ALLOCATION AND INTERFACE SYNTHESIS ALGORITHMS FOR COMPONENT-BASED DESIGN

James Smith

Technical Report No.: CSL-TR-00-793

February 2000

This project is in part supported by ARPA under contract DABT63-95-C0049 and the support of the MARCO/DARPA Gigascale Silicon Research Center under contract SA2206-23106PG-1.

Allocation and Interface Synthesis Algorithms for Component-Based Design

James D. Smith Jr.

Technical Report No.: CSL-TR-00-793

February 2000

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
William Gates Computer Science Building, A-408
Stanford, CA 94305-9040
<pubs@shasta.stanford.edu

Abstract

Since 1965, the size of transistors has been halved and their speed of operation has been doubled, every 18 to 24 months, a phenomenon known as Moore's Law. This has allowed rapid increases in the amount of circuitry that can be included on a single die. However, as the availability of hardware real estate escalates at an exponential rate, the complexity involved in creating circuitry that utilizes that real estate grows at an exponential, or higher, rate. Component-based design methodologies promise to reduce the complexity of this task and the time required to design integrated circuits by raising the level of abstraction at which circuitry is specified, synthesized, verified, or physically implemented. This thesis develops algorithms for synthesizing integrated circuits by mapping high-level specifications onto existing components. To perform this task, word-level polynomial representations are introduced as a mechanism for canonically and compactly representing the functionality of complex components. Polynomial representations can be applied to a broad range of circuits, including combinational, sequential, and datapath dominated circuits. They provide the basis for efficiently comparing the functionality of a circuit specification and a complex component. Once a set of existing components is determined to be an appropriate implementation of a specification, interfaces between these components must be designed. This thesis also presents an algorithm for automatically deriving an HDL model of an interface between two or more components given an HDL model of those components. The combination of polynomial representations and interface synthesis algorithms provides the basis for a component-based design methodology.

Acknowledgment

I would like to take this opportunity to thank my advisor, Giovanni De Micheli, for his guidance and support throughout my undergraduate and graduate studies. He has provided invaluable guidance not only relating to the technical contributions of this research, but also to the clarity and precision of this paper and many others. My greatest strides in understanding have come as a result of our discussions and his encouragement to be clear and precise in communication.

I would also like to thank my parents, James Smith Sr. and Frances Summe-Smith for their 28 years of tutelage, love, and support. No son has ever been luckier to have two parents dedicated to his academic and personal success.

I would also like to thank my wife, Patricia Uyehara Smith, for her love and support for the last four years. The strength that she provides allows me to attack the challenges of academic and professional life.

I would also like to acknowledge the support of ARPA under contract DABT63-95-C0049 and the support of the MARCO/DARPA Gigascale Silicon Research Center under contract SA2206-23106PG-1.

Table of Contents

Table of Figures

Chapter 1
Introduction

1.1 Design of Hardware Systems

Modern general-purpose and application-specific hardware is restricted practically by two factors: (1) the amount of circuitry that can physically be placed on a printed circuit board or silicon die and (2) the amount of time required to determine and implement these structures. Since 1965, the semiconductor industry has, following Moore's Law, reduced the size of transistors by a factor of two, and doubled their speed of operation, every 18 to 24 months. This has allowed rapid increases in the amount of circuitry that can be included on a single die. Furthermore, this trend is expected to continue through the year 2012 [SIA97]. However, as the availability of hardware real estate escalates at an exponential rate, the complexity involved in creating circuitry that utilizes that real estate also grows at an exponential, or higher, rate. As a result, during periods in which design methodologies remain constant, the time required to determine and implement integrated circuits has grown quickly. The ratio between the availability of silicon real estate and the amount of circuitry that can be developed for that real estate in a single year is termed the *design gap*.

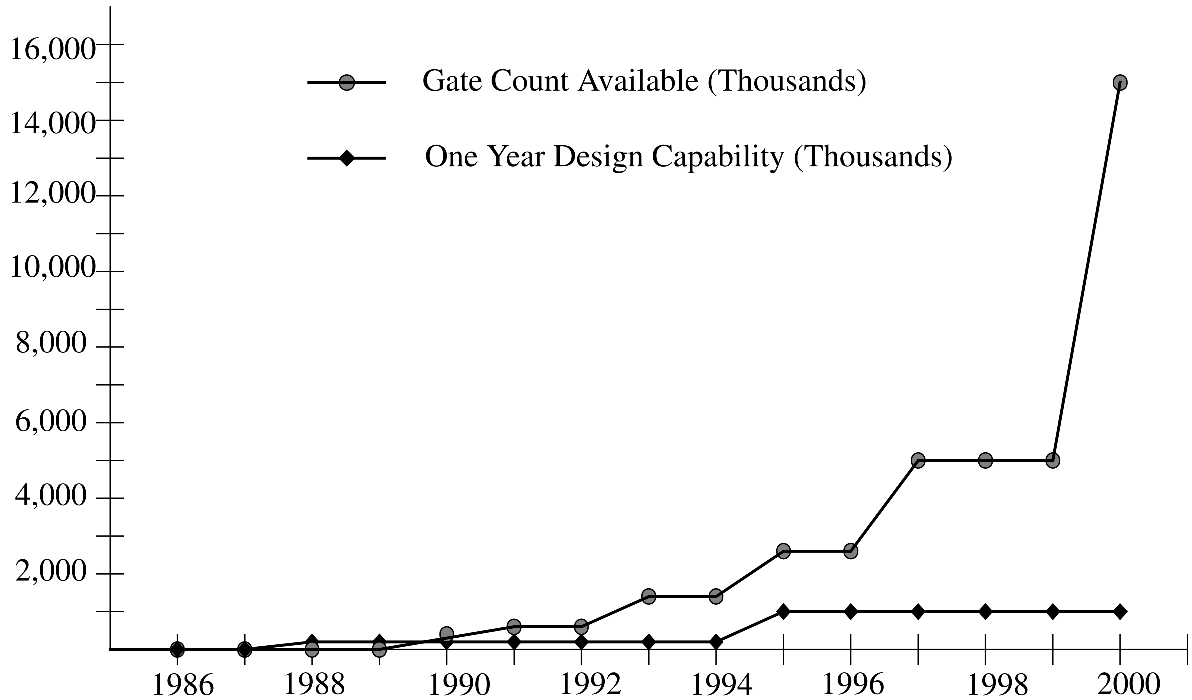


Fig. 1 The *design gap* [Da98]

The current dominant design methodology for general-purpose and application-specific systems has existed for nearly ten years, and a corresponding design gap has developed (Figure 1). As a result, systems that were designed in a matter of months five years ago now take several years. Following this trend, by the year 2001, a completely new integrated circuit that utilizes all available silicon die space will take ten years to get to market.

Hardware design methodologies are characterized by the language used to specify the design, the techniques used to transform that specification into a format from which an integrated circuit can be fabricated, and the techniques used to verify that design has been specified and transformed correctly. The current dominant methodology, in a simplified view, is one in which a design is specified at the Register Transfer Level (RTL) with a Hardware Design Language (HDL), such as Verilog HDL or VHDL. The design is then transformed into logic gates, such as NAND or NOR gates, a task termed *logic synthesis*. A

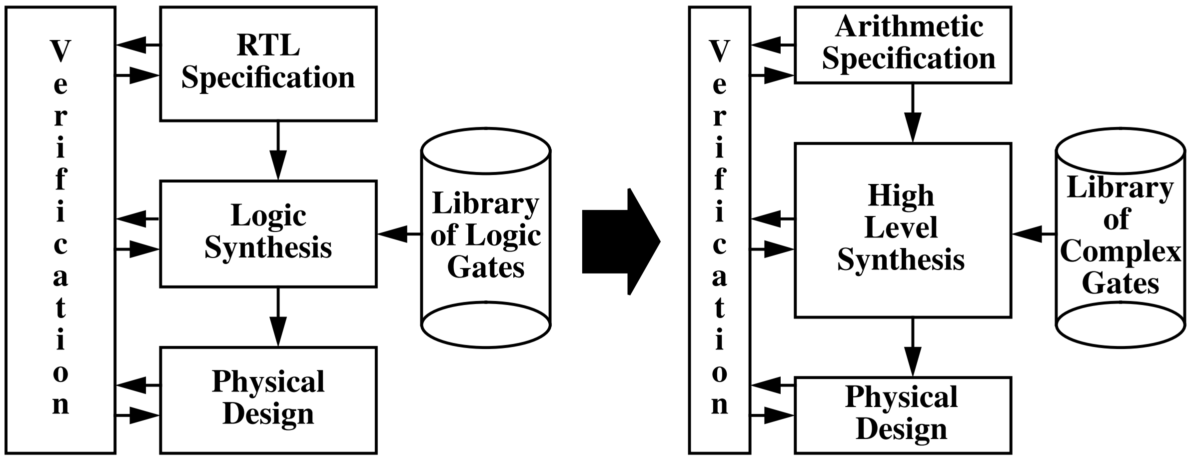


Fig. 2 Transformed design methodology

plan for placing and connecting the resulting logic gates on the silicon die is then developed, a task termed *layout*. During each one of these steps, the design, in its corresponding state, is verified. In the specification stage, this generally requires simulation of the design. During logic synthesis and layout, formal techniques can be used, in addition to simulation, to prove that the design implements the same functionality as it did prior to the transformation performed in that stage.

As transistor sizes shrink, the complexity of specification and synthesis, verification, and physical design increases exponentially. When a transistor is reduced in size by half, four times as many transistors can theoretically be placed on an integrated circuit. This means that the amount of circuitry that must be specified goes up by a factor of four. Similarly, the number of logic gates, or other implementation constructs, that must be simulated or proven to be correct increases by a factor of four. For physical design, not only does the number of gates increase by a factor of four, but the number of interconnects increases by an even greater factor, and the physical characteristics of silicon and metal are altered. In each of these three areas (specification/synthesis, verification, and layout) a body of research has been undertaken to address the exponential increase in design complexity. High level synthesis and design reuse aim to alleviate the specification and synthesis problem. Formal verification and symbolic simulation attempt to simplify the complexity of verification. Research in deep submicron layout, floorplanning, and device characterization seeks to address the physical design issues and couple them tightly to logic synthesis. This work targets the first of these research topics, high level synthesis and design reuse.

1.1.1 High Level Synthesis and Design Reuse

High level synthesis and design reuse promise to shorten the time required to specify a design, perform synthesis and layout, and verify that the design is correct. Specification can be performed at a level of abstraction greater than RTL, allowing more functionality to be specified more quickly. Furthermore, by mapping a specification onto blocks that are more complex than logic gates, synthesis can be performed more quickly, allowing for wider design space exploration. In addition, optimizations can be performed at the architectural or algorithmic level, rather than the logic level, allowing for improvements that have a much greater impact on system performance. In using existing designs, an engineer can reuse layouts, floorplans, logic synthesis scripts, or initial cell placements, allowing highly optimized layouts to be created quickly. Finally, by specifying a design with higher level constructs and mapping that specification onto existing designs, simulation and formal verification can be performed at a higher level, reducing the time required to verify that the design is correct.

Hardware systems are frequently composed of datapath blocks that perform mathematical operations, such as multiplication, Fast Fourier Transform (FFT), or Discrete Cosine Transform (DCT), and control blocks that determine when these operations are performed. In order to take advantage of the benefits conferred by higher level specifications, we propose altering the current design methodology such that the datapath blocks are specified by mathematical operations, while preserving the ability to specify control at the register-transfer or behavioral level. In addition, to take advantage of the benefits conferred by design reuse, this methodology performs synthesis with complex blocks that are at the level of multiplication, FFT, or DCT, rather than basic logic gates (Figure 2).

While synthesis with complex blocks promises to reduce the complexity of design specification, verification, and layout, the synthesis task itself becomes much more difficult. The problem of mapping a design specification to library elements, termed *allocation*, or *binding*, requires comparisons of functionality between a high level specification and library elements that may be described at a low level. The problem of choosing the best mapping, termed *optimization*, requires evaluation of the impact that a mapping has on the rest of the system. Furthermore, the problem of connecting two complex blocks, termed *interface synthesis*, requires computation of logic that allows blocks to communicate despite having different communication paradigms. The body of this work focuses on the first and third of these problems, allocation and interface synthesis.

1.1.2 Levels of Abstraction

RTL design methodologies today operate at the logic, or gate, level of abstraction. At the logic level, systems are specified by signal bits, logic operations, and a subset of mathematical expressions (generally restricted to addition, subtraction and multiplication). This specification is mapped to logic gates. If a system, or subsystem, is combinational, it can be seen as a partial order of logic gates and modeled by a *logic network*. A common abstraction for combinational circuits is a *logic function*. If a system, or subsystem, is sequential, it can be seen as a partial order of logic gates and synchronous elements, such as registers. Sequential circuits can be modeled by a *synchronous logic network*. A common abstraction for sequential circuits are Finite State Machines (FSMs).

At higher levels of abstraction, systems can be specified, created, and verified more quickly. At the arithmetic level of abstraction, a system is specified by collections of associated signal bits, also known as words, control operations, and the complete set of mathematical expressions including division, exponentiation, transcendental functions, and combinations of those operations. This specification is mapped to blocks that perform arithmetic operations. At this level, a system can be seen as a partial order of arithmetic operations and can be modeled by a *dataflow graph*. Sequences of arithmetic operations may be executed conditionally, a situation that can be modeled as a *guarded dataflow graph*. Sequences of arithmetic operations may also be executed many times, a situation that can be modeled as a loop with embedded guarded dataflow graphs.

1.2 Component Matching

Reusing existing circuitry can significantly reduce the time required to construct new systems. The proliferation of reusable blocks has promised opportunities to complete new designs more quickly and with fewer errors. However, searching the space of existing implementations is time consuming and fraught with pitfalls, as the suitability of existing blocks is determined by manual methods or verbal descriptions. This search promises to become more complex as the number and need for reusable designs increases ([Ma98], [DaBoBe99]). The models and methods presented in this research enable automation of this search by generating circuit representations that are at the arithmetic level of abstraction.

This enables efficient and accurate functionality comparisons of high-level specifications and complex components.

Traditional matching applications operate at the logic level, allocating logic gates given an HDL specification. Current extensions to logic level matching include arithmetic macrocells to handle simple, commonly used arithmetic operations such as addition and multiplication. Macrocell mapping is performed manually or by comparing symbolic descriptions. Macrocells are considered to be *hard* if they are physically a piece of hardware, are considered to be *soft* if they are an HDL specification of functionality, and are considered to be *firm* if they are described by logic gates that have been placed and routed.

Component matching is the problem of allocating complex blocks given a system specification. This problem reduces to determining whether or not the functionality of a library element is the same as the functionality of part of a specification. For example, in designing the baseline JPEG encode block of Figure 3, subblocks are required to perform a Discrete Cosine Transform, quantization, DC (zero frequency) encoding, and AC (non-zero

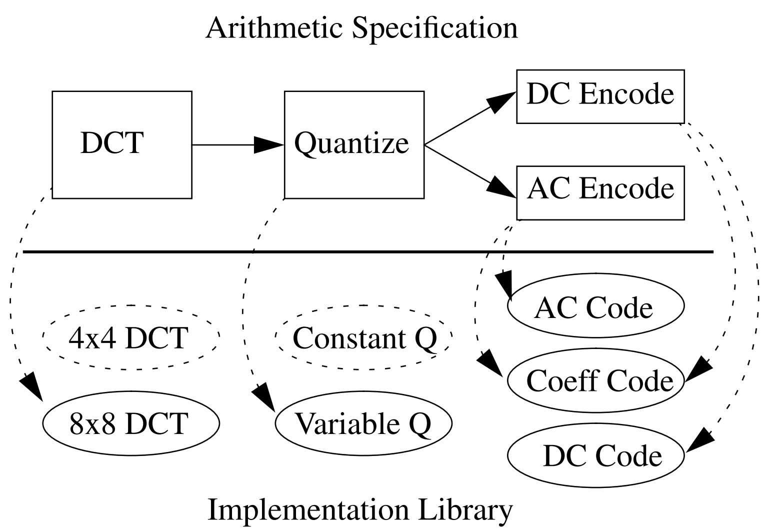


Fig. 3 Mapping JPEG encode onto existing designs

frequency) encoding. A library of existing complex blocks is provided by Intellectual Property (IP) vendors. The elements of the library range in level of complexity from adder or multiplier to DCT, quantizer, or encoder. This is in contrast to traditional libraries that contain simple elements that range in complexity from NAND or NOR gates to adders or multipliers. In performing component matching, complex library elements are searched for to implement the functionality described by each sub-block of this specification. This match can be determined by deriving a single word-level representation for both the system specification and the Boolean equations that describe the functionality of library elements. The JPEG system can then be synthesized by matching the arithmetic specification of each of these functions to the word-level representation of each library element.

Component matching is closely related to verifying that a specification and an implementation match exactly, but presents important differences. In matching a component to a specification, it is valuable to detect components that implement functionality that is similar to, but not necessarily the same as, that of the specification. For example, in performing DCT operations, a specification may require computation of the cos(x). One possible implementation may be a function that does not implement cos(x) exactly, but instead implements an approximation of the function to preserve area and increase performance. Furthermore, a specification may indicate that computation of cos(x) can take up to three cycles; however, existing implementations may exist that require only two cycles. Thus, the specification and implementation are similar, but do not match exactly, allowing for tradeoffs in execution time, silicon area, power consumption, precision and other qualities.

The complex components discussed to this point can be specified very efficiently with polynomial models. For example, cos(x) can be approximated by:

$$1 - x^2/2! + x^4/4! - x^6/6! + ... + x^n/n!$$

This research derives methods for computing analogous word-level polynomial models for existing components given a bit-level description of the component. These methods are ideally suited for circuits that implement arithmetic functions and can be applied to combinational and sequential circuits.

This comparison often must be performed between arithmetic and bit-level abstractions of the functionality. Polynomial methods provide a means for generating word-level polynomial representations, given bit-level descriptions of an implementation. In generating a mathematical structure common to both levels of abstraction, allocation of complex components can be performed, closing the semantic gap between specifications and implementations. A common example of such a process is comparing a specification generated in MATLAB against implementations modeled with Boolean logic or HDLs. This technique is used by the POLYSYS synthesis tool to map arithmetic specifications onto existing designs that are described by Boolean equations.

The techniques presented here are most effective for allocating blocks that are arithmetic intensive, but may contain significant control logic. Common application domains that fit this description include computer graphics and digital signal processing. To illustrate the application of the polynomial methods developed in this research, we map a JPEG encode specification to complex elements and compare the specification of a filter suitable for controlling the velocity of a tape through a tape drive to an existing filter. The arithmetic specification for the JPEG encode block and the Infinite Impulse Response (IIR) filter are derived from MATLAB, while the existing components are described by Boolean equations.

1.3 Interface Synthesis

After determining the space of valid implementations of a specification, the components that make up these implementations must be tied together. In traditional logic level synthesis, this is simply a matter of instantiating a wire between two logic elements. However, when synthesizing with libraries of complex components or manually reusing existing designs, communicating blocks must transmit data using compatible protocols. For example, a CPU frequently implements a bus protocol that requires a vastly different set of port operations to send and receive data than a synchronous DRAM. The problem of interface synthesis is generating logic that allows two such components to communicate with one another.

An industrial consortium, the Virtual Socket Interface Alliance (VSIA), has recognized the importance of automating the connection of system components. To this end, VSIA and has proposed standardization of data formats, test methodologies, and interfaces ([VSIA99]). However, fragmentation and competition within the IP design industry is likely to preserve the need for synthesizing interfaces between complex components.

Components may communicate asynchronously or synchronously and each component may operate at the same or different frequencies. Furthermore, multiple components may communicate with a single component through a single port or through many ports. Each of these options presents a unique challenge in automating interface generation. The techniques presented here target interface generation for synchronous components that may or may not operate at different frequencies and that connect two or more components.

1.4 Thesis Contributions

This research develops algorithms that enable synthesis of integrated circuits using libraries of complex components. To achieve this goal, this work develops algorithms for matching high level specifications to complex components and for generating interfaces between the chosen components. Within the topic of component matching, we develop algorithms for matching circuits that implement predominantly arithmetic functionality, but may contain control operations. Within the topic of interface synthesis, we automatically generate logic that allows communication between two or more synchronous components that have different, built-in communication protocols.

The problem of matching high-level specifications to complex components is attacked by comparing the functionality of the specification, which is described with both arithmetic and control operations, against that of the component, which is described by logic operations and synchronous elements. Matching algorithms can be classified according to several factors: (1) the level of abstraction at which matching is performed, (2) the size of the representation, (3) the time complexity of computing the representation, (4) the canonicity of the representation used for matching, and (5) the quantification of differences between specification and implementation.

We develop polynomial methods for performing component matching. These methods rely on representing component functionality with polynomial expressions. These expressions are at the word level, requiring transformation of bit level circuit descriptions into word-level polynomials. This transformation allows for comparisons of specifications which are described at the arithmetic level and implementations which are described at the bit level. Furthermore, by using a polynomial representation to perform matching, the complexity of the representation is independent of the number of circuit input bits. As demonstrated in Chapter 3, polynomial representations can be computed in polynomial time with respect to the number of input bits. This representation is also canonical, guaranteeing that valid matches will not be interpreted as invalid matches. Using polynomial representations, differences between a specification and an implementation can be computed according to two metrics: (1) the maximum numerical magnitude of the difference between the specification and implementation polynomials, or (2) the polynomial representation of a component that could compensate for their differences. This allows detection of components that approximately implement a specification and of components that partially implement a specification.

The problem of generating a synchronous interface between complex components is attacked by generating a finite state machine that converts each component protocol into a standard protocol. This allows the two components to communicate by simply connecting the appropriate ports. The techniques presented in this work can be implemented to synthesize interfaces between synchronous components that may or may not operate at the same frequency. In addition, these techniques allow the generation of multi-way interfaces in which multiple components communicate over the same bus. The input to the interface generator is the HDL model or the logic equations that describe the component. The output is an HDL model of the state machine that enables conversion to the standard protocol.

The component matching algorithms that are described in this work are best suited to datapath dominated circuits. Though these techniques can detect those circuits that are control dominated, they will be come exponentially complex if used to represent this class of circuits. The interface synthesis algorithms are best suited to combining components that have complex built-in communication protocols. These techniques focus on generating an interface that enables correct communication, rather than optimizing the frequency of operation, area, or power consumption of the interface. Thus, for those circuits with simple built-in protocols, for which interface generation is not a complex task, the related algorithms described in Section 2.3 are likely to be superior.

1.5 Thesis Outline

Chapter 2 provides a background on the techniques that others have developed to represent and verify circuits and to generate interfaces between circuits. Several of these techniques are used in this research. Representation mechanisms include Binary Decision Diagrams, Directed Acyclic Graphs, Binary Moment Diagrams, Multi-Terminal Binary Decision Diagrams, and Object Oriented Methods. The interface synthesis techniques reviewed in this chapter span specification languages, computation of glue logic for similar interfaces, and high-level optimization of communication.

Chapter 3 derives polynomial methods for combinational circuits. Polynomial methods provide a mechanism for deriving the arithmetic functionality of a circuit that is described by logic equations. We prove that polynomial representations can be derived for any circuit and are guaranteed to be canonical. We also derive a mechanism for detecting control operations within a component. In many cases, this allows control functionality to be represented and matched by the techniques of Chapter 2 and datapath functionality to be represented and matched using polynomial methods. This results in a compact representation for circuits with both arithmetic and control. Finally, we develop a technique for efficiently implementing polynomial methods.

Chapter 4 develops extensions to polynomial methods for representing and comparing circuits that contain synchronous elements and feedback paths. In addition, a mechanism for generating approximate polynomial representations, which are more compact than the exact representation, and the error associated with that representation is computed. Two examples are then completed, one in which the DC path of a JPEG Encode block is mapped to complex elements, and one in which a filter which controls a tape drive compensator is mapped to an existing component. Finally, experiments that illustrate the performance of polynomial methods are described.

Chapter 5 derives the algorithm for computing interfaces between synchronous components. We specify a standard communication protocol to which the built-in protocols of existing components are mapped. We also describe the interface architecture in which this communication is implemented. The algorithm for generating the state machine that converts a component's protocol into the standard protocol is then presented. An example illustrating the synthesis of an interface to a MIPS CPU is then completed.

Chapter 6 describes the POLYSYS synthesis suite. POLYSYS uses the techniques described in Chapters 3, 4, and 5 to synthesize systems given libraries of complex blocks. This software implements Internet protocols to create of libraries of complex blocks, match a design specification to vendor components which are remotely distributed, and synthesize interfaces to those same components. The distributed nature of POLYSYS, allows publication of vendor libraries and evaluation of implementation choices without disclosure of intellectual property.

Chapter 7 describes the synthesis of an Antialiasing Line Rasterizer using the algorithms presented above. The rasterizer specification includes control operations and complex mathematical operations. The library to which this specification is mapped is composed of complex elements that perform mathematical operations, such as multiplication, square root, and combinations of multiplication and addition, as well as branching operations. Interfaces are then generated between the chosen components.

Chapter 8 summarizes the contributions of this research and proposes future research directions. These directions include potential improvements of the techniques presented in this research and extensions to enable high-level synthesis of hardware/software systems.

Chapter 2
Background

2.1 Introduction

In this chapter, we will provide a background on the structures that this research employs. This includes Binary Decision Diagrams and Binary Moment Diagrams. We will also describe why these structures alone are not suitable for comparing the functionality of a specification to complex components. Furthermore, we will review the existing techniques for representing the functionality of complex blocks, including additional decision diagram structures, state based representations, and less precise representations of functionality. We will describe the domains of research within interface synthesis, including modeling languages, synthesis of glue logic, and high-level optimization of communication. Finally we will compare and contrast existing work to the algorithms presented in this work.

2.2 Functional Representations

Reusable blocks have traditionally been characterized by verbal descriptions ([BaRo99], [Se99]), such as “ethernet core” or “rasterizer”, combined with component-specific attributes, such as “floating point” or “integer”, and waveforms. Precise descriptions of functionality are usually restricted to smaller blocks such as combinational logic gates or simple arithmetic operations (e.g. addition or multiplication). For example, in allocating a JPEG block, current techniques may require that the specification and implementation both be described by the keyword “JPEG”. This description is imprecise, however, as potential JPEG implementations may implement different compression schemes, different levels of accuracy, or operations on data sets of different sizes.

Component matching has historically been restricted to matching bit-level circuit specifications to logic gates. Many structures have been used to represent and manipulate Boolean functions. Early techniques used structures such as truth tables, Karnaugh maps, and canonical sum of products forms [HiPe74]. These forms of representation are always of exponential size with respect to the number of input bits. For some functions, reduced and factored transformations of these structures may alleviate the exponential nature of the representation, but result in a non-canonical representation. Lack of canonicity makes component matching very difficult, as a failed comparison of specification and implementation is no guarantee that the two are incompatible.

2.2.1 Directed Acyclic Graph Representations

Binary Decision Diagrams ([Br86]) are ideal for mapping classes of combinational logic onto a library of gates. A BDD is a directed acyclic graph with two sink nodes, representing logic values 0 and 1. Non-sink nodes are labeled with a Boolean input variable and are the roots of two edges, representing the assignments of 0 and 1 to that variable ([BrRuBr90]). This is the equivalent of recursively expanding a function F(**x**): $B^{m} \rightarrow B$, B = {0, 1} about each of its cofactors (Boole-Shannon Expansion):

$$F(\mathbf{x}) = x'_{m-1} \cdot F_{x', m-1}(x) + x_{m-1} \cdot F_{x, m-1}(x)$$

A reduced BDD with fixed variable ordering is a canonical form of Boolean logic and, for many functions, is not of exponential complexity with respect to input size [Br92]. Furthermore, the ease of composition of BDDs enables efficient computation of combinations and complements of Boolean equations. However, for other functions, such as integer addition and multiplication, the potentially exponential size of BDD structures makes comparison of BDDs time consuming and memory intensive.

BDDs can yield information on whether or not a Boolean specification and implementation match at the bit level, but offer no path for quantifying the degree to which the two differ at the arithmetic level. That is, two functions that have similar, but not equal, BDD structures may implement drastically different arithmetic functions, while two very different BDDs may implement approximately the same mathematical operation. For example, the Verilog HDL assignments:

y[15:0] = {0, x[15:1]};

y[15:0] = {0, x[15:1]+x[0]'};

have very different BDD representations. The first assignment requires only one BDD node per output bit while the second representation requires at least 136 total BDD nodes to represent the operation. However, in the integer domain, these two assignments specify very similar functionality ($\lfloor x/2 \rfloor$ and $\lceil x/2 \rceil$ respectively). BDDs are an efficient mechanism for performing simple component matching when both specification and implementation are described by Boolean equations, but become prohibitively complex for many arithmetic operations.

Multi-Terminal BDDs ([ClFu93]), also known as Algebraic Decision Diagrams

([BaFrGa93]), provide a mechanism for expressing a mapping from the binary domain to the integer domain. This is similar to a binary decision tree in which the leaves are integer values. Such a decomposition is the equivalent of an implicit enumeration of input/output pairs. By placing a strict ordering on vertices during tree traversal, MTBDDs do guarantee the canonicity of the representation. However, for even simple circuits, MTBDDs do require an exponential number of vertices relative to the number of input bits.

Binary Moment Diagrams (BMDs) ([BrCh95]) have been developed to ease the memory and time required to manipulate mathematical structures. BMDs are word-level representations of Boolean functionality. If the Boole-Shannon Expansion used to generate BDDs is altered by replacing x'_{m-1} with $(1-x_{m-1})$, the following expansion results for $\mathbf{F(x)}$: $B^m \to B^k$, $B = \{0, 1\}$:

$$\mathbf{F(x)} = \mathbf{F}_{x'_{m-1}}(\mathbf{x}) + x_{m-1} \cdot \left(\mathbf{F}_{x_{m-1}}(\mathbf{x}) - \mathbf{F}_{x'_{m-1}}(\mathbf{x}) \right)$$

For functions $\mathbf{F(x)}$ that implement linear mathematical functionality, the difference $\mathbf{F}_{x_{m-1}}(\mathbf{x}) - \mathbf{F}_{x'_{m-1}}(\mathbf{x})$ is always constant. Thus, the decomposition is guaranteed to be of linear complexity for functions such as addition and multiplication. Furthermore, by factoring constants out of the expressions for $\mathbf{F}_{x'_{m-1}}(\mathbf{x})$ and $\mathbf{F}_{x_{m-1}}(\mathbf{x}) - \mathbf{F}_{x'_{m-1}}(\mathbf{x})$, common sub-expressions frequently arise with the BMD structure. BMDs with edges weighted according to the constant that has been factored, is termed a *BMD. By sharing common sub-expressions, *BMDs are of linear complexity for many mathematical operations, including $F(x) = 2^x$. However, for most functions that implement non-linear operations, the complexity of BMD structures scale with the degree of non-linearity. For functions that do not implement mathematical operations, BMD structures are potentially of exponential complexity. *BMDs have been used to verify the functionality of circuits that implement linear functionality [ChBr96] and could be adapted to perform component matching for those circuits. However, *BMDs are unsuitable for use in many non-linear functions because of the resulting complexity. Furthermore, composition of BMD structures is an exponentially complex operation. This complicates quantifying the difference between a specification and implementation and matching a specification to multiple components.

