A Methodology for Synthesis with Reusable Components from an Arithmetic Specification

James Smith
Stanford University
Computer Systems Laboratory
Stanford, CA 94305

Giovanni De Micheli
Stanford University
Computer Systems Laboratory
Stanford, CA 94305

Abstract

Reuse of complex blocks promises to allow designers to implement greater functionality in hardware and focus on system level trade-offs without extending design time. However, the availability of tools and methodologies that can construct systems out of complex blocks is limited. This paper introduces a methodology for design reuse based on polynomial representations for complex blocks. These techniques have been implemented in the POLYSYS *synthesis tool and applied to the synthesis of an antialiased line rasterizer with reusable components.*

1. Introduction

As transistor sizes shrink, the struggle to include greater functionality on a single chip while maintaining short design cycles has forced designers to consider, if not implement, design reuse. State of the art integrated circuits, which could be designed in a matter of months 5 years ago, are projected to take over 5 years by the year 2000, when over 15 million logic gates will be incorporated into a single die ([1]). Design reuse promises to shorten design cycles, but today's CAD methodologies are generally developed assuming logic gates as basic building blocks, not complex blocks. A gap exists between methodologies that are oriented around design with basic logic gates and the need to design systems with complex, reusable blocks.

The methodology presented here performs synthesis with complex blocks by determining an allocation of blocks that can implement a specification, optimizing this allocation through scheduling and resource sharing, and generating interfaces between blocks. This paper focuses on the allocation of reusable designs, a task which promises to become increasingly complex as design reuse becomes a more common practice and the libraries of potential implementations grow.

The techniques discussed in this paper, while applicable to control dominated systems, are ideally suited to synthesis of systems that require complex mathematical computations, such as those encountered in digital signal processing and graphics. A prime application of this methodology is mapping MATLAB-generated system specifications to existing components and complex, soft macro-cells.

The methodology presented here is geared to enable design reuse by characterizing the functionality of intellectual property with polynomials. System specification can then be performed at the arithmetic level of abstraction. This specification can be compared efficiently against a library of existing elements whose functionality is described with polynomial representations. For example, the functionality of a block that generates the impulse response for a low pass box filter is represented by the polynomial $1 - 1.64x^2 + 0.81x^4 - 0.19x^6$. Such representations can be computed from a Boolean description of the implementation ([2]).

2. Related Work

Reusable blocks have traditionally been characterized at a high level of abstraction by verbal descriptions, such as "ethernet core" or "rasterizer", or a at low level by HDLs or Boolean equations. The imprecision of high level descriptions prevents IP from being effectively reused. The complexity of precise, low level descriptions are restricts their application to reuse of small blocks such as combinational logic gates. Present methodologies employ structures such as BDDs ([3], Binary Moment Diagrams ([4], [5]), Hybrid Decision Diagrams ([6]), and Multi Terminal BDDs ([7]) to allocate basic logic gates and verify gate level designs.

In order to improve a designer's ability to explore potential implementations and gauge the performance of complete systems, Martin ([8]) outlines the need to move design modeling, exploration, and verification to a higher level of abstraction. This need is imposed by the prohibitively expensive operation of evaluating the performance and functionality of systems that are specified at low levels of abstraction.

The algorithms presented in [2] and [9] described a mechanism for deriving a word-level polynomial representation for a complex design given a Boolean circuit description. These representations were shown to exist for combinational and sequential circuits and were proven to be canonical with respect to circuit functionality. In addition, a mechanism was presented for modeling circuits that did not implement polynomial functions as polynomials. These algorithms were shown to be of polynomial complexity and were applied to synthesize a JPEG encode block from complex blocks and an IIR filter for controlling tape drives.

3. Polynomial Methods

The algorithms for computing polynomial representations are detailed in [2] and [9]. They are briefly reviewed here to provide a background for the following sections. A polynomial representation for a complex block can be determined by treating the Boolean description of the block, y = F(x), where x and y are bit vectors, as a collection of coordinates (x, y). These coordinates can then be fit to a minimum-order polynomial. If the order of this polynomial is known to be n, then n+1 coordinates can be extracted from the function and a set of n+1 equations and variables (the coefficients of the polynomial) can be constructed and solved. Thus, the problem of generating a word-level polynomial representation for a Boolean function reduces to determining the order of the polynomial.

