Using Symbolic Algebra in Algorithmic Level DSP Synthesis

Armita Peymandoust Giovanni De Micheli

Stanford University
Computer Systems Laboratory
Stanford, CA 94305

{armita, nanni}@stanford.edu

ABSTRACT

Current multimedia applications require the design of data-path intensive circuits. Unfortunately, current design tools and methods support design abstraction at a level that is inferior to the expectation of designers. Namely, most arithmetic-level optimizations are not supported and they are left to the designers' ingenuity. In this paper, we show how symbolic algebra can be used to construct an arithmetic-level decomposition algorithm. We also introduce our tool, SymSyn, that performs arithmetic library mapping and optimization of data-flow descriptions into data paths using arithmetic components.

1. INTRODUCTION

The growing market of multi-media applications has required the development of complex ASICs with significant data-path portions. Automating the design of data paths from high-level specifications is necessary to meet time to market requirements. The optimal choice of the arithmetic units implementing complex data flows affects strongly the cost, performance and power consumption of the silicon implementations. Unfortunately, most high-level synthesis tools and methods cannot synthesize data paths that intelligently use arithmetic libraries without synthesis directives (*pragmas*).

On the other hand, current high-level synthesis tools are effective in capturing HDL models of the circuits and mapping them into control/data-flow graphs (CDFGs), performing scheduling, resource sharing retiming, and control synthesis [1]. The approach presented in this paper fits seamlessly into the current high-level synthesis flow. We propose to analyze the data-flow segments of the CDFG models in light of the arithmetic units available as library blocks, and to construct data paths that exploit at best the given library. We assume that design is done using libraries that contain, beyond the basic elements such as adders and multipliers, more complex cells such as multiply/accumulate (MAC), sine, cosine, ... An example of such a library is Synopsys Designware [2] library.

Two factors are key to automate the optimal mapping of data flow blocks. First, a common functionality description formalism for data flow and library components. Second, a method supporting the decomposition of the data flow into a set of library elements. The functionality description formalism needs to be compact, canonical, and unambiguous. Polynomial representations have been shown to be effective for data flow representation and for supporting matching of data flow clusters to library cells [3, 4]. Unfortunately, such methods were limited to test for a match in the library of existing components. In case a match did not exist, there was no automated way to search for possible interconnections of library blocks matching the data flow cluster.

In this paper, we propose a decomposition method based on symbolic manipulation of polynomials. We leverage results from symbolic algebra to construct an algorithm that finds a minimal-component decomposition of a polynomial representing a (portion of) data-flow. The decomposition is done in terms of arithmetic library elements, also represented as polynomials.

Moreover, we relax the assumption that a bit-level implementation should realize exactly the given specification. In other word, we allow for some tolerance in the decomposition and matching process. To a certain extent, the tolerated error can be seen as an "arithmetic-level don't care" that can be spent toward achieving a low-cost implementation. Note that many multi-media applications are well suited for tolerating computational inaccuracy, as long as the resulting effects (e.g. audio, video degradation) are limited.

As a motivating example, we consider an antialias function of a MP3 decoder that has the following equation in one basic block:

$$z = \frac{1}{2\sqrt{x^2 + y^2}}$$

A straightforward realization of this equation would use a divider and a square root operator, which are large and slow components and may not be available in the component library. For the sake of the example, we assume there is no square root and division in our library. Alternatively, we assume the existence of adder, multiplier and multiplier-accumulator (MAC) in our library. Thus the computation $c=x^2+y^2$ can be easily done. Next, using symbolic manipulations we first substitute x^2+y^2 by c. We obtain:

$$z = \frac{1}{2\sqrt{c}}$$

We can approximate the given equation to a polynomial representation for a certain range of c using Taylor series expansion.

$$z \cong \frac{1}{64}c^6 - \frac{9}{32}c^5 + \frac{115}{64}c^4 - \frac{75}{16}c^3 + \frac{279}{64}c^2 - \frac{81}{32}c + \frac{85}{64}$$

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

The error can be easily computed using standard approximation methods [11]. If we perform a Horner based transform on the polynomial approximation of z, we obtain:

$$z \cong \frac{85}{64} + \left(-\frac{81}{32} + \left(\frac{279}{64} + \left(-\frac{75}{16} + \left(\frac{115}{64} + \left(-\frac{9}{32} + \frac{1}{64} c \right) c \right) c \right) c \right) c \right) c$$

The formula given above can be implemented using a chain of 5 MACs, or one MAC in 5 cycles. Figure 1 demonstrates one possible implementation. Note that a1, a2, a3, a4, a5, and a6 are the constants in the formula shown above.