Hybrid Decision Diagrams ([ClFu95]) attempt to overcome some of the limitations of BMDs and BDDs. HDDs are similar to BMDs and BDDs, however, the expansion for a function **F(x)** can be different for each input bit x_i. The expansion for each input bit is chosen from one of the following:

$$\mathbf{F}(\mathbf{x}) = (1 - x_{m-1}) \cdot \left(\mathbf{F}_{x'_{m-1}}(\mathbf{x})\right) + x_{m-1} \cdot \left(\mathbf{F}_{x_{m-1}}(\mathbf{x})\right)$$

$$\mathbf{F}(\mathbf{x}) = \mathbf{F}_{x'_{m-1}}(\mathbf{x}) + x_{m-1} \cdot \left(\mathbf{F}_{x_{m-1}}(\mathbf{x}) - \mathbf{F}_{x'_{m-1}}(\mathbf{x})\right)$$

$$\mathbf{F}(\mathbf{x}) = \mathbf{F}_{x_{m-1}}(\mathbf{x}) + (1 - x_{m-1}) \cdot \left(\mathbf{F}_{x'_{m-1}}(\mathbf{x}) - \mathbf{F}_{x_{m-1}}(\mathbf{x})\right)$$

$$\mathbf{F}(\mathbf{x}) = 1 - x_{m-1} \cdot \left(1 - \mathbf{F}_{x_{m-1}}(\mathbf{x})\right) + (1 - x_{m-1}) \cdot \left(1 - \mathbf{F}_{x'_{m-1}}(\mathbf{x})\right)$$

$$\mathbf{F}(\mathbf{x}) = 1 - \mathbf{F}_{x'_{m-1}}(\mathbf{x}) + x_{m-1} \cdot \left(\mathbf{F}_{x'_{m-1}}(\mathbf{x}) - \mathbf{F}_{x_{m-1}}(\mathbf{x})\right)$$

$$\mathbf{F}(\mathbf{x}) = 1 - \mathbf{F}_{x_{m-1}}(\mathbf{x}) + (1 - x_{m-1}) \cdot \left(\mathbf{F}_{x_{m-1}}(\mathbf{x}) - \mathbf{F}_{x'_{m-1}}(\mathbf{x})\right)$$

Note that the first expansion is the equivalent to that of a traditional BDD. The second expansion is equivalent to that of BMD structures.

A greedy algorithm is used to choose the appropriate set of decompositions that reduce the size of the resulting HDD. That is for each variable, the expansion that gives the smallest HDD size is selected. Thus, a function is canonically represented by its resulting

HDD and the expansions used to construct the HDD. Composition operations are less complex than those performed with BMDs and *BMDs. However, it requires not only the composition of the HDDs of the functions being composed, but also of the expansions used to construct each HDD. HDDs are effective for some linear equations, such as addition, but is exponentially large for operations such as multiplication and non-linear operations. Furthermore, HDDs are canonical with a fixed expansion, but are not when expansions are not fixed. Two blocks that specify or implement the same functionality with different logic operations may be expanded differently using a greedy algorithm, resulting in incorrectly failed matches.

Power Hybrid Decision Diagrams, developed in [ChBr97], are similar to HDDs, in that the transformations used to expand a function can vary between input bits. However, PHDDs reduce the number of transformations to three (simply because the last three transformations rarely provide any benefits) and re-introduce the concepts of factoring and edge weights used by *BMDs. The edge weights represent factors that are a power of 2 and negation edges are allowed. This makes PHDDs well suited to handling the non-linearities associated with floating point multiplication, but causes exponentially large data structures for floating point addition. Furthermore, PHDDs present similar canonicity problems to those inherent to HDDs and are extremely complex to compose.

Methods have been introduced for modeling and manipulating circuits that implement polynomial functions using Zero-suppressed BDDs ([Mi93]). Zero-suppressed BDDs remove BDD branches which lead to the zero leaf, reducing the number of nodes in the representation. In modeling polynomial functions, ZBDD graph edges are associated with a weight ([Mi96]). In traversing the directed acyclic graph, the exponent of a particular input is computed by multiplying the weights encountered in going from the root node to a leaf. Leaves are integer values that indicate the coefficient of a term. This structure provides an efficient representation for those circuits for which a polynomial description is specified, but becomes exponentially large if discontinuities exist in the function. Furthermore, no construction method exists for this structure, preventing computation of polynomial ZBDDs for components described only with Boolean equations.

2.2.2 Higher-Level Representations

The techniques discussed thus far provide detailed information not only about circuit functionality, but also implementation details. Higher-level representations describe functionality, but hide details of the actual implementation. A formalism for representing the functionality of sequential circuits is a finite state machine. An FSM is a high-level representation in the sense that it describes a circuit based on the states of internal and external signals. More formally, it is described by a 6-tuple $\{I, O, S, \lambda, \gamma, \rho\}$ where:

I is the set of inputs

O is the set of outputs

S is the set of states

λ is the next state relation, $\lambda: I \times S \rightarrow S$

γ is the output relation: $\gamma: I \times S \rightarrow O$

ρ is the initial state.

Such a description may hide details such as state encodings, providing a more intuitive structure for representing system functionality. However, state machine representations are not unique, as many sets of states can implement equivalent functionality. Furthermore, FSM complexity is proportional to the number of states. Thus, a state machine may be extremely complex for simple arithmetic operations. For example, if a counter is represented by an FSM, and the state space corresponds to the current value of the counter, an exponential number of states are required. To combat state explosion, states can be represented implicitly. For example, in the case of the counter, state enumeration may be replaced by the next state expression $\lambda = \lambda + 1$. The techniques described in Section 4.3 can be used to compute implicit representations of the next state relation.

Similar to FSM representations, instruction set representations hide implementation details while describing component functionality. An instruction set is classified as an *architectural-level* representation because it is a symbolic description of component resources. An instruction set hides implementation details by labeling execution options. For example, the label ADD may represent an option in which addition is performed. More formally, an instruction set is specified by the 2-tuple {I, O}, where I is the set of instructions and O is the operation performed by each instruction. Instruction sets are a compact form of representation, however they may not uniquely represent component functionality. For example, the label given to an instruction may not match the label used in specification, even though functionality is equivalent. In this work, we will use polynomials as a common description of component operations. An operation can be included in an instruction set by labeling the polynomial representation. For example, the cosine polynomial given in Chapter 1 may be given the label COS in an instruction set, allowing a high-level specification to be created by specifically calling the COS instruction.

2.2.3 Inexact Representations

In implementing design reuse, many industrial and academic synthesis tools have focused on object-oriented techniques ([KuAyJo94], [VaGi98]). These techniques are a mechanism for classifying component functionality according to keywords or concepts.

Such a classification scheme enables fast comparison of a specification and its potential implementations. However, this technique is not canonical, as many keywords or classes could be used describe functionality. It requires that the same taxonomy be used to specify the design that is used to classify components in the library of implementations. Furthermore, comparison requires that the properties of a specification and implementation be the same. For example, a specification for a DCT may include a characteristic indicating the function is performed on a 4x4 block. A library element, however, may assume that block size is the traditional 8x8 pixels and include characteristics such as latency and frequency of operation. Furthermore, unlike the techniques presented above, object oriented matching provides no rules for composing library elements, preventing the derivation of larger blocks from smaller components.

In order to raise the complexity of blocks for which a functional characterization can be generated, algorithms have been developed to reduce the size of circuit representations. This can be achieved by generating data structures that represent an approximation of circuit functionality. For example, in [RaMc98], a compact circuit approximation is derived that minimizes the number of input assignments for which the approximation and the actual circuit differ. In contrast, our work generates a compact circuit approximation that minimizes the numerical distance between the functionality of the representation and the actual block. Similarly, the allocation mechanism presented in this research determines the accuracy of a match by the numerical distance between a specification and a possible implementation.

Efficient component matching requires data structures that are canonical, constructible in polynomial time, and allow for simple composition. This dissertation will demonstrate methods for determining polynomial representations for circuits that are described at the bit level. Furthermore, we will prove that a unique minimum order polynomial representation exists for all circuitry without feedback. In representing hardware as polynomials, blocks can be efficiently compared with one another to determine if they implement the same functionality. In addition, polynomials are easily composable, allowing efficient determination of the functionality of hierarchical or partitioned blocks.

2.3 Interface Synthesis

Interface synthesis research can be broadly categorized in three areas. One body of research has focused on creating interface specification languages. These languages define constructs that allow swift creation of synthesizable interface specifications. A second research thrust has targeted synthesis of “glue logic” that allows two otherwise incompatible components to communicate. This logic is generated not from a specification of the interface, but from the implementation details of the communicating components. A third body of work has focused on improving the performance of systems of components by optimizing component communication at a high level. Work in each of these areas presents solutions for synthesizing interfaces between hardware components and between hardware and software components.

In [RoVi97], the authors proposed a design methodology in which the design of communication subsystems is separated from that of computational subsystems. This methodology, termed Interface-Based Design, incorporates a token passing abstraction for communication at the earliest stages of the design cycle. This allows high level simulations, with IP blocks, to be performed early in the design cycle without implementing low level communication logic. As the design moves to the implementation phase, a framework is provided for transforming the high level token passing scheme into actual bus protocols, such as PCI or EISA. In performing functionality-based partitioning (i.e. communication functionality and computational functionality), this work proposes to speed simulation and enable more effective synthesis and verification by completing these tasks with smaller, more well-defined blocks.

The research presented here focuses on generating low-level, synthesizable description of synchronous interfaces between hardware components. Although we focus on the synthesis of glue logic, hooks are included to easily control the interface at a high level. This provides a means for implementing static or dynamic schedules determined by higher-level optimization algorithms.

2.3.1 Interface Modeling Languages

Interface modeling languages allow a designer to explore the interface design space and generate a synthesizable description of the interface. The most commonly used industrial tool that encompasses this functionality is the Synopsys Protocol Compiler, based on the research of [SeHoMe96]. Protocol Compiler provides a framework for graphically defining an interface using state and data-frame based semantics. This specification is transformed into an HDL model and synthesized using traditional gate level synthesis techniques.

Many interface modeling languages are simply a means of representing Protocol Flow Graphs (e.g. AMICAL from [GaVaNa94]). PFGs describe how and when to provide input and receive output ([MaHa95]). Nodes in a PFG may indicate an operation to be performed (e.g. assignment), a decision to be made (e.g. a branch), or a synchronization point. Many high level optimization algorithms are based on abstract PFGs in which details of a node operation is hidden. Many lower-level interface synthesis techniques require concrete PFGs, in which the logic executed at each node is specified. Edges of a PFG indicate the dependencies between node operations.

The PRO-GRAM (protocol grammar) language, developed by [ObKuHe96], provides constructs that are particular to interfaces. These constructs include port definitions which declare the directions of interface signals, the width of communication busses and the frequency at which they operate. Communication operations are abstractions that are described in detail by the patterns of bits that are written to or read from a port during that operation. Grammar rules describe legal operation alternatives based on the state of the interface. PRO-GRAM also allows for the description of memory constructs for buffering data transactions.

2.3.2 Synthesis of Glue Logic

An algorithm for synthesizing glue logic is described in [PaRoSa98], enabling generation of interfaces between hardware blocks that implement incompatible protocols. Given a state-based description of each communicating component, this algorithm computes the product of the two state machines. This product represents a superset of the state machine that maps one component protocol onto the other and vice-versa. This set of states is pruned according to the following rules:

(1) Remove states that are the result of an illegal sequence of operations (prunes states that are the product of unrelated states).

(2) Remove states that output a piece of data that has not been received (i.e. states that introduce a data hazard).

(3) Remove redundant, non-minimum latency paths to the data transfer state.

Non-deterministic state transitions that remain after the above rules are exercised are resolved arbitrarily. This algorithm is guaranteed to return a minimum latency interface between two components. However, it requires that components perform data transfers in single transactions, rather than stream data, to prevent blowup of the product state machine. Furthermore, this algorithm restricts communication to two components that operate at the same frequency and bandwidth.

System level interface synthesis expands the problem of interface generation from communication between hardware components to communication between hardware and software components. Work presented in [ChOrBo95] utilizes detailed timing, operational, and bandwidth information to generate glue logic and device drivers that enable hardware/software communication. Specification information includes a control flow graph that describes systems components, including the processor, any peripheral devices and the desired operations. For example, for the processor, access routines and their associated I/O resources must be described. For peripherals, signal sequences that represent port assignments required to read and write the device must be described. Ports of communicating components must also be classified as address, data, or guard ports. Guard ports are control signals that must be activated to cause a component to be sensitive to address and data ports. From this information, processor I/O ports are assigned to peripheral ports and logic is generated that mimics the signal sequences. If an assignment to an I/O port can not be achieved (for example, in the case of a software block), logic generation is aborted and a memory-mapped I/O interface is attempted. Memory-mapped I/O requires a range of addresses be reserved for I/O operations. Load/store operations can then be performed by the processor to communicate with the peripheral device. This effectively allows an algorithm that is implemented in software to exercise hardware functionality through a device driver.

2.3.3 High Level Optimization of Communication

Optimization of communication at a high level can be separated into three problems: (1) determining which components can share busses and bus control signals; (2) determining the constraints imposed in communication; and (3) scheduling interface operations. This section will review related work that covers combinations of these three areas.

Much work has been presented on optimizing communication between subsystems given a set of communication constraints ([ErHeBe93], [JeElOb94], [KaLe94]). In [CoDe94], a subsystem's communication protocol and constraints are transformed from Verilog HDL to an algebraic systems called Control-Flow Expressions (CFEs). This algebra represents dataflow symbolically as either an action or condition and focuses on the control flow of the interface. Control flow constructs represent sequential operations, alternative operations, parallel operations, loops, and unconditional repetition. In addition, two special symbols represent no operation over a single cycle and a zero delay null operation. Constraints such as timing constraints, resource bindings, and synchronization are represented using the same control constructs. If the interface of each communicating component is described in common terms (i.e. the CFE symbol for an data read operation in one component is the same as the symbol for a data write operation in another component), then a CFE compactly and completely represents the possible operations that can be performed in communicating data between components. From this algebraic expression, a state machine that enables communication between components is derived using integer linear programming techniques.

Research presented in [GaGl96], investigated techniques for computing communication constraints, clustering data transfers of multiple components to a common bus, and determining an optimal schedule for bus accesses within each cluster. This is performed assuming a memory mapped communication structure in which communicating subsystems transfer information through a shared memory block. The communication paradigm is specified by high-level constructs such as *send* and *receive*. From this specification, analysis of mobility of send and receive operations within a control/data flow graph is performed. From the derived mobility rules, transfers that do not overlap are clustered. A minimum cost bus implementation is then determined for each cluster using branch and bound techniques. This requires computation of communication costs for many possible schedules. Once a minimum cost implementation is determined, these techniques attempt to merge generated busses to reduce the complexity of the implementation.

Rather than build a new specification language, [GuRo94] uses a subset of VHDL constructs to describe low level interface signal assignments in a manner that allows synthesis tradeoffs to performed at the behavioral level. This requires encapsulation of interface semantics within a VHDL procedure. These semantics are described by signal assignments and wait statements. This reduces the interface synthesis problem into a well defined scheduling problem that can be handled in high level synthesis. Constraints can be determined from data dependencies within the VHDL specification. The resulting schedule is then mapped into an interface architecture that is comprised of a resource set that includes registers, ALUs, busses, and multiplexors. By specifying the interface as a sequence of atomic operations, performance optimizations can be made at the behavioral level, rather than the logic level, having a greater impact on system performance. This provides a mechanism for partitioning interface functionality from algorithmic functionality and optimizing the sequence of steps required to perform a data transfer.

2.4 Comparison with Related Work

Polynomial representations are a stateless, word-level, functional description of existing components. The representation is computed assuming that a polynomial can represent the component's functionality. It also provides a mechanism for detecting components or parts of a component for which this assumption does not hold. Being a word-level representation, polynomial representations provide a much more efficient means for performing component matching than bit level representations do. Furthermore, not only are polynomial representations more efficient for performing matching than existing word-level structures, but they also provide a mechanism for approximating component functionality, performing inexact matching, and detecting control operations. Imprecise representations of component functionality potentially are more compact, enabling faster comparisons of specification and implementation, but allow for the possibility of inaccurate implementations of a specification and undetected implementation possibilities. Polynomial representations do become complex for control dominated components. However, control operations can be detected, as shown in Section 3.3, and then be represented by state-based structures or BDDs.

The interface synthesis algorithms presented here are geared to determine glue logic between two hardware components. These algorithms extend other glue logic synthesis techniques, as they allow synthesis of multi-way interfaces between components that operate at different frequencies and have multiple data transfer states. Synthesis of glue logic, in many cases, precludes the need for interface modeling languages, as the interface logic is automatically determined. Furthermore, the interface architecture proposed in this work provides hooks for implementing schedules and priorities determined by high-level optimization algorithms.

Chapter 3
Polynomial Methods

3.1 Introduction

Polynomial methods provide a means for comparing a specification, given at the arithmetic level, with components that are described at the bit level. To perform this comparison, a specification is partitioned into groups of arithmetic operations. Meanwhile, a polynomial representation is determined for a component, given its Boolean description. A numerical comparison is then performed between the groups of arithmetic operations extracted from the specification and the component's polynomial representation. This process is depicted in Figure 4. This chapter will focus on the task of determining the polynomial representation of a component given the Boolean equations that describe its functionality.

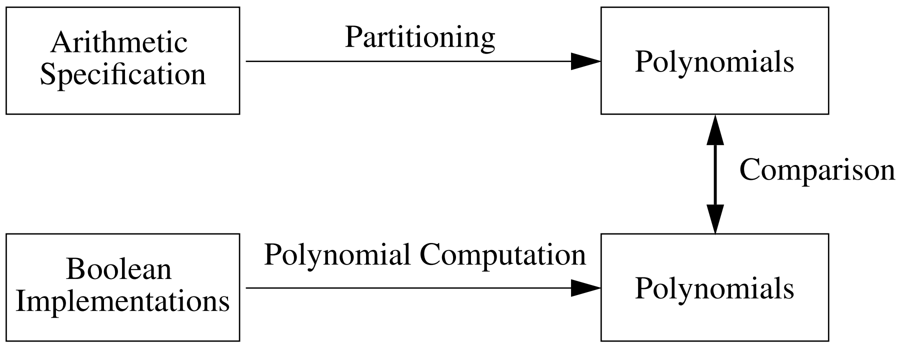


Fig. 4 Illustration of the application of polynomial methods

3.2 Polynomial Representations

To map an arithmetic specification to a complex element that is described at the logic level, a word-level polynomial that encapsulates the element's functionality is derived. To begin, we consider completely-specified Boolean functions for the sake of simplicity. This assumption will be removed in Section 3.4. Generating a word-level polynomial representation for a Boolean function may appear to be an inconsistent problem because Boolean functions are inherently discontinuous. However, a Boolean function, $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^{m} \rightarrow B^{k}$, $B = \{0, 1\}$, where **x** and **y** are bit vectors of length m and k, respectively, can be treated as a set of coordinates (x, y), where $x, y \in Z$:

$$x = \text{Encode}(\mathbf{x}) \qquad \mathbf{x} = \text{Decode}(x)$$

$$y = \text{Encode}(\mathbf{y}) \qquad \mathbf{y} = \text{Decode}(y)$$

Thus, "Encode" is an integer interpretation of a Boolean vector, such as two's complement or sign magnitude, and "Decode" is the inverse transformation. The following encoding examples will be referred to in succeeding sections:

$$0 = \text{Encode}(\mathbf{0}) \qquad \mathbf{0} = \text{Decode}(0) = 00...00$$

$$1 = \text{Encode}(\mathbf{1}) \qquad \mathbf{1} = \text{Decode}(1) = 00...01$$

-1 = Encode(**-1**) **-1** = Decode(-1) = 11...11

-2 = Encode(**-2**) **-2** = Decode(-2) = 11...10

A minimum order polynomial can be determined that fits the set of coordinates (x, y). 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.

Example 3.2.1 Consider a 3-input, 4-output combinational circuit that can be specified as the following set of coordinates (**x**, **y**) in the Boolean domain where $\mathbf{x} \in B^3$, $\mathbf{y} \in B^4$: {(000, 0000), (001, 0001), (010, 0100), (011, 1001)}. The corresponding encoding for $x, y \in Z$ is {(0,0), (1, 1), (2, 4), (3, 9)}. The minimum order polynomial that passes through these points is of order 2. Thus the polynomial representation is of the form $y = c_2x^2 + c_1x + c_0$. Extracting 3 coordinates yields the following set of linear equations:

$$0c_2 + 0c_1 + c_0 = 0$$

$$1c_2 + 1c_1 + c_0 = 1$$

$$4c_2 + 1c_1 + c_0 = 4$$

Solving this set of linear equations yields the polynomial representation $F(x) = x^2$. Thus, the circuit implements a squaring function.

3.2.1 Existence and Uniqueness

The following theorem is the basis for determining the polynomial representation of circuits described at the bit level. This theorem, derived from the binomial distribution from traditional calculus, is proven for integers and used to prove the existence and uniqueness of polynomial representations of Boolean functions.

Theorem 3.1

Given a polynomial function $F(x)$ of order n, where $x \in Z$, the function $F(x+1) - F(x)$ is of order exactly *n-1*.

Proof

Let $F(x) = \sum_{i=0}^{n} c_i \cdot x^i$

$$F(x+1) - F(x) = \sum_{i=0}^{n} c_i \cdot (x+1)^i - c_i \cdot x^i$$

Each term of order *i* in F(*x*) contributes a polynomial of order exactly *i-1* to *F(x+1) - F(x)*:

$$c_i (x+1)^i - c_i x^i = c_i \cdot \left(\sum_{j=0}^{i} \binom{i}{j} \cdot x^j - x^i \right) = c_i \left(\sum_{j=0}^{i-1} \binom{i}{j} \cdot x^j \right)$$

Thus, when *F(x+*1*) - F(x)* is computed, the polynomial term $c_n x^n$ of *F(x)* contributes a polynomial of order exactly *n-1* and is the only term to do so. Therefore, *F(x+*1*) - F(x)* is of order exactly *n-1*. ❑

Although this research will focus on integer encodings of Boolean vectors, note that Theorem 3.1 holds for any domain of *x* in which addition, subtraction, and multiplication are defined and the associative, distributive, and identity properties hold (e.g. $x \in R$). Furthermore, the theorem is independent of the details of the encoding of $\mathbf{x} \in B^m$ (e.g. two's complement, sign magnitude, fixed point, floating point). To illustrate Theorem 3.1 for $x \in Z$, note that if F(*x*) = x^3, then $F(x+1) - F(x) = x^3 + 3x^2 + 3x + 1 - x^3 = 3x^2 + 3x + 1$. From Theorem 3.1, a useful corollary can be derived.

Corollary 3.1.1

For all $x, m \in Z$, the following set of row vectors is linearly independent:

$$A = \begin{bmatrix} (x)^{m} & (x)^{m-1} & \dots & x^{0} \\ (x+1)^{m} & (x+1)^{m-1} & \dots & (x+1)^{0} \\ \dots & \dots & \dots & \dots \\ (x+m)^{m} & (x+m)^{m-1} & \dots & (x+m)^{0} \end{bmatrix}$$

Proof

The set of row vectors can be reduced by multiplying it by nonsingular matrices. The matrices shown in the following computation are nonsingular (the determinant of each is 1):

$$B = \begin{bmatrix} 1 & 0 & 0 & .. & 0 & 0 \\ -1 & 1 & 0 & .. & 0 & 0 \\ 0 & -1 & 1 & .. & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & .. & 0 & 0 \\ 0 & 1 & 0 & .. & 0 & 0 \\ 0 & -1 & 1 & .. & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & -1 & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & 0 & 0 & .. & 0 & 0 \\ 0 & 1 & 0 & .. & 0 & 0 \\ 0 & 0 & 1 & .. & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & -1 & 1 \end{bmatrix} \begin{bmatrix} (x)^{m} & (x)^{m-1} & .. & x^{0} \\ (x+1)^{m} & (x+1)^{m-1} & .. & (x+1)^{0} \\ .. & .. & .. & .. \\ (x+m)^{m} & (x+m)^{m-1} & .. & (x+m)^{0} \end{bmatrix}$$

$$= \begin{bmatrix} (x)^{m} & (x)^{m-1} & \dots & x^{0} \\ (x+1)^{m} - (x)^{m} & (x+1)^{m-1} - (x)^{m-1} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ (x+m)^{m} - \left(\binom{m}{1} \cdot (x+m-1)^{m}\right) + \dots + (-1)^{m} & 0 & \dots & 0 \end{bmatrix}$$

The rows of matrix B are linearly independent. Therefore the original set of vectors A are linearly independent ❑.

Example 3.2.2 To illustrate Corollary 3.1.1, notice that, for $x = 0$ and m = 3:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 8 & 4 & 2 & 1 \\ 27 & 9 & 3 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 7 & 3 & 1 & 0 \\ 19 & 5 & 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 6 & 2 & 0 & 0 \\ 12 & 2 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 6 & 2 & 0 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix}$$

Thus, the initial set of vectors is linearly independent.

The following theorems establish the existence of polynomial representations for combinational univariate functions and the uniqueness of the minimum order polynomial representation.

Theorem 3.2 (*Existence*)

Let $\mathbf{x} \in B^m$, $\mathbf{y} \in B^k$, and $x, y \in Z$ be the integers corresponding to **x**, **y**. Given a Boolean function **y = F(x)**: $B^m \to B^k$, there exists a polynomial $y = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_0$, where $n < 2^m$, that defines the corresponding function F: $x \to y$.

Proof

If $\mathbf{x} \in B^m$, then there are 2^m possible values that x can take on $\{0, 1, \ldots, 2^m-1\}$ and 2^m corresponding values that y can take on $\{F(0), F(1), \ldots, F(2^m-1)\}$. The solution to the set of linear equations ($\mu = 2^m-1$):

$$\begin{bmatrix} (0)^{\mu} & (0)^{\mu-1} & \ldots & 1 \\ (1)^{\mu} & (1)^{\mu-1} & \ldots & (1)^0 \\ \ldots & \ldots & \ldots & \ldots \\ (\mu)^{\mu} & (\mu)^{\mu-1} & \ldots & (\mu)^0 \end{bmatrix} \bullet \begin{bmatrix} c_{\mu} \\ c_{\mu-1} \\ \ldots \\ c_0 \end{bmatrix} = \begin{bmatrix} F(0) \\ F(1) \\ \ldots \\ F(\mu) \end{bmatrix}$$

exists if no row of the matrix:

$$\begin{bmatrix} (0)^{\mu} & (0)^{\mu-1} & \ldots & 1 \\ (1)^{\mu} & (1)^{\mu-1} & \ldots & (1)^0 \\ \ldots & \ldots & \ldots & \ldots \\ (\mu)^{\mu} & (\mu)^{\mu-1} & \ldots & (\mu)^0 \end{bmatrix}$$

is a linear combination of the others. We know this is true from Corollary 3.1.1. Note that the dimension of **y** does not affect the polynomial representation of **F**. ❑

Theorem 3.3 (*Uniqueness*)

The minimum order polynomial representation of a Boolean function **y = F(x)**: $B^m \to B^k$, is unique.

Proof

Assume there exist two minimum order polynomial representations for **F(x)**, where $x, y \in Z$ are the integers corresponding to **x**, **y**:

$$y = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$$

$$y = b_n x^n + b_{n-1} x^{n-1} + \ldots + b_0$$

=> there are two possible solutions to the set of linear equations:

$$\begin{bmatrix} (0)^n & (0)^{n-1} & \ldots & 1 \\ (1)^n & (1)^{n-1} & \ldots & (1)^0 \\ \ldots & \ldots & \ldots & \ldots \\ (n)^n & (n)^{n-1} & \ldots & (n)^0 \end{bmatrix} \bullet \begin{bmatrix} c_n \\ c_{n-1} \\ \ldots \\ c_0 \end{bmatrix} = \begin{bmatrix} \text{Encode}(\mathbf{F}(00...00)) \\ \text{Encode}(\mathbf{F}(00...01)) \\ \ldots \\ \text{Encode}(\mathbf{F}(\text{Decode}(n))) \end{bmatrix}$$

=> there exists a row in the matrix:

$$\begin{bmatrix} (0)^n & (0)^{n-1} & \ldots & 1 \\ (1)^n & (1)^{n-1} & \ldots & (1)^0 \\ \ldots & \ldots & \ldots & \ldots \\ (n)^n & (n)^{n-1} & \ldots & (n)^0 \end{bmatrix}$$

that is a linear combination of the others. But from Corollary 3.1.1 we know that this is not possible and we have a contradiction. Therefore, the minimum order polynomial is unique.❑

Example 3.2.3 An example of the application of Theorems 3.2 and 3.3 is the following set of Boolean equations (input width $m = 2$ and output width $k = 5$) that model an existing circuit:

$$F_0(x) = x_0$$

$$F_1(x) = x_1 \cdot x_0$$

$$F_2(x) = 0$$

$$F_3(x) = x_1$$

$$F_4(x) = x_1 \cdot x_0$$

$y = x^3$ is the unique, minimum order polynomial ($n = 3$) that represents this circuit, and would match a specification that requires the computation of the third power of x.

3.2.2 Polynomial Computation

In the previous section, we have proven that any combinational circuit can be uniquely represented by a minimum-order polynomial. Once the order of this polynomial is determined, then the coefficients of the polynomial can be calculated by examining a finite number of circuit outputs. Thus, the problem of determining a canonical polynomial representation for a circuit can be reduced to finding the order of the polynomial that represents that circuit.

To begin deriving a method for determining the order of a Boolean function, remember from Theorem 3.2 that a polynomial representation $F(x)$, where $x \in Z$, always exists for a Boolean function **y = F(x)**: $B^m \rightarrow B^k$, Furthermore, from Theorem 3.1, we might deduce that the order of **F(x)** will be reduced by exactly one by computing **F(x+1) - F(x)**. Therefore, the order of *F(x)* could be determined exactly by recursively performing

F(x) = F(x+1) - F(x) until this difference is identically zero for all values of **x**. In the algorithm discussed here, two's complement arithmetic is employed to compute this difference. The number of iterations required to set **F(x+1) - F(x) = 0** is the order of the unique, minimum-order polynomial *F(x)* that represents the circuit.

In computing the order of a Boolean function, we assume that each output bit (y_0, y_1, ..., y_{k-1}) of the function **y = F(x)** is represented as a Binary Decision Diagram. While this does present an exponentially-sized data structure for some functions, we will show a heuristic in Section 4.6 that reduces this data structure to linear complexity with respect to the number of input bits. In Sections 3.2.2.1 to 3.2.2.4, we derive, in detail, the steps required to compute **F(x+1) - F(x)** and determine if **F(x+1) - F(x) = 0**. These sections provide the rationale for the order computation algorithm shown in Figure 7.

3.2.2.1 Determining **F(x+1)**

The first step in computing **F(x+1) - F(x)** is to determine **F(x+1)**. This can be performed in polynomial time by replacing each bit {x_i: i = 1, 2, ..., m-1) of **x** with $(x_i \oplus x_{i-1} \cdot x_{i-2} \cdot \ldots \cdot x_0)$ and x_0 by x_0' in the BDD of **F(x)**.