The order of a function y = F(x) is exactly one greater than the polynomial generated by computing F(x+1) - F(x). The order of F(x) is therefore equivalent to the number of iterations of F(x+1) - F(x) that are required to generate an order zero polynomial. For example, the bit level Boolean function:

$$F_0(x) = x_0$$
$$F_1(x) = \quad x_1 \cdot x_0$$
$$F_2(x) = 0$$
$$F_3(x) = x_1$$
$$F_4(x) = \quad x_1 \cdot x_0$$

requires 3 computations of F(x+1) - F(x) to generate a zero order polynomial. Thus, the minimum order polynomial that represents this circuit is of order three and passes through the coordinates {(0, 0), (1, 1), (2, 8), (3, 27)}. The polynomial that satisfies these requirements is $y = x^3$. Polynomial representations can be computed for sequential circuits and those with feedback paths ([9]).

4. Specification

The methodology proposed here utilizes an HDL specification of the target system in which the synthesizeable subset of HDL constructs is expanded to include arithmetic operations such as division, exponentiation, and transcendental functions. This accommodates traditional HDL design flows, allows for fast simulation of complex arithmetic blocks, and provides a mechanism for reusing existing blocks. For example, the following Verilog model, with arithmetic extensions, specifies a block that performs antialiased line rasterization:

```
Antialias(init, x1, y1, x2, y2, x, y, yinc, ydec, I, Iinc, Idec)
begin
    input [7:0] x1, x2;
    input [7:0] y1, y2;
    output [7:0] x, y, yinc, ydec;
    output [7:0] I, Iinc, Idec;
    reg [7:0] dy, dx;
    reg [7:0] incrE, incrNE;
    reg [7:0] invDenom, twoVdx;

    dx = x2 - x1;
    dy = y2 - y1;
    incrE = 2*dy;
    incrNE = 2*dy - dx;
    invDenom = 1/(2*((dx^^2 + dy^^2)^^(1/2)));
    if (init) begin
        twoVdx = 0;
        {x, y} = {x1, y1};
    end else begin
        if (d<0) begin
            twoVdx = d + dx;
            d = d + incrE;
        end else begin
            twoVdx = d - dx;
            d = d + incrNE;
            y = y + 1;
        end
        x = x + 1;
    end
    yinc = y + 1;
    ydec = y - 1;
    I = twoVdx*invDenom;
    Iinc = 2*dx*invDenom - twoVdx*invDenom;
    Idec = 2*dx*invDenom + twoVdx*invDenom;
end
```

Arithmetic operators, such as ^^ (exponentiation) and / (division), are combined with traditional synthesizeable Verilog operators and statements, such as +, -, *, and "if", to allow arithmetic specification of complex blocks.

5. Library

The synthesis library includes traditional logic gates and complex elements for which a polynomial representation has been determined. A traditional logic gate is characterized by Boolean functionality, delay, and area. Complex blocks are characterized by several factors, including, but not limited to: (1) polynomial functionality, (2) domain over which each functionality equation is valid, (3) latency, (4) operational frequency, (5) area, (6) power, (7) precision, (8) input word width, and (9) output word width. Parameters that are used to compute the value of these equations are determined from the specification and can be either block specific or global. For example, a library element that is an 8 bit multiplier and is to be implemented in a process with minimum feature size *lambda* may be described as shown in Fig. 1. The argument *lambda* must be defined within a pragma statement.

6. Allocation

In allocating elements that can implement a specification, our methodology attempts to find complex blocks that match part or all of the specification's functionality. This task is complicated by several factors: (1) a function can generally be implemented in many different ways, (2) an existing design may implement only part of specification, and (3) the functionality of existing design may approximate, but not match exactly, the functionality of the specification. By representing the functionality of both the specification and the existing design with polynomials, unnecessary implementation details can be ignored, partial implementations can be detected algebraically, and approximation differences can be quantified.

