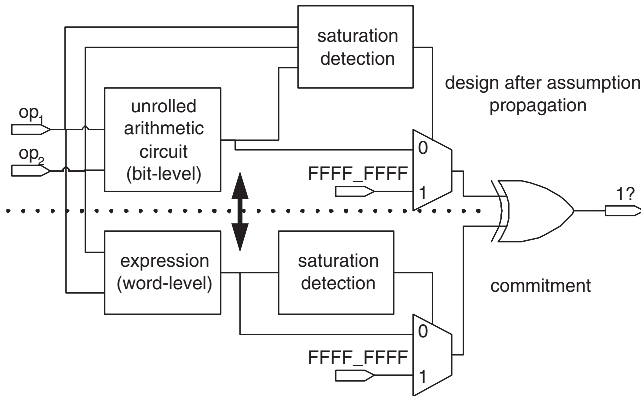

Figure 3: SAT-problem before abstraction

Suppose a design is composed of a hard-coded arithmetic circuit and a saturation detection unit. For performance reasons the saturation unit also uses inputs to estimate the saturation condition such that only part of the result is needed to obtain a correct saturation condition. However, in the property part of the problem (below the dotted line) the saturation condition is calculated directly on the result of an expression. In this situation a SAT solver implicitly has to verify equivalence of the design and the expression in the property and this causes an exponential blowup.

With normalization it is easily verified that the arithmetic circuit portion implements an arithmetic word-level expression. The problem is then simplified by sharing the word-level expression and removing the unrolled circuitry. In essence, we have *abstracted* the arithmetic circuit portion in the design by the word-level expression.

This abstracted instance can then be handled using hybrid solvers like [2, 5]. In our experiments we even solved 32-bit instances of this kind of problem after abstraction using a standard SAT solver.

6. EXPERIMENTAL RESULTS

In this section we provide data from the experiments we conducted with our implementation of the normalization algorithm called ABLProp.

This section is divided into two parts. In the first part we explore the scalability of our approach with respect to the bit width of the arithmetic operations. Then, the second part describes our experience with applying the method to verify the integer pipeline of an industrial microprocessor.

6.1 Scalability

To demonstrate the scalability of the proposed approach we applied it to normalize the following equations:

$$ab + ac = a(b + c) \tag{2}$$

$$a(bc) = (ab)c \tag{3}$$

$$\begin{aligned} a * b = {} & a[0 : n-1] * b[0 : n-1] \\ & + 2^n a[n : 2n-1] * b[0 : n-1] \\ & + 2^n a[0 : n-1] * b[n : 2n-1] \\ & + 2^{2n} a[n : 2n-1] * b[n : 2n-1] \end{aligned} \tag{4}$$

We assumed all the operations to be specified at the word-level and generated ABLs in different bit widths. Note that equation (4) results when an n-bit multiplier is used to calculate the result of a $2n$-bit multiplication in four clock cycles. Table 2 presents the CPU times to prove the equations for different operand bit widths. Note that the first column always reports the bit width of a single operand. For equation (4) the bit width corresponds to n.

bit width	runtime for equation [s]		
	(2)	(3)	(4)
4	0.01	<0.01	<0.01
8	0.03	0.02	0.02
16	0.43	0.57	0.09
24	3.21	5.59	0.23
32	15.92	30.06	0.46
48	193.46	386.41	1.37
64	1151.86	2303.47	3.16

Table 2: Experimental results for scalability

Table 2 indicates good scalability of the normalization approach with respect to operand and result bit widths. A closer look at the numbers confirms that the runtime complexity is polynomial. But, more importantly, using our approach we can verify very large instances of arithmetic blocks in short time. For example, all 192 output bits of the largest instance of Eq. 3 are verified in about 40 minutes of total CPU time.