3.2.2.2 Determining **-F(x)**

The next step in computing **F(x+1) - F(x)** is determining **-F(x)**. Using two's complement arithmetic, this could be performed by inverting each output bit F_i(**x**) of **F(x)** and adding one (**-F(x) = F'(x) + 1**, where **F'(x)** is the bitwise complement of **F(x)** and **1** is the vector 00..01). Computation of **F'(x)** is simple as it only requires inverting each leaf of each BDD that represents the output F_i(**x**). However, if we make the assumptions that **F(x)** is an *m* bit function, **x** is an *m* bit word, and the BDD of F_i(**x**) has at least *m* nodes, computing **F'(x) + 1** is of complexity $O(m^4)$, due to the propagation of the carry (carry computation requires $m(m+1)/2$ logic operations each of which is of complexity m^2).

To reduce the complexity the negation, we transform the problem of recursively computing **F(x) = F(x+1) - F(x)** until **F(x) = 0** to the problem of recursively computing **F(x) = F(x+1) + F'(x)** until **F(x) = -1**. This is the equivalent of computing *F(x+1) - F(x) - 1* in two's complement encoding. This computation reduces the order of **F(x)** by one on each iteration, but avoids the complexity introduced by incrementation. This is possible because, on successive computations of *F(x+1) - F(x) - 1*, the subtraction of one does not accumulate:

1st iteration: *F(x) = F(x+1) - F(x) - 1*

2nd iteration: *F(x) = (F(x+2) - F(x+1) - 1) - (F(x+1) - F(x) - 1) - 1*

= (F(x+2) - F(x+1)) - (F(x+1) - F(x)) - 1

Thus, instead of computing **F(x+1) - F(x)** to reduce the order of **F(x)** by one, we compute **F(x+1) + F'(x)** which is a computationally simpler way to reduce the order of **F(x)** by one.

3.2.2.3 *Performing* **F(x+1) + F'(x)**

Once **F(x+1)** and **F'(x)** have been determined, the two functions are summed to produce the new, reduced order **F(x)**. If this summation is performed in ripple carry fashion, it is an exponentially complex operation with respect to word length, due to the propagation of the carry (for the i^{th} bit, the carry computation requires 3^i logic operations). To eliminate the computation of the carry, a carry-save addition can be performed. Let us define:

$$F_{sum}(x) = \mathbf{F(x+1)} \oplus \mathbf{F'(x)}$$

$$F_{carry}(x) = \mathbf{F(x + 1)} \cdot \mathbf{F'(x)}$$

where $\oplus$ and $\cdot$ are applied bitwise. Thus, **F(x)** is uniquely specified as:

$$F(x) = F_{sum}(x) + (F_{carry}(x) << 1)$$

Note that there are now two terms that must be complemented when recursively computing **F(x) = F(x+1) + F'(x)**. These terms are $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$<<1. Complementing both terms requires, according to two's complement arithmetic, a bitwise inversion and an increment of each term. In order to avoid these increments and their associated carry operations, order reduction can be performed by recursively computing:

$$\mathbf{F(x) = F_{sum}(x+1) + (F_{carry}(x+1)} << 1) + \mathbf{F'_{sum}(x) + (F'_{carry}(x)} << 1)$$

until **F(x) = -2**. The condition for terminating recursion has changed to **F(x) = -2** because the equivalent computation in two's complement arithmetic is:

$$F_{sum}(x+1) + (F_{carry}(x+1) << 1) - (F_{sum}(x) + (F_{carry}(x) << 1)) - 2$$

$$= F(x+1) - F(x) - 2$$

Since **F(x+1)** and **F'(x)** are specified as the summation of a sum and carry term, their summation can be performed in two steps, as if two carry-save additions (Figure 5) were executed.

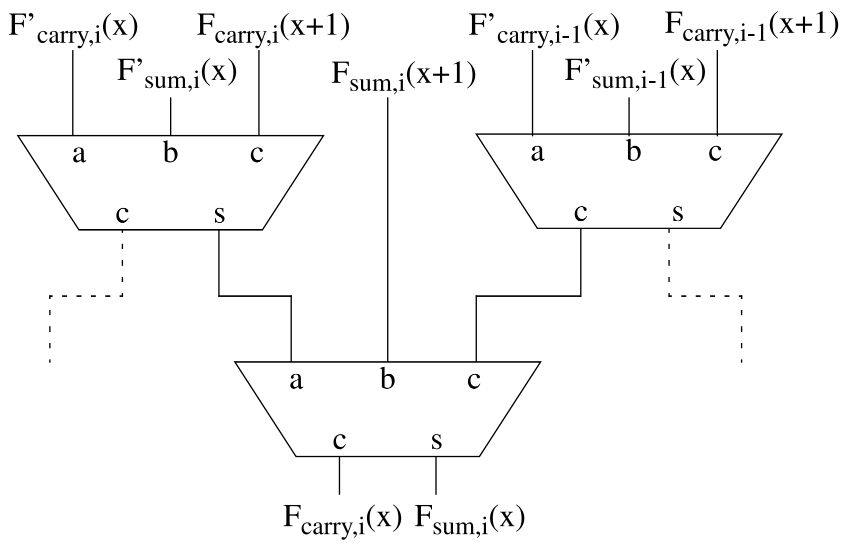


Fig. 5 Physical visualization of the two stage carry-save addition for computation of **F(x+1) + F'(x)**

With these transformations, the order of **F(x)** is successively being reduced by one by recursively computing **F(x) = F(x+1) + F'(x)**. This computation is of polynomial complexity with respect to the size of the BDD representation of **F(x)**.

3.2.2.4 Checking if **F(x) = -2**

Using a two's complement encoding, the following transformations can be used to determine if the recursively computed **F(x) = -2**, without performing a ripple carry addition:

$$F(x) = -2$$

$$\Leftrightarrow \mathbf{F_{sum}(x) + (F_{carry}(x) \ll 1) = -2}$$

$$\Leftrightarrow \mathbf{F_{sum}(x) + (F_{carry}(x) \ll 1) + 1 = -1}$$

To avoid performing the ripple carry addition, a two-stage carry-save increment is performed at the end of each recursive step:

$$\mathbf{F_{sum}(x) + (F_{carry}(x) \ll 1) + 1 = S_{test} + C_{test}}$$

by performing the following logic operations (i = 1, 2, ... , k-1):

$$S_{test_0}(\mathbf{x}) = F'_{sum_0}(\mathbf{x})$$

$$C_{test_0}(\mathbf{x}) = F_{sum_0}(\mathbf{x})$$

$$S_{test_i}(\mathbf{x}) = \left(F_{sum_i}(\mathbf{x}) \oplus F'_{carry_{i-1}}(\mathbf{x})\right) \oplus \left(F_{sum_{i-1}}(\mathbf{x}) + F_{carry_{i-2}}(\mathbf{x})\right)$$

$$C_{test_i}(\mathbf{x}) = \left(F_{sum_i}(\mathbf{x}) \oplus F'_{carry_{i-1}}(\mathbf{x})\right) \cdot \left(F_{sum_{i-1}}(\mathbf{x}) + F_{carry_{i-2}}(\mathbf{x})\right)$$

Each bit of the resulting sum ($\mathbf{S_{test}}$) is checked for tautology and each bit of the resulting carry ($\mathbf{C_{test}}$) is checked whether it is tautologically zero. We refer to this test as the *tautology check*, and it is necessary and sufficient to guarantee $\mathbf{F_{sum}(x) + (F_{carry}(x)} \ll 1) \mathbf{+ 1 = - 1}$ as proven in Theorem 3.4. As a result, the ripple carry computation does not need to be performed.

Theorem 3.4

Given three Boolean vectors $\mathbf{G_{sum}}, \mathbf{G_{carry}}, \mathbf{G} \in B^k$, where $\mathbf{G = G_{sum} + (G_{carry}}\ll 1)$, then $\mathbf{G = -1}$ iff $G_{sum_0} = 1$, $G_{sum_i} \oplus G_{carry_{i-1}} = 1$ and $G_{sum_i} \cdot G_{carry_{i-1}} = 0$ for all i = 1, 2, ..., k-1.

Proof

Forward implication (by induction):

Base Case: $\mathbf{G = G_{sum} + (G_{carry}}\ll 1) \Rightarrow G_0 = G_{sum_0}$ and $G_1 = G_{sum_1} \oplus G_{carry_0}$

$\mathbf{G = -1} \Rightarrow G_0 = 1 \Rightarrow G_{sum_0} = 1$

$\mathbf{G = -1} \Rightarrow G_1 = 1 \Rightarrow G_{sum_1} \oplus G_{carry_0} = 1$ and $G_{sum_1} \cdot G_{carry_0} = 0$

Assume: $G_{sum_j} \oplus G_{carry_{j-1}} = 1$ and $G_{sum_j} \cdot G_{carry_{j-1}} = 0$ for all $j \leq i$

Inductive step: $G_{j+1} = 1$ and $G_{sum_j} \cdot G_{carry_{j-1}} = 0$ for all $j \leq i$

=> $G_{sum_{j+1}} \oplus G_{carry_j} = 1$ and $G_{sum_{j+1}} \cdot G_{carry_j} = 0$.

Reverse implication:

$G_{sum_0} = 1 \Rightarrow G_0 = 1$.

$G_{sum_i} \oplus G_{carry_{i-1}} = 1$, $G_{sum_i} \cdot G_{carry_{i-1}} = 0$ for all i => $G_i = 1$.

=> **G = -1**. ❑

The following assignments allow Theorem 3.4 to be used to perform the tautology check:

$$G_{sum_i}(\mathbf{x}) = \left(F_{sum_i}(\mathbf{x}) \oplus F'_{carry_{i-1}}(\mathbf{x})\right)$$

$$G_{carry_i}(\mathbf{x}) = \left(F_{sum_{i-1}}(\mathbf{x}) + F_{carry_{i-2}}(\mathbf{x})\right)$$

$$S_{test_i}(\mathbf{x}) = G_{sum_i}(\mathbf{x}) \oplus G_{carry_{i-1}}(\mathbf{x})$$

$$C_{test_i}(\mathbf{x}) = G_{sum_i}(\mathbf{x}) \cdot G_{carry_{i-1}}(\mathbf{x}).$$

In summary, **F(x) = -2** if and only if $S_{test_i}(\mathbf{x}) = 1$ and $C_{test_i}(\mathbf{x}) = 0$ for all i = 0, 1, ..., k-1.

3.2.2.5 Bounding Function

A function *y* = *F(x):* $Z \rightarrow Z$ has a corresponding Boolean function **y = F(x)**: $B^m \rightarrow B^k$, **x** = Decode(*x*), and **y** = Decode(*y*), defined only over the domain [0, 2^m-1]. This is important to consider when performing order computations because **F(x+1) - F(x)** actually corresponds to $F(0) - F(2^m-1)$ if **x = -1** (e.g. 11...11). In performing order computations, this may result in *F(x)* appearing to be non polynomial over the domain $[-\infty,\infty]$ even if **F(x)** does have a polynomial representation over the range of possible values for **x** (Figure 6). Thus, in executing order computations, it is necessary to determine a bounding function that specifies which values do not need to be considered

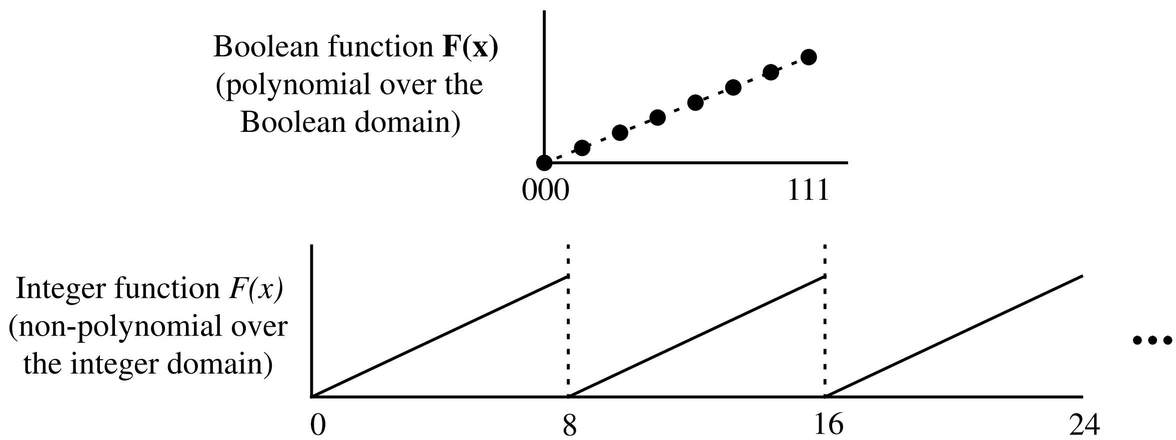


Fig. 6 A Boolean function that is polynomial that appears to be non-polynomial in the integer domain

when performing tautology checks.

Definition 3.1 Given a function **F(x)**, where $\mathbf{x} \in B^m$, the bounding function B(**x**) on the nth order iteration is:

$$B(\mathbf{x}) = \sum_{i = 2^m - n}^{2^m - 1} (\mathbf{x} = \mathrm{Decode}(i))$$

In words, this is the sum of the Boolean vectors whose corresponding integer values are greater than 2^m-n. For example, after one iteration of order reduction with respect to an *m* bit vector **x**, the bounding function would be B = $x_{m-1} \cdot x_{m-2} \cdot \ldots \cdot x_0$. After two iterations, the bounding function would be B $= x_{m-1} \cdot x_{m-2} \cdot \ldots \cdot x_0$ + $x_{m-1} \cdot x_{m-2} \cdot \ldots \cdot x'_0$.

If the input is out of range when incremented, i.e. **x** = 11...11, then the resulting **F(x+1) - F(x)** is immaterial, since the input pattern can not be applied. Thus, **F(x+1) + F'(x) = -1** requires that if $\mathbf{S_{test}}$ is not a tautology, the bounding function must be true. Similarly, if $\mathbf{C_{test}}$ is not tautologically zero, the bounding function must be true if **F(x+1) + F'(x) = -1**. The tautology check requires that:

$$\left(S_{test_i}(\mathbf{x}) + B(\mathbf{x})\right) \cdot \left(C'_{test_i}(\mathbf{x}) + B(\mathbf{x})\right) = 1 \text{ for all i= 0, 1, ... , k-1.}$$

Example 3.2.4 If, after two order computations, $S_{test_0}(\mathbf{x}) = (x_{m-1} \cdot x_{m-2} \cdot \ldots \cdot x_0)'$ and all other bits of $\mathbf{S_{test}}$ and $\mathbf{C'_{test}}$ are a tautology, then $S_{test_0}(\mathbf{x})$ + B = $(x_{m-1} \cdot x_{m-2} \cdot \ldots \cdot x_0)' + x_{m-1} \cdot x_{m-2} \cdot \ldots \cdot x_0 + x_{m-1} \cdot x_{m-2} \cdot \ldots \cdot x'_0 = 1$ and the bit satisfies the tautology check. Thus, within the interval x = [0, 2^m-1], the original Boolean function **F(x)** is of order 2.

3.2.2.6 The Complete Algorithm

The complete algorithm for computing the order of a Boolean function **F(x)**, given its BDD representation, is shown in Figure 7. Step (1) initializes the function $\mathbf{F_{sum}(x)}$ to **F(x)**

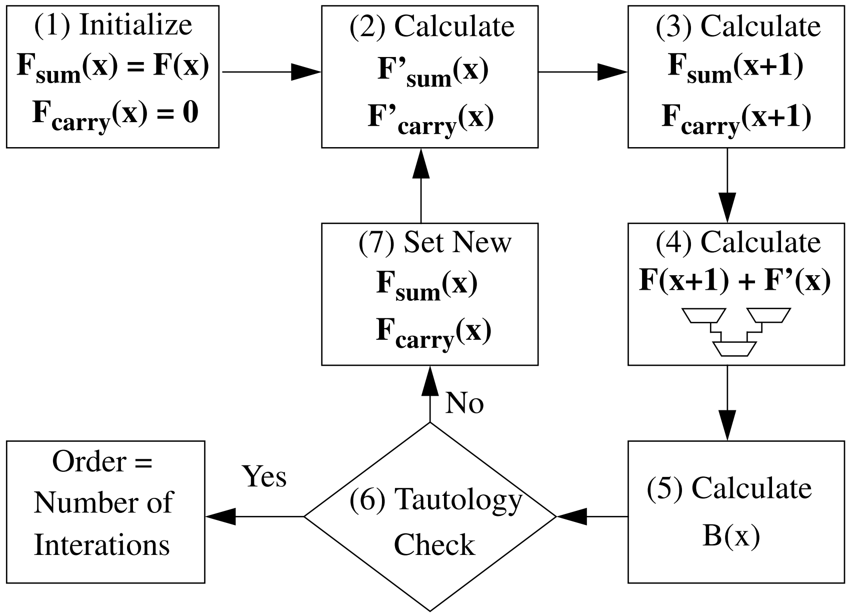


Fig. 7 Algorithm for computing the order of a Boolean function **F(x)**

and the function $\mathbf{F_{carry}(x)}$ to **0**, an operation of linear complexity with respect to the size of the BDD representation of **F(x)**. Step (2) computes **F'(x)** by complementing $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$, an operation of constant complexity with respect to BDD size. Step (3) computes the function **F(x+1)** by replacing **x** with **x+1** in the functions $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$, an operation of quadratic complexity with respect to BDD size. Step (4) then reduces the order of **F(x)** by exactly one by computing the sum **F(x+1)** + **F'(x)**. This computation is performed by adding the results of Steps (2) and (3) with a two-stage carry-save addition, producing a new $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$. This step is of quadratic complexity with respect to BDD size. Step (5) computes the bounding function B(**x**) that restricts the domain over which the sum **F(x+1)** + **F'(x)** is evaluated, an operation that is of constant complexity relative to BDD size. Step (6) then checks the sum **F(x+1)** + **F'(x)** to see if each output bit is a tautology within the bounds specified by B(**x**), an operation of constant complexity with respect to BDD size. If the tautology check is unsuccessful, then Step (7) sets $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$ to the result of Step (5) and initiates a new recursion, an operation of linear complexity with respect to BDD size. Otherwise, the order of the minimum order polynomial representation is one less than the number of recursive computations that were performed.

Example 3.2.5 Consider the function **y = F(x)**, where $\mathbf{x} \in B^2$ and $\mathbf{y} \in B^5$, that implements $F(x) = x^2$. Initializing the sum **s** to **F(x)** and the carry **c** to zero yields the following input vectors:

$s_0 = x_0$ $c_0 = 0$

$s_1 = 0$ $c_1 = 0$

$s_2 = x_0' \cdot x_1$ $c_2 = 0$

$s_3 = x_0 \cdot x_1$ $c_3 = 0$

$s_4 = 0$ $c_4 = 0$

The following steps are followed to determine the order of these input vectors:

(1) **F(x+1)**:

$s_0 = x_0'$ $c_0 = 0$

$s_1 = 0$ $c_1 = 0$

$s_2 = x_0 \cdot (x_1 \oplus x_0)$ $c_2 = 0$

$s_3 = x_0' \cdot (x_1 \oplus x_0)$ $c_3 = 0$

$s_4 = 0$ $c_4 = 0$

(2) **F'(x)**:

$s_0 = x_0'$ $c_0 = 1$

$s_1 = 1$ $c_1 = 1$

$s_2 = x_0 + x_1'$ $c_2 = 1$

$s_3 = x_0' + x_1'$ $c_3 = 1$

$s_4 = 1$ $c_4 = 1$

(3) **F(x+1) - F(x)** (1st iteration)

$s_0 = 1$ $c_0 = 0$

$s_1 = x_0'$ $c_1 = 1$

$s_2 = x_1 \oplus x_0$ $c_2 = x_1' \oplus x_0$

$s_3 = x_0 + x_1$ $c_3 = x_1'$

$s_4 = x_0' \cdot x_1$ $c_4 = 1$

(4) Tautology Check

$s_0 = 0$ $c_0 = 0$ fails

(5) **F(x+1) - F(x)** (2nd iteration)

$s_0 = 0$ $c_0 = 1$

$s_1 = 1$ $c_1 = 0$

$s_2 = 1$ $c_2 = 0$

$s_3 = x_0' + x_1'$ $c_3 = x_0$

$s_4 = x_0' \oplus x_1$ $c_4 = x_0' \cdot x_1$

(6) Tautology Check

$s_0 = 1$ $c_0 = 0$

$s_1 = 0$ $c_1 = 0$ fails

(7) **F(x+1) - F(x)** (3rd iteration)

$s_0 = 0$ $c_0 = 1$

$s_1 = 0$ $c_1 = 0$

$s_2 = 1$ $c_2 = 1$

$s_3 = x_1$ $c_3 = x_1'$

$s_4 = x_0 \cdot x_1'$ $c_4 = x_0 + x_1$

(8) Tautology Check

$s_0 = 1$ $c_0 = 0$

$s_1 = 1$ $c_1 = 0$

$s_2 = 1$ $c_2 = 0$

$s_3 = 1$ $c_3 = 0$

$s_4 = 1$ $c_4 = 0$

Three iterations reduce **F(x)** to 0 for all **x**. Thus, **F(x)** is of order 2.

Each step within the order computation algorithm is of polynomial complexity with respect to the number of nodes in the BDD representation of **F(x)**. However, the minimum-order polynomial representation may be of exponential order with respect to the number of bits in the input word **x**. Thus, the number of recursions that are performed may be exponential. Sections 3.3 and 4.4 detail partitioning and approximation algorithms for efficiently generating polynomial representations for those circuits whose representations would otherwise be of exponential order.

Once the order of the function has been determined to be n, **F(x)** is evaluated at **x** = 00...00, **x** = 00...01, ... , **x** = Decode(n). Solving the following set of linear equations for c_0, c_1, ... , c_n yields the polynomial representation of the Boolean function:

$$\begin{bmatrix} (0)^n & (0)^{n-1} & \ldots & 1 \\ (1)^n & (1)^{n-1} & \ldots & (1)^0 \\ \ldots & \ldots & \ldots & \ldots \\ (n)^n & (n)^{n-1} & \ldots & (n)^0 \end{bmatrix} \bullet \begin{bmatrix} c_n \\ c_{n-1} \\ \ldots \\ c_0 \end{bmatrix} = \begin{bmatrix} \text{Encode}(\mathbf{F}(00...00)) \\ \text{Encode}(\mathbf{F}(00...01)) \\ \ldots \\ \text{Encode}(\mathbf{F}(\text{Decode}(n))) \end{bmatrix}$$

3.2.3 Extension to Multivariable Functions

The techniques described above consider only univariable functions. However, multivariable polynomials exhibit the same features that allow order computation to be performed recursively. That is, **F(x, y) = F(x+1, y) + F’(x, y)** recursively reduces the order of **F(x, y)** with respect to **x** by one on each iteration if **y** is held constant. Thus, the order of **F(x, y)** can be determined with respect to **x** and with respect **y**. However, the unique, minimum-order polynomial computation requires solving a set of $n_x n_y$ simultaneous linear equations, where n_x is the order with respect to **x** and n_y is the order with respect to **y**.

3.3 Representation of Functions Containing Branches

To this point, the methods we have described allow computation of a polynomial representation for combinational circuits. As proven in Theorem 3.2, polynomial representations exist for all combinational circuits. For those circuits that implement arithmetic functions, such as those generated by composing addition and multiplication operations, this representation is of very low order (e.g. one term to represent multiplication, two terms to represent addition). Consider, however, models of combinational circuits that contain branches, i.e. discontinuities. For such circuits, polynomial representations, if computed using only the techniques described above, are usually of exponential order with respect to input word size. This is because a branch in the Boolean domain usually describes a set of coordinates in the integer domain that can only be fit to an exponentially-large polynomial. However, a high order polynomial representation is an indicator that a branch exists within a circuit. This indicator can be use to partition circuit inputs into domains in which polynomial representations of low complexity exist. The boundaries of these domains are termed *discontinuities*.

Example 3.3.1 Consider the JPEG Coefficient Encoder **coefficient = F(q)**, with a 16 bit input and 4 bit output, which selects an output based on the range of the quantized input values.

```
if (q == 0000000000000000) coefficient = 0000;
else if (q < 00000000000000010) coefficient = 0001;
else if (q < 00000000000000100) coefficient = 0010;
...
else coefficient = 1111;
```

The encoder is performing an operation within each branch that is represented by polynomials of order zero. However, using the order computation methods described above, the discontinuities at the integer values $q = 2^i$ cause the overall circuit to have a polynomial representation of order 2^{16}.

To prevent an exponential number of order computation recursions from being performed on functions that contain branches, we use a heuristic based on a *discontinuity threshold*. Once the number of iterations has reached this threshold, the function is assumed to contain branches. The threshold is determined heuristically and enables efficient detection of discontinuities. Discontinuity detection, in turn allows order computation to be performed on each branch of the circuit model.

Given a function **y = F(x)**: $B^m \rightarrow B^k$, with order greater than the discontinuity threshold, discontinuities can be detected by performing order computation on **F(x)** for the case $x_{m-1} = 0$ and the case $x_{m-1} = 1$. If the orders for each computation are different, and below the discontinuity threshold, a discontinuity has been detected and exists between **x** = 01...11 and **x** = 10...00. If the order of **F(x)**, for $x_{m-1} = 0$ or $x_{m-1} = 1$, is still above the threshold, then a discontinuity exists within the corresponding domain. Within that domain, an order computation is then performed on **F(x)** for the case $x_{m-2} = 0$ and the case $x_{m-2} = 1$. Domain partitioning continues until the discontinuity is detected.

Similar to performing a binary search, detection of a single discontinuity is of linear complexity with respect to the number of input bits, not considering the complexity of the order computation.

Example 3.3.2 Consider the function **y = F(x),** where $\mathbf{x} \in B^4$, that is implemented by the following Verilog code:

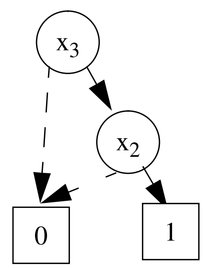
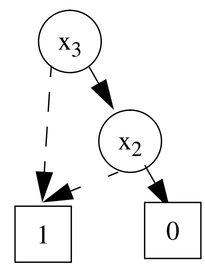
```
if (x > 4'b1011)
```

```
then y = x*x*x;
else y = x*x;
```

If we proceed blindly, computing the order of **F(x)** will generate an order of 2^4 because of the discontinuity at **x** = 1011. However, if we start with an initial discontinuity threshold of 4, then after four order iterations, the uppermost bit of x will be set to zero, then one, and the order computations will be performed for each case. The order computation for $x_3 = 0$ will result in an order of 2. The order computation for $x_3 = 1$ will again reach the fourth iteration without passing the tautology check. The second most significant bit is set to zero, then one, and the order computation is performed again. Then order computation for $x_3x_2 = 11$ will result in an order of 3 and the computation for $x_3x_2 = 10$ will result in an order of 2. Since both computations converged, but converged to different values, there is a discontinuity on the interval boundary. Thus, over the integer interval [0, 11] an order of 2 is determined and over the intriguer interval [12, 15] an order of 3 is determined.

Performing discontinuity detection is the equivalent of detecting the control operations of a circuit. Each discontinuity represents a branch within a circuit. This effectively splits a circuit into control operations (the conditions under which a branch is executed) and datapath operations (the functionality executed within a branch). Thus, a high order polynomial is indicative of control operations while a low order polynomial is indicative of datapath operations. Each operation type can then be represented by the structure that is smallest in size for that class of operations. The conditions under which a branch is executed can be represented by a BDD, while the branch functionality is represented by a polynomial.

Example 3.3.3 The function of Example 3.3.2 has been partitioned into two branches. The BDDs that represent the conditions for executing each branch comprise the control operations. The polynomials that represent the functionality within each branch represent datapath the datapath operations:

	Branch 0	Branch 1
Control BDD		
Datapath Polynomial	$y = x^3$	$y = x^2$

3.4 Don't Care Sets

Sub-blocks within a specification are frequently conditionally executed. Example 3.2.2 illustrated a specification in which the operation x^3 is performed when $x > 11$ and x^2 is performed when $x < 12$. If no library element matches the complete specification, it may be advantageous to map the blocks performing x^3 and x^2 to existing components. However, an existing component need not exactly match the polynomial functionality of the sub-blocks. Under conditions during which the sub-blocks are not executed, the output of a component used to implement that functionality is immaterial. This is the equivalent of an Controllability Don't Care (CDC) set. Similarly, output values of the sub-block for which the output is immaterial are the equivalent of the Observability Don't Care set (ODC). These relaxed functionality constraints can be incorporated into the computation of the polynomial representation.

When bounding the domain over which the polynomial representation was computed (Section 3.2.2.5), the tautology check was performed on the function:

$$\left(S_{test_i}(\mathbf{x}) + B(\mathbf{x}) \right) \cdot \left(C'_{test_i}(\mathbf{x}) + B(\mathbf{x}) \right)$$

This equation restricts order computation to the input domain that does not include those values that satisfy B(**x**). Similarly, CDC(**x**) defines values of the input domain that do not need to be considered when performing order computation. Thus, the CDC set can be accounted for by performing the tautology check on the following equation:

$$\left(CDC(\mathbf{x}) + S_{test_i}(\mathbf{x}) + B(\mathbf{x}) \right) \cdot \left(CDC(\mathbf{x}) + C'_{test_i}(\mathbf{x}) + B(\mathbf{x}) \right)$$

The ODC set is a function of output values. Thus, to incorporate ODC(**F**) into the tautology check computation, each bit of **F** must be replaced in the ODC expression with the BDD that describes that bit, resulting in the expression ODC(**x**). Thus, the following tautology check takes into account Don't Care sets:

$$\left(ODC(\mathbf{x}) + CDC(\mathbf{x}) + S_{test_i}(\mathbf{x}) + B(\mathbf{x}) \right) \cdot \left(ODC(\mathbf{x}) + CDC(\mathbf{x}) + C'_{test_i}(\mathbf{x}) + B(\mathbf{x}) \right)$$

3.5 Summary

Design reuse is an effective method for reducing the time required to design large systems. However, in order automate such a methodology, a means of representing reusable components compactly and canonically is required. In addition, implementation of a large system is made easier if component representations provide rules for composition, allowing synthesis of larger systems from individual components.

The techniques presented in this chapter allow construction of mathematical, specifically polynomial, models of existing components given only a logic level description of the component's functionality. Polynomial models are guaranteed to exist for all combinational blocks and each block is guaranteed to have a unique representation. By generating a polynomial representation for an existing component, comparison with a mathematical specification can be performed at the word level, speeding the allocation process. These techniques can be applied to components that perform control and datapath operations, through domain partitioning. In addition, components with any number of input words can be represented by polynomial models.

Computation of polynomial representations can performed by determining the order of the minimum order representation. This figure is then used to extract the appropriate number of coordinates from a component to compute polynomial coefficients. Separation of control from datapath operations within a component can be performed based on the order of polynomial used to represent a component or sub-block of a component. High order representations indicate that a block contains control functionality while low order representation are indicative of datapath operations.