6.1 Component Matching

Once polynomial representations have been determined for each system partition, the representation is compared against the functionality of library elements. This is performed by determining the maximum value of the difference between the polynomial representations of the specification partition and the library element. For an exact match, the differnce between these polynomials must be zero and the bit width of the implementation must be greater than or equal to that of the specification. For example, the following library element that describes a 12 bit adder can be used to implement the partition (*twoVdx* = *d* + *dx*) of the Antialias specification:

```
Adder12 {
        domain0 = 1;
        function0 = x + y;
        period = 20*lamba;
        latency = 1;
        precision = 0;
        area = 2e5*lambda^2;
        input_width = {12, 12};
        output_width = {13};
}
```

A library element that does not match a

```
Divider8 {
domain0 = [1, 2];
function0 = 384-128x;
domain1 = [3, 4];
function1 = 1-(x-2)/2+(x-2)^2/2^2-(x-2)^3/2^3;
domain2= [5, 8];
function2 = 1-(x-4)/4+(x-4)^2/4^2-(x-4)^3/4^3;
domain3 = [9, 16];
function3 = 1 - (x-8)/8+(x-8)^2/8^2-(x-8)^3/2^3;
domain4 = [17, 32];
function4 = 1- (x-16)/16+(x-16)^2/16^2-(x-16)^3/16^3;
domain5 = [33, 64];
function5 = 1- (x-32)/32+(x- 32)^2/32^2-(x-32)^3/32^3;
domain6 = [65, 128];
function6 = 1-(x-64)/64+(x-64)^2/2^2-(x-64)^3/64^3;
domain7 = [129, 255];
function7 = 1-(x-128)/128+(x-128)^2/128^2-(x-128)^3/128^3;
}

SquareRoot8 {
domain0 = 1;
function0 = 8 + (x-64)/16 - (x-64)^2/2^12 + (x-64)^3/2^19;
period = 10*lamba;
latency = 5;
precision = 2;
area = 4e6*lambda^2;
input_width = {8};
output_width = {8};
}

Adder8 {
domain0 = 1;
function0 = x + y;
period = 5*lamba;
latency = 1;
precision = 0;
area = 5e4*lambda^2;
input_width = {8, 8};
output_width = {9};
}

Multiplier8 {
domain0 = 1;
function0 = xy;
period = 5*lamba;
latency = 3;
power = 2*lambda;
precision = 0;
area = 1e6*lambda^2;
input_width = {8, 8};
output_width = {16};
}

BasicGates {
And
Or
Nand
Nor
}
```

Fig. 1 Library elements used to synthesize an antialiased line rasterizer.

specification partition may still be used to implement a subset of that partition. This can be achieved by composing the library element with other elements to implement the specification. Composition is performed efficiently using polynomials, as the polynomial representations of the elements in question can be composed using traditional algebra to match the polynomial representation of the specification. For example, consider the partition {*Idec = 2*dx*invDenom + twoVdx*invDenom*}. Comparing the specification for that partition against the polynomial representation for Adder12 yields a match under the conditions {x = *2*dx*invDenom*, y = *twoVdx*invDenom*}. These conditions can be satisfied by instantiating the Multipler8 library element in the design.

6.2 Inexact Matching

In allocating complex designs to implement a specification, trade-offs can be made between performance and precision. However this requires the identification of library elements that implement a specification within some bound. For example, the specification for the antialiased line rasterizer operated on 8 bit pixels (outputs *I*, *Iinc*, and *Idec*, were 8 bits long). Library elements may exist that can perform this operation on 6 bit pixels with greater throughput than would be possible with 8 bit pixels. By identifying a match within the bounds $[0, 2^2-1]$ between the specification and the library element, the precision of the specification may be sacrificed for reduced power or area, or higher frequency of operation. Inexact matching can be performed in several ways, by matching a specification and library element that have: (1) equivalent polynomial representations, but different bit widths, (2) polynomial representations that are the same within some error bound, or (3) similar Taylor expansions.

As outlined above, an implementation and specification may perform the same functionality, as specified by their polynomials, but with different bit widths. For example, assume the 8 bit adder shown in Fig. 1 (Adder8) could be instantiated in place of Adder12 by shifting the outputs of the previous stages right by 2 bits and the output of Adder8 left by two stages. The resulting implementation would be smaller and could operate at a higher frequency, and would be accurate within $[0, 2^2-1]$.