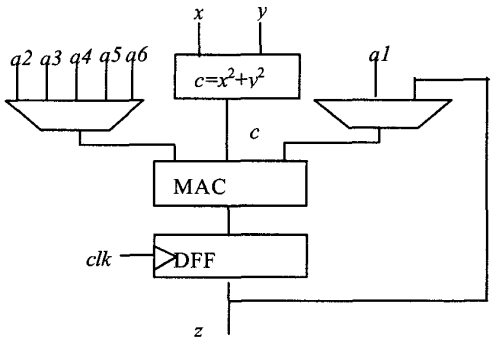

Figure 1. An implementation for $\frac{1}{\sqrt{x^2+y^2}}$

The paper presents a synthesis tool, called SymSyn, that automates the algebraic manipulations shown in this example and presents the underlying theory. SymSyn converts the basic blocks of a behavioral description, representing data-flow portions of the design to their polynomial representations and uses numerical methods for exact and inexact matching with library elements. If a match is not found, the data flow is decomposed into the library elements using symbolic computer algebra.

The paper is organized as follows: Section 2 gives an overview on symbolic algebra and explains how Gröbner basis is used in polynomial decomposition algorithms. In Section 3, we present how we can leverage results from symbolic algebra to construct an algorithm that finds a minimal-component decomposition of a polynomial representing a (portion of) data flow. In Section 3, we also explain our data flow synthesis tool, SymSyn, with an example. Finally, Section 4 presents some experimental results.

2. SYMBOLIC COMPUTER ALGEBRA

Traditional mathematical computation with computers and calculators is based on arithmetic of fixed-length integers and fixed-precision floating-point numbers, otherwise known as numeric computer algebra. Such system does not allow manipulations of undetermined quantities (symbolic manipulation), such as variable x in $(x+1)*(x-1)$. In contrast, modern symbolic computation systems support exact rational arithmetic, arbitrary-precision floating-point arithmetic, and algebraic manipulation of expression containing undetermined values (symbols).

The algebraic object that we would like to manipulate symbolically is a multivariate polynomial that represents a (portion of) data path of our design. We need to decompose this polynomial into polynomials of the building blocks in the target library. Such decomposition is called *simplification modulo set of polynomials* and uses Gröbner basis in symbolic computer algebra. In the following subsection we will review Gröbner basis [6, 9] and its application to the *simplification* algorithm. Commercial symbolic computer programs, such as Maple [7], have a built-in routine that performs *simplification modulo set of polynomials*. In Maple this method is called *simplify*.

We describe next the underlying theory of *simplification modulo set of polynomials*. The reader solely interested in its application to data path synthesis may proceed to Section 2.2.

2.1 Gröbner Bases

Let R be a commutative ring, a non-empty subset $I \subseteq R$ is an ***ideal*** when [6, 9]:

1. $p + q \in I$ for all $p, q \in I$, and
2. $r \cdot p \in I$ for all $p \in I$ and $r \in R$.

We denote $R[\mathbf{x}] = R[x_1, x_2, \ldots, x_n]$ as the ring of all multivariate polynomials with variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Every finite set of polynomials $P = \{ p_1, p_2, \ldots, p_k \} \subset R[x_1, x_2, \ldots, x_n]$ generates an ideal

$$< P > = \{ \sum_{i=1}^{k} a_i \cdot p_i \mid a_i \in R[x_1, x_2, \ldots, x_n] \}.$$

The set P is called a ***basis*** for this ideal. For example, the set of polynomials $P = \{ p_1, p_2, p_3 \}$ defined below generates a polynomial ideal over $R[x_1, x_2, x_3]$.

$p_1 = x_1^3 x_2 x_3 - x_1 x_3^2$, $p_2 = x_1 x_2^2 x_3 - x_1 x_2 x_3$, $p_3 = x_1^2 x_2^2 - x_3^2$

$< P > = \{ a_1 \cdot p_1 + a_2 \cdot p_2 + a_3 \cdot p_3 \mid a_1, a_2, a_3 \in R[x_1, x_2, x_3] \}$.