The techniques presented to this point are applicable to combinational blocks. Introduction of synchronous elements in a design removes the guarantee of the existence of a polynomial representation, though not the uniqueness of those that do exist. Furthermore, although the techniques presented to this point are of quadratic complexity with respect to the size of the component's BDD, the input domain may be partitioned an exponential number of times for certain classes of functions. The resulting polynomial representation would then be exponentially complex. Chapter 4 will present solutions to the limitations of the techniques presented in this chapter.

Chapter 4
Advanced Polynomial Methods

4.1 Introduction

The techniques presented in Chapter 3 provide a means of constructing a word level polynomial that represents the functionality performed by a block of combinational logic. Furthermore, a means of detecting branches, or control operations, within a block was presented. This allowed component matching to be performed efficiently for combinational circuits that contained few branch operations. However, many circuits implement synchronous elements and contain many branch operations. In this chapter, we present a mechanism for determining the polynomial representation of components that contain synchronous elements. In allowing synchronous elements, we introduce the possibility that a block may contain a feedback path, or loop. While such blocks do not always have polynomial representations, we derive methods for performing component matching using polynomials. Furthermore, for a subset of components that contains many control operations, we develop a technique for computing a polynomial that approximates the component's functionality. The error of this approximation is also determined.

4.2 Synchronous Acyclic Circuits

In Chapter 3 (specifically Theorem 3.2) it was established that polynomial representations, $y = F(x)$, exist for all combinational circuits. This was due to the fact that combinational circuits specify a finite number of input/output pairs ($\mathbf{x}$, $\mathbf{y}$) with corresponding integer values (x, y) that can be treated as coordinates to which a polynomial can be fit. Synchronous circuits pose an additional problem because circuit outputs are not only a function of the current inputs but also previous inputs. Thus, the polynomial representation of a synchronous circuit contains terms that are dependent on previous input values: $y = F(x, x@1, x@2, ..., x@p)$. The symbol $x@i$ indicates the value of x that is delayed by i cycles [De94].

4.2.1 Determining Combinational Equivalents

A polynomial representation for synchronous acyclic circuits can be computed by computing the polynomial representation for the equivalent combinational circuit with delayed input values. Consider a synchronous circuit represented by a synchronous logic network - i.e. a directed acyclic graph whose vertices represent combinational logic functions, whose edges represent function dependencies, and whose edge-weights represent synchronous delays introduced by registers. The sequential depth of the network, p, is the weight of the longest path. A synchronous logic network can be transformed into a combinational function of delayed input variables with delay less than or equal to p.

A synchronous logic network can be defined as the multi-graph {V, E, W}, where

(1) V is the set of vertices v_i that represent operations;

(2) E is the set of edges e_{ijk} that represent node connectivity, where e_{ijk} is an edge that connects nodes v_i and v_j (note that a third subscript k is necessary as there can be

multiple edges connecting two nodes);

(3) W is the set of weights w_{ijk} that indicate the number of synchronous elements between two nodes.

A synchronous logic network is transformed into an equivalent combinational network by duplicating all subgraphs in the synchronous network that end with a sink node with outgoing edges of different weight. The duplicated nodes are renamed to reflect the delay incurred on the associated outgoing edge (e.g. v_i becomes $v_i@w_{ijk}$). The outgoing edge in the duplicated subgraph is re-weighted to have zero delay (e.g. $e_{ijk} = 0$). The following algorithm details this transformation and is initialized by setting $v_{current}$ to be the input nodes of the synchronous logic network:

```
synch_to_comb(v_current, V, E, W) {
  w = minimum(w_currentjk);
  for each e_currentjk {
    if (w_currentjk != 0) {
      duplicate_predecessors(v_current, w_currentjk, V, E, W);
      /* Duplication requires creation of a new node
            v_current@w_currentjk, and its associated edges.
            Associated edge weights are zero. */
    }
  }
  if (w == 0) {
    remove_predecessors(v_current, V, E, W);
  }
```

```
    for each e_currentjk {
      synch to comb(v_j, V, E, W);
    }
  }
```

An example of the transformation of a synchronous network into its equivalent combinational network is shown in Figure 8.

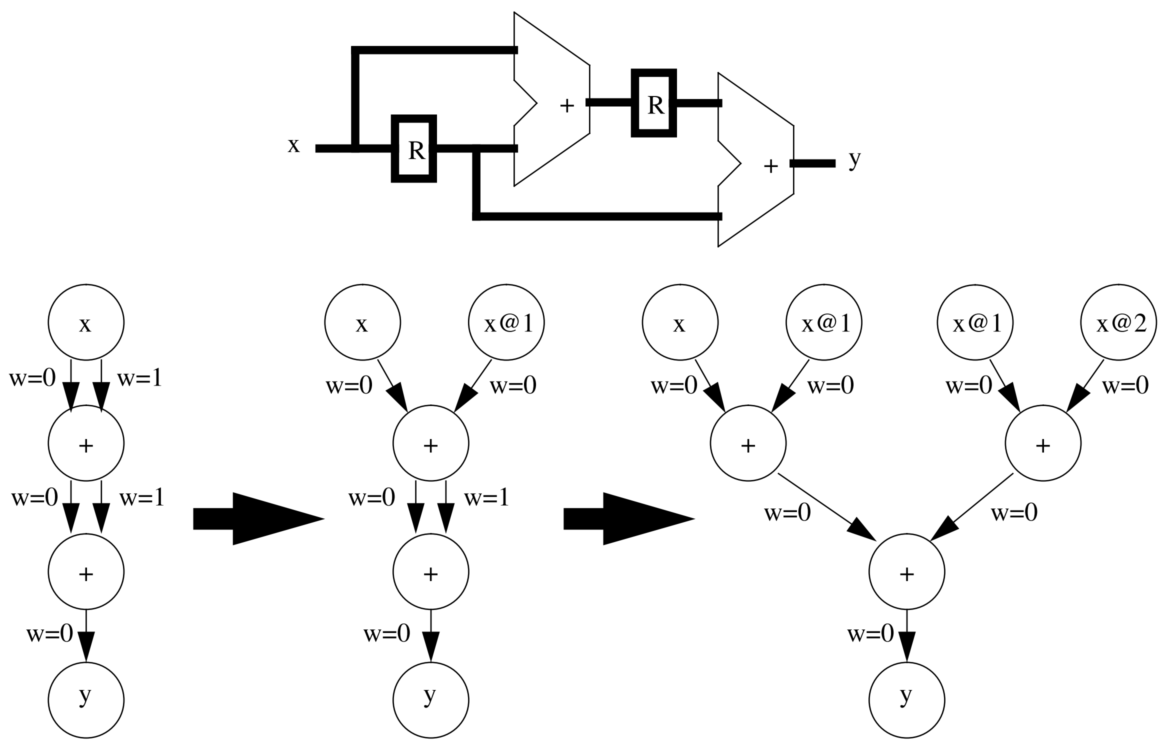


Fig. 8 Transformation of a sequential adder into a combinational circuit

Given an acyclic synchronous network with depth p, the equivalent combinational function is **F(x, x@1, x@2, ..., x@p)**. Note that p is finite due to the restriction that the circuit does not have feedback. A polynomial representation for **F(x)** can now be determined from **F(x, x@1, x@2, ..., x@p)**. The order of **F(x, x@1, x@2, ..., x@p)** is determined with respect to each **x@j,** for $0 \le j \le p$, as independent variables, and the coefficients of the polynomial representation are determined. In the example of Figure 8, this would result in the polynomial representation *F(x) = x + 2x@1 + x@2*.

4.3 Synchronous Cyclic Circuits

The method for determining polynomial representations for sequential acyclic circuits relied on the acyclic nature of the circuit to guarantee that a finite number of time-shifted inputs were required. However, by breaking the feedback path of a cyclic circuit **F(x)**, the previous techniques can be used to derive the order of the cyclic circuit. This is achieved by introducing an input $\mathbf{F_{feedback}}$, and determining the order of $\mathbf{F(x, F_{feedback})}$ with respect to **x** and $\mathbf{F_{feedback}}$.

A synchronous cyclic circuit can be modeled as a Mealy/Moore finite state machine (FSM) that may or may not have an initial state. For example, a rasterizer is a synchronous cyclic circuit with an initial state and an Infinite Impulse Response filter is a synchronous cyclic circuit with no initial state. For the sake of this analysis, we consider three different topologies of synchronous cyclic circuits: (1) an FSM with no initial state, (2) an FSM with an initial state that does not reach a steady state, and (3) an FSM with an initial state that reaches a steady state after a finite number of cycles. In the terminology of Section 1.1.2 each of these cases can be seen as a loop of guarded dataflows. As shown in Figure 9, we can represent each of these topologies as a function **F(x)** that may have up to three branches: a branch corresponding to an initialization state ($\mathbf{f_1(x)}$), a branch corresponding to the transient states ($\mathbf{f_2(x, F_{feedback})}$), and a branch corresponding to a steady state (labeled **=**).

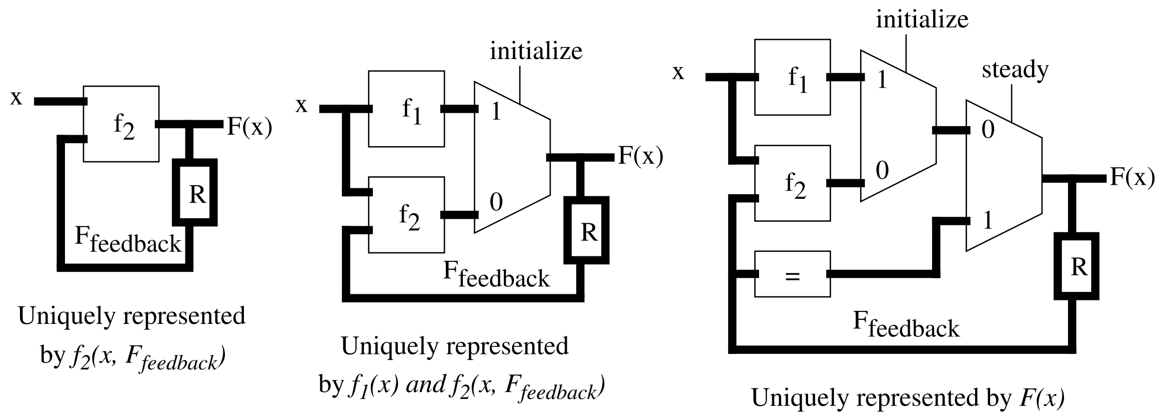
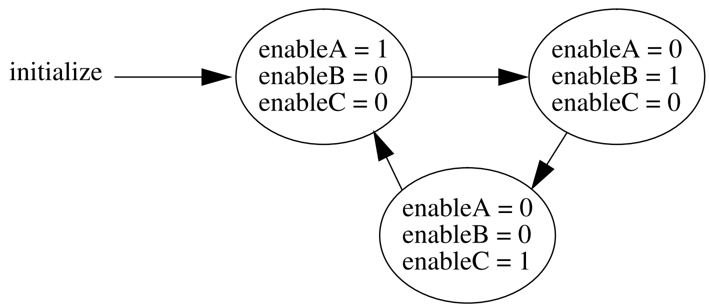


Fig. 9 Synchronous cyclic circuit models: (1) with only a transient feedback branch,
(2) with a transient and an initialization path
(3) with a transient, initialization, and steady state branch.

The initialization branch is a dataflow guarded by the signal *initialize*, the transient branch is a dataflow guarded by the expression $\overline{initialize + steady}$, and the steady-state branch is a dataflow guarded by the signal *steady.*

Using the techniques described previously, we can compute a polynomial representation for each branch. An initialization branch has a polynomial representation that contains no terms with the variable $F_{feedback}$. A steady-state branch has the polynomial representation $F(x, F_{feedback}) = F_{feedback}$. If the function contains no initialization branch or no steady state branch (topology (1) or (2)), then no polynomial representation $F(x)$ exists. However, the circuit is uniquely represented by the polynomial $F(x, F_{feedback})$. In the case of topology (1), $F(x, F_{feedback})$ is simply $f_2(x, F_{feedback})$. In the case of topology (2), $F(x, F_{feedback})$ is comprised of two domains (corresponding to $initialize = 1$ and $initialize = 0$ in Figure 9), and is $f_1(x)$ within the first domain and $f_2(x, F_{feedback})$ within the second domain. Example 4.2.1 illustrates computation of a polynomial representation for FSM with topology (2).

Example 4.3.1 Consider the finite state machine with a one bit input (initialize) and a three bit output **F**(initialize) = {enableA, enableB, enableC} that provides round-robin access to memory for three clients:



Breaking the feedback loops yields the function **F**(initialize, $\mathbf{F_{feedback}}$). Performing order computation results in the detection of four branches, each of which is order zero (i.e constant). For example, in the branch that is executed under the condition initialize = 1, the output **F**(initialize, $\mathbf{F_{feedback}}$) = {enableA, enableB, enableC} = 100. Thus the polynomial representation for this branch is $F(initialize,\ F_{feedback}) = 4$. Coefficient computation for each branch yields the following order zero polynomial representations for $F(initialize,\ F_{feedback})$:

Domain	Polynomial
$initialize = 1$	$F(initialize,\ F_{feedback}) = 4$
$initialize = 0$ AND $3 < F_{feedback}$	$F(initialize,\ F_{feedback}) = 2$
$initialize = 0$ AND $1 < F_{feedback} < 4$	$F(initialize,\ F_{feedback}) = 1$
$initialize = 0$ AND $F_{feedback} < 2$	$F(initialize,\ F_{feedback}) = 4$

An initialization branch exists, but no steady-state branch exists, thus $F(initialize,\ F_{feedback})$ uniquely represents the finite state machine (although other finite state machines exist that perform the same operation with different state encodings).

The remainder of this analysis focuses on circuits for which $F(x,\ F_{feedback})$ is not a unique representation, i.e. those circuits that contain both an initialization state and steady state (topology (3)).

4.3.1 Order Computation with Feedback

Assume function **y = F(x)**: $B^m \rightarrow B^k$ implements three branches, one initialization branch ($\mathbf{f_1(x)}$), one steady state branch, and one transient feedback branch ($\mathbf{f_2(x, F_{feedback})}$). We assume that a signal controls the number of iterations through the transient feedback path. We can then evaluate the circuit based on the number of iterations of the transient feedback branch.

The order of $\mathbf{f_1(x)}$ with respect to **x**, referred to as n_{x1}, can be determined by computing the equivalent combinational circuit and using the techniques presented in Chapter 3. As a result, the polynomial representation of this branch, $f_1(x)$, can be determined. Furthermore, if $\mathbf{y = F_{feedback}(x)}$ is treated as an input to $\mathbf{f_2(x, y)}$, then the order of $\mathbf{f_2(x, y)}$ with respect to **x**, referred to as n_{x2}, and with respect to **y**, referred to as n_y, can also be determined. As a result, the polynomial representation of this branch, $f_2(x, y)$, can be determined. After initialization, the order of **F(x)** is n_{x1}, and after the first iteration of the non-steady-state feedback branch, the order of **F(x)** is less than ($n_y n_{x1} + n_{x2}$) and greater than $n_y n_{x1}$. In general, if the order of **F(x)** is n_t after t iterations, then the order of **F(x)**, after one more iteration of the non-steady-state feedback branch, is less than $n_y n_t + n_2$ and greater than $n_y n_t$. Thus, the upper bound on the order of **F(x)** after t iterations is:

$$n_y^t \cdot n_{x1} + \sum_{i=0}^{t-1} n_y^i \cdot n_{x2}$$

To determine the order of **F(x)** there are three cases that need to be considered:

(1) t is known,

(2) t is not known, $n_y = 1$, and there is no xy term in $f_2(x,y)$,

(3) t is not known, and ($n_y \neq 1$ or there is an xy term in $f_2(x,y)$).

In *case (1)*, the order of **F(x)** can be bounded according to the equation above. In *case (2)*, since there is no xy term in $f_2(x,y)$, the order of **F(x)** does not increase on successive iterations, and is simply the greater of n_{x1} and n_{x2}. For both of these cases, since the order of **F(x)** is bounded, a polynomial representation exists for **F(x)**. If the upper bound on the order is n_u, this representation can be determined by extracting n_u+1 points from the circuit to create the system of linear equations that determine the polynomial coefficients. In *case (3)*, the order of **F(x)** is dependent on t and is therefore unbounded and has no polynomial representation. However, like the cyclic circuits with no initialization or steady state branch, the polynomial representation $F(x, F_{feedback})$ uniquely specifies the functionality of the circuit, and can be used to perform matching as shown in Section 4.5.3.

Example 4.2.2 Consider a Boolean circuit **F(x, y)** with inputs **x**, **y**, output **z**, that performs multiplication through iterative addition by executing the following initialization branch and feedback branches:

```
initial begin        always @ (z or x or d) begin
    z = x;                   if (d) z = z + x;
    d = y;                   if (d) d = d - 1;
end                     end
```

Breaking the feedback loops introduces variables $\mathbf{z_{feeback}}$ and $\mathbf{d_{feedback}}$ and results in computation of the following set of polynomials:

initialize = 1	initialize = 0 and d≠ 0	initialize = 0 and d = 0
$z = x$	$z = z_{feedback} + x$	$z = z_{feedback}$
$d = y$	$d = d_{feedback} - 1$	$d = d_{feedback}$

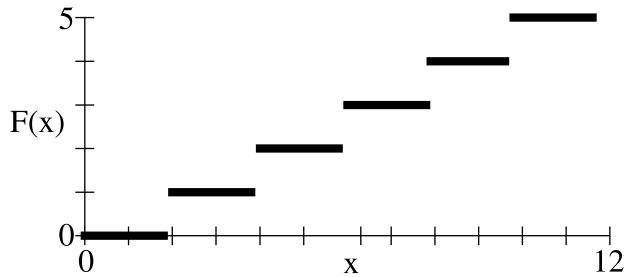
Since the feedback polynomial $z = z_{feedback} + x$ is of order one with respect to $z_{feedback}$ and contains no $xz_{feedback}$ term, *case (2)* is satisfied, and the order of **F(x, y)** with respect to **x** is the greater of n_{x1} and n_{x2}, both of which are 1. Since the feedback polynomial $d = d_{feedback} - 1$ is of order one with respect to $d_{feedback}$ and contains no $yd_{feedback}$ term, *case (2)* is also satisfied for this polynomial, and the order of **F(x, y)** with respect to **y** is the greater of n_{y1} and n_{y2}, which are one and zero respectively. Thus **F(x, y)** is of order one with respect to both inputs (i.e. $n_x = 1$ and $n_y = 1$) , requiring $(n_x+1)(n_y+1) = 4$ points to be extracted from the circuit. The points $(x, y, z) = \{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}$ can be extracted, yielding the following system of equations:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \bullet \begin{bmatrix} c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The solution to the system of equations yields the polynomial representation $F(x, y) = xy$.

4.4 Approximations

Polynomial representations are an efficient way to encapsulate the functionality of arithmetic circuits. Furthermore, circuits that implement non-arithmetic operations can be modeled efficiently by determining subdomains over which the circuit implements functionality that has a low-order polynomial representation, as shown in Section 3.3. However, this representation becomes very complex when the number of subdomains is large. For example, as illustrated in Figure 10, circuits that approximate arithmetic functions frequently generate many subdomains.



Domain	Polynomial
[0, 1]	F(x) = 0
[2, 3]	F(x) = 1
[4, 5]	F(x) = 2
[6, 7]	F(x) = 3
[8, 9]	F(x) = 4
...	...

Fig. 10 Subdomains generated by the function F(x) = x >> 1.

Example 4.4.1 Consider a circuit that implements **F(x) = (x** >> 1), where **x** is an *m* bit word, requires 2^{m-1} subdomains (Figure 10) in its polynomial representation. Rather than represent **F(x)** as a list of subdomains of **x** and corresponding polynomials *F(x)* that describe **F(x)** exactly over those subdomains, it is much more efficient to represent **F(x)** as the polynomial *x/2* and specify the maximum error between the continuous function *x/2* and the exact polynomial representation *F(x)*.

Given a Boolean function **y = F(x)**: $B^{m} \to B^{k}$, with corresponding integer values *(x, y)*, an approximate polynomial representation $y_{approx} = F_{approx}(x)$ can be determined. The approximate polynomial representation is determined such that $\left|F(x) - F_{approx}(x)\right| \le \Delta$ for all *x*, where Δ is a given accuracy. Approximation allows a low-order polynomial representation to be generated for a Boolean function that would otherwise have a polynomial representation of high order. Sections 4.4.1 through 4.4.4 derive in detail the approximate polynomial representation $F_{approx}(x)$ and the tolerance Δ within which the approximation is accurate.

4.4.1 Computing Approximations

As proven in Theorem 3.1, the order of a function is reduced by one by computing the difference **F(x+1) - F(x)**. The algorithms to this point have relied on the resulting fact that, if the order of **F(x)** is *n*, then recursively performing this difference *n+1* times will reduce the function to zero. Now we relax the requirement that **F(x+1) - F(x)** be exactly zero. If performing this difference *n* times results in a function that is not zero, but is numerically close to zero, then the polynomial representation *F(x)* of **F(x)** can be approximated well by a polynomial of degree *n*.

To translate this to approximating a Boolean function **F(x)** with a polynomial, again consider the function **y = F(x)**: $B^{\mathrm{m}} \rightarrow B^{\mathrm{k}}$. If the most significant *q* bits of **y** are 1, then for the two's complement integer encoding of $y \in Z$, y = Encode(**y**), the inequality $-2^{\mathrm{k-q}} < y$ holds. Similarly, if the most significant *q* bits of **y -** Decode($2^{\mathrm{k-q}}$) (performed using two complement arithmetic) are 1, then the inequality $y < 2^{\mathrm{k-q}}$ holds. As, a result, if $\mathbf{F}^{-}$ is defined to be **F(x+1) - F(x)** and $\mathbf{F}^{+}$ is defined to be **F(x+1) - F(x) -** Decode($2^{\mathrm{k-q}}$), then the following statement holds: if the upper *k* bits of the bitwise or of $\mathbf{F}^{-}$ and $\mathbf{F}^{+}$ are 1, then $-2^{k-q} < F(x+1) - F(x) < 2^{k-q}$. The bound on $F(x+1) - F(x)$, allows us to derive an approximation of $F(x)$:

Let **F(x+1) - F(x) = G(x)**

Given **F(0)**

=> **F(1) = G(0) + F(0)**

=> **F(2) = G(1) + F(1) = G(1) + G(0) + F(0)**

...

$$\Rightarrow \mathbf{F}(\mathbf{x}) = \left(\sum_{i=0}^{x-1} \mathbf{G}(\mathbf{i})\right) + \mathbf{F}(\mathbf{0})$$

If Encode($\mathbf{G(i)}$) is small (e.g. -2^{k-q} < Encode($\mathbf{G(i)}$) < 2^{k-q}, for suitable q), the polynomial representation of $\left(\sum_{i=0}^{x-1} \mathbf{G}(\mathbf{i})\right)$ is well approximated by the line $x \cdot \left(\text{Encode}\mathbf{F}(11...11) - \mathbf{F}(\mathbf{0})/2^m\right)$.

F(x) can then be approximated by:

$$F_{approx}(x) \approx x \cdot \left(\text{Encode}(\mathbf{F}(11...11) - \mathbf{F}(\mathbf{0}))/2^m\right) + \text{Encode}(\mathbf{F}(\mathbf{0}))$$

Example 4.4.2 Consider the eight-bit function $\mathbf{y = F(x)}$ where $\mathbf{x} \in B^8$ and $\mathbf{y} \in B^8$:

$y_0 = x_1$;	$y_4 = x_5$;
$y_1 = x_2$;	$y_5 = x_6$;
$y_2 = x_3$;	$y_6 = x_7$;
$y_3 = x_4$;	$y_7 = 0$;

This circuit could be partitioned into 64 subdomains and represented exactly with 64 order-0 polynomials (similar to Figure 10). However, the first difference iteration reveals that the upper 7 bits of **F(x+1) + F'(x)** are 1, yielding the bound $-1 < F(x+1) - F(x) < 1$. Therefore, *F(x)* can be approximated by the first order polynomial $F_{approx}(x)$ = x(Encode($\mathbf{F}$(11...11) - $\mathbf{F(0)}$))/2^8 = .498x.

4.4.2 Computing Error for the Linear Approximation

In this section, we will compute a bound on the accuracy Δ of a linear approximation to the polynomial representation *F(x)*. The difference between *F(x)* and $F_{approx}(x)$, termed $\Delta(x)$, is:

$$\Delta(x) = \sum_{i=0}^{x-1}\left(\mathrm{Encode}\,(\mathbf{G}\,(\mathbf{i})\,) - \mathrm{Encode}\,(\mathbf{F}(11...11) - \mathbf{F}(\mathbf{0}))\,/\,2^{m}\right)$$

Since $-2^{k-q} < \mathrm{Encode}(\mathbf{G(i)}) < 2^{k-q}$, it requires only k-q bits to represent $\mathbf{G(i)}$. Assuming $k \approx q$ for a good approximation, computation of $(\mathrm{Encode}(\mathbf{G(i)}) - \mathrm{Encode}(\mathbf{F}(11...11) - \mathbf{F(0)})/2^m)$ need only be performed only over a short word length (k-q bits). Since the Encode operation is distributive (i.e. Encode(A) + Encode(B) = Encode(A + B)), the following equivalence holds:

$$2^m(\mathrm{Encode}(\mathbf{G(i)}) - \mathrm{Encode}(\mathbf{F}(11...11) - \mathbf{F(0)})/2^m)$$

$$= \mathrm{Encode}((\mathbf{G(i)} << m) - (\mathbf{F}(11...11) - \mathbf{F(0)}))$$

Defining $\delta(\mathbf{i}) = (\mathbf{G(i)} << m) - (\mathbf{F}(11...11) - \mathbf{F(0)})$ yields:

$$\Delta(\mathrm{x}) \cdot 2^{m} = \sum_{i=0}^{x-1} \mathrm{Encode}\,(\delta(\mathbf{i}))$$

Replacing $\delta(\mathbf{i})$ by the sum of its bits $\delta_j(\mathbf{i})$ (i.e. $\mathrm{Encode}\delta(\mathbf{i}) = \sum_{j=0}^{m+k-q-1} 2^{j} \cdot \delta_j(\mathbf{i})$) yields:

$$\Delta\,(x) = \sum_{i=0}^{x-1}\left(\sum_{j=0}^{m+k-q-1} 2^{j-m} \cdot (\delta_j(\mathbf{i}))\right)$$

An upper bound on $\Delta(x)$ can then be determined from each bit $\delta^{+}{}_{j}(\mathbf{i})$ of the positive values of $\delta(\mathbf{i})$:

$$\Delta(x) < \sum_{i=0}^{x-1}\left(\sum_{j=0}^{m+k-q-1} 2^{j-m} \cdot \left(\delta^{+}{}_{j}(\mathbf{i}) \neq 0\right)\right)$$

Similarly, a lower bound can be determined from each bit $\delta^{-}_{j}(\mathbf{i})$ of the negative values of $\delta(\mathbf{i})$:

$$\Delta(x) > \sum_{i=0}^{x-1}\left(\sum_{j=0}^{m+k-q-1} 2^{j-m}\cdot\left(\delta^{-}_{j}(\mathbf{i}) \neq 0\right)\right)$$

Following two's complement arithmetic, if the most significant bit of $\delta(\mathbf{i})$ is zero, then $\delta(\mathbf{i})$ is positive and, if the most significant bit of $\delta(\mathbf{i})$ is one, then $\delta(\mathbf{i})$ is negative. Since the most significant bit of $\delta(\mathbf{i})$ is $\delta_{m+k-q-1}(\mathbf{i})$, the bits $\delta^{+}_{j}(\mathbf{i})$ and $\delta^{-}_{j}(\mathbf{i})$ can be determined by computing the positive and negative cofactor of $\delta_{j}(\mathbf{i})$ with respect to $\delta_{m+k-q-1}(\mathbf{i})$.

Computing $\Delta(x)$ for all 2^{m} values of x is prohibitively complex due to the size of the domain and the fact that $\Delta(x)$ is a summation of x values. However the equivalence holds:

$$\Delta(x) = \Delta(0) + \bar{x}_o \cdot (\Delta(x+1) - \Delta(x)) + \bar{x}_o \cdot \bar{x}_1 \cdot (\Delta(x+2) - \Delta(x)) + \ldots$$

$$+ \bar{x}_o \cdot \bar{x}_1 \cdot \ldots \cdot \bar{x}_{m-1} \cdot \left(\Delta(x + 2^{m-1}) - \Delta(x)\right)$$

$$\text{Thus, } \Delta(x) = \sum_{i=0}^{m}\left(\prod_{j=0}^{i} \bar{x}_i\right)\cdot\left(\Delta(x + 2^{i}) - \Delta(x)\right)$$

Therefore, to circumvent this summation and determine a bound on $\Delta(x)$, the maximum values for the following are determined:

$\Delta(x+1)$ - $\Delta(x)$ where bit x_0 of **x** is 0

$\Delta(x+2)$ - $\Delta(x)$ where bits x_0, x_1 of **x** are 0

...

$$\Delta(x+2^{m-1}) - \Delta(x) \text{ where bits } x_0, x_1, ..., x_{m-1} \text{ of } \mathbf{x} \text{ are } 0$$

The maximum value of the computation $\Delta(x+2^j) - \Delta(x)$, where bits x_0, x_1, ..., x_j of **x** are 0, yields the maximum error contributed by bit *j* of the input. The sum of the maximum values of each of the above equations provides the maximum error contributed by all bits, which is a bound on the error of the approximation. Thus, a bound on the accuracy of the linear approximation is:

$$\Delta < [\Delta(x+1) - \Delta(x)] + [\Delta(x+2) - \Delta(x)] + ... + [\Delta(x+2^{m-1}) - \Delta(x)]$$

As shown in Example 4.4.3, values of $\Delta(x)$ are be reached by summing a subset of the above equations.

Example 4.4.3 $\Delta(0) = 0$ $\quad$ $\Delta(1) = [\Delta(0+1) - \Delta(0)]$

$\Delta(2) = [\Delta(0+2) - \Delta(0)]$ $\quad$ $\Delta(3) = [\Delta(1+2) - \Delta(1)] + [\Delta(0+1) - \Delta(0)]$

$\Delta(7) = [\Delta(6+1) - \Delta(6)] + [\Delta(4+2) - \Delta(4)] + [\Delta(0+4) - \Delta(0)]$.

Example 4.4.4 For the approximation computed in Example 4.3.2, the resulting $\boldsymbol{\delta}(\mathbf{i})$ is:

$\delta_0 = 1$;	$\delta_5 = 0$;
$\delta_1 = 0$;	$\delta_6 = 0$;
$\delta_2 = 0$;	$\delta_7 = 1$;
$\delta_3 = 0$;	$\delta_8 = i_0'$;
$\delta_4 = 0$;	$\delta_9 = i_0'$;

The error contributed by $\Delta(x+1)$-$\Delta(x)$ when $x_0 = 0$ is Encode($\boldsymbol{\delta}(\mathbf{x})$)/$2^8$. This is always negative because the most significant bit of $\boldsymbol{\delta}(\mathbf{x})$ is 1 when $x_0 = 0$: Encode($\boldsymbol{\delta}(\mathbf{x})$)/$2^8$ = -127/ 2^8 = -.5 units. The error contributed by $\Delta(x+2)$-$\Delta(x)$, when $x_1x_0 = 00$, is Encode($\boldsymbol{\delta}(\mathbf{x+1})$ + $\boldsymbol{\delta}(\mathbf{x})$). This is always positive because the most significant bit of $\boldsymbol{\delta}(\mathbf{x+1}) + \boldsymbol{\delta}(\mathbf{x})$ is 0 when $x_0 = 0$: Encode($\boldsymbol{\delta}(\mathbf{x+1}) + \boldsymbol{\delta}(\mathbf{x})$) = (129 - 127)/$2^8$ = .008 units. Similarly, other differences $\Delta(x+2^i)$-$\Delta(x)$ contribute only positive error. Other differences $\Delta(x+2^i)$-$\Delta(x)$ contribute a total of .5 units of positive error, resulting in the error bound: $-.5 < \Delta < .5$. Thus, the circuit implements the polynomial $F(x) = .498x$ within .5 units. This approximate representation is far less complex than the 64 polynomials that would be required to represent the circuit exactly.