6.2 Industrial Application

We examined our approach for the next-generation 32-bit super-scalar pipelined microprocessor TriCore 2 [1] with advanced DSP capabilities developed by Infineon. The goal is to prove that the integer pipeline of the processor performs all specified integer operations and returns the correct result in the result register. In our experience, this can be done for most of the instructions using Infineons SAT-based bounded model checker Gateprop. Severe problems occur, however, if the pipeline also implements advanced arithmetic DSP operations, especially multiplication instructions. The integer pipeline of the examined design performs more than thirty different multiply-accumulate instructions and each of them in several variations. None of the resulting SAT instances could be solved by a state-of-the-art SAT solver. Word-level solvers turn out not to be applicable because most of the arithmetic circuitry is hard coded at the bit-level.

For example, there are signed and unsigned multiplications of 32- and 16-bit operands and packed instructions where two 16-bit multiplications are performed on two 32-bit input words in parallel. Moreover, there are different modes of saturation. Since all these instructions are executed by the same hardware, a lot of control logic exists around the hard-coded multipliers used in the design. Moreover, the

multiply/accumulate unit is pipelined such that intermediate results are stored in registers and cannot be modeled at the word level. Therefore, a property like the one sketched in Table 1 is also infeasible for word-level decision procedures like ILP or CLP.

We integrated ABLProp, our implementation of the normalization algorithm, with Gateprop, Infineons industrial SAT-based bounded model checker. The latter was used to synthesize design and property into a netlist of bit vector functions. This netlist has two outputs p and a that refer to property and assumptions. We have to prove the implication $a \Rightarrow p$. As a first step, we assign the value 1 to a and propagate this into the bit vector netlist. After this propagation step we translate the arithmetic bit vector functions into an ABL and normalize this ABL. The normalized ABL and the non-arithmetic bit vector functions are converted to CNF and handed over to the SAT solver [6].

Using this flow, we verified a representative subset of arithmetic instructions with multiplication. Table 3 shows the CPU times of ABLProp including assumption propagation and the solution of the final SAT-instance. In the first column of the table the type of arithmetic instruction is specified. For the two 32-bit operands 16L and 16U refer to the lower and the upper halfword. *ip_res* refers to the result register of the integer pipeline. Furthermore, << refers to a leftshift and || refers to the concatenation of two bit vectors.

<table>
<tr><th>Multiplication Type</th><th>CPU time</th></tr>
<tr><td>ip_res==signed(16L)*signed(16L)
|| signed(16U)*signed(16U)</td><td>6.67 s</td></tr>
<tr><td>ip_res==signed(16U)*signed(16L)
||signed(16L)*signed(16U)</td><td>6.68 s</td></tr>
<tr><td>ip_res==signed(16U)*signed(16L)
||signed(16L)*signed(16L)</td><td>6.75 s</td></tr>
<tr><td>ip_res==signed(16L)*signed(16U)
||signed(16U)*signed(16U)</td><td>6.76 s</td></tr>
<tr><td>ip_res==signed(16L)*signed(16L)<< 1
||signed(16U)*signed(16U)<< 1</td><td>14 s</td></tr>
<tr><td>ip_res==signed(16U)*signed(16L)<< 1
||signed(16L)*signed(16U)<< 1</td><td>14.02 s</td></tr>
<tr><td>ip_res==signed(16U)*signed(16L)<< 1
||signed(16L)*signed(16L)<< 1</td><td>14.11 s</td></tr>
<tr><td>ip_res==signed(16L)*signed(16U)<< 1
||signed(16U)*signed(16U)<< 1</td><td>14.25 s</td></tr>
<tr><td>ip_res==signed(16U)*signed(16U)</td><td>8.34 s</td></tr>
<tr><td>ip_res==signed(16L)*signed(16L)</td><td>3.22 s</td></tr>
<tr><td>ip_res==signed(32)*signed(16U)</td><td>4.03 s</td></tr>
<tr><td>ip_res==signed(32)*signed(16L)</td><td>4.1 s</td></tr>
<tr><td>ip_res==32 most significant bits of
signed(32)*signed(32)</td><td>76.5 s</td></tr>
<tr><td>ip_res==signed(32)*signed(16U)
sign Extended to 64 Bits</td><td>16.29 s</td></tr>
<tr><td>provide abstraction for
ip_res == unsigned(32)*unsigned(32)
saturated to 32 Bits</td><td>75.09s</td></tr>
</table>

Table 3: Experimental results on industrial properties

To verify the last instruction we used normalization for abstraction as described is Section 5. The CPU time reported refers to the proof that the abstraction is correct.