In the second application of inexact matching, the polynomial representations of the specification and the library element may be different, but the maximum magnitude of their difference may be small. For example, in synthesizing *invDenom = 1/(2*((dx^2 + dy^2)^(1/2)))*, a library element that performed *x/2* could be used. However, *x/2* can not be performed exactly with binary values since the result is non integer for odd values of x. A library element that performs *x>>1*, with an 8 bit input, has the approximate polynomial representation *.498x* (see [SmDe99]). In instantiating *x>>1* to implement *x/2* for an 8 bit input, the maximum error between specification and implementation is guaranteed to be less than the maximum value of *.5x - .498x* over the interval $[0, 2^8-1]$, which is .51.

A third application of inexact matching is required for specifications that do not have exact polynomial representations. For example, in synthesizing *invDenom = 1/(2*((dx^^2 + dy^^2)^^(1/2)))*, a library element is required to perform the function *1/x*. This specification does not have an exact polynomial representation, but can be approximated by a polynomial using Taylor's expansion (within a neighborhood of *a*):

$$1/x = \sum_{i=0}^{\infty} (-1)^i \cdot (x-a)^i / a^i$$

A library element that performed *256/x* for an eight bit input would be split into eight subdomains, since Taylor's approximation converges to an accurate polynomial representation only when the input domain is restricted. As a result, the functionality characteristics of *256/x* are modeled by Divider8, as shown in Fig. 1. Using the error bound from Taylor's expansion, *1/x* can be inexactly matched to Divider8 within the bounds [0, 1].

7. Implementation and Example

The techniques presented above for allocating reusable designs have been implemented in the POLYSYS synthesis tool and are used here to the synthesize the antialiased line rasterizer. The library of reusable designs contains elements that perform multiplication, addition, inversion, and square root. These elements are characterized in the library, as shown in Figure 2, from their Verilog implementation. The Verilog module for each reused design, along with the Verilog code for logic that is not be mapped to a complex library element, is passed to the Synopsys Design Compiler for gate level synthesis.

The first partition for which a match to an existing component is sought is {*dx = x2 - x1*}. While there is no exact match for this specification, the polynomial representation for Adder8, $x + y$, matches the specification if $x = x2$ and $y = -x1$. In order to complete this match, a component is then sought with polynomial representation $-x$. Since no such component exists, synthesis using logic gates is performed to invert *x1*. A similar set of steps is required to synthesize the next partition {*dy = y2 - y1*}.

The partition {*incrE = 2*dy*} is matched to

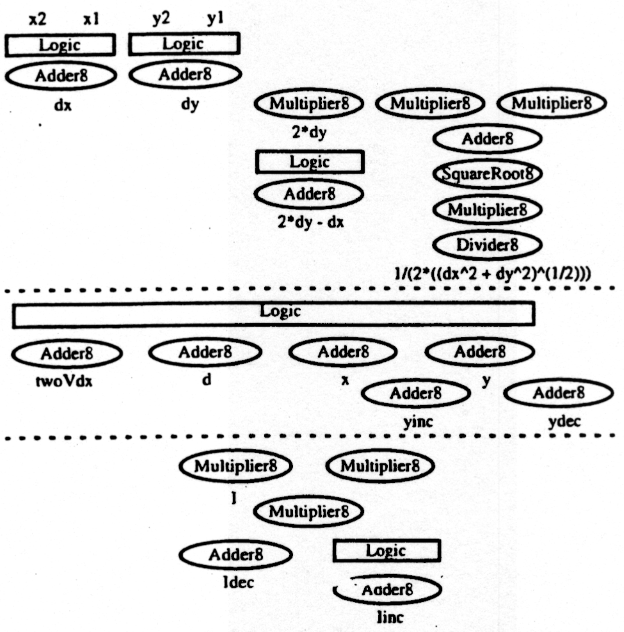

Fig. 2 Antialiased line rasterizer mapped to reusable designs and optimized through scheduling and resource sharing.

Multiplier8 under the condition $x = 2$. Furthermore, the partition {$incrNE = 2*dy - dx$} is matched to Adder8 under the conditions $x = 2*dy$ and $y = -dx$. Multiplier8 and basic gates are then allocated to implement $2*dy$ and $-dx$.