4.4.3 Non-Linear Approximations

A function **F(x)** may implement a non-linear operation (e.g. $\mathbf{F(x) = (x^2} >> 1))$ that is well approximated by a non-linear polynomial representation (e.g. $F(x) = x^2/2$). In this case, the first iteration of **F(x+1) - F(x)** may not satisfy the condition $-2^{k-q} < F(x+1) - F(x) < 2^{k-q}$. If a suitable bound is found for the n^{th} iteration of **F(x+1) - F(x)**, termed $\mathbf{G_n(x)}$, instead of the first iteration, then a non-linear approximation for $F(x)$ can be computed from Newton's forward difference interpolating formula ([KhTo86]):

$$F_{approx}(x) = \text{Encode}(\mathbf{F}(\mathbf{0})) + \binom{x}{1} \cdot \frac{\text{Encode}(\text{F}(11...11) - \mathbf{G}_{\text{approx, }0}(\mathbf{0}))}{2^m - 1} + \ldots$$

$$+ \binom{x}{n-1} \cdot \frac{\text{Encode}(\mathbf{G}_{\mathbf{n}-2}(11...11) - \mathbf{G}_{\text{approx, }\mathbf{n}-1}(\mathbf{0}))}{2^m - (\text{n}-1)} +$$

$$+ \binom{x}{n} \frac{\text{Encode}(\mathbf{G}_{\mathbf{n}-1}(11...11) - \mathbf{G}_{\text{approx, }\mathbf{n}}(\mathbf{0}))}{2^m - \text{n}}$$

The following paragraphs derive this formula and the algorithm for computing $G_{approx,i}$.

Performing the difference F(x+1) - F(x) n times, over the range of all possible input values, would yield the following values:

$$\begin{array}{ccccccccccc}
F(0) & & F(1) & & F(2) & & F(3) & & \ldots & & F(11..11) \\
& G_0(0) & & G_0(1) & & G_0(2) & & \ldots & & G_0(11..11) & \\
& & G_1(0) & & G_1(1) & & \ldots & & G_1(11..11) & & \\
& & & & & \ldots & & & & & \\
& & & & G_n(0) & \ldots & G_n(11..11) & & & &
\end{array}$$

Note that $F(0), G_0(0), \ldots, G_n(0), G_n(1), \ldots, G_n(11.11)$ uniquely specify a function. We approximate the last row, as in the linear case, with the equation $y = G_n(0) + x(G_{n-1}(11..11)-G_n(0))/2^m$. Replacing the values $G_i(0)$ with the values $G_{approx,i}(0)$ would yield the following values ($M = (G_{n-1}(11..11)-G_n(0))/2^m$):

$$\begin{array}{cccccc}
F_{approx}(0) & F_{approx}(1) & F_{approx}(2) & F_{approx}(3) & \ldots & F_{approx}(11..11) \\
& G_{approx,0}(0) & G_{approx,0}(1) & G_{approx,0}(2) & \ldots & G_{approx,0}(11..11) \\
& & G_{approx,1}(0) & G_{approx,1}(1) & \ldots & G_{approx,1}(11..11) \\
& & & \ldots & & \\
& & G_{approx,n-1}(0) & \ldots & & G_{approx,n-1}(11..11) \\
& & M & M & \ldots & M
\end{array}$$

The difference between the expression represented by this set of forward differences and those of the original function is:

$$[F(0) + \Sigma(G_0(0) + \Sigma(G_1(0) + \ldots + \Sigma(G_n(0)))] -$$

$$[F_{approx}(0) + \Sigma(G_{approx,0}(0) + \Sigma(G_{approx,1}(0) + \ldots + \Sigma M)]$$

(limits not shown for readability)

This difference can be minimized by the assignments:

$$G_{approx,n}(0) = (G_{n-1}(11..11) - G_n(0) - \Sigma M)/(2^m-n)$$

$$G_{approx,n-1}(0) = (G_{n-2}(11..11) - G_{n-1}(0) - \Sigma G_{approx,n-1}(i))/(2^m-n+1)$$

$$G_{approx,n-2}(0) = (G_{n-3}(11..11) - G_{n-2}(0) - \Sigma G_{approx,n-2}(i))/(2^m-n+2)$$

$$\ldots$$

$$G_{approx,0}(0) = (F(11..11) - G_0(0) - \Sigma G_{approx,0}(i))/2^m$$

Note that $\Sigma G_{approx,j}(i)$ can be computed explicitly and compactly:

$$\Sigma G_{approx,j}(i) = \Sigma G_{approx,j}(0) + \Sigma\Sigma G_{approx,j+1}(i)$$

$$= \Sigma G_{approx,j}(0) + \Sigma\Sigma G_{approx,j+1}(0) + \Sigma\Sigma\Sigma G_{approx,j+2}(i)$$

$$...$$

$$= \Sigma G_{approx,j}(0) + \Sigma\Sigma G_{approx,j+1}(0) + \Sigma\Sigma\Sigma G_{approx,j+2}(0) + ... + \Sigma\Sigma...\Sigma M$$

$$= \binom{x}{1} \cdot G_{approx,\,j}(0) + \binom{x}{2} \cdot G_{approx,\,j+1}(0) + ... + \binom{x}{2} \cdot M$$

$$= \sum_{i=0}^{n} \binom{x}{i} \cdot G_{approx,\,j+i}(0)$$

Example 4.4.5 Consider the circuit that implements (**F(x) = (x²**>>1)) where x is an 8 bit input. After two difference iterations, $G_2(x) <= 2$. By computing $(G_1(11..11)-G_2(0))/2^m$ we obtain the assignment M = 254/256. We then obtain the values $G_{approx,0}(0) = (F(11..11) - G_0(0) - \Sigma M)/2^m-1$ = 32512-0-32385 = 127/255. Thus, the approximation of the circuit is:

$$F_{approx}(x) = 0 + \binom{x}{1} \cdot \frac{127}{255} + \binom{x}{2} \cdot \frac{254}{256}$$

$$= .496x^2 + .002x$$

4.4.4 Computing Error for the Non-Linear Approximation

In Section 4.4.2, for the linear case, the error was determined by computing an upper bound for the equation:

$$\Delta(x) \cdot 2^m = \sum_{i=0}^{x-1} \mathrm{Encode}(\delta(\mathbf{i}))$$

If we expand this to the general, nonlinear case, and define $G_{approx,j}(x) - G_j(x) = \delta_j(x)$ and $F_{approx}(x) - F(x) = \Delta(x)$, the forward difference representation of the total error is:

$$\begin{array}{ccccccccc}
\Delta(0) & & \Delta(1) & & \Delta(2) & & \Delta(3) & ... & \Delta(11..11) \\
& \delta_0(0) & & \delta_0(1) & & \delta_0(2) & ... & \delta_0(11..11) & \\
& & \delta_1(0) & & \delta_1(1) & ... & \delta_1(11..11) & & \\
& & & & ... & & & & \\
& & & & \delta_{n-1}(0)/2^m \; ...\delta_{n-1}(11..11)/2^m & & & &
\end{array}$$

Thus, given $\Delta(0) = 0$, the worst case error is the maximum value of the expression:

$$\Delta(x) = \Sigma\ \delta_0(0) + \Sigma\Sigma\ \delta_1(0) + \Sigma\Sigma\Sigma\ \delta_2(0) + \ldots + \Sigma\Sigma\ldots\Sigma\ \mathrm{Encode}(\delta_n(x))/2^m$$

(limits not shown for readability)

Similar to the linear case, $\Sigma\Sigma\ldots\Sigma\mathrm{Encode}(\delta_n(x))/2^m$ can not be computed directly, as it requires an exponential number of addition operations. In the linear case, this was handled by using the following equivalence:

$$\Delta(x) = \sum_{i=0}^{m}\left(\prod_{j=0}^{i}\bar{x}_i\right)\cdot\left(\Delta(x+2^i) - \Delta(x)\right)$$

$\Delta(x) = \Sigma\mathrm{Encode}(\delta(x))$ was computed by performing a number of Boolean additions that is equivalent to the input length (i.e. of linear complexity with respect to input length). For the case n=2, $\Delta(x) = \Sigma\ \delta_0(0) + \Sigma\Sigma\mathrm{Encode}(\delta_1(x))/2^m$, can similarly be computed by first explicitly computing $\Delta_1(x) = \Sigma\mathrm{Encode}(\delta_n(x))/2^m$ from the above equation, then computing $\Delta(x) = \Sigma\ (\delta_0(0) + \Delta_1(x))$ as in the linear case. For the general case, the approximation error can be determined by the following iterative computations:

(1) Compute $\Delta_1(x) = \Sigma\mathrm{Encode}(\delta_n(x))/2^m$

(2) Compute $\Delta_2(x) = \Sigma\ (\delta_{n-1}(0) + \Delta_1(x))$

(3) Compute $\Delta_3(x) = \Sigma\ (\delta_{n-2}(0) + \Delta_2(x))$

...

(n-2) Compute $\Delta_{n-2}(x) = \Sigma\ (\delta_1(0) + \Delta_{n-3}(x))$

(n-1) Compute $\Delta(x) = \Sigma\ (\delta_0(0) + \Delta_{n-1}(x))$

Note that the number of addition operations performed is linearly proportional to the order of the approximation multiplied by the number of input bits (i.e. $m*n$). Example 4.3.6 illustrates this computation.

Example 4.4.6 Consider the circuit of Example 4.4.5 that implements ($\mathbf{F(x)} = (\mathbf{x^2}>>1)$) where x is an 8 bit input. The circuit is approximated by the order two polynomial $F(x) = .496x^2 + .002x$. The resulting $\delta_n(\mathbf{i})$ (n=2) is:

$\delta_{n,0} = 0;$ $\delta_{n,5} = 0;$

$\delta_{n,1} = 0;$ $\delta_{n,6} = 0;$

$\delta_{n,2} = 0;$ $\delta_{n,7} = 0;$

$\delta_{n,3} = 0;$ $\delta_{n,8} = 1;$

$\delta_{n,4} = 0;$ $\delta_{n,9} = i_0';$

$\delta_{n,10} = i_0';$

Computing $\Delta_1(x) = \Sigma\text{Encode}(\delta_n(i))$ by performing m additions yields:

$\Delta_{1,0} = 0;$ $\Delta_{1,5} = 0;$

$\Delta_{1,1} = 0;$ $\Delta_{1,6} = 0;$

$\Delta_{1,2} = 0;$ $\Delta_{1,7} = 1;$

$\Delta_{1,3} = 0;$ $\Delta_{1,8} = i_0';$

$\Delta_{1,4} = 0;$ $\Delta_{1,9} = i_0';$

$\Delta_{1,10} = i_0';$

Computation of an upper bound on $\Delta(x) = \Sigma\,(\delta_0(0) + \Delta_1(i))$ is then performed as in the linear case. Each of $\Delta(x+2^i)$-$\Delta(x)$, where i = 0, 1, ..., m-1 is computed. For each of the 8 resulting differences, the cofactor with respect to the high order bit yields an expression for maximum negative error. Summing up these values yields a maximum negative error of -0.5 units. Computing the cofactor with respect to the complement high order bit yields an expression for maximum positive error. Summing up these values yields a maximum positive error of 0.5 units. Thus, the approximation $F(x) = .496x^2 + .002x$ models the circuit accurately with error bounds $-0.5 < \Delta < 0.5$.

4.5 Matching

Consider a circuit specification S(x) that defines the functionality of a circuit. Given a library of existing components, where each component is described by a Boolean function **F(x)**, polynomial representations provide a means for quantifying the difference between the specification S(x) and a potential implementation **F(x)**. This can be achieved by computing the polynomial $\varepsilon(x) = S(x) - F(x) + \Delta$, where *F(x)* is the polynomial representation of **F(x)** within an accuracy of Δ, and using traditional numerical methods to find the maximum value of $\varepsilon(x)$. In quantifying the maximum error ε of an implementation and guaranteeing that ε is within a given tolerance, system traits such as performance, power, and area can be optimized by selecting faster or smaller designs that implement less accurate arithmetic.

Example 4.5.1 Consider the specification for an eight bit 3x3 sharpening filter used for processing grayscale images:

S(x[0, 0], x[0, 1], x[0, 2], x[1, 0], x[1, 1], x[1, 2], x[2, 0], x[2, 1], x[2, 2]) =

(- x[0, 0] - x[0, 1] - x[0, 2]

- x[1, 0] + 8x[1, 1] - x[1, 2]

- x[2, 0] - x[2, 1] - x[2, 2])/9;

Consider an implementation **F(x)** with the following approximate polynomial representation:

F(x, x@1, x@2, x@3, x@4, x@5, x@6, x@7, x@8) ≈

(- x - x@1 - x@2

- x@3 + 8x@4 - x@5

- x@6 - x@7 - x@8)/8;

Δ = .875;

For $0 < x < 2^8$, $\varepsilon(x) < 29$ grayscale units. This implementation yields a sharpening filter that yields an image that is of similar quality to that specified, but likely smaller and faster than an exact implementation, since division by 8 can be performed much more efficiently than division by 9.

4.5.1 Transcendental Specifications

A means of approximating a specification for transcendental functions can be derived from the results of Taylor series approximation. Given a specification S(x), with Taylor series $S_{approx}(x) = 1 + (dS(0)/dx)x/1! + (d^2S(0)/dx^2)x^2/2! + ... + (d^nS(0)/dx^n)x^n/n!$, the difference between $S_{approx}(x)$ and S(x) is $\varepsilon(x) = (d^{n+1}F(c)/dx^{n+1})x^{n+1}/(n+1)!$ where $0 < c < x$. Thus, if the error in a Taylor series approximation to a function can be bounded, then the difference between an implementation that matches that approximation and the specification can be bounded.

Example 4.5.2 An implementation that is determined to be of order 4 and yields the polynomial representation $F(x) = 1 - x^2/4 + x^4/24$ matches the cosine function used in DCT with an error $\varepsilon < .0083$ over the interval [0, 1].

4.5.2 Composition

The ease with which polynomials can be composed, using traditional algebraic manipulations, can allow seemingly inappropriate implementations to be combined to fulfill a specification.

Example 4.5.3 The Boolean function **F(x)** with polynomial representation $F(x) = x^2$ may appear to be a completely inappropriate match for the polynomial specification of cos(x) derived in Example 4.5.2. However, if an adder $\mathbf{F_{sum}(x, y)}$ ($F_{sum}(x) = x + y$), negation element $\mathbf{F_{neg}(x)}$ ($F_{neg}(x) = -x$), and shifter $\mathbf{F_{shift}(x, y)}$ ($F_{shift}(x,y) = x/2^y$) exist in the implementation library, **F(x)** can be allocated and composed with the adder to approximate the cos(x):

$$F_{sum}(1, F_{sum}(F_{neg}(F_{shift}(F(x), 2)), F_{shift}(F(F(x)), 5)))$$

The polynomial representation that results from this composition is $F(x) = 1 - x^2/4 + x^4/32$. The specification derived in Example 4.5.2 and $F(x)$ differ only in the coefficient of the 4th order term. For $0 < x < 1$, this altered coefficient yields a polynomial that is within 1.3% of the specification.

4.5.3 Cyclic Circuits

As discussed in Section 4.3.1, when the order of a circuit with feedback can be bounded, a polynomial representation for that circuit can be determined exactly and the matching techniques described above can be used. Given a specification with bounded order n_s and a cyclic component **F(x)** with unbounded order, the inequality:

$$n_y^t \cdot n_{x1} \le n_s \le n_y^t \cdot n_{x1} + \sum_{i=0}^{t-1} n_y^i \cdot n_{x2}$$

can be solved for t (where n_{x1} is the order of the initialization branch $\mathbf{f_1(x)}$, n_{x2} is the order of feedback branch $\mathbf{f_2(x, F_{feedback})}$ with respect to **x**, n_y is the order of the feedback branch with respect to the feedback input, and t is the number of times the feedback branch is executed). The solution to this inequality provides the bounds on t within which $F(x)$ can have the same order as S(x), and therefore possibly implement S(x).

If S(x) has unbounded order, then S(x) is implemented by **F(x)** if and only if the specification of the initialization branch of S(x), $s_1(x) = f_1(x)$ and the specification of the feedback branch of S(x), $s_2(x, S_{feedback}(x)) = f_2(x, F_{feedback}(x))$. Thus, if a function **F(x)** does not have a bounded order, and therefore no polynomial representation, it can still be compared to a specification S(x) by comparing the initialization and feedback polynomials of S(x) and $F(x)$. An example of this is shown in Section 4.7.2.

4.6 Complexity Issues

The order computation techniques described for combinational circuits in Chapter 3, as well as the extension described thus far in Chapter 4, are of quadratic complexity with respect to the size of the BDD representation of **F(x)** and output word length. Solving the set of linear equations for polynomial coefficients is of cubic complexity with respect to the order of the polynomial, and we assume this order is small (less than the discontinuity threshold). However, the underlying BDD data structure can be of exponential complexity for common functions. Thus, reducing the complexity of polynomial computation requires reducing the complexity of the order computation which, in turn, requires reduction of the complexity of the BDD.

Assume a function **F(x)** has a BDD with 2^m intermediate nodes, where **x** is an *m* bit word. If **x** is partitioned into two words ($x_{m-1}x_{m-2}...x_{m/2}00...0$) and ($00...0x_{m/2-1}x_{m/2-2}...x_0$), the BDDs that describe each partition will require no more than two sets of $2^{m/2}$ intermediate nodes. Similarly, partitioning **x** into *C* words will result in a worst case total node count of $T = C2^{m/C}$. Minimizing T with respect to m yields:

$$dT/dC = 2^{m/C} - (m/C)\,2^{m/C} \cdot \log_{10}2$$

$$= \left(2^{m/C} \cdot (1 - (m/C) \cdot \log_{10}2)\right)$$

$$\Rightarrow C = m \cdot \log_{10}2$$

Partitioning **x** into words of length $(1/\log_{10}2) \approx 4$ will minimize BDD complexity. This will result in overall BDD complexity of $(m/4) \cdot 2^4 = 4m$.

Input partitioning reduces the complexity of computing polynomial representations by performing curve fitting with a subset of the points within an input domain. These points are spread logarithmically across the input domain.

Example 4.6.1 If the input to the function $\mathbf{F(x) = x^2}$**:** $B^{32} \rightarrow B^{64}$ is partitioned into 8 input words, each of length 4, the following input values, expressed with an integer encoding, are considered when performing order computation:

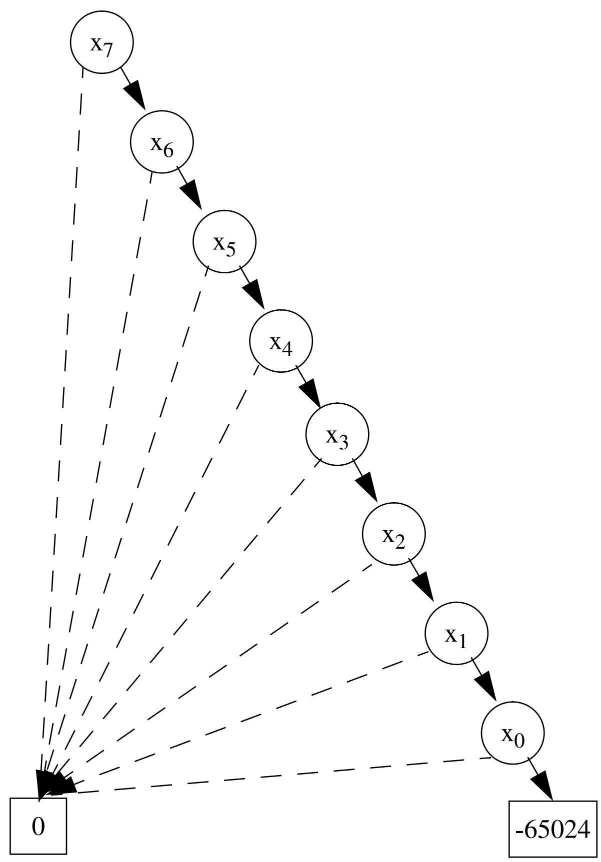
$$[2^0 \cdot 1, 2^0 \cdot 2, ..., 2^0 \cdot 15], [2^4 \cdot 1, 2^4 \cdot 2, ..., 2^4 \cdot 15], ..., [2^{28} \cdot 1, 2^{28} \cdot 2, ..., 2^{28} \cdot 15].$$

Thus, inputs outside of this subdomain are not considered when performing order computation. For the majority of complex elements that implement mathematical operations, this does not introduce inaccuracies in the representation, as the curve that is described by the subdomain exactly matches that described by the complete domain. However, if a circuit implements an input/output pair that satisfies the following conditions: (1) the input value is excluded from the subdomain (e.g. $x = 2^4 + 1 = 17$ in Example 4.6.1) and (2) that input/output pair does not lie on the polynomial (e.g. if $F(17) \neq 289$) that is determined from the subdomain, the polynomial representation is not accurate. Once $F(x)$ has been computed from the restricted domain, possible inaccuracies can be detected by determining if $\mathbf{F(x)}$ -$F(x)$ is identically zero. Since this difference is likely to be nonzero for few input points, we assume the *BMD representation for this difference is of low complexity. Thus, the accuracy of the polynomial representation, and the points at which it is inaccurate, can be determined efficiently by computing the maximum value of $\mathbf{F(x)}$ - $F(x)$ from its *BMD representation.

Example 4.6.2 Consider the following component description with a single 8 bit input:

$$\mathbf{F(x)} = x_7x_6x_5x_4x_3x_2x_1x_0 +$$
$$(\bar{x}_7+\bar{x}_6+\bar{x}_5+\bar{x}_4+\bar{x}_3+\bar{x}_2+\bar{x}_1+\bar{x}_0)(128x_7+64x_6+32x_5+16x_4+8x_3+4x_2+2x_1+x_0)^2$$

Using the complexity reduction technique, the input word is split into two words $\mathbf{x_u} = x_7x_6x_5x_4$ and $\mathbf{x_l} = x_3x_2x_1x_0$. Order computation with respect to each word yields an order two polynomial. Coefficient computation results in the polynomial representation $F(x) = x^2$. To determine the accuracy of the polynomial representation the *BMD for $\mathbf{F(x)}$ - $F(x)$ is computed:



From the *BMD, the polynomial representation is determined to contain a single inaccuracy at $x = 255$ that is in error by -65024 units.

Note that partitioning will guarantee that the order of the polynomial representation for a component is less 2^4. For those circuits implementing functions of order greater than 2^4, a polynomial representation is determined through domain partitioning and approximation, as explained in Sections 3.3 and 4.4. In practice, very few circuits implement functions of order greater than 2^4.

4.7 Applications

To illustrate the application of polynomial methods, the synthesis of two applications using the POLYSYS synthesis suite is described. POLYSYS implements polynomial methods to perform component matching. A JPEG Encode block is first synthesized to demonstrate order computation and discontinuity detection. An IIR filter is then mapped to an existing filter to demonstrate synthesis with synchronous library elements and approximation.

4.7.1 JPEG Encode Application

Generating polynomial descriptions allows a specification and implementation to be compared by computing the numerical difference between the polynomials. Consider the DC path for the JPEG encode system described in Figure 3 and specified in more detail in Figure 11. The inputs x(i, j) describe grayscale values for an 8x8 pixel block and output DC represents the encoded DC value for that pixel block.

(1) DCT

$$DCT = \sum_{i=0}^{7} \sum_{j=0}^{7} x(i,j)$$

(2) Quantize

$$Q = DCT/128 - DC_{previous}$$

(3) Coefficient Code

$$C = \log_2 Q$$

(4) DC Code

C	BaseCode	Length
0	010	3
1	011	4
2	100	5
3	00	5
4	101	7
5	110	8
6	1110	10
7	11110	12
8	111110	14
9	1111110	16
10	11111110	18
11	111111110	20

DC = (BaseCode<<C) + (Q mod 2^C)

Fig. 11 Arithmetic specification of the blocks for the DC path of JPEG encode (inputs: x(i, j), output: DC).

Specifications for four system blocks are described: (1) DCT, (2) Quantize, (3) Coefficient Coding, and (4) DC Coding. Three library elements were generated by synthesizing the Verilog code shown in Figure 12. Polynomial representations were computed from the resulting netlists.

Component 1
```
assign F1 = x1 + x2 + x3 + ... + x64;
```

Component 2
```
assign F2 = x1 - x2;
```

Component 3
```
always @ (x1) begin
    if (x1[11]) begin
        F3 = {9'b11111110, x1[10:0]};
    end elsif x1[10] begin
        F3 = {8'b1111110, x1[9:0]};
    end elsif x1[0] begin
        F3 = {3'b011, x1[0]};
    ...
    end else begin
        F3 = 3'b010;
end
```

Fig. 12 Verilog implementations synthesized to produce library elements F1, F2, and F3.

The first component requires that an order computation be performed for each input. The order of element **F1(x1, x2, .., x64)** with respect to each input is determined to be one and, after coefficient computation, the polynomial representation is:

$$F1(x1, x2, ..., x64) = x1 + x2 + ... + x64.$$

The order of element **F2(x1, x2)** block is similarly determined to be one with respect to **x1** and **x2** and the resulting polynomial representation is:

$$F2(x1, x2) = x1 - x2.$$

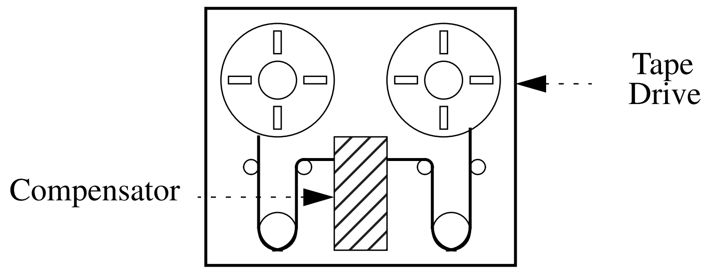
Order computation for element **F3(x1)** yields an order greater than the discontinuity threshold of 4. As a result, the upper bits of the inputs to each block are successively set to 0 and 1, as described in Section 3.3, and the following partitions and corresponding polynomial representations are determined:

Domain	DC Polynomial
$x1 = 0$	$F3(x1) = 2 + x1$
$0<x1<2$	$F3(x1) = 6 + x1$
$1<x1<4$	$F3(x1) = 16 + x1$
$3<x1<8$	$F3(x1) = x1$
$7<x1<16$	$F3(x1) = 80+ x1$
$15<x1<32$	$F3(x1) =192 + x1$
$31<x1<64$	$F3(x1) = 896 + x1$
$63<x1<128$	$F3(x1) = 3840 + x1$
$127<x1<256$	$F3(x1) = 15872 + x1$
$255<x1<512$	$F3(x1) = 64512 + x1$
$511<x1<1024$	$F3(x1) = 26e4 + x1$
$1023<x1<2048$	$F3(x1) = 1e6 + x1$

Performing a numerical comparison between the specification for DCT and *F1(x1, x2, ..., x64)*, the specification for Quantization and *F2(x1, x2)*, and the specification for Coding and *F3(x1)* reveals an exact match for each ($\varepsilon = 0$). Thus, the specification can be implemented by composing the complex components that exist in the library.

4.7.2 IIR Filter Application

Many embedded applications require digital filters to control mechanical operations. Common examples include altitude control systems for satellites, yaw dampers in airplanes, and fuel injection controllers in automobiles. We will apply polynomial methods to determine an existing filter, from a library of filters, suitable for reuse in a tape drive controller (Figure 13). Although reuse is not commonly implemented in tape drive design, tape drive compensation filters provide a compact, illustrative example of how approximation, synchronous elements, and feedback are handled by polynomial methods.



Transfer Function:

$$H(z) = \frac{.094 - .28z^{-1} + .19z^{-2} + .19z^{-3} - .28z^{-4} + .094\, z^{-5}}{1 - 5z^{-1} + 10z^{-2} - 10z^{-3} + 5z^{-4} - z^{-5}}$$

Fig. 13 Digital filter used as a compensator for controlling the move of a tape through a tape drive

The velocity of the tape with the tape drive is controlled by a voltage applied to the reel motor. This voltage is a function of past velocities, and therefore past voltages, as well as the displacement required to position the tape properly. An existing circuit implementation within the library of filters is shown in Figure 14, with combinational blocks already described by polynomials.

```
input: x                    output: F
x_q = REG(x)                F_q = REG(F)
x_qq = REG(x_q)             F_qq = REG(F_q)
x_qqq = REG(x_qq)           F_qqq = REG(F_qq)
x_qqqq = REG(x_qqq)         F_qqqq = REG(F_qqq)
x_qqqqq = REG(x_qqqq)       F_qqqqq = REG(F_qqqq)

  H1 = 160F_q - 320F _qq + 320F_qqq
  H2 = - 160F_qqqq + 32F_qqqqq
  H3 = x - 3x_q + 2x_qq + 2x_qqq
  H4 = 3x_qqqq + x_qqqqq
  H = H1 + H2 + H3 + H4
  F = H>>5
```

Fig. 14 Circuit description for library element to be compared to tape controller specification.

The challenge is to determine if the circuit can be allocated to implement the following specification, generated from MATLAB:

$$\begin{aligned}S(x) = {} & 5S(x@1) - 10S(x@2) + \\ & 10S(x@3) - 5S(x@4) + S(x@5) + \\ & .09375x - .28125(x@1) + .1875(x@2) + \\ & .1875(x@3) - .28125(x@4) + .09375(x@5)\end{aligned}$$

The first step in generating a polynomial representation for the circuit described in Figure 14 is to break the feedback paths. This results in $\mathbf{F_{feedback}}$ replacing **F** in the list of equations and being added to the list of inputs. The next step in generating a polynomial representation requires generation of the equivalent combinational circuit. Progressing down directed acyclic graph that represents $\mathbf{F(x, F_{feedback})}$, the first rooted subgraph represents the assignment **x_q = REG(x)**. This subgraph is duplicated, generating an additional circuit input **x@1,** and the original subgraph is removed. Subsequently, the rooted subgraph ending with **x_qq** is duplicated, generating an additional circuit input **x@2** and the original subgraph corresponding to the assignment to **x_qq** is removed. Continuing this process, the equivalent combinational circuit is generated, resulting in a circuit with the following inputs: {**x**, **x@1**, ..., **x@5**, $\mathbf{F_{feedback}}$, $\mathbf{F_{feedback}}$**@1**, ..., $\mathbf{F_{feedback}}$**@5**}. The nodes in the original graph that represented assignments to each of {**x_q**, ..., **x_qqqqq**, **F_q**, ..., **F_qqqqq**} were removed as they have been replaced by {**x@1**, ..., **x@5**, $\mathbf{F_{feedback}}$**@1**, ..., $\mathbf{F_{feedback}}$**@5**}. The complete set of resulting equations is:

$$H1 = 160F_{feedback}@1 - 320F_{feedback}@2 + 320F_{feedback}@3$$

$$H2 = - 160F_{feedback}@4 + 32F_{feedback}@5$$

$$H3 = x - 3x@1 + 2x@2 + 2x@3$$

$$H4 = 3x@4 + x@5$$

$$H = H1 + H2 + H3 + H4$$

$$\mathbf{F = H}>>5$$

At this point, the circuit description has no feedback paths and no registers.