For the abstracted model we could prove correctness of *ip_res* using Gateprop. Table 4 reports run times for the two properties verifying the overflow and non-overflow case. On the instances without abstraction the proof was aborted after 2500s.

Property	CPU time [s]	
	abstracted model	original model
no overflow	152.32	>2500
overflow	56.61	>2500

Table 4: CPU times for saturation

7. CONCLUSION

We have presented a normalization technique at the arithmetic bit level. To the best of our knowledge this is the first verification technique that can handle hard-coded pipelined arithmetic data paths. Our approach can smoothly be integrated into modern SAT-based BMC environments. Furthermore, it provides abstractions that are useful to enhance environments using hybrid solver technology.

8. ACKNOWLEDGMENT

This work was funded in part by Infineon and BMBF under grants No. 01M3072E and 01M3069A. We thank Raik Brinkmann, Infineon, for a fruitful collaboration.

9. REFERENCES

[1] Infineon TriCore 2 Architecural Manual. http://www.infineon.com/tricore.

[2] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT-based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. Conference on Automated Deduction (CADE)*, pages 195–210, 2002.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. Intl. Design Automation Conference (DAC-99)*, pages 317–320, June 1999.

[4] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proc. Asia and South Pacific Design Automation Conference (ASPDAC-02)*, Bangalore, India, 2002.

[5] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *Proc. Design Automation Conference (DAC-03)*, pages 830–835, 2003.

[6] N. Een and N. Sörensson. An extensible SAT-solver. In *Proc. 6. Intl. Conf. on Theory and Applications of Satisfiability Testing(SAT 2003)*, May 2003.

[7] F. Fallah, S. Devadas, and K. Keutzer. Functional vector generation for HDL models using linear programming and boolean satisfiability. *IEEE Transactions on CAD*, CAD-20(8), 2001.

[8] P. Johannsen. BOOSTER: Speeding up RTL property checking of digital designs by word-level abstraction. In *Proc.Intl. Conf. Computer Aided Verification(CAV-01)*, pages 373–377, July 2001.

[9] P. Johannsen and R. Drechsler. Formal verification on the RT level computing one-to-one design abstractions by signal width reduction. In *Proc. IFIP International Conference on Very Large Scale Integration(IFIP VLSI-SOC 2001)*, Montpellier, France, 2001.

[10] D. Stoffel and W. Kunz. Verification of integer multipliers on the arithmetic bit level. In *Proc. International Conference on Computer-Aided Design (ICCAD-01)*, pages 183–189, San Jose, CA, November 2001.

[11] M. Wedler, D. Stoffel, and W. Kunz. Arithemetik reasoning in DPLL-based SAT solving. In *Proc. Conference on Design, Automation and Test in Europe (DATE-04)*, Paris, France, 2004.

[12] K. Winkelmann, D. Stoffel, G. Fey, and H. Trylus. Cost-efficient block verification for a UMTS up-link chip-rate coprocessor. In *Proc. Conference on Design, Automation and Test in Europe (DATE-04)*, Paris, France, 2004.

[13] Z. Zeng, M. Ciesielski, and B. Rouzeyre. Functional test generation using constraint logic programming. In *Proc. IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2001)*, Montpellier, France, 2001.

[14] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: A unified approach to RTL satisfiability. In *Proc. Conference on Design, Automation and Test in Europe (DATE-01)*, Munich, Germany, 2001.