Abstraction of Word-level Linear Arithmetic Functions from Bit-level Component Descriptions

Pallab Dasgupta P.P. Chakrabarti Amit Nandi Sekar Krishna
Arindam Chakrabarti
Department of Computer Science & Engineering,
Indian Institute of Technology, Kharagpur, INDIA 721302
pallab,ppchak@cse.iitkgp.ernet.in

Abstract

RTL descriptions for word-level arithmetic components typically specify the architecture at the bit-level of the registers. The problem studied in this paper is to abstract the word-level functionality of a component from its bit-level specification. This is particularly useful in simulation since word-level descriptions can be simulated much faster than bit-level descriptions. Word-level abstractions are also useful for reducing the complexity of component matching, since the number of words is significantly smaller than the number of bits. This paper presents an algorithm for abstraction of word-level linear functions from bit-level component descriptions. We also present complexity results for component matching which justifies the advantage of performing abstraction prior to component matching.

1. Introduction

The RTL description of a word-level arithmetic function is typically specified at the bit-level of the registers. For example, the RTL descriptions of a ripple-carry adder and a carry-save adder differ at the bit-level, where as, at the word level they both have the same functionality, namely addition. The bit-level RTL description of an arithmetic function is significantly more complex than its behavioral word-level description.

The abstraction problem studied in this paper is as follows. We are given a component description which has a set of n words, $v_1, v_2, \ldots, v_n$, as inputs and a set of k words, $y_1, y_2, \ldots, y_k$, as outputs[1]. The logic implemented by the component can be specified at the bit level. We are to determine whether the function represented by each output y_i can be abstracted into some word-level linear arithmetic function, f_i, such that $y_i = f_i(v_1, v_2, \ldots, v_n)$. We consider components which are solely combinational.

The abstraction problem is closely related to the verification problem [2, 3], but presents an important difference. In the verification problem, we are given the word-level function, f, and are required to verify whether the component implements f. In the abstraction problem, we are required to determine f.

Word-level abstraction of components is useful for several reasons. Firstly, simulating word-level operations is significantly faster than simulating its bit-level description. A word-level abstraction allows us to rewrite the component in terms of word-level operations prior to simulation. This is an important gain, since simulators spend most of the time in simulating combinational logic between cycles, and many instances of a component can be present.

Secondly, a word-level representation has fewer variables, and is therefore better suited for component matching and verification. Most of the literature on verification [2, 3, 5, 7] assumes that the function, f, and the component are specified in terms of the same set of input variables. Thus by using the same variable ordering, we are able to convert both the component and the given function into canonical forms such as ROBDDs [1] or BMDs [2] and test their equivalence. Component matching is a more difficult problem, since the names of the variables used in a component may be different from the names of the inputs of the function, and the mapping from the inputs of the function to the inputs of the component is not given.

In this paper, we present an algorithm for determining word-level abstractions for linear arithmetic components. The algorithm works in two phases. In the first phase, we create a *BMD [2, 3] representation of the component and hypothesize a word-level function from it. In the second phase, we verify the hypothesis. We show that if the hypothesis fails, then the component does not have a word-level functionality, otherwise the hypothesis yields the cor-

[1] Boolean variables are treated as single bit words.

0-7695-0993-2/2001/$10.00 © 2001 IEEE

rect abstraction.

As a related result, we show that the component matching problem is NP-hard for word-level linear arithmetic functions. We further show that even when the number of inputs of the component is identical to the number of variables of the given function, component matching is unlikely to be possible in polytime, since the graph-isomorphism problem (which is not known to be in P) reduces to this version of the component matching problem.

2 Abstraction, verification, and matching

The word-level abstraction problem is defined as follows. We are given a component description, C, which has a set of input words, $v_1, v_2, \ldots, v_n$, and one or more output words, $w_1, w_2, \ldots, w_k$. The component defines the output words as functions of the input words, which may be at the bit level or at the word level. We need to determine whether an output word, w_i, models a word-level linear arithmetic function on the set of input words. The following example compares the component abstraction, verification and matching problems.