Unfortunately, while P generates the infinite set $<P>$, the polynomials p_i in P may not yield much insight into the nature of this ideal. However, Buchberger [8] has shown that an arbitrary ideal basis can be transformed into a basis with special properties, which is called the *Gröbner basis*. We will now give a brief description of Buchberger's algorithm.

A monomial of the form $x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$, where $x_1, x_2, \ldots, x_n$ are the variables of the polynomial and $i_1, i_2, \ldots, i_n$ are non-negative integers, is called a ***term***. We denote the set of terms of the polynomial ring $R[\mathbf{x}]$ by $T_{\mathbf{x}}$, where $\mathbf{N}$ is the set of non-negative integers:

$$T_{\mathbf{x}} = \{ x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} \mid i_1, i_2, \ldots, i_n \in \mathbf{N} \}.$$

The *leading monomial* of polynomial $p \in R[\mathbf{x}]$ with respect to a total ordering of the variables, such as the lexicographical ordering, is the monomial in p whose term is the maximal among those in p; we denote this monomial by $M(p)$. We also define $\mathrm{hterm}(p)$ to be the maximal term, and the $\mathrm{hcoeff}(p)$ to be the corresponding coefficient, therefore

$M(p) = \mathrm{hcoeff}(p) \cdot \mathrm{hterm}(p)$.

As an example consider $p \in R[x_1, x_2]$ that is written in lexicographical order:

$p = 3x_1^2 x_2 + 5x_1^2 + x_2^2$,

$M(p) = x_1^2 x_2$, $\mathrm{hterm}(p) = x_1^2 x_2$, $\mathrm{hcoeff}(p) = 3$.

For nonzero $p, q \in R[\mathbf{x}]$ we say that p reduces modulo q if there exists a monomial in p which is divisible by $\mathrm{hterm}(q)$. Let

$\alpha \in$ R[**x**]-{0}, i.e. the ring of polynomials after removing the trivial 0 polynomial. If $p = \alpha \cdot t + r$ where $t \in T_x$, $r \in$ R[**x**], and $u = \frac{t}{\text{hterm}(q)}$, $u \in T_x$, then we write $p \rightarrow_q p'$ to signify that p reduces to p' (modulo q) and p' is equal to:

$$p' = p - \frac{\alpha t}{\text{M}(q)} \cdot q = p - \frac{\alpha}{\text{hcoeff}(q)} u \cdot q$$

For example, we have:

$p = 6x^4+13x^3-6x+1$,

$q = 3x^2+5x-1$,

$p \rightarrow_q p'$; $p' = p - 2x^2 \cdot q = 3x^3+2x^2-6x+1$.

If p reduces to p' modulo a polynomial in a set of polynomials $Q = \{q_1, q_2, \ldots, q_n\}$, we say that p reduces modulo Q and write $p \rightarrow_Q p'$ (p' = Reduce(p,Q)); otherwise we say that p is irreducible modulo Q. We denote, $p \rightarrow^+_Q q$ if and only if there is a sequence such that:

$p = p_0 \rightarrow_Q p_1 \rightarrow_Q \cdots \rightarrow_Q p_n = q$.

Algorithm 2.1 Full Reduction of p Modulo Q.

```
procedure Reduce(p, Q)
  # Given a polynomial p and a set of polynomials Q
  # from the ring R[x], find a q such that p →*_Q q.
  # Start with the whole polynomial.
  r ← p; q ← 0
  # if no reducers exist, strip off the leading
  # monomial; otherwise, continue to reduce.
  while r ≠ 0 do{
    R ← R_{r,Q}
    while R ≠ ∅ do{
      f ← select a polynomial ∈ R
      R ← R −{f}
      r ← r − (M(r)/M(f)) f
    }
    q ← q +M(r); r ← r − M(r)
  }
  return(q)
end
```

If $p \rightarrow^+_Q q$ and q is irreducible, we will write $p \rightarrow^*_Q q$. It can be shown that for a fixed set Q and a given term ordering, the sequence of reductions is finite [9]. Therefore, we may construct Algorithm 2.1 which, given a polynomial p and set Q, finds a polynomial q such that $p \rightarrow^*_Q q$. In Algorithm 2.1, $R_{p,Q}$ denotes the set polynomials in Q-{0} that hterm(p) is divisible by hterm(q). Note that any member of $R_{p,Q}$ can be chosen in each iteration, but this choice affects the efficiency of the algorithm. For the sake of simplicity, we assume an efficient selection is implemented in *selectpoly*.