The partition {$invDenom = 1/(2*((dx^{\wedge\wedge}2 + dy^{\wedge\wedge}2)^{\wedge\wedge}(1/2)))$} is not in polynomial form, thus the first step is to create a polynomial representation of that specification. Computing the Taylor series expansion about $2*((dx^{\wedge\wedge}2 + dy^{\wedge\wedge}2)^{\wedge\wedge}(1/2)))$ reveals a match to Divider8 under the conditions $x = 2*((dx^{\wedge\wedge}2 + dy^{\wedge\wedge}2)^{\wedge\wedge}(1/2)))$. This statement is matched to Multiplier8 under the conditions $x = 2$ and $y = (dx^{\wedge\wedge}2 + dy^{\wedge\wedge}2)^{\wedge\wedge}(1/2))$. The latter condition is not in polynomial form, requiring Taylor expansion about $dx^{\wedge\wedge}2 + dy^{\wedge\wedge}2$. The resulting polynomial matches SquareRoot8 under the conditions $x = dx^{\wedge\wedge}2 + dy^{\wedge\wedge}2$. This condition is then satisfied by allocating Adder8 and two instances of Multiplier8.

The next partitions for which a complex element is sought are those that specify the computation of *twoVdx*, *d*, *x*, *y*, *yinc* and *ydec*. Adder8 is allocated to implement each summation and additional logic gates are required to implement subtraction, as performed earlier. The last three minimal partitions that are bound are those for computing *I*, *Iinc*, and *Idec*. Logic level synthesis is performed to implement the partitions with unbound logic, such as {*if (init) ... else ...*}.

Scheduling the operations to be performed and sharing resources among partitions results in the binding shown in Figure 2. In this case, since components do not employ complex communication protocols, interface synthesis is reduced to connecting the ports of the complex elements. The physical characteristics of the synthesized antialiased line rasterizer are shown in Figure 3.

Library	LSI LCB007
Gate Count	14094
Max. Operating Frequency	25MHz
Est. Area	8.85mm^2
Est. Power Consumption	3.35W

Fig. 3 Physical characteristics of antialiased line rasterizer synthesized from reusable blocks.

8. Conclusion

In performing high level synthesis and reusing existing designs, a mechanism for representing the functionality and physical characteristics of complex blocks efficiently and accurately is required. We have presented a methodology in which the functionality of complex blocks can be canonically represented at a high level of abstraction. This allows existing designs to be allocated efficiently in implementing a specification. This methodology can be followed in tandem with existing methodologies that operate at the bit level, as demonstrated in the synthesis of an antialiased line rasterizer.

Polynomial methods are ideally suited for automating the allocation of existing designs that perform arithmetic computations, such as those required for DSP and graphics applications. They provide a mechanism for performing exact and inexact matching of a specification, generated from a high level tool such as MATLAB, to an existing implementation, described in Verilog or with Boolean equations.

Future work will focus on expanding the application of the methodology presented here to high level synthesis of hardware/software systems and to synthesis with complex analog elements. Software modules that implement arithmetic functionality and complex analog blocks are prime candidates for polynomial representation.

References

[1] Dataquest report on the Electronic Design Automation industry, December, 1996.

[2] J. Smith and G. De Micheli, "Polynomial Methods for Component Matching and Verification", *Proceedings of the ACM/IEEE International Conference on Computer Aided Design*, 1998.

[3] R. Bryant "Graph Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, C-35(8), 1986.

[4] R. Bryant and Y.A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams", *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.

[5] Y.A. Chen and R. Bryant, "ACV: An Arithmetic Circuit Verifier", *Proceedings of the ACM/IEEE International Conference on Computer Aided Design*, 1996.

[6] E.M. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams", *Proceedings of the ACM/IEEE International Conference on Computer Aided Design*, 1995.

[7] E.M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transformations for Large Boolean Functions with Applications to Technology Mapping", *Proceedings of the 30th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, 1993.

[8] G. Martin, "Design Methodologies for System Level IP", *Proceedings of the Conference of Design Automation and Test in Europe*, 1998.

[9] J. Smith and G. De Micheli, "Polynomial Methods for Allocating Complex Components", *Proceedings of the Conference of Design Automation and Test in Europe*, 1999.

[10] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.