Order computation with respect to each of {$\mathbf{F_{feedback}}$, $\mathbf{F_{feedback}}$**@1**, ..., $\mathbf{F_{feedback}}$**@5**} results in an order of one for each input. However, the order of the circuit with respect to each of {**x**, **x@1**, ..., **x@5**} is very large, indicating that a representation of an approximation of this circuit will be more efficient. Computation of **F(x+1, x@1, ...) - F(x, x@1, ...)** reveals that $-1 < F(x+1, x@1, ...) - F(x, x@1, ...) < 1$. A similar result is determined for **x@1**, ..., **x@5**. Thus, the term that each of {***x,*** *x@1*, ..., *x@5*} contributes to the polynomial representation of the circuit can be represented by an approximation of order 1, of the form x(Encode(**F**(11.11) **- F(0)**))/2^n. Following the error quantification steps outlined in Section 4.4.2, the bound on the error contributed by approximating each term of the polynomial that contains one of $\{\boldsymbol{x}, x@1, ..., x@5\}$ is $-.968 < \Delta < .968$. After performing coefficient computation, the following polynomial representation for the circuit is determined:

$$\begin{aligned} F(x) = {} & 5F_{feedback}(x@1) - 10F_{feedback}(x@2) + \\ & 10F_{feedback}(x@3) - 5F_{feedback}(x@4) + \\ & F_{feedback}(x@5) + .093749x - .281246(x@1) + \\ & .18749(x@2) + .18749(x@3) - .281246(x@4) + \\ & .093749(x@5) \end{aligned}$$

After closing the loop by setting $\mathbf{F_{feedback}} = \mathbf{F}$, the specification S(x) and implementation **F(x)** can be compared by comparing their representative polynomials. The coefficients of S(x) and *F(x)* do not match exactly, due to the approximation of *F(x)*, but are the same within 10^{-4}. Thus, the existing component can be allocated to implement the specification if the circuit tolerance of 10^{-4} is acceptable.

4.8 Experimental Results

To quantify the performance of order computation, a combinational multiplier, with input lengths ranging from 4 bits to 64 bits, was constructed out of combinational 4 bit multipliers, and the polynomial representation determined using POLYSYS. Multiplier logic was synthesized from Verilog to construct the Boolean equations that implement the Synopsys DesignWare multiplier. These equations were then ported to the Cal-2.0 BDD package which was used to perform BDD operations in POLYSYS. Experiments were performed on a 200MHz R4400 Indy Workstation with 256MB of memory.

The time required to determine the order of this circuit is shown in Figure 15(a) and, for the 64 bit multiplier, the order was computed in under 80 seconds. Note that by using the complexity reduction methods from Section 4.6, order computation was performed on successive 4 bit chunks of each input word. This yielded a maximum BDD size of 61 nodes which fit completely in the 16KB cache.

As expected, execution time varied with the square of the size of the input word. This is due to the function **F(x, y)** being of order one with respect to each input and having two inputs. Note that a similar computation for a function with polynomial representation $F(x) = x + K$ would have been of linear complexity with respect to the size of **x** and a more complex function such as that with polynomial representation $F(x) = x^2y^2$ would have varied with the fourth power of the size of the input word.

To quantify the performance of polynomial methods for synchronous circuits, experiments were conducted, to gauge the relationship between the execution time required to generate equivalent combinational circuits and the number of registers (Figure 15(c)). The circuits on which this was performed were 16 bit accumulators with between one and 5 register stages (i.e. $F(x) = x + x@1$, $F(x) = x + x@1 + x@2$, ..., $F(x) = x + x@1 + ... + x@5$). Execution time varied quadratically with the number of registers. Note that the register removal tool is written in Perl and the execution times in Figure 15(c) can be reduced greatly using compiled code.

Further experiments were conducted to determine the execution time of circuit approximation relative to input bit width. Polynomial approximations were computed for the circuit that implements the function **y = (x** >> 1**)** for input bit widths ranging from 4 to 128 bits (Figure 15(d)). While of high order complexity, approximations completed quickly, even for the widest datapaths. The accuracy of circuit approximation was determined for several circuits of bit width 16 (Figure 15(e)), all of which resulted in an error of less than 2 units over the integer range [0, 2^{16}-1]. These experiments were performed with compiled code.

Word Sizes	Logic Ops	Exec. Time
4	2003207	0.41s
8	8012236	1.34
16	32050480	4.76
32	128197824	19.31
64	512783104	79.30

Fig. 15(a) Execution time required to determine F(x, y) = xy is of linear complexity with respect to x and y.

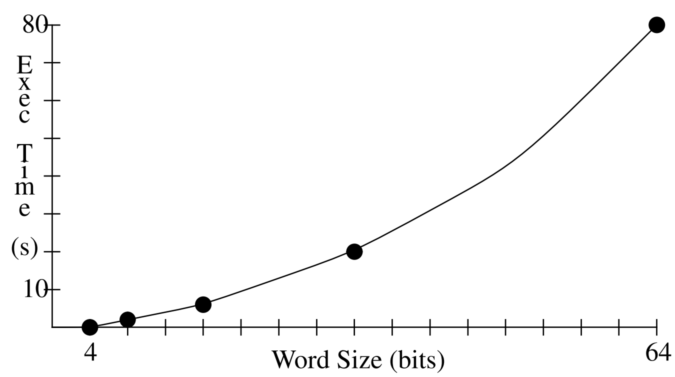


Fig. 15(b) Graph of execution times in 15(a).

Accumulator Stages	Number of Registers	Exec. Time
1	16	7.76s
2	32	25.84
3	48	79.47
4	64	177.50
5	80	326.19

Fig. 15(c) Execution time required for register removal on 16 bit accumulators.

Word Sizes	Exec. Time
4	0.01s
8	0.05
16	0.19
32	1.11
64	9.14
128	76.80

Circuit Function	Approx. Error
x/8	0.87
x/4	0.75
x/2	0.50
3x/4	1.75
7x/8	1.87

Fig. 15(d) Execution time for determining an approximation to the function x/2
(e) Accuracy of approximation for several 16 bit functions

4.9 Summary

In performing high level synthesis with complex components, automating component matching requires a means for quickly determining whether an existing block performs the function outlined in the specification. Current methods for completing this task become prohibitively memory intensive or time consuming for circuits that implement complex functions. Chapters 3 and 4 have described an algorithm for performing component matching with complex library elements by constructing word-level polynomial representations for combinational and sequential circuits.

Circuit specifications can be efficiently matched to existing implementations by generating the unique minimum order polynomial functions for the specification and the implementation and comparing those polynomials. These functions can be generated with quadratic complexity with respect to the number of input bits to each function. Discontinuities in the specification or implementation can be detected, allowing polynomial representations to be computed for intervals between discontinuities. For sequential circuits, the equivalent combinational circuit can be derived, from which a polynomial representation can be computed. Furthermore, an approximate polynomial representation can be derived for those circuits that contain many discontinuities and the error of that approximation can be quantified. An application of these techniques was demonstrated in mapping the specification of a JPEG Encode block and an IIR filter to existing complex blocks.

Using polynomial representations, differences between a specification and implementation can be quantified, allowing tradeoffs between precision and speed. In addition, the ease with which polynomials can be composed can allow such differences to be compensated for by combining multiple existing blocks or constructing logic around a single block.

The methods presented in this chapter are well suited to matching blocks that have compact arithmetic representations, such as those found in DSP, computer graphics, and ALUs. Furthermore, these methods provide a means for separating control operations, such a branches, from arithmetic operations and detecting blocks that contain many discontinuities, such as controllers, based on the order of the polynomial representation.

Chapter 5
Interface Synthesis

5.1 Introduction

In order to automate design reuse, methods for connecting system components must be developed. The goal of this chapter is to develop an algorithm to automate the process of generating interfaces between hardware subsystems. The algorithm presented here can be used to generate a cycle-accurate, synchronous interface between two hardware subsystems, that communicate with different protocols, given an HDL model of each subsystem. It is important to note that the techniques presented here provide hooks for implementing arbitration algorithms that can be determined from higher level optimization algorithms. The algorithms presented here have been implemented in POLYSYS to generate interfaces between complex library components. In particular, these techniques can be used to generate interface between elements that have been allocated using the techniques of Chapters 3 and 4. For example, POLYSYS has been used to generate an interface between a MIPS microprocessor and the SRAM that comprises its secondary cache. Interface generation for the MIPS R4000 is described.

The basic purpose of an interface is to facilitate the movement of data. Data could be addresses, commands, values destined for a memory location, or some combination of these descriptions. In order to allow hardware subsystems that follow different protocols for moving data to communicate with one another, the algorithms presented here map these protocols into a standard communication scheme. This scheme is then implemented in an interface architecture that is general enough to accommodate the requirements of any target interface. The terminology *client* is used in this chapter to indicates a component that is sending data and *resource* indicates a component that is receiving data. Note that a component may, at times be a client (e.g. a CPU sending a read request to memory), and at times be a resource (e.g. the same CPU receiving data from memory).

5.2 Overview

The algorithms presented here are used to generate synchronous component interfaces. The components may operate at different frequencies and may employ unidirectional or bidirectional busses. Bidirectional busses, such as those employed by PCI or VME, are handled by treating the bus as two unidirectional busses and computing two interface controllers. Combining the resulting controllers yields a state machine that governs communication over the bidirectional bus. Multi-way interfaces that allow multiple clients to interact with multiple resources are synthesized by dynamically establishing a point to point link between the client and the resource (Figure 16). This work assumes that the datatype that a client is sending matches the datatype that the resource expects to receive. It further assumes that the built-in component protocols are *well-posed* in that the component is deterministically drivable into a data transfer state or the data transfer states can be detected from the current and previous values of component control signals.

The interface architecture is described in Section 5.3. From the component model described in Section 5.4.1, a state machine is synthesized to map the component's communication protocol into a standard protocol that other interfaces can understand (Section 5.4.2). The data formats that a client component employs are translated into formats that the resource component can understand (Section 5.4.3).

The POLYSYS hardware composition tool requires the user to supply an HDL description of the component being interfaced to, the name(s) of the ports across which data is transferred, and the names of ports that the interface does not have access to (uncontrollable ports). The names of the uncontrollable ports (e.g. reset) must be supplied so that POLYSYS does not manipulate these ports when creating a component interface.

5.3 The Interface Architecture

Unlike most hardware interface synthesis techniques, this research links hardware components through a standard architecture rather than by attempting to map one component interface into another. This allows interfaces to be synthesized for a broad range of components. In addition, it allows multiple components to be linked via the same interface.

5.3.1 Architectural Blocks

The interface architecture includes a state machine for protocol conversion, a send and receive buffer for transaction information (which must be saved while a resource is unavailable), and an arbiter to govern access to a resource (Figure 16). Although hooks are

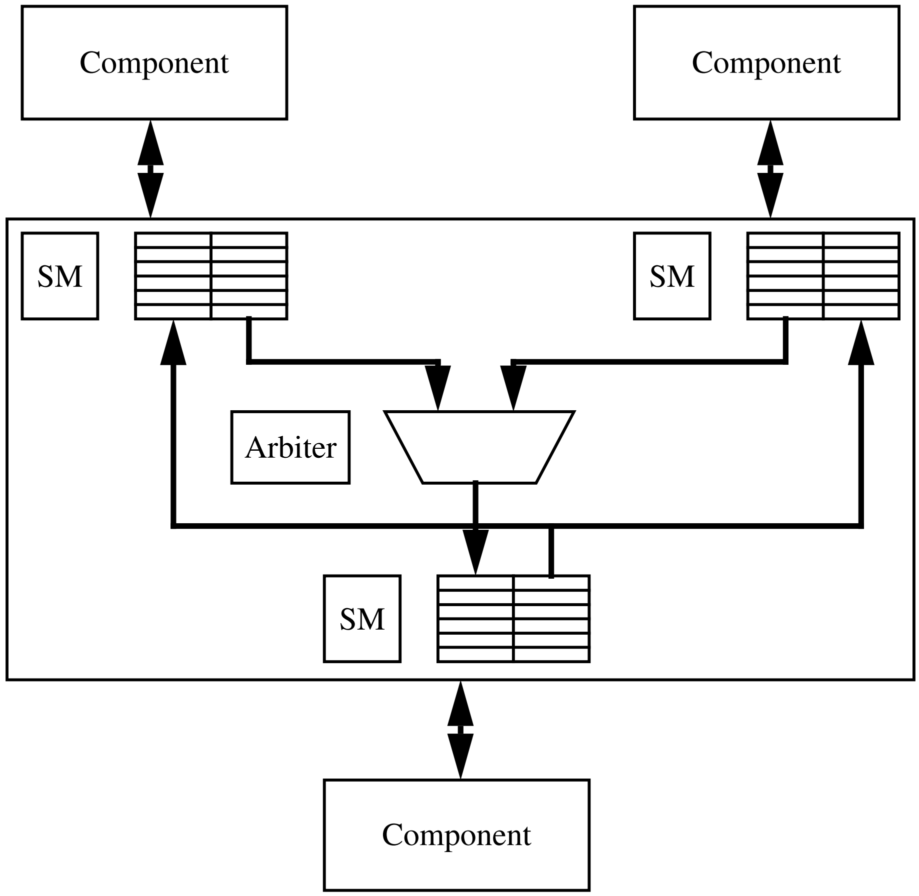


Fig. 16 High level view of a three component implementation of the interface architecture

provided to allow the implementation of an optimized arbitration algorithm, the details of the arbitration scheme are not required for interface synthesis.

When the state machine for protocol conversion detects that a component is sending data, the data is placed in the receive buffer for that interface. This data will be passed to the send buffer for the resource interface. Correspondingly, when the state machine detects that there is data in its send buffer, it executes the necessary signal assignments to transfer the data to the resource component. Sizing of these queues is currently a manual task, but could be automated in the manner of [AmBo91]. The default arbiter implements a round robin arbitration scheme to select a client receive buffer that will transfer its data into the appropriate resource send buffer.

5.3.2 Communication Scheme

The standard protocol employs four control signals each for the receive buffer and the send buffer (Figure 17). For the each buffer, the signals are implemented as follows:

Request - input to the buffer controller that indicates data is being sent to the interface;

Stall - output from the buffer controller that indicates that the buffer is full (the state machine must prevent other components or interfaces from sending any more data);

Valid - output from the buffer controller that indicates that valid data is in the buffer and ready to be transferred;

Acknowledge - input to the buffer controller that indicates that data has been read from the buffer (the buffer controller will increment the read address).

Resources are scheduled in the arbiter by selectively *Acknowledge*ing the client *Valid* signal. Data is transferred within the interface through four unidirectional busses: two for sending data to and receiving data from the arbiter, and two for sending data to and receiving data from the external component.

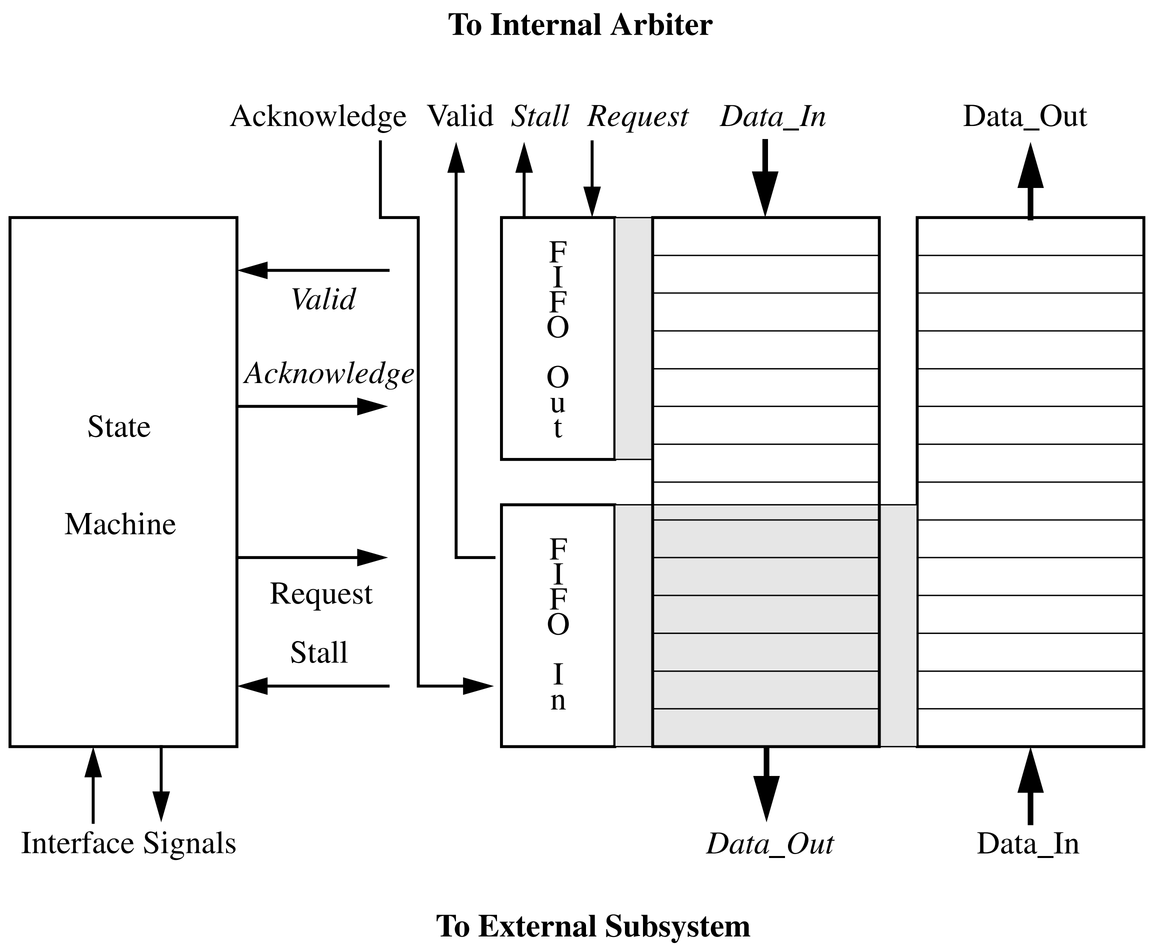


Fig. 17 Interface communication scheme

5.4 State Machine Generation

Given a component model that describes bus functionality (or a superset of bus functionality), conditions for transferring data to or from that component are determined. A sequence(s) of assignments to component ports is determined that will cause these conditions to become true. After the required assignments to component ports have been determined, they are executed on the component model, so as to resolve the values of control ports that are inputs to the synthesized interface. Once the values of all necessary input and output ports have been resolved, a state machine is generated that executes the required assignments and monitors the necessary control ports. This state machine provides a mapping from the communication protocol for a system component to the standard protocol that allows inter-component communication. The collection of operations necessary to perform a data transfer is referred to as a function.

5.4.1 Component Model

The component is abstracted as a list of assignments to variables and conditions for each assignment to be executed. This is similar to a guarded dataflow graph in which the guard is the condition for assignment execution and the dataflow is the set of assignments that share a guard.

Definition 5.1 A *component* is described by the list of tuples $\{C_i := <n_i, C_i, A_i, s_i>\}$ where $C_i = \{c_{ij}\}$ are the conditions under which the values $A_i = \{a_{ij}\}$ are assigned to the variable n_i. For the assignment to n_i, we assume there are n_i possible values $\{a_{ij}, j= 1,2, \ldots , n_i\}$ each selected by one and only one condition $c_{ij} \in \{0, 1\}$, and s_i indicates whether the assignment is combinational or synchronous. Thus,

$$n_i = \Sigma\, c_{ij} \cdot a_{ij}$$

Variables that are not component ports are referred to as internal variables.

5.4.2 Protocol Conversion Algorithm

The state machine for protocol conversion is represented, analogous to the Moore model, as a collection of state assignments and conditions for state transitions.

Definition 5.2 The *state machine for protocol conversion* is described by the tuple SM := $<S, T>$ where:

$$S := \{S_i\} \text{ is a list of states}$$

$T := \{(t_{ij} = 1) => S_i \text{ -> } S_j\}$ is a list of conditions governing state transitions.

The algorithm for generating the state machine for protocol conversion is completed in five steps: (1) generate a sequence S_f of functional states that cause a function to be executed, (2) generate a sequence S_x of exit states that cause an executing function to be halted ($S = S_f \cup S_x$), (3) generate the conditions $\{t_{ij}\}$ that govern the state transitions, (4) combine state sequences for multiple functions, and (5) reduce the number of states. The name of the component's data bus, referred to as the target variable, is supplied to initiate generation of the interface state machine. The algorithm is outlined in Figure 18.

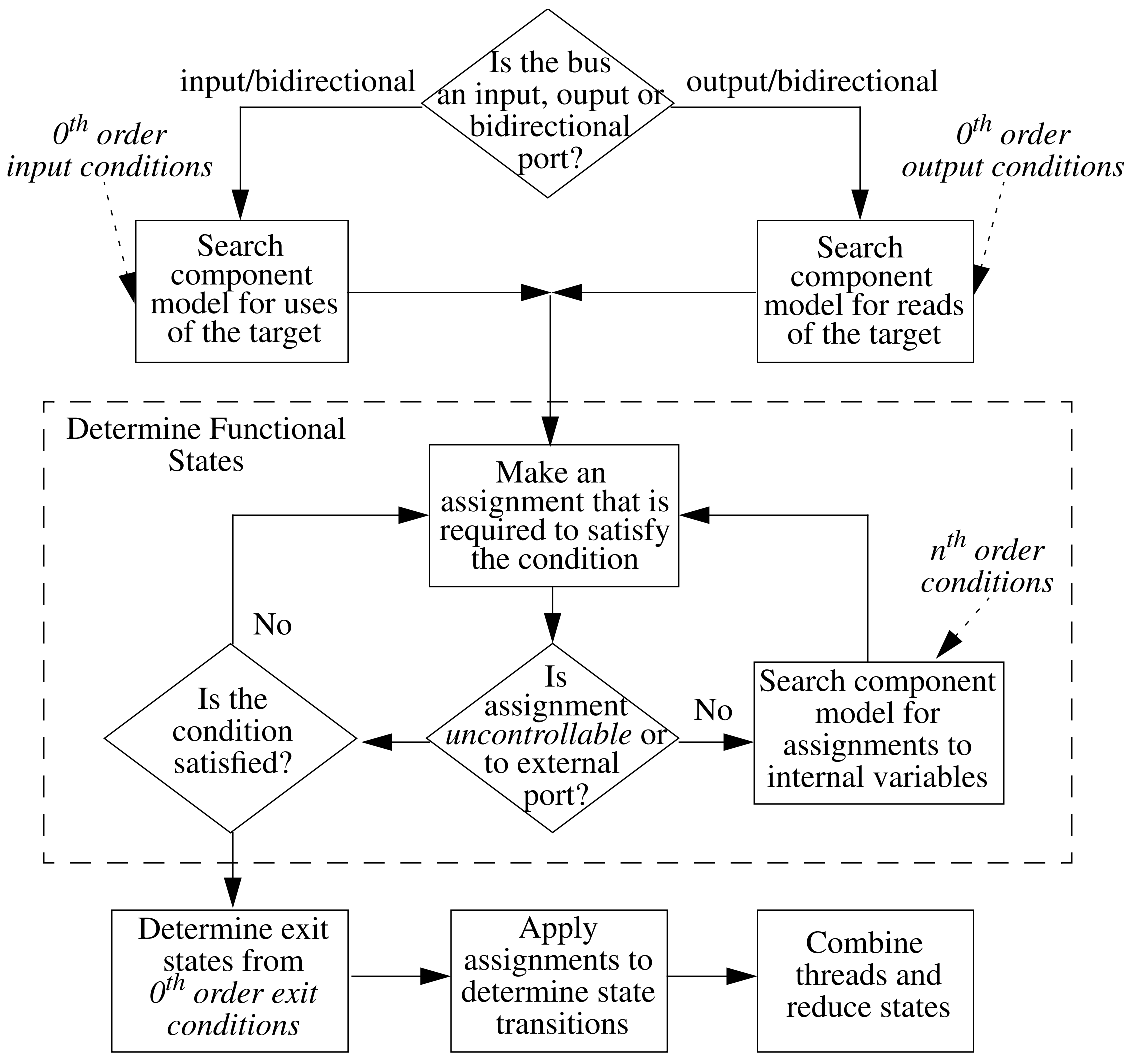


Fig. 18 Algorithm for generating the state machine for protocol conversion

If the target variable is an input or bidirectional port, the component model is searched for a use of this variable such as:

```
if signalA = valueA then
    signal <= target
if target = value and
    signalB <= valueB then ...
```

The conditions under which the target variable (target) is used are the 0^{th} order input conditions.

Definition 5.3 The set of *0^{th} order input conditions* for the target variable n_x is $I_0 := \{c_{ij}\}$ where either c_{ij} or a_{ij} is dependent on n_x.

In the previous example, $n_x = \{target\}$ and $I_0 = \{signalA = valueA, signalB = valueB\}$

If the target variable is an output or bidirectional port, the component model is searched for an assignment to this port such as:

```
if signalA = valueA then
    target <= signal
```

The conditions for these assignments are the 0^{th} order output conditions.

Definition 5.4 The set of *0^{th} order output conditions* for the target variable n_x is $O_0 := \{c_{xj}\}$.

In the previous example, $n_x = \{target\}$ and $O_0 = \{signalA = valueA\}$.

5.4.2.1 Executing a Function

Assignments that satisfy 0^{th} order conditions must be executed as part of the functional state sequence. That is, if $I_0 = \{signalA = valueA\}$, then state S_0 must contain the assignments {signalA <= valueA}. If signalA is a component port, then a single assignment can be made to allow that function to be completed. However, if signalA is an internal variable of the component, then conditions must be determined that will cause that variable assignment. For example, if a component description contained the sequential statements:

```
if externalSignalA = valueA and
        internalSignal = value then
    signal <= target
```

```
if externalSignalB = valueB then
    internalSignal <= value
```

then the following state sequence is generated:

State 1: extSignalB <= valueB

/* causes intSignal <= value in State 0 */

State 0: extSignalA <= valueA

These conditions that must be satisfied to cause the internal variable assignment are the n^{th} order conditions.

Definition 5.5 The set of *n^{th} order input conditions* for the target variable n_x is $I_n := \{c_{kl}\}$ where, given the (n-1)th order input conditions for n_x, i.e. I_{n-1},

$$\text{there exists an } c_{ij} \in I_{n-1}$$

$$\text{such that } (c_{kl} = 1) \Rightarrow (c_{ij} = 1).$$

In the previous example, n_x = {target}, I_0 = {externalSignalA = valueA and internalSignal = value} and I_1 = {externalSignalB = valueB}. An analogous definition applies to n^{th} order output conditions. If conditions of all orders can be satisfied by a sequence of assignments to component ports, the interface state machine can deterministically drive a component into functional states.

If an internal variable assignment requires an n^{th} order condition that in turn requires the same assignment, the variable assignment is uncontrollable and can not be deterministically driven to that value. For example, if a component description contained the sequential statements:

```
if reset = TRUE or intSignalA = valueA then
```

```
        target <= value
        intSignalB <= valueB,
    if intSignalB = valueB then
        intSignalA <= valueA,
```

then intSignalA <= valueA is uncontrollable if reset is uncontrollable.

Definition 5.6 An internal variable assignment ($n_k = a_{kl}$) is *uncontrollable* if $c_{kl} \in I_n$ and

(1) $c_{kl} \in I_m$ where $m < n$, or

(2) c_{kl} is dependent on another uncontrollable assignment.

If an uncontrollable internal variable assignment is encountered, no n^{th} order conditions for that assignment are generated. The thread of states that sought to cause execution of the assignment is aborted since the internal assignment can not be caused deterministically. This prevents the algorithm from looping indefinitely on a component description such as the one described in the previous example.

A component can not be deterministically driven to a functional state when an uncontrollable signal is encountered. However, given the original assumption that a component's built-in protocol is well-posed, the functional states can be detected by examining component control signals (Section 5.4.2.3).

5.4.2.2 Exiting a Function

In the same way that a sequence of states is determined to execute a function, another sequence of states is determined to end that function. That is, when the data transfer is completed, the interface must exit its corresponding data transfer state.

A component exits a data transfer state when an assignment is made that contradicts a 0^{th} order condition. This can be achieved with assignments to component ports or internal variables. For example, if a component description contained the statement:

```
if externalSignalA = valueA and
    internalSignal = value then
    signal <= target
```

then satisfying the condition not (externalSignalA = valueA and internalSignal = value) must cause the data transfer state in the interface to be exited. Such conditions are the 0^{th} order exit conditions.

Definition 5.7 The set of *0^{th} order exit conditions* for the target variable n_x is $E_0 := \{c_{kl}\}$ where, given the 0^{th} order input (or output) conditions for n_x, i.e. I_0,

there exists an $c_{xj} \in I_0$,

such that $(c_{kl} = 1) \Rightarrow (c_{xj} = 0)$.

n^{th} order exit conditions are generated and satisfied in the same manner that n^{th} order input and output conditions are generated and satisfied.

The exit state sequence is combined with the data transfer state sequence to obtain a complete state machine that can execute a function and be reset.

5.4.2.3 Generating State Machine Conditions

After the conditions for executing and exiting a function have been completely satisfied or found to be uncontrollable, the required assignments are executed on the component model. If the component drives a port to a valid value in a cycle, then a conditional statement governing the state transition must be added to the state corresponding to that cycle. For example, if the component description contained the sequential statements:

```
if externalSignalA = valueA and
        internalSignal = value then
    signal <= target
if uncontrollableSignal = valueX and
        externalSignalB = valueB then
    internalSignal <= value
    externalSignalC <= valueC
```

then the condition τ_{10} = {externalSignalC = valueC} must be satisfied to allow the state transition from state S_1 = {externalSignalB <= valueB} to state S_0 = {externalSignalA <= valueA}.

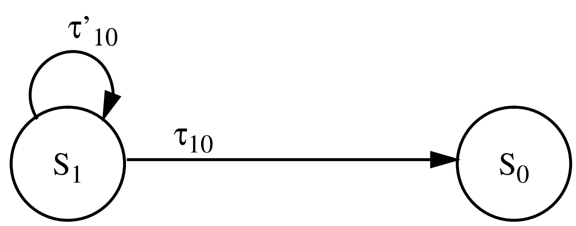


Fig. 19 Conditions governing state transitions.