As mentioned previously any finite set of polynomials Q generates an ideal $\langle Q \rangle$ and Q is called the basis of this ideal. If a nonzero polynomial p is reduced to zero modulo Q, we can determine that p is a member of the ideal generated by Q:

$p \rightarrow^*_Q 0 \Rightarrow p \in \langle Q \rangle$.

However the converse is not true for all basis of $\langle Q \rangle$.

Definition: An ideal basis $G \subset$ R[**x**] is called a *Gröbner bases* (with respect to a fixed term ordering and the implied permutation of variables) when

$p \rightarrow^*_G 0 \Leftrightarrow p \in \langle G \rangle$.

We define the *S-polynomial* of $p, q \in$ R[**x**] as:

$$\text{Spoly}(p,q) = \text{LCM}(\text{M}(p), \text{M}(q)) \cdot [\frac{p}{\text{M}(p)} - \frac{q}{\text{M}(q)}].$$

In can be shown that [6, 9], G is a Gröbner basis when:

1. the only irreducible polynomial in $\langle G \rangle$ is $p = 0$;
2. Spoly(p, q) $\rightarrow^+_G 0$ for all $p, q \in G$;
3. if $p \rightarrow^*_G q$ and $p \rightarrow^*_G r$, then q = r.

Buchberger's algorithm (Algorithm 2.2) uses the properties above to convert a finite set $Q \subset$ R[**x**] into a Gröbner basis [8].

Algorithm 2.2 Buchberger's Algorithm for Gröbner Bases.

```
procedure Gbasis(Q)
  # Given a set of polynomials Q, compute G such
  # that <G> = <Q> and G is a Gröbner basis.

  G ← Q; k ← length(G)
  # We denote the i-th element of the ordered set G by G_i
  B ← {[i, j] : 1 ≤ i < j ≤ k}

  while B ≠ ∅ do {
    [i, j] ← select a pair from B
    B ← B − {[i, j]}
    h ← Reduce(Spoly(G_i, G_j), G)
    if h ≠ 0 then {
      G ← G ∪ {h}; k ← k + 1
      B ← B ∪ { (i, k) : 1 ≤ i < k} }}
  return (G)
end
```

In order to check whether a polynomial p is a member of the ideal $\langle Q \rangle$, we would first use Algorithm 2.2 to form G a Gröbner basis for $\langle Q \rangle$. Next, using Algorithm 2.1, we check whether Reduce(p, G) returns zero.

2.2 Gröbner Bases and Data-path Synthesis

We now describe the application of the theory described previously. Let L be the set of polynomial representations of the library elements. In order to synthesize a data path for a polynomial representation S using library L, S should be a member of $\langle L \rangle$. In order to examine membership in $\langle L \rangle$, we need to calculate G the Gröbner basis of $\langle L \rangle$ and use Reduce(S, G). If S reduces to zero then $S \in \langle L \rangle$. If S is reduced to zero only using polynomials in G that are also in L, then S can be built from the given library elements. As an example, consider:

$S = x + x^2 + x^3 + y + xy + x^2y$;

$L = \{1+x+x^2, x+y\}$;

$G =$ Gbasis(L) = $\{x+y, y^2-y+1\}$;

Reduce(S, G) returns zero, therefore $S \in \langle L \rangle$.

While performing Reduce(S, G), we determine that:

$S = (x+y)(1+x+x^2)$;

therefore S can be decomposed into elements of $\langle L \rangle$.

3. DECOMPOSITION ALGORITHM

Here we introduce a new algorithm that automatically maps a polynomial representation of a data flow to a set of complex arithmetic components. This algorithm in conjunction with classical high-level synthesis algorithms can be used for efficient high-level DSP synthesis. This algorithm is empowered by Gröbner basis fundamentals described in the previous section.

3.1 Symbolic Algebra and Library Matching

After extracting the CDFG of an algorithmic level DSP model, we use symbolic computer algebra to intelligently decompose the data flow to library components and synthesize the data path. The symbolic algebra routine used in this algorithm is *simplification modulo set of polynomials* that has been described in Section 2. As a reminder, to simplify a polynomial p modulo the side relation set L, we build a Gröbner basis from L, $G \leftarrow$Gbasis(L), and use Reduce(p, G) to obtain the simplified answer. The built-in function that implements *simplification modulo set of polynomials* in Maple is called *simplify* [7]. In order to comply with Maple terminology, we call the set of polynomials the *side relations*.