Example 1 The Verilog module shown in Fig 1 describes a circuit at the bit-level. We are to determine whether the module implements any word-level linear arithmetic function. In this case, the answer is yes, and the function is $r = xy + 2yz$. If we now modify the circuit slightly, say, by assigning output `r[0]` as `y[1]&q[0]` instead of `y[0]&q[0]`, then the output word, r, can no longer be described by a linear arithmetic function of x, y, and z.

The verification problem is simpler than the abstraction problem, since we are given the function, $r = xy + 2yz$, are are asked to verify whether the component implements the function. Since the *BMD representation of a linear arithmetic function is canonical for a given variable ordering, we can perform the verification by constructing the *BMD representation for the component and checking whether it is identical to the *BMD representation of the function with the same variable ordering. Fig 2 shows the *BMD representation for the circuit of Fig 1 with the variable ordering: $y[1] \prec y[0] \prec z[1] \prec x[1] \prec z[0] \prec x[0]$.

In the component matching problem we are given a function, say, $f = 3pq$, and are required to determine whether the given component matches the function. In this case, it does match, if p is assigned to y, and q is assigned to both x and z. Thus for component matching we have to determine the matching between the parameters of the given function and the inputs of the component, provided such a correspondence exists at the word-level. □

3 The abstraction algorithm

In this section, we present an algorithm which determines whether a given output word of a given component represents any word-level linear arithmetic function on its input words. If any such function exists, then the algorithm finds it. The algorithm works in two phases:

```
module shadd (r, x, y, z);
output [4:0] r;
input  [1:0] x;
input  [1:0] y;
input  [1:0] z;
wire   [3:0] q;
 assign q[0] = x[0];
 assign q[1] = x[1]^z[0];
 assign q[2] = z[1]^(x[1] & z[0]);
 assign q[3] = x[1] & z[0] & z[1];
 assign r[0] = y[0] & q[0];
 assign r[1] = (y[0] & q[1])^(y[1] & q[0]);
 assign r[2] = (y[1] & q[1])^(y[0] & q[2])
              ^(y[0] & y[1] & q[0] & q[1]);
 assign r[3] = (y[1] & q[2])^(y[0] & q[3])
        ^((q[0]|q[2]) & q[1] & y[1] & y[0]);
 assign r[4] = (y[1] & q[3])^((q[2]|(q[3]
            & q[0])) & q[1] & y[1] & y[0]);
endmodule
```

Figure 1. Bit-level Verilog description for a word-level circuit

Phase 1: Hypothesis Creation:: In the first phase, we construct a *BMD representation for the output word of the component and create a word-level hypothesis from it. Intuitively, a hypothesis is a *guess* of the word-level function. We shall show that if the output word represents a word-level function, then the hypothesis is the same function (that is, the guess is correct). However, if the output word does not represent any word-level function, we may still get a hypothesis, and hence we require the second phase of the algorithm.

Phase 2: Hypothesis Verification:: In the second phase, we verify whether the hypothesis is correct. To do this, we create the *BMD for the hypothesis and test its equivalence with the *BMD of the component.

The most important part of the algorithm is the creation of the hypothesis, which we now detail.

3.1 Creating the hypothesis

Let f be a function on a set of n words. Let x be any one of these n words. The moment decomposition of f with respect to $x[i]$ (that is, the i^{th} bit of x) is:

$$f \quad = \quad f_{\overline{x[i]}} + x[i] \cdot (f_{x[i]} - f_{\overline{x[i]}}) \quad = \quad f_{\overline{x[i]}} + x[i] \cdot f_{\dot{x}[i]}$$

$f_{\dot{x}[i]}$ is called the *linear moment* of f with respect to $x[i]$. Based on the moment decomposition, it is possible to expand f into a canonical sum-of-prducts form, which is