Thus, if the interface state machine can not deterministically drive a component into a particular state (in the example above, the component state corresponding to S_1), it will be able to determine when the component has reached that state by evaluating its control signals. For example, in Figure 19, the interface state machine can not make an assignment that will force the component to enter a data transfer state. Thus, the interface waits in state S_1 until the component asserts the port signals (externalSignalC) associated with condition

τ_{10}. All components with well-posed built-in protocols must either be deterministically drivable into data transfer states or must assert control signals that can be decoded to detect data transfer states. A component that does not satisfy this condition can never be communicated with reliably, as it is impossible to determine when it is transferring or receiving data.

5.4.2.4 Combining Multiple Threads of Execution

A function's execution may require or allow more than one sequence to be executed in parallel. For example, the component description,

```
if internalSignalA = valueA and
        internalSignalB = valueB then
    signal <= target
```

requires multiple variables (internalSignalA, internalSignalB) to be set in tandem. This results in multiple threads of execution being generated, which must be combined into a single state machine.

ANDing two state sequences is a straightforward combination of state assignments and state transition conditions. The final states in a sequence and their predecessors are ANDed. An example is shown in Figure 20.

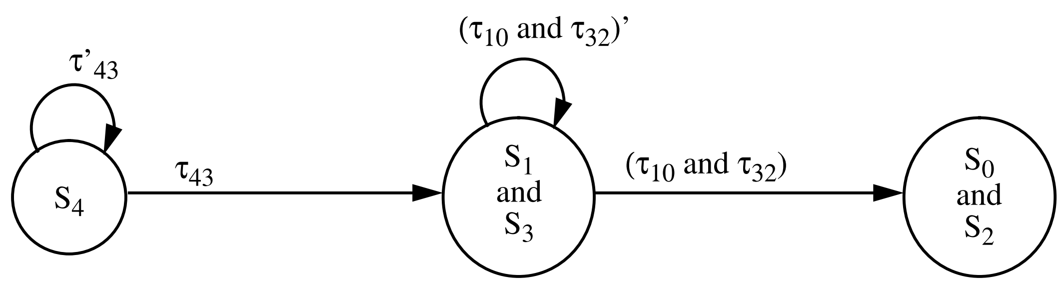


Fig. 20 The results of ANDing the state sequences (S_0, S_1) and (S_2, S_3, S_4).

If any two ANDed states contain contradictory assignments the threads are discarded.

In ORing two threads, only the head states are combined. Previous states are not ORed to prevent state sequences that contain a portion of each thread from being executed. An example is shown in Figure 21.

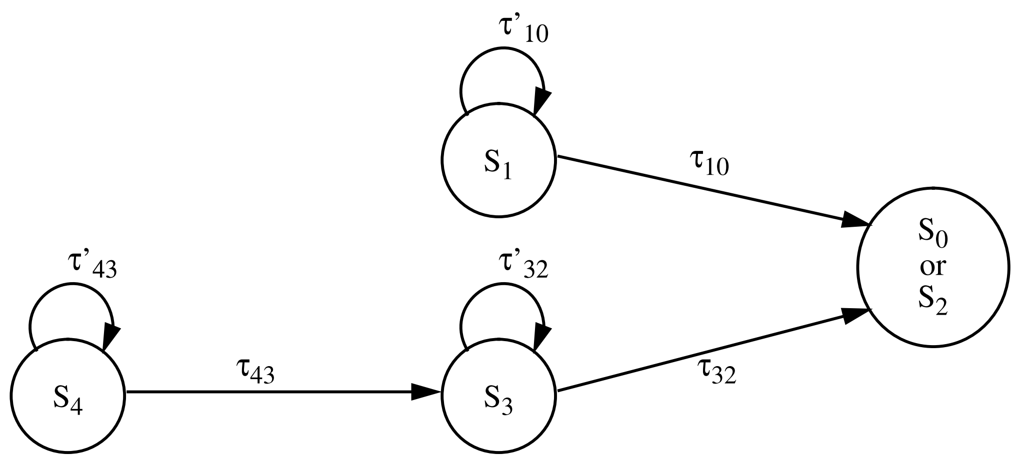


Fig. 21 The results of ORing the state sequences (S_0, S_1) and (S_2, S_3, S_4). Duplicated states are removed as shown in the next section.

5.4.2.5 State Reduction

A component will frequently share states between execution threads. For example, a component may contain a state in which it polls its subsystems for writes, and executes a different sequence for each write:

```
if state = POLL_STATE then
    case (subsystem)
    subA_write:
        state <= subA_WRITE_STATE
    subB_write:
```

```
state <= subB_WRITE_STATE
```

The number of states is reduced by joining threads of execution at points where their respective states are congruent.

There are two sets of requirements that states can satisfy to be considered congruent. First, two states are congruent if they stem from congruent previous states and the conditions for entry into both states are the same. Second, two states are considered to be congruent if they contain the same variable assignments and one of them is an exit state. The second requirement allows a state machine to be reset once it has executed a functional sequence.

Definition 5.8 States S_i and S_j are congruent if either

(1) for all x, there exists a y such that $\tau_{xi} = \tau_{yj}$ and S_x and S_y are congruent, or

(2) assignments of S_i = assignments of S_j and S_i satisfies E_0.

Congruent states are combined by creating a state in which the assignments of both states are executed. If one of the states is an exit state, then the conditions for entrance into the newly created state are ORed.

5.4.3 Datapath Translation

The state machine for protocol conversion allows two components to communicate by translating their control signals into the standard interface. However, components often employ different datapaths that must be reconciled if they are to communicate with one another. Two components may communicate through busses of different widths. Additionally, components may have multiple data transfer states, each of which is sending or receiving a different datatype. Thus, translating datapaths between interfaces imposes two requirements: (1) datapath widths must be reconciled, and (2) datatypes must be extracted from a transaction so that client "data-send" states can be matched with the appropriate resource "data-receive" states.

Datapath widths can be determined from the component description. When data is read from a client receive buffer, it is read into a register that is the width of the resource datapath. If the resource datapath is wider than the client datapath, an *Acknowledge* is returned to the client interface for every word that is popped off the client receive buffer. If the resource datapath is thinner than the client datapath, an *Acknowledge* is returned to the client interface when the resource register is filled.

Datatype extraction can be performed using a simplified version of the structures that ([ChOrBo95], [MaHa95]) used to achieve interface synthesis. These two employed signal sequences (SEQs) and protocol flow graphs (PFGs), respectively, to model the bit patterns that are required to interact with a component. In the case of POLYSYS, a sequence of higher level descriptions can be provided to allow the interface to detect which cycles or bits are transmitting an address.

Example 5.4.1 A component may write data to another by transferring a base address followed by a burst of data words. The other component may read data by reading an address followed by a single word of data. The data transfer states may be matched by specifying datatypes with the simplified PFGs shown in Figure 22.

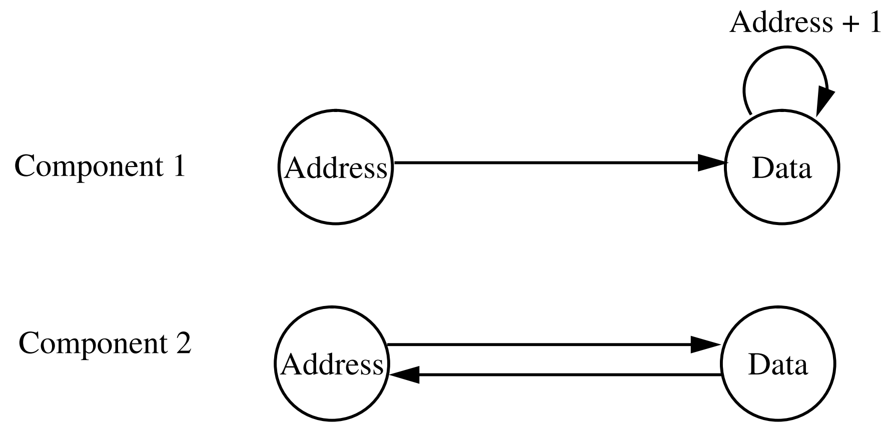


Fig. 22 Simplified PFGs that specify datatypes.

The PFGs shown in Figure 22 are simplified in that nodes describe only data transfers, not control operations.

An alternative mechanism for performing datatype extraction is to label the data transfer states for communicating components. For example, if a client contains two transfer states, one with the label DATA0 and another with label DATA1, the client interface input buffer will contain not only the data corresponding to that transfer, but also an encoding of the DATA0 or DATA1 label. This data and label is then transferred to the resource output buffer of the interface. When the data and label are popped from the resource output buffer, the interface state machine for the resource will drive the resource component into the data transfer state that has an equivalent label.

5.4.4 Hooks for Arbiter Implementation

The effect of an interface on system performance can be quite severe ([KnMa98]) due to the frequency of operation of the interface, communication overhead, bus widths and, in the case of multiway interfaces, arbitration of access to system resources. However, when reusing existing components, the frequency of operation of the interface and the bus width are fixed by the component. For example, a MIPS R4000 CPU communicates at 100MHz over a 32 bit bus and these characteristics are fixed regardless of the implementation. Furthermore, the communication overhead introduced in mapping a component protocol onto the standard protocol is exactly 6 cycles:

(1) assertion of the client *Request* signal to load data to the client input buffer.

(2) assertion of the client *Valid* signal to indicate data is in the client input buffer.

(3) assertion of the arbiter *Request* signal to load data to the resource output buffer.

(4) assertion of the arbiter *Acknowledge* signal to signal data has been transferred to the resource output buffer.

(5) assertion of the resource *Valid* signal to indicate data is in the resource output buffer.

(6) assertion of the resource *Acknowledge* signal to indicate that data has been read from the resource output buffer.

Thus, system performance is most impacted by the relationship of this additional overhead to the size of data transfers and the scheduling of access to system resources.

The size of data transfers and the scheduling of access to system resources are both controlled by the arbiter. Both of these characteristics are directly affected by the assertion of the arbiter *Request* signal. The arbiter can force all data transfers to be of a certain length by not asserting *Request* until the client input buffer is appropriately full. Furthermore, the arbiter can implement a scheduling order or priority by asserting *Request*s to a resource based on the provided schedule. Furthermore, pre-emption can be performed simply by deasserting the *Acknowledge* to a client in the middle of a data transfer. However, the client input buffer does not maintain data once it has been sent. Thus, pre-emption requires that a transfer can be completed by sending data in parts.

Example 5.4.1 The following Verilog code illustrates an arbiter that governs access to a resource by two prioritized clients and allows preemption.

```
if (Valid_Client0) begin
     Request_Client0 <= 1;
     Request_Client1 <= 0;
     Acknowledge_Client0 <= Acknowledge_Resource;
     Acknowledge_Client1 <= 0;
end else if (Valid_Client1) begin
     Request_Client0 <= 1;
     Request_Client1 <= 0;
     Acknowledge_Client0 <= 0;
     Acknowledge_Client1 <= Acknowledge_Resource;
end else begin
     Request_Client0 <= 0;
     Request_Client1 <= 0;
     Acknowledge_Client0 <= 0;
     Acknowledge_Client1 <= 0;
end
```

5.5 Example - Simple MIPS SysAD Interface

This example demonstrates how POLYSYS generates a state machine that converts the communication protocol of a MIPS processor with a simplified SysAD interface to the standard protocol. This generated interface can communicate with a similarly synthesized interface for another component such as RAM, a DSP, etc. In this section, however, we derive only the state machine that maps the MIPS bus protocol onto the standard protocol. The HDL model of the SysAD interface (not including the bidirectional buffer for SysAD) is given in Appendix A. In addition to the HDL model, the bus for data transferral (SysAD) and a list of uncontrollable ports (Reset, CPUwr, CPUrd, etc.) is supplied to the tool.

Upon determining that the target variable (SysAD) is a bidirectional port, POLYSYS first creates the state machine for input through the target variable. The component model is searched for the for the assignment * <= SysAD. The set of 0^{th} order input conditions I_0 = {Reset = 0 and bus_state = S_RECV and Valid_In = 1 and SysCmdi[8] = 1} is returned. State S_0 = {Reset = 0, bus_state = S_RECV, Valid_In = 1, SysCmdi[8] = 1} is created. Since bus_state is an internal variable and not uncontrollable, it is made the new target bus and the component model is searched for the assignment bus_state <= S_RECV. The set of first order input conditions I_1 = {Reset = 0 and bus_state = S_RECV_SPIN} is returned. State S_1 = {Reset = 0, bus_state = S_RECV_SPIN} is created, and the component model is searched for the assignment bus_state <= S_RECV_SPIN. The set of second order input conditions I_2 = {Reset = 0 and bus_state = S_SEND_IDLE and ((CPUwr and WrRdy) or (CPUrd and RdRdy)) = 0 and ExtRqst = 1} is returned and states S_2 = {Reset = 0, bus_state = S_SEND_IDLE, WrRdy = 0, ExtRqst = 1} and S3 = {Reset = 0, bus_state = S_SEND_IDLE, RdRdy = 0, and ExtRqst = 1} are created (note that CPUwr and CPUrd are uncontrollable and states corresponding their assignment are not generated). This process continues until n^{th} order conditions can be completely controlled from external ports or require uncontrollable assignments (e.g. Reset = 0, bus_state = S_SEND_IDLE).

The 0^{th} order exit conditions are determined by negating the 0^{th} order input conditions. The set of 0^{th} order exit conditions is E_0 = {ValidIn = 0 or SysCmd[8] = 0 or bus_state != S_RECV or Reset = 1}. This exit condition generates three exit states, two of which satisfy the first congruency criteria ({ValidIn=0} and {SysCmd[8] = 0}).

Now that all possible complete sequences of states have been determined, the state assignments are executed to determine the valid values on external ports during each state. Since Release and ValidOut are driven to valid values during state S_1, condition τ_{10} = (Release = 0 and ValidOut = 0) must be satisfied to complete the transition S_1 -> S_0. Similarly, $\tau_{21} = \tau_{31}$ = (Release = 1 and ValidOut = 0) must be satisfied to complete the transitions S_2 -> S_1 and S_3 -> S_1.

Given the assignments for each state S and the conditions T for transitioning between states, the state sequences are joined at congruent states. The three sequences generated by the exit states all contain congruent states leading up to the functional state. Thus, the threads are joined at each of these states. However, the exit states that perform {ValidIn = 0} and {ValidIn = 1, SysCmd[8] = 0, SysCmd[6:5] = H_NULL} are mutually exclusive, thus there are multiple branches exiting the functional state. According to the second congruency criteria, the exit state from {ValidIn = 0} is congruent to state S_0 and the exit state from {ValidIn = 1, SysCmd[8] = 0, SysCmd[6:5] = H_NULL} is congruent to state S_3. The state machine for transferring data to the MIPS processor is shown in Figure 23.

```
if (Reset) begin
        state = S_0;
end else begin
    case (state)
     S_3: begin
        if (valid) state = S_2;
    end;
    S_2: begin
        WrRdy = 0;
        RdRdy = 0;
        ExtRqst = 1;
        if (Release && !ValidOut) state = S_1;
    end;
    S_1: begin
        if (!Release && !ValidOut) state = S_0;
    end;
    S_0: begin                        /* label DATA0 */
        ValidIn = 1;
        SysCmd[8] = 1;
        SysAD = Data_Out;
        acknowledge = 1;
        if (!valid) begin if (...) begin
            ValidIn = 1;
            SysCmd[8] = 1;
            SysCmd[6:5] = H_NULL;
            state = S_3;
        end else begin
            ValidIn = 0;
        end;
    end;
    endcase;
end;
```

Fig. 23 State machine for transferring data to the MIPS CPU

The process above is repeated for data transferred from the MIPS processor to the interface. The only difference being that, initially, the functional states are determined by searching the expression list for SysAD <= *. The output state machine and input state machine are joined at states S_2 and S_3 in state reduction. The physical characteristics of the synthesized interface, not including the send and receive buffers, are shown in Figure 24. The speed, area and power consumption of the bus control logic synthesized above is not optimized in this research because this logic is rarely a significant factor in the overall speed, area and power consumption of the interface.

Library	LSI LCB007
Gate Count	1464
Max. Operating Frequency	166MHz
Est. Area	0.92 mm^2
Est. Power Consumption	348mW

Fig. 24 Physical characteristics of automatically generated interface to MIPS R400 (Does not include 256B receive and send buffer)

5.6 Summary

Composing blocks that are developed by different design groups with different communication protocols is an imperative in automating design reuse and IP sharing. This chapter has focused on converting the communication protocols of one or more components into a standard protocol. The interface architecture presented here provides a mechanism for implementing communication through the standard interface. This architecture enables the composition of synchronous blocks and provides hooks for optimizing system performance by prioritizing component communication. The example provided illustrates its ability to generate an interface to the standard protocol for a hardware block.

The techniques described above allow a designer to automatically generate an HDL model of an interface between two or more blocks given an HDL description of the corresponding blocks. We have assumed that communicating components utilize the same datatypes. In implementing these techniques, parsing of the input HDL has been completed in a front end module that can be adapted to different coding styles or representations. Interfaces between blocks with multiple busses can be generated when the control of these busses is separate. Furthermore, by communicating data through input and output buffers with separate read and write functionality, components can be connected that operate at different frequencies.

Chapter 6
POLYSYS Implementation

6.1 Introduction

In order for hardware engineers to follow a design methodology that incorporates reuse of complex components, the engineer must be able to search for usable components and evaluate the impact of their incorporation. However, the space of usable components is not only large and growing, but these components have been developed by many different vendors. Furthermore, vendors are loath to disclose intellectual property prior to purchase, complicating the search and evaluation process ([DaBoBe99]). The POLYSYS synthesis suite implements a client-server based search mechanism allowing designers to locate components and evaluate their functionality. These components are maintained at remote vendor sites, removing the need to transfer many large vendor libraries to customer sites and protecting the vendor's intellectual property. The distributed nature of this approach to synthesis allows for scalability in the number of components and vendors, adaptability in terms of the operations performed during synthesis, and robustness in terms of availability and security ([SpNe97]).

The POLYSYS synthesis client allows engineers to create hardware specifications, map these specifications onto existing blocks, and generate interfaces between these blocks without downloading the HDL or Boolean model of the block. The client partitions a specification into blocks that are likely to have existing implementations within a vendor library. The polynomial model for each block is then derived in the form of C functions. This model is then transferred to vendor sites to determine components that implement the functionality specified by the C functions. Upon discovery of a match to this model, the client requests generation of an interface to the component that implements the desired functionality.

This POLYSYS synthesis server allows vendors who create reusable components to publish the functionality of these components without disclosing implementation details. The server creates polynomial models of component functionality given a Boolean description of the component, using the techniques described in Chapters 3 and 4. This models is derived in the form of C functions. When the server receives a request to find a match for a given specification, component C models are compared numerically against the C models of the specification. The maximum numerical distance between these two C models is returned as well a key that indicates the matching component. Upon receiving an interface generation request that includes a component key, the server computes and returns an interface to the corresponding component, using the techniques described in Chapter 5. The complete client server implementation of POLYSYS is pictured in Figure 25.

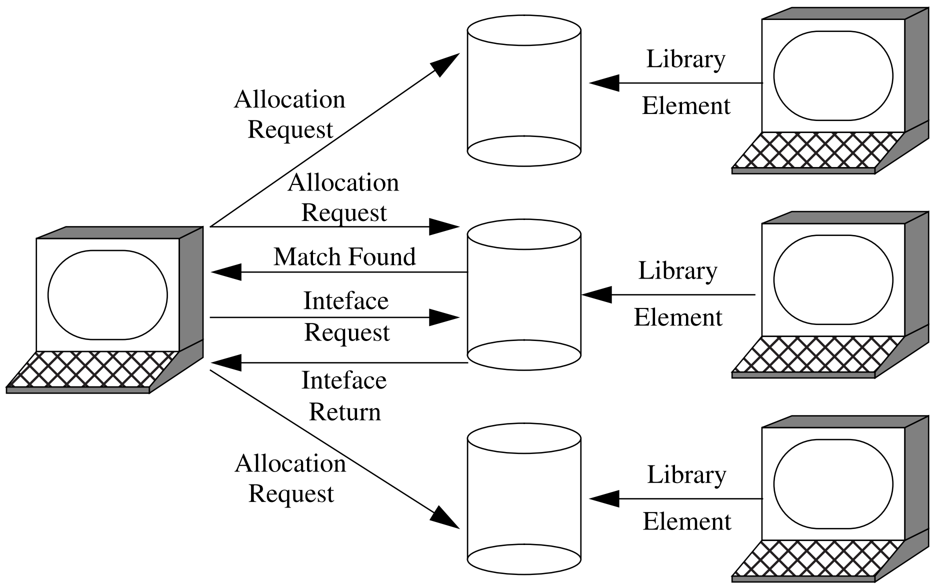


Fig. 25 POLYSYS synthesis suite

6.2 Client

A client allows entry of the system specification textually or by assembling functionality graphically. Textual specification is completed with RTL or behavioral Verilog HDL with extensions. Extensions include arithmetic operations such as division, exponentiation, and transcendental functions and a synthesis directive to indicate how a specification should be partitioned when allocation is performed (Section 6.2.1).

A client is composed of collection of input/output threads. When performing allocation, an output thread delivers specifications to vendor sites for allocation and an input thread awaits the results of the allocation. When requesting interface generation, an output thread delivers a key that indicates the component for which an interface is being requested and an input thread receives the resulting interface. To remove the need for tracking which response is associated with which request, a single input and output thread is established between each client and each vendor for each operation.

6.2.1 Specification Partitioning

In order to match a complex element to a system block, the polynomial model for the block's specification must be constructed. Performing this task requires that the input and output points of the block be determined. This is equivalent to partitioning a design into blocks which are candidates for mapping to an existing design. One very simple approach to this partitioning problem is presented.

A system partition is generated by each of the following constructs: (1) pragma begin and end blocks; (2) module begin and end blocks; and (3) the order and input/output characteristics of library elements. This policy is chosen because each of these constructs provides a semantic indication that the code contained within the partition is likely to be mapped to a reusable module.

While the first two policy decisions are based on the system specification, the order and input/output characteristics are determined from vendor libraries. The library element that has the highest order polynomial representation, the one with most inputs, and the one with most outputs is determined for each vendor library. A partition is created within a specification only if it is of lower order than the maximum order of library elements, has fewer inputs than the maximum number of inputs for library elements, and has fewer outputs than the maximum number of outputs for library elements. For example, in the specification of an antialiased line rasterizer (Section 7.1.1), in which the library maximums for order, number of inputs, and number of outputs were 2, 3, and 2 respectively, a partition would not be created that contained the following statements, as it would have an order of 1, 4 inputs, and 2 outputs:

```
dx = x2 - x1;
dy = y2 - y1;
```

A partition would be created, however, that contained the following statements (order 1, 2 inputs, 2 outputs):

```
incrE = 2*dy;
incrNE = 2*dy - dx;
```

All partitions for the computation of *incrE* and *incrNE* are shown in Figure 26. A

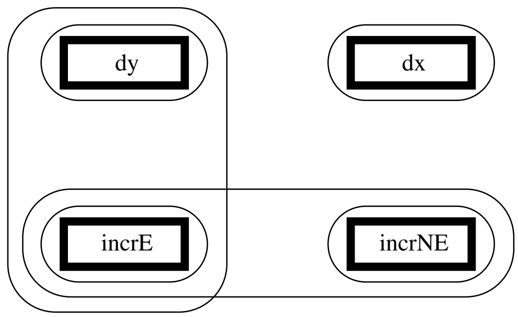


Fig. 26 Partitions for the computation of *incrE* and *incrNE* in the antialiased line rasterizer.

polynomial representation is determined for each partition that does not contain another partition, termed an *atomic partition*. These representations are then composed to determine polynomial representations for larger partitions. For example, the polynomial representation for the non-atomic partition:

```
dy = y2 - y1;
incrE = 2*dy;
```

would be constructed by substituting for *dy* in the polynomial representation for *incrE*, yielding *incrE = 2*(y2 - y1)*.

6.2.2 C Model Generation

In order to allow efficient computation of the numerical difference between a specification and a potential implementation, a partition is translated into a corresponding set of C functions. A partition is described by a set of functions with integer outputs that implement the mathematical operations performed in that partition and a set of corresponding functions with Boolean outputs that indicate within which domain those mathematical operations are performed.

Example 6.2.1, a specification requesting that multiplication always be performed yields the following C functions:

```
int polynomial0 (int x[]) {        BDD domain0 (int x[]) {
     return x[1] * x[0];            return trueBDD;
}                                   }
```

The specification for the coefficient coding portion of the JPEG encode example of Chapter 4 yields the following C functions:

```
int polynomial0 (int x[]) {        BDD domain0 (int x[]) {
     return 2;                       return (x[0] == 0);
}                                   }
int polynomial1 (int x[]) {        BDD domain1 (int x[]) {
     return 6 + x[0];                return (x[0]>0)&&(x[0]<2);
}                                   }
```

...

```
int polynomial15 (int x[]) {
     return 1000000 + x[0];
}
BDD domain15 (int x[]) {
     return ((x[0] > 1023) && (x[0] < 2048));
}
```

Upon completing computation of the C model for each system partition, the client requests a connection with the allocation server of each vendor. The C models are then sent to the vendor and the client awaits the results of component matching.

6.3 Server

Servers that perform computation of polynomial models of complex components, matching of polynomial models against system specifications, and interface generation are maintained at vendor sites. This allows designers to search for components without requiring vendors to release libraries.

6.3.1 Library Server

The library server computes a polynomial representation of a complex component given the Boolean equations that describe that component. When computation of a polynomial representation is performed, the server is programmed by four variables: (1) one that indicates whether an approximation should be computed for this function, (2) the discontinuity threshold to be used when separating arithmetic functionality from control operations, (3) the input word width, and (4) the output word width.

This representation is recorded, as described in Section 6.2.2 as a set of functions with integer outputs that implement the mathematical operations performed in that partition and a set of corresponding functions with Boolean outputs that indicate when those mathematical operations are performed.

When an element is added to the library, characteristics can be associated with that element. These characteristics include formulas or figures that enable computation of: (1) the power consumed by the component (2) the latency of operation, (3) the maximum operational frequency, and (4) an estimate of area (in gates) required by the component.

These figures can be a function of minimum feature size or operational frequency and are the basis for selecting a component from a group of valid implementations.

6.3.2 Allocation Server

The allocation server determines if any library elements match a specification by computing the numerical distance between the specification and each component in the library. Both the specification and library elements are represented by C functions that describe the operations that they perform and the domains within which those operations are performed.

6.3.2.1 Domain Computation

For each component, each domain of the specification (denoted SD_i) is compared against each domain of the component (denoted CD_j) by computing $(SD_i \cdot CD_j)$. If the result of this computation is the zero BDD, then the two domains do not overlap, and the numerical distance between the corresponding arithmetic functions need not be computed.

If the result of $(SD_i \cdot CD_j)$ is a non-zero BDD, then the two domains do intersect and the integer bounds within which these two functions overlap must be computed. Since the domains SD_i and CD_j are constructed such that each domain is defined uniquely by a single upper and lower bound in the integer domain, the overlap of any two domains can be determined by performing a binary search for the upper and lower bound of the domain. The following pseudocode illustrates these searches for a BDD with input words x[] where each input word x[i] contains input bits x[i][j]:

```
int[] generateUpperBound(BOOL x[][], BDD bdd) {
    foreach x[i] {
        upperBound = 0;
```

```
            foreach x[i][j] {
                bddPos = cofactorPos (bdd, x[i][j]);
                bddNeg = cofactorNeg (bdd, x[i][j]);
                if ((bddPos != false) || (bddNeg == false)) {
                    bdd = bddPos;
                    upperBound += (1<<j);
                } else
                    bdd = bddNeg;
            }
            upperBounds[i] = upperBound;
        }
        return upperBounds;
    }
    int generateLowerBound(BOOL x[][], BDD bdd) {
        foreach x[i] {
            lowerBound = 0;
            foreach x[i][j] {
                bddPos = cofactorPos (bdd, x[i][j]);
                bddNeg = cofactorNeg (bdd, x[i][j]);
                if ((bddPos == false) || (bddNeg != false))
                    bdd = bddNeg;
                else {
                    bdd = bddPos;
                    lowerBound += (1<<j);
                }
            }
            lowerBounds[i] = lowerBound
        }
        return lowerBounds;
    }
```

6.3.2.2 Numerical Distance Computation

Comparing a component and a specification now boils down to computing the numerical distance between two polynomials *F(x)* and *S(x)* over a domain *lowerBound* < *x*

< *upperBound*. This computation can be performed by finding the maximum value of *F(x) - S(x)* and *S(x) - F(x)* over the computed domain. Since the polynomial expressions implemented in digital systems tend to be simple and of low order, we rely on low-overhead methods that are completely numerical (i.e. require no symbolic derivative computations) to compute local extrema. Note that computing global extrema of polynomials is an NP-complete problem. Heuristic techniques, do not guarantee determination of global extrema. However, they are very effective in practice. This problem alone is an active and broad field of research. Methods for computing local and global maxima are reviewed in this section.

Computation of the maximum value of a function can be performed in many ways. For univariable functions, classic calculus would direct computation of the first derivative of that function. Points at which the first derivative is zero are local extrema. Furthermore, all local extrema are guaranteed to be found by this method. However, for multivariable functions, the computations required for derivative-based methods require the solution of systems of potentially nonlinear equations. There are many efficient techniques for finding a solution to systems of nonlinear equations, including the Newton-Raphson Method ([BoKr99]). However, finding a single solution guarantees only a local maximum. The problem of finding all solutions to such a system, thus guaranteeing a global maximum, is an NP-complete problem. A classic technique for finding all roots of a system of polynomial equations is provided in [GaZa79]. This technique solves a trivial system of equations, and iteratively introduces small perturbations into the system to make it similar to the system in question. A review of techniques for finding all roots of a system of polynomial equations can be found in [Mo98].

A commonly used technique for computing local maxima, without solving systems of nonlinear equations, is the Gradient Method. The Gradient Method is a search based technique that requires computation of the gradient vector G (the partial derivatives) of M(x) at a particular point P. From this vector, the direction of the search S is computed by the equation S = G/||G||. Single parameter minimization is then performed for M(x) in the direction of S by computing the maximum value of M(P + tS). A new point P' = P+tS is then computed an the search begins again. This continues until M(P') == M(P).

A completely numerical method for computing the maximum value of a polynomial is to evaluate the function M(x) = S(x)-F(x) many times and search for a local minimum. However, this requires intelligent selection of points at which to evaluate the function. One such method is simulated annealing ([KiGeVe83]). Simulated annealing requires computation of an initial value of M(x). A semi-random displacement from that initial value is then generated and the polynomial evaluated at the new location. If the value at the new location is greater than the previous location, the search position is updated. If the value at the new location is not greater than the previous location, the new location is accepted probabilistically. This method computes a global maximum, but is not guaranteed to find the maximum within a finite time period.

POLYSYS employs a technique developed by Nelder and Meade, known as Downhill Simplex Minimization ([NeMe65]). A simplex is a generalized triangle in N dimensions. This technique is initialized by generating a non-degenerate simplex that is specified by N points that are coordinates through which M(x) passes. From this simplex, the point with maximum M(x), denoted B, and second greatest M(x), denoted G, are determined. The minimum point, denoted W, is also determined. From these five points, three additional points are computed:

R = B+G-W

E = 3*(B+G)/2 - W

$$C = \text{Minimum}((B+G)/4 + W/2, 3*(B+G)/4 - W/2)$$

$$M = (B+G)/2$$

$$S = (B+W)/2$$

M(x) increases as x moves from W to B and from W to G. Thus, it is feasible that M(x) takes on greater values at points away from W on the line between B and G. Thus, a simplex that is obtained by replacing W by R may be closer to the maximum of M(x). If the maximum lies far beyond R, determined by testing if R is greater than W, the simplex may be extended by replacing W by E. If the maximum lies between W and R, determined by testing if R and W are of equal value, then the simplex may be contracted by replacing W by C. If the function at point C is not greater than the value at W, then the simplex contains the maximum, and can be shrunk by replacing W with S and G with M. This technique converges when B and W are the same within some bounds. Downhill Simplex Minimization requires no symbolic manipulation and computationally compact.

The numerical methods described above do not guarantee convergence to a global maximum. However, since the order of M(x) is known, an upper limit on the number of local minima and maxima can be determined. Iterative determination of extrema can be concluded when this limit is reached.

6.3.3 Interface Server

The interface server generates a state machine that maps a component's protocol onto the standard protocol. This process is initiated by remote method invocation on the client that specifies a key indicating the component for which an interface is desired and the bus to which an interface is desired. The key is used to lookup the HDL or Boolean model in the vendor and a state machine for protocol conversion is synthesized for the appropriate bus.

By generating interfaces to a standard interface, the interface server does not require knowledge of a remote component with which the local component will communicate. The output thread returns an HDL model of the state machine for protocol conversion.

6.4 Summary

POLYSYS is a distributed system for mapping a specification, provided as control and arithmetic operations, to complex components. A system is specified locally by the designer while the components to which it is mapped are maintained remotely at vendor sites. When a viable mapping is determined, an interface between communicating components in the mapping is synthesized. This system implements polynomial methods to perform component matching efficiently. This allows component vendors to publish the functionality of their blocks without disclosing implementation details. Furthermore, it allows designers to determine potential system implementations and characteristics without integrating the libraries of many vendors. In addition, by converting component protocols to the standard protocol of Chapter 5, and implementing the standard architecture, interface synthesis can be performed on vendor sites. This removes the necessity for vendors to disclose interface implementation details prior to component purchase.

Chapter 7
Case Study: Rasterizer

7.1 Overview of Antialiasing Line Rasterizer

Rasterization is a common hardware operation and is basic to all 2-dimensional and 3-dimensional graphics processors perform. It is the task of drawing a line between two points on a screen [FoVaFe90]. However, points on a screen are discrete regions, and a line can only be drawn by illuminating these regions. As a result, simple line rasterization yields a line with an undesirable visual effect known as *staircasing*. Antialiasing is a technique by which this visual effect is reduced or eliminated. This can be achieved by partially illuminating pixels that are immediately adjacent to the line. This technique yields lines that appear to smooth and continuous to the human eye (Figure 27).

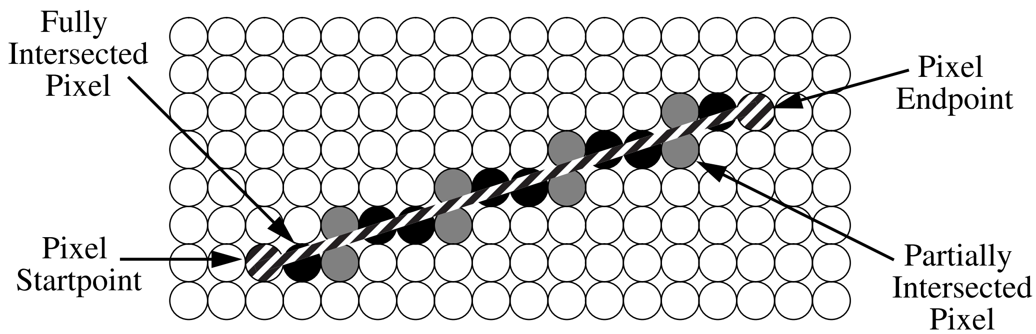


Fig. 27 Results of drawing an antialiased line between two circular pixels on a screen.

7.1.1 Specification

The methodology proposed here utilizes an HDL specification of the target system in which the synthesizable 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. The following Verilog HDL 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 synthesizable Verilog operators and statements, such as +, -, *, and “if”, to allow arithmetic specification of complex blocks.

Latency, timing, area, and precision constraints, can be applied to elements of the specification through pragma statements. Additional operation specific constraints can also be applied to design subblocks. For example, consider the following constraints on the computation of *invDenom* in the specification of Antialias:

```
// pragma begin latency = 4, period = 10, precision = 1
invDenom = 1/(2*((dx^^2 + dy^^2)^^(1/2)));
// pragma end
```

The pragma block constrains the circuitry within the block to complete in no more than four 100 MHz clock cycles, and the result must match the specification to within 1 bit.

Local and global system parameters, as well as objective functions for design optimization, may be defined within pragma statements. For example, if part of a specification is to be mapped to a .25 micron process, and optimized for area, then the begin end block may contain the local parameter and objective function definition:

```
// pragma begin lamba = .25, objective = area
```

If the entire design is to be mapped to a .25 micron process, the specification should contain the global parameter definition:

```
// pragma global lamba = .25
```

Local parameter and objective function definitions supercede global parameter definitions.

7.1.2 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 8 equations: (1) polynomial functionality, (2) domain over which each functionality equation is valid, (3) latency, (4) operational frequency, (5) area, (6) precision, (7) input word width, and (8) 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 follows:

```
Multiplier8 {
     domain0 = 1;
     function0 = xy;
     period = 10*lamba;
     latency = 3;
     precision = 0;
     area = 1e6*lambda^2;
     input_width = {8, 8};
     output_width = {16};
}
```

The argument *lambda* must be defined within a pragma statement as a global or as a block-specific variable.

Additional parameters may be added to library elements. Constraints can then be added to blocks that contain these user defined parameters. For example, power consumption may be defined within a library element as a function of system operating *frequency* and supply *voltage*. Use of this attribute would then require definitions for the parameters *frequency* and *voltage* within the system specification.

Note that the library may be indexed according to the order of the polynomial representation of the implementation. Thus, when existing designs are allocated to implement a specification, only designs with polynomial representations that are of the same order as the specification need to be considered.

7.2 Traditional Design Flow

A traditional design flow would require detailed specification of several complex blocks. Without performing a mapping to high level blocks, the operations of division and square root would need to be specified at the logic level. Furthermore, complex operations such as combinations of multiplication and addition would likely need to be specified independently to achieve good synthesis results.

Mapping the specification of the antialiased line rasterizer to logic gates is a time consuming process, as decomposition to the logic level and the resulting optimization is extremely complex for division, multiplication, addition, and square root. With fully specified division and square root operations, synthesis using design compiler required nearly one hour to complete. Furthermore, sharable resources, such as the structures used to compute *2*dx*invDenom* and *twoVdx*invDenom* are not detectable at the logic level.

In synthesizing at the logic level, a netlist is produced with 14,064 gate equivalents. No knowledge of the regularity of structures within the block, such as the multipliers, dividers, and root structures is available. In addition, since high level resource sharing could not be performed with low level objects, multiple copies of *2*dx*invDenom* and *twoVdx*invDenom* must be placed and routed, further extending design time.

7.3 Design with Reuse

The techniques presented above for allocating reusable designs have been implemented in the POLYSYS high level synthesis tool and are used here to 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 28, from their Verilog implementation.

```
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;
  precision = 0;
  area = 1e6*lambda^2;
  input_width = {8, 8};
  output_width = {16};
}

BasicGates {
  And
  Or
  Nand
  Nor
}
```

Fig. 28 Library elements used to synthesize an antialiased line rasterizer

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*}.

```
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;
```

Fig. 29 Matched partitions of the design *Antialias*

The partition {*incrE = 2*dy*} is matched to 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^^2 + dy^^2)^^(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^^2 + dy^^2)^^(1/2)))_ reveals a match to Divider8 under the conditions x = _2*((dx^^2 + dy^^2)^^(1/2)))_. This statement is matched to Multiplier8 under the conditions _x = 2_ and _y = (dx^^2 + dy^^2)^^(1/2))_. The latter condition is not in polynomial form, requiring Taylor expansion about _dx^^2 + dy^^2_. The resulting polynomial matches SquareRoot8 under the conditions _x = dx^^2 + dy^^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 atomic partitions that are bound are those for computing _I_, _Iinc_, and _Idec_.

In attempting to bind non-atomic partitions, such as {_dx = x2 - x1_, _dy = y2 - y1_}, no elements are found. Thus, the existing allocation is retained, and 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 30. In this case, since components do not employ

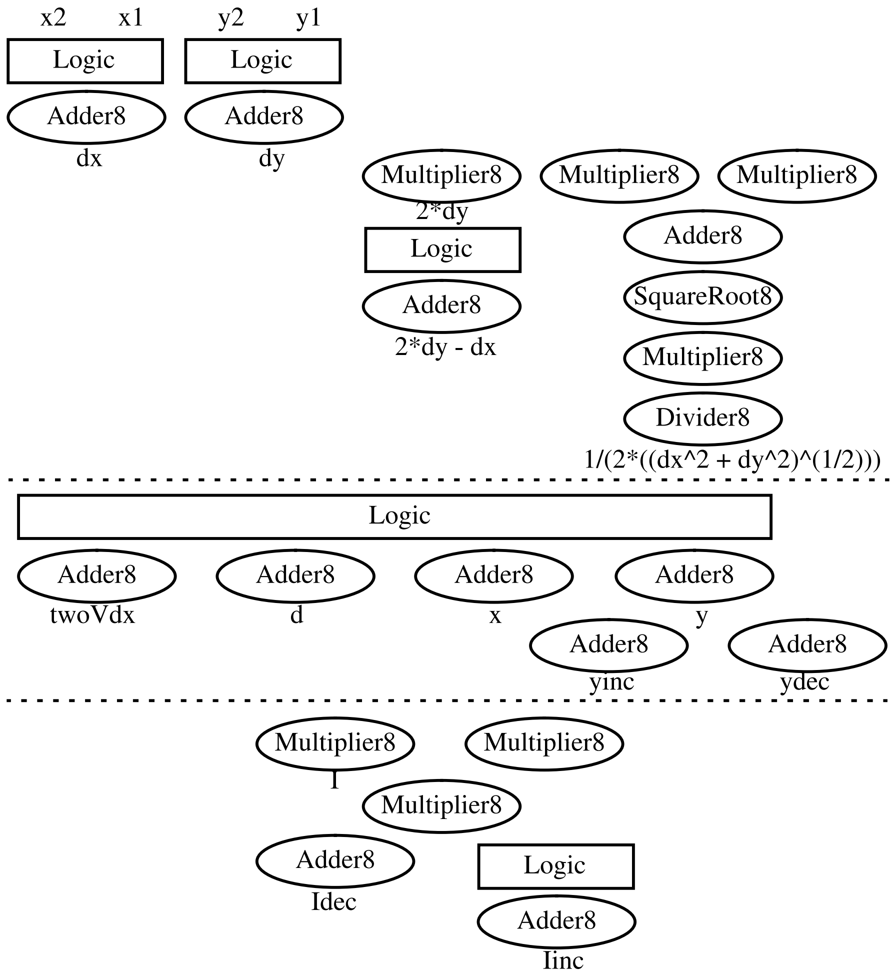


Fig. 30 Antialiased line rasterizer mapped to reusable designs and optimized through scheduling and resource sharing.

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 31.

Library	LSI LCB007
Gate Count	12851
Max. Operating Frequency	25MHz
Est. Area	8.85mm^2
Est. Power Consumption	3.35W

Fig. 31 Physical characteristics of antialiased line rasterizer synthesized from reusable blocks.

The rasterizer synthesized using reusable soft elements achieved approximately the same frequency performance metrics with slightly lower area as compared to a traditional design methodology. The area difference was due primarily to sharing of complex mathematical operations between assignments. However, since the complex library elements SquareRoot8 and Division8 did not have to be redesigned, the number of lines required to specify the design was reduced from 159 to 51.

As compared to an RTL methodology, polynomial methods require significantly less effort in design specification. In addition, since mapping is performed at the arithmetic level, rather than the gate-level, design tasks such as scheduling and resource sharing can be performed at a higher level. In the case of the rasterizer, this resulted in a smaller design than that produced by an RTL methodology. Furthermore, in using polynomial methods to perform component matching rather than symbolic techniques, we are able to build and represent complex operations, such as that describing *invDenom*, that have no common symbolic representation. Polynomial representations provide a formalism for canonically describing and efficiently allocating the components that implement an antialiasing line rasterizer.

Chapter 8
Conclusion

8.1 Summary of Contributions

As transistor sizes shrink by a factor of two every 12 to 18 months, the struggle to maintain reasonable design cycles becomes ever more difficult. Incremental improvements in CAD algorithms and improved computing power provide only temporary relief. New CAD methodologies are required to stimulate discontinuous reductions in design cycles. Design reuse promises to spark this reduction as it is a paradigm change that can affect all stages of the design cycle: specification, synthesis, layout, and verification. However, this paradigm change requires new algorithms for automating stages of the design cycle.

In automating design reuse, the synthesis stage is radically transformed, as vendor libraries no longer contain logic blocks that can be represented compactly and canonically with existing data structures. Canonical decision diagrams and graph based structures tend to be exponentially large for arithmetic functions or control functions. Classification techniques, on the other hand, are compact, but are not canonical, as terminology varies between vendors and relevant component functionality is not described.

Polynomial methods provide a means for compactly and canonically representing circuits that implement arithmetic and control functionality. This is achieved by partitioning circuits into arithmetic and control sub-blocks based on the order of the polynomial that represents circuit functionality exactly. For low order partitions, the polynomial itself is an efficient representation of the circuit. For high order partitions, traditional structures are effective for representing circuit functionality. Polynomial methods allow a specification to be compared against potential implementations by computing the numerical distance between the two. This not only enables fast allocation of exact implementations, but also allows for detection of approximate and partial implementations.

Connection of allocated library elements in current synthesis algorithms is achieved simply by instantiating a wire between the appropriate ports on communicating elements. However, when allocating complex blocks, this mechanism is no longer valid, as these blocks may implement vastly different communication paradigms. To solve this problem, standardized interfaces have been encouraged. However, this approach ignores the body of components that already exist and requires cooperation among design teams who are often in competition with one another. Other techniques enable the generation of glue logic between interfaces that have similar communication paradigms, but break down when two dissimilar components attempt to communicate. Dissimilar components can be connected by designing interfaces using high-level interface specification languages, but this requires additional user intervention to generate this specification.

In order to enable this communication, we have presented a mechanism for transforming a component's communication protocol into a standard protocol and architecture. This technique introduces minimal logic overhead and rarely affects the frequency of operation of the system. In addition, the algorithm we have presented allows for generation of multi-way interfaces and interfaces between synchronous components that operate at different frequencies.

The techniques we have presented have been implemented in the POLYSYS synthesis suite. This software allows vendors to create polynomial models of the components they have developed. As a result, they are able to publish component functionality while hiding the implementation details that are the intellectual property. Designers, in turn, are able to locally create system specifications, map them to elements in component libraries that are located at vendor sites, and generate interfaces between the components. This tool suite has been used to synthesize the DC path of a JPEG encode block, a filter for controlling the speed of a tape drive, and a Bresenham antialiasing line rasterizer.

8.2 Future Directions

An important part of matching a specification to existing components is determining which parts of the specification are likely to have been implemented as individual components. In POLYSYS, this is performed based on the semantic characteristics of the specification such as partitioning directives, individual module declarations, and “begin...end” blocks. It is further based on the ratio of inputs to outputs within a group of assignments. Partitioning may be better performed using knowledge of the component library. For example, elements in the library have an upper limit on the number of domains used in their polynomial representation and an upper limit on the order of the polynomial representation. These limits can be as a decision metric in selecting specification partitions.

In representing circuit functionality as polynomials, detection of partial implementations is possible. In the simplest case, this can be achieved simply by subtracting or dividing the polynomial representation of a component from that of the specification. In addition, constant substitution also provides a simple means of transforming a library element to generate a match to a specification. However these simple operations do not detect all possible partial implementations of a specification. Improvements on these techniques can allow the determination of combinations of library elements that result in a valid implementation of a specification.

In the area of interface synthesis, future research will seek to expand the techniques presented here to generate interfaces between software and hardware components in a system that implements memory mapped I/O. The communication protocol for software interfaces is restricted by the instruction set architecture of the microprocessor on which the software is running. Instead of communicating with a hardware subsystem by assigning values to and reading values from ports, the software driver communicates by writing and reading hardware registers. Thus, the hardware/software composition requires the interface state machine generator described above and another layer, similar to [BoDeLi96], that encapsulates port assignments in microprocessor operations.

This research has focused on the problems of component matching and interface synthesis. Significant inroads have been made in solving the specification and optimization stages of high level synthesis. Behavioral HDLs have raised the level of abstraction of system specifications over traditional register transfer level HDLs. Synthesis from C language specifications also promises to raise the level of abstraction at which systems are specified. Incorporation of these, or other specification languages into POLYSYS promises to shorten the design cycle. Similarly, in the optimization stage, algorithms for performing scheduling, resource sharing, and estimation of system performance promise to return more effective results when selecting components from the space of valid implementations.

The work presented here has focused on synthesis of hardware systems. However, algorithms implemented in software can similarly be partitioned into arithmetic operations and control operations. This provides the potential to map a system specification not only to hardware components, but also to software components. Such a mapping would also require generation of interfaces between hardware and software blocks. This problem can potentially be solved by abstracting the port assignments generated in computing the state machine for protocol conversion into microprocessor operations.

Appendix A - Verilog Model of MIPS CPU Interface

```
module r4600_interface (SysADi, SysADCi, SysCmdi, SysCmdPi,
     SysADo, SysADCo, SysCmdo, SysCmdPo,
     RdRdy, WrRdy, ExtRqst, Release, ValidIn, ValidOut,
     Reset, Clock, SysOe,
     CPUwr, CPUrd, CPUrsp, CPUdataI,
     CPUdataO, CPUack, CPUvalid, CPUaddr);

input [63:0] SysADi;
input [7:0] SysADCi;
input [8:0] SysCmdi;
input SysCmdPi;
output [63:0] SysADo;
output [7:0] SysADCo;
output [8:0] SysCmdo;
output SysCmdPo;
input RdRdy, WrRdy;
input ExtRqst;
output Release;
input ValidIn;
output ValidOut;
input Reset;
input Clock;
output SysOe;
input CPUwr;
input CPUrd;
input CPUrsp;
input [63:0] CPUdataI;
output [63:0] CPUdataO;
output CPUack;
output CPUvalid;
input [63:0] CPUaddr;
reg [63:0] SysADo;
```

```
reg [7:0] SysADCo;
reg [8:0] SysCmdo;
reg SysCmdPo;
reg Release;
reg ValidOut;
reg SysOe;
reg [63:0] CPUdataO;
reg CPUack;
reg CPUvalid;
reg [3:0] bus_state;

parameter S_SEND_IDLE = 4'b0000, S_SEND_ADDR = 4'b0001,
    S_SEND_DATA = 4'b0010, S_SEND_SPIN = 4'b0011,
    S_RECV = 4'b0100, S_RECV_SPIN = 4'b0111;

parameter H_READ = 2'b00, H_WRITE = 2'b01, H_NULL = 2'b10;

always @ (posedge Clock) begin
    ValidOut = 0;
    Release = 0;
    SysOe = 1;
    CPUack = 0;
    if (Reset) begin
        bus_state = S_SEND_IDLE;
    end else begin
        case (bus_state)
        S_SEND_IDLE: begin
            if ((CPUwr && WrRdy) || (CPUrd && RdRdy)) begin
                bus_state = S_SEND_ADDR;
            end else if (ExtRqst) begin
                Release = 1;
                bus_state = S_RECV_SPIN;
                SysOe = 0;
            end
```

```
        end
    S_SEND_ADDR: begin          /* label ADDRESS */
        if (CPUwr) begin
            SysCmdo = 9'b001000000;
            bus_state = S_SEND_DATA;
        end else if (CPUrd) begin
            SysCmdo = 9'b000000000;
            bus_state = S_SEND_IDLE;
        end
    end
    S_SEND_DATA: begin          /* label DATA1 */
        SysADo = CPUdataI;
        if (!CPUwr) begin
            SysCmdo = 9'b100000000;
            bus_state = S_SEND_IDLE;
        end else begin
            SysCmdo = 9'b110000000;
        end
        ValidOut = 1;
        CPUack = 1;
    end
    S_SEND_SPIN: begin
        SysOe = 0;
        bus_state = S_SEND_IDLE;
    end
    S_RECV: begin               /* label DATA0 */
        SysOe = 0;
        if (ValidIn) begin
            if (SysCmdi[8]) begin
                CPUdataO = SysADi;
            end else case (SysCmdi[6:5])
                H_NULL:
                    bus_state = S_SEND_SPIN;
                endcase
```

```
                end
            end
            S_RECV_SPIN: begin
                SysOe = 0;
                bus_state = S_RECV;
            end
            endcase
        end
    end
    endmodule;
```

Appendix B - List of Acronyms

AC - Alternating Current

ALU - Arithmetic Logic Unit

ADD - Algebraic Decision Diagram

BDD - Binary Decision Diagram

BMD - Binary Moment Diagram

*BMD - Multiplicative Binary Moment Diagram

CDC - Controllability Don't Care

CFE - Control Flow Expression

CPU - Central Processing Unit

DC - Direct Current

DCT - Discrete Cosine Transform

DRAM - Dynamic Random Access Memory

EISA - Extended Industry Standard Architecture

FFT - Fast Fourier Transform

FSM - Finite State Machine

HDD - Hybrid Decision Diagram

HDL - Hardware Design Language

IIR - Infinite Impulse Response

IP - Intellectual Property

I/O - Input/Output

JPEG - Joint Photographic Experts Group

MATLAB - Mathematics Laboratory

MTBDD - Multi-Terminal Binary Decision Diagram

ODC - Observability Don't Care

PCI - Peripheral Component Interconnect

PFG - Protocol Flow Graph

PHDD - Power Hybrid Decision Diagram

RTL - Register Transfer Level

SEQ - Signal Sequences

VME - Versa Module Eurocard

VSIA - Virtual Socket Interface Alliance

ZBDD - Zero-suppressed Binary Decision Diagram

References

[AmBo91] T. Amon and G. Borriello, "Sizing Synchronization Queues: A Case Study in Higher Level Synthesis", DAC91: *Proceedings of the 28th Design Automation Conference*, pp. 690-693, 1991.

[BaFrGa93] R.I. Bahar, E.A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and their Applications," ICCAD93: *Proceedings of International Conference on Computer Aided Design*, pp. 188-191, 1993.

[BaRo99] C. Barna and W. Rosenstiel, "Object Oriented Reuse Methodology for VHDL", DATE99: *Proceedings of the Conference of Design Automation and Test in Europe*, pp. 689-693, 1999.

[BoDeLi96] I. Bolsens, H. DeMan, B. Lin, K. Van Rompaey, S. Vercauteren, D. Verkest, "Hardware-Software Codesign of Digital Telelcommunication Systems", *IEEE Transactions on Computer Aided Design*, pp. 391-418, March 1997.

[BoKr99] R.K. Bock and W. Krischer, *The Data Analysis Brief Book*, March 1999.

[Br86] R. Bryant "Graph Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, C-35(8), pp. 677-691, 1986.

[Br92] R. Bryant "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams", *ACM Computing Surveys*, Vol. 24, No. 3, pp. 293-318, September 1992.

[BrCh95] R. Bryant and Y.A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams", DAC95: *Proceedings of the 32nd Design Automation*

Conference, pp. 535 - 541, 1995.

[BrRuBr90] K. Brace, R. Rudell, and R.Bryant, "Efficient Implementation of a BDD Package", DAC90: *Proceedings of the 27th Design Automation Conference*, pp. 40-45, 1990.

[ChBr96] Y.A. Chen and R. Bryant, "ACV: An Arithmetic Circuit Verifier", ICCAD96: *Proceedings of the International Conference on Computer Aided Design*, pp. 361-365, 1996.

[ChBr97] Y.A. Chen and R. Bryant, "*PHDD: An Efficient Graph Representation for Floating Point Circuit Verification", ICCAD97: *Proceedings of the International Conference on Computer Aided Design*, pp. 2-7, 1997.

[ChOrBo95] P. Chou, R. B. Ortega, G. Borriello, "Interface Co-Synthesis Techniques for Embedded Systems", ICCAD95: *Proceedings of the International Conference on Computer-Aided Design*, pp. 280-287, 1995.

[ClFu93] E.M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transformations for Large Boolean Functions with Applications to Technology Mapping", DAC93: *Proceedings of the 30th Design Automation Conference*, pp. 54-60, 1993.

[ClFu95] E.M. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams", ICCAD95: *Proceedings of the International Conference on Computer Aided Design*, pp. 159 - 163, 1995.

[CoDe94] C. Coelho and G. De Micheli, ``Dynamic Scheduling and Synchronization Synthesis of Concurrent Digital Systems Under System-Level Constraints'', ICCAD94: *Proceedings of the International Conference on Computer-Aided Design*, pp. 175-181, 1994.

[Da98] *Dataquest report on the Electronic Design Automation Industry*, December, 1998.

[DaBoBe99] M. Dalpasso, A. Bogliolo, and L. Benini, “Specification and Validation of Distributed IP-Based Designs with JavaCAD”, DATE99: *Proceedings of the Conference of Design Automation and Test in Europe*, pp. 684-688, 1999.

[De94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, pp. 458-462, 1994.

[ErHeBe93] R. Ernst, J. Henckel, and T. Benner, “Hardware/Software Cosynthesis for Microcontrollers”, *IEEE Design and Test of Computers*, pp. 64-75, December 1993.

[FoVaFe90] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice*, pp. 132-141, 1990.

[GaGl96] M. Gasteier, M. Glesner, “Bus-Based Communication Synthesis on System-Level”, ISSS96: *Proceedings of the 9th International Symposium on System Synthesis*, pp. 65-70, 1996.

[GaVaNa94] D. Gajski, F. Vahid, and S. Narayan, “A System Design Methodology: Executable-Specification Refinement”, EDTC94: *Proceedings of the European Design and Test Conference*, pp. 458-463, 1994.

[GaZa79] C.B. Garcia and W.I. Zangwill, “Finding All Solutions to Polynomial Systems and Other Systems of Equations”, *Mathematical Programming*, Vol. 16, pp. 159-176, 1979.

[GuRo94] P. Gutberlet, W. Rosenstiel, “Specification of Interface Components for

Synchronous Data Paths”, ISSS94: *Proceedings of the 7th International Symposium on System Synthesis*, pp. 134-139, 1994.

[HiPe74] F. Hill and G. Peterson, *Introduction to Switching Theory and Logical Design*, 1974.

[JaElOb94] A. Jantsch, P. Ellervee, J. Oberg, A. Hemani, and H. Tenhunen, “Hardware/ Software Partitioning and Minimizing Memory Interface Traffic”, EuroDAC94: *Proceedings of European Conference on Design Automation,*, pp. 226-231, 1994.

[KaLe94] A. Kalavade and E.A. Lee, “A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem”, *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, pp. 42-48, 1994.

[KhTo86] A.W. Al-Khafaji and J. Tooley, *Numerical Methods in Engineering Practice*, pp. 281-293, 1986.

[KiGeVe83] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, ”Optimization by Simulated Annealing”, *Science*, pp. 671-680, May 1983.

[KnMa98] P.V. Knudsen and J. Madsen, “Integrating Communication Protocol Selection with Partitioning in Hardware/Software Codesign, ISSS98: *Proceedings of the 11th International Symposium on System Synthesis*, pp. 111-116, 1998.

[KuAyJo94] S. Kumar, J. Aylor, B. Johnson, and W. Wulf, “Object-Oriented Techniques in Hardware Design”, *IEEE Computer*, pp. 64-70, June 1994.

[Ma92] J. Mathews, *Numerical Methods for Mathematics, Science, and Engineering, Second Edition*, pp. 400-413, 1992.

[Ma98] G. Martin, “Design Methodologies for System Level IP”, DATE98: *Proceedings of the Conference of Design Automation and Test in Europe*, pp. 286-289, 1998.

[MaHa95] J. Madsen and B. Hald, “An Approach to Interface Synthesis”, ISSS95: *Proceedings of the 8th International Symposium on System Synthesis*, pp. 16-21, 1995.

[Mi93] S. Minato, “Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems”, DAC93: *Proceedings of the 30th Design Automation Conference*, pp. 272-277, 1993.

[Mi96] S. Minato, “Implicit manipulation of Polynomials Using Zero-Suppressed BDDs”, EDTC96: *Proceedings of the European Design and Test Conference*, pp. 449-454, 1996.

[Mo98] B. Mourrain, “An Introduction to Linear Algebra Methods for Solving Polynomial Equations”, HERCMA98: *Proceedings of the 4th Hellenic European Conference on Computer Mathematics & Applications*, pp. 1-22, 1998.

[NeMe65] J.A. Nelder, R. Mead, “A Simplex Method for Function Minimization”, *Computer Journal*, Vol. 7, pp. 308-324, 1965.

[ObKuHe96] J. Oberg, A. Kumar, and A. Hemani, “Grammar-based Hardware Synthesis of Data Communication Protocols”, ISSS96: *Proceedings of the 9th International Symposium on System Synthesis*, pp. 14-19, 1996.

[PaRoSa98] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli, “Automatic Synthesis of Interfaces between Incompatible Protocols”, DAC98: *Proceedings of the 35th Design Automation Conference*, pp. 8-13, 1998.

[RaMc98] K. Ravi, K. McMillan, T. Shiple, F. Somenzi, “Approximation and Decomposition of Binary Decision Diagrams”, DAC98: *Proceedings of the 35th Design Automation Conference*, pp. 445-450, 1998.

[RoVi97] J. Rowson, A. Sangiovanni-Vincentelli, “Interface-Based Design”, DAC97: *Proceedings of the 34th Design Automation Conference,* pp. 178-183, 1997.

[Se99] R. Seepold, “IP and Reuse”, DATE99: *Proceedings of the Conference of Design Automation and Test in Europe*, p. 725, 1999.

[SeHoMe96] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck, “A System for Compiling and Debugging Structured Data Processing Controllers”, EuroDAC96: *Proceedings of European Conference on Design Automation*, pp. 86-91, 1996.

[SIA97] Semiconductor Industry Association, *National Technology Roadmap for Semiconductor*s, 1997.

[SmDe98] J. Smith and G. De Micheli, “Polynomial Methods for Component Matching and Verification”, ICCAD98: *Proceedings of the International Conference on Computer Aided Design*, pp. 678-685, 1998.

[SpNe97] M. Spiller and R. Newton, “EDA and the Network”, ICCAD97: *Proceedings of the International Conference on Computer Aided Design*, pp. 470-476, 1997.

[VaGi98] F. Vahid and T. Givargis, “Incorporating Cores into System-Level Specification”, ISSS98: *Proceedings of the 11th International Symposium on System Synthesis*, pp. 43-48, 1998.

[VSIA99] “The VSI Alliance”, http://www.vsia.com/, 1999.