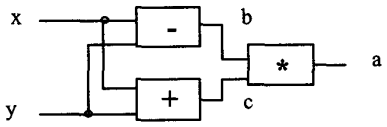

Figure 2. An implementation of x^2-y^2

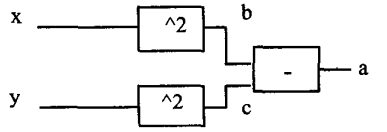

Figure 3. Another implementation of x^2-y^2

Note that any polynomial representation can be implemented using only adders and multipliers. Therefore, an implementation is guaranteed if the library includes adder and multiplier. Our goal is to find non-trivial solutions that are minimal in terms of component count. As an example, consider a data flow implementing `x^2-y^2` and a library that includes add, multiply subtract and square functions. Using Maple syntax we have:

```
>  a:=x^2-y^2: siderels:={b=x-y, c=x+y}
>  simplify(a, siderels, [x,y,b,c]);
>  b*c
```

This is equivalent to the implementation shown in Figure 2. Note that `siderels` is a subset of our library. Maple computes the Gröbner basis G of `siderels` and prints out the result of the Reduce(`a, siderels`). The result indicates that:

```
a:=x^2-y^2:=b*c:=(x-y)*(x+y)
```

If the side relation set is changed, other possible solutions for the specification might be found, for example:

```
>  a:=x^2-y^2: siderels:={b=x^2, c=y^2}
>  simplify(a, siderels, [x,y,b,c]);
>  b-c
```

results in the implementation shown in Figure 3.

As shown, different side relation sets can result in different implementation of the specification. Therefore, to find the best possible implementation, the side relation set should be set equal to all subsets of the library. Since this is exponentially expensive, a guided architectural exploration is necessary. Algorithm 3.1 gives a high level view to the heuristic used to bound the complexity of this search.

Algorithm 3.1 Decompose S into elements of library L
procedure Decompose(S, L)
  # Given a polynomial representation of the spec S
  # and a set of polynomials L as component library,
  # decompose S into elements of library L.
  # initialize tree
  treeroot(S);
  depth ← 0
  bound ← -1
  while depth ≠ bound **do** {
    bound ← **Explore**(S, L, *depth*)
    depth ← *depth* +1
  }
  report best solution in tree
end

used in Decompose procedure
int function Explore(S, L, d)
  bound ← -1
  for all $n \in$ in tree with depth d **do**{
    for all $sr \in L$ **do**{
      result = simplify(n, sr);
      # make *result* a child of node n
      addchild(n, *result*);
      if *result* $\in L$
      # solution is found
      bound = treedepth(*result*); }}
  # returns –1 if no solution is found yet.
  return(*bound*)
end

Let S be the polynomial representation of the data flow. We start by simplifying S modulo each library element as the side relation. We store the simplification results in a tree data structure. If a simplification result is identical (or within an acceptable tolerance) to the polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. If the simplification result stored in a tree node does not correspond to a library element, we recursively apply the same steps to the new tree node.

To further reduce the search space a bounding function is used. The bounding function is the number of library components used to build the specification. In other words, if we find a solution with two library components we will not explore solutions requiring more than two components. But we will uncover all two-component solutions and choose the one with optimal cost (area or delay). The number of components used is the same as the depth of the simplification tree; therefore the tree is bounded by the depth of the first solution found.

Such bounding function is chosen assuming that if a component is custom designed to perform a combination of arithmetic operations, it is more cost effective than connecting a series of components that perform the same arithmetic operations. Clearly, the merit of the result is strongly dependent on the available library.

3.2 Implementation and Example

Algorithm 3.1 is implemented in our tool SymSyn using C programming language and calls to Maple V [7] for the symbolic manipulations. To clarify the algorithm described above, we choose our library to be a subset of the DesignWare library consisting of six combinational elements; multiplier, adder, subtracter, multiplier-accumulator, sine, and cosine. As an example, consider synthesizing a phase shift keying (PSK) modulator used in digital communication. A data-flow segment of PSK has the following polynomial representation (S):

```
> S:=1-.5*x0^2-x0*x1-.5*x1^2+
  .041667*x0^4+.166668*x0^3*x1+
  .250002*x0^2*x1^2+.166668*x0*x1^3+
  .041667*x1^4;
```

As the first step, SymSyn initializes a tree data structure and stores polynomial S in the root of the tree. For all library elements, SymSyn makes a call to Maple and requests *simplify* with side relation set equal to the library element. The results reported by Maple are kept as new children of the S tree node.

In the first iteration of our example the side relation is set to the first element in the library, the multiplier. Shown below are the Maple commands. The first two lines are the requests sent by SymSyn and the third line is the simplification result reported by Maple to SymSyn. SymSyn searches for a component in the library that implements the result, but it is not successful to find one for this instance.

```
> siderel := {y=x0*x1};
> simplify(S, siderel, [x0,x1,y]);
> .041667*x0^4+.166668*x0^2*y-
  .5*x0^2+.041667*x1^4+.166668*x1^2*y-
  .5*x1^2+.250002*y^2-1.*y+1.
```

In the second iteration, the same steps are performed with the adder as the side relation. The simplification result now matches an approximation to the cosine function. Therefore, SymSyn marks this node as one possible solution. The following Maple commands show the result of this iteration. Note that the result is a Taylor series approximation of cosine. Since cosine is one of our library elements, we have found one possible solution, Figure 4.

```
> siderel := {y=x0+x1};
> simplify(S, siderel, [x0,x1,y]);
> 1.+.041667*y^4-.5*y^2
```

Since there is a solution with depth equal to *one* in the tree, a bound of *one* is set on the tree growth. SymSyn performs the steps described above for the rest of library elements and keeps the results in root offsprings. After going through all library elements, SymSyn finds only one solution using two components. The solution is demonstrated in Figure 4. SymSyn will stop decomposing the leaf nodes, since continuation would result in a search for solutions with three or more components while the objective is to find a solution using minimal number of components.

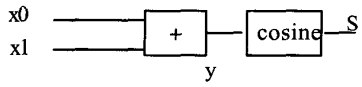

Figure 4. Mapping S to two components

4. EXPERIMENTAL RESULTS

SymSyn implements Algorithm 3.1 described in this paper in C programming language with calls to Maple V [7] for the symbolic manipulations. The program input is polynomial representation of data flow and a database of polynomial representations of library elements. Output reported is components used to implement the data flow and the way they are connected.

Table 1. SymSyn results for some examples

Data flow Examples	Lexicographical Mapping		SymSyn Output	
	#of compo-nents	Cost	#of compo-nents	Cost
1-x0^2/2+ x0^4/24+x0+x1x2	11	14.5	3	9.7
Cos(sin(x0))	24	34	2	13
$X^2+2xyz+y^2z^2$	9	9.5	2	3.4
anti-alias	27	37.5	8	14.4
PSK	33	45.5	2	7.5
Turbo decoder	104	139.5	4	31.5

We have tested the efficiency of SymSyn with a number of data-path examples. The results are shown in Table I. In this table, the cost reported is normalized by the cost of an adder. For example, we assume that the cost of an adder is 1 and cost of a multiplier is 1.5. In the first set of results, we assume that the polynomial representation is mapped only to multipliers and adders. This is same as lexicographical component inference that is typical in commercial behavioral synthesis tools. The number of components refers to the numbers of adds and multiplies in the data-path polynomial. The cost is the cost of an adder multiplied by the number of adds, plus the cost of a multiplier multiplied by the number of multiplies in the data-path polynomial.

The second set of results is derived by SymSyn. The cost is sum of cost of the components used in data path to implement the polynomial representation as recommended by SymSyn. The library used for the examples is the DesignWare library [2], except for the turbo decoder that needs *ln(x)* and *exp(x)* operation not available in DesignWare.

The first two data flows in Table I are simple benchmark polynomials. The third polynomial is a basic block in a one-dimensional inverse discrete cosine transform (IDCT). The forth data flow is the anti-alias block described in the introduction. IDCT and anti-alias are widely used in portion in audio and video

compression standards such as JPEG, MPEG, and MP3. The last two examples come from the digital communication field. The fifth data flow performs phase shift keying (PSK) modulation.

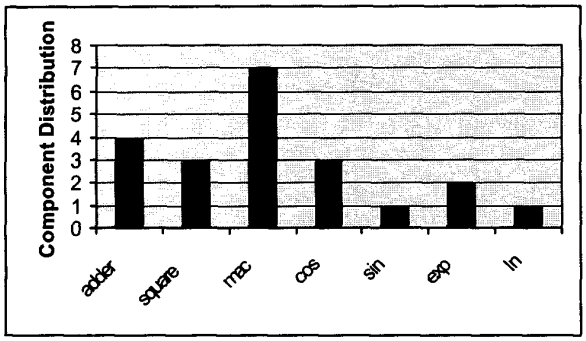

Figure 5. Component distribution in SymSyn output

In order to qualify the examples used in Table I, we have shown the distribution of components used in SymSyn output in Figure 5. Note that the component used mostly is the multiply/accumulate (MAC); this result is typical in data-intensive circuits.

Table 2. Area in library units using tsmc.35 library

Data flow Examples	BC Results without SymSyn	BC Results with SymSyn
1-x0^2/2+ x0^4/24+x0+x1x2	311502	143037
cos(sin(x0))	1065695	167584
$x^2+2xyz+y^2z^2$	456704	178177
anti-alias	1120542	317591
PSK	1614610	34271

With the intention of achieving more realistic area estimate for our set of examples, we used Behavioral Compiler to produce the set of results shown in Table 2. The fist column shows the combinational area of Behavioral Compiler results without any mapping directive. The second column shows the combinational area of the same examples, with mapping directives suggested by SymSyn incorporated in the HDL code. It can be observed that improvements are comparable or better than estimated results by SymSyn.

5. SUMMARY

This paper has introduced a decomposition algorithm to map data flow to a set of complex arithmetic library components. This algorithm fits seamlessly in the high-level synthesis flow and enhances the quality of result of data intensive circuit synthesis. Our method takes advantage of two previously developed concepts; one is the polynomial representation of library blocks and the second is symbolic computer algebra. Polynomial representation is used to represent the functionality of library components and the data flow segment of the chip under design. Symbolic computer algebra is used to decompose the data flow to a set of library components. From a practical standpoint, the contribution of this paper is to make arithmetic library binding an automated process, and eliminate the need for synthesis directives.

Symbolic computer algebra is a powerful set of algorithms not previously used in the field of synthesis. We believe these algorithms open a new set of opportunities in high-level synthesis research. Even though, algebraic manipulations are best suited for combinational arithmetic designs, classical scheduling, resource sharing, and retiming algorithms can be applied to the data-path output to achieve optimized/pipelined designs.

The research presented here is especially promising in the fields of graphics and digital signal processing where there is a tolerance for computational error as long as the degradation in audio or video is limited [10]. This tolerance can be used to approximate non-polynomials data flows to polynomial representations, which are well-suited inputs for our tool SymSyn. This paper does not explain the approximation tools and truncation errors since there is a wide body of mathematical literature available on these topics [11]. In future work, timing driven architectural exploration algorithms will be studied.

6. ACKNOWLEDGMENTS

This research is supported by ARPA/MARCO Gigascale Research Center and Synopsys Inc. We would like to thank both organizations for their support.

7. REFERENCES

[1] G. De Micheli, "Synthesis and Optimization of Digital Circuits", *Mc Graw Hill*, Hightstown, NJ, 1994.

[2] DesignWare Library, http://www.synopsys.com/, 1994.

[3] J. Smith and G. De Micheli, "Polynomial Methods for Component Matching and Verification", *Proceedings of the ICCAD*, pp. 678-685, 1998.

[4] J. Smith and G. De Micheli, "Polynomial Methods for Allocating Complex Components", *Proceedings of the DATE Conference*, pp. 382-383, 1999.

[5] R. Brayton and C. McMullen, "The decomposition and factorization of logic synthesis", *IEEE ISCS*, May 1982.

[6] T. Becker and V. Weispfenning, *Gröbner Bases*, Springer-Verlag New York, 1993.

[7] Maple V, Waterloo Maple Inc., http://www.maplesoft.com/, 1988.

[8] B. Buchberger, "Some Properties of Gröbner Bases for Polynomial Ideals", *ACM SIG-SAM Bulletin*, pp.19-24, 1976.

[9] K. Geddes, S. Czapor, and G. Labahn, *Algorithms for Computer Algebra*, Kluwer Academic Publishers, 1992.

[10] M. Willems, H. Keding, T. Grötket, and H. Meyr, "Fridge: An interactive Fixed-Point Code Generation Environment for HW/SW CoDesign", Proceedings of Int. Conf. On Acoustics, Speech, and Signal Processing 1997.

[11] J. F. Hart et al., "Computer Approximations", New York: Wiley, 1968.