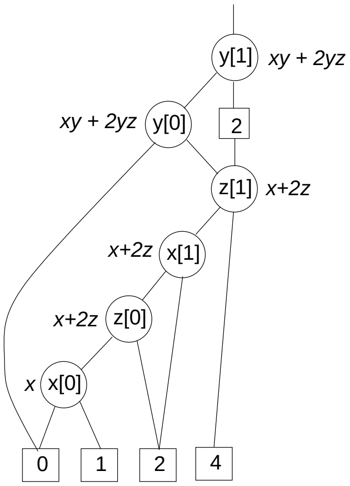

Figure 2. *BMD for module shadd

known as the *moment expansion* of the function. For example, the moment expansion of the function $r = xy + 2yz$ from Example 1 is as follows:

$$\begin{aligned} r \;=\; & x[0]y[0] + 2z[0]y[0] + 2x[1]y[0] + 4z[1]y[0] + \\ & 2x[0]y[1] + 4z[0]y[1] + 4x[1]y[1] + 8z[1]y[1] \end{aligned}$$

It is easy to see that the moment expansion of a linear function will never contain any term involving two bits of the same word. Also, each term of the moment expansion is unique.

The hypothesis creation scheme is derived out of the following results about binary moment decompositions of linear arithmetic functions.

Lemma 1 *f is a word-level linear arithmetic function only if the linear moment, $f_{\dot{x}[i]}$, of f with respect to $x[i]$, is independent of x.*

Proof: Obviously, $f_{\dot{x}[i]}$ is independent of $x[i]$. Suppose $f_{\dot{x}[i]}$ is dependent on $x[j]$, for some j, $j \neq i$. This means that in the moment expansion of function, f, we have one or more terms which have the conjunction of $x[i]$ and $x[j]$. Clearly, this is not possible if f is linear. □

Definition 1 *[Word-level abstract of term:]*
We define the word-level abstract for a term of the moment expansion as follows. Replace each bit of the term by the corresponding word name divided by the positional weight of the bit in the word. For example, an occurance of $x[i]$ will be replaced by $x \cdot 2^{-i}$. □

The *word-level abstract* of some terms are as follows:

$$\begin{aligned} 2x[1]y[0] &\equiv 2\left(\frac{x}{2}\right)y &= xy \\ 40z[2]q[1]p &\equiv 40\left(\frac{z}{4}\right)\left(\frac{q}{2}\right)p &= 5zqp \\ 10x[0]x[1]y[0] &\equiv 10x\left(\frac{x}{2}\right)y &= 10x^2y \end{aligned}$$

Definition 2 *[Word-level inconsistent terms:]*
Two terms of the moment expansion of a function, f, are inconsistent at the word level iff their word-level abstracts differ only at the coefficients. □

Thus two terms with word-level abstracts $4xyz$ and $2xzy$ respectively are inconsistent, but those with word-level abstracts $4xyz$ and $2xz$ are not.

Definition 3 *[Word-level hypothesis:]*
If the moment expansion of a function, f, contains word-level inconsistent terms, or terms whose word-level abstracts are nonlinear (such as $5x^2y$), then it's word-level hypothesis is null. Otherwise, the word-level function obtained by replacing each set of terms having the same word-level abstract by the word-level abstract, is called the word-level hypothesis of f. □

It is easy to see that the moment expansion for r shown earlier has the word-level hypothesis: $r = xy + 2yz$.

Theorem 1 *If f is a word-level linear arithmetic function, then it's moment expansion has a word-level hypothesis which is identical to f.*

Proof: If f is a word-level arithmetic function, then it can be written in a canonical sum-of-products form at the word level, where no term is higher order and no two terms differ only at the coefficient. Each of these word-level terms contribute a unique set of terms in the moment expansion of f. For example, a word-level term $10xyz$ will contribute terms of the form $10(2^i x[i])(2^j y[j])(2^k z[k])$. Clearly, the word-level abstracts of each of these terms will be $10xyz$. Thus, each term at the word-level contributes a set of distinct terms in the moment expansion, whose word-level abstracts are identical to the word-level term. The result follows. □

Theorem 1 has an important corollary.

Corollary 1 *Two distinct word-level linear arithmetic functions cannot have the same word-level hypothesis.* □

The reverse of Theorem 1, however, is not true in general. Even if a function has a word-level hypothesis, it is not necessarily a word-level linear arithmetic function. For example, consider the function, r', defined as follows:

$$\begin{aligned} r' \;=\; & x[0]y[0] + 2z[0]y[0] + 2x[1]y[0] + 4z[1]y[0] \\ & + 2x[0]y[1] + 4z[0]y[1] + 8z[1]y[1] \end{aligned}$$

This function differs from the function, r, whose moment expansion was shown earlier, though r' has the same word-level hypothesis as that of r. By Corollary 1, r' is not a word-level linear arithmetic function.

A *BMD representation of the function allows us to find the word-level hypothesis (if it exists) without actually constructing the moment expansion of the function. Fig 3 is

```
Algorithm Hypothesize(node: $n$)
  Let $x[i]$ be the variable which labels node n. Let l and r
  denote the left and right children of n. Let $w_l$ and $w_r$ be
  the weights of the left and right edges respectively. $f_n$
  denotes the word-level hypothesis at n. $visited[n]$ is a
  flag which is set when the node n is first visited
  1. If $visited[n]$ is true, return $f_n$.
  2. Determine $f_r$, the word-level hypothesis at $r$:
      2.1 If $r$ is a terminal node, then
          $f_r$ is the constant labeling $r$.
      2.2 Otherwise, determine $f_r$ using Hypothesize(r)
  3. If $f_r$ contains any term involving $x$, then report
      that no word-level hypothesis exists and Exit.
  4. Create a word-level function, $f_r'$, from $f_r$ by
      multiplying each term of $f_r$ by $2^{-i}xw_r$.
  5. Determine $f_l$, the word-level hypothesis at $l$:
      5.1 If $l$ is a terminal node, then
          $f_l$ is the constant labeling $l$.
      5.2 Otherwise, determine $f_l$ using Hypothesize(l)
  6. Create a word-level function, $f_l'$, from $f_l$ by
      multiplying each term of $f_l$ by $w_l$.
  7. If any term of $f_r'$ is inconsistent with any term of $f_l'$,
      report that no word-level hypothesis exists and Exit.
  8. Create $f_n$ as the sum of each distinct product term
      from the set of terms of $f_r'$ and $f_l'$
  9. Set $visited[n]$ to True and Return $f_n$.
End.
```

Figure 3. Algorithm Hypothesize

an outline of the algorithm for creating the hypothesis at a node, n, of a *BMD. The word-level hypothesis at each node of the *BMD of Fig 2 is shown beside the nodes. We assume that the left edge leads to the negative co-factor and the right edge leads to the linear moment.

3.2 Outline of the abstraction algorithm

```
Algorithm Word-Level-Abstract(component: $C$)
  For each output word $v$ of $C$:
  1. Create a *BMD for $v$
  2. Use Algorithm Hypothesize to create a
      word-level hypothesis for $v$
  3. If such a hypothesis, $f$, exists for $v$, then
      3.1 Create a *BMD for $f$.
      3.2 Verify whether the *BMD for $f$ is identical
          to the *BMD for $v$. If they are identical then
          return the word-level hypothesis, $f$
  4. Otherwise, report that $v$ is not word-level linear.
End
```

Theorem 2 *Algorithm Word-Level-Abstract is correct and complete. It returns a word-level function, f, for an output word, v, of a given component, C, iff there exists any such f which is a word-level linear arithmetic representation of v.*

Proof: Correctness is guaranteed by Step 3 of the algorithm, where we verify the equivalence between f and v before returning f. Completeness follows from Theorem 1. □

4 Implementation

We have developed a package called CDDP for *BMDs and other decision diagrams. The interface of the package is similar to that of the *kbdd* and *boole* packages developed at the Carnegie Mellon University, USA. CDDP supports bit-level logical operations such as & (AND), | (OR), and ˆ (EXOR), as well as word-level arithmetic operations such as $+$ and $*$. Boolean variables are treated as one-bit words.

Using the CDDP package we can construct the *BMD for an output word, v, of a given component. In order to determine the word-level hypothesis of v, one can use the command: `hypothesize v`.

Table 1. Results for 16-bit and 32-bit functions

	16-bit words			32-bit words		
⟨ #reg, order ⟩	BMD size (KB)	Cons. time (ms)	Hyp. time (ms)	BMD size (KB)	Cons. time (ms)	Hyp. time (ms)
⟨ 1,1 ⟩	2	2	1	5	2	1
⟨ 2,1 ⟩	6	2	1	13	4	1
⟨ 2,2 ⟩	6	2	2	12	3	6
⟨ 3,1 ⟩	11	3	2	22	5	2
⟨ 3,2 ⟩	14	3	6	28	8	12
⟨ 3,3 ⟩	11	6	22	22	16	173
⟨ 4,1 ⟩	15	3	1	31	7	1
⟨ 4,2 ⟩	20	6	9	42	11	21
⟨ 4,3 ⟩	31	17	84	64	69	511
⟨ 4,4 ⟩	17	40	359	34	312	5557
⟨ 5,1 ⟩	19	3	1	43	10	3
⟨ 5,2 ⟩	24	6	8	49	14	33
⟨ 5,3 ⟩	32	15	96	64	44	495
⟨ 5,4 ⟩	17	43	357	69	661	11095
⟨ 5,5 ⟩	24	641	5651	49	10344	184914

Table 1 shows the results of using CDDP for word-level functions of different types. The first column describes the nature of the linear function. In the tuple $\langle \#reg, order \rangle$, the first entry specifies the number of registers in the function. The second entry (that is, *order*) specifies the maximum number of registers in a product term (in the SOP form of the function). We studied the memory usage, BMD construction time, and time required by Hypothesize. Since the set of variables is the same, the second step of abstraction (namely equivalence checking) requires constant time (negligible). The second, third and fourth columns report the figures for 16-bit word-level functions and the fifth, sixth and seventh columns report the figures for 32-bit word-level functions. The results were taken on a Digital Alpha 500/333 workstation with 333 MHz clock and 128MB

RAM.

The results show that Hypothesize can efficiently abstract word-level functions from bit-level component descriptions. Incorporating hypothesize and abstraction in more sophisticated BMD packages is likely to further improve the space and time requirements.

5 Component matching

In this section, we first show that the component matching problem for word-level linear arithmetic functions is NP-hard in general. We then show that even in the special case where we require a one-to-one correspondence between the words of the function and the words of the component, the problem is not easy, since the graph isomorphism problem reduces to this special case[2].

Theorem 3 *The component matching problem for word-level linear arithmetic functions is NP-hard.*
Proof: We reduce the integer set partition problem to this problem. The integer set partition problem requires us to find a partition P of a given set $A = \{a_1, a_2, \ldots, a_m\}$ of m positive integers, such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$. In other words, if S denotes the sum of all the integers in A, then the task is to identify a partition P whose sum is $S/2$. The integer set partition problem is known to be NP-complete [4].

Given an instance, $A = \{a_1, a_2, \ldots, a_m\}$, of the integer set partition problem, we create two linear arithmetic functions, namely $f = a_1x_1 + a_2x_2 + \cdots + a_mx_m$ and $g = (S/2)y_1 + (S/2)y_2$, where S is the sum of all integers in A. Clearly, a valid partition exists iff we can define a many-to-one mapping from the set of words $x_1, \ldots, x_m$ of f to the pair of words y_1, y_2 of g, such that the sum of the coefficients of the x_i which are mapped to y_1 is $S/2$, and the sum of the remaining coefficients (corresponding to the remaining words of f) also add up to $S/2$.

It is easy to write a component, C, implementing f in polynomial time by considering some constant dimension for the words $x_1, \ldots, x_m$. Then a solution to the integer set partition problem exists iff a solution to the problem of matching the function g with C exists. The result follows. □

Theorem 4 GRAPH ISOMORPHISM $\propto$ COMPONENT MATCHING
Proof: Consider an instance of the graph isomorphism problem, where $G_a = (V_a, E_a)$ and $G_b = (V_b, E_b)$ are the given graphs. Obviously, $|V_a| = |V_b|$ and $|E_a| = |E_b|$, otherwise the graphs are clearly not isomorphic.

We construct a word-level linear arithmetic function, f_a from G_a as follows. For each edge $(v_i, v_j) \in E_a$, we construct a word-level term v_iv_j. f_a is defined as the sum of these terms. For example, if $E_a = \{(v_1,v_3),(v_1,v_4),(v_3,v_4),(v_2,v_3),(v_2,v_4)\}$, then $f_a = v_1v_3 + v_1v_4 + v_3v_4 + v_2v_3 + v_2v_4$. Clearly, we can construct f_a in $O(|E_a|)$ time. We construct f_b from G_b in a similar way.

If the graphs are isomorphic, then there will exist a bijection, $\mathcal{F}$ from V_a to V_b, such that for each i and j, $(v_i, v_j) \in E_a$ iff $(\mathcal{F}(v_i), \mathcal{F}(v_j)) \in E_b$. It is easy to see that by replacing each v_i in f_a by $\mathcal{F}(v_i)$, we get f_b. Thus if the graphs are isomorphic, then f_b matches f_a.

It is easy to see that f_a and f_b are in canonical sum-of-products form. Thus, f_b matches f_a only if there is a bijection from the words in f_a to the words in f_b which unifies the two functions. Clearly, such a bijection shows that G_a and G_b are isomorphic.

Thus, G_a and G_b are isomorphic if and only if f_b matches f_a. Next we construct an equivalent component description, C, from f_b. This is possible in time polynomial in the length of f_b, if we consider some constant dimension for each word in f_b.

Clearly, G_a and G_b are isomorphic iff the component C matches f_a. □

Theorem 4 shows that no known polytime algorithm can be used for the component matching problem even when a one-to-one correspondence is sought. However, the number of words in a typical component is small compared to the number of bit-level variables taken together. Therefore the better option appears to be to do word-level abstraction followed by component matching at the word-level, rather than attempting component matching at the bit-level.

Acknowledgements

The authors acknowledge Synopsys (India) for partial support of this work. P. P. Chakrabarti acknowledges the Dept. of Sci & Tech, Govt. of India for partial support of this work. Pallab Dasgupta acknowledges the Indian National Science Academy for partial support of this work.

References

[1] Bryant, R.E., Symbolic Boolean manipulation with ordered binary decision diagrams, *ACM Computing Surveys*, 24, 3, 293-318, 1992.

[2] Bryant, R.E., and Chen, Y.A., Verification of arithmetic circuits with binary moment diagrams, In *Proceedings of 32th DAC*, 535-541, 1995.

[3] Chen, Y.A., and Bryant, R., ACV: An Arithmetic Circuit Verifier, In Proceedings of ICCAD'96, 361-365, 1996.

[4] Garey, M.R., and Johnson, D.S., *Computers and Intractability: A guide to the theory of NP-Completeness*, New York Press, 1979.

[5] Malik, S., Wang, A.R., Brayton, R., and S-Vincentelli, A., Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of ICCAD*, 6-9, 1988.

[6] Papadimitriou, C.H., *Computational Complexity*, Addison-Wesley, 1994.

[7] Smith, J., and Micheli, G.D., Polynomial Methods for Component Matching and Verification. In *Proceedings of ICCAD'98*, 1998.

[2] Curiously, the graph isomorphism problem is neither known to be in P, nor known to be NP-Complete, even after several decades of research[4, 6]