System-Level Design Techniques And Tools For Synthesis Of Application-Specific Digital Systems

Chih-Tung Chen

CENG Technical Report 94-17

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4476

August 1994

SYSTEM-LEVEL DESIGN TECHNIQUES AND TOOLS FOR SYNTHESIS OF APPLICATION-SPECIFIC DIGITAL SYSTEMS

by

Chih-Tung Chen

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

August 1994

Copyright 1994 Chih-Tung Chen

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90007

This dissertation, written by

.............Chih-Tung Chen.....................................

under the direction of his........ *Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of*

DOCTOR OF PHILOSOPHY

....
Dean of Graduate Studies

Date ....

DISSERTATION COMMITTEE

Alice C. Parker
Chairperson

[illegible]

[illegible]

Dedication

To my wife, my parents and my son.

Acknowledgements

I take this opportunity to express my sincere appreciation to my advisor, Professor Alice Parker, for her guidance and support through the years. She was always there to help me with research and personal problems and to motivate me when I was discouraged. It is my fortune to have known her.

I would like to thank Professors Ming-Deh Huang and Sandeep Gupta for serving on my dissertation committee, and Professors Melvin Breuer, Michel Dubois, and Dennis Mcleod for serving on my guidance committee. Their comments and suggestions were most valuable.

I also thank all my friends and colleagues at USC, past and present, for their friendship and various help. In particular, I like to mention Yung-Hua Hung, Jen-Pin Weng, Sen-Pin Lin, Pravil Gupta, Charles Njinda, Diogenes Silva Jr., Atul Ahuja, Shiv Prakash and Kayhan Kucukcakar.

I especially thank my parents for their continuous love and support. They give me their complete trust and accept so naturally that their son is still a student for countless years. To my beloved child, Phillip, your interruption made the time more enjoyable and kept my goal in perspective.

Finally, I would like to thank my wife, Shu-Mei, for her patience, support and sacrifices throughout my graduate study. She has given up her most precious years in her life to give her husband a chance to achieve his selfish goal. I gratefully dedicate this thesis to her.

I would like to acknowledge the financial support from the Advanced Research Projects Agency under Contract No. J-FBI-90092, the Department of Air

Force, the Department of Army and the Department of Navy under Contract No. N00039-87-C-0194, and the National Science Foundation under Contract No. GER-9023979.

Contents

List Of Figures

List Of Tables

Abstract

Previously, most behavioral synthesis tools emphasized the synthesis of single chip and single process designs. However, application-specific digital systems today are usually too large to fit into a single chip and consist of multiple concurrent processes. This thesis addresses two important issues for the design of such systems; namely, system partitioning and synthesis of concurrent processes. The system partitioning methodology presented here is aimed at partitioning the system at the process level onto multiple chips while considering chip packaging options as well as the potential process design alternatives in the system. Synthesis techniques for concurrent processes are also introduced so that not only the performance and area constraints of each individual process can be met but also the communication among the processes can be synchronized.

In this thesis, we also address the issue of design verification. Although synthesized designs are often considered to be correct by construction, in reality there is no such guarantee unless the whole synthesis process, including techniques and programs, can be formally validated. Hence, we developed an efficient verification approach using a hybrid symbolic/numeric simulation to check both the functional and timing correctness of synthesized RTL designs.

Finally, a VHDL to DDS compiler software which transforms the system specification written in the VHDL hardware description language into a synthesizable representation for the ADAM synthesis system is also included here.

Chapter 1

Introduction

1.1 Background

VLSI technology has reached densities of over one million transistors per chip, and VLSI chips are being used in more diverse applications than ever before. At the same time, the life cycles of electronic products are rapidly decreasing. An effective way to deal with the increasing complexity of designs while reducing the design turnaround time is to apply *design automation* on more levels of abstraction at which circuits are designed.

Today, design automation tools for layout generation, placement and routing, module generation, simulation, and logic synthesis have become fairly reliable and widely available. As the market pressure constantly demands higher-level design tools, automation of the entire design process from conceptualization to silicon has become an important goal and an intensive research field in the last decade. The benefits of such a methodology include not only shorter design time, but also ease of modification of the design specifications and the ability to explore different design tradeoffs more effectively.

This thesis addresses the issues that arise in behavioral and system-level synthesis of application-specific digital systems.

1.2 Behavioral Synthesis

Behavioral synthesis, also known as *high-level synthesis*, is a process which takes a behavioral specification of a digital system along with a set of constraints and goals on the resulting hardware and finds a structure that realizes the given behavior while satisfying the given goals and constraints. The behavior is usually described algorithmically in some hardware description language (HDL) such as VHDL [Ins88], ISPS [Bar81], or HardwareC [MK88]. The structure is a register-transfer level (RTL) implementation which includes a data path as well as a control path. The data path is a network of registers, functional units, multiplexers, and buses. The control path, on the other hand, is usually a finite-state machine which drives the data path in order to produce the required behavior.

Due to its complexity, behavioral synthesis is often divided into several distinct yet inter-dependent tasks [MPC88]. First, the behavioral specification must be translated and possibly optimized into an internal representation, generally referred to as a control data flow graph (CDFG), that models both the control and data flow of the input behavior. Next, scheduling, allocation and binding are performed to map the CDFG into structure. *Scheduling* discretizes the execution of operations in the CDFG by assigning them to control steps as shown in Figure 1.1. The *allocation* task determines the number and types of required hardware resources including functional units, storage elements and interconnection paths. Resource *binding* assigns operations and values to specific allocated hardware resources. At this stage, an RTL data-path implementation is produced as shown in Figure 1.2. Once the schedule and the data path have been determined, the required control path can be synthesized to activate components in the data path according to that schedule. Most behavioral synthesis approaches stop at this stage and pass the RTL implementation to the lower level design automation tools for further optimization and the layout generation.

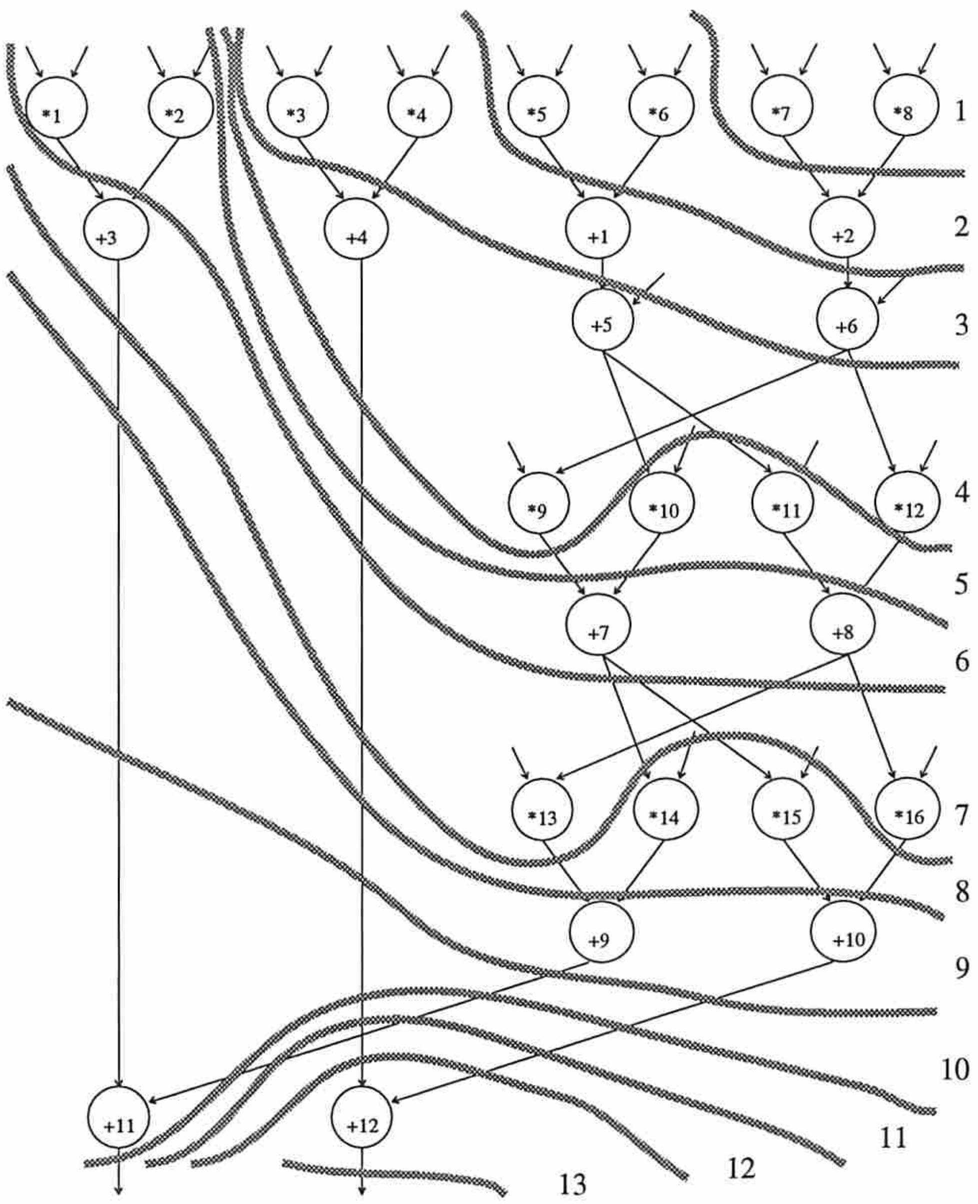


Figure 1.1: A schedule for a CDFG

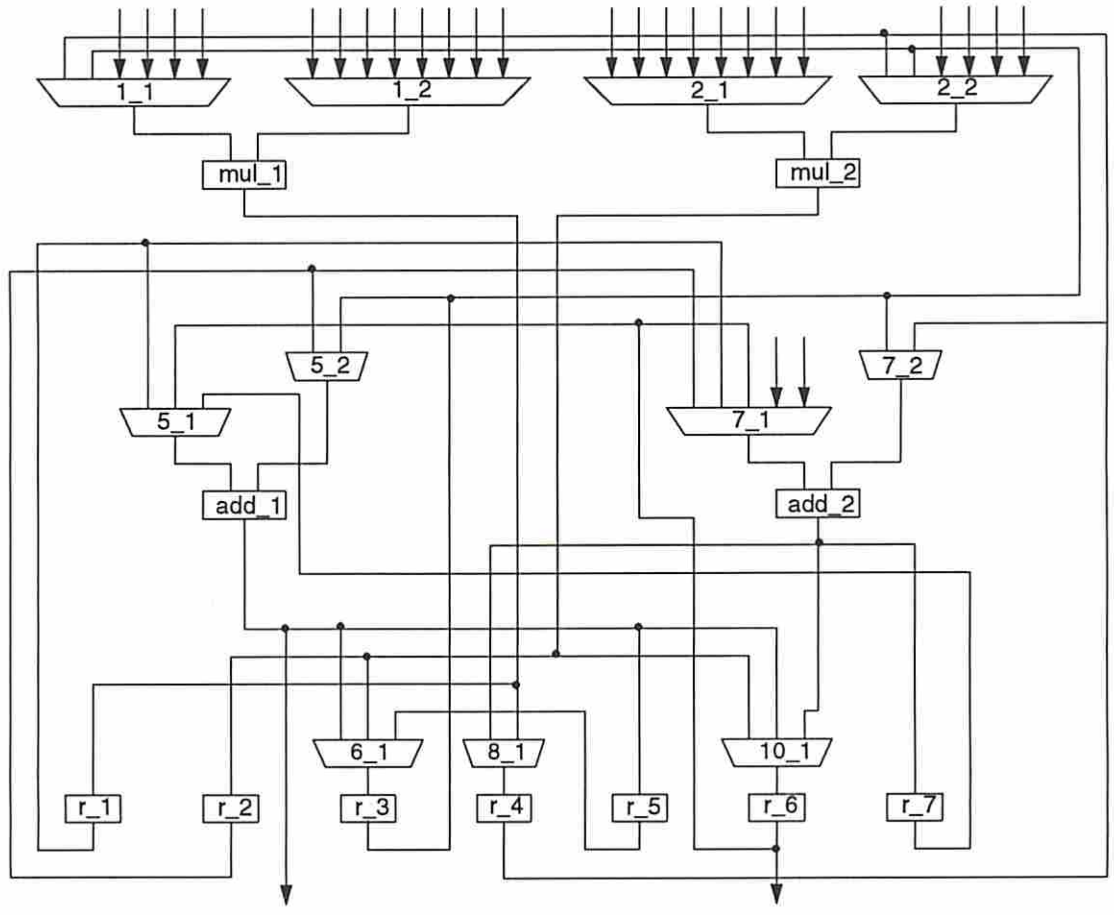


Figure 1.2: An RTL data-path implementation of the behavior in Figure 1.1

1.3 Motivations

Although the sizes of implementable VLSI chips have increased drastically in recent years, most modern digital systems still can hardly fit into a single chip. Traditional behavioral synthesis research has had the objective of producing single-chip VLSI implementations from behavioral specifications. After a design is synthesized as a single-chip design, it may not be possible to partition the design onto multiple chips while satisfying the constraints; especially, timing constraints. Even if a feasible partitioning can be found, it is likely that the multi-chip implementation would be inferior since all the synthesis decisions and optimization made assume a single-chip implementation as the target design.

Another common characteristics of hardware systems is that they are inherently parallel. In fact, complex application-specific systems often consists of multiple concurrent tasks (processes). For example, three GM production designs illustrated by Fuhrman [Fuh91] contain from 4 to 10 concurrent processes. A typical JPEG image compression system consists of 6 major tasks; namely, forward/inverse discrete cosine transform, quantization/dequantization and encoding/decoding.

Most existing behavioral synthesis approaches primarily address synthesis of a single component (process) in a system without considering its system-level interaction. Using this synthesis paradigm for multiple-process systems, each process must be synthesized separately, and the integration and synchronization of the processes usually have to be done manually by the designers. However, if processes are synthesized one at a time, the decisions made previously may affect and constrain the synthesis of other processes in the system, which is also likely to result in an inferior system implementation. Furthermore, these synthesis approaches are limited in their ability to handle I/O and communication requirements, with a few exceptions [Nes87, Hay90, KM92]. In fact, many of them only consider the overall latency between the inputs and outputs of synthesized designs. Hence, it may become very difficult for the designers to integrate the processes in the system after their implementation have been synthesized without taking into account the synchronization issue.

In order to design complex ASIC systems effectively with acceptable design time and quality using design automation tools, we identify several important *system-level issues* to be addressed and incorporated into the current behavioral synthesis paradigm.

- *Partitioning.* As the design prediction techniques become more advanced and accurate [KP93], it is now feasible and preferable to partition a system onto multiple chips before the behavioral synthesis process. Partitioning before synthesis allows the design space to be explored in a system-level view. Several system-level tradeoffs, such as chip count, chip packaging, the performance and resource requirements of data processing and data transfers, can be taken into account during partitioning. Consequently, proper directions can be provided to the subsequent synthesis process and the synthesized multi-chip system will likely to be feasible with respect to the given constraints.

- *Synthesis.* To design a system such that its processes can coordinate their actions flawlessly, the synthesis approach requires simultaneously solving all the timing and synchronization constraints imposed by one process on another. Also, if the time when an external synchronization will occur is not known *a priori*, the delay of the corresponding operation becomes *data dependent*; hence, synthesis algorithms can no longer assume that all operations have fixed delays. Furthermore, we need to budget the chip resources allocated to each process on a chip since the total resources taken by the processes on a chip are limited by the chip package to be used.

- *Verification.* A common approach to avoiding costly design iterations is to find the design problems as early as possible. Although synthesized designs are often argued to be *correct by construction*, in reality there is no such guarantee unless the whole synthesis process, including techniques and softwares, can be formally validated. As the synthesis tools advance to higher levels of abstraction, they become even more sophisticated and may still be under constant evolution. Also, they often require complex interaction with other tools and/or the designers. Hence, the chances for errors are greatly increased, and an effective verification methodology to cross-check the synthesized results becomes highly desirable.

With a better understanding of these system-level issues, an effective *system design* methodology can be formed to increase the design quality and to reduce the design cycles.

1.4 Problem Approach

This research focuses on a system design methodology for synthesizing multi-chip systems with multiple concurrent processes as shown in Figure 1.3. In this figure, the specific problems addressed in this thesis are highlighted by bold-line boxes.

The first task is to compile the system specification written in VHDL into a unified design representation called the Design Data Structure (DDS) [KP85]. At this step, the behavior of each process is transformed into a pair of data-flow and control-flow graphs so that the parallelism is extracted and the representation is ready for synthesis. In addition, the inter-process communication is extracted and represented by the DDS bindings in a manner to be described in Section 3.2.2.

Next, several process behavioral transformations could be evaluated to trade off among hardware sharing, communication overhead, and control complexity. For example, we could *collapse* two processes into one so that the the inter-process

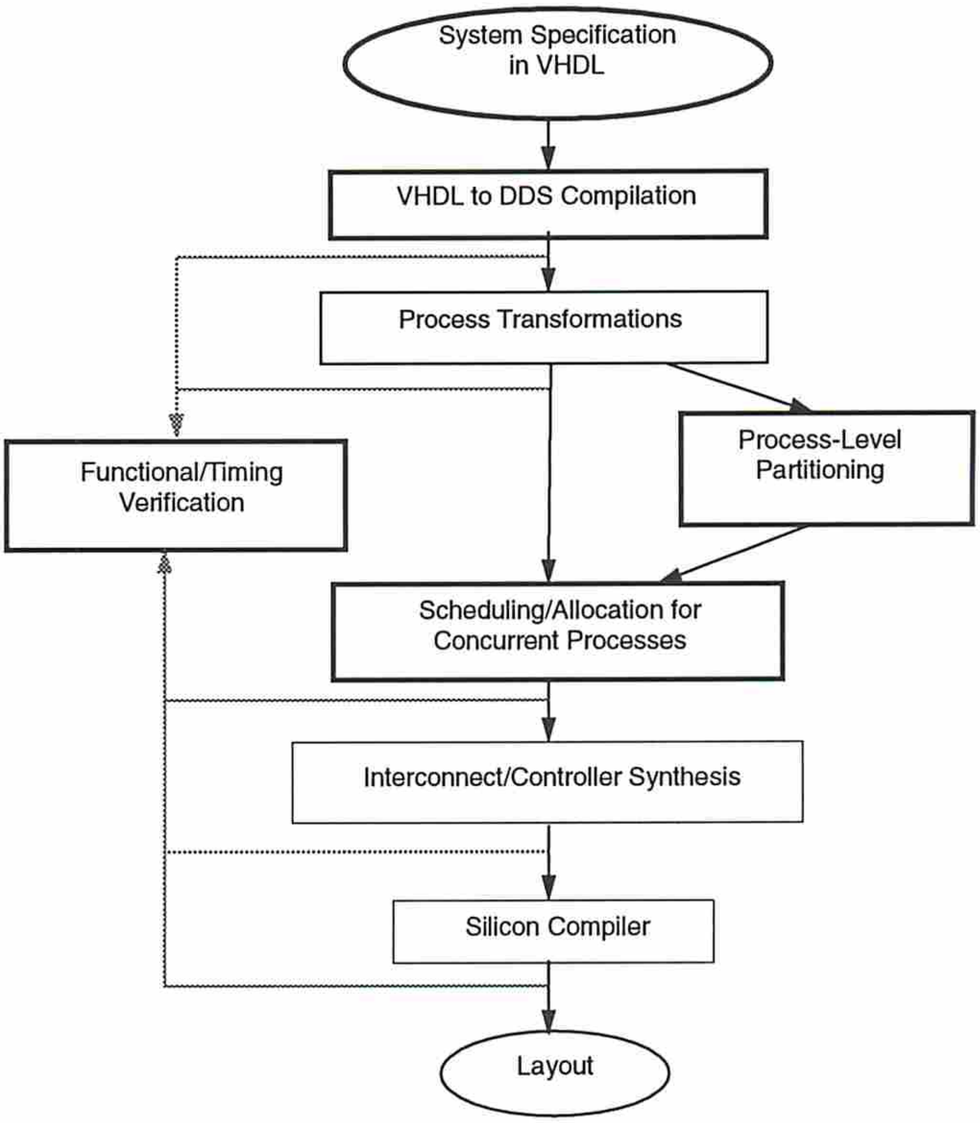

System Specification in VHDL
VHDL to DDS Compilation
Process Transformations
Process-Level Partitioning
Functional/Timing Verification
Scheduling/Allocation for Concurrent Processes
Interconnect/Controller Synthesis
Silicon Compiler
Layout

Figure 1.3: System synthesis process

communication between them can be replaced by direct data transfers, and the hardware resources can be shared in the merged behavior. On the other hand, if a process is found too big to fit in a single chip while meeting its constraints, its behavior should be *decomposed* into a number of smaller processes. The process transformations essentially try to determine a proper coarse-grain concurrency for the system by resetting the process boundaries.

In this system synthesis methodology, partitioning of the system onto multiple chips is performed at the process level. The advantage of this approach is that the number of objects to be considered during partitioning is small and the functional boundaries specified by the designers are preserved. Also, with the aid of advanced prediction tools, the exploration of process design alternatives can be done concurrently with partitioning. Finally, the chip count and the chip capacities (area and pins) can be traded off according to the available chip packaging options.

After partitioning, a concurrent approach is used to synthesize the processes in the system. In this approach, each process is mapped to its own data path with a single thread of control. The objective is to schedule and allocate each process in such a way that all the timing and synchronization constraints are met and the hardware resources are distributed to processes according their performance requirements. Once the schedule and allocation of each process have been determined, its RTL implementation and layout can be generated independently from other processes using lower level design tools.

Finally, a distinctive feature of this synthesis methodology is the inclusion of a design verification step to ensure the correctness of the RTL implementations on not only with respect to their functional behaviors but also their I/O sequencing and timing. Due to the fact that synthesized implementations are derived from their specifications in a well-defined manner, a hybrid symbolic/numeric approach is developed in this research to perform this verification task formally and yet automatically.

Some potential applications of the system design methodology described here include but are not limited to

- design of cost-effective systems,
- rapid prototyping of complex system designs, and
- partial or full system redesign.

1.5 Thesis Organization

The organization of this thesis is as follows:

Chapter 2 surveys the related work on four major areas of this research: design specification and representation, system partitioning, multiple-process synthesis and design verification.

In Chapter 3, we describes the problem of HDL compilation in general and the translation from VHDL to the DDS representation in particular. The required techniques for control/data flow analysis and flow graph generation/optimization are presented. We will also discuss the modeling of arrays, input/output and inter-process communication in both VHDL and DDS. The implementation of the prototype software VHDL2DDS based on this work has created a VHDL front-end for the ADAM synthesis system and has led to numerous top-down design experiments from behavioral VHDL descriptions to chip layouts.

Chapter 4 describes the research on the system-level partitioning problem. A novel partitioning approach is presented to partition a system at the process level, to explore each process's design alternatives, to determine proper chip count, and to consider chip packaging options concurrently. Two partitioning techniques, an MILP formulation and a genetic-search procedure, will be described. Experimental results, including a JPEG image compression system, using the prototype software ProPart are also given.

Chapter 5 deals with the synthesis of designs with unbounded-delay operations under timing constraints and with multiple communicating processes. The concept of single-threaded processes will be introduced and used as the basis of our synthesis approach. We will also show how to satisfy detailed timing constraints when unbounded-delay operations are present and how to synchronization inter-process communication.

In Chapter 6, we discuss the RTL design verification problem. We will show that though the correctness of synthesized designs cannot be guaranteed, there exists several important properties in synthesized RTL designs which can be used to simply the verification task. An effective verification approach based on a hybrid symbolic/numeric simulation will be introduced. The advantage of this verification approach is that it not only can formally verify the data path but also can faithfully exercise the control path and allow timing issues, such as delays, clocking schemes, and I/O protocols, to be taken into account.

Finally, Chapter 7 summarizes this thesis and outlines future research directions.

Chapter 2

Related Research

In this chapter, a number of related research efforts will be described. They are divided into four major areas; namely, design specification and representation, system partitioning, multiple-process synthesis and design verification. Some related research directly relevant to the research described in this thesis will be described further in the appropriate chapters.

2.1 Design Specification and Representation

In general, a behavioral synthesis system requires a behavioral specification language which has the expressive power to describe all the design behaviors in the targeted domain. In addition, the semantics of the language must be translated into a representation that is suitable for synthesis.

To date, there is still no agreement in the synthesis community on how a behavioral specification language should be designed. In fact, several input languages have been used by current behavioral synthesis systems. For example,

- VHDL [Ins88], used by the ADAM system at USC, the SpecCharts language [VNG91] at UCI, and the System Architect's Workbench (SAW) [TLW+90] at CMU,

- HardwareC, used by Hercules [MK88],
- ISPS [Bar81] and Verilog, used by SAW [TLW+90],
- Silage [Hil85], used by Cathedral [GRVM90].

The selection of a specification language depends on the domain of the targeted designs, the degree of expressivity and the ease of implementation. A survey of several specification languages can be found in [Nar92].

The internal representations used by most behavioral synthesis systems to model the design behaviors are also different in style, but share a common objective which is to capture the essential control and data flow information of the design behavior using flow graphs. Most of these flow-graph representations can be classified into either *disjoint* or *hybrid* control and data flow representations.

The Design Data Structure [KP85] of the ADAM system is a typical disjoint control and data flow representation. The control and data flow information are kept in separate graphs and a set of bindings is used to relate objects in these graphs. Descart [OG86] and VSS [LG88] use a control-flow graph similar to the one used in standard software compilers. Each node in the control-flow graph represent a basic block and the computations within each basic block are mapped to a separate data-flow graph.

The hybrid control and data-flow representation merges both the control and data flow information into one graph. The Value Trace (VT) [McF78] of the CMU-DA system and the Control and Data Flow Graph (CDFG) of the HAL system [PK89] follow this scheme. The SIF representation used in the Olympus system [MKMT90] uses a hierarchical sequencing graph to show both data and control flow dependencies. The DSFG representation [LGP+91] for capturing DSP designs in the Silage language is another hybrid representation in which the signal-flow graph is annotated with control information and design constraints.

The task of compiling design descriptions and producing the flow-graph representation as output is complicated by the fact that the design behavior is described by sequential statements in some hardware description language like VHDL. Therefore, extensive local/global flow analysis and graph optimization are needed to produce fully parallelized flow graphs. Many of the analysis techniques required for this compilation task are similar to those used traditional software compilers [ASU86]. However, additional global data-flow analysis and graph generation steps are needed. Though HDL compilation using these steps had been performed as early as 1978 [McF78], formalization of the problem and detailed description have not been presented to our knowledge.

2.2 System Partitioning

Partitioning plays a key role in the design of digital systems. There are various goals that are commonly achieved during system partitioning. For example, a digital system may be partitioned to reduce the design complexity, to perform concurrent design, and to satisfy physical-capacity constraints. System partitioning approaches can be classified into two categories: *structural partitioning* and *behavioral partitioning.*

In structural partitioning, the system specification (behavior) is first synthesized into structure, and then the structure is partitioned onto chips, modules and boards. Numerous approaches have been proposed in solving circuit partitioning problems at the logic or RT level, such as group migration [KL70, FM82], simulated annealing [CH90, GS84], evolution [SR89], clustering [LLT69], and even interactive partitioning [BKM$^+$66]. While structural partitioning usually can provide a solution that satisfies physical constraints such as area and pins, it ignores the fact that the design of the structure can be heavily influenced by the system partitioning. For example, after the system structure is synthesized as a single-chip design, it may not be possible to partition the design onto multiple chips while satisfying the constraints (especially timing constraints).

Behavioral partitioning is a process of dividing the system behavior onto a number of partitions which can be synthesized into separate hardware modules or chips. There are essentially two levels of granularity at which behavioral partitioning can be performed. Previous approaches have been mostly at the operation level. A detailed survey of these approaches has been done by Vahid [Vah91]. Alternatively, behavioral partitioning can be performed at a higher level of granularity, such as processes, procedures and memory blocks.

In BUD [McF86], McFarland uses a clustering algorithm based on a *similarity* measure to partition control data flow graphs (CDFGs) in a manner that encapsulates scheduling and allocation decisions. A similar approach was proposed by Lagnese and Thomas in APARTY [LT91] by employing multiple stages of clustering with different closeness functions. The partitioning method used in YSC [CvE87] by Camposano and van Eijndhoven clusters logic and operations in the behavioral specification into groups in order to improve the tractability of subsequent logic synthesis. In Vulcan [GM90], Gupta and De Micheli use a hierarchical hypergraph model for repeated two-way partitioning of the largest vertex at each level of the hierarchy until the constraints are met.

CHOP by Küçükçakar [Kuc91] evaluates a partitioned CDFG using a comprehensive behavioral area-delay prediction tool called BEST [KP93] to explore possible alternative implementations of each partition. The tool then searches for a combination of predicted design points for the partitions which will make the synthesized multi-chip design feasible while taking into account the design integration overhead. An interactive partitioning interface and a simple two-way partitioning procedure are also provided.

A system-level behavioral partitioning tool called SpecPart is described by Vahid and Gajski [VG92]. In this work, the objects to be partitioned are elevated to a higher level of abstraction such as processes, procedures, and other code groupings imposed by the specification language. An estimated area and delay is assigned to each object and a group migration technique similar to the Kernighan-Lin algorithm is used for partitioning. Gupta and De Micheli [GM92] also described an approach for partitioning the system specification at the process level into hardware and software components in order to satisfy the performance and cost constraints. In SpecSyn [GGVN93], Gajski *et al.* proposed a system-level design methodology and framework for mapping system functionalities onto a set of mixed-technology modules through a comprehensive designer interface.

2.3 Multiple-Process Synthesis

Though behavioral synthesis of digital hardware has received enormous attention over the years, most approaches[1] have focused on the synthesis of single-process designs. Hence, synthesis of systems with multiple processes is often achieved by synthesizing individual processes separately and then interconnecting their structure [Fuh91]. This kind of approach requires that I/O and inter-process communication be specified manually by the designers.

Hung [HP92] and Gebotys [Geb92] extended traditional behavioral synthesis to multiple-chip designs. Under these approaches, the design behavior is still represented by a single CDFG. The CDFG is first partitioned and then the partitioned CDFG is scheduled globally so that each chip's execution is fully synchronous with the others and the inter-chip communication is achieved completely in lock step.

[1]A comprehensive study of these approaches and related synthesis topics can be found in [MPC88].

Like many previous scheduling techniques, these approaches still assume that all the operations in the CDFG have *fixed* execution delays.

There is a class of work [TW93, Nes87, Hay90] which address the issue of I/O and inter-process communication as a separate problem, known as *interface synthesis.* These techniques, however, are mostly applicable to control-dominated designs with little or no data computation. The approach reported by Takach and Wolf [TW93] first analyzes a network of communicating processes to generate a small set of communication constraints. Solving the set of generated communication constraints and the internal constraints of each process results in a feasible schedule for the network.

For the synthesis of more general designs, Ku introduced a technique called *relative scheduling* [KM92] which can handle operations with unbounded delays under detailed timing constraints. In relative scheduling, the start time of an operation is specified in terms of offsets from a set of *anchors* (unbounded-delay operations). This approach requires resource binding be performed before scheduling. Once resource binding is determined, the operations bound to the same resource must be serialized by adding proper sequencing dependencies among them to resolve conflicts. Furthermore, the control scheme for the hardware architecture targeted by relative scheduling is fairly complicated.

Recently, Ku *et al.* [KFJM92] applied relative scheduling to designs with multiple processes. Like Takach's approach [TW93], the inter-process communication events are first extracted from the process specifications and composed into a *causality* graph. The causality graph is then scheduled using relative scheduling. The sequencing and timing constraints implied by this composed interface schedule are then reflected to the individual processes as external timing constraints, and each process is scheduled and synthesized accordingly. Since relative scheduling requires resource allocation/binding to be performed before scheduling, this approach is not suitable for the synthesis of multiple-process designs under global resource constraints; e.g., the area capacity of a chip. A preferred approach is one that can concurrently trade off the performance and resource requirements of each process during scheduling.

2.4 Design Verification

There are a variety of systems and approaches for design verification. A comprehensive survey can be found in [CP88, McF93]. The most common approach for design verification at virtually every abstraction level is *numeric* simulation. However, without simulating all possible operating conditions, numeric simulation can only show the existence of errors but not their absence [CP88].

Formal methods, on the other hand, use mathematical reasoning rather than test-case experiments to show that the design will behave properly as required by the specification. Generally, formal verification relies on a formal model, which must be mathematically sound, to represent specifications, implementations and properties. Additionally, a set of reasoning rules associated with this formalism must be developed to perform proofs. Some reputed formalisms are higher-order logic [HD86], temporal logic [Boc82] and behavior expressions [MP83]. However, it requires a great deal of expertise and time to express designs or to do a nontrivial proof in any of these formalisms [McF]. Hence, formal verification at present is still extremely difficult, if not impossible, to automate.

There is another class of verification approaches called *symbolic* simulation. The idea behind symbolic simulation is to evaluate circuit behavior over expanded sets of signal values so that a number of operating conditions can be simulated in a single run. In late 1970's, researchers at IBM applied symbolic simulation to hardware verification at the register-transfer level [Dar79]. The research activities on this problem only lasted till the early 1980's [Cor81] due to the weakness of the symbolic manipulation methods. For example, the algebraic expressions built up during the course of simulation may become too large and cumbersome to manipulate effectively. Also, conditional branching and looping constructs proved even more intractable, since in general the simulator could not determine which conditional branches would be taken or the conditions under which a loop would terminate. The fundamental problem is that many properties of algebra on the integer domain are undecidable [Bry90]. Hence, it becomes very difficult for the algebraic manipulator to determine the equivalence of two complex expressions.

Recently, there is a renewed interest in symbolic simulation at the logic level due to the introduction of Ordered Binary Decision Diagrams (OBDDs) [Bry86]. This representation has been shown to be *canonical*; i.e, each Boolean function has a unique representation. Hence, a BDD-based symbolic simulator [BBB+87] becomes very useful for verifying combinational circuits. The verifier can simply introduce Boolean symbols for each primary input and simulate the circuit to obtain a Boolean function in OBDD for each primary output. These functions can then be compared with the ones derived from the circuit specification. For sequential circuits, however, more sophisticated approaches are required if the circuit and its specification are using different state encodings. One approach [BF89] is to require the relation between the state encodings be specified explicitly. A hindrance of BDD-based approaches is that the size of an OBDD can become exponential in terms of the number of Boolean symbols. Hence, the scale of circuits that can be handled is limited. In CATHEDRAL [GRVM90], a divide-and-conquer approach called *SFG-tracing* [GGP+91] is employed. In SFG-tracing, the specification is first partitioned, and the partitions in the specification are used to verify the implementations by BDD-based symbolic simulation.

RLEXT [KW89] is a rule-based system developed by Knapp and Winslett for automatic verification of synthesized RTL designs,. RLEXT relies on a unified representation similar to the USC Design Data Structure (DDS) [KP85] for design information including the behavioral specification, the structural implementation and the relation between them. A set of rules are defined to detect possible design inconsistencies like value collisions, unbound operations and data dependency violations. On novel feature of RLEXT is that it allows the user to modify the design structure and then provides the ability to repair some design-rule violations automatically. This approach, however, does not consider the control logic nor the interaction between the datapath and the controller. How the timing issues such as clocking schemes and delays are handled is not described.

Chapter 3

System Specification, Representation and Translation

In general, the first task of an automatic synthesis system is to translate a behavioral specification of a design into a representation that is suitable for synthesis. In fact, this task may be the most critical one when the specification language and the representation system have incompatible models to describe the given design. This is because most specification languages and their simulation models need to be intuitive to the designers; contrarily, the internal design representations for behavioral synthesis are geared toward ease of synthesis.

Several languages have been used to specify behavioral descriptions as input to high-level synthesis systems. The major consideration in selection of the behavioral specification language involve specifying the behavior without introducing unnecessary structure or timing information which overconstrain the synthesis process. In the USC ADAM system, VHDL [Ins88] has been adopted as the primary specification language due to its standardization and popularity. Although the popularity of using VHDL as a behavioral specification language for behavioral synthesis is increasing, the major problem with VHDL is that it was intended to be a simulation language. Therefore, many semantics of the language are not suitable for synthesis or may severely overconstrain the design. Hence, the VHDL subset used by the ADAM system is restricted to purely functional behavior. The essential timing constraints must be supplied separately to the synthesis tools by the designers. ADAM's VHDL subset is described in Appendix A.

The design representations used by most behavioral synthesis approaches are quite different in style but, in general, they shares a similar objective which is to capture the necessary control and data flow information of the design behavior using one or two flat or hierarchical graphs. The ADAM system use a unified multi-level design representation called the DDS (Design Data Structure) [KP85] to model the design under development. This unique form of representation divides design information into four models (graphs) which describe the data flow behavior, the control and timing behavior, the logical structure and the physical structure of a hardware design. In addition, the relations among the objects in these models are represented by two or three-way *bindings*.

The task of VHDL to DDS translation is complicated by the fact that each VHDL process is described by sequential statements while DDS employs *single-assignment* data flow graphs for behavioral representation. Hence, extensive local/global data flow analysis and graph optimization are needed to produce fully parallelized data flow graphs for behavioral synthesis.

In this chapter, we will discuss several important aspects of the translation from VHDL to DDS; namely, flow analysis, optimization and modeling. A prototype compiler called VHDL2DDS has been developed and used to experiment with many examples including an AR filter, a robot-arm controller, arithmetic Fourier transformation, an FFT butterfly node and a JPEG image compression system. This tool currently serves as the front end of the ADAM system for accepting new designs to be synthesized. Since the details regarding VHDL and DDS have been described thoroughly elsewhere [Ins88, KP85], their terminology shall be used without further explanation in the following discussion.

3.1 Translation of VHDL Processes into DDS

In practice, most hardware designs consist of a number of concurrent processes, where a process is defined as an independent thread of control. Therefore, a hardware system described in VHDL is modeled by a number of concurrently running processes, each of which represents a piece of hardware that gets activated when one or more of its inputs on the sensitivity list changes or simply runs forever if it has neither a sensitivity list nor any wait statement. The behavior of a process in VHDL is described by a sequence of statements. For example, Figure 3.1 shows a sample VHDL description that consist of one process for a ones-counter module. Hence, the fundamental problem in the VHDL to DDS translation is to produce a data flow graph (DFG) and a control timing graph (CTG) in DDS for each process along with bindings between the two so that together they are functionally equivalent to the process description in VHDL.

3.1.1 Problem Statement

Basically, a VHDL process is described algorithmically by a sequence of statements. The statements are sequentially executed unless they specify alternative branching destinations. The main side effect of these statements is to modify the current state (variables) of the process.

In DDS, the behavior specification of a design is represented by a data flow graph (DFG), a control timing graph (CTG) and a set of bindings B between DFG and CTG. The DFG is a directed acyclic bipartite graph (O, V, E), where O is a set of operations, V is a set of values, and E is a set of directed edges that connect operations and values. This graph is a *single-assignment* graph. That is, a value in DFG is the result of function application and can be generated only once. The CTG is also a directed graph (P, R), where P is a set of points represent control or timing events and R is a set of directed edges (ranges) represent relations between

```
-- Ones-Counter - An Example VHDL Description

package OCtypes is
  type BYTE is array(0 to 7) of BIT;
end OCtypes;

use work.OCtypes.all;
entity ONES_CNT is
  port ( A: in BYTE; C: out INTEGER);
end ONES_CNT;

architecture BEHAVIOR of ONES_CNT is
begin
  process
    variable NUM, I: INTEGER;
  begin
    NUM := 0;
    I := 0;
    while I < 8 loop
      if A(I) = '1' then
        NUM := NUM + 1;
      end if;
      I := I + 1;
    end loop;
    C <= NUM;
  end process;
end BEHAVIOR;
```

Figure 3.1: A sample VHDL description

points. Special points are defined in CTG to represent conditional branches and loops. The modeling of conditional branches and loops in DDS will be discussed shortly in Section 3.1.2.4. The bindings B are a set of 2-tuples (O, R) which tentatively assign dataflow operations to timing ranges.

The task of VHDL to DDS translation can be briefly described as following:

- map the usage of each variable in the process into a sequence of values in the DFG.
- map the data manipulation constructs in the statements into the operations of the DFG.
- map the essential control flow implied by the process description into the CTG and produce the proper bindings between the DFG and CTG.

This translation is both important and difficult since it is analogous to the parallelization problem that transforms a sequential algorithm into a maximally parallel one. The objective is to extract the parallelism from the design specification in VHDL so that the parallel flow graphs in DDS can be selectively serialized during high-level synthesis to produce a RTL implementation that satisfies the design constraints.

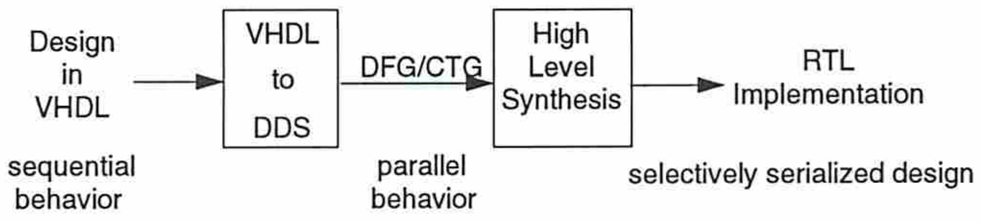


Figure 3.2: The parallelization during the VHDL to DDS translation

3.1.2 VHDL Translation Approach

There are two major issues to be dealt with in this mapping; namely, how to separate the control flow and the data computation information and how to analyze the data usage and dependency in a process description. In order to tackle this massive task, we approach it by a sequence of steps as shown in Figure 3.3. Briefly speaking, our approach is to first construct a flow graph from the parsed VHDL description[1] by performing control flow analysis [ASU86]. The vertices in this flow graph are so-called *basic blocks.* The flow graph effectively isolates the computations from the control flow by confining them in the basic blocks. Then, local data flow analysis is performed to collect intra-block data dependencies, followed by global data-flow analysis to find the inter-block data dependencies. After this step, the annotated flow graph becomes a combination of the data flow and control flow graphs. Finally, a number of graph reduction rules are executed to optimize the flow graph. In what follows, we will briefly discuss each of these steps.

3.1.2.1 Control Flow Analysis

In a VHDL process, the statements are sequentially executed unless the current statement specifies an alternative branching destination. Therefore, the sequence of statements from the one which was the target of the previous branching to the one currently altering the flow of execution constitutes a computation unit.

Definition 3.1 *A sequence of consecutive statements $S_i, \ldots, S_j$ that does not have the possibility of branching out in the middle is called a* basic block B_k *[ASU86].*

Definition 3.2 *A flow graph G is a directed graph $(\mathcal{B}, \mathcal{E})$ where $\mathcal{B}$ is a set of vertices representing the basic blocks and each edge $(B_x, B_y) \in \mathcal{E}$ denotes a conditional or unconditional branch from the basic block x to y.*

[1]We use a commercial VHDL syntax analyzer from Compass Design Automation to parse the VHDL descriptions.

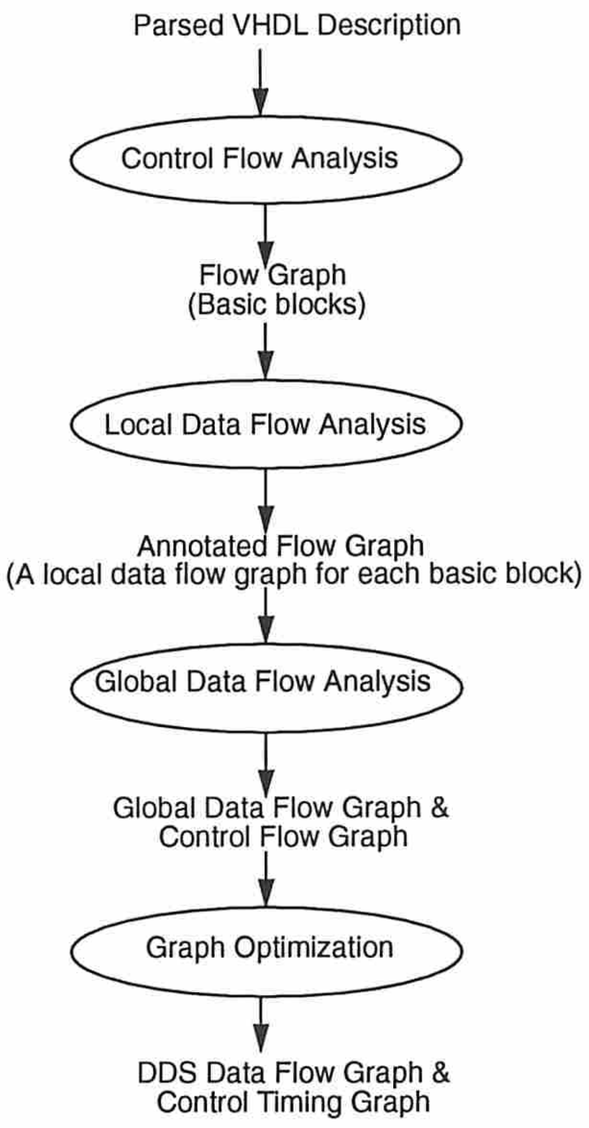


Figure 3.3: The steps to translate a VHDL process into DDS

Control flow analysis is a procedure that partitions a sequence of statements into basic blocks and constructs a flow graph that represents the flow of control.

Let the first statement of a basic block be called the *leader*. The set of leaders in a sequence of statements can be determined by the following rules:

- The first statement is a leader.
- Any statement that is the target of a conditional or unconditional branch is a leader.
- Any statement that immediately follows a branching statement is a leader.

After the leaders are identified, the following rules can be used to construct the flow graph of a sequence of statements $\{S_1, \ldots, S_N\}$. Let the leaders be $\{S_{l_1}, \ldots, S_{l_M}\}$, where $l_1 < \ldots < l_M$:

1. For each leader S_{l_i}, produce a basic block vertex B_k that consists of the statements from S_{l_i} to $S_{l_{i+1}-1}$.
2. For each branching statement, add an edge from the basic block in which it is located to each one of its target basic blocks. The edge is associated with the branching condition if the statement is conditional or simply *true* otherwise.

For example, after performing control flow analysis on the ones-counter example given in Figure 3.1, six small basic blocks can be found. The corresponding flow graph is shown in Figure 3.4.

The flow graph represents the essential flow of control of the VHDL process being translated. In fact, it can be regarded as the preliminary CTG in DDS since each basic block corresponds one or more sequential *ranges* in the CTG and each

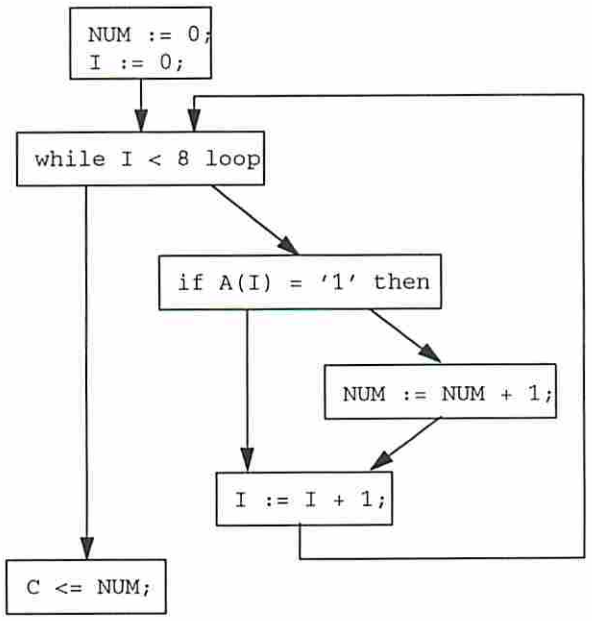


Figure 3.4: The flow graph of the one-counter example

edge[2] between two basic blocks can be transformed into a range from the ending point of the source block to the starting point of the destination block.

3.1.2.2 Local Data Flow Analysis

Each basic block in the flow graph is a computation unit. The only forms of statements that will appear in a basic block are those which perform computation and/or produce side effects; namely, assignments, procedure calls and expressions. Each of them can be modeled as $(I(S), O(S), F(S))$ where $I(S)$ is a set of variables which are referenced by the statement S, $O(S)$ is a set of variables whose values are updated by S, and $F(S)$ is the set of operations performed by S.

Here, we are interested in the inter-statement data dependency within each basic block. That is, we want to know which statements within a basic block

[2]Except the loop-back edges. Each such edge is implied by a pair of special points in the CTG which denote the places where control is returned or transferred when a loop-back is made.

define the values for each I set and similarly who will use the values of each O set. Let $B = \{S_i, \ldots, S_j\}$ and $i \leq k \leq j$. The following rules can be used to collect the definition/use information:

1. The *definition* of a variable $x \in I(S_k)$ is either the largest d such that $i \leq d < k$ and $x \in O(S_d)$ or it is defined outside B if no such S_d can be found.

2. The *use* of a variable $y \in O(S_k)$ is a list of statements U after S_k such that y is in their I sets but y is not in the O set of any statement within this region (from S_k to the last statement in U).

For example, let a basic block contain the following statements:

$$\begin{aligned} a_1 &= in_1 * p_1 \\ a_2 &= in_1 * p_2 \\ b_1 &= in_2 + a_2 \\ out_1 &= b_1 * p_3 \\ c_1 &= out_1 * p_4 \\ out_2 &= a_1 + c_1 \end{aligned}$$

After producing the input and output sets of each statement and analyzing the inter-statement data dependency, we have the definition/use table shown in Table 3.1.

Additionally, we need to know the input set of B that are the variables whose values are defined elsewhere but are referenced in B and the output set of B that contains all the variables being updated by B. These two sets are important in the global data flow analysis. $I(B)$ and $O(B)$ can be determined by the next two equations.

$$I(B) = I(S_i) \bigcup \left\{ \bigcup_{k=i+1}^{j} \left(I(S_k) - \bigcup_{l=i}^{k-1} O(S_l) \right) \right\}$$

$$O(B) = \bigcup_{k=i}^{j} O(S_k)$$

Statement	*Input Set*		*Output Set*	
	val	def	val	use
1: $a_1 = in_1 * p_1$	in_1	0	a_1	7
	p_1	0		
2: $a_2 = in_1 * p_2$	in_1	0	a_2	3
	p_2	0		
3: $b_1 = in_2 + a_2$	in_2	0	b_1	4
	a_2	2		
4: $tmp = b_1 * p_3$	b_1	3	tmp	5, 6
	p_3	0		
5: $c_1 = tmp * p_4$	tmp	4	c_1	7
	p_4	0		
6: $out_1 = tmp$	tmp	4	out_1	0
7: $tmp = a_1 + c_1$	a_1	1	tmp	8
	c_1	5		
8: $out_2 = tmp$	tmp	7	out_2	0

Table 3.1: The use/definition table of the basic block example

As a result, the basic block example given earlier will have an input set $\{in_1, in_2, p_1, p_2, p_3, p_4\}$ and an output set $\{a_1, a_2, b_1, c_1, tmp, out_1, out_2\}$.

After this analysis, it is conceptually easy to produce a piece of DFG that corresponds to each basic block. This is because the definition and use information provides the inter-statement data dependencies while the syntax trees of the expressions within a statement give the intra-statement data dependencies. By transforming each operation in the F sets of all the statements in the basic block into a node and each data dependency into a link between nodes, a piece of DFG is built. For instance, Figure 3.5 shows a piece of data flow graph that corresponds to our basic block example.

3.1.2.3 Global Data Flow Analysis

Unfortunately, the inter-block data dependency is much more complex than the intra-block one previously discussed, because the flow of execution is no longer a

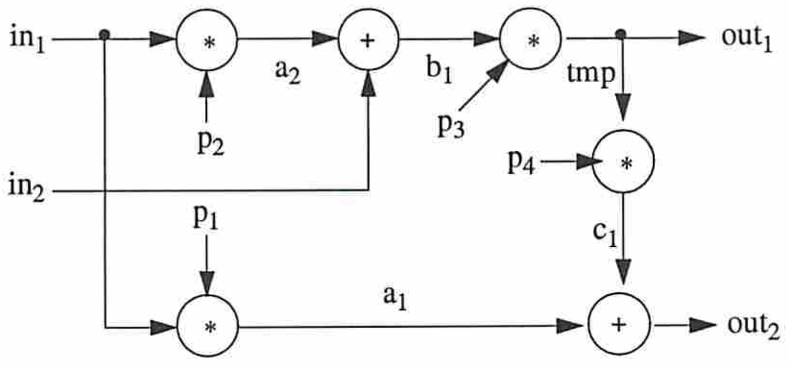


Figure 3.5: The data flow graph of the basic block example

straight line as within a basic block. For example, a basic block after a conditional structure may have more than one basic block which define its input values; consequently, the correct data dependencies are decided by the branching conditions. Hence, it is not sufficient to simply scan backward (forward) along the incoming (outgoing) paths of a basic block in the flow graph.

Our approach for this global data flow analysis is outlined in the following steps:

1. Hierarchically group the basic blocks within the same structure, such as *if-then-else* or *while-loop* structures, into a *meta block* as shown in Figure 3.6.

2. Calculate the input and output sets of those meta blocks according to Table 3.2 [3].

3. Analyze the definition and use relationships among the meta blocks hierarchically.

Readers may wonder why the input set of a *if-then-else* meta block shown in Table 3.2 contain some variables also in its output set. Specifically, any variable

[3]Other forms of conditional and iterated statements can be transformed into those two shown in Table 3.2.

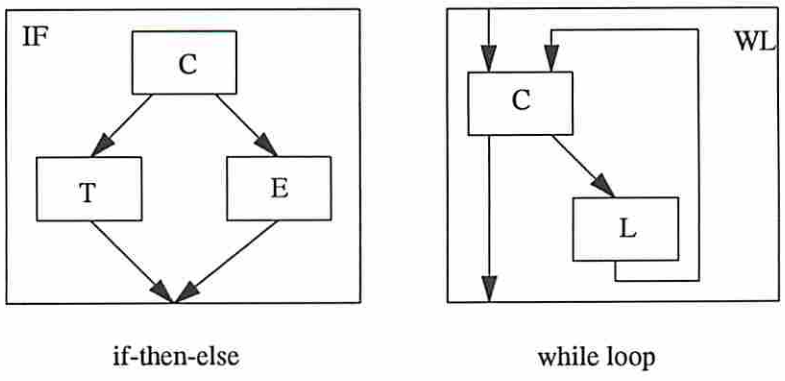
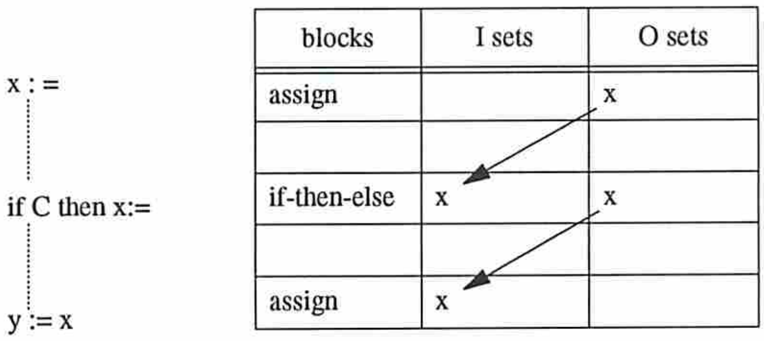


Figure 3.6: Meta blocks for if-then-else and while-loop structures

Type of meta block	Input and output sets
if C then T else E	$I(IF) = I(C) \cup I(T) \cup I(E) \cup \{O(IF) - (O(T) \cap O(E))\}$ $O(IF) = O(T) \cup O(E)$
while C loop L	$I(WL) = I(C) \cup I(L) \cup O(L)$ $O(WL) = O(L)$

Table 3.2: Calculation of input and output sets for the meta blocks

appearing in the output set of the IF block but not modified by both the T and E blocks must be included in the input set of the IF block. This is illustrated in Figure 3.7. Here y get the value of x either from the **if** statement if C is *true* or



blocks	I sets	O sets
assign		x
if-then-else	x	x
assign	x	

Figure 3.7: A conditional assignment

the old value of x from the earlier assignment. By enforcing x as one of the input variables of the **if** meta block, this block can then supply the correct value for y by merging these two possible definitions of x during the global definition and use analysis.

By introducing meta blocks into the flow graph, the global data flow analysis can be done hierarchically. Since each level of hierarchy basically consists of a sequence of basic blocks or meta blocks, the inter-block data dependency can be analyzed using the techniques similar to the local data flow analysis discuss earlier except that the granularity is at the block level instead of at the statement level. The definition and use analysis within each meta block, however, is done in a bottom up fashion. An if-then-else meta block IF consists of three subblocks, a condition C, a then body T and an else body E. After we have analyzed these three subblocks, the definition and use of $I(IF)$ and $O(IF)$ can be obtained by the following rules:

1. The *definition* of each variable $x \in O(IF)$ is a 2-tuple (def_T, def_E). If $x \in O(T)$, def_T is from subblock T; otherwise, def_T is defined outside the IF meta block and will be determined at next level of hierarchy. def_E is determined in a similar way for subblock E.

2. The *use* of each variable $y \in I(IF)$ denotes every subblock whose input set contains y.

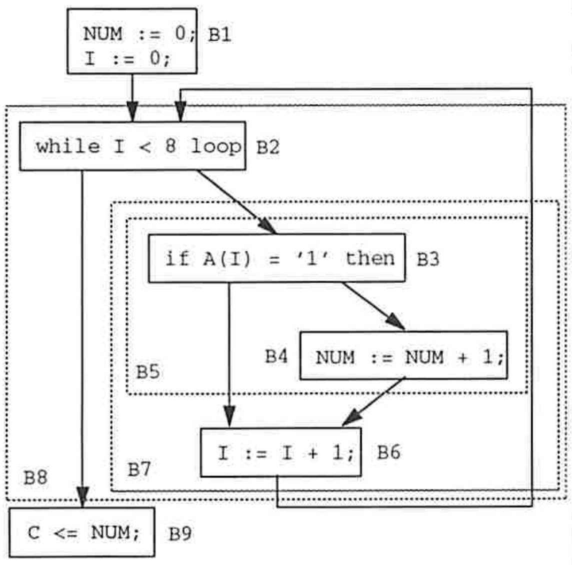
On the other hand, a while-loop meta block WL consists of two subblocks, a condition C and a loop body L. Its data dependency is analyzed in the following way:

1. The *definition* of each variable $x \in O(WL)$ is a 2-tuple (def_L, def_{init}), where def_L is from subblock L and def_{init} is the initial value of x defined outside WL meta block.

2. The *use* of each variable of $y \in I(WL)$ denotes every subblock whose input set contains y.

3. If a variable z in $I(C)$ or $I(L)$ is also in $O(L)$, its definition becomes a 2-tuple (def_L, def_{init}), where def_L is is the previous-iteration value from subblock L and def_{init} is the initial value defined outside WL.

Figure 3.8 shows the block hierarchy of the one-counter example and the result of global data flow analysis. The definition/use table shown in this figure describes the inter-block data dependency. For example, from this table we can find that the definition of I in Block 3 comes from its parent block (Block 5), which in turns from higher levels (Block 7 and Block 8) of the block hierarchy. From Block 8, we know that the initial value of I comes from Block 1 and its previous-iteration value is from Block 7.



Block	Input set			Output set		
	val	def	use	val	def	use
1				NUM I		8 8
2	I	8				
3	A I	5 5				
4	NUM	5		NUM		5
5	A I NUM	7 7 7	3 3 4,5	NUM	4,5	7
6	I	7		I		7
7	A I NUM	8 8 8	5 5,6 5	I NUM	6 5	8 8
8	A I NUM	0 1,7 1,7	7 2,7,8 7	I NUM	7,8 7,8	9
9	NUM	8		C		0

Figure 3.8: Global data flow analysis of the one-counter example

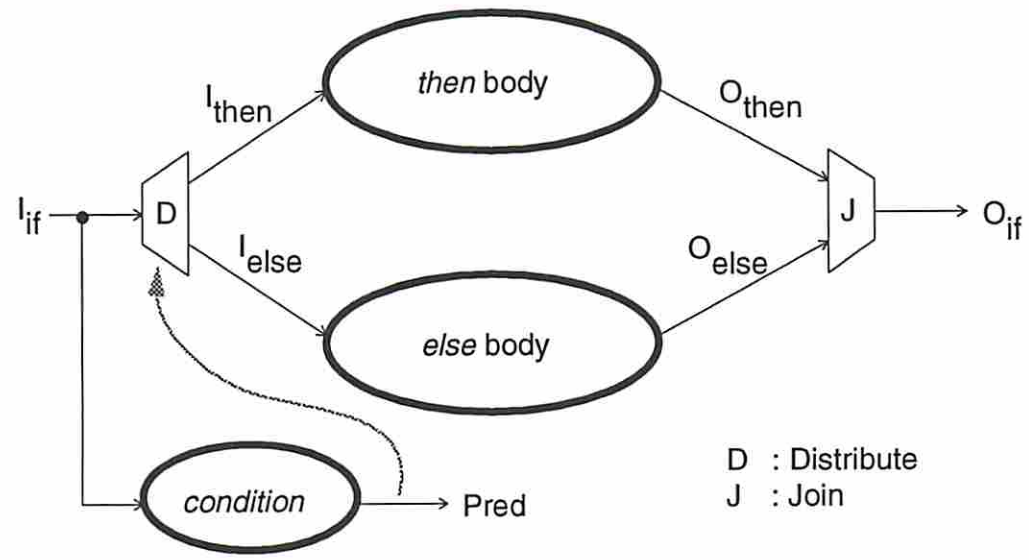
3.1.2.4 Graph Generation

After the global data flow analysis, all the inter-block data dependencies are found and the global DFG and CTG can be generated. Generalized DDS templates for the if-then-else and while-loop structures are shown in Figures 3.9 and 3.10. In each of the templates, two pseudo operations, D (distribute) and J (join), are used to support conditional data dependencies. Each D operation is like a 2-to-1 demultiplexer which distributes the input value to one of its outputs according to the predicate. Contrarily, a J operation is like a 1-to-2 multiplexer which merges two possible input values and presents one at its output. The while-loop template uses a pair of pseudo operations, LB (loop begin) and LI (loop iterate), to denote an implicit value feedback from LI to LB so that the DFG can remain acyclic. These pseudo operations do not necessarily correspond to actual hardware modules after synthesis. Their presence is mainly for synthesis tools to detect mutual exclusiveness and loop feedback in DFG.

In the CTGs of both templates, a pair of *or-fork* and *or-join* points are used to represent conditional execution. Each outgoing range of an *or-fork* point is associated with a condition. The flow of execution follows the range whose condition is satisfied. The CTG of a while-loop structure begins at an α point and iterates at an ω point, at which time the flow of execution returns to the α point. Note that the range between the ω and *or-join* points will never be taken. Its presence is mainly to make the CTG connected.

Figures 3.11 shows the DFG and CTG generated for the ones-counter example. The implicit control and feedback edges are also shown in the DFG using grey arrows.

DFG:



CTG:

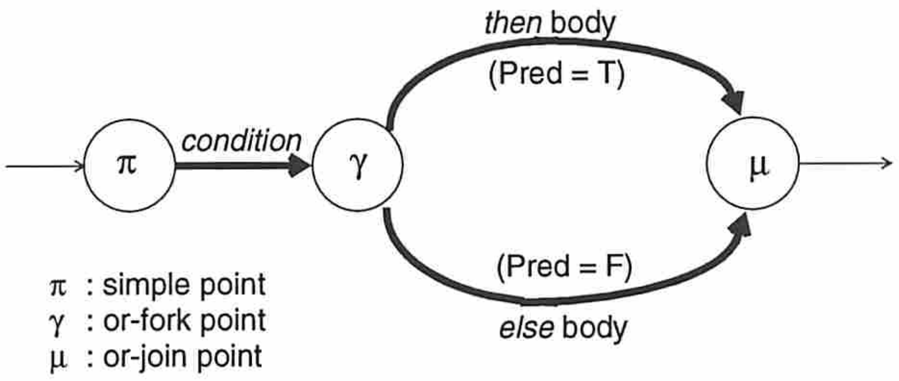
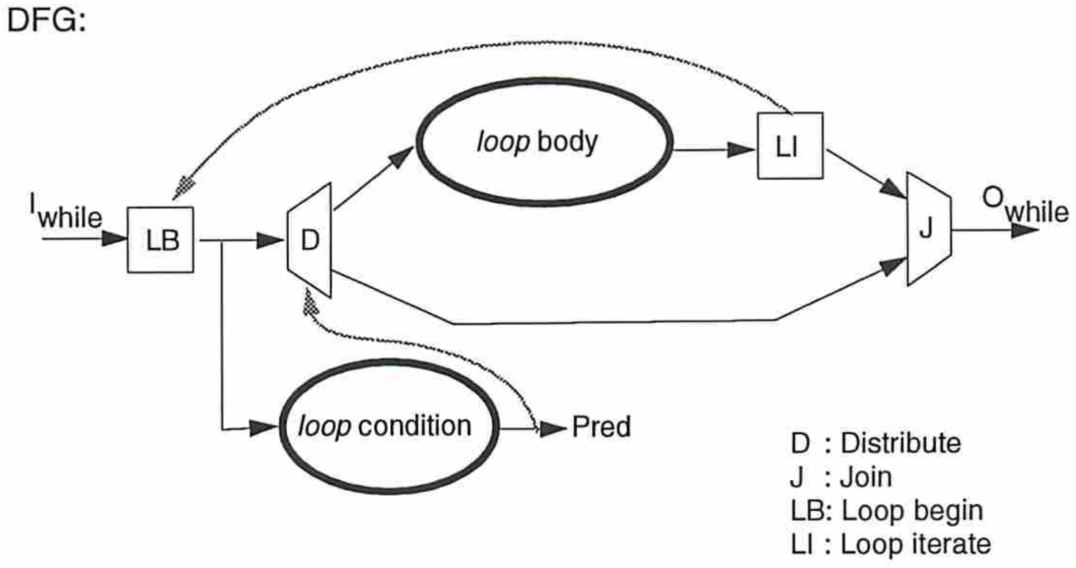


Figure 3.9: The DDS template of an *if-then-else* structure



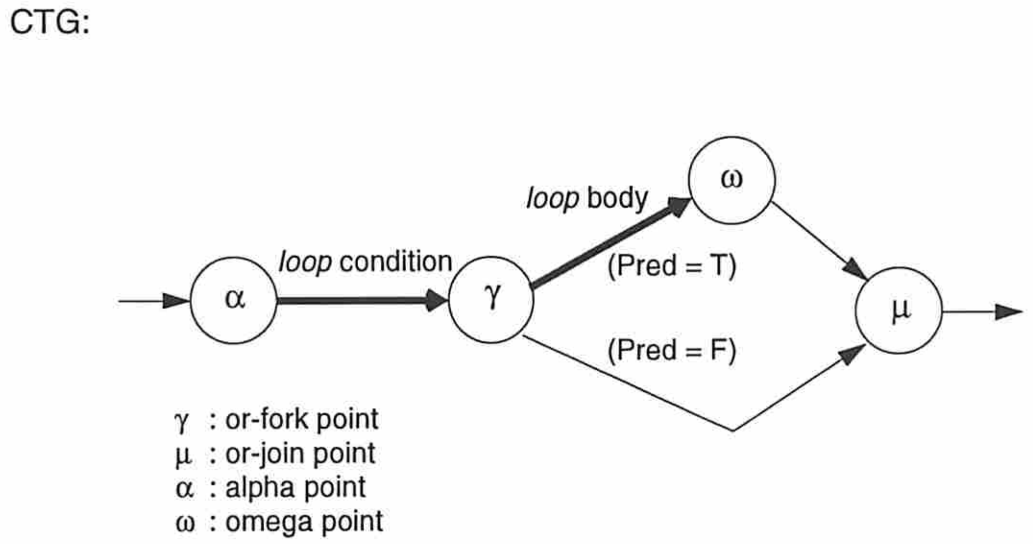


Figure 3.10: The DDS template of a *while-loop* structure

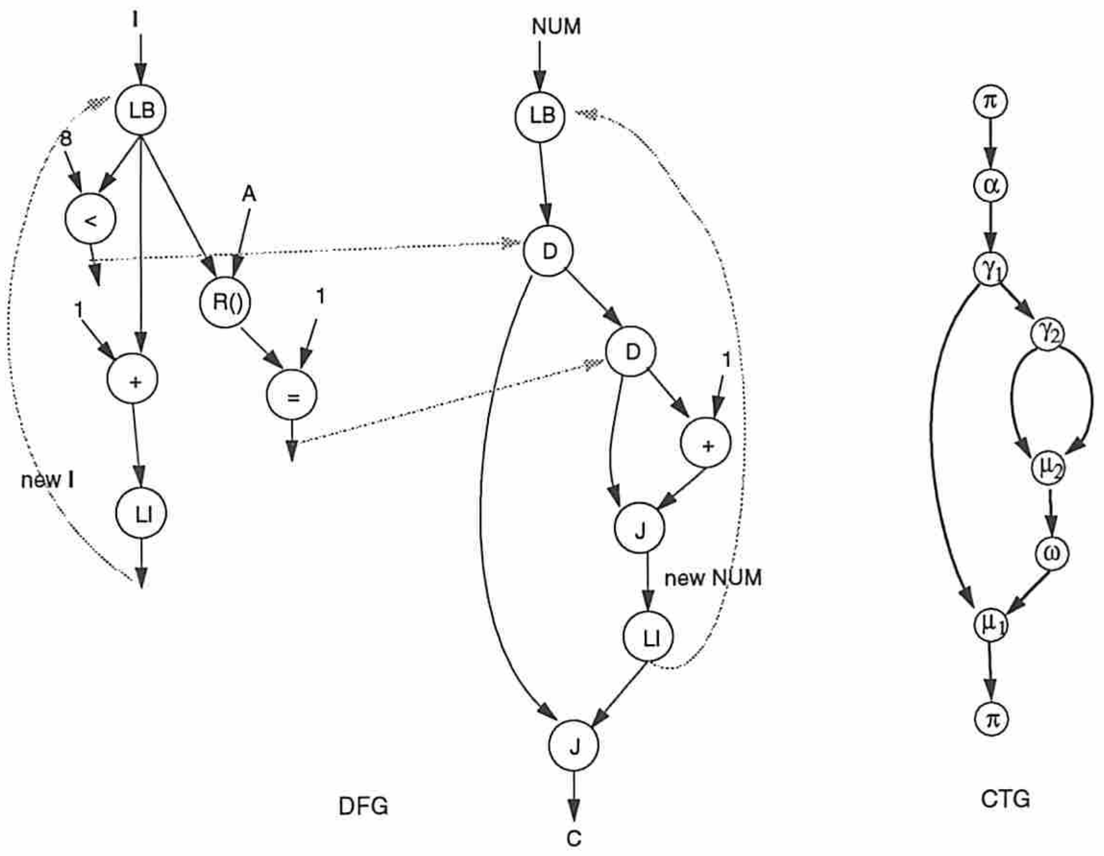


Figure 3.11: The DFG and CTG of the ones-counter example

3.1.2.5 Graph Optimization

Like traditional compilers for programming languages, there is plenty of opportunity to optimize the graphs during the VHDL to DDS translation. However, some transformations, such as tree-height reduction and loop unfolding, do not always result in a hardware design that is preferred by designers. For example, tree-height reduction [NP91] can reduce the critical paths of data flow graphs, but it can also introduce additional operations. We feel that these kinds of transformations should be performed separately and interactively by the designer. Here, we focus on those transformations that are guaranteed to improve the final design[4]. The flow graphs and the definition/use information are particularly useful in performing these transformations.

Lemma 3.1 *For any basic block B_k in a flow graph where $k > 1$, if B_k does not have any incoming edges then it is* dead code.

Proof: Since B_k does not have any incoming edge, there exists no path from B_1 to B_k. Therefore, when the execution begins at B_1, there will be no way to reach B_k under any input. $\Box$

This lemma provides an easy way to eliminate the redundant parts of a behavioral specification, but it does not guarantee to identify all of them. This is because those basic blocks with incoming edges are also dead codes if their path conditions can never be evaluated to *true*.

After the definition and use analysis, some values in the data flow graph may not be used by any operations and they are not the primary outputs either. These values are called *dangling*. The following rules can be used to eliminate the dangling parts of a data flow graph:

[4]Some straightforward transformations like constant propagation and copy operation elimination will not be discussed here.

1. If there is no entry in the use list of a value and it is not a primary output, mark it *dangling.*

2. If the outputs of an operation are all dangling, remove it and all its output values from the graph. Additionally, for each input of this operation, delete it from all the use lists in which it appears.

Lemma 3.2 *For any two operations $O_1(x_1, \ldots, x_n)$ and $O_2(y_1, \ldots, y_n)$, if O_1 and O_2 are of the same type and x_i and y_i are defined by the same value for all i, then O_1 and O_2 are* common subexpressions. *Therefore, one of them can be removed from the graph and the definition and use information of its outputs can be redirected to the other.*

Proof: The proof is trivial. Since O_1 and O_2 perform the same function, they should produce equivalent results under the same input condition. Furthermore, if there is a commutative property, the order of inputs may not be important as long as their correspondence can be established. $\square$
By recursively applying this lemma to a data flow graph, many of the common subexpressions can be eliminated.

Lemma 3.3 *For any operation $O(x_1, \ldots, x_n)$ within a loop, if x_i are defined outside the loop for all i, then O is a* loop invariant *operation which can be moved out of the loop.*

Proof: Since all the inputs of O are defined before the loop is entered, the input condition of O will not change at any iteration of the loop. Consequently, its outputs will remain the same at every iteration. Therefore, it is equivalent to compute O once before entering the loop. $\square$
However, care must be taken when performing this transformation. This is because the loop may execute zero times. In this situation, moving O out of the loop will cause an additional operation to be executed, which may produce an invalid result.

Hence, each output of O may need to be merged with its previous value before the loop if there is a subsequent use of it after the loop.

3.2 Array and Input/Output Modeling

Unlike scalar variables, input/output ports and arrays have to be handled in a different way since they are associated with structural attributes. For example, we cannot safely assume that a sequence of assignments to an output port is equivalent to the last assignment to this port. This is because the designer may intend to transfer a set of data via a single pair of input/output ports. In addition, when an element of an array gets modified by an assignment statement, the subsequent references to the array cannot be made parallel with the assignment unless we can make sure that they refer to different elements in the array[5].

3.2.1 Arrays

Arrays cannot be handled like scalar variables in a single-assignment data flow graph, because an array is a composite object that consists of elements of the same type. A sequence of assignments to the elements of an array cannot be done in parallel when there is a possibility that two assignments may refer to the same element. Even if two assignments can be guaranteed to refer to two different elements of an array, they may still have to be sequentialized if the array is going to be implemented in a single-port memory.

In our model, for each array type declared in VHDL, two additional operations, *array read* operations and *array write*, are implicitly defined as shown in Figure 3.12. An array read operation takes the designated array and the associated indexes as inputs and provides the value of the referred element as the output.

[5]We parallelize array accesses in the event that the synthesis software has available multiport memories.

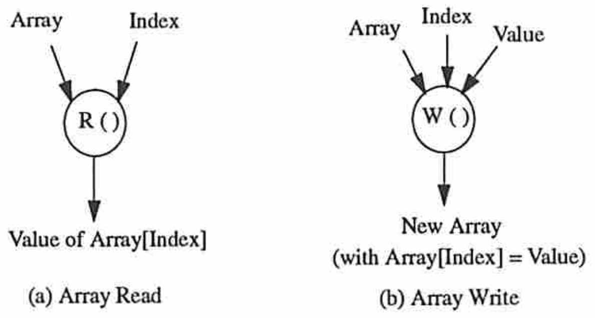


Figure 3.12: The array read and write operations

On the other hand, an array write operation takes an additional input for the value to be written to the referred element and produces an new instance of the array. If an *indexed name* appear at the right (left) hand side of an assignment statement, an array read (write) operation is created in the DFG.

A sequence of array read operations occurring before the array is modified are made in parallel by VHDL2DDS. Hence, the subsequent synthesis tools can selectively sequentialize them according to the number of read ports of the memory module chosen to implement the array.

On the other hand, a sequence of writes to an array are explicitly sequentialized according to the order of assignments appearing in VHDL. This sequentialization is done by making the array input of a write operation refer to the instance of the array produced by the previous write operation. The current implementation of VHDL2DDS does not analyze the indexes of array write operations in order to parallelize those array write operations which will refer to different elements in the array.

3.2.2 Input/Output and Inter-Process Communication

As we mentioned earlier, most hardware systems consists of a number of concurrent processes which may communicate with each other or the environment. For instance, the VHDL example given in Figure 3.13 consists of two processes which communicate with each other through the global variables *start*, *finished* and *p2temp*. Additionally, process one is communicating externally through two input ports *data1* and *data2* as well as a output port *result*.

3.2.2.1 Types of Communication

In general, the communication is considered *synchronous* if the send action in one process and the receive action in another refer to the same clock; otherwise, it is *asynchronous* and must be done through explicit handshaking. Furthermore, the communication can be either *static* or *dynamic*. Communication is static if the exact time it takes place can be determined during synthesis. That is, the send and receive operations are synchronized statically in time. This type of communication requires little synchronization overhead, but it imposes a strict timing constraint for synchronization on the schedules of the sending and receiving processes. We define dynamic communication to be one whose time cannot be scheduled to a fixed time; e.g., waiting for the assertion of an external signal. Obviously, dynamic communication needs more synchronization overhead (such as handshaking signals and the associated logic) than static communication does. It may also lead to higher controller costs due to the busy waiting on both the sending and receiving processes. The communication can be further classified into either *buffered* or *unbuffered*. In buffered communication, the execution of the sender or receiver does not have to be *blocked* (suspended) unless the buffer is full or empty.

```
entity example is
port (  data1 : in bit_vector(3 downto 0);
        data2 : in bit_vector(3 downto 0);
        result : out bit_vector(3 downto 0));
end example;

architecture example of example is
signal start, finished : bit;
signal p2temp : bit_vector(3 downto 0);
begin
  p1 : process
  begin
   start <= '1';
   wait on finished until finished = '1';
   start <= '0';
   result <= p2temp;
   wait on data1, data2;
  end process;

  p2 : process
  variable res : bit_vector(3 downto 0);
  begin
   finished <= '0';
   res := data1 + data2;
   p2temp <= res;
   finished <= '1';
   wait on start;
  end process;
end example;
```

Figure 3.13: A VHDL example with two processes

3.2.2.2 Modelling Communication in VHDL and DDS

In VHDL, inter-process communication is specified by the assignments or references of the global signals which are shared by the processes. Similarly, inter-chip communication is done via the input/output ports. Additionally, sensitivity lists and wait statements provide the mechanism to specify blocking communication events.

A reference to an input port or a global signal in a statement of a process implies a read operation of a communication event is taking place. For instance, the following statement from the VHDL example shown in Figure 3.13 consists of two read operations from input ports *data1* and *data2*:

```
res := data1 + data2;
```

Similarly, an assignment to an output port or a global signal becomes a write operation of the corresponding communication event. For example, the statement

```
p2temp <= res;
```

implies a write operation of an inter-process communication event via the global signal *p2temp*. A communication event consists of a write operation in one process and a number of read operations in the others.

In this model, a sequence of references to an input port or a global signal not only imply a number of read operations but also a sequential ordering among them which must be obeyed by the implementation[6]. This is also applied to a sequence of assignments to an output port or a global signal. A typical example of this situation is to transfer a set of data via one pair of input/output ports.

A blocking communication event can be specified by using a **wait** statement or a sensitivity list that contains the signal in question. Otherwise, the communication event is intended to be implemented statically as a non-blocking one. For example, the statements

[6]See McFarland's model [MP83] for an early formalization of this.

```
wait on x, y;
z := x + y;
```

imply that here the communication events for x and y are to be implemented as blocking ones.

In DDS, the timing and sequencing of communication can be represented hierarchically by the process-level CTGs and a system-level CTG as shown in Figure 3.14. A read or write operation in a process is represented by a binding (O, T, C), where

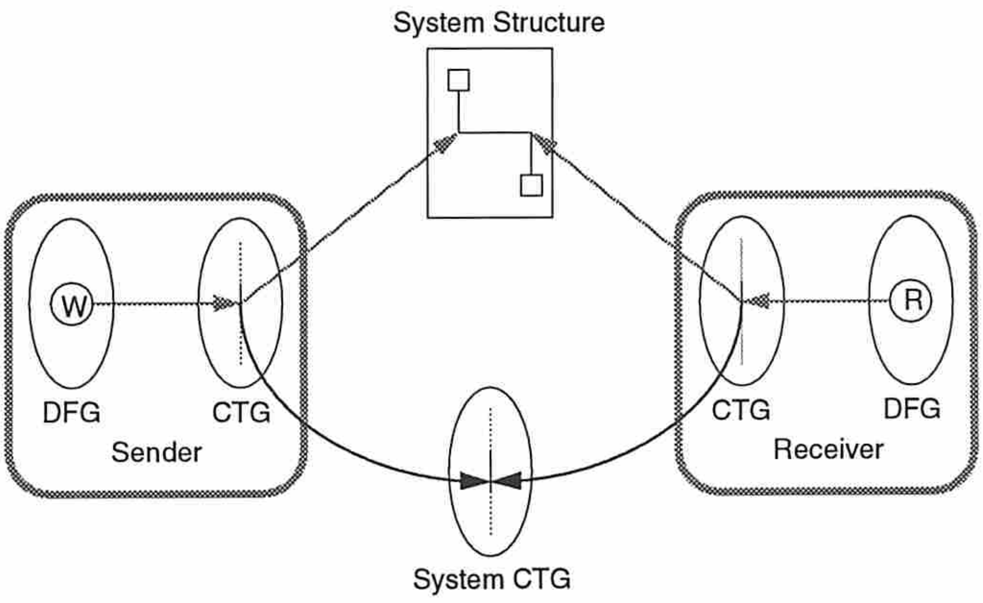


Figure 3.14: The communication model in DDS

O is a read or write operation in the DFG of the process, T is a range in the process's CTG, and C is a structural carrier which is either an input/output port or an inter-process communication link. The ranges of a sequence of read (write) operations from (to) an input/output port or a global signal are explicitly ordered in the CTG.

Similarly, each event for inter-process or inter-chip communication can be represented by a range in the system-level CTG. This range for a communication

Example	Description	Line	Condition	Loop	Synthesized
arf.vhdl	AR filter	79	no	no	yes
aft.vhdl	arithmetic fourier transform	145	no	no	yes
fir.vhdl	FIR filter	37	no	no	yes
ellip.vhdl	elliptic wave filter	90	no	no	yes
robot.vhdl	robot arm controller	99	yes	no	yes
diffeq.vhdl	differential equation solver	94	yes	no	yes
dct.vhdl	discrete cosine transform	71	no	yes	yes
oc.vhdl	ones counter	29	yes	yes	no
sqt.vhdl	square root function	25	yes	yes	no

Table 3.3: A partial list of examples translated by VHDL2DDS

event is a *composite* range that is composed by the ranges of a write action in one process and the read actions in the others. If the communication event is a blocking one, the duration of the range is specified as *unbounded.* Otherwise, a fixed delay, if known, can be given to the range to specify the communication time for an unblocking event. The relationships and constraints among the event ranges in the system-level CTG can be added by the designer to describe the system timing requirements and the interaction to the external world.

3.3 Experiments

The VHDL2DDS program has been involved in many top-down chip design experiments using the ADAM high-level synthesis system. Table 3.3 lists some of the examples that were translated by VHDL2DDS. In this section, we will use three examples to show the features of VHDL2DDS.

Figure 3.15 shows a simple VHDL description which illustrates the way that VHDL2DDS handles conditional branches, loops and arrays. Basically, this module performs a simple integer square root function. Though this description does little computation, its control flow is a good test for global data flow analysis. The data flow graph produced by VHDL2DDS is shown in Figure 3.16. In this graph,

```
-- An simple integer square root function
entity SQT is
   port(x : in integer;-- the input value
        y : out integer);-- the result
end SQT;

architecture BEHAVIOR of SQT is
begin
   process
      variable a, b: integer;
   begin
      if x > 0 then
         a := 1;
         b := x;
         while abs(a - b) > 1 loop
            a := (a + b) / 2;
            b := x / a;
         end loop;
         y <= a;
      else
         y <= x;
      end if;
   end process;
end BEHAVIOR;
```

Figure 3.15: The VHDL description of a square root function

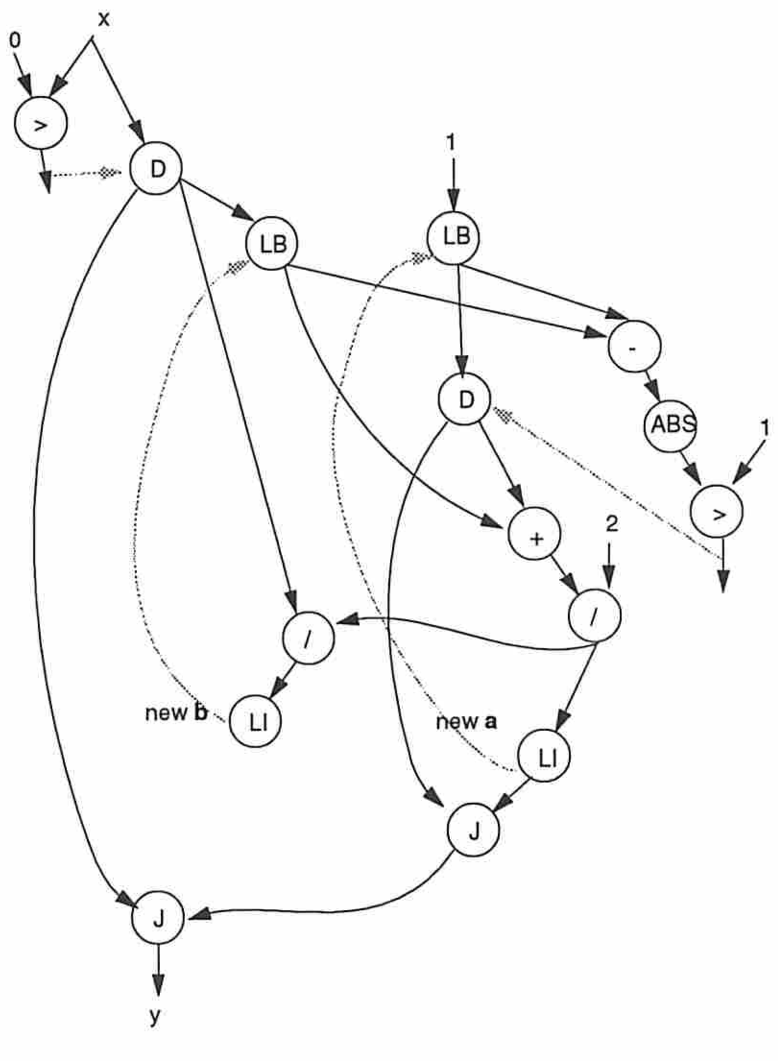


Figure 3.16: The data flow graph of the square root function

two pairs of LB (loop begin) and LI (loop iterate) pseudo operations are used to denote the implicit feedback values for a and b to be used in the next iteration of the while loop. The pair of D (distribute) and J (join) pseudo operations at the left hand side of the figure correspond to the **if** statement in the VHDL description and the D/J pair at the right hand side is the loop-exit condition for the **while** statement. The control edges (flags) and feedback edges are shown by grey arrows in this graph.

The robot arm controller example given in Table 3.3 was originally written in the C language. The C code was translated into a VHDL description in order to be synthesized through the ADAM system. Figure 3.17 shows the VHDL statements of this example. Though the VHDL description of this example contains four sequential conditional branches, VHDL2DDS was able to retain only the essential data dependencies and produced a highly parallelized data flow graph as shown in Figure 3.18.

Recently, a JPEG image compression system was designed using the Unified System Construction system [GCDBP94], which is an integration of several newer system-level tools with the ADAM synthesis system. Figure 3.19 shows the VHDL description of an 8x8 16-bit Discrete Cosine Transform module used in the compression system. This module performs 8 point DCT row-wise over an 8x8 frame. Figure 3.20 shows the data flow graph produced by VHDL2DDS. The feedback edges are not shown here to make the graph easier to interpret. In this graph, there are sixteen array read operations to $InFrame$ in parallel, but the eight array write operations to $OutFrame$ in the lower right part of the figure have been sequentialized by VHDL2DDS according to the order of the assignments in the VHDL description.

```
-- The VHDL statements of the robot arm controller example
ev1 := uv1 - xv1;
ev2 := uv2 - xv2;
u10 := kv1 * ev1;
u20 := kv2 * ev2;
em1 := xvh1i - xv1;
em2 := xvh2i - xv2;

if em1 < Z1 then             -- 1st conditional branch
  tmp1 := Z1 - em1;
else
  tmp1 := Z1 + em1;
end if;

if tmp1 < evthresh then      -- 2nd conditional branch
  em1 := Z2;
else
  em1 := Z2+em1;
end if;

if em2 < Z3 then             -- 3nd conditional branch
  tmp1 := Z3 - em2;
else
  tmp1 := Z3 + em2;
end if;

if tmp1 < evthresh then      -- 4nd conditional branch
em2 := Z4;
else
em2 := Z4 + em2;
end if;

xvh1o<=u10*T1+xv1;
xvh2o<=u20*T1+xv2;

mh110:=km11*em1*u11i+mh11i;
mh120:=km12*em1*u21i+mh12i;
mh210:=km21*em2*u11i+mh21i;
mh220:=km22*em2*u21i+mh22i;

q1<=mh110*u10+mh120*u20;
q2<=mh210*u10*mh220*u20;

mh11o<=mh110;
mh12o<=mh120;
mh21o<=mh210;
mh22o<=mh220;
u11o<=u10;
u21o<=u20;
```

Figure 3.17: The VHDL statements of the robot arm controller example

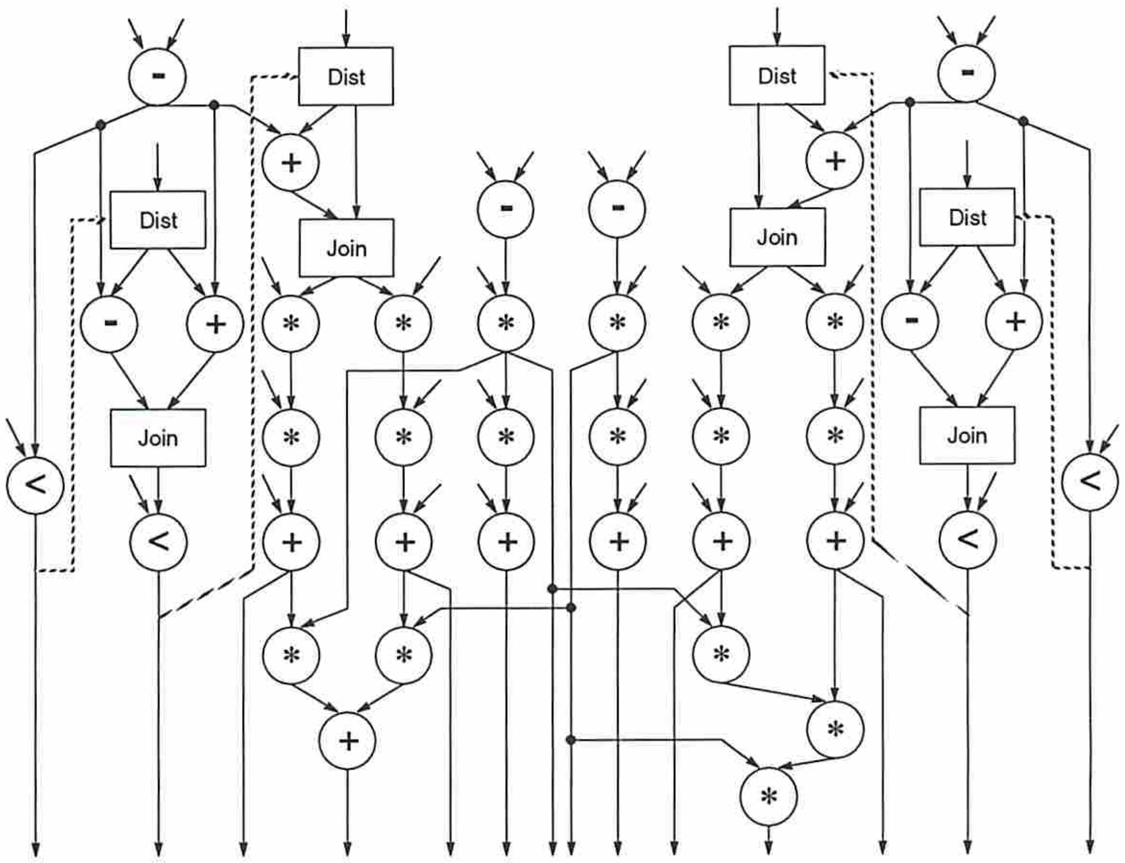


Figure 3.18: The data flow graph of the robot arm controller

```
 -- dct.vhdl : an 8x8 16-bit DCT VHDL behavioral description
package dct_data is
  type Frame is array(0 to 7, 0 to 7) of integer;
  constant c1,c2,c3,c4,c5,c6,c9,c12,c14,c15,c16 :integer;
end dct_data;

use work.dct_data.all;
entity dct is
  port(InFrame :in Frame;
    OutFrame :out Frame);
end dct;

use work.dct_data.all;
architecture behavior of dct is
begin process
  variable t1,t2,t3,t4, m1,m2,m3,m4: integer;
  variable i: integer;
begin
  for i in 0 to 7 loop
    t1 := InFrame(0,i)+InFrame(7,i);
    t2 := InFrame(3,i)+InFrame(4,i);
    t3 := InFrame(1,i)+InFrame(6,i);
    t4 := InFrame(2,i)+InFrame(5,i);
    m1 := t1+t2;
    m2 := t3+t4;
    m3 := t1-t2;
    m4 := t3-t4;
    OutFrame(0,i) <= c12*(m1+m2);
    OutFrame(4,i) <= c12*(m1-m2);
    t1 := c15*(m3+m4);
    OutFrame(2,i) <= c14*m3 + t1;
    OutFrame(6,i) <= c16*m4 + t1;
    m1 := InFrame(0,i)-InFrame(7,i);
    m2 := InFrame(1,i)-InFrame(6,i);
    m3 := InFrame(2,i)-InFrame(5,i);
    m4 := InFrame(3,i)-InFrame(4,i);
    t2 := c6*(m1+m2+m3+m4);
    t3 := c3*(m1+m4);
    t4 := c9*(m2+m3);
    OutFrame(1,i) <= c1*m1 + t3 + t2 + c2*m3;
    OutFrame(3,i) <= c5*m4 + t2 + c1*m2 + t4;
    OutFrame(5,i) <= c2*m1 + t2 + t4 - c4*m3;
    OutFrame(7,i) <= t3 + c4*m4 + t2 + c5*m2;
    end loop;
  end process;
end behavior;
```

Figure 3.19: The VHDL description of DCT

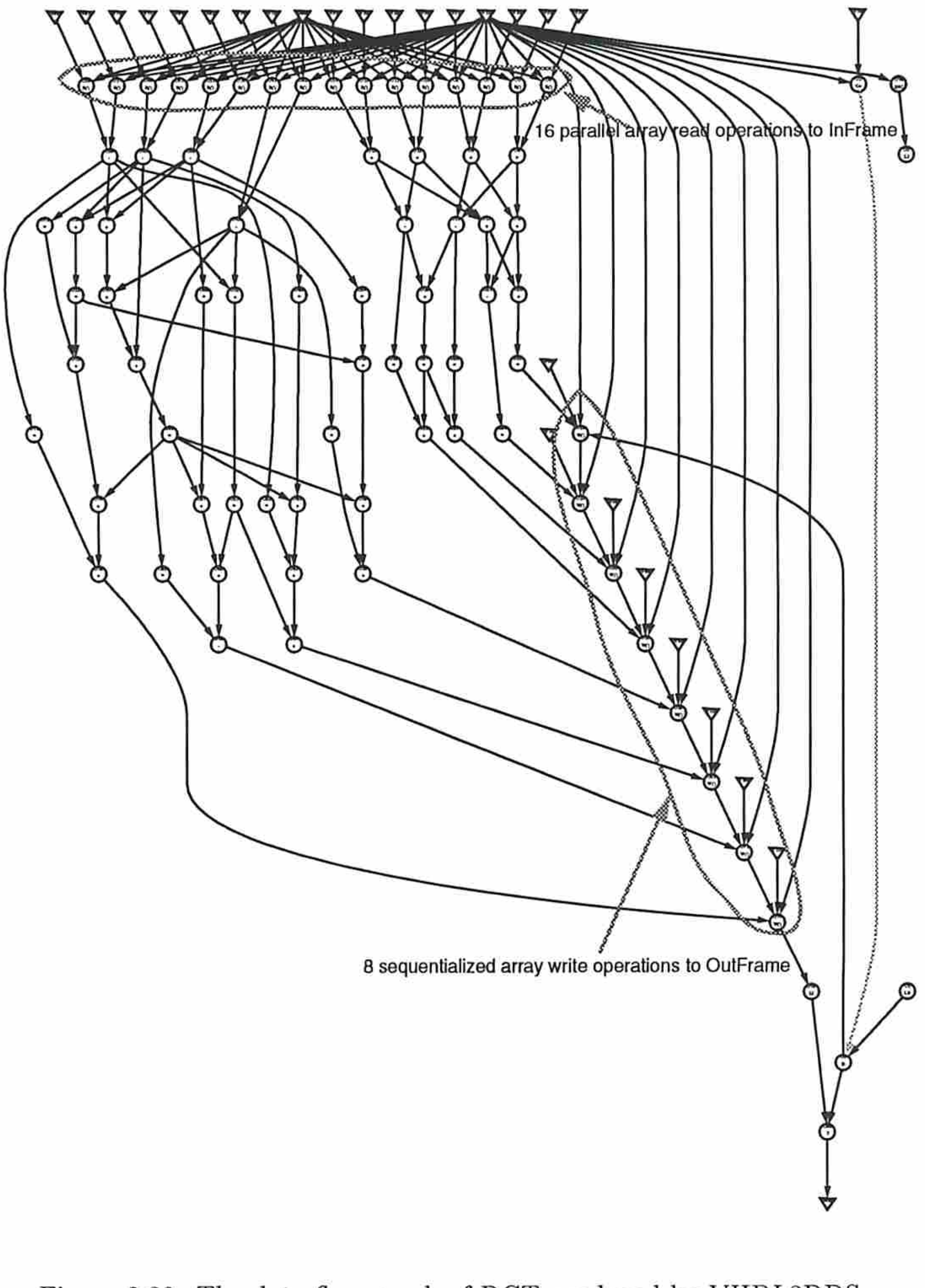

16 parallel array read operations to InFrame
8 sequentialized array write operations to OutFrame

Figure 3.20: The data flow graph of DCT produced by VHDL2DDS

3.4 Summary

In this chapter, we have discussed the problem of translating design specifications in behavioral VHDL into a flow-graph representation called DDS for synthesis. The primary objective of this translation is to extract the parallelism from the given sequential design specification by retaining only the essential data and control dependencies in the generated flow graphs. The techniques for control flow analysis, local/global data flow analysis, and graph generation/optimization have been presented. We also have described the modelling of arrays, input/output and inter-process communication in VHDL as well as DDS.

A compiler called VHDL2DDS has been implemented and is fully operational using the techniques described here. VHDL2DDS currently serves as the VHDL front-end of the ADAM high-level synthesis system, accepting new designs to be synthesized. It has been involved in numerous chip design experiments.

We believe the specific methodology presented here could also be applied to translate other HDLs or high-level languages into a synthesizable format. In addition, it can also be used to generate parallel machine code from sequential programs for highly parallel computers such as data-driven machines.

Chapter 4

System-Level Partitioning of Application-Specific Digital Systems

4.1 Introduction

The complexity of modern digital systems keeps increasing even as device sizes shrink dramatically. As a result, many such systems cannot fit into a single integrated circuit using available chip packages while satisfying all the given design constraints. Consequently, these system must be partitioned onto a number of chips at some point during the design process.

System partitioning can be classified into either structural partitioning or behavioral partitioning. In structural partitioning, the system behavior is first converted to structure, and then the structure is partitioned among chips. Structural partitioning usually can provide a solution that satisfies physical constraints such as area and pins. However, after the system structure is synthesized as a single-chip design, it may not be possible to partition the design onto multiple chips while satisfying the constraints (especially timing constraints). Alternatively, if the system partitioning is performed before the behavioral synthesis, such partitioning can heavily influence decisions made in subsequent synthesis tools and may therefore lead to higher performance designs and more efficient use of area and pins.

A major difficulty with behavioral partitioning is that the feasibility and quality of the partitioning is hard to measure. Generally speaking, with the help of new prediction techniques [Kuc91], behavioral partitioning is advantageous.

There are essentially two levels of granularity at which behavioral partitioning can be performed. Previous approaches to behavioral partitioning have mostly focused on partitioning the design behavior, usually a control data flow graph (CDFG), at the operation level onto a number of partitions which can be synthesized into separate hardware modules or chips.

On the other hand, behavioral partitioning can be done at a higher level of granularity, such as processes, procedures and memory blocks. Figure 4.1 shows the system partitioning of an ASIC-based CCITT H.261 video decoder from Bellcore Inc. In this figure, one can find that the system has already been divided into a

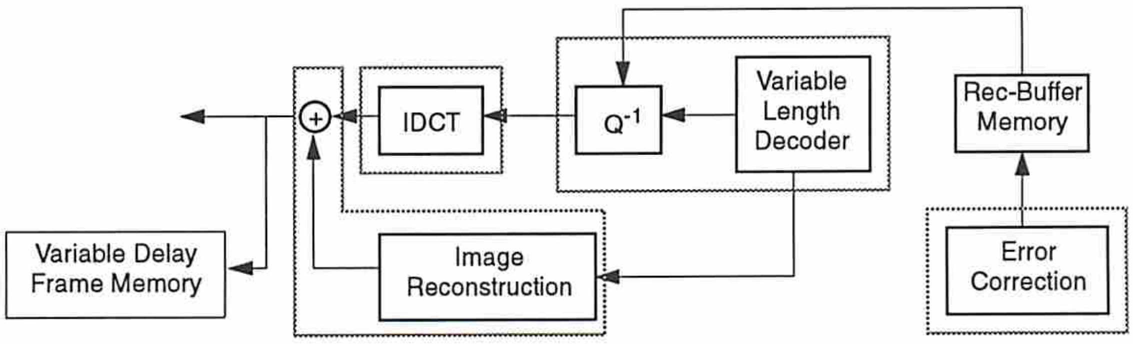


Figure 4.1: An ASIC-based CCITT H.261 video decoder from Bellcore Inc.

number of well-defined *functional entities* by the designers. As chip capacities keep increasing, each chip is likely to contain more than one functional entity. In this context, we use the term *process-level partitioning* to denote the form of system partitioning that preserves the functional boundaries specified by the designers.

There are several advantages to process-level partitioning. First, there are far fewer objects at the process level than those at the operation level, which allows us to utilize more comprehensive partitioning techniques like mixed-integer linear programming and to take into account more partitioning issues concurrently like the chip count, chip packaging and performance constraints. Also, since process-level partitioning preserves functional boundaries specified by the designer, the objects grouped into a chip are familiar to the designers. Therefore, it becomes much easier for the designers to test these chips functionally as compare to those which are partitioned at the operation or lower levels and may consist of many fragments of different system functions. In addition, if a process needs to be changed in the future redesign of the system, only the chip where the process is located is affected and the rest of the multi-chip system can remain intact.

Another important issue when partitioning a system with multiple processes is that each process to be synthesized later may have a wide range of alternative implementations with varying area and delay characteristics. The right combination of process implementations has to meet local/global design constraints as well as chip-packaging constraints. For example, Figure 4.2 shows two process $p1$ and $p2$ assigned to a chip. If a performance constraint T is imposed on the signal

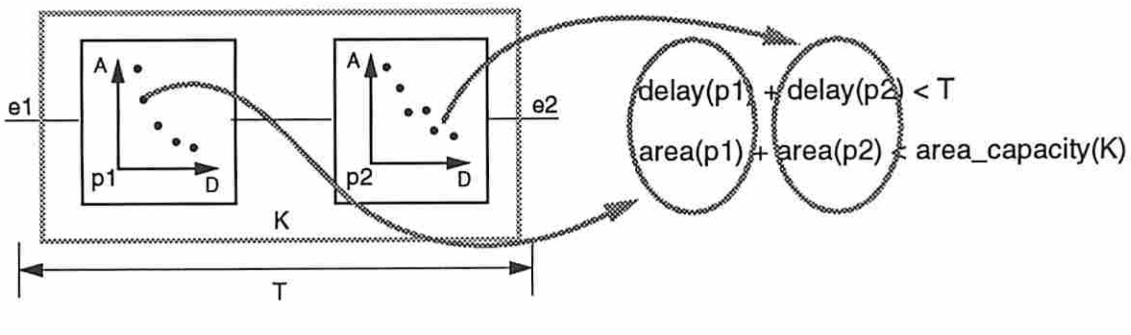

e1
A
D
p1
A
D
p2
e2
K
T
delay(p1) + delay(p2) < T
area(p1) + area(p2) ≤ area_capacity(K)

Figure 4.2: Selection of process implementations

path from $p1$ to $p2$, the design points chosen for $p1$ and $p2$ have to satisfy both the performance and chip-area constraints as shown in the figure. At present, selection of cost-performance characteristics for individual processes is mostly done manually by designers. However, the quality of partitioning results is highly sensitive to these design decisions and many tradeoffs are possible. By exploring the design alternatives of the processes in the system during partitioning, design point selection can be done not only to meet the given performance constraints but also in consideration of effects on the partitioned system in terms of the cost, size, reliability, and so on.

In this chapter, a process-level system partitioning approach will be presented. The novel aspect of this approach is that the exploration of process design alternatives is done concurrently with partitioning. In addition, the chip count and chip capacities (area and pins) are not simply a number of given constraints to be satisfied; instead, they are traded off according to the available chip packaging options. Prototype software, ProPart, based on this process-level partitioning approach has been developed and used to experiment with several examples including a JPEG image compression system.

4.2 Problem Approach

The problem which we are trying to solve here can be briefly described as

> *Partitioning a digital system with multiple processes into a number of custom chips so that the design constraints are met and the cost of these custom chips is minimized.*

Figure 4.3 shows an overview of our partitioning approach.

In our approach, the system to be partitioned is defined by a hypergraph $G(P, E)$, where P is a set of processes and E represents the interconnections among P. Each process in P is either an unrealized one to be synthesized later or an

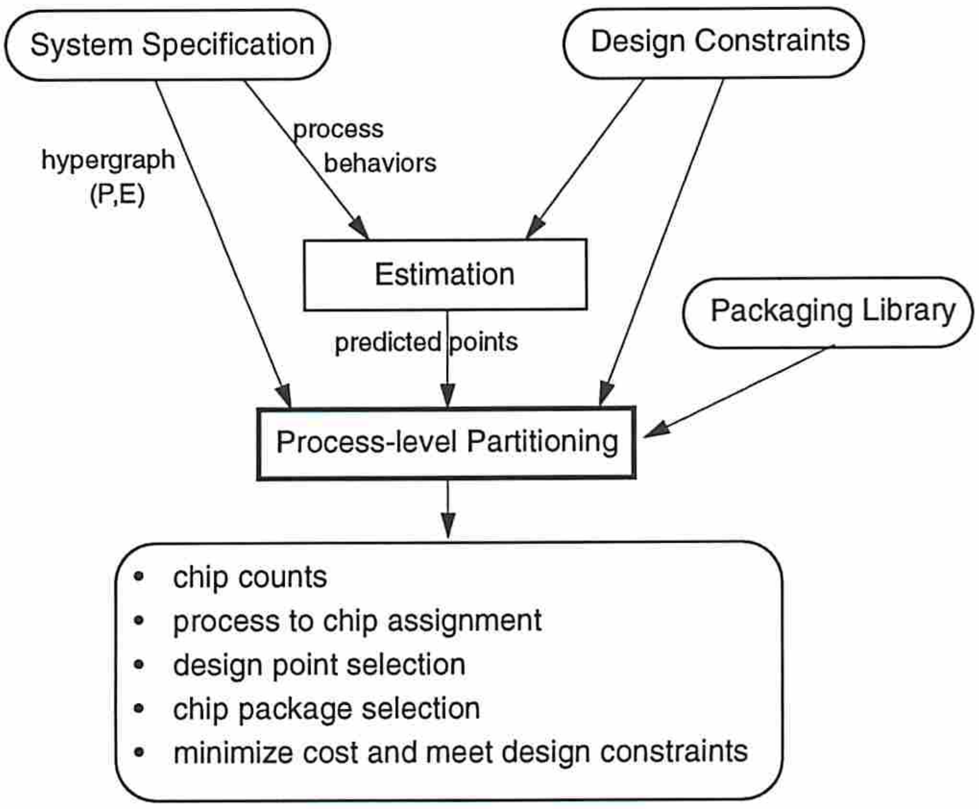


Figure 4.3: Overview of ProPart's partitioning approach

already-implemented chip or chip portion to be used "as is"[1]. For each process to be synthesized, its behavior can be passed to behavioral area-delay estimation tools like BEST [KP93] in order to predict a number of possible design points to be chosen during partitioning. In contrast, each implemented macro is assumed to have known area and delay parameters[2]. The hyperedges in E are multi-point interconnections weighted by their bitwidths. Each such edge represents a communication link among the connected processes. Given such a hypergraph, the number of pins needed for a partition (chip) is calculated as the sum of the weights of all hyperedges which cross the partition boundary.

Other inputs of the problem include a package library and a number of design constraints. The package library contains a set of available or preferred chip packages to be chosen for each chip. The area and pin capacities of these packages as well as their costs are specified in the library. Designers are allowed to impose constraints on performance and cost as well as on the chip count.

Our process-level partitioning approach involves in several interacting tradeoffs and partitioning decisions. The decisions to be made during partitioning besides assigning processes to chips are

- determining the number of chip partitions (chip count),
- selecting a suitable chip package for each chip partition from the library,
- selecting a feasible combination of design points for the processes in the system.

These partitioning decisions are inter-dependent in a complicated way. The design point selection and the process to chip assignment determines the sizes of chip

[1]The selection of an already-implemented macro versus one to be designed has already been made by higher-level tools.

[2]Programmable macros, of course, have unknown memory costs and delays. Characteristics of such macros can be predicted using appropriate estimators [RP93].

partitions and the required numbers of pins. Typically the dimensions of a chip can range from 2 mm^2 to 25 mm^2. The area and pin capacities of a chip depend on the package chosen for the chip. For example, a Dual In-line Package (DIP) allows around 64 pins while a Pin Grid Array (PGA) package may allow as many as 300 pins. The cost of a chip is usually proportional to its size, pin count and chosen package. Hence, to reduce the cost of chips, small chip partitions are favored. However, this may increase the chip count. On the other hand, since the off-chip delay is much larger than the on-chip delay, large chip partitions can help in reducing the off-chip delay and in satisfying the performance constraints. A small chip count (large chip partitions) can also increase the reliability of the system, but may suffer from low chip yields.

Therefore, the ideal techniques for our process-level system partitioning problem would be ones which can make the partitioning decisions and tradeoffs described here concurrently. Two such partitioning techniques, an MILP method and a genetic-search method, will be described in the following section and Section 4.5.2 respectively.

4.3 An MILP Partitioning Method

In this section, we will present a mathematical programming method to solve the process-level system partitioning problem described in the previous section. As we have indicated earlier, the mathematical programming method is feasible since there are relatively few objects at the process level and the method gives us the ability to make the partitioning decisions and satisfy the constraints concurrently. In addition, there exist techniques [Pra93, GE92] to improve the run time for solving ILP models by at least an order of magnitude. In fact, Prakash's result [Pra93] is particularly encouraging to us due to the similarity between his model and ours. For instance, the levels of abstraction on which these two models are based are roughly equivalent though the application domains are completely different. Furthermore, several analogies can be drawn between the decisions to be made during Prakash's synthesis and our partitioning.

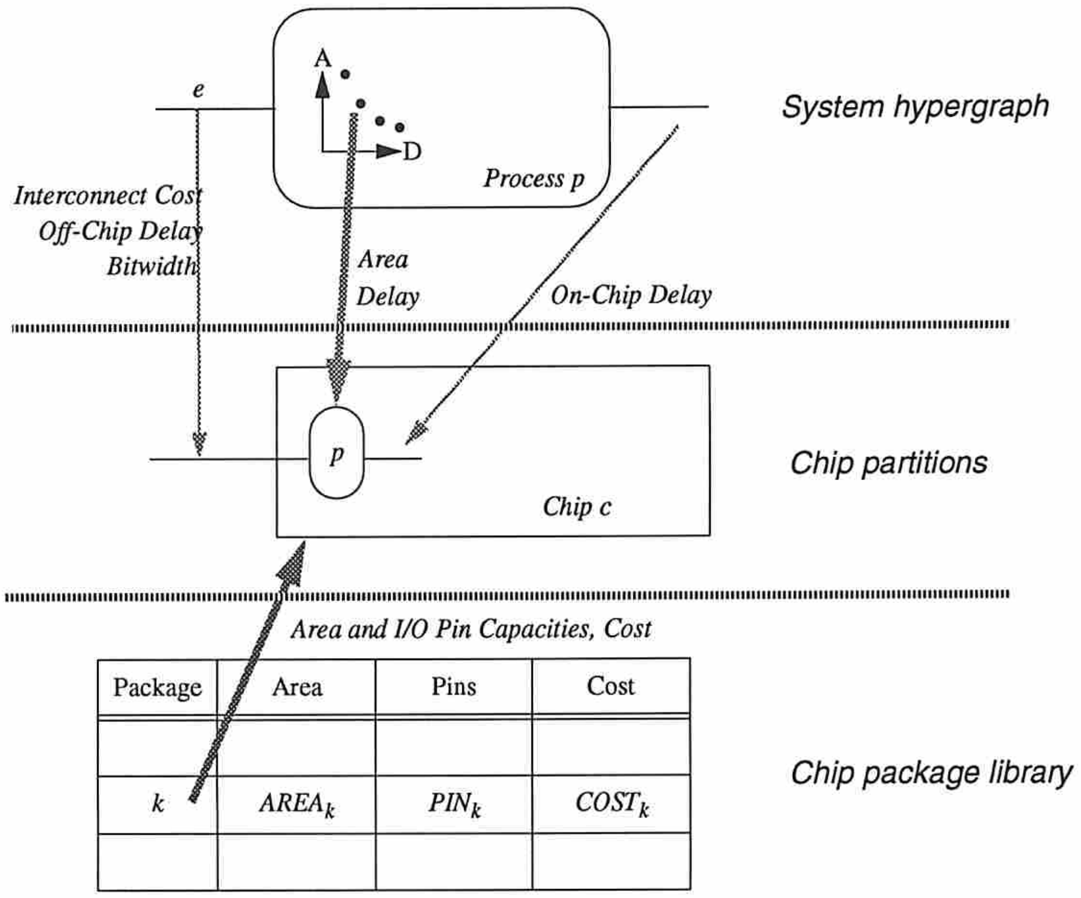
Before describing the detailed formulation, we first give an overview of the partitioning model that our formulation is based on. The notation and objective function will be discussed next, followed by the detailed formulation. The linearization of some non-linear constraints in the formulation will be discussed at the end of this section.

4.3.1 Overview of the Partitioning Model

Figure 4.4 shows various partitioning parameters which will be taken into account in our MILP formulation. First, the design point chosen for a process determines the area consumed at the chip where the process is assigned and the delay added to the signal path that the process involves.

If not all the processes connected by a hyperedge are assigned to the same chip partition, the hyperedge becomes an inter-chip connection. If a hyperedge is an inter-chip connection, it will introduce an off-chip delay to the signal path that it involves and will add an off-chip connection cost to the overall system cost. In addition, it will consume a number of pins (equal to its weight) at every chip which it connects. On the other hand, if a hyperedge is an on-chip connection, it will only introduce an on-chip delay to the signal path that it involves. In Section 4.5.1, we will extend this partitioning model to trade off communication delay, cost and I/O pins.

The package selected for a chip partition determines the area and pin capacities of the chip and contributes a chip packaging cost to the overall system cost. The number of pins required for a chip partition is the sum of the weights of all hyperedges which cross the chip partition boundary. Similarly, the area requirement of a chip partition is determined by the processes assigned to the chip. Both the area



Package	Area	Pins	Cost
k	$AREA_k$	PIN_k	$COST_k$

Figure 4.4: Overview of the process-level partitioning parameters

and pin requirements of a chip partition must meet the capacities imposed by the selected chip package.

Finally, if any performance constraint is imposed on a signal path, the delays introduced by the processes and the hyperedges on this path must satisfy the performance constraint.

4.3.2 Notation

- $P = \{p \mid p \text{ is a process in the system to be partitioned}\}$. α_p and d_p denotes the area and delay of process p. They are constants if p is an already-implemented macro to be used "as is"; otherwise, they are real variables that depends on the design point chosen for p.

- $C = \{c \mid c \text{ is one of the chip partitions}\}$[3]. The number of elements in C can be determined either by the design constraint on the total chip count from the designer or by a reasonable upper bound like the number of processes in the system.

- $K = \{k \mid k \text{ is one of the available chip packages}\}$. The area and pin capacities of k are denoted by $AREA_k$ and PIN_k respectively and its cost is $COST_k$.

- $E = \{e \mid e \text{ is a hyperedge in the system}\}$. If e becomes an inter-chip connection, its cost and estimated delay are denoted by $COST_e$ and D_e^{off}. Otherwise, its estimated on-chip delay is D_e^{on} and it will not introduce an additional cost to the system. The bitwidth of e is denoted by BW_e.

- $P_e = \{p \mid \text{every process } p \text{ connected by the edge } e\}$.

[3]Each partition after system partitioning corresponds to a chip in the system. In the following discussion, we will refer to a chip partition in C by simply saying "a chip c in C".

- $x_{p,c}$: A binary variable which denotes whether or not process p is assigned to chip c.
- $y_{c,k}$: A binary variable which denotes whether or not to select package type k for chip c.
- β_c: A binary variable which indicates whether chip c is needed (not empty) or not.
- b_e: A binary variable which indicates whether edge e becomes an inter-chip or on-chip connection.

In summary, the partitioning variables described previously provide the following information:

- The binary variables $x_{p,c}$ and $y_{c,k}$ give us the process-to-chip assignment and the chip package selection respectively.
- From the set of binary variables β_c we can determine and/or control the final chip count.
- The set of binary variables b_e are used to denote the inter-chip connections as well as the pin requirement of each chip.
- The real variables α_p and d_p of process p, once determined by the selection of a design point, become the area and performance constraints when synthesizing p. α_p can also be used to perform a process-level floorplan before synthesizing the system structure. Thus, more accurate global wiring delays can be available to the synthesis tools.

4.3.3 Objective Function

The objective function we try to optimize here is to minimize the overall cost after system partitioning. It consists of the cost of the new custom chips and the cost of the inter-chip connections. This cost function $\mathcal{F}$ to be minimized can be stated as follows:

$$\sum_{c\in C}\sum_{k\in K} COST_k \times y_{c,k} + \sum_{e\in E} COST_e \times b_e \tag{4.1}$$

Other objective functions can be used as well if necessary. For example, designers may simply want to perform a N-way system partitioning while minimizing inter-chip connections. This can be achieved by the following objective function:

$$\sum_{e\in E} BW_e \times b_e$$

and the following constraint on the total chip count:

$$\sum_{c\in C} \beta_c = N$$

4.3.4 Constraints

The formulation for process-level system partitioning consists of 7 classes of constraints.

4.3.4.1 Process to Chip Assignment

In our partitioning model, each process must be assigned to one and only one chip. This can be achieved by the following constraint using the binary variables $x_{p,c}$:

$$\sum_{c\in C} x_{p,c} = 1 \qquad \forall p \in P \tag{4.2}$$

Hence, one and only one of $x_{p,c}$ will be 1 for each process p.

If there exist processes which cannot be assigned to a single chip using any package in the library while meeting their design constraints, then no solution will be found by this model. In such cases, these processes can be decomposed by operation-level partitioning approaches into a number of smaller synchronous processes before partitioning the system. However, we believe such cases will become less frequent as chip capacities continue to increase.

4.3.4.2 Chip Package Selection

Each chip c in C is needed (not empty) if and only if there are one or more processes assigned to it. This can be stated as:

$$(\beta_c = 1) \Leftrightarrow (\sum_{p \in P} x_{p,c} > 0) \qquad \forall c \in C \tag{4.3}$$

The $\Leftrightarrow$ symbol denotes the *if-and-only-if* relation. Hence, for each chip c, β_c becomes 1 if and only if there is at least one $x_{p,c}$ which is equal to 1.

If a chip is needed, we must select a chip package for it from the library. Therefore, for each chip c, one and only one of binary variable $y_{c,k}$ must be 1 when β_c is equal to 1; otherwise, all $y_{c,k}$ must be 0. This is achieved by:

$$\sum_{k \in K} y_{c,k} = \beta_c \qquad \forall c \in C \tag{4.4}$$

Since the constraint on the total chip count, if given, is reflected in the number of elements in C, the final number of chips that are needed will always be less than or equal to the given constraint. If the designer insists on a particular number of chips after partitioning, it can be achieved by adding the following constraint:

$$\sum_{c \in C} \beta_c = CHIPCOUNT \tag{4.5}$$

where $CHIPCOUNT$ is the final chip count given by the designer.

4.3.4.3 Inter-chip Connection

An hyperedge e will become an inter-chip connection if and only if not all the processes connected by e are assigned to the same chip. This is reflected by the following constraint:

$$b_e = 1 - \sum_{c \in C} \min_{p \in P_e} x_{p,c} \qquad \forall e \in E \tag{4.6}$$

Hence, b_e will be 0 when the term $\sum_{c \in C} \min_{p \in P_e} x_{p,c}$ at the right hand side of this equation becomes 1. Due to the function *min*, the only case that this term is equal to 1 is when there is a chip c such that $x_{p,c}$ is 1 for every process p in P_e[4]. In such case, all the processes connected by e are assigned to chip c. Otherwise, every *min* function will be 0, which means that not all the processes connected by e are assigned to the same chip. Therefore, b_e is equal to 1 and e is an inter-chip connection.

4.3.4.4 Pin Capacity Constraints

Obviously, the I/O pins used at each chip cannot exceed the pin capacity of the selected package for this chip. This can be enforced by

$$\sum_{e \in E} b_e \times BW_e \times \max_{p \in P_e} x_{p,c} \le \sum_{k \in K} PIN_k \times y_{c,k} \qquad \forall c \in C \tag{4.7}$$

The left hand side of this constraint represents the total bitwidth of the inter-chip connections involving chip c. When an edge e is an inter-chip connection involving chip c, both b_e and $\max_{p \in P_e} x_{p,c}$ will be 1. The right hand side of this constraint is simply the pin capacity of the package selected for chip c.

[4]There exists at most one such chip in any case; hence, $\sum_{c \in C} \min_{p \in P_e} x_{p,c}$ will never be greater than 1.

4.3.4.5 Area Capacity Constraints

Like the I/O pin constraints, the area consumed by the processes assigned to a chip cannot exceed the area capacity of the selected package for this chip. This can be constrained by

$$\sum_{p\in P} x_{p,c} \times \alpha_p \leq \sum_{k\in K} AREA_k \times y_{c,k} \qquad \forall c \in C \tag{4.8}$$

Here, α_p is a constant if process p is an already-implemented macro to be used "as is"; otherwise, it is a variable which depends on the design point chosen for p.

4.3.4.6 Timing Constraints

The designer may impose a number of timing constraints on the system. In our model, each timing constraint is associated with a path in the system hypergraph. Each such path is represented by a sequence $p_0, e_1, p_1, e_2, \ldots, p_{t-1}, e_t, p_t$ such that $p_i \in P$ and $e_i \in E$ for all i and $p_{i-1}, p_i \in P_{e_i}$ for $i = 1, \ldots, t$. The timing constraint over this path becomes

$$d_{p_0} + d_{e_1} + d_{p_1} + \ldots + d_{e_t} + d_{p_t} \leq T_t \tag{4.9}$$

where

- T_t is a constant that represents the upper bound delay over this path.
- d_{e_i} denotes the delay of edge e_i for $i = 1, \ldots, t$. Since e_i may become either an on-chip or off-chip connection, d_{e_i} is a variable. Let $D_{e_i}^{on}$ and $D_{e_i}^{off}$ be the estimated on-chip and off-chip delays of e_i respectively. We have the following equation:

$$d_{e_i} = D_{e_i}^{off} \times b_{e_i} + (1 - b_{e_i}) \times D_{e_i}^{on} \tag{4.10}$$

- d_{p_i} is the delay of process p_i for $i = 0, \ldots, t$. It is a constant if p_i is an already-implemented macro; otherwise, it is a variable which depends on the design point chosen for p_i.

4.3.4.7 Exploration of Design Alternatives

As we discussed earlier, each process may have many possible design alternatives to be chosen during partitioning. For processes to be synthesized later, behavioral area-delay estimation tools like BEST [KP93] can be used to predict a number of possible design points from their behavioral specifications.

If local performance and area constraints are imposed on a process whose design alternatives are to be explored during partitioning, the local constraints can be used to trim the design space first. For example, in Figure 4.5, there are only 3 feasible design points left for process p after trimming its design space using the local performance and area constraints.

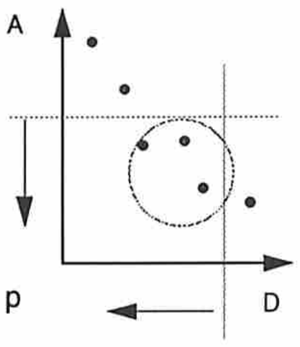


Figure 4.5: Trimming the design space to be explored using local performance and area constraints

If the number of feasible design points for a process p is small, a set of binary variables $z_{i,p}$ is introduced to select one and only one of them. Let I_p be the set of design points for p. We have the following constraint.

$$\sum_{i \in I_p} z_{i,p} = 1 \qquad \forall p \in P \tag{4.11}$$

In addition, the α_p (area) and d_p (delay) of process p become

$$\begin{aligned} \alpha_p &= \sum_{i \in I_p} z_{i,p} \times AREA_{i,p} \\ d_p &= \sum_{i \in I_p} z_{i,p} \times DELAY_{i,p} \end{aligned}$$

However, if the number of design points for a process is large, it will not be feasible to use binary variables for selection. In such case, the design space is approximated by a set of piece-wise linear equations [NW88] as shown in Figure 4.6. For example, let the design space of a process p contains four design

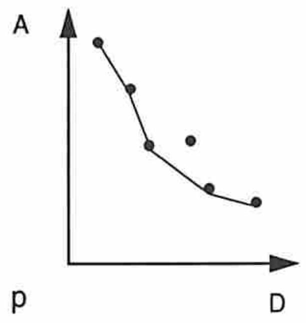


Figure 4.6: Approximating the design space using piece-wise linear equations

points, $\{(A_1, D_1), (A_2, D_2), (A_3, D_3), (A_4, D_4)\}$. We introduce the following three piece-wise linear constraints into the partitioning model:

$$\alpha_p - A_1 \geq \tfrac{A_1 - A_2}{D_1 - D_2} \times (d_p - D_1)$$

$$\alpha_p - A_2 \geq \tfrac{A_2 - A_3}{D_2 - D_3} \times (d_p - D_2)$$
$$\alpha_p - A_3 \geq \tfrac{A_3 - A_4}{D_3 - D_4} \times (d_p - D_3)$$

Since these piece-wise linear constraints involve only real variables, the run time of solving MILP models will only be affected moderately by the number of these constraints.

4.3.5 Linearization

The formulation presented earlier contains some non-linear constraints which have to be linearized in order to be solved by MILP solvers.

Constraint 4.3 contains an *if-and-only-if* relation. It can be replaced by the following linear constraints:

$$\begin{aligned} \beta_c &\geq x_{p,c} \qquad \forall p \in P \\ \sum_{p \in P} x_{p,c} &\geq \beta_c \end{aligned}$$

Hence, if β_c is 0, then the first set of constraints will ensure that $x_{p,c}$ is 0 for all p. If β_c is 1, the second constraint requires that at least one p in P such that $x_{p,c}$ is 1.

The right hand side of Constraint 4.6 contains the following non-linear terms:

$$\min_{p \in P_e} x_{p,c} \qquad \forall c \in C$$

Each of these terms can be replaced by a binary variable $n_{e,c}$ subject to the following constraints:

$$\begin{aligned} n_{e,c} &\leq x_{p,c} \qquad \forall p \in P_e \\ n_{e,c} + (|P_e| - 1) &\geq \sum_{p \in P_e} x_{p,c} \end{aligned}$$

The first set of constraints ensure that $n_{e,c}$ is 0 if there exists a p in P_e such that $x_{p,c}$ is 0. If $x_{p,c}$ is 1 for all p in P_e, the right hand side of the second constraint becomes $|P_e|$; hence, $n_{e,c}$ must be 1.

Similarly, each *max* function in Constraint 4.7 can be replaced by a binary variable $m_{e,c}$ used in the following constraints:

$$\begin{aligned} m_{e,c} &\geq x_{p,c} \quad \forall p \in P_e \\ m_{e,c} &\leq \sum_{p \in P_e} x_{p,c} \end{aligned}$$

The first set of constraints ensures that $m_{e,c}$ is 1 if there exists a p in P_e such that $x_{p,c}$ is 1. If $x_{p,c}$ is 0 for all p in P_e, the second constraint will force $m_{e,c}$ to be 0.

After replacing each *max* term by a binary variable as described above, the left hand side of Constraint 4.7 will contain $b_e \times m_{e,c}$ which can be further replaced by another binary variable $w_{e,c}$ subject to

$$\begin{aligned} w_{e,c} &\leq b_e \\ w_{e,c} &\leq m_{e,c} \\ w_{e,c} + 1 &\geq b_e + m_{e,c} \end{aligned}$$

Finally, each non-linear term $x_{p,c} \times \alpha_p$ in Constraint 4.8 can be replaced by a real variable $v_{p,c}$ confined by

$$\begin{aligned} v_{p,c} &\geq 0 \\ v_{p,c} &\geq \alpha_p - (1 - x_{p,c}) \times BIG \end{aligned}$$

where BIG is a constant with a reasonable large value. In addition, the original cost function $\mathcal{F}$ to be minimized is changed to

$$\mathcal{F} + BIG \times v_{p,c}$$

As a result, if $x_{p,c}$ is 1, $v_{p,c}$ will be set to α_p; otherwise, $v_{p,c}$ will be 0.

4.4 Experiments

A prototype software, ProPart, based on our process-level system partitioning approach has been developed and used to experiment with several examples. It consists of an MILP model generator, an MILP solver and a solution analyzer. The MILP solver is a branch-and-bound program called *Bozo*, written by Hafer [HH90], which invokes a commercial linear programming package, XLP, developed by XMP Software, Inc.

4.4.1 A Mobile Phone System Example

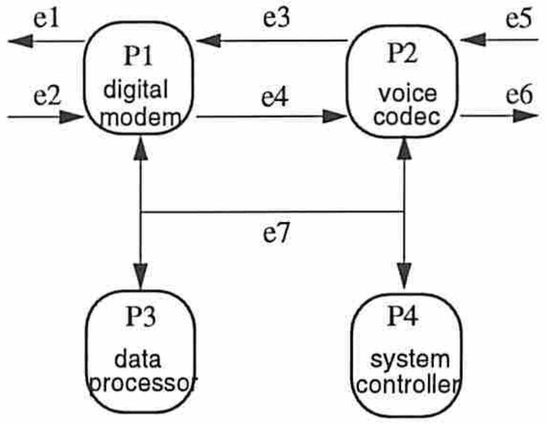


Figure 4.7: A multiple-process system to be partitioned

Figure 4.7 shows a multiple-process system derived from the digital cellular mobile telephone system illustrated in the referenced article [BS92]. In this example, there are four processes and seven edges Three processes, $p1$, $p2$ and $p3$, are assumed to be synthesized later. Their approximated area-delay curves are given in Figure 4.8. Process $p4$ is assumed to be an already-implemented macro whose

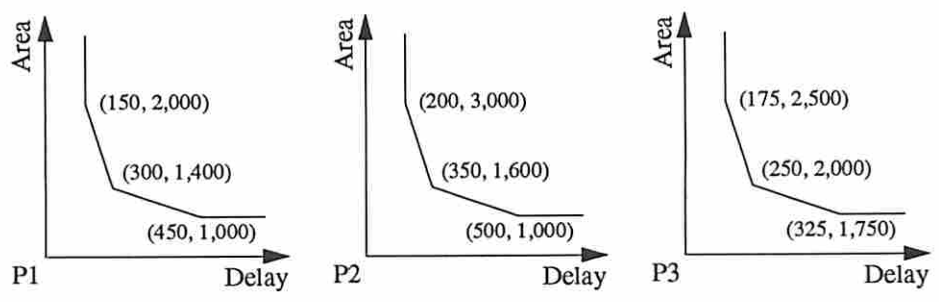


Figure 4.8: The approximated area/delay curves of processes $p1$, $p2$ and $p3$

Edge	Bitwidth	On-chip Delay	Off-chip Delay	Off-chip Cost
$e1, \ldots, e6$	16	10	30	160
$e7$	64	10	30	640

Table 4.1: Edge parameters for the mobile phone example

area is 950 and delay is 200. A performance constraint, 800, is imposed on the path from $p1$ to $p2$. The delay from $p3$ to the inputs of $p1$ or $p2$ through edge $e7$ is limited to 300. Finally, the parameters of these edges are shown in Table 4.1.

The package library shown in Table 4.2 was first used to partition the system. The MILP model produced by ProPart consists of 130 constraints and 64 variables (47 binary variables). Bozo was able to find a minimal-cost solution in few seconds on a Sun SPARCsystem 300. The solution is a single-chip partitioning using

Package	Area	Pins	Cost
$k1$	3000	80	800
$k2$	6000	100	1800
$k3$	9000	120	4000
$k4$	15000	140	6600

Table 4.2: Package Library 1

Process	Area	Delay
$p1$	1400	300
$p2$	1040	490
$p3$	1866.7	290

Table 4.3: Design points chosen for processes $p1$, $p2$ and $p3$

Package	Area	Pins	Cost
$k1$	2000	128	800
$k2$	4000	140	1000
$k3$	5000	150	4000
$k4$	6000	160	6600

Table 4.4: Package Library 2

package $k2$. The design points chosen for processes $p1$, $p2$ and $p3$ are shown in Table 4.3.

In order to show the tradeoff ability of our approach, we modified the package library (shown in Table 4.4) as well as the timing constraints. Specifically, we reduced the area capacities of each package but increased their pin capacities. In addition, we lowered the costs of smaller packages so that multiple-chip solutions could become competitive. We also tightened two timing constraints to 450 and 250 respectively so that processes $p1$, $p2$ and $p3$ would require design points with less delays but larger areas. As we expected, Bozo found a minimal-cost two-chip partitioning in a similar run time. The solution is shown in Figure 4.9.

4.4.2 A Powertrain Control System Example

Similar experiments were also performed on a system derived from the GM powertrain control application [Fuh91] as shown in Figure 4.10. This example contains

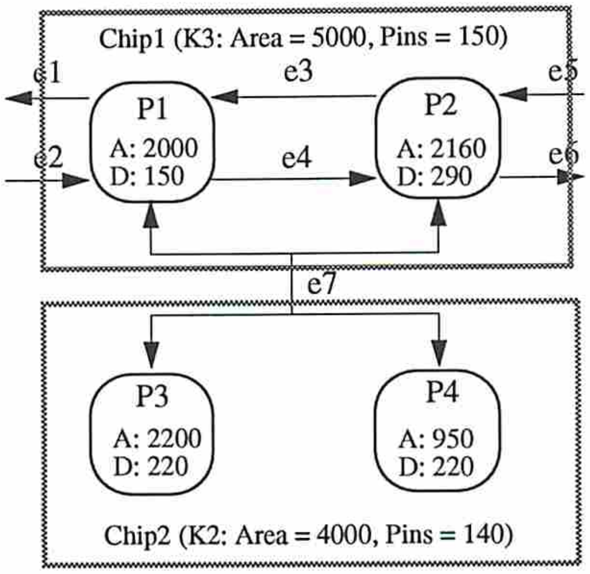


Figure 4.9: A two-chip partitioning for the mobile phone example

10 processes and 17 edges. Five general processes $p1, \ldots, p5$ are assumed to be synthesized later. $u1, \ldots, u4$ are counters and $f1$ is a finite-state machine. Table 4.5 summarizes the parameters used in the experiments. Two experiments were performed using two previous package libraries, Table 4.2 and 4.4, joined with two sets of timing constraints as shown in Table 4.5. The MILP model generated by ProPart consists of 405 constraints and 184 variables (155 binary variables). The minimal-cost solution under package library 1 and the first set of timing constraints was a single-chip partitioning using package $k4$. The design points chosen for processes $p1, \ldots, p5$ are shown in Table 4.6. When the package library 2 and the second set of timing constraints were used, the minimal-cost solution found was a three-chip partitioning as shown in Figure 4.11. In these experiments, feasible solutions were found in few seconds; however, minimal-cost solutions took 6-12 hours on a Sun 4/200 system. We have not made any attempt to optimize the run time of the MILP solver.

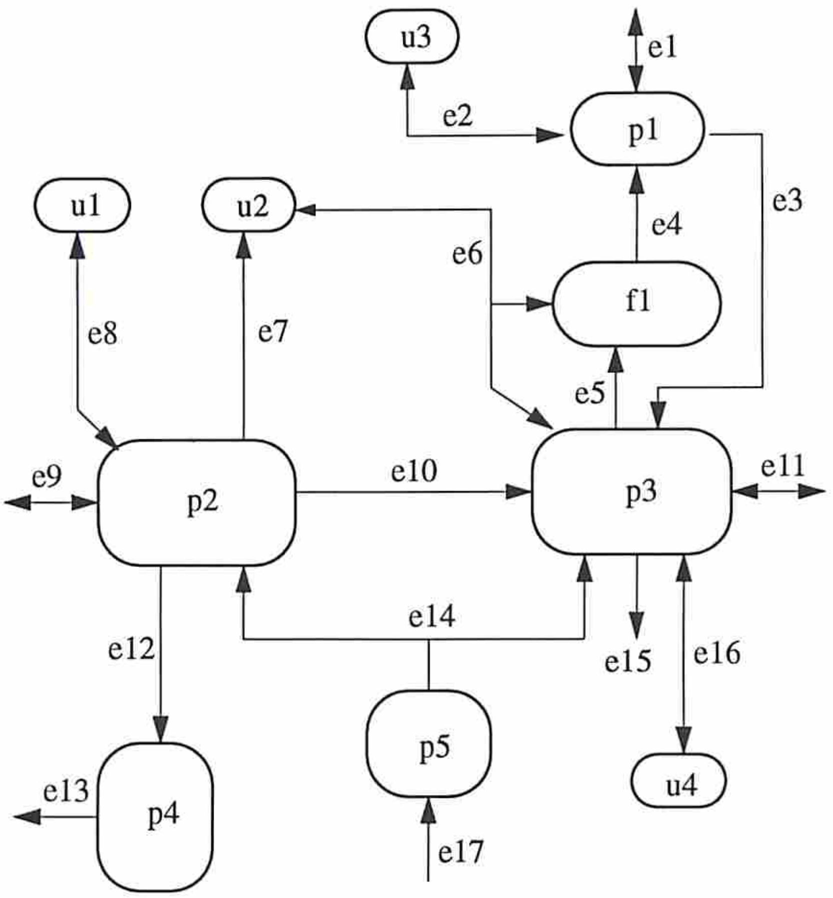


Figure 4.10: An example derived from the GM powertrain application

Process Characteristics

process	area/delay points
u1,...,u4	500/20 (fixed)
f1	1500/40 (fixed)
p1	2000/150 1400/300 1000/450
p2	3000/200 1600/350 1000/500
p3	2500/175 2000/250 1750/325
p4	2500/175 1500/325 1100/475
p5	1900/125 1200/250 950/350

Edge Parameters

Edge	Bitwidth	On-chip Delay	Off-chip Delay	Cost
e1	24	10	30	240
e9	32	10	30	320
e11	32	10	30	320
others	8	10	30	80

Timing Constraints

	Path	1st Set	2nd Set
t1	p2 e10 p3	900	680
t2	p2 e12 p4	850	705
t3	p1 e3 p3	800	580
t4	p5 e14	300	280

Table 4.5: The parameters of the powertrain example

Process	Area	Delay
$p1$	1000	450
$p2$	1000	500
$p3$	1750	325
$p3$	1460	340
$p3$	1100	290

Table 4.6: Design points chosen for the single-chip partitioning of the powertrain example

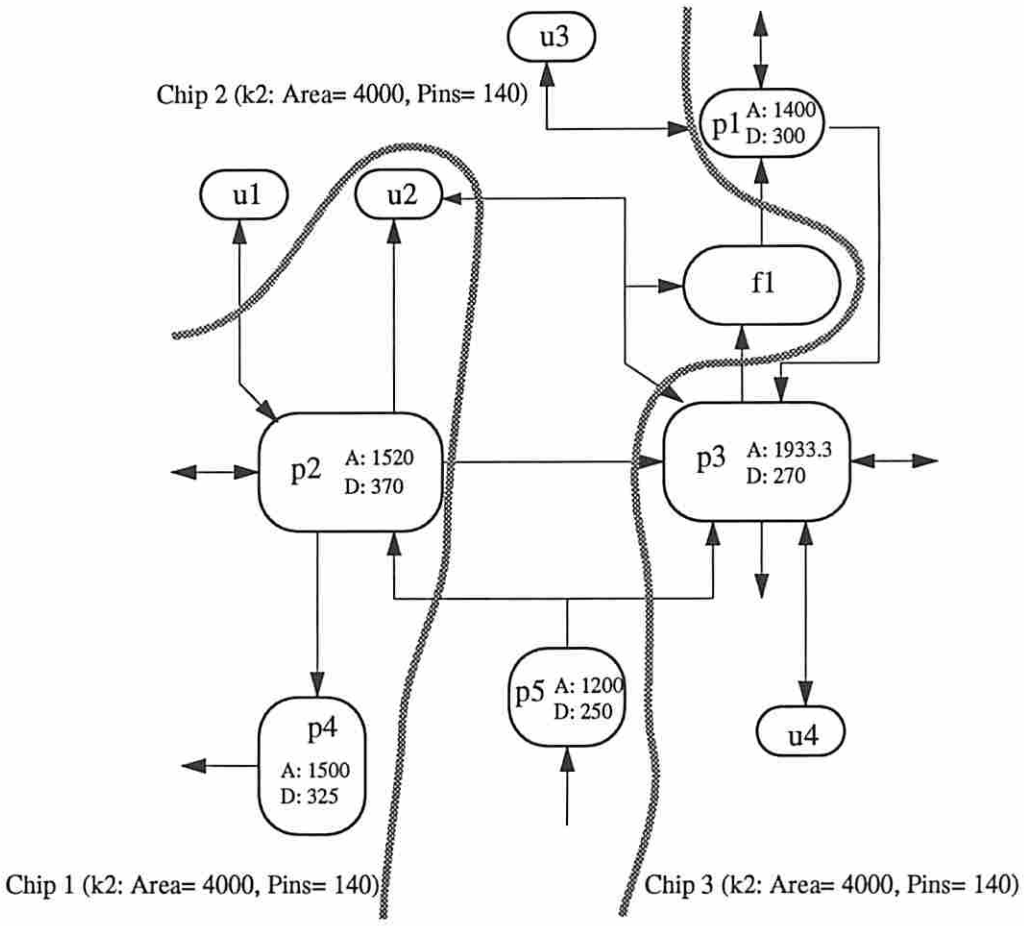

u3
Chip 2 (k2: Area= 4000, Pins= 140)
p1 A: 1400 D: 300
u1
u2
f1
p2 A: 1520 D: 370
p3 A: 1933.3 D: 270
p5 A: 1200 D: 250
u4
p4 A: 1500 D: 325
Chip 1 (k2: Area= 4000, Pins= 140)
Chip 3 (k2: Area= 4000, Pins= 140)

Figure 4.11: A three-chip partitioning of the powertrain example

4.4.3 A JPEG Image Compression System Example

Our system partitioning methodology was applied to the design of a JPEG still-image compression system [Wal91] using a number of the Unified System Construction (USC) tools, including ProPart. The USC project at the University of Southern California involves the production of an integrated set of system-level tools for synthesizing multi-chip, heterogeneous application-specific systems which meet cost, performance and power constraints.

Figure 4.12 shows the input JPEG system specification, the design flow and the output of various tools. Four major synthesis tools were used in this experiment. For data path synthesis, we used SMASH [GP94] (scheduling) and MABAL [KP89] (allocation and binding). The ProPart tool that we discussed in this chapter was used to partition the JPEG system. A multiprocessor synthesis tool called SOS [Pra93] was used to study the architecture tradeoff of the 2D-DCT (Discrete Cosine Transform) function. This experiment has been described in detail in another article [GCDBP94]. We will only discuss the design activities related to the system partitioning here.

In this experiment, we began with the synthesis of the DCT function. The 2D-DCT was first decomposed into repeated row-column 1D-DCTs prior to the application of the synthesis tools (Figure 4.13). The 1D-DCT macro was synthesized first and used to construct a 2D-DCT, clearly a bottom-up step. SMASH was used to generate five schedules with varying cost and performance for a 1D-DCT macro from a behavioral VHDL description of the 8-point DCT described in the referenced article [FLS+92]. These 1D-DCT schedules were then processed by MABAL to generate the RTL datapath netlists. The netlists were analyzed to obtain the area characteristic of the datapaths as shown in Table 4.14. The areas for functional units, multiplexers and registers were determined from the netlists,

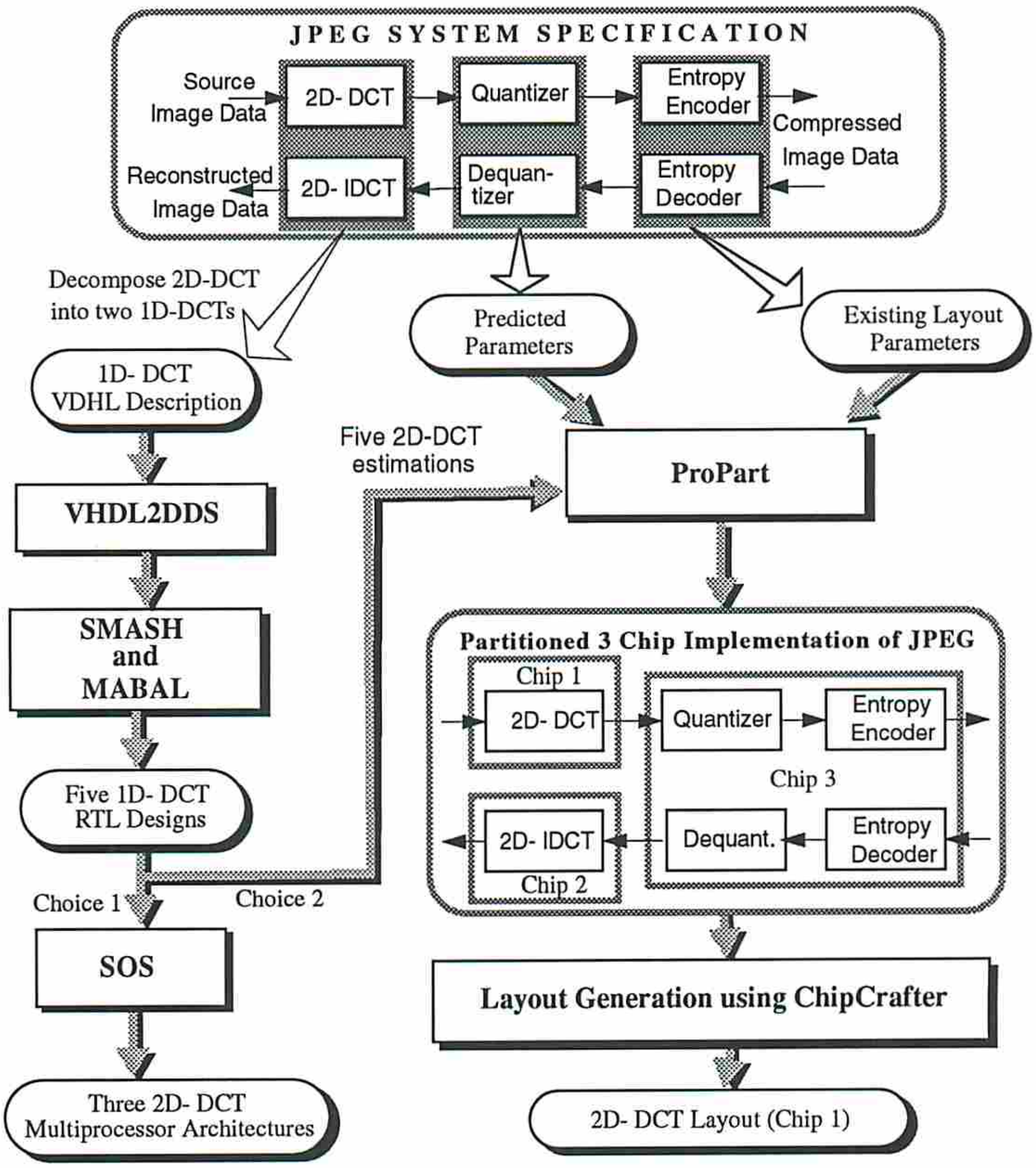


Figure 4.12: Design flow for the JPEG system example

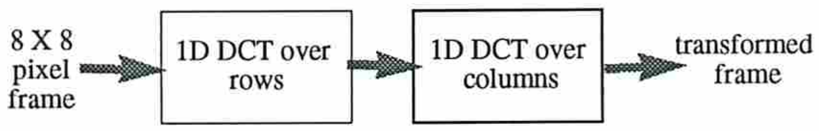


Figure 4.13: Decomposing a 2D-DCT into repeated row-column 1D-DCTs

and wiring area was estimated manually using a rule-of-thumb which we observed in our earlier experiments [5] [PGH91].

Design Number	Functional area ($10^6\ \mu m^2$)	Interconnect area ($10^6\ \mu m^2$)		Total area ($10^6\ \mu m^2$)
		Mux + Registers	Wiring	
	A	B	C = 2(A+B)	
1	29.35	3.74	66.18	99.27
2	19.68	3.87	47.09	70.63
3	17.20	4.05	42.50	63.74
4	9.92	3.67	27.18	40.77
5	5.12	4.12	18.48	27.73

Figure 4.14: 1D-DCT RTL designs from MABAL

Before partitioning the JPEG system using ProPart, we estimated the performance and area of all the functions in the system. A 2D-DCT architecture consisting of two 1D-DCT modules and an 8×8 frame buffer was selected as described in the literature [FLS+92]. The worst-case datapath delay was used to calculate the performance for each design, assuming a two-phase non-overlapping clocking scheme. The quantizer performance and area were estimated similarly,

[5] We did not use our wiring area estimation tools in this experiment due to lack of time.

and the parameters estimated are comparable to those reported in the literature [FLS+92]. We used the parameters of an existing implementation for the Huffman coding [PP93].

(a) Process Characteristics

Process	Estimated Area/Delay points
2d-dct	217096/480 159819/780 146024/900 100106/1200 74004/1560
quan	19036/89 (fixed)
enco	12200/100 (fixed)
deco	12200/100 (fixed)
dequ	19036/89 (fixed)
2d-idct	217096/480 159819/780 146024/900 100106/1200 74004/1560

(b) Package Library

Package	Area	Pins	Cost
k100	84174	260	1214
k209	175528	304	4240
k271	227306	344	6849
k464	389800	452	18923

Note: 1. Area is 10^3 sq. microns.
2. Delay is in nano second.
3. Cost is a function of area and pin capacities.

Figure 4.15: Parameters used by ProPart

After estimating the performance and area of all the functions in the JPEG system, it was partitioned by ProPart. The data for each function in the system is summarized in Table 4.15 (a). As we can see from the table, both the 2D-DCT and IDCT functions consist of five possible design alternatives to be chosen during partitioning. The package library[6] used in this experiment is shown in Table 4.15 (b). The MILP model generated by ProPart consists of 162 constraints and 83 variables (73 binary variables). The minimal-cost partitioning was found in less than 1 minute on a Sun 4/200 system. The solution is a three-chip partitioning as shown in Figure 4.16. The design point selected for 2D-DCT corresponds to the second 1D-DCT design produced by SMASH and MABAL. Note that ProPart placed the DCT and IDCT on separate chips, and lumped the remaining functions onto a single chip. Finally, the layouts of the 1D-DCT macro and 2D-DCT chip

[6]The library data was derived from a commercial ASIC library (LSI LCA300K).

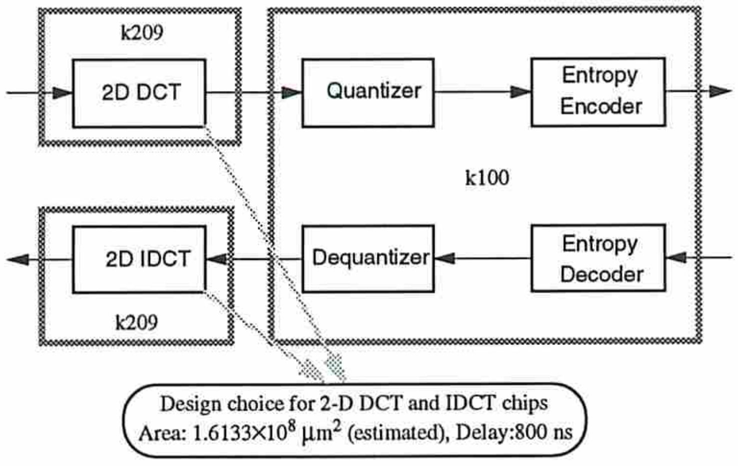

k209
2D DCT
Quantizer
Entropy Encoder
k100
2D IDCT
Dequantizer
Entropy Decoder
k209
Design choice for 2-D DCT and IDCT chips
Area: 1.6133×10^8 μm^2 (estimated), Delay:800 ns

Figure 4.16: A three-chip partitioning of the JPEG system

were generated using Cascade Design Automation's ChipCrafter according to the design point selected by ProPart.

From this experiment, we found that the overall system design flow was not completely top-down. There were some portions where the tools were used in a bottom-up fashion. For example, the use of macro synthesis, then the use of the macro parameters in partitioning, and finally the incorporation of macros into chips clearly show a cycling between chip-level and system-level design tasks. We believe that this mixture of top-down and bottom-up design flows will be common in the design of large digital systems. This is because the most suitable methodologies to implement various system functions or to estimate their parameters are usually different and also vary from one design situation to another. We found that our system partitioning approach does provide this kind of flexibility.

4.5 Extensions

In this section, two extensions to our system partitioning approach will be discussed. First, we will show how to extend our MILP partitioning method to trade off communication delay, cost and I/O bandwidth. A genetic-search partitioning method will also be described.

4.5.1 Tradeoffs of Communication Delay and Hardware

The system partitioning approach described previously assumes that when a hyperedge becomes an inter-chip connection, the number of pins required at each chip connected the edge is fixed. For some data communication between processes, there exist tradeoffs between the communication delays and hardware such as I/O pins and buffers. For example, a 16-bit parallel data transfer between two processes on separate chips could be converted to two 8-bit transfers in order to reduce pins at the expense of a longer communication delay. In this section, we will discuss a way to incorporate communication tradeoffs into our MILP formulation for system partitioning.

Like the exploration of each process's design alternatives as discussed in Section 4.3.4.7, the communication tradeoffs can be achieved by selecting a suitable communication alternative for each hyperedge in the system during partitioning. For each hyperedge e, let I_e be a set of communication alternatives for e. Each point i in I_e defines the off-chip communication delay $D_{i,e}^{off}$, the bitwidth $BW_{i,e}$, the buffer area $BA_{i,e}$ and the cost $COST_{i,e}$ for the hyperedge e when i is selected. The selection of the communication alternatives can be done by using a set of binary variables $z_{i,e}$ such that

$$\sum_{i \in I_e} z_{i,e} = 1 \qquad \forall e \in E \tag{4.12}$$

In our original MILP formulation, the BW_e (bitwidth), D_e^{off} (off-chip delay) and $COST_e$ are constants. They now become variables subject to

$$\begin{aligned} bw_e &= \sum_{i \in I_e} z_{i,e} \times BW_{i,e} \\ d_e^{off} &= \sum_{i \in I_e} z_{i,e} \times D_{i,e}^{off} \\ cost_e &= \sum_{i \in I_e} z_{i,e} \times COST_{i,e} \end{aligned}$$

An additional variable ba_e is introduced for each edge e to denote the buffer area that will be consumed at each chip it connects.

$$ba_e = \sum_{i \in I_e} z_{i,e} \times BA_{i,e}$$

Finally, the area-capacity constraint needs to be modified to reflect the buffer area consumed by the inter-chip communications. Specifically, Constraint 4.8 becomes

$$\sum_{p \in P} x_{p,c} \times \alpha_p + \sum_{e \in E} b_e \times ba_e \times \max_{p \in P_e} x_{p,c} \le \sum_{k \in K} AREA_k \times y_{c,k} \qquad \forall c \in C \qquad (4.13)$$

where the second term at the left hand side of this constraint represents the total buffer area used by all the hyperedges which involve chip c and become inter-chip communications.

Of course, all enhancements and additions to this model increase MILP run time, and so alternative partitioning methods have been explored.

4.5.2 A Genetic-Search Partitioning Method

In this section, we will discuss the application of Genetic Algorithms (GA) to our system partitioning problem in order to find acceptable solutions in a more *manageable* run time than the MILP method presented earlier. GA is a more suitable alternative partitioning method of our MILP method than group-migration techniques like the Kernighan-Lin algorithm. This is because the objects (processes) to be partitioned may not have constant attributes such as area and delay if they have several design alternatives to be explored; hence, it is not clear how to measure the improvement resulted from the movement of such an object across partitions in a group-migration approach. On the other hand, GA is a global optimization technique which can be applied to complex problems whose parameters are inter-dependent and should be considered concurrently .

Overview of Genetic Algorithms

GAs use a computational model inspired by evolution. They encode a potential solution to a specific problem on a simple chromosome-like data structure. GAs are iterative procedures which maintain a *population* of candidate solutions. Each structure in the population is usually represented by a fixed-length binary string which represents a vector of parameters to the objective function. During each iteration, the current population is evaluated, and, on the basis of that evaluation, a new population of candidate solutions is formed using several genetic operators like *selection*, *crossover* and mutation. The selection operator ensures that the expected number of times that a solution is chosen for the next generation is proportional to the "goodness" of the solution. The crossover and mutation operators are used to introduce variation into the new population in order to search other points in the search space. Under the crossover operator, two parent solutions exchange portions of their binary representation to generate new sample points in the search space. After crossover, each bit in the population undergoes mutation with some low probability. The termination of a GA procedure may be triggered by finding an acceptable approximate solution, by fixing the total number of iterations, or some other application-dependent criterion. For an in-depth introduction of GA, readers could examine additional references [Gol89, Whi93].

Encoding of the System Partitioning Problem

Generally, there are only two major components of a GA that are problem dependent; namely, the *solution encoding* and the *evaluation function.* In the view of a GA, the problem to be solved is like a black box with a series of control dials representing different parameters, and the output of the black box is only a value returned by the evaluation function which indicates how well a particular combination of parameter settings solves the given problem. In what follows, we will discuss how to encode our system partitioning problem as a genetic-search problem and demonstrate the idea using a task-independent GA system called GAucsd [SG92].

Parameter Representation

GAs work on the coding of the problem parameters rather than the actual problem. Hence, we need to determine

1. what are the essential parameters that can characterize a valid system partitioning, and
2. how to encode them in a way that the crossover and mutation operators will still generate valid solutions.

In our system partitioning problem, there are two primary partitioning decisions: the process-to-chip assignment and the design point selection. The chip package selection is a secondary decision which can be determined using the best-fit strategy once the primary decisions are made.

The process-to-chip assignment can be represented by a number of assignment parameters. Each process in the system is associated with an assignment parameter whose value denotes the chip partition where the process is assigned. These assignment parameters can be encoded by a sequence of fixed-length bit vectors.

The length of these bit vectors corresponds to the upper bound of the final chip count.

Similarly, the exploration of process's design space can be done by using a selection parameter whose value determines a specific design point from the set of possible alternatives. However, encoding such a selection parameter into a fixed-length bit vector may result in several unnecessary bit patterns. For example, if the set of design alternatives of a process consists of 100 points, we need at least 7 bits to cover this range. This coding consists of 128 discrete values; therefore, there are 28 additional bit patterns. One solution is to randomly duplicate the design points in the set to fill out these unnecessary bit patterns; otherwise, they should be evaluated by default to a worse possible point which will never be selected when compared to others in the set.

For example, the JPEG example given in Section 4.4.3 can be encoded by six 2-bit assignment parameters, one for each function in the system, and two 3-bit selection parameters for exploring the design alternatives of DCT and IDCT. Figure 4.17 shows the representation of a JPEG partitioning solution under this coding.

Assignment Parameters						Selection Parameters	
DCT	Quantizer	Encoder	IDCT	Dequantizer	Decoder	DCT	IDCT
00	01	01	10	01	01	001	001

Figure 4.17: The parameter coding of the JPEG example

Since the parameters are encoded by a sequence of non-overlapping fixed-length bit vectors, applying crossover and mutation operators to any position in such a coding structure will always produce bit strings which represent valid partitioning solutions. However, a valid partitioning solution is not necessarily a feasible solution which meets the design constraints. The search for feasible partitioning solutions has to be done through the evaluation function.

Evaluation Function

In a GA, the evaluation function provides a measure of performance (or cost) with respect to a particular combination of parameters which represents a point in the search space. In terms of our system partitioning problem, this evaluation function not only has to reflect the partitioning cost as discussed in Section 4.3.3 but also need to guide the genetic search away from the solutions which violate the design constraints. Hence, we define this function $\mathcal{F}$ as follows:

$$\mathcal{F} = \sum_{c \in C} COST_c + \sum_{e \in E} COST_e + \sum_{t \in T} W_t \times (EXCESS_t)^2 \tag{4.14}$$

where

- $COST_c$ is the cost of packaging chip partition c,
- $COST_e$ is the cost when a hyperedge e becomes an inter-chip connection,
- $EXCESS_t$ is the amount of violation of a timing constraint t imposed on the system, and
- W_t is a constant weight which indicates the relative importance of violating timing constraint t.

$COST_e$ and $EXCESS_t$ can be calculated directly from the coding structure which encapsulates the parameters for the process-to-chip assignment and the design point selection. This is because for each particular combination of these parameters the set of hyperedges which cross chip partitions can be identified and the delay of each element on a signal path with a timing constraint can be determined.

$COST_c$, however, depends on not only the process-to-chip assignment and the design point selection but also the available chip package options specified in the given library. This can be done by incorporating a best-fit chip-package selection into the evaluation function. In other words, the available chip packages in the library are sorted according to their costs. For each chip partition in a candidate solution which is not empty, the chip package which meets the area and pin requirements and costs the least is selected. If there is no chip package in the library which can satisfy the area or pin requirements of a chip partition c, a large value is given to $COST_c$ to make this solution unfavorable during the genetic search.

Since the amount of violation of timing constraints is reflected in the cost function to be minimized, the solutions found will most likely be feasible. However, the GA does not guarantee to find a feasible solution which will meets all the timing constraints if the initial population is chosen completely at random. Hence, we can either seed the initial population with some feasible solution found manually or using an other heuristic technique. Alternatively, we can increase the size of the population so that the initial solutions selected randomly will provide enough variance to cover potential search paths toward feasible solutions.

Experiment

To validate the GA partitioning method described here, we experiment with the JPEG system example given in Section 4.4.3 using GAucsd [SG92], a task-independent GA system. In this experiment, the problem of partitioning the JPEG system was encoded into a C-language evaluation function using the coding structure shown in Figure 4.17. We used a population size of 64, crossover rate of 43%, mutation rate of 0.53% and 2040 generations. The same 3-chip partitioning solution given by our MILP method as shown in Figure 4.16 was produced by GAucsd in less than 3 seconds on a HP 9000/720 workstation.

Remarks

We found that the genetic algorithm is a promising optimization technique for the system partitioning problem not only because it can provide acceptable solutions in less run time than our MILP method but also because it will allow us to handle issues like yield and power which are difficult to incorporate into the MILP formulation due to their non-linearity. For example, to partition a system into an MCM with an acceptable yield, the yield of the MCM is a non-linear function of the yields of the attached dies, where the yield of a die is also an non-linear function of the die size. This partitioning problem can be encoded into a genetic-search problem whose evaluation function performs these non-linear calculations of MCM and die yields.

4.6 Summary

In this chapter, we have presented a system-level partitioning approach to partition a system at the process level, to explore each process's design alternatives, to determine proper chip count, and to consider chip packaging options concurrently. An MILP partitioning method was given and implemented in a prototype tool called ProPart. Several experiments including a JPEG image compression system were performed to demonstrate the usefulness of this tool. Two extensions were discussed including the communication tradeoffs during partitioning and a genetic-search partitioning method.

We believe that system partitioning at a higher level of granularity such as processes and procedures will become more and more advantageous and necessary as both chip capacities and system complexity keep increasing. For future development, our system partitioning approach should be extended in two direction: non-uniform technology and mixed packaging devices. The first issue deals with mapping system functionalities onto a number of interconnected components, which may be ASICs, pre-designed parts or programmable devices. These components in turn can be distributed among mixed packaging devices such as chips, multi-chip modules (MCM) and boards in order to satisfy or optimize the constraints on cost, size, yield, power, and other design characteristics.

Chapter 5

Synthesis of Systems with Unbounded-Delay Operations and Communicating Processes

5.1 Introduction

Though high-level synthesis of digital hardware has received enormous attention over the years, most approaches have focused on the synthesis of single-process designs. Under these approaches, the design specification is usually represented by a single graph which captures the essential data and control flow of the design behavior, and synthesis tasks such as scheduling and module allocation/binding are applied to the operations of this graph. Additionally, an important assumption is often made in previous approaches is that all the operations in the graph have *fixed* execution delays.

In practice, we find that complex application-specific systems often consist of multiple concurrent and interacting processes. For example, three GM production designs illustrated in [Fuh91] contain from 4 to 10 concurrent processes. Synthesizing a system with multiple concurrent processes poses new challenges to synthesis tools. First, the processes may need to interact with the external environment or with each other. Due to the I/O and inter-process communication, the synthesis of each process often involves detailed timing constraints as well as operations with *unbounded* delays; i.e., delays are unknown at the compile time. Second, the synthesis tool has to concurrently solve all the timing constraints imposed by one process on another in order to synchronize all the processes in the system. Also, since the processes on a chip are competing for chip resources such as area and pins and since the total resources on a chip are limited by the chip package, resource allocation for each process should be done by trading off the performance and resource requirements of all the processes on the chip. Finally, if we synthesize one process at a time, the decisions made previously may affect and constrain the synthesis of other processes in the system, which may result in an inferior system implementation.

Traditional approaches for designing multiple-process systems are to synthesize individual processes separately. The integration and synchronization of the processes in the system are usually done manually by the designers. For example, System Architect's Workbench (SAW) [TLW+90], a single-process synthesis system, was used in the design of three industrial applications as mentioned in [Fuh91]. In these design experiments, each process was described in a separate ISPS or Verilog file and synthesized individually. The synthesized netlists of the processes were then manually interconnected. The I/O and inter-process communication was specified manually by the designers.

There is a class of work [TW93, Nes87, Hay90] which address the issue of I/O and inter-process communication as a separate problem, known as *interface synthesis*, from the data path synthesis. These techniques, however, are mostly applicable to control-dominated designs with little or no data computation. For the synthesis of more general designs, Ku introduced a technique called *relative scheduling* [KM92] which can handle operations with unbounded delays under detailed timing constraints. In this approach, the start time of an operation is specified in terms of offsets from a set of *anchors* (unbounded-delay operations). Since relative scheduling requires module allocation/binding to be performed before scheduling, we found it not very suitable for the synthesis of multiple processes under global resource constraints, e.g., the area capacity of a chip. In addition, the control scheme used by this approach is quite complicated.

In this chapter, we will present an synthesis approach which is not only suitable for the synthesis of designs with unbounded-delay operations under detailed timing constraints but also applicable to the synthesis of systems with multiple communicating processes. Unlike Ku's relative scheduling, our approach allows us to trade off between performance and resource requirements of the processes during scheduling, to satisfy the timing constraints, and to synchronize the inter-process communication, despite the presence of unbounded-delay operations. Furthermore, the control overhead can be reduced in our approach.

In what follows, we first give an overview of Ku's relative scheduling and discuss its limitations in Section 5.2. Then in Section 5.3 we present our approach for synthesizing designs with unbounded-delay operations. Our approach is based on the observation that each process generally corresponds to one thread of control and there exists a sequential order among the unbounded-delay operations in the process description. By preserving this order, scheduling of single-threaded processes can still be done statically in terms of control steps despite the presence of unbounded-delay operations. Consequently, many good synthesis techniques originally developed for designs with only fixed-delay operations can still be utilized in our approach with some modifications. An ILP scheduling method as well as some experimental results are also given in this section. In Section 5.4, we extend our approach to handle systems with multiple communicating processes, where two additional issues need to be addressed; namely, inter-process communication and global resource allocation. Finally, in Section 5.5 we describe a heuristic approach modified from *freedom-based scheduling* to meet our scheduling requirements.

5.2 Overview of Relative Scheduling

As we have discussed earlier, traditional scheduling approaches assume fixed execution delays for the operations in the design behavior. These techniques are not suitable for the synthesis of many real-time ASIC designs that involve detailed timing constraints and operations with unbounded delays.

In [KM92], Ku presented a technique called *relative scheduling* that supports operations with fixed and unbounded delays. In relative scheduling, the scheduling problem is given in the form of a directed constraint graph $G(V, E)$, where the vertices V represents the operations and the edges E denotes represents the dependencies. Each edge (v_i, v_j) is associated with a weight $w_{i,j}$ that defines either the upper or lower bounds between the execution of v_i and v_j. In the following, some notation and definitions used by Ku in his article [KM92] are given to briefly illustrate relative scheduling.

Definition 5.1 *The* anchors $A \subseteq V$ *of a constraint graph* $G(V, E)$ *are the source vertex* v_0 *and all vertices with unbounded delay.*

The anchors will serve as reference points for specifying the start time of operations in relative scheduling.

Definition 5.2 *The* anchor set *of a vertex* v_i *is a subset of anchors* $A(v_i) \subseteq A$ *such that* $a \in A(v_i)$ *if there exists a path in the* forward *constraint graph* G_f[1] *from* a *to* v_i *containing at least one edge with unbounded weight.*

In other words, an anchor a is in the anchor set of a vertex if the vertex can begin execution only after the completion of a. Furthermore, the anchor set of a vertex represent the unknown factors that can affect the activation time of the vertex (operation).

[1] G_f is obtained by removing the backward edges (maximum timing constraints) from the original constraint graph G.

Definition 5.3 *The start time of a vertex v_i, denoted by $T(v_i)$, is recursively defined as follows:*

$$T(v_i) \equiv \max_{a \in A(v_i)} \{T(a) + \delta(a) + \sigma_a(v_i)\}$$

where $T(a)$ is the start time of anchor a, $\delta(a)$ is the execution delay of a, and $\sigma_a(v_i)$ is the offset of v_i with respect to a.

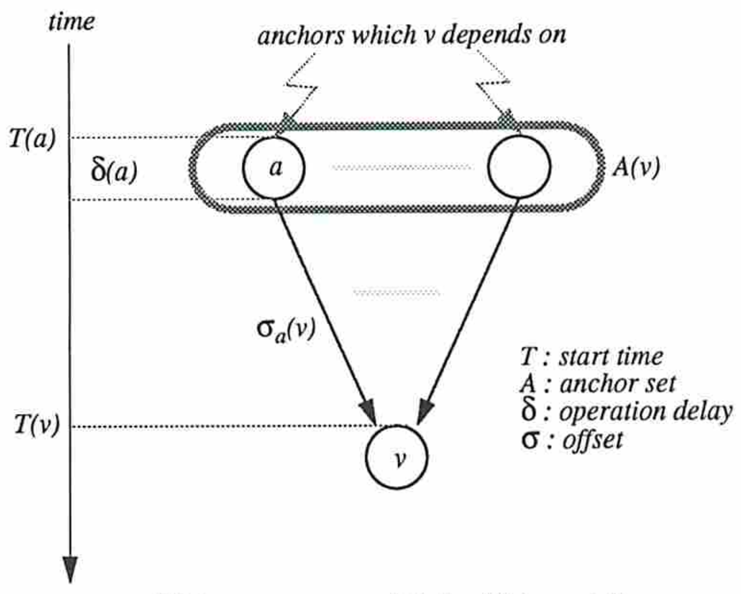
The offset $\sigma_a(v_i)$ defines the amount of time that v_i has to wait after the completion of a.

Definition 5.4 *A* relative scheduling *Ω of a constraint graph $G(V, E)$ is the set of offsets of each vertex $v_i \in V$ with respect to each anchor in its anchor set $A(v_i)$ such that all the timing constraints can be satisfied.*

The execution model of relative scheduling is illustrated in Figure 5.1. A vertex v can begin execution only after all the anchors in its anchor set $A(v)$ are completed. When an anchor $a \in A(v)$ is completed, it sends a signal to an counter or shifter which delays $\sigma_a(v)$ amount of time before sending a completion signal to vertex v. The vertex v is activated after it receives a completion signal from every anchor that it depends on.

Remarks

An important assumption made in Ku's relative scheduling technique is that module allocation/binding have been performed prior to scheduling. Any conflict caused by the assignment of multiple operations to a single module must be resolved in advance by adding proper sequencing dependencies among these operations. Without scheduling information, these serialization decisions made for module sharing may result in inefficient use of hardware resources or poor performance. For the synthesis of multiple-process systems under global resource



$$T(v) = max_{a \in A(v)}(T(a) + \delta(a) + \sigma_a(v))$$

Note: anchors are operations with unbounded delays such as waiting for an event to occur

Figure 5.1: The execution model of relative scheduling

constraints, techniques that combine scheduling with resource allocation are generally preferred so that the performance and resource requirements of each process can be traded off.

Furthermore, the control scheme for the hardware architecture targeted by relative scheduling is fairly complicated. This is because the control logic is implemented as an interconnection of finite-state machines (FSM), one for each *state* vertex[2] of the constraint graph [MK88]. These FSMs are called *control elements.* A control element interacts with others via handshaking signals. Two handshaking signals, *enable* and *done*, are defined to indicate when a control element is enabled and when it has finished. The enable signal for an operation is logical conjunction of the done signals generated by the completion of the anchors that the operation depends on and delayed by appropriate amount of time (offsets) using counters or shifters. For example, Figure 5.2 shows a counter-based approach for an operation v that depends on two anchors a and b with offsets 2 and 3 respectively. This fine-grain control is inevitable in relative scheduling due to the execution of operations with respect to their anchor sets as described earlier.

The "processes" in Ku's model are different from the traditional view of processes. In his model, a process is defined by a constraint graph whose *anchors* are only partially ordered. Therefore, there may be more than one anchor whose completion time is unknown at any given time. This makes the execution times of the associated operations undeterministic. That is a major reason that module binding has to be done before scheduling in Ku's approach and the control scheme has to be so complicated.

[2]State vertices are those operations that require at least one cycle for execution.

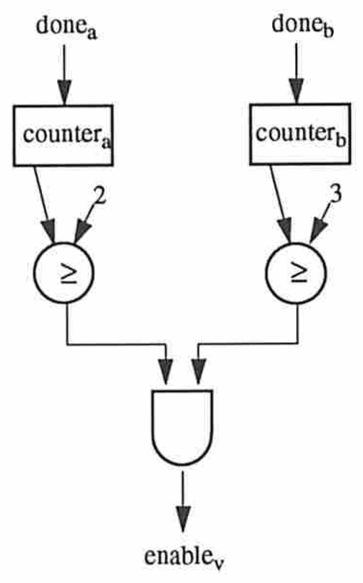


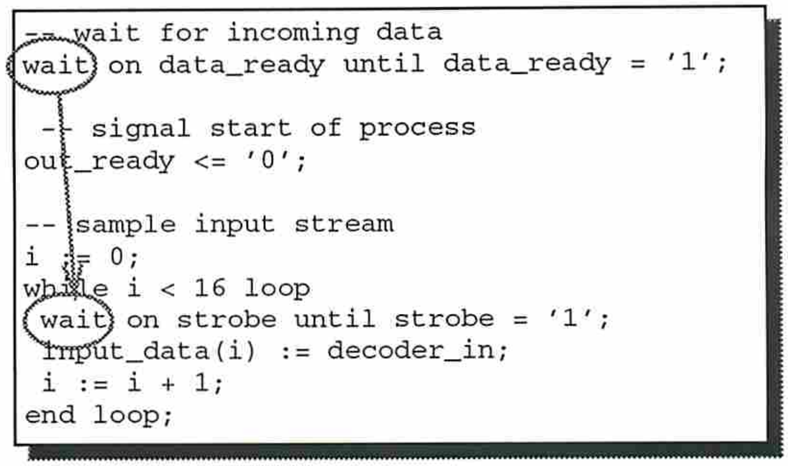
Figure 5.2: An example of the counter-based control in relative scheduling

5.3 Synthesis of Single-Threaded Processes

In this section, we will present a synthesis approach which allows us to trade off performance and resource requirements during the synthesis of designs with unbounded-delay operations while reducing the cost of control as well.

Observations

In general, each process described in a design specification corresponds to one thread of control in designers' minds as well as in simulation models. The execution of a process proceeds from one waiting state (an anchor) to another in a sequential order that is explicitly given in the process description. In addition, the execution between two consecutive waiting states is *deterministic*. For example, Figure 5.3 shows a portion of VHDL specification of a decoder process in an error-correction system. In this description, two *wait* statements are expected to and actually have



```
-- wait for incoming data
wait on data_ready until data_ready = '1';

 -- signal start of process
out_ready <= '0';

-- sample input stream
i := 0;
while i < 16 loop
 wait on strobe until strobe = '1';
 input_data(i) := decoder_in;
 i := i + 1;
end loop;
```

Figure 5.3: An example of anchor ordering in a process description

to be executed sequentially even though there is no data dependency between them.

We found that both the scheduling of designs with unbounded-delay operations and the required control scheme can be simplified if we preserve the anchor order embedded in the process description. Briefly speaking, our idea is to schedule each anchor to an *exclusive* control step in which the process execution will stay until the anchor is completed. In addition, the order of these control steps *matches* the anchor order given in the process description. The remaining operations are scheduled to meet the performance requirement, the resource availability and the timing constraints.

Although one may argue that a certain degree of parallelism may be undiscovered when keeping the anchor order, we believe that the ability to trade off the performance and resource requirements during scheduling is preferable to Ku's relative scheduling for the synthesis of multiple-process systems under global design constraints.

5.3.1 Scheduling Approach

Traditional scheduling approaches assume that operations with "fixed" delays are assigned to a sequence of "equal-length" control steps. We call this kind of approach *static scheduling*. Numerous static scheduling techniques have been proposed, including ILP-based, rule-based, and heuristic methods. They can be further classified into scheduling under performance constraints, scheduling under resource constraints and simultaneously scheduling and resource allocation. Unfortunately, these kind of scheduling approaches are not applicable to designs with unbounded-delay operations.

In this section, we will introduce a scheduling approach based on the idea of *single-threaded* processes. An advantage of this approach is that scheduling can be done statically in terms of a sequence of "variable-length" control steps; hence, most of the existing static scheduling techniques, under some modifications, will become applicable to designs with unbounded-delay operations. In the following discussion, we will try to use Ku's notation as much as possible for consistency.

Definition 5.5 *A constraint graph $G(V, E)$ is* single-threaded *if its* anchors *A are an ordered list $\{a_0, a_1, \ldots, a_K\}$, where a_0 is the source vertex v_0 and there exists a path from a_{i-1} to a_i in G for $i : 1 \ldots K$.*

In other words, the anchors of a single-threaded G are sequentialized by either data dependencies or explicit sequencing edges among them according to the order of their appearance in the process description. If there are anchors which can be executed concurrently, this explicit sequentialization of the anchors could represent some performance penalty as compared with the relative scheduling approach.

In our model, a schedule of a single-threaded $G(V, E)$ is defined by a ordered list of control steps, $\{C_0, C_1, \ldots, C_N\}$, where $length(C_i)$, $i : 0 \ldots N$, denotes the duration of control step C_i. The execution of G proceeds sequentially and cyclically from C_0 to C_N. Note that the control steps may have different lengths.

Definition 5.6 *A* schedule *of a single-threaded $G(V, E)$ is an integer labelling $\sigma : V \rightarrow N$ such that $\sigma(v)$ is in the range of $[0 \ldots N]$ for all $v \in V$.*

Obviously, for a schedule to be valid, all the dependencies and timing constraints must be met.

Normally, the execution of a process stays in an unbounded-delay operation when it is activated, and the process execution proceeds to other operations only after the completion of the unbounded-delay operation. This is translated into the following definition in our model:

Definition 5.7 *For all anchors a of a single-threaded $G(V, E)$, a is scheduled to an* exclusive *control step $C_{\sigma(a)}$ such that $length(C_{\sigma(a)})$ is equal to $delay(a)$ which is unbounded. The schedule of the anchor set A of G is $\{C_{\sigma(a_0)}, C_{\sigma(a_1)}, \ldots, C_{\sigma(a_K)}\}$ such that $\sigma(a_0) = 0$ and $\sigma(a_0) < \sigma(a_1) < \ldots < \sigma(a_K)$.*

The scheduling of the anchor set A of a process actually divides the scheduling space into $K + 1$ zones to which non-anchor vertices can be scheduled (see Figure 5.4). Formally, a zone is a range of control steps defined by the following equation:

$$zone(i) = \begin{cases} [C_{\sigma(a_i)+1}, \ldots, C_{\sigma(a_{i+1})-1}] & \text{if } 0 \leq i < K \\ \left[C_{\sigma(a_K)+1}, \ldots, N\right] & \text{if } i = K \end{cases}$$

The duration of each zone is a variable which depends on the performance requirement, the resource availability, and the timing constraints among those operations.

For designs with conditional branching and loop constructs, the scheduling approach described in this section can be used as the basis for hierarchical scheduling. In other words, scheduling is applied hierarchically in a bottom-up fashion, where the body of a loop is another constraint graph of lower hierarchy and each branch of a conditional construct is also a constraint graph.

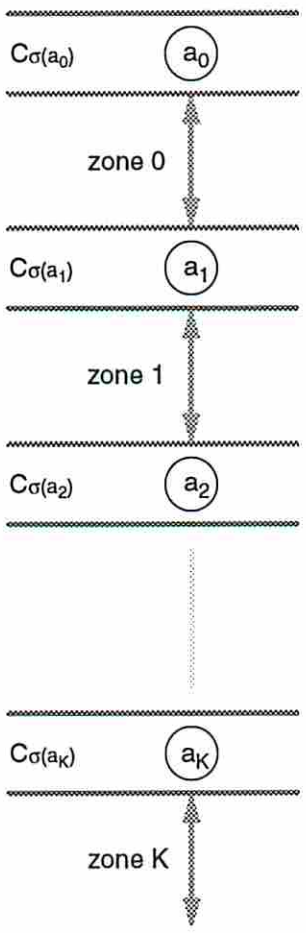


Figure 5.4: The scheduling of the anchor set of a single-threaded process

5.3.2 Timing Constraints

A major complication when synthesizing designs with unbounded-delay operations is to meet the timing constraints no matter how long these unbounded-delay operations will actually take. Generally, timing constraints are used to define the upper and lower bounds between the execution of two operations.

Since the delays of the anchors are unbounded, it is only meaningful to define a timing constraint from the end time of the first operation to the start time of the second one. In our model, the start time of a vertex v is defined as follows:

$$Ts(v) \equiv \sum_{i=0}^{\sigma(v)-1} length(C_i) + \sum_{\forall v_j \in pred(v) \wedge \sigma(v_j)=\sigma(v)} delay(v_j)$$

The first summation is the time when the control step $C_{\sigma(v)}$ starts, and the second one denotes the total delay taken by v's predecessors which are also scheduled to $C_{\sigma(v)}$ (chaining). The end time of v can be stated as follows:

$$Te(v) \equiv Ts(v) + delay(v)$$

Definition 5.8 *A* minimum *timing constraint between two operations v_i and v_j is defined by a lower bound $l_{ij} \geq 0$ such that*

$$Ts(v_j) \geq Te(v_i) + l_{ij}$$

Similarly, a maximum *timing constraint is defined by a upper bound $u_{ij} \geq 0$ such that*

$$Ts(v_j) \leq Te(v_i) + u_{ij}$$

If the given timing constraints are inconsistent, they may be not satisfiable under any schedule. This is even more important in cases involving unbounded-delay operations. For minimum timing constraints, their satisfiability is only affected by the lower-bound delays of the unbounded-delay operations. This is illustrated in Figure 5.5. In this example, the lower-bound delay of anchor a is 1 cycle; hence, the

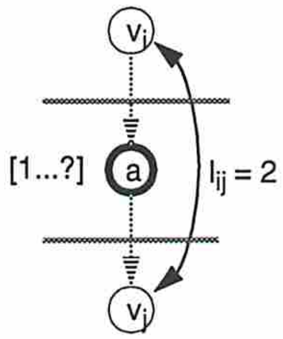


Figure 5.5: A minimum timing constraint satisfied under an unbounded-delay operation

separation between v_i and v_j in this schedule will be at least 2 cycles as required by the minimum timing constraint l_{ij}.

However, a maximum timing constraint u_{ij} can be easily unsatisfiable if there exists an anchor in the path from v_i to v_j as shown in Figure 5.6. This is because the

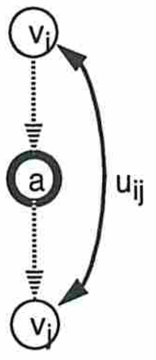


Figure 5.6: An unsatisfiable maximum timing constraint

separation between v_i and v_j depends on $\delta(a)$, the delay of a, which is unbounded. Therefore, there exists a value of $\delta(a)$, e.g. $u_{ij} + 1$, to make $Ts(v_j)$ larger than

$Te(v_i) + u_{ij}$. The following lemma can be proved in a similar way for checking the satisfiability of maximum timing constraints in a single-threaded process.

Lemma 5.1 *For all maximum timing constraints u_{ij} of a single-threaded $G(V, E)$, u_{ij} is* feasible *if and only if there is no path from v_i to v_j which goes through an anchor other than v_i and v_j themselves.*

Additionally, a maximum timing constraint u_{ij} imposes restrictions on the scheduling of v_i and v_j. This is illustrated in Figure 5.7, where v_i is either the anchor a_x or a fixed-delay operation scheduled to $zone(x)$. Since the separation from $Te(v_i)$ to $Ts(v_j)$ cannot be larger than u_{ij}, v_j can not be scheduled later than a_{x+1}. Otherwise, the satisfiability of u_{ij} will depend on the delay of a_{x+1}.

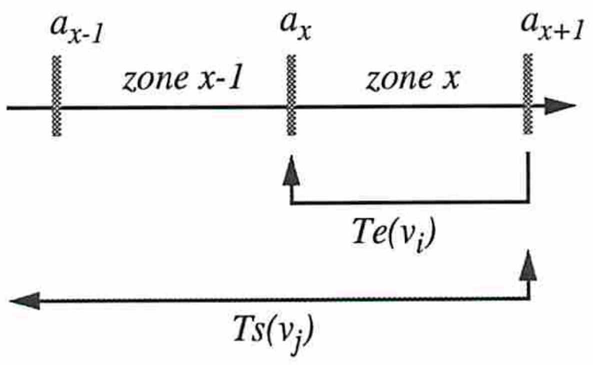


Figure 5.7: Scheduling restrictions imposed by u_{ij}

These scheduling restrictions are formally described as follows:
Let $\kappa : V \rightarrow N$ be the function defined as follows:

$$\kappa(v) = \begin{cases} k & \text{if } v \text{ is an anchor } a_k \\ z & \text{if } v \text{ is not an anchor and } v \text{ is scheduled to } zone(z) \end{cases}$$

Theorem 5.2 *For each feasible maximum timing constraints u_{ij} of a single-threaded $G(V,E)$, if v_j is an anchor then $\kappa(v_j) \leq \kappa(v_i) + 1$; otherwise, $\kappa(v_j) \leq \kappa(v_i)$.*

Proof: Let $\kappa(v_i) = x$ and $\kappa(v_j) = y$.

Case I. v_j is an anchor.

Hence, v_j corresponds to a_y in A. Assume $y > x + 1$. Then, $C_{a_{x+1}}$ is between $zone(x)$ and C_{a_y}. Since $\kappa(v_i) = x$, v_i is scheduled in either C_{a_x} or $zone(x)$. Therefore, the gap between $Te(v_i)$ and $Ts(v_j)$ depends on the delay of a_{x+1} which is unbounded. u_{ij} cannot be satisfied in general. Therefore, $y \leq x + 1$.

Case II. v_j is not an anchor.

Hence, v_j is scheduled to $zone(y)$. Assume $y > x$. Then, C_{a_y} is between $zone(x)$ and $zone(y)$, which means that the separation between $Te(v_i)$ and $Ts(v_j)$ depends on the delay of anchor a_y. u_{ij} cannot be satisfied in general. Therefore, $y \leq x$. □

5.3.3 Resource Allocation

In our scheduling approach, although the control steps to which the anchors are scheduled have unbounded lengths, the order of control steps $\{C_0, C_1, \ldots, C_N\}$ being executed is *deterministic*. Hence, the life span of operations and values still can be determined statically in terms of control steps. The life span of an operation or value is defined by the control step where it starts (or is created) to the one where it ends (or is no longer needed).

The ability to determine the life span of operations and values allows us to determine or even control the resource requirements at each control step during scheduling. For example, two operations of a scheduled single-threaded $G(V,E)$ can share a single module if there is no overlap between their life spans or they are mutual exclusive. Also, we can schedule less critical operations to those control steps whose resources are under-utilized.

Since resource sharing can be done statically in terms of control steps in our scheduling approach, we are able to trade off performance and resource requirements, despite the presence of unbounded-delay operations, by performing scheduling under timing constraints while minimizing resources or by doing scheduling and resource allocation simultaneously.

5.3.4 Control Scheme

The control scheme required to support the designs synthesized by our approach is much simpler than the one employed in relative scheduling. As the word "single-thread" implies, the control unit of a single-threaded process is a single finite-state machine (FSM). This is because a single-threaded process will stay in one and only one of the control steps $\{C_0, C_1, \ldots, C_N\}$ at any given time. Furthermore, the task to be performed at each control step is fixed. Hence, most existing control synthesis techniques for static scheduling are still applicable here with some modification. For example, each non-anchor control step can be mapped to one or a sequence of states of the FSM as usual according to its length and the clock cycle. A simple anchor like waiting for the expected condition to be met can be done by a busy-waiting state with a conditional state transition as shown in Figure 5.8.

5.3.5 An ILP Method

In this section, we will demonstrate our scheduling approach using a integer-linear programming (ILP) method. Our formulation is based on Hsu's work [HLH91]; however, we expect no major difficulty to apply more comprehensive formulations like [GE92] to our scheduling approach. The problem which we are trying to solve here can be defined as follows:

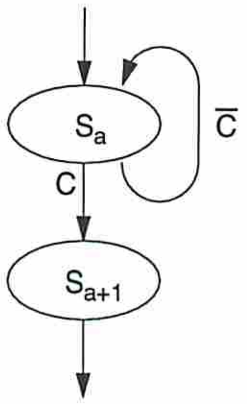


Figure 5.8: An implementation of a simple unbounded control step

> Given a single-threaded control-data flow graph $G(V, E)$, find a minimal-cost schedule that satisfies the given set of timing constraints.

The following notation will be used in the formulation:

- $V = \{v_0, \ldots, v_L\}$. v_0 is the source vertex. i and j are used as vertex indexes.
- $A = \{a_0, \ldots, a_K\}$ is the ordered list of anchors in G, where $a_0 \equiv v_0$. k is the anchor index.
- $C = \{C_0, \ldots, C_N\}$ denotes the control steps to be scheduled, where N is an upper bound of the total number of control steps. n is the control step index.
- There are M types of functional units. $v_i \in F_m$ if F_m can perform operation v_i. The cost of F_m is CS_m.

For simplicity of illustration, we made the following assumptions:

- The lower-bound delay of each anchor is one cycle.
- Each non-anchor operation takes one cycle.
- Only the costs of functional units are considered.

- All the maximum timing constraints have been checked to be feasible.

The techniques to model multi-cycle operations and chaining in ILP can be found from several other formulations [GE92, WGB, Pra93] for static scheduling.

The following discussion begins with the detailed formulation. The linearization of non-linear constraints will be described next. Finally, we will show some experiment results using a public-domain ILP solver.

Formulation

Operation to Control Step Assignment

Let $x_{i,n}$ be a binary variable such that $x_{i,n} = 1$ if v_i is scheduled to control step C_n; otherwise, $x_{i,n} = 0$. Obviously, each operation, including the anchors, must be assigned to one and only one control step.

$$\sum_{n=0}^{N} x_{i,n} = 1 \qquad \forall v_i \in V \tag{5.1}$$

However, the control step to which an anchor $a_k \in A$ is scheduled must be exclusively used by a_k. Let a_k correspond to $v_i \in V$.

$$(x_{i,n} = 1) \Rightarrow \sum_{j \in V - \{v_i\}} x_{j,n} = 0 \qquad \text{for } 0 \leq n \leq N \tag{5.2}$$

Let σ_{v_i} be the index of the control step to which v_i is scheduled. σ_{v_i} can be obtained by

$$\sigma_{v_i} = \sum_{n=0}^{N} n * x_{i,n} \qquad \forall v_i \in V \tag{5.3}$$

Data Dependency

For each $(v_i, v_j) \in E$, we know that v_j must be scheduled after v_i. This can be ensured by the following constraint:

$$\sigma_{v_j} - \sigma_{v_i} \geq 1 \tag{5.4}$$

In fact, each data dependency (v_i, v_j) is like a minimum timing constraint $l_{i,j}$ such that $l_{i,j} = 1$.

Minimum Timing Constraints

Each minimum timing constraint $l_{i,j}$ can be enforced easily by the following constraint:

$$\sigma_{v_j} - \sigma_{v_i} \geq l_{i,j} \tag{5.5}$$

Since the lower-bound delay of each anchor scheduled between σ_{v_i} and σ_{v_j} is 1, the minimum timing constraint $l_{i,j}$ will be satisfied as long as the above constraint is met.

Maximum Timing Constraints

Unlike minimum timing constraints, maximum timing constraints need to be analyzed carefully in order to guarantee their satisfaction in all circumstances. For each maximum timing constraint $u_{i,j}$, there are four possible cases:

Case I. Both v_i and v_j are anchors.

Let them correspond to a_x and a_y in A. From Theorem 5.2, we know $y \leq x + 1$. If $y = x + 1$, we add the following constraint to the formulation:

$$\sigma_{v_j} - \sigma_{v_i} \leq u_{i,j}$$

Otherwise, $u_{i,j}$ is ignored since $y < x$ implies $\sigma_{v_j} < \sigma_{v_i}$.

Case II. v_i is an anchor but v_j isn't.

Let v_i be a_x in A. From Theorem 5.2, we know $zone(v_j) \leq x$. This can be enforced by

$$\sigma_{v_j} < \sigma_{a_{x+1}} \qquad \text{and}$$

$$\sigma_{v_j} - \sigma_{v_i} \leq u_{i,j}$$

Case III. v_i is not an anchor but v_j is.

Let v_j be a_y in A. From Lemma 5.2, we know $zone(v_i) \geq y - 1$. This can be constrained by

$$\sigma_{v_i} > \sigma_{a_{y-1}} \qquad \text{and}$$

$$\sigma_{v_j} - \sigma_{v_i} \leq u_{i,j}$$

Case IV. Both v_i and v_j are not anchors.

From Theorem 5.2, we know $zone(v_j) \leq zone(v_i)$. Assuming $zone(v_i)$ is x, this maximum constraint can be enforced by

$$\sigma_{v_j} < \sigma_{a_{x+1}} \qquad \text{and} \tag{5.6}$$

$$\sigma_{v_j} - \sigma_{v_i} \leq u_{i,j}$$

Unfortunately, x is not a constant because it depends on σ_{v_i}. Alternatively, Constraint 5.6 can be replaced by the following constraints to ensure $zone(v_j) \leq zone(v_i)$:

$$(\sigma_{v_i} < \sigma_{a_k}) \Rightarrow (\sigma_{v_j} < \sigma_{a_k}) \qquad \forall a_k \in A \tag{5.7}$$

Resource Allocation

Let f_m denote the number of function units of type m used in the solution. Hence, the number of operations of type m scheduled to each control step cannot exceed f_m. This is stated as follows:

$$\sum_{v_i \in F_m} x_{i,n} \leq f_m \qquad \text{for } 0 \leq n \leq N \tag{5.8}$$

If we want to minimize the total resources, f_m is a variable to be included in the objective function. For scheduling under resource constraints, f_m becomes a constant.

Objective Function

To find a minimal-cost schedule that satisfies the given timing constraints, the objective function $\mathcal{F}$ to be minimized can be stated as follows:

$$\sum_{m=1}^{M} CS_m * f_m \tag{5.9}$$

We can also try to minimize the total number of control steps C_{step} while meeting the resource constraints by adding the following constraint to the formulation:

$$C_{step} = max(\sigma_{v_i}) \qquad \text{for all } v_i \text{ without successors.} \tag{5.10}$$

Finally, we can simply search for a feasible solution to meet both timing and resource constraints without a objective function.

Linearization

Since there are some non-linear constraints in the formulation presented earlier, they have to be linearized in order to be solved by ILP solvers.

Constraint 5.2 can be easily linearized as follows:

$$(1 - x_{i,n}) * BIG \geq \sum_{j \in V - \{v_i\}} x_{j,n}$$

where BIG is a reasonable larger number; e.g., a number which is slightly larger than $\|V\|$ can be used here. Hence, if $x_{i,n}$ is 1, $\sum_{j \in V - \{v_i\}} x_{j,n}$ will be 0 as required. On the other hand, if $x_{i,n}$ is 0, the above constraint will always be satisfied.

To linearize Constraint 5.7, it is first split into the following two constraints:

$$(\sigma_{v_i} < \sigma_{a_k}) \Leftrightarrow (b_{i,k} = 1) \quad \text{and} \tag{5.11}$$

$$(b_{i,k} = 1) \Rightarrow (\sigma_{v_j} < \sigma_{a_k}) \tag{5.12}$$

where $b_{i,k}$ is a binary variable which is equal to 1 if and only if $\sigma_{v_i} < \sigma_{a_k}$. Constrain 5.12 can be linearized by

$$\sigma_{a_k} > \sigma_{v_j} - BIG * (1 - b_{i,k})$$

Hence, if $b_{i,k}$ is 1, σ_{v_j} will be less than σ_{a_k}. Constraint 5.11 can be replaced by the following constraint:

$$\sigma_{a_k} - \sigma_{v_i} \leq BIG * b_{i,k}$$

Hence, if $\sigma_{a_k} > \sigma_{v_i}$, $b_{i,k}$ will be 1. However, this constraint does not guarantee $b_{i,k}$ to be 0 when $\sigma_{a_k} \leq \sigma_{v_i}$. This requirement can be enforced by adding the term $BIG * b_{i,k}$ to the cost function $\mathcal{F}$ to be minimized.

5.3.6 Experiments

A number of experiments were performed using a public-domain ILP solver, called *lp_solve*[3], to validate our scheduling method. Figure 5.9 shows a single-threaded constraint graph which was the running example used by Ku in his article [KM92].

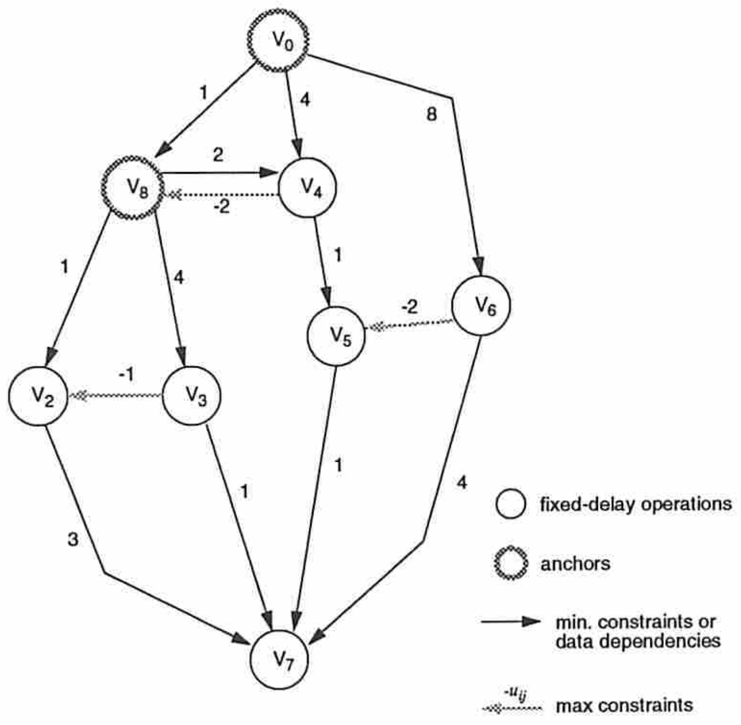


Figure 5.9: A single-threaded constraint graph

In this example, there are two anchors (v_0 and v_{v8}) and three maximum timing constraints. The ILP model produced for these experiments consists of 56 constraints and 86 binary variables.

First, we scheduled this example with the objective of minimizing the number of control steps. A 13-control-step schedule shown in Figure 5.10 was found in

[3] *Lp_solve* is an efficient C program based on a sparse matrix dual simplex LP solver for solving mixed-integer linear programming problems. It is written by Berkellar at Eindhoven University of Technology, Design Automation Section, P.O. Box 513, NL-5600 MB Eindhoven, Netherlands.

less than 2 seconds on a HP 9000/720 workstation. This solution is same as the minimum schedule given in [KM92] using relative scheduling.

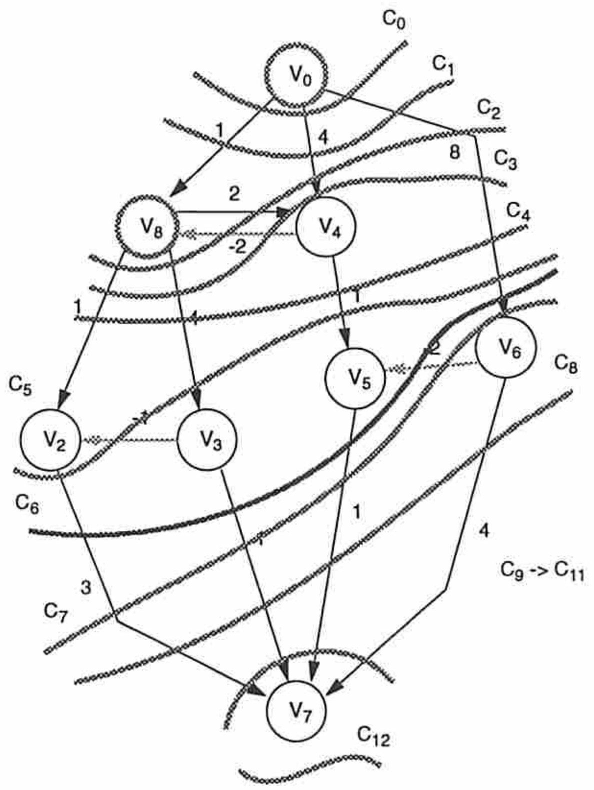


Figure 5.10: A schedule with minimum number of control steps

Next, we tried to schedule this example while minimizing the total resources. A schedule which also used 13 control steps was found as shown in Figure 5.11. Compared to the previous schedule in Figure 5.10, this solution defers v_5 by one control step to C_7 while still satisfying all the timing constraints. Consequently, v_3 and v_5 can share a single module if there exists a functional unit that can perform both of them. On the contrary, since Ku's relative scheduling is similar to as-soon-as-possible (ASAP) scheduling and does not consider resource utilization during scheduling, the minimum schedule it produces will require at least two modules since both v_3 and v_5 are scheduled to C_6. This clearly shows the advantage of our scheduling approach which allows us to trade off performance and resource requirements during scheduling.

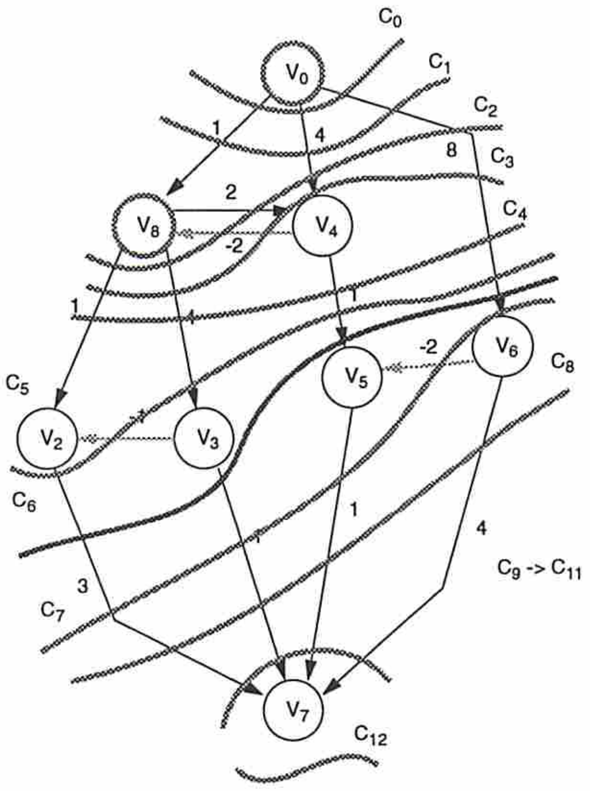


Figure 5.11: A schedule which requires minimum resources

5.4 Synthesis with Multiple Processes

In this section, we will apply our synthesis model to handle systems with multiple communicating processes. There are two major issues to be considered when synthesizing systems with multiple processes.

1. to synchronize processes which interact with each other, and

2. to distribute the resources to each process on a chip according to its performance and resource requirements.

In the following discussion, we will first review our communication model. The problem of multiple-process synthesis which we are focusing on will be defined next. We will then analyze the feasibility of a given set of communication events before discussing the synchronization of both blocking and non-blocking communication events. Finally, we will show how to modify the ILP scheduling method presented earlier to handle systems with multiple processes.

5.4.1 Communication Model

Our inter-process communication model has been discussed in Section 3.2.2. In short, the inter-process communication of a system is defined by a set of communication *events*. Each communication event is a *point-to-point* communication which consists of a *write* operation in the sending process and a *read* operation in the receiving process as shown in Figure 5.12. An event is called *blocking* if its syn-

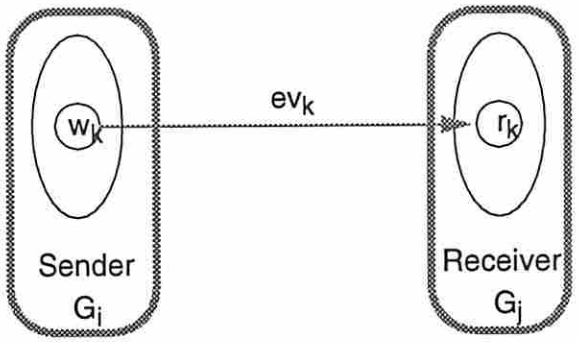


Figure 5.12: Modeling of a inter-process communication event

chronization has to be achieved dynamically via hand-shaking. On the other hand, the write and read operations of a non-blocking event are synchronized statically in time.

Notation

In our synthesis model, each process is represented by a single-threaded constraint graph $G(V, E)$. A point-to-point communication event $ev_k^{i,j}$ is defined by a tuple (w_k^i, r_k^j), where $w_k^i \in V_i$ and $r_k^j \in V_j$. w_k^i is a write vertex (operation) in the sending process G_i. Similarly, r_k^j corresponds to a read vertex in the receiving process G_j.

If $ev_k^{i,j}$ is a blocking event, the time for $ev_k^{i,j}$ to complete is not deterministic in both the sending and receiving processes. Therefore, w_k^i and r_k^j are anchors in G_i and G_j respectively. On the other hand, if $ev_k^{i,j}$ is a non-blocking event, both the sending and receiving processes are required to be synchronized in time when $ev_k^{i,j}$ occurs. In other words, w_k^i and r_k^j must be started at the same time when scheduling G_i and G_j. They are represented by normal vertices with a fixed delay.

Problem statement

The problem to be solved here can be stated as follows:

> Given a set of single-threaded processes $PS = \{G_0, \dots, G_n\}$ and a set of communication events $EV = \{ev_0, \dots, ev_m\}$, schedule and allocate each process in PS such that the timing constraints associated with each process are met, each event in EV is synchronized, and the total cost (resource) constraint is minimized or met.

5.4.2 Communication Feasibility

Unfortunately, it is not always possible to synchronize every communication event in EV due to potential occurrences of *deadlock* or lack of *synchronization points*[4] for non-blocking events. Deadlocks occur when there are circular dependencies among the blocking events. On the other hand, each non-blocking event is required to be

[4]Basically, a synchronization point between two processes is a point in time where their execution will start becoming synchronous. We will formally define it later.

scheduled to a period of time where the execution of both the sending and receiving processes is synchronous. Hence, it is important to analyze the feasibility of the given set of communication events before scheduling.

Definition 5.9 *The* dependency graph *of a set of communication events EV is a directed graph $DG(V, E)$, where $V \equiv EV$ and $(ev_i, ev_j) \in E$ if $\exists$ a process $G_k \in PS$ such that there exists a path from ev_i to ev_j in G_k.*

In other words, DG represents a partial order among the communication events. If ev_i is a predecessor of ev_j in DG, it implies that ev_i must take place before ev_j.

Theorem 5.3 *A set of communication events EV is* infeasible *if its dependency graph DG has a cycle.*

Proof: Let ev_i and ev_j be two vertices on the cycle, and let Ts and Te denote the start time and the end time of an event respectively. Since ev_i is a predecessor as well as a successor of ev_j, we know

$$Te(ev_i) \leq Ts(ev_j) \tag{5.13}$$

$$Te(ev_j) \leq Ts(ev_i) \tag{5.14}$$

However, each event takes some time to complete, which means

$$Ts(ev_i) < Te(ev_i) \tag{5.15}$$

$$Ts(ev_j) < Te(ev_j) \tag{5.16}$$

Combining Equations 5.13 with 5.15 and 5.14 with 5.16, we have

$$Ts(ev_i) < Ts(ev_j) \tag{5.17}$$

$$Ts(ev_j) < Ts(ev_i) \tag{5.18}$$

Equation 5.17 contradicts 5.18; hence, EV is infeasible. □

In short, an acyclic dependency graph ensures the communication to be deadlock-free.

5.4.3 Synchronization of Blocking Events

Since a blocking communication event is synchronized dynamically via handshaking, it will be accomplished as long as both the write and read operations are executed by the sending and receiving processes eventually. The scheduling of processes does not have a direct effect on the synchronization of blocking events. In fact, if every event of an acyclic dependency graph DG is blocking, the inter-process communication can be made valid by simply scheduling individual processes properly. This is because a valid scheduling of each process will not only satisfy the timing constraints but also meet all its data and sequencing dependencies. Consequently, if two vertices (blocking events here) are ordered in DG, then they cannot be scheduled out of order in any valid schedule. Hence, each vertex in DG will take place only after all its predecessors have completed.

5.4.4 Synchronization of Non-blocking Events

If a communication event is non-blocking, its sending and receiving processes cannot be scheduled independently. A non-blocking event $ev_k^{i,j} = (w_k^i, r_k^j)$ is synchronized only if the scheduling of processes G_i and G_j can guarantee that the start time of w_k^i and r_k^j are the same. Unfortunately, this synchronization cannot be done by simply scheduling both w_k^i and r_k^j to the same control step in G_i and G_j like [HP92, Geb92] due to the occurrences of unbounded-delay operations. Instead, we will introduce the notion of synchronization points and show how they can be utilized to synchronize non-blocking events.

Definition 5.10 *A* synchronization point *of two processes G_i and G_j is a tuple (C_x^i, C_y^j) such that C_x^i and C_y^j are the control steps of G_i and G_j respectively and the execution of G_i and G_j always leave C_x^i and C_y^j simultaneously.*

Some synchronization points of two interacting processes can be found even before their scheduling has been done. For example, if two processes G_i and G_j have a common starting state, (C_0^i, C_0^j) will be a synchronization point. Each blocking event $ev_k^{i,j} = (w_k^i, r_k^j)$ will contribute a synchronization point $(C_{\sigma(w_k)}^i, C_{\sigma(r_k)}^j)$ between G_i and G_j.

If there exists a synchronization point between two processes, their execution will be synchronous for a period of time from the synchronization point to the next control step with an unbounded length in either process. In fact, each non-blocking event requires such a period of time in which its write and read actions can be scheduled and synchronized. This is illustrated in Figure 5.13 and in the following lemma.

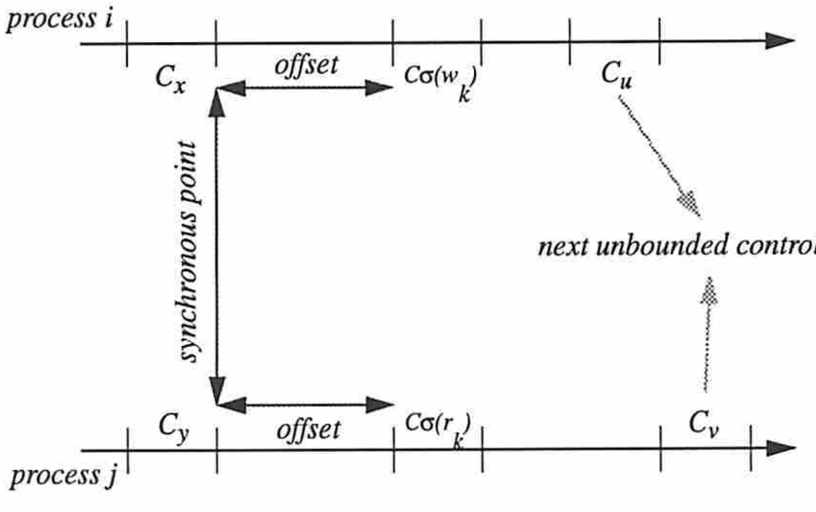


Figure 5.13: The synchronization of a non-blocking communication event

Lemma 5.4 *Let (C_x^i, C_y^j) be a synchronization point between processes G_i and G_j and let C_u^i and C_v^j be the control steps with unbounded lengths which immediately follow C_x^i and C_y^j. A non-blocking event $ev_k^{i,j} = (w_k^i, r_k^j)$ scheduled in the following regions,*

$$x < \sigma(w_k^i) < u \quad \text{and} \tag{5.19}$$

$$y < \sigma(r_k^j) < v, \tag{5.20}$$

is synchronized if

$$offset(C_x^i, w_k^i) = offset(C_y^j, r_k^j)$$

where $offset(a, b)$ is the time from the end of a to the start of b.

Proof: From Equations 5.19 and 5.20, we know that there is no control step with an unbounded length between C_x^i and $C_{\sigma(w_k)}^i$ as well as between C_y^j and $C_{\sigma(r_k)}^j$. Therefore, $offset(C_x^i, w_k^i)$ and $offset(C_y^j, r_k^j)$ are bounded and equal. Also, since (C_x^i, C_y^j) is a synchronization point, the execution of G_i and G_j will leave C_x^i and C_y^j simultaneously; i.e., $Te(C_x^i) = Te(C_y^j)$. Hence, we have

$$\begin{aligned} & Te(C_x^i) + offset(C_x^i, w_k^i) = Te(C_y^j) + offset(C_y^j, r_k^j) \\ \Rightarrow \quad & Ts(w_k^i) = Ts(r_k^j) \end{aligned}$$

In other words, the start time of w_k^i and r_k^j are the same. □

5.4.5 ILP modification

In what follows, the ILP formulation presented in Section 5.3.5 will be extended to handle systems with multiple processes. Assuming the dependency graph DG of the given set of communication events EV is acyclic, the original ILP formulation except the objective function can be simply duplicated once for each process in

PS. Since DG is acyclic, the blocking events will be synchronized by themselves as long as each process is scheduled properly.

For each non-blocking event $ev_k^{i,j} = (w_k^i, r_k^j)$ in EV, the following constraint is added to the formulation for each pair of a_x^i and a_y^j such that both a_x^i and a_y^j are anchors and $(C^i_{\sigma(a_x)}, C^j_{\sigma(a_y)})$ is a synchronization point between G_i and G_j.

$$\begin{aligned} & (\sigma(a_x^i) < \sigma(w_k^i) < \sigma(a_{x+1}^i)) \quad \wedge && (5.21)\\ & (\sigma(a_y^j) < \sigma(r_k^j) < \sigma(a_{y+1}^j)) && \\ \Rightarrow \; & \sigma(w_k^i) - \sigma(a_x^i) = \sigma(r_k^j) - \sigma(a_y^j) && (5.22) \end{aligned}$$

This constraint is basically a repetition of Lemma 5.4. If the left-hand side of this constraint is $true$, w_k^i and r_k^j are to be synchronized with respect to $(C^i_{\sigma(a_x)}, C^j_{\sigma(a_y)})$ using the same offsets.

Obviously, Constraint 5.21 needs to be linearized. First, it can be replaced by:

$$\begin{aligned} \sigma(a_x^i) < \sigma(w_k^i) &\Leftrightarrow b_k^1 = 1 && (5.23)\\ \sigma(w_k^i) < \sigma(a_{x+1}^i) &\Leftrightarrow b_k^2 = 1 && (5.24)\\ \sigma(a_y^j) < \sigma(r_k^j) &\Leftrightarrow b_k^3 = 1 && (5.25)\\ \sigma(r_k^j) < \sigma(a_{y+1}^j) &\Leftrightarrow b_k^4 = 1 && (5.26)\\ b_k^5 &= b_k^1 \wedge b_k^2 \wedge b_k^3 \wedge b_k^4 && (5.27)\\ b_k^5 = 1 &\Rightarrow \sigma(w_k^i) - \sigma(a_x^i) = \sigma(r_k^j) - \sigma(a_y^j) && (5.28) \end{aligned}$$

Constraints 5.23 to 5.26 can be linearized in a similar way as we did for Constraint 5.7 in page 120. Constraint 5.27 is equivalent to

$$\begin{aligned} b_k^5 &\leq b_k^1 \\ &\vdots \\ b_k^5 &\leq b_k^4 \end{aligned}$$

$$b_k^5 \geq (b_k^1 + \ldots + b_k^4) - 3$$

Hence, if any of $b_k^1, \ldots, b_k^4$ is 0, b_k^5 will be 0. If all $b_k^1, \ldots, b_k^4$ are 1, b_k^5 becomes 1 as well.

Finally, Constraint 5.28 can be replaced by:

$$\sigma(w_k^i) - \sigma(a_x^i) \leq \sigma(r_k^j) - \sigma(a_y^j) + (1 - b_k^5) * BIG$$
$$\sigma(w_k^i) - \sigma(a_x^i) \geq \sigma(r_k^j) - \sigma(a_y^j) - (1 - b_k^5) * BIG$$

Hence, if b_k^5 is 1, $\sigma(w_k^i) - \sigma(a_x^i)$ will be equal to $\sigma(r_k^j) - \sigma(a_y^j)$.

In order to distribute the resources to each process according to the performance requirement, the objective function $\mathcal{F}$ can be replaced by one which represents the total resources consumed by all the processes in PS.

$$\sum_{p \in PS} \sum_{m=1}^{M} CS_m * f_m^p \tag{5.29}$$

Therefore, the solution found by the above formulation will be one which meets the timing constraints associated with each process, synchronizes every communication event and minimizes the total cost.

5.4.6 Experiments

Figure 5.14 shows a two-process example derived from a network package decoding/encoding system [KFJM92]. In this example, there are three communication events (shown as dotted lines) between the decoder G_1 and encoder G_2 processes. One of them (V_3, N_5) is a blocking event and the other two are non-blocking. Furthermore, three maximum constraints are imposed on process G_1 and two are imposed on G_2. The ILP model produced for this example consists of 114 constraints and 211 variables (190 binary).

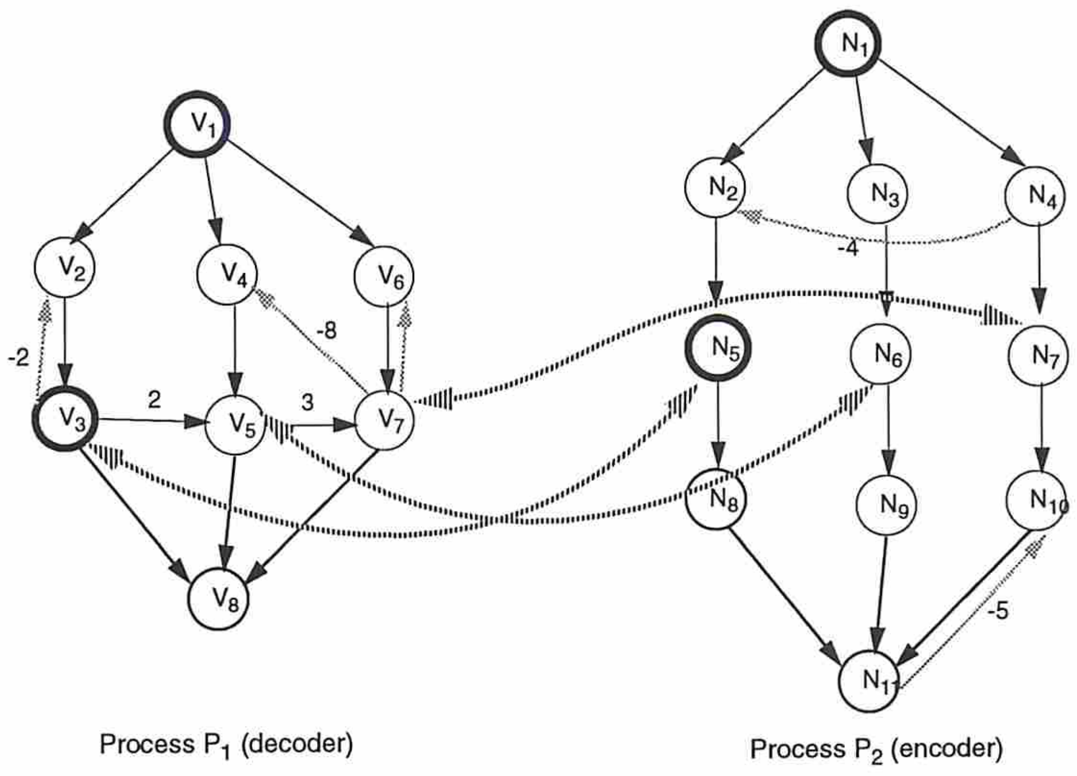


Figure 5.14: A Two-Process Example

First, we scheduled both processes in a minimum number of control steps while satisfying all the timing constraints as well as synchronizing the three communication events. As a result, G_1 and G_2 are scheduled in 9 and 10 control steps respectively as shown in Figure 5.15. In this schedule, two non-blocking events (V_5, N_6) and (V_7, N_7) are synchronized by scheduling them with respect to the synchronization point (V_3, N_5) using offsets 2 and 5 respectively.

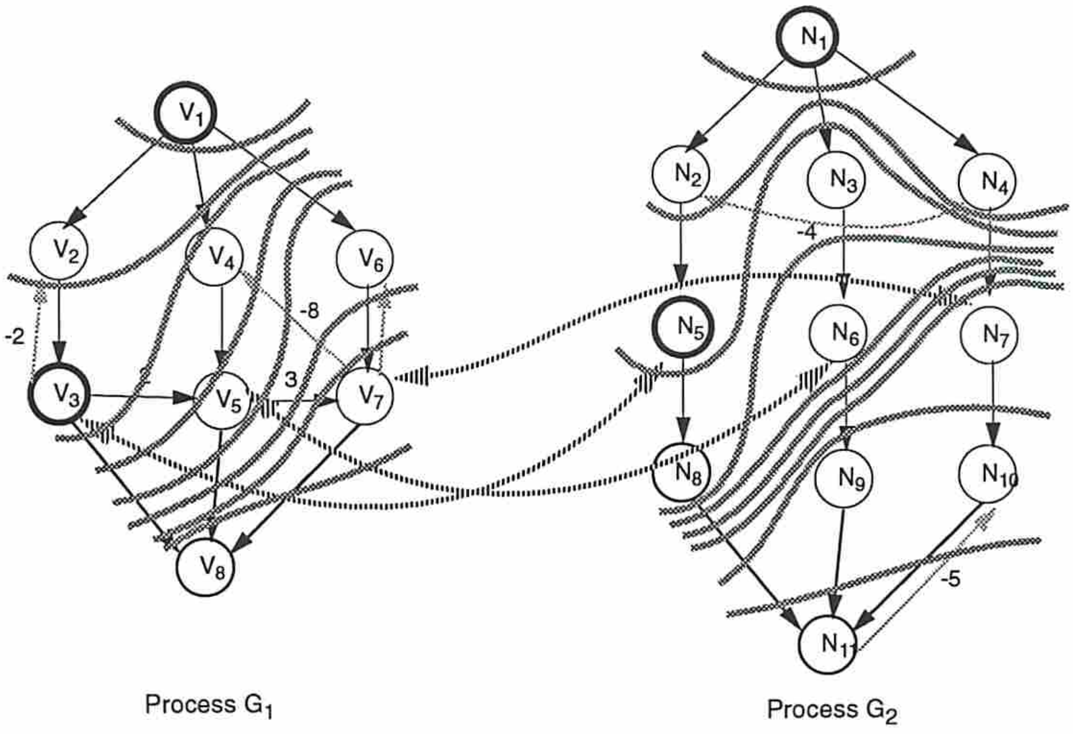


Figure 5.15: A schedule with a minimum number of control steps

Next, we tried to schedule this example while minimizing the total resources. Due to the tight timing constraints, the solution found as given in Figure 5.16 does not show improvement over the previous solution in terms of resource allocation. The total number of control steps, however, remains the same.

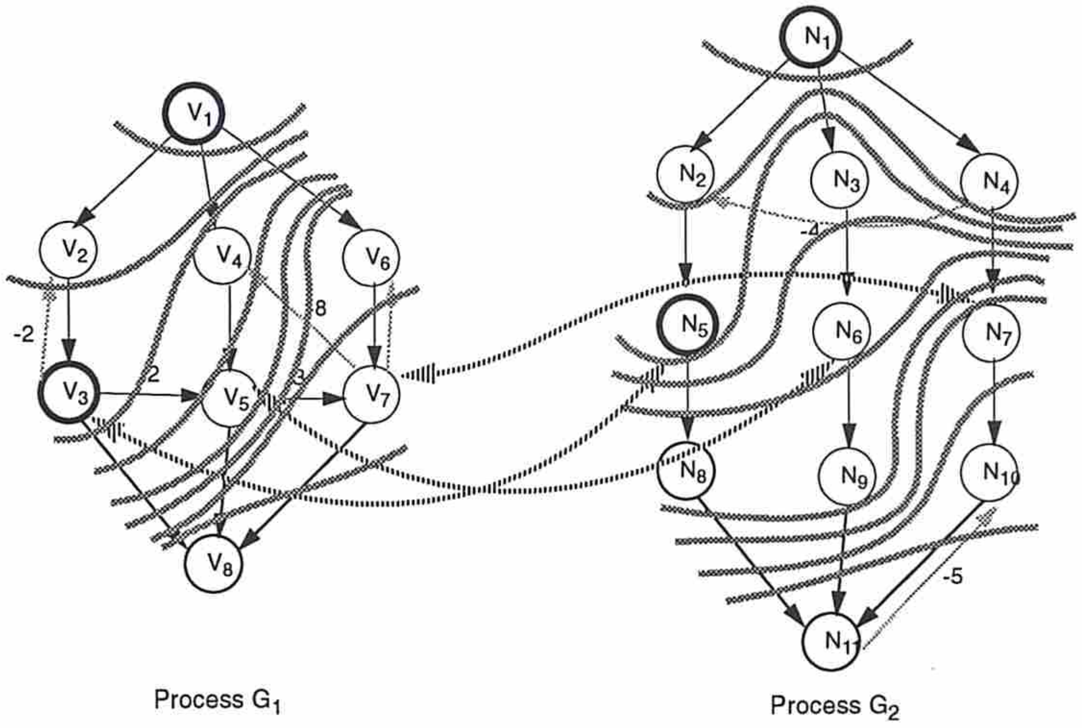


Figure 5.16: The solution obtained by minimizing the total resources

5.5 A Heuristic Approach for Multi-Process Synthesis

In the earlier sections, we demonstrated our synthesis approach for designs with unbounded-delay operations and communicating processes using an ILP method. However, for large designs, the size of the ILP models may be too large to obtain a solution within a reasonable run time. Fortunately, as we discussed earlier, our synthesis approach is compatible with most existing static scheduling techniques. Hence, a good heuristic approach can be obtained easily by modifying an existing heuristic-search procedure. In this section, we will discuss how to modify the freedom-based scheduling technique [PPM86] to handle unbounded-delay operations under timing constraints and communicating processes.

The basic idea behind freedom-based scheduling is to schedule the operations on the critical path first. For the remaining off-critical-path operations, their *freedoms* are calculated. The freedom of an operation is determined from the earliest time the execution of the operation can start to the latest time at which the operation has to be finished. The operations are assigned to a control step in the order of increasing freedom.

Freedom-based scheduling can be modified in the following way to handle unbounded-delay operations under detailed timing constraints.

1. All the operations with unbounded delays have to be scheduled to an exclusive control step.
   Since the critical path is the longest path in a process's constraint graph, the unbounded-delay operations of a single-threaded process should, by definition, be all on the critical path. Hence, by scheduling the critical path first, we can ensure that each unbounded-delay operation is assigned to an

empty control step. In addition, when scheduling remaining fixed-delay operations, the control steps occupied by the unbounded-delay operations can be avoided.

2. The calculation of freedom should take into account both minimum and maximum timing constraints.
   When detailed timing constraints are present among the operations, the freedom of an operation not only depends on the data dependencies but also is affected by the associated timing constraints.

   Basically, the freedom of an operation v is determined by an execution interval $[el(v), lt(v)]$, where $el(v)$ is the earliest time that v can be scheduled and $lt(v)$ the latest time. Since each minimum timing constraint l_{ij} can be represented by an edge (v_i, v_j) weighted by l_{ij} in the constraint graph, $el(v_j)$ of an operation v_j can be recursively defined as:

$$\begin{aligned} el(v_j) &= \max_{v_i \in im\_preds(v_j)} \{el(v_i) + w_{ij}\} \\ el(v_j) &= 0 \quad \text{if } im\_preds(v_j) = \emptyset \\ el(v_j) &= \sigma(v_j) \text{ if } v_j \text{ has been scheduled to } \sigma(v_j) \end{aligned}$$

   where $im\_preds(v_j)$ is the set of immediate predecessors of v_j and w_{ij} is either the delay of v_i or a minimum timing constraint l_{ij}. Similarly, the latest execution time $lt(v_i)$ of an operation v_i can be defined as:

$$\begin{aligned} lt(v_i) &= \min_{v_j \in im\_succs(v_i)} \{lt(v_j) - w_{ij}\} \\ lt(v_i) &= N \quad \text{if } im\_succs(v_i) = \emptyset \\ lt(v_i) &= \sigma(v_i) \text{ if } v_i \text{ has been scheduled to } \sigma(v_i) \end{aligned}$$

where $im\_succs(v_i)$ are the set of immediate successors of v_i, N is the schedule length, and w_{ij} is either the delay of v_j or a maximum timing constraint u_{ij} between v_i and v_j.

For systems with multiple communication processes, if we schedule individual processes one at a time and then propagate the synchronization constraints from one process to another, feasible solutions may not be found. Hence, a better approach would be the one that can ensure the synchronization of all the inter-process communication events. The basic principle of freedom-based scheduling can still be applied here; i.e., the operations with higher scheduling difficulties should be scheduled first.

As we have discussed in Section 5.4.3, the scheduling of processes does not have a direct effect on the synchronization of blocking events. In addition, the scheduling of a non-blocking event requires a synchronization point, e.g. a blocking event, between the sending and receiving processes. Therefore, we can begin with scheduling the critical path including the unbounded-delay operations of each process. Once all the unbounded-delay operations are scheduled, the synchronization points among the processes can be identified. Therefore, each non-blocking communication event can be scheduled simultaneously at both the sending and receiving processes according to the available synchronization points between them. Finally, the remaining operations can be scheduled according to their freedoms as usual. This approach is outlined as follows:

```
multi-process-fbs ($PS, EV$)
/* $PS$ is a set of processes */
/* $EV$ is a set of communication events among $PS$ */
for (each process $G_i \in PS$) do
    schedule the critical path of $G_i$;
    /* All the anchors in $G_i$ should have been scheduled now */
    calculate the freedoms of all non-critical-path operations in $G_i$;
endfor;
while ($\exists$ non-blocking events in $EV$ that have not been scheduled) do
    select the event $ev_k^{i,j} = (w_k^i, r_k^j)$ with the smallest freedom;
    schedule $w_k^i$ and $r_k^j$ in $G_i$ and $G_j$ respectively using the same offset
            with respect to a synchronization point between $G_i$ and $G_j$;
    update the freedoms of the affected operations;
endwhile;
for (each process $G_i \in PS$) do
    while ($\exists$ operations in $G_i$ that have not been scheduled) do
        schedule the operation with the smallest freedom;
        update the freedoms of the affected operations;
    endwhile;
endfor;
```

5.6 Summary

In this chapter, we presented an approach for synthesis of designs with unbounded-delay operations under timing constraints and with multiple communicating processes. Compared to *relative scheduling*, this approach allows us to trade off between performance and resource requirements during scheduling as well as to reduce the control overhead. Our approach is based on the observation that each process generally corresponds to one thread of control and there exists a sequential order among the unbounded-delay operations in the process description. By preserving this order, scheduling of single-threaded processes can still be done statically in terms of control steps despite the presence of unbounded-delay operations. Consequently, many good synthesis techniques originally developed for designs with only fixed-delay operations can still be utilized in our approach with

some modifications. In this chapter, we demonstrated our scheduling method using an ILP formulation and also discussed how to modify an existing heuristic technique, *freedom-based scheduling*, to meet our scheduling requirements.

Chapter 6

Verification of Synthesized RTL Designs

Due to the cost of engineering and fabrication and the critical marketing time, design errors of digital systems should be eliminated at all costs. Although one may argue that the synthesized designs should be *correct by construction*, in reality there is no such guarantee unless the whole synthesis process, including techniques and programs, can be formally validated. However, to validate a large software system like a high-level synthesis system formally is still impractical, if not impossible, for current formal verification techniques [McF93]. A more practical alternative is to verify the synthesized designs with respect to their specifications.

Although designs can be verified at various levels of abstraction, it is desirable to find any design problem as early as possible. In addition, a high-level synthesis system may produce many designs for a given set of constraints; without proper verification, there is a lack of sense of correctness while the designs are being evaluated or compared. On the other hand, pure functional validation is not enough because many design errors are also related to the control and timing of the design. Hence, we believe that there is a strong need for an automatic tool which can check both the functionality and timing of synthesized designs efficiently, to be integrated in a high-level synthesis system.

In this chapter, we will present an efficient approach for checking the RTL designs produced by the USC ADAM high-level synthesis system. This methodology is also applicable to other synthesis systems incorporating a similar design flow. Our approach is motivated by the observation that the structural designs are derived in a well-defined manner from the behavioral specifications [MPC88] in the ADAM system. These RTL designs possess several common properties so that symbolic simulation can be effectively utilized to perform the checking task. Using this approach, we are able to not only verify the design functionality formally but also take into account the interaction between the data path and the controller as well as the timing issues, such as delays and the clocking scheme.

This chapter is organized as follows. First, we describe the problem of verifying synthesized RTL designs in Section 6.1, and analyze a generic high-level synthesis model in order to identify properties of automatically synthesized designs. In Section 6.2, we give an overview of our approach for checking synthesized RTL designs. The hybrid symbolic/numeric simulation model used in our approach will be described in Section 6.3. In Section 6.4 we will show that there is an *isomorphic* property between the behavioral specifications and the extracted behaviors of the corresponding implementations. A behavior-comparison procedure based on this property will then be given. Finally, we will present some experiment results and summarize this chapter.

6.1 Problem Statement

In Figure 6.1, we show a generic model of high-level synthesis. The problem which we are solving here can be briefly described as follows:

> *Show whether or not the RTL implementation I will perform the required computation specified in the design behavioral specification S for every execution instance.*

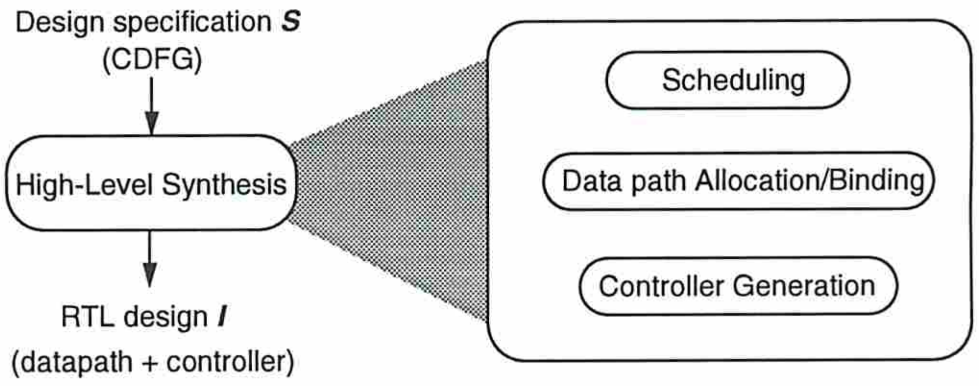


Figure 6.1: A high-level synthesis model

The design specification S here is an control data flow graph $CDFG_S$ which defines the required computations to be done for every execution instance. The implementation I itself is a static structure which consists of a data path and a controller. We are asked in this problem to obtain the dynamic relationship between sequences of inputs and outputs while I is physically operated under the specified clocking scheme and input/output protocol. However, even if we can faithfully obtain this dynamic behavior of I, it is still very difficult to prove the correctness if I and S are regarded as two independent entities. This is because the problem is similar to showing whether or not two arbitrary behaviors are functionally equivalent, which is believed to be *undecidable.* In Section 6.6, we will apply the computation theory to demonstrate this undecidability of solving the general RTL verification problem, where S and I are considered independent. Even a simpler problem like determining the equivalence of two finite-precision algebraic expressions cannot be done in either P or NP time if $NP \neq$ *co-NP*[1].

[1]Informally, co-NP (complement of NP) defines a set of problems whose complement problems are in NP. Currently, it is not known yet whether NP is closed under complementation, although it is generally doubted.

Lemma 6.1 *Showing the equivalence of two finite-precision algebraic expressions is a co-NP problem which cannot be done in either P or NP time if NP $\neq$ co-NP.*

Proof: The proof is straightforward. First, we know that the *not tautology* problem[2] is an NP-complete problem and its complement, the *tautology* problem, is a co-NP problem which cannot be solved in P nor NP time if NP $\neq$ co-NP [GJ79].

The *tautology* problem can be easily reduced in polynomial time to the problem of showing the equivalence of two Boolean expressions since each instance of this problem is the same as showing whether or not the Boolean expression in question is equivalent to constant 1. The later problem in turn can be easily reduced to one which determines the equivalence of two finite-precision algebraic expressions since Boolean expressions are a subset of finite-precision algebraic expressions. Therefore, the latter problems are both co-NP problems which cannot be done in P nor NP time if NP $\neq$ co-NP. □

In high-level synthesis, the synthesis system actually derives the structural design from the behavioral specification in a *well-defined* manner [MPC88]. Consequently, the specification S and the implementation I are not independent. In fact, there are *links* between S and I. These links can be utilized in verifying their correspondence. In the following section, we will describe the links between S and I in terms of a number of common properties of the synthesized RTL implementations.

6.1.1 Properties of the Synthesized RTL Designs

In high-level synthesis, the RTL implementation I is the result of a mapping from $CDFG_S$. The major tasks of this mapping involve assigning the operations to

[2]A *tautology* is a Boolean expression that has the value 1 for all assignments of values to its variables. The *not tautology* problem is to return 1 if a given Boolean expression is not a tautology and return 0 otherwise.

control steps (scheduling), assigning the operations and values to hardware (data path allocation/binding), and generating a controller to deliver the required control signals (control synthesis) [MPC88]. Consequently, the implementation I, if mapped correctly, will have the following properties:

Property 6.1 For each operation *op* in $CDFG_S$, there exists a functional unit u in I such that u is designated perform *op* and *op* is achieved by directing all the input values of *op* to the corresponding input ports of u.

For example, Figure 6.2 shows a CDFG and a possible RTL implementation. The Adder in I is designated to perform both additions $+_1$ and $+_2$ in $CDFG_S$. $+_1$ is achieved by directing a and b to the input ports of Adder through Mux1 and Mux2. Similarly, the multiplication $*$ in $CDFG_S$ is bound to the Multiplier in I.

Property 6.2 For each data dependence (op_i, op_j) in $CDFG_S$, there exists an interconnect path in I between the functional units u_i and u_j which are designated to perform op_i and op_j. The interconnect path is set up by using buses and/or switching devices. If the output of u_i is not consumed by u_j within a cycle, a storage element is needed in the interconnect path to store the value before sending it to the input port of u_j.

In Figure 6.2, the data dependency between $+_1$ and $*$ is done through the interconnect path, Adder $\rightarrow$ Reg1 $\rightarrow$ Multiplier. Similarly, the dependency between $+_2$ and $*$ is achieved by the path, Adder $\rightarrow$ Mux3 $\rightarrow$ Reg2 $\rightarrow$ Multiplier.

Property 6.3 $CDFG_S$ defines the required computations which have to be done in I for every execution instance.

For example, after applying four input values a, b, c and d to the RTL design I in Figure 6.2, I is expected to perform two additions and one multiplication and produce an output which is equal to $(a + b) * (c + d)$ after three cycles.

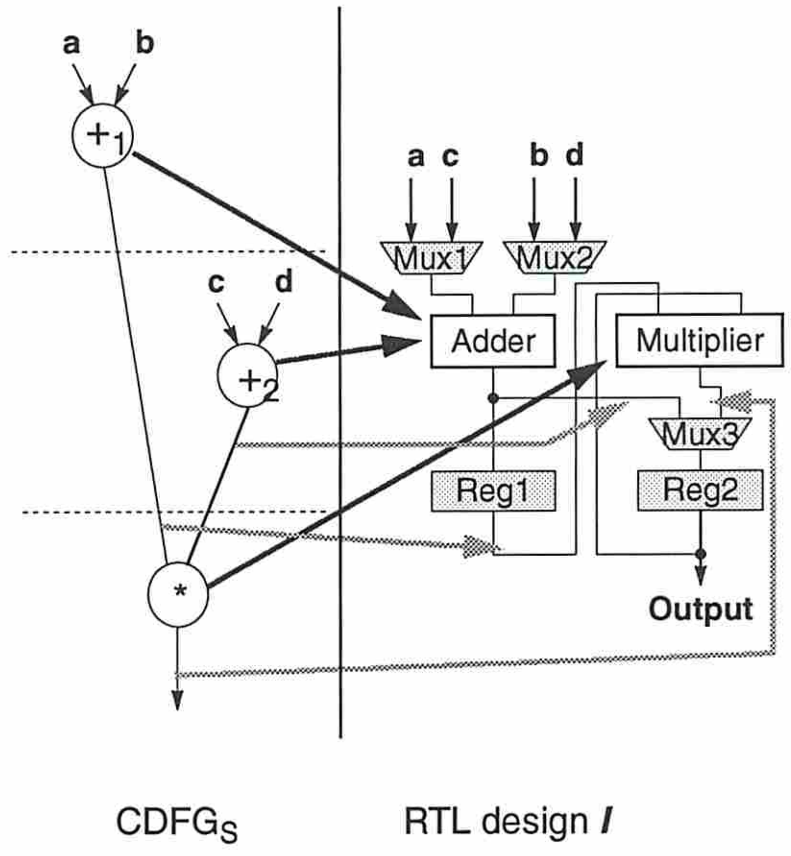


Figure 6.2: An example of the links between $CDFG_S$ and its synthesized RTL design I

We will show in Sections 6.3 and 6.4 how these properties can be effectively utilized to verify synthesized designs. In fact, the information regarding these properties is generally available after the synthesis process. For example, this information is represented explicitly in the Design Data Structure (DDS) of the ADAM high-level synthesis system by means of *bindings* [KP85]. This information is particularly useful in diagnosing design errors because these bindings represent the design decisions made during the synthesis process and the events that should occur while the design is operating. If errors are found, the implementation bindings which are generated during the simulation can be traced to determine the cause.

6.2 Approach Overview

From the previous section, we find that the correctness of a synthesized RTL design is really determined by whether or not it will perform the required data operations and data transfers as specified in the given CDFG. Hence, we developed an approach which combines symbolic simulation at the RT level with a behavior-comparison procedure based on the properties described in Section 6.1.1.

The motivations to apply symbolic simulation in our approach are twofold. First, it provides formal results because the simulator operates over a *symbolic* domain, and at the same time we are able to take into account design timing in terms of the clocking scheme, delays, and input/output protocols. Second, the symbolic simulation results are ready for comparison with the design specification for high-level synthesis since they both can be represented in a similar form such as a data flow graph.

Figure 6.3 shows a flow chart which briefly illustrates our approach. First,

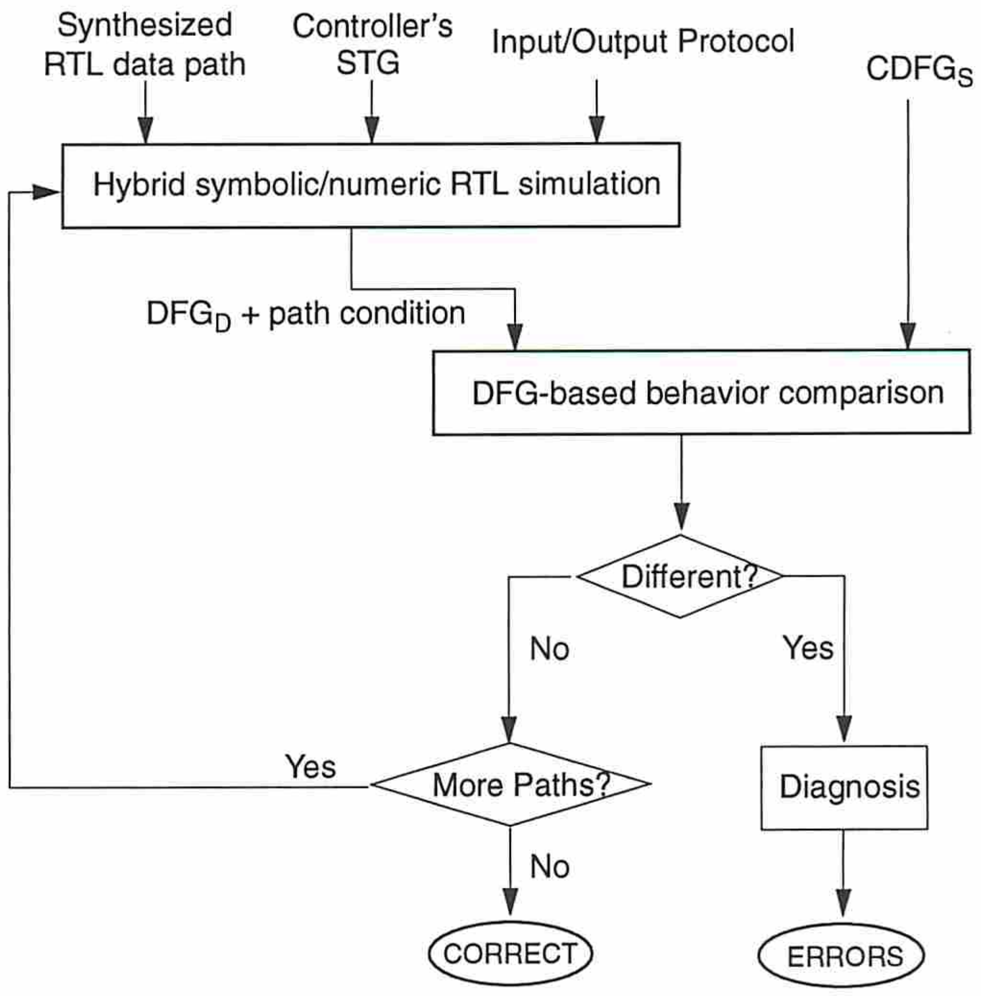


Figure 6.3: An overview of our approach for checking synthesized RTL designs.

the inputs to our approach includes the behavioral specification $CDFG_S$, the synthesized RTL data path, the state-transition graph of the controller, and the input/output protocol. The control flow of the design is analyzed to produce a list of all possible execution paths. For each execution path, the associated *path condition* will be used to drive the subsequent simulation.

The hybrid symbolic/numeric simulation performed next proceeds from one execution path to another. It is a hybrid model because the data path is evaluated symbolically but the controller is simulated numerically so that all the control signals will be either 1, 0 or *unknown* throughout the simulation. The result of the simulation is represented by a data flow graph which describes the actual data operations and data transfers occurring in the synthesized data path.

Finally, the simulation result is compared graphically with the given $CDFG_S$ under the same path condition. If the comparison procedure finds any difference between these two graphs, there exist design errors and the current simulation result is diagnosed to find the possible causes. The whole process is repeated until no more execution paths are left. If the data flow graph obtained from the simulation matches with the given $CDFG_S$ for every execution path, the given RTL design is considered to be synthesized correctly.

6.3 Hybrid Symbolic/Numeric Simulation

As we have discussed earlier, what is most important for a synthesized RTL design is whether or not it performs the required data operations and the correct sequencing of data transfers for each execution instance. Also, many design errors are related to the control and timing of the design. Hence, we need to be able to extract the circuit behavior in terms of the symbolic data operations and data transfers that occur in the data path and at the same time to emphasize exercising

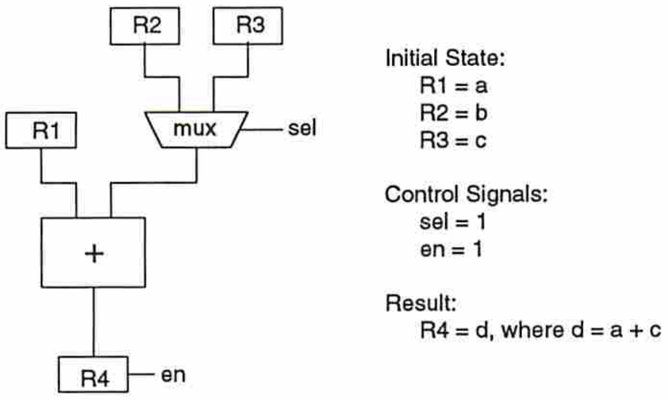


Figure 6.4: An example of symbolic simulation

the control and modeling the timing. The hybrid symbolic/numeric simulation to be described here performs exactly this task.

The idea behind symbolic simulation is similar to extending arithmetic over numbers to symbolic algebraic operations over symbols and numbers. For example, Figure 6.4 shows an RTL circuit with an adder, a two-to-one multiplexer and four registers. Let the symbolic values a, b and c represent the initial register values stored in R1, R2 and R3 respectively. Suppose both the control signals *sel* and *en* are 1. After simulating the circuit symbolically, a new symbolic value d is produced by the adder and stored in R4, and d is equivalent to $a + c$. In this way, we have the response to all possible initial conditions of R1, R2 and R3 in one simulation run.

The simulation model we use is event-driven. Figure 6.5 shows a typical flow of event-driven simulation [ABF90]. The main difference between our model and the traditional ones is that the evaluation of activated elements and the representation of signal values are symbolic in our model. In addition, a transport delay model can be used to reflect the circuit operation more accurately.

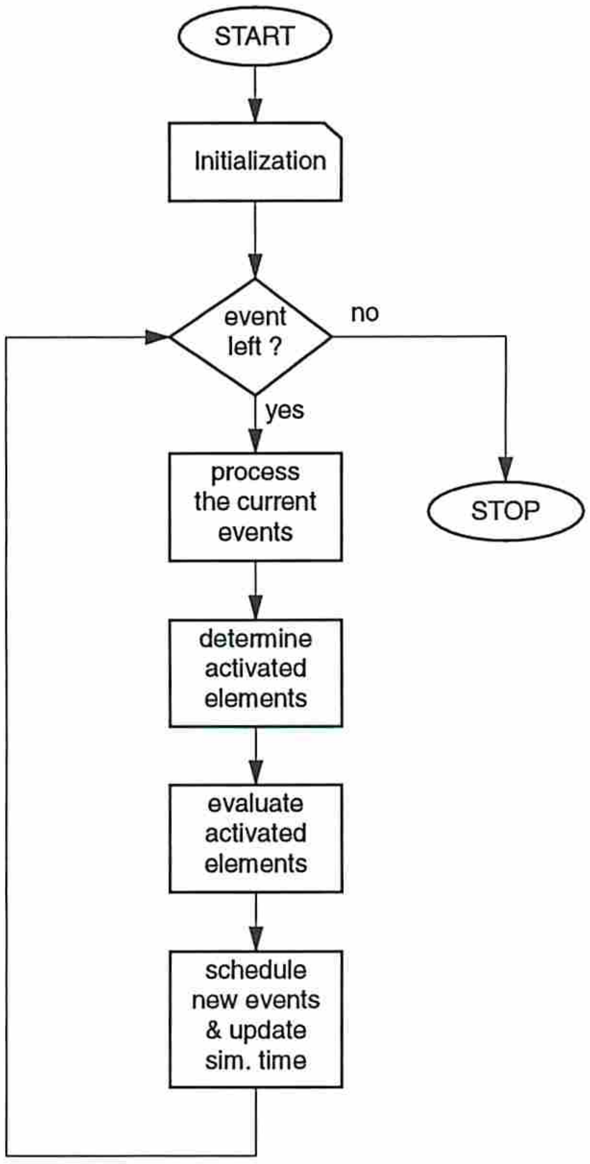
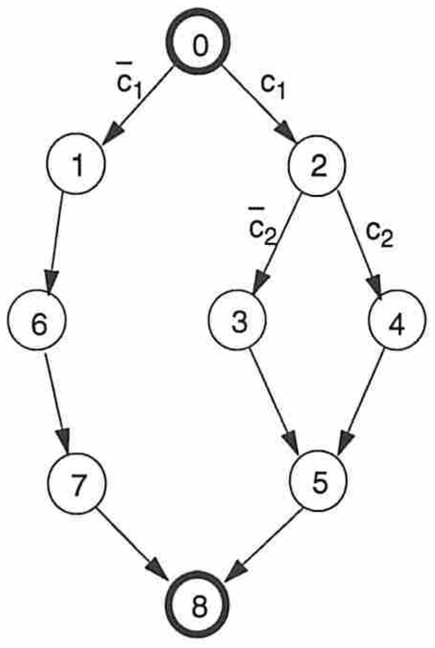

START
Initialization
event left ?
no
yes
process the current events
STOP
determine activated elements
evaluate activated elements
schedule new events & update sim. time

Figure 6.5: Typical flow of event-driven simulation



Executaion Paths	Path Condition
0-1-6-7-8	c_1=0
0-2-3-5-8	c_1=1, c_2=0
0-2-4-5-8	c_1=1, c_2=1

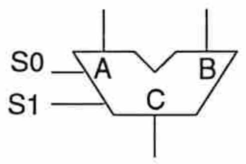
Figure 6.6: Execution paths of a state-transition graph

For designs with conditional branches, the state-transition graphs of their controllers will consists of several possible execution paths.

Definition 6.1 *An execution path is a direct path from an* initial *state to an* end *state in the state-transition graph of a finite-state machine (FSM) controller.*

Definition 6.2 *A path condition is an assignment to a set of Boolean variables which together determine alternative paths through the state-transition graph.*

For example, the state-transition graph shown in Figure 6.6 contains three possible execution paths, each of which is associated with a *path condition* that represents the assumptions made along the path during the control flow analysis.



S0	S1	C
0	0	A + B
0	1	A - B
1	0	A & B
1	1	A I B

propagation delay = 10 ns

Figure 6.7: A four-function ALU behavioral model

6.3.1 Element Evaluation

The evaluation of an element is done to compute its output values according to its current input values.

6.3.1.1 Data Path

The evaluation of a datapath module depends on its behavior model. For example, in ADAM's DDS, this information is available from the behavioral model of the component used to implement the module. Currently, we represent this information in the form of function tables. The function table of a data path module defines the manipulation of symbolic data for each possible condition on the control lines. For example, the behavioral model for a simple four-function ALU is shown in Figure 6.7. There are three kinds of datapath modules:

- **Functional Units.** These modules are used to perform the operations given in a specification (CDFG). When a functional unit is evaluated, a new symbolic operation is performed. In general, its control inputs, if any, are used for function selection. Then, one or more new symbolic values are produced at its output ports after a specified delay time. The input/output symbolic values are related by the symbolic operation being performed.

- **Switching Devices.** The result of evaluating a switching device is that symbolic values are transferred from its input ports to its output ports. The current values of its control lines determine the paths on which the transfers take place. Similarly, the output change is separated from the input change by a propagation delay. No new symbolic value is produced in the evaluation of a switching device.

- **Storage Elements.** Symbolic values can be written into or read from storage elements via their data input/output ports. Both registers and on-chip memory are allowed in our simulation model. A storage element is evaluated whenever its clock or enable signals change. The memory addresses are regarded as control signals; therefore, they are numeric.

If the input condition of a module being evaluated is invalid, its outputs and data storage, if any, are set to *unknown*.

The datapath carriers (nets) are used for propagating the symbolic values. A carrier connecting more than one output port requires those outputs to be tristate. Normally, at most one tristate output is enabled at any time. A *value collision* occurs if two or more output ports drive a carrier at the same time [KW89]. This type of design error can be easily detected by the simulator.

6.3.1.2 Controller

The controller is evaluated when there is a change on its clock signal. If the clock change results in a state transition, the outputs are computed and the controller moves to the next state in the current execution path. The path condition of the current execution path is updated, if necessary, at each state transition. For example, if the state transition requires the inputs i_1 and i_2 to be 1 and 0 and if the symbolic values currently appearing at i_1 and i_2 are a and b respectively, then $a = 1$ and $b = 0$ will be added to the path condition. If there is a conflict between the assumptions made in the state transition and the path condition to be updated, the current execution path is a *false path* which will never occur. The simulation will proceed to the next execution path immediately if a false path is found. On the other hand, if a required input for the state transition contains an *unknown* value, the simulator aborts the current execution path and reports a data-dependency violation.

6.3.2 Representation of Symbolic Data

The symbolic values and operations which occur during the simulation are the actual events exhibited by the RTL design. These values and operations constitute the actual datapath behavior of the structural implementation to be compared with the design specification. Hence, it is very important to represent these symbolic data in a way that is suitable for comparison with the specification.

In our simulation model, the symbolic values and operations produced during the simulation are used to build a bipartite data flow graph, which is similar to the representation used for the design specification. A vertex is created in the data flow graph whenever a new symbolic value or operation is produced during the simulation. If a symbolic value is the result of an operation performed by a functional unit, the operation becomes its direct predecessor. Similarly, the symbolic values which appear at the input ports of a functional unit become the direct predecessors of the operation being performed.

Our model is different from the early works [Dar79, Cor81] on symbolic simulation at the RT level in the following ways:

1. The overhead to propagate the algebraic expressions is eliminated since we focus on merely collecting the actual data operations and data transfers that occur in the data path.

2. A powerful algebraic manipulator is not required since we do not try to simplify the expressions during the simulation. Instead, the data flow graph representing the simulation result is compared with the specification using the graph-isomorphism property.

These differences are the reason that symbolic simulation can be effectively applied to solve our problem.

Besides the data flow graph which represents the computations performed by the design, the simulator can also check the bindings between the specification and the implementation. This is because the occurrence of a symbolic value or operation, the structural component in which it take place, and the current simulation time constitute a binding which represents what is actually happening during the simulation. By collecting the actual operation and value bindings from the simulation and comparing them with those specified in the DDS, we are able to determine which design decisions are causing the problem if the design fails.

6.3.3 Examples

In this section, we will use a single-path example and a multiple-path example to illustrate our hybrid symbolic/numeric simulation.

A simple RTL data path and its controller's state-transition graph without conditional branches are shown in Figure 6.8. Initially, the contents of Reg1 and Reg2 are *unknown*. Four symbolic values a, b, c and d are applied to the input ports of the data path before simulation. At the first cycle, since the *enable* signals of Mux1 and Mux2 are 0, the left input of Mux1 and Mux2 are selected and a and b propagate to the inputs of Adder. Hence, an addition operation on a and b is

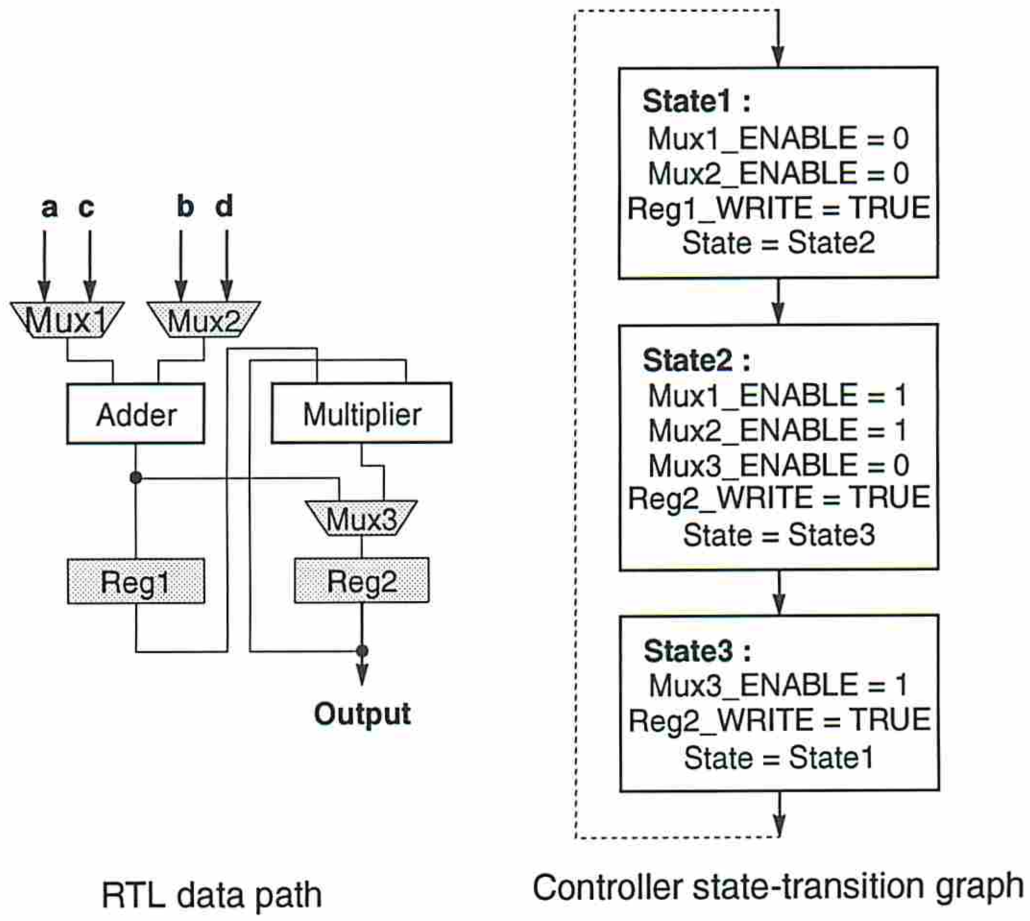
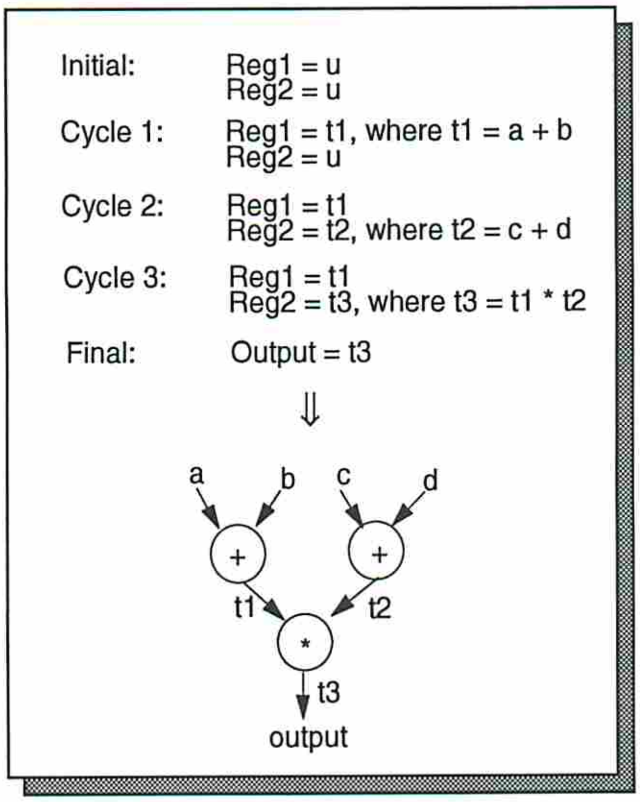


Figure 6.8: An example of a synthesized RTL design without conditional branches

performed by the data path, and a new symbolic value $t1$ which is the result of the addition operation is created at the output of Adder. Since the *write* signal of Reg1 is 1, $t1$ is stored in Reg1 at the end of the cycle. Figure 6.9 summaries the simulation result of this single-path example. As we described in Section 6.3.2, the



Initial:	Reg1 = u
	Reg2 = u

Cycle 1:	Reg1 = t1, where t1 = a + b
	Reg2 = u

Cycle 2:	Reg1 = t1
	Reg2 = t2, where t2 = c + d

Cycle 3:	Reg1 = t1
	Reg2 = t3, where t3 = t1 * t2

Final:	Output = t3

Figure 6.9: The simulation result of the single-path example

symbolic values and operations produced during the simulation are represented by a data flow graph as shown in the bottom half of Figure 6.9.

Figures 6.10 shows a synthesized RTL design with two possible execution paths, $1-2-4-5$ and $1-2-3-5$. For each execution path, a symbolic value $t1$ which is the result of a comparison ($>$) operation on p and q is produced and stored in Reg1 at the end of the first cycle. At the second cycle, since Reg1 now contains $t1$, the path condition is set to $t1 = 1$ for the execution path $1-2-4-5$ and $t1 = 0$ for

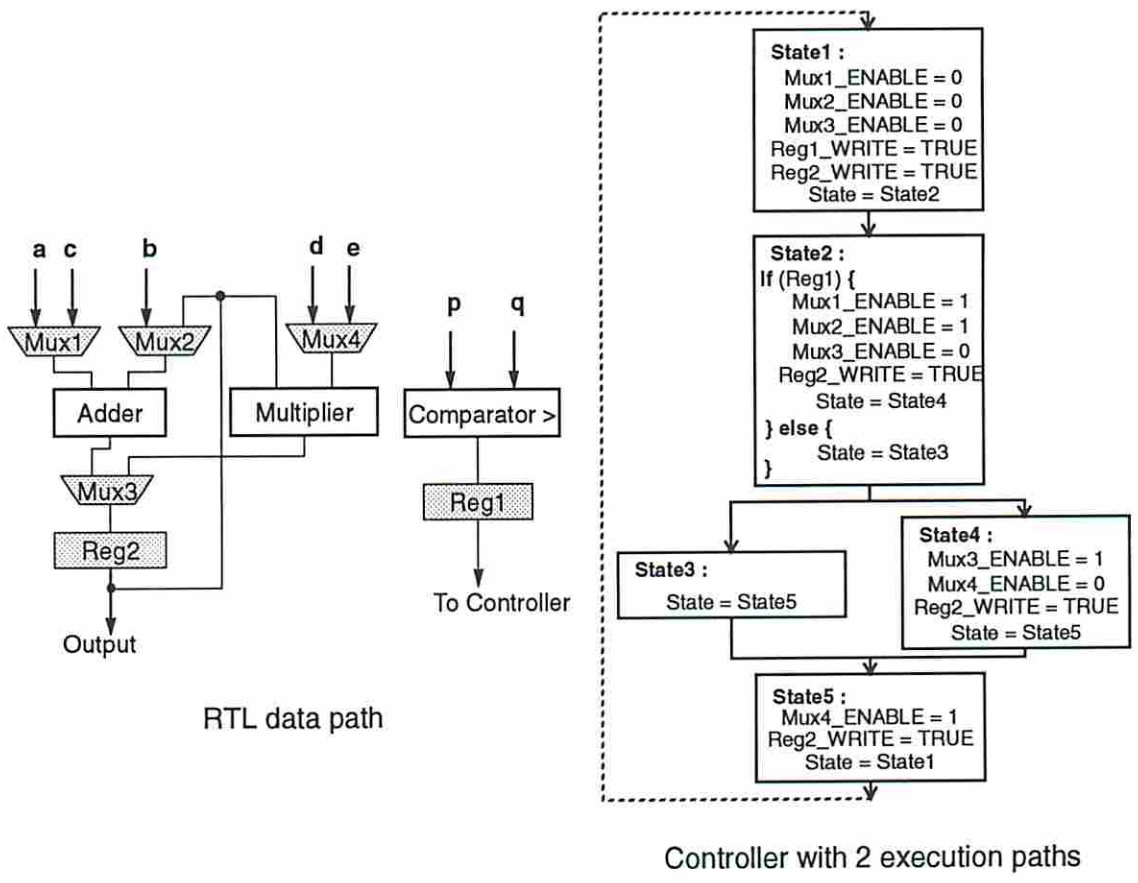

a
c
b
d
e
p
q
Mux1
Mux2
Mux4
Adder
Multiplier
Comparator >
Mux3
Reg1
Reg2
To Controller
Output
RTL data path
State1 : Mux1_ENABLE = 0 Mux2_ENABLE = 0 Mux3_ENABLE = 0 Reg1_WRITE = TRUE Reg2_WRITE = TRUE State = State2
State2 : If (Reg1) { Mux1_ENABLE = 1 Mux2_ENABLE = 1 Mux3_ENABLE = 0 Reg2_WRITE = TRUE State = State4 } else { State = State3 }
State3 : State = State5
State4 : Mux3_ENABLE = 1 Mux4_ENABLE = 0 Reg2_WRITE = TRUE State = State5
State5 : Mux4_ENABLE = 1 Reg2_WRITE = TRUE State = State1
Controller with 2 execution paths

Figure 6.10: An example of a synthesized RTL design with conditional branches

$1-2-3-5$. The simulation result of this multiple-path example is summarized in Figure 6.11. The data flow graph obtained from the simulation for each execution

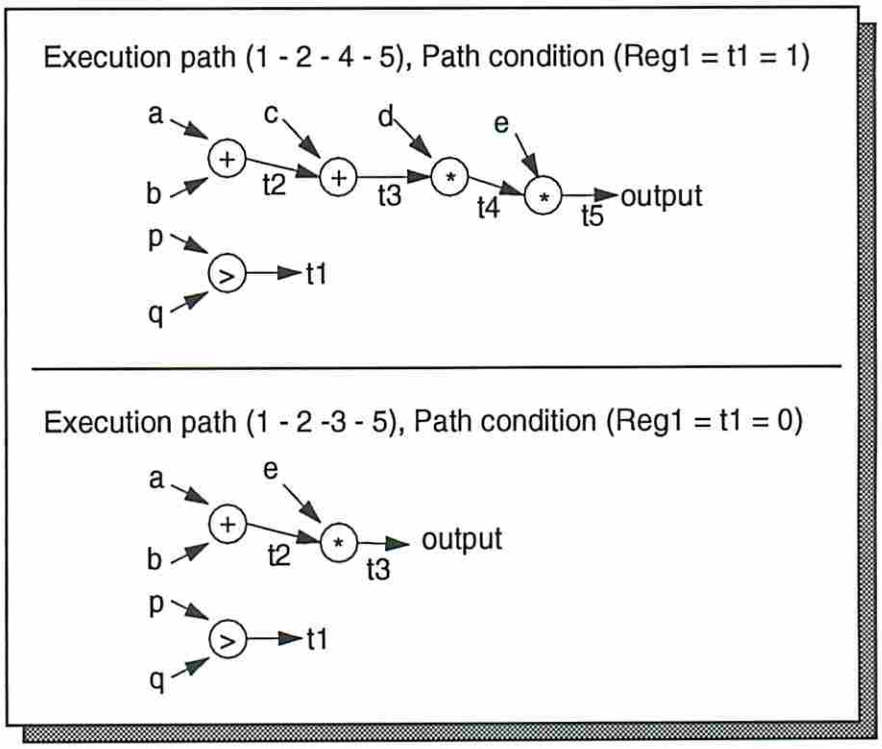

Execution path (1 - 2 - 4 - 5), Path condition (Reg1 = t1 = 1)
a
b
c
d
e
t2
t3
t4
t5
output
p
q
t1
Execution path (1 - 2 -3 - 5), Path condition (Reg1 = t1 = 0)
a
b
e
t2
t3
output
p
q
t1

Figure 6.11: The simulation result of the multiple-path example

path will be compared separately with the given design specification $CDFG_S$ under the same path condition. This comparison procedure will be discussed in the following section.

6.4 Graph-Based Behavior Comparison

In our approach, the behavior comparison is based on the data flow graph model. Since the design specification is already represented in a CDFG which is a superset of this model, only the behavior of the structural implementation has to be translated into this model. The hybrid symbolic/numeric simulation described in

Section 6.3 performs the translation task for us. Therefore, the verification of the RTL implementation I becomes the problem of comparing two CDFGs, $CDFG_S$ derived from the input specification and $CDFG_I$ derived during the simulation. $CDFG_I$, however, is actually a set of data flow graphs (DFG), each of which corresponds to the result of simulating I for one of its execution paths.

Because the design I is the result of a mapping from $CDFG_S$, there exists a strong relationship between $CDFG_S$ and $CDFG_I$ (see Section 6.1.1). In fact, we will show that there is an isomorphic property between the two. Consequently, a graph-matching procedure based on this property has been developed to compare them efficiently.

6.4.1 The Isomorphic Property

From Section 6.1.1, we know that the RTL implementation I, if mapped correctly, will have several properties. In summary, I will perform the required computations specified by $CDFG_S$ for every execution instance. The computations are done by making sure the input values of each required operation are available at the corresponding input ports of the designated functional unit which is configured properly.

Let DFG_I be the result of simulating I for one execution instance under the path condition *pc*. If the specification $CDFG_S$ is interpreted symbolically under the same path condition *pc*, the result is a data flow graph DFG_S such that the predicate of each operation in DFG_S is evaluated to $true$ under *pc*. For example, Figure 6.12 shows a $CDFG_S$ with conditional branches. Two DFG_S extracted from $CDFG_S$ under $t1 = 1$ and $t1 = 0$ are shown in the right hand side of this figure.

Before we show the isomorphic property between DFG_S and DFG_I, we will first establish the correspondence for all their primary input/output values. This

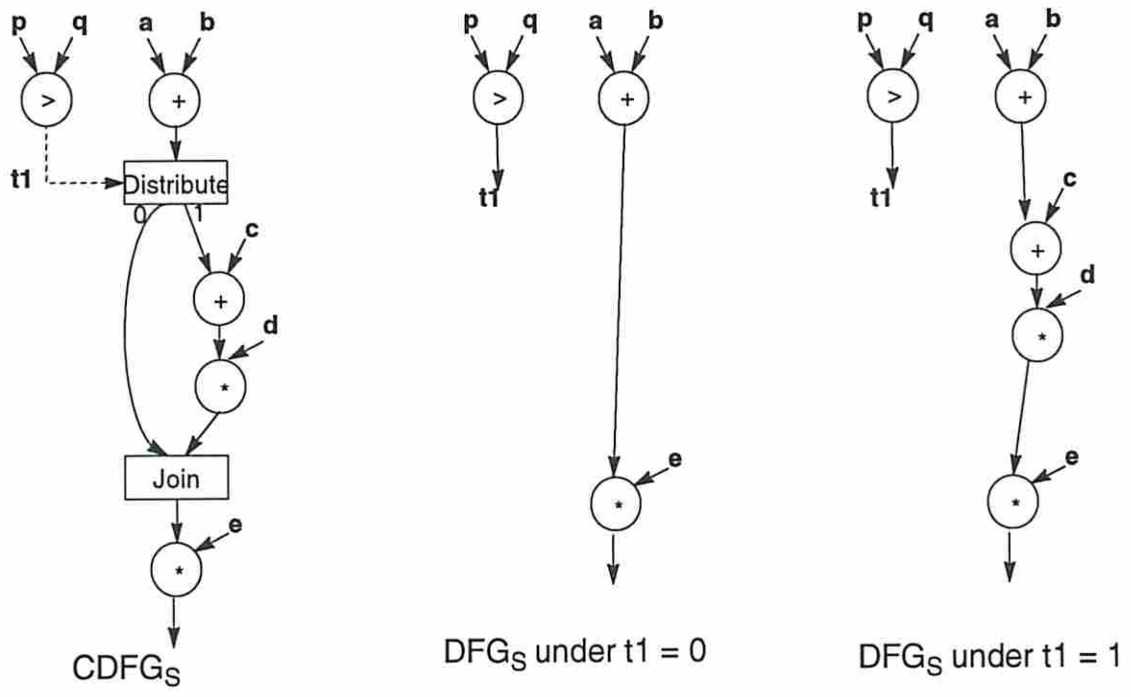


Figure 6.12: An example of extracting DFG_Ss from a $CDFG_S$

correspondence is important because it provides the starting point to compare these two graphs.

Lemma 6.2 *There exists an one-to-one correspondence between DFG_S and DFG_I for the primary input/output values.*

Proof: The primary input values are applied to I according to the input protocol. For each primary input value in_S of DFG_S, there exists an input port *iport* of I and some period of time $[t_s, t_e]$ such that a symbolic value in_I, which corresponds to in_S, is created and applied to *iport* externally from t_s to t_e during the simulation. Therefore, in_I is in DFG_I as a primary input value and it is assumed to correspond to in_S by the input protocol.

Similarly, for each primary output value out_S of DFG_S, there exists an output port *oport* of I and some time t such that a symbolic value out_I is read from *oport* at time t. Hence, the symbolic value out_I is in DFG_I and is assumed by the output protocol to correspond to out_S. □

Intuitively, the isomorphic property between DFG_S and DFG_I exists because I is synthesized in such a way that each required operation in DFG_S will be performed by a designated functional unit at some time and every data dependency will be preserved by establishing a proper interconnection. Hence, if I is synthesized correctly, each corresponding primary output of DFG_S and DFG_I should have similar geometric properties, which is explained by the following theorem.

Theorem 6.3 *For each pair of the corresponding primary output values (out_S, out_I) of DFG_S and DFG_I, the cones*[3] *of out_S and out_I are* isomorphic.

[3]A cone of a vertex v in a graph $G = (V, E)$ is a subgraph $C = (V', E')$ such that

- $V' = \{ v \} \cup predecessors(v)$
- for all v_1, v_2 in V', if edge (v_1, v_2) in E then (v_1, v_2) is also in E'.

Proof: From Properties 6.1 and 6.2, we know that for each operation opr_S in DFG_S, there exists a functional unit u of the design I and some time t such that u is configured to perform the operation type of opr_S; otherwise, a synthesis error occurs.

Hence, if opr_S is a required computation, a list of symbolic values in DFG_I which corresponds to the input values of opr_S in DFG_S, must appear at the respective input ports of u at time t so that an operation opr_I which is equivalent to opr_S is performed. Consequently, a list of new symbolic values which corresponds to the output values of opr_S in DFG_S will be produced in DFG_I.

In other words, there exists a one-to-one mapping[4] from DFG_S to DFG_I for all the operations and values in DFG_S.

Let C_S and C_I be the cones of out_S and out_I respectively. We claim that there exists a one-to-one and *onto* relation between C_S and C_I. This relation is one-to-one as we have discussed earlier. If this relation were not *onto*, there would exist either a vertex or a edge in C_I which did not have a counterpart in C_S.

Case I. Let $v_{I,1}$ be the vertex that has no correspondence.

Since $v_{I,1}$ is a predecessor of out_I, there exists a path in C_I from $v_{I,1}$ to out_I. In this path, there must be an edge $(v_{I,2}, v_{I,3})$ such that $v_{I,3}$ corresponds to $v_{S,3}$ in C_S but $v_{I,2}$ does not have a counterpart in C_S as shown in Figure 6.13. Hence, $(v_{I,2}, v_{I,3})$ is an incident edge of $v_{I,3}$ which does not correspond to any of $v_{S,3}$. However, the correspondence between $v_{I,3}$ and $v_{S,3}$ implies that there is a one-to-one correspondence for all their incident edges. Therefore, we have a contradiction.

Case II. Let $(v_{I,1}, v_{I,2})$ be the edge in C_I that does not have a counterpart in C_S.

From Case 1, we know that every vertex in C_I must have a counterpart

[4]It is not necessary an *onto* mapping.

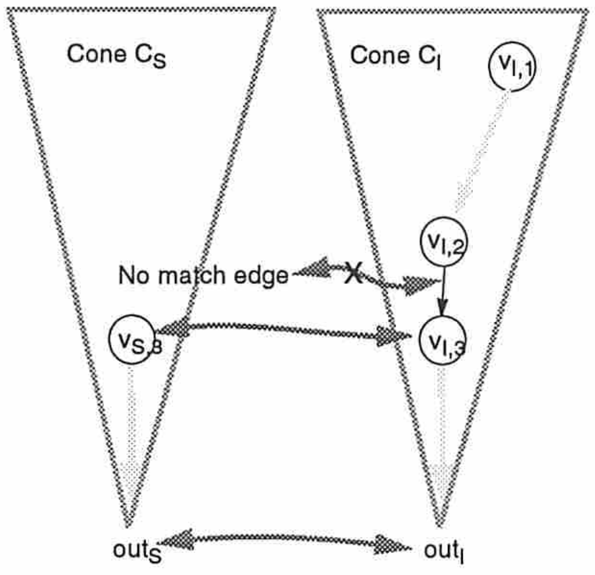


Figure 6.13: A vertex in C_I has no correspondence in C_S

in C_S. Let $v_{S,2}$ be the vertex in DFG_S that corresponds to $v_{I,2}$. Then, $v_{I,2}$ has an incident edge $(v_{I,1}, v_{I,2})$ which does not correspond to any of $v_{S,2}$. Therefore, this edge do not exist.

Thus, there exists a one-to-one and onto relation between C_S and C_I for all the vertices and edges; i.e., C_S and C_I are isomorphic. □

The isomorphic property between C_S and C_I not only implies that there is a one-to-one correspondence between their vertices and edges such that the incidence relationship is preserved, but also requires that each pair of corresponding vertices are compatible. In other words, if the corresponding vertices are operations, they must be of the same type. If they are values, they have same bitwidths.

Figure 6.14 shows the DFG_S and DFG_I obtained (under the path condition $t1 = 1$) respectively from the specification and simulation of the multiple-path example given earlier. The isomorphic property between the cones of the outputs of DFG_S and DFG_I can be seen easily.

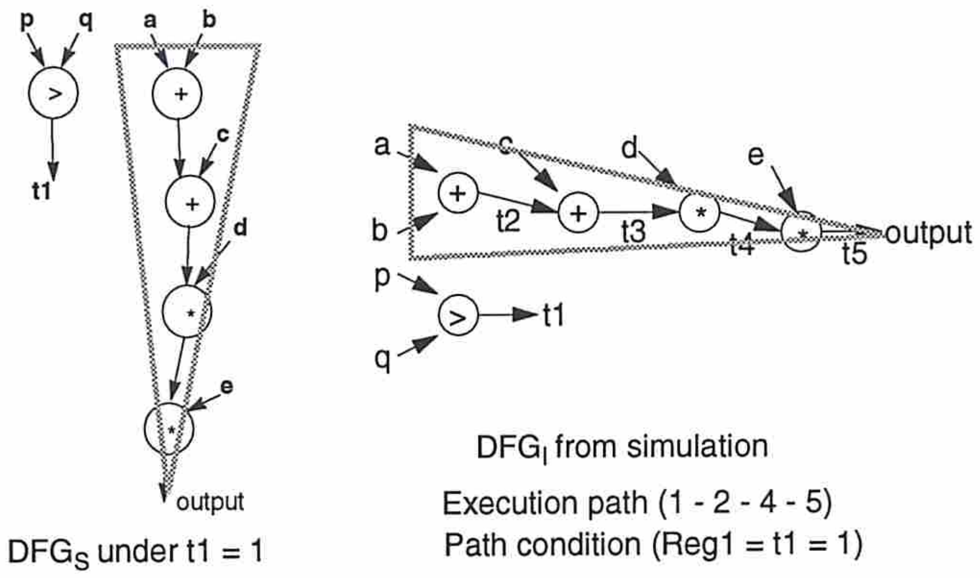


Figure 6.14: An example to show the isomorphic property between the cones of two corresponding outputs of DFG_S and DFG_I

6.4.2 A Graph Matching Procedure

Knowing that there is an isomorphic property between $CDFG_S$ and $CDFG_I$, it becomes straightforward to develop a method for behavior comparison. In fact, all we need to do is to *check whether or not the cones of their corresponding output values are isomorphic for all the execution paths.*

Unlike the general isomorphism problem in graph theory, which is still an important unsolved problem, it is much easier to check the isomorphic property between the cones of the corresponding output values because the following reasons:

1. The correspondences of the primary input and output values of the cones are known in advance.

2. The correspondences of two operations can be established as soon as they are determined to be of the same type and all their input values are equivalent.

The first reason gives us the starting point to compare the cones. The second one enables us to perform the comparison in an iterative-improvement way which establishes the correspondences between the cones incrementally from the primary inputs to the outputs. For example, Figure 6.15 shows two small cones to be compared. Once we know that a and b are equivalent to x and y respectively, the correspondences of two addition operations and their outputs can be established as shown by arrows 3 and 4 in the figure.

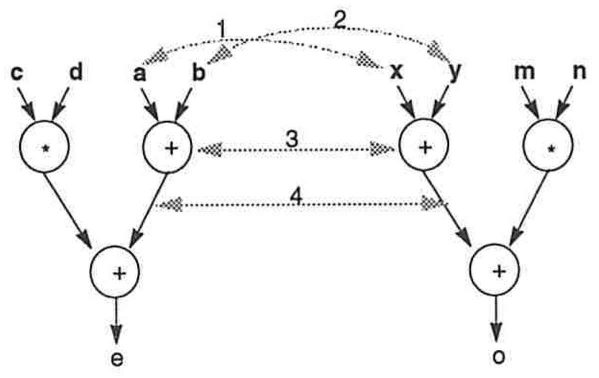


Figure 6.15: An example of matching operations and values between two cones

In what follows, we will present a polynomial-time procedure for checking the isomorphic property between the cones of two corresponding output values. Let out_S and out_I of DFG_S and DFG_I be a pair of corresponding output values and let C_S and C_I be their respective cones to be checked. The following procedure will return true if C_S and C_I are equivalent; otherwise, *false* is returned.

cones_equiv_check(C_S, C_I)

1. Create an attribute for each vertex in C_S and C_I and initialize it to nil.

2. For each pair of corresponding primary input values of C_S and C_I, give their attributes a unique identifier.

3. Find all operations in C_S or C_I such that

    - their attribute is still nil; and
    - the attributes of all their input values are not nil.

   Insert these operations into a ready list L_{ready}.

4. Find all the operations in L_{ready} such that

    - they are of the same type; and
    - all of their corresponding input values have the same attributes.

   If found then

    (a) Remove these operations from L_{ready}.

    (b) Give the attributes of these operations a unique identifier.

    (c) For each set of the corresponding output values of these operations, give their attributes a unique identifier as well.

5. Repeat step 3 until no change can be made.

6. If there exists a vertex in C_S or C_I whose attribute is still nil, return $false$. Otherwise, return $true$.

The worse-case time complexity of this procedure is $O(n^2)$, where n is the number of operations in a cone. The average complexity, however, is much lower because the number of operations in the ready list is usually only a small fraction of the total operations in the cones and a separate ready list can be created for each function type to further reduce the number of operations to be matched.

The number of possible execution paths of a design to be verified, however, is $O(2^c)$, where c is the number of conditional branching constructs in the design behavior. Although the number of execution paths is exponential with respect to c, c is relative small and manageable as compared to the total operations in the design behavior.

6.5 Experiments

In order to show the effectiveness of our approach, we performed a number of experiments with the designs synthesized from the USC ADAM system. In fact, our preliminary experiments immediately identified that the controllers generated for these designs by the early version of the ADAM control signal generator (CSG) [WP92] were incorrect. CSG was then revised using the FINESSE system from Cascade Design Automation as the back-end FSM synthesis tool.

Shortly after CSG was revised, we experimented with a non-pipelined AR filter. MAHA [PPM86] was used for scheduling and MABAL [KP89] for datapath allocation and binding. This design is characterized as follows:

- It has 4 time steps.
- There are 26 input ports and 2 output ports and both the input and output values are not latched.
- The data path consists of 6 multipliers, 4 adders, 6 registers, 14 3-to-1 and 5 2-to-1 multiplexors.
- A two-phase non-overlapping clocking scheme is used.

The analysis of the controller generated by CSG resulted in only one execution path with four states. The experiment was carried out by holding the input values (symbolic) at the input ports during the execution and obtaining two output values from the output ports at the end of the 4th clock cycle. The cones of these two output values, as shown in Figure 6.16, were extracted from the data flow graph built during the simulation. These two cones were compared correctly with the original data flow graph shown in Figure 6.17.

We also experimented with a robot arm controller whose control flow is much more complex than the previous one. The control data flow graph of the robot-arm

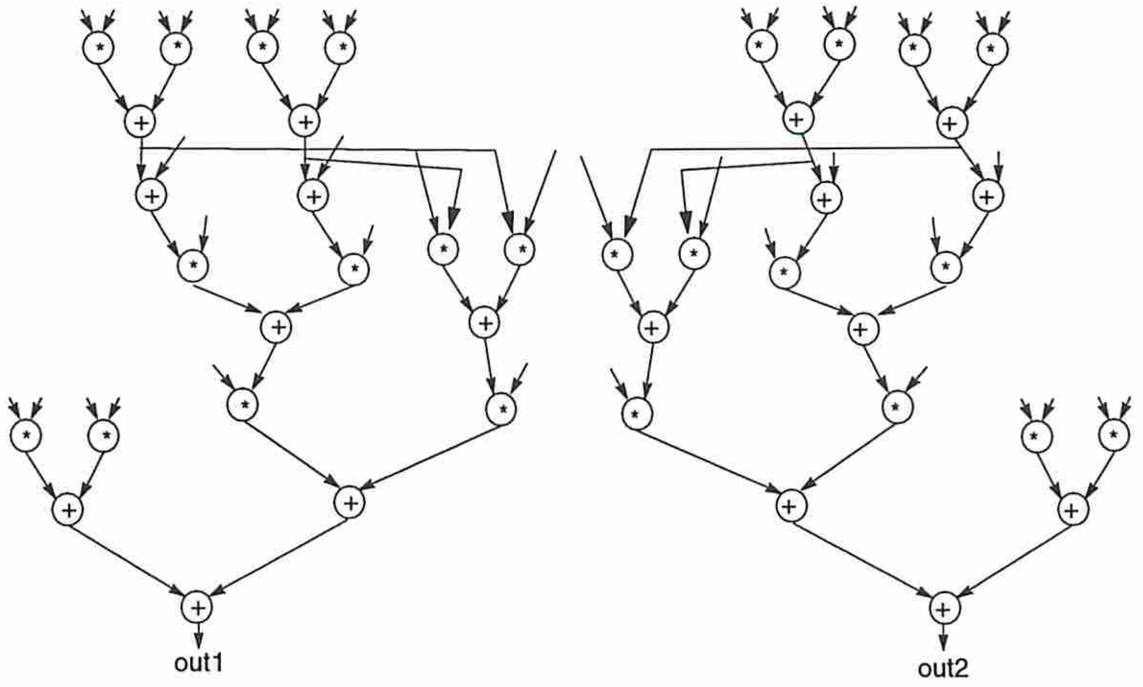


Figure 6.16: The cones of two output values obtained from the hybrid simulation of a non-pipelined AR filter

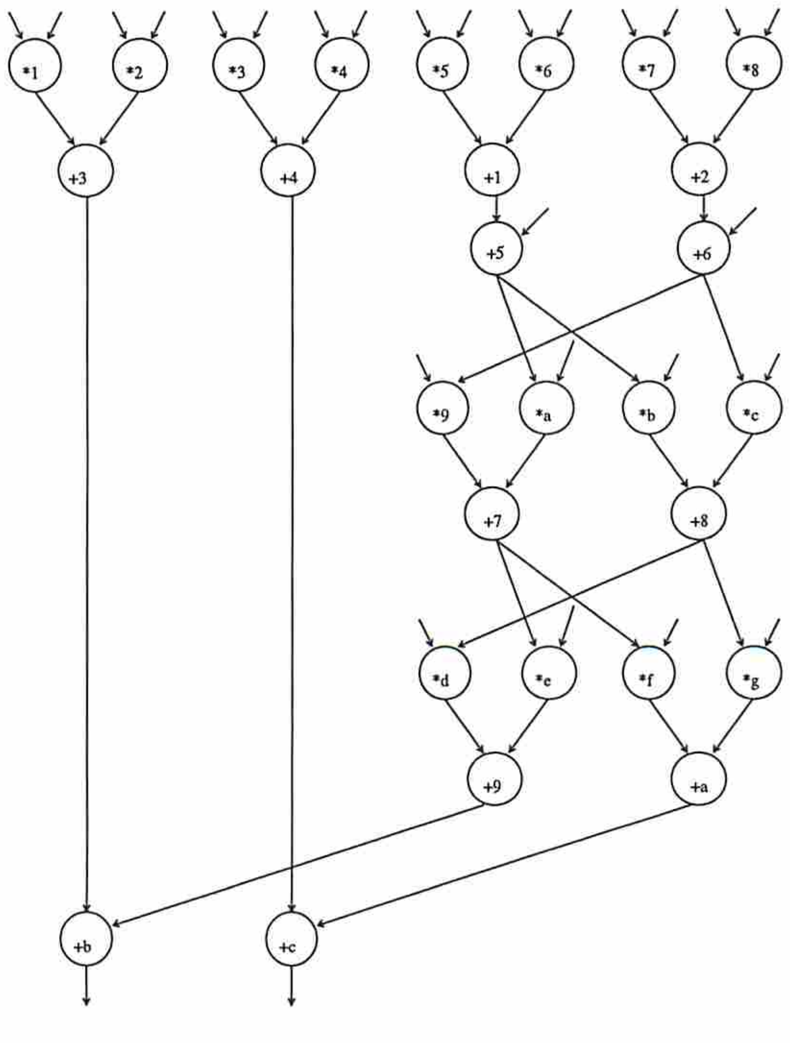


Figure 6.17: The data flow graph of the AR filter example

controller can be found in Figure 3.18. The design was synthesized in a similar way except we required the inputs values to be latched. The RTL implementation has 12 time steps, 16 possible execution paths, 25 input and 14 output ports. The controller was generated by CSG using 2 status registers. The verification of this RTL implementation was done by comparing 14 pairs of cones for each execution path. From this experiment, some design inefficiencies were found:

- All the constant values were required to be supplied externally, which results in inefficient use of input ports.
- Conditional values were unnecessarily routed to the output ports.
- Some of the input values were not latched as specified.

6.6 Analysis of the General RT-Level Design Verification Problem

As we have seen from previous sections, the design specification and its synthesized RTL implementation are not independent. Utilizing the links between the two, the problem of verifying synthesized RTL designs become *tractable*. In this section, we will study the difficulty of the general RTL design verification problem, where the specification and its implementation are considered to be two independent entities.

The RTL verification problem, in the most general form, can be described as follows:

Given:

- a behavioral design specification S.
- an RTL implementation I.

Goal:

show the behaviors B_S and B_I of S and I respectively are *functionally equivalent.*

The specification S specifies what the system should do, and it is usually described functionally in some HDL (Hardware Description Language). On the other hand, the implementation I, typically described by a netlist, models the system structurally as an interconnection of RTL components. The verification task is to show that, for all feasible inputs, the outputs produced by I (B_I) are equivalent to the outputs computed by S (B_S) [McF93]. The behaviors B_S and B_I are regarded as the way the system or its components interact with their environment, i.e., the mapping from inputs to outputs [MPC88]. However, if the system possesses sequential behavior[5], B_S and B_I describe the mapping from "sequences" of inputs to "sequences" of outputs. Therefore, the definition of equivalence of two behaviors has to be modified accordingly.

Since S and I are given in different languages, it is necessary to translate both of them into a model in which their behaviors can be compared. For example, we can simulate both S and I using their respective simulators for all feasible inputs, in theory, to get B_S and B_I and verify their output correspondence. Alternatively, we can describe both S and I in a formalism in which a theorem-prover can be used, hopefully, to prove that B_S and B_I are equivalent. Whichever way we use, a "reasonable" verification model representing the behaviors must be capable of expressing every design in the functional model of S and in the RTL model of I. Figure 6.18 shows the relationships between the verification model and the associated functional and RTL models.

[5]The output at any point depends on the current state which in turns relies on the past history of inputs [McF].

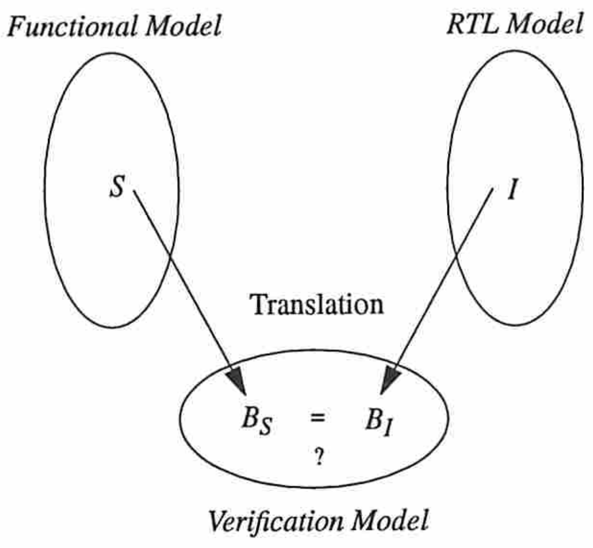


Figure 6.18: The verification model and its relationships with the functional and RTL models.

Definition 6.3 *A verification model is called* complete *if for each element ϕ in the functional and RTL models there exists a counterpart element ψ in the verification model such that ϕ and ψ are functionally equivalent.*

Definition 6.4 *A complete verification model is* feasible *if we can find a translation function to map each ϕ in the functional and RTL models to its counterpart ψ in the verification model.*

In other word, a *complete* verification model is at least as powerful as the functional and RTL models in terms of the expressive capability, and the model is *feasible* (useful) only if we can effectively translate all the designs into it.

If we do not consider the fact that I is derived from S, B_S and B_I simply correspond to two independent points in a complete verification model. Let Ω_V be the complete verification model. Solving the general RTL verification problem is then equivalent to finding a decision function f such that for all $\psi_i, \psi_j \in \Omega_V$

$$f(i,j) = \begin{cases} 1 & \text{if } \psi_i \equiv \psi_j \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

In what follows, we will show that it is not possible to find such a decision function no matter which complete verification model we use. However, some background computation theory needs to be introduced first.

Informally, a function is a *partial recursive function (prf)* if it is effectively "computable" [MY78]. In other words, given a definition of a partial recursive function we can produce an algorithm, e.g. write a C program, to compute it. A *programming system* is a list of programs $\phi_0, \phi_1, \ldots$ which includes all of the *prfs*. A programming system PS is an *acceptable programming system (APS)* if and only if the following conditions are met.

- For all *prf* f, there exists an index i such that $\phi_i = f$. That is, there is at least a program in PS for each *prf*.
- For all index i, ϕ_i is a *prf*. In other words, every program in PS is a *prf*.
- There exists a *universal* program ϕ_u such that for all i and x $\phi_u(i,x) = \phi_i(x)$, where x is an function argument over N (natural numbers).
- There exists a total recursive function c such that $\phi_{c(i,j)} = \phi_i \circ \phi_j$ for all i and j, where $\circ$ denotes function composition.

APSs are used in computation theory to model the common characteristics of "reasonable" programming systems mathematically. Hence, any results applied to APSs also hold for all "reasonable" programming systems, and certainly for all existing general purpose programming languages like C, PASCAL, and VHDL [MY78].

Now, we can introduce the notion of undecidable (algorithmically unsolvable) problems concerning APSs. Let N be the set of natural numbers.

Definition 6.5 *For all $\Gamma \subseteq N$, the function $C_\Gamma : N \rightarrow \{0,1\}$ is called the characteristic function of Γ if and only if*

$$C_\Gamma(i) = \begin{cases} 1 & \textit{if } i \in \Gamma \\ 0 & \textit{otherwise} \end{cases}$$

Definition 6.6 *For all $\Gamma \subseteq N$, Γ is* decidable *if and only if C_Γ is a* prf.

The following lemma describes an important *undecidable* set which will be used to show the undecidability of the general RTL verification problem.

Lemma 6.4 *For all APS $\{\phi\}$, the set $V = \{< i,j >| \phi_i \equiv \phi_j\}$[6] is undecidable.*

The proof of this lemma can be found in [MY78]. Basically, Lemma 6.4 says that there is no algorithm for deciding whether or not two arbitrary programs in any APS are equivalent.

Finally, we can establish the main result of this section with the following theorem.

Theorem 6.5 *The general RTL verification problem is* algorithmically unsolvable *for all complete verification models.*

Outline of proof: Recalling earlier discussion, we know that solving the general RTL verification problem is equivalent to finding a decision function f to determine whether or not two behaviors in a verification model are equivalent as defined in Equation 6.1. Let Ω_V be a complete verification model. The general RTL verification problem can be represented by the following set:

$$V = \{< i,j >| \psi_i \equiv \psi_j \text{ for all } \psi_i, \psi_j \text{ in } \Omega_V\}$$

[6] $<,>$ is any pairing function which can establish an effective one-to-one correspondence between the 2-tuples in $N \times N$ and the numbers in N. For example, $f(x,y) = 2^x(2y+1) - 1$.

Hence, the decision function f is actually the characteristic function of the set V.

From Lemma 6.4, we know that V is undecidable for all APSs. Therefore, if Ω_V is an APS, f is not a partial recursive function. In other words, there exists no such a program (f) which can decide whether two arbitrary behaviors in Ω_V are equivalent or not. Here we argue that any *complete* verification model Ω_V should satisfy the definition of an APS[7] since its associated functional model for design specifications is based on some hardware description language such as VHDL or C which certainly qualifies as a "reasonable" programming system. Hence, the general verification problem is algorithmically unsolvable for all *complete* verification models. □

6.7 Summary

In this chapter, we have identified several properties of automatically synthesized RTL designs. From these properties, we found that the correctness of a synthesized RTL design is really determined by whether or not the data operations and transfers occurring in the data path will conform to the given design specification (a control data flow graph). Consequently, a hybrid symbolic/numeric simulation was introduced in this chapter to extract the behaviors of synthesized RTL designs into data flow graphs which can be compared directly with their specifications. We have also shown that there exists an isomorphic property between the data flow graphs obtained from the specification and the hybrid simulation respectively. A polynomial time procedure based on this isomorphic property was presented for behavior comparison.

The experiments we have conducted show that the correctness of synthesized designs cannot be taken for granted, especially when the synthesis tools are in their

[7]A formal proof will need to show this argument mathematically. However, we feel that this enormous work will lead us out of focus of this research.

early releases. Hence, we believe that the verification methodology presented in this chapter is valuable not only for obtaining confidence on the correctness of the synthesized RTL designs but also for identifying unforeseen problems in synthesis tools.

In the future development, techniques to handle loops which are not unrolled are needed, in which case the controller's state-transition graph becomes cyclic. When there are cycles in a state-transition graph, our definition of possible execution paths has to be changed. One solution is to traverse each cycle only once; therefore, the number of possible execution paths will not be affected by the actual iterations taken by the loops. This approach avoids the problem of determining the conditions under which a loop would terminate during the symbolic simulation. Additionally, we need a method to check that the values produced by the loop body are fed back correctly for subsequent iterations.

Chapter 7

Conclusion

As we have mentioned earlier, most high-level synthesis systems address the task of transforming a behavioral description of a design into an equivalent register-transfer level structure, where the design behavior is often assumed to contain a single process and the structure is to be implemented on a single chip. This thesis has attempted to address many design issues associated with the synthesis of multiple-chip systems with multiple concurrent processes. A system-level synthesis methodology is proposed in this thesis using system-level partitioning, multiple-process scheduling/allocation and RTL design verification to reduce the design time required for finding good quality system designs. Below we summarize the contributions of the research presented in this thesis and discuss some future research topics in each of these areas.

7.1 System Partitioning

In this thesis, a new approach for the system partitioning problem is presented. An important aspect of this approach is that partitioning is performed at the process level where the number of objects to be considered are far fewer than those at the operation level and the functional boundaries specified by the designers are preserved. Furthermore, the exploration of process design alternatives is done concurrently with partitioning, and the chip count and the chip capacities (area and pins) are not simply a number of given constraints to be satisfied; instead, they are traded off according to the available chip packaging options. We believe that system partitioning at a higher level of granularity such as processes and procedures will become more and more advantageous and necessary as both chip capacities and system complexity keep increasing.

Two partitioning methods were described in this thesis. First, an MILP formulation was presented and implemented in a prototype tool called ProPart. Several experiments including a JPEG image compression system were performed to demonstrate the usefulness of this tool. A genetic-search technique was also described to find acceptable solutions in a more manageable run time. The genetic-search technique was found to be a promising optimization technique for system partitioning to handle complex issues like yield and power.

In the future, the communication tradeoffs need to be done more thoroughly, considering pin sharing among different data transfers and considering different pin/buffer requirements at the sender and the receivers. Non-uniform technology should also be taken into account. In other words, the processes could be partitioned onto a number of *mixed* components, which may be ASICs, pre-designed parts or programmable devices. These components in turns could be distributed among various packaging devices such as chips, multi-chip modules (MCM) and boards in order to satisfy or optimize the constraints on cost, size, yield, power, and other design characteristics.

7.2 Multiple-Process Synthesis

In Chapter 5, we presented a new approach for synthesis of designs with unbounded-delay operations under timing constraints and with multiple communicating processes. Compared to *relative scheduling*, this approach allows us to trade off between performance and resource requirements during scheduling as well as to reduce the control overhead.

Our synthesis approach is based on a notion of single-threaded processes. In other words, each process corresponds to one thread of control and there exists a sequential order among the unbounded-delay operations in the process description. We found that scheduling of a single-threaded process can be done statically in terms of control steps by preserving the sequential order of unbounded-delay operations embedded in the process description. Two important issues were also addressed; namely, how to satisfy detailed timing constraints when unbounded-delay operations are present and how to synchronize inter-process communication.

An advantage of our approach is that it is compatible with many good synthesis techniques originally developed for designs with only fixed-delay operations. In Chapter 5, we demonstrated our scheduling method using an ILP formulation and also proposed a heuristic procedure modified from *freedom-based scheduling.*

There is still much work to be done here. Though our approach can be used as the basis for hierarchical scheduling to handle design with conditional branches and loops, timing constraints can only be applied on the operations within the same level of hierarchy. A thorough analysis method will be needed to distribute the constraints across the hierarchy or a non-hierarchical approach should be developed. The handling of inter-process communication needs to be extended to allow *one-to-many* and *buffered* communication events. Furthermore, the inter-process communication scheme given by the designers may be too conservative and may contain some blocking communication events that can be converted to non-blocking ones. Techniques to automatically determine a minimal-cost inter-process communication scheme during synthesis or as a separate step should be investigated.

7.3 RTL Design Verification

In the research on design verification, we have identified several important properties of synthesized RTL designs. As a result, we found that the correctness of a synthesized RTL design is mainly determined by whether or not the data operations and transfers occurring in the data path conform to the given design specification (a control data flow graph). Therefore, we presented a hybrid symbolic/numeric simulation to extract the behaviors of synthesized RTL designs into data flow graphs which can then be compared directly with their specifications. We have also shown that there exists an isomorphic property between the data flow graphs obtained from the specification and the hybrid simulation respectively. A polynomial time procedure based on this isomorphic property was given for behavior comparison.

The advantage of this verification approach is that it not only can formally verify the synthesized data path but also can faithfully exercise the control path, which is also a major source of design errors. The value of this approach was shown by its ability to identify problems with the early CSG tool in the ADAM system. This also proves that the correctness of synthesized designs cannot be guaranteed, especially when the synthesis tools are in their early releases.

In the future, better handling of loops is definitely needed. When there are loops which are not fully unrolled, the controller's state-transition graph becomes cyclic. Therefore, our definition of possible execution paths has to be changed. More research should be done on diagnosis of design errors in order to trace the possible causes automatically. Finally, modification of our verification methodology to handle multiple-process designs is also needed.

7.4 Other Contributions

In Chapter 3, we have formulated the behavioral VHDL compilation problem in detail. The techniques for control flow analysis, local/global data flow analysis, and graph generation/optimization were presented. We also discussed the modeling of arrays, input/output and inter-process communication in VHDL as well as in the DDS representation. A prototype software called VHDL2DDS based on these techniques has been developed and is fully operational. VHDL2DDS currently serves as the VHDL front-end of the ADAM high-level synthesis system, and has been used in numerous chip design experiments.

One work done during the course of this research but not discussed in this thesis is on integrating synthesis and test in the USC ADAM project. The goal is to combine tradeoffs in cost, performance and testability during the high-level synthesis process. An EDIF interface[1] [Che91] was developed to serve as the bridge between the ADAM synthesis system and the built-in test system, and to export ADAM's synthesized designs via the standard EDIF format [Eng87]. Hence, a fully automated design path from VHDL behavioral specifications to testable chip layouts was created.

A number of experiments were also performed via the EDIF interface to study the tradeoffs among the performance, cost and testability and to assess what features of the synthesized design have an impact on the design testability. Five example designs were synthesized and made testable. Four of them are the AR filter with different design characteristics and the other one is a robot arm controller. Two AR filter testable designs were laid out using the Seattle Silicon Compiler. From these experiments, a great deal of insight about the synthesis techniques

[1]The EDIF interface is a multi-way design translation system involving three design representations: MABAL [KP89], EDIF [Eng87] and CBASE [GCG+89].

which support testability and the area overhead for making design testable was obtained. The details of these experiments can be found in [NCPB91].

Reference List

[ABF90] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* W. H. Freeman and Company, New York, 1990.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bar81] M. Barbacci. Instruction Set Processor Specification (ISPS): The Notation and its Applications. *IEEE Transactions on Computers*, C-30(1):24–40, January 1981.

[BBB+87] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. In *24th Design Automation Conference*, pages 9–16. ACM/IEEE, 1987.

[BF89] S. Bose and A.L. Fisher. Verifying Pipelined Hardware Using Symbolic Logic Simulation. In *Int'l Conference on Computer Design.* IEEE, October 1989.

[BKM+66] M. Beardslee, C. Kring, R. Murgai, H. Savoj, R. K. Brayton, and A. R. Newton. SLIP: A Software Environment for System Level Interactive Partitioning. In *Int'l Conference on Computer-Aided Design*, pages 280–283. IEEE, 1966.

[Boc82] G. V. Bochmann. Hardware Specification with Temporal Logic: An Example. *IEEE Transactions on Computers*, C-31(2):223–231, March 1982.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry90] R. E. Bryant. Symbolic Simulation - Techniques and Applications. In *27th Design Automation Conference*, pages 517–521. ACM/IEEE, 1990.

[BS92] S. H. Bang and B. J. Sheu. A Multi-Chip Module for Hand-Held Digital Cellular Mobile Telephone. In *Multi-Chip Module Conference*, pages 115–118. IEEE, March 1992.

[CH90] A. Chatterjee and R. Hartley. A New Simultaneous Circuit Partitioning and Chip Placement Approach Based on Simulated Annealing. In *27th Design Automation Conference*, pages 36–39. ACM/IEEE, 1990.

[Che91] C. T. Chen. *Manual Pages for the ADAM EDIF Interface.* Department of Electrical Engineering - Systems, University of Southern California, July 1991.

[Cor81] W. E. Cory. Symbolic Simulation for Functional Verification with ADLIB and SDL. In *18th Design Automation Conference*, pages 82–89. ACM/IEEE, 1981.

[CP88] P. Camurati and P. Prinetto. Formal Verification of Hardware Correctness: Introduction and Survey of Current Research. *IEEE Computer*, 21(7):8–19, July 1988.

[CvE87] R. Camposano and J. van Eijndhoven. Partitioning a Design in Structural Synthesis. In *Int'l Conference on Computer Design*, pages 564–566. IEEE, October 1987.

[Dar79] J. A. Darringer. The Application of Program Verification Techniques to Hardware Verification. In *16th Design Automation Conference*, pages 375–381. ACM/IEEE, 1979.

[Eng87] Engineering Department, Electronic Industries Association. *Electronic Design Interchange Format Version 2 0 0*, 1987.

[FLS+92] H. Fujiwara, M. L. Liou, M. T. Sun, K. M. Yang, M. M. Maruyama, K. Shomura, and K. Ohyama. An All-ASIC Implementation of a Low Bit-Rate Video Codec. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2):123–134, June 1992.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristics for Improving Network Partitioning. In *19th Design Automation Conference*, pages 175–181. ACM/IEEE, 1982.

[Fuh91] T. E. Fuhrman. Industrial Extensions to University High Level Synthesis Tools: Making it Work in the Real World. In *28th Design Automation Conference*, pages 520–525. ACM/IEEE, 1991.

[GCDBP94] P. Gupta, C. T. Chen, J.C. DeSouza-Batista, and A. C. Parker. Experience with Image Compression Chip Design using Unified System Construction Tools. In *31th Design Automation Conference*. ACM/IEEE, 1994.

[GCG+89] R. Gupta, W. Cheng, R. Gupta, I. Hardonag, and M. Breuer. An Object-Oriented VLSI CAD Framework. *IEEE Computer*, 22(5):28–37, May 1989.

[GE92] C. H. Gebotys and M. I. Elmasry. Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis. In *28th Design Automation Conference*, pages 2–7. ACM/IEEE, 1992.

[Geb92] C. H. Gebotys. Optimal Synthesis of Multichip Architectures. In *Int'l Conference on Computer-Aided Design*, pages 238–241. IEEE, 1992.

[GGP+91] M. Genoe, L. Glaesen, E. Proesmans, E. Verlind, and H. De Man. Illustration of the SFG-Tracing Multi-Level Behavioral Verification Methodology, by the Correctness Proof of a High to Low Level Synthesis Application in CATHEDRAL-II. In *Int'l Conference on Computer Design*. IEEE, October 1991.

[GGVN93] D. Gajski, J. Gong, F. Vahid, and S. Narayan. The SpecSyn Design Process and Human Interface. Technical Report TR ICS 93–3, Department of Information and Computer Science, University of California, Irvine, 1993.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[GM90] R. Gupta and G. De Micheli. Partitioning of Functional Models of Synchronous Digital Systems. In *Int'l Conference on Computer-Aided Design*, pages 216–219. IEEE, 1990.

[GM92] R. Gupta and G. De Micheli. System Synthesis via Hardware-Software Co-design. Technical Report CSL–TR–92–548, Department of EE and CS, Stanford University, October 1992.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[GP94] P. Gupta and A. C. Parker. SMASH: A Program for Scheduling Memory-Intensive Application Specific Hardware. In *7th Int'l Symposium on High-Level Synthesis*, May 1994.

[GRVM90] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man. An Efficient Microcode Compiler for Application Specific DSP Processors. *IEEE Transactions on Computer-Aided Design*, 9(9):925–937, September 1990.

[GS84] J. Greene and K. Supowit. Simulated Annealing Without Rejected Moves. In *Int'l Conference on Computer Design*, pages 658–663. IEEE, October 1984.

[Hay90] S. Hayati. *The Synthesis of Control-Dominated Application Specific Integrated Circuits Using Global Based Design Management.* PhD thesis, Department of Electrical Engineering - Systems, University of Southern California, November 1990.

[HD86] F. K. Hanna and N. Daeche. Specification and Verification of Digital Systems using Higher-Order Logic. *IEE Proceeding*, 133(5):242–254, September 1986.

[HH90] L. J. Hafer and E. Hutchings. Bringing up Bozo. Technical Report CMPT TR 90–2, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, March 1990.

[Hil85] P. Hilfinger. A High-level Language and Silicon Compiler for Digital Signal Processing. In *Int'l Symposium on Circuits and Systems*, pages 213–216. IEEE, May 1985.

[HLH91] C. T. Hwang, J. H. Lee, and Y. C. Hsu. A Formal Approach to the Scheduling Problem in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design*, 10(4), April 1991.

[HP92] Y. H. Hung and A. C. Parker. High-Level Synthesis with Pin Constraints for Multiple-Chip Designs. In *29th Design Automation Conference*, pages 231–234. ACM/IEEE, 1992.

[Ins88] The Institute of Electrical and Electronics Engineers Inc. *IEEE Standard VHDL Language Reference Manual*, 1988.

[KFJM92] D. Ku, D. Filo, C. N. Coelho Jr., and G. De Micheli. Interface Optimization for Concurrent Systems under Timing Constraints using Interface Matching. In *6th Int'l Workshop on High-Level Synthesis*, November 1992.

[KL70] W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49:291–307, January 1970.

[KM92] D. Ku and G. De Micheli. Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits. *IEEE Transactions on Computer-Aided Design*, 11(6):696–718, June 1992.

[KP85] D. W. Knapp and A. C. Parker. A Unified Representation for Design Information. In *Int'l Symposium on Computer Hardware Description Languages and their Applications*, 1985.

[KP89] K. Kucukcakar and A. C. Parker. MABAL - A Software Package for Module And Bus ALlocation. In *Int'l Journal of Computer Aided VLSI Design*, June 1989.

[KP93] K. Kucukcakar and A. C. Parker. BEST: Behavioral Area-Delay Predictor. Technical Report CEng 93–24, Department of Electrical Engineering - Systems, University of Southern California, May 1993.

[Kuc91] K. Kucukcakar. *System-Level Synthesis Techniques with Emphasis on Partitioning and Design Planning.* PhD thesis, Department of Electrical Engineering - Systems, University of Southern California, October 1991.

[KW89] D. W. Knapp and M. Winslett. A Formalization of Correctness for Linked Representations of Datapath Hardware. In *IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, November 1989.

[LG88] J. Lis and D. Gajski. Synthesis from VHDL. In *Int'l Conference on Computer Design*, pages 378–381. IEEE, 1988.

[LGP+91] D. Lanneer, G. Goossens, M. Pauwels, J. Van Meerbergen, and H. De Man. An Object-Oriented Framework Supporting the Full High-Level Synthesis Trajectory. In *10th Int'l Symposium on Computer Hardware Description Languages and their Applications*, pages 281–300, 1991.

[LLT69] E. L. Lawler, K. N. Levitt, and J. Turner. Module Clustering to Minimize Delay in Digital Networks. *IEEE Transactions on Computers*, C-18(1):47–57, January 1969.

[LT91] E. Lagnese and D. Thomas. Architectural Partitioning for System Level Synthesis of Integrated Circuits. *IEEE Transactions on Computer-Aided Design*, 10(7), July 1991.

[McF] M. C. McFarland. Practical Lessons in Verification and High-Level Synthesis. AT&T Bell Laboratories, Murray Hill, NJ 07974-2070.

[McF78] M. C. McFarland. The Value Trace: A Data Base for Automated Digital Design. Technical Report DRC-01-4-80, Dept. of Electrical Engineering, Carnegie-Mellon University, December 1978.

[McF86] M.C. McFarland. Using Bottom-Up Design Techniques in the Synthes is of Digital Hardware from Abstract Behavioral Descriptions. In *23th Design Automation Conference*, pages 474–480. ACM/IEEE, June 1986.

[McF93] M. C. McFarland. Formal Verification of Sequential Hardware: A Tutorial. *IEEE Transactions on Computer-Aided Design*, 12(5):633–654, May 1993.

[MK88] G. De Micheli and D. Ku. HERCULES - A System for High-Level Synthesis. In *25th Design Automation Conference.* ACM/IEEE, 1988.

[MKMT90] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus Synthesis System. *IEEE Design and Test of Computers*, October 1990.

[MP83] M. C. McFarland and A. C. Parker. An Abstract Model of Behavior for Hardware Descriptions. *IEEE Transactions on Computers*, C-32(7):621–636, July 1983.

[MPC88] M. C. McFarland, A. C. Parker, and R. Camposano. Tutorial on High-Level Synthesis. In *25th Design Automation Conference*, pages 330–336. ACM/IEEE, 1988.

[MY78] M. Machtey and P. Young. *An Introduction to the General Theory of Algorithms.* the Computer Science Library. Elsevier North Holland, Inc., 1978.

[Nar92] S. Narayan. A Survey of System-Level Specification Languages. Technical Report TR ICS 92–100, Department of Information and Computer Science, University of California, Irvine, 1992.

[NCPB91] C. Njinda, C. T. Chen, A. C. Parker, and M. Breuer. Integrating Synthesis and Test in ADAM. In *IFIP International Workshop on the Relationship Between Synthesis, Test, and Verification*, November 1991.

[Nes87] J. Nestor. *Specification and Synthesis of Digital Systems with Interfaces.* PhD thesis, Carnegie-Mellon University, April 1987.

[NP91] A. Nicolau and R. Potasman. Incremental Tree Height Reduction For High Level Synthesis. In *28th Design Automation Conference*, pages 770–774. ACM/IEEE, 1991.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley Inter-science, 1988.

[OG86] A. Orailogulu and D. Gajski. Flow Graph Representation. In *23th Design Automation Conference*, pages 503–509. ACM/IEEE, 1986.

[PGH91] A. C. Parker, P. Gupta, and A. Hussain. The Effects of Physical Design Characteristics on the Area - Performance Tradeoff Curve. In *28th Design Automation Conference*, pages 530–534. ACM/IEEE, June 1991.

[PK89] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 8(6):661–679, June 1989.

[PP93] H. Park and V.K. Prasanna. Area Efficient VLSI Architectures for Huffman Coding. *Int'l Conference on Acoustics, Speech and Signal Processing*, 1993.

[PPM86] A. C. Parker, J. Pizarro, and M. J. Mlinar. MAHA: A Program for Datapath Synthesis. In *23th Design Automation Conference*, pages 461–466. ACM/IEEE, 1986.

[Pra93] S. Prakash. *Synthesis of Application-Specific Multiprocessor Systems.* PhD thesis, Department of Electrical Engineering - Systems, University of Southern California, 1993.

[RP93] J. Raghavendran and A. C. Parker. Hardware/software tradeoffs in adam. Technical Report CEng 93–28, Department of Electrical Engineering - Systems, University of Southern California, 1993.

[SG92] N. Schraudolph and J. Grefenstette. A User's Guide to GAucsd 1.4. Technical Report CS92-249, University of California, San Diego, July 1992.

[SR89] Y. Saab and V. Rao. An Evolution-Based Approach to Partitioning ASIC Systems. In *26th Design Automation Conference*, pages 767–770. ACM/IEEE, 1989.

[TLW+90] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn. *Algorithmic and Register-Transfer Level*

Synthesis: The System Architect's Workbench. The Kluwer International Series In Engineering and Computer Science. Kluwer Academic Publishers, 1990.

[TW93] A. Takach and W. Wolf. Scheduling Constraint Generation for Communicating Processes. Technical report, Department of Electrical Engineering, Princeton University, February 1993.

[Vah91] F. Vahid. A Survey of Behavioral-Level Partitioning Systems. Technical Report TR ICS 91–71, Department of Information and Computer Science, University of California, Irvine, 1991.

[VG92] F. Vahid and Daniel D. Gajski. Specification Partitioning for System Design. In *29th Design Automation Conference*, pages 219–224. ACM/IEEE, 1992.

[VNG91] F. Vahid, S. Narayan, and D. Gajski. SpecCharts: A Language for System Level Synthesis. In *Int'l Symposium on Computer Hardware Description Languages and their Applications*, 1991.

[Wal91] G. K. Wallace. The JPEG Still Picture Compression Standard. *ACM Communications*, 34(4):31–44, April 1991.

[WGB] T. C. Wilson, G. W. Grewal, and D. K. Banerji. An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. VLSI-CAD Group, Department of Computing and Information Science, University of Guelph, Guelph, Ontario, Canada N1G-2W1.

[Whi93] D. Whitley. A Genetic Algorithm Tutorial. Technical Report CS-93-103, Colorado State University, November 1993.

[WP92] J. P. Weng and A. C. Parker. CSG: Control Path Synthesis in the ADAM System. Technical Report CEng 92–03, Department of Electrical Engineering - Systems, University of Southern California, April 1992.

Appendix A

The VHDL Subset of the ADAM System

The VHDL used in the ADAM system and in the early USC system is a subset of the IEEE Standard VHDL[Ins88] since we are only concerned with representing behavioral specifications in VHDL. This subset was carefully defined to avoid features incompatible with the notion of behavioral description or unable to be represented in DDS, while still giving sufficient expressive power for most applications.

The allowed VHDL constructs are limited to the following:

1. *Design Entities*

   The primary hardware abstraction in VHDL is the *design entity.* A design entity is defined by an *entity declaration* together with a corresponding *architecture body.*

   - *Entity Declarations*

     The entity declaration basically defines the inputs and outputs of the design entity. A given entity declaration is restricted to be used by only one design entity; that is, it cannot be shared in this VHDL subset. The restrictions described in Item 5 (*Declarations*) are applied accordingly to the *entity header* and the *entity declarative part* of a given entity declaration[1]. The *entity statement part* must be empty in each design entity; in other words, the behavior of a design entity must only be specified in the corresponding architecture body.

- *Architecture Bodies*

  There are three general styles of descriptions possible within an architecture body: *structural*, *dataflow* and *behavioral*. However, only the behavioral one is supported in ADAM/USC. Behavioral descriptions specify data transforms in terms of algorithms for computing output responses to input changes. The feature of multiple asynchronous processes is not yet supported in the current version of VHDL2DDS; therefore, each architecture body is required to have one and only one concurrent statement in the *architecture statement part*.

2. *Subprograms*

   Since the *configuration* is not included in the VHDL subset, *subprograms* serve as the major mechanism for building the desired design hierarchy[2]. The definition of a subprogram can be given in two parts: a *subprogram declaration* and a *subprogram body*. Subprograms without subprogram bodies are usually used as the leaf nodes in the design hierarchy and the interfaces to the modules in the system library. Both *procedures* and *functions* are allowed. *Subprogram overloading* is not supported, and the *operator overloading* is limited to once for each scope of declarations.

3. *Packages*

[1]In fact, this rule is applied to all declarative parts in this VHDL subset. It will not be stated explicitly in the rest of the VHDL subset definition unless additional restrictions are required.

[2]In fact, this limitation makes the design entity unsuitable for describing *internal* blocks because there is no way to bind a collection of design entities into a design hierarchy without using a *configuration declaration*.

Packages provide a means of defining declarations which can be shared by different design units. One of the major usages of packages in ADAM is to define the interfaces of some implementation-dependent module libraries. In such a case, the *package declaration* has no corresponding *package body*. No special restrictions except those in Item 5 are imposed on packages.

4. *Types*

   In VHDL, a type is characterized by a set of values and a set of operations. All implicitly declared operations for a given type declaration are supported and will be translated automatically by VHDL2DDS. However, they are not recommended to be used in the VHDL descriptions because there may be no corresponding modules in the module libraries. As a result, the explicitly declared subprograms for a type are more appropriate in terms of module bindings. Two classes of types are allowed with restrictions; namely, *scalar* types and *composite* types.

   - *Scalar Types*

     Scalar types are limited to the predefined types BIT, BOOLEAN, and INTEGER only. Currently, users can not define their own scalar types. The INTEGER type is assumed to be a 32-bit implementation.

   - *Composite Types*

     The composite type is the only user-definable type class in this VHDL subset. It is further limited to *array* types only. An array object is a composite object consisting of elements that have the same type. Its primary usages are to model different bit-width values and memories. The maximal dimensionality of an array type is limited to 2. Both *unconstrained array* types and *constrained array* types are allowed. The *index definition* of an unconstrained array type must be INTEGER, and the *index constraint* of a constrained array type must be *ranges*. BIT_VECTOR is the predefined array type supported by VHDL2DDS. Since *subtypes* are not supported, there are several limitations on the uses of array types. First, a constrained array type is not defined as an unconstrained array type plus a subtype of this unconstrained array type. It itself is a data type. Also, defining a constrained array type from an existed unconstrained array is not allowed. This makes the unconstrained array types of little use.

For each array type, two additional operations are implicitly defined by this VHDL subset. They are *array read* operations and *array write* operations. If an *indexed name* appear at the right (left) hand side of an assignment statement, an array read (write) operation will be used. This feature is well suited for modeling memories; however, it cannot model the extraction of a subvalue from a multi-bit value.

5. *Declarations*

In addition to design entities, subprograms, packages and types, the other kinds of declarations allowed are *object* declarations and *interface* declarations.

- *Object Declarations*

  All three classes of objects are allowed; namely, *constants*, *signals*, and *variables*. An object declaration declares an object of a specified type. The feature of *deferred constants* is not supported. Signals will be treated as variables; that is, only the syntactical aspect of signals is preserved, but their semantics will be identical to variables in terms of the VHDL to DDS translation. Therefore, a signal declaration is not allowed to have a *resolution function*, *guards*, or the *signal kind*.

- *Interface Declarations*

  Interfaces objects also include constants, signals, and variables. The restrictions described above are applied accordingly. In addition, the *mode* of an interface object is limited to either **in** or **out**.

6. *Names*

   All forms of names except *attribute* names and *slice* names are allowed. The identifier for an entity, a package, a subprogram, or an interface object has only the first 5 characters significant after translation. An index name is considered to be an array read (write) operation instead of simply denoting an element of an array.

7. *Expressions*

   An expression is a formula that defines the computation of a value. It consists of a set of operators and their operands. Though all VHDL predefined operators are supported by VHDL2DDS, VHDL2DDS does not assume any specific implementation to a predefined operator, nor is it aware of the availability of any library module for binding. Care must be taken not to use any predefined operator unless the user can make sure there exists some corresponding module in the library or the operator in question will somehow be implemented. In ADAM, a more appropriate approach is to define a package for each available module library using function declarations or overloaded operators and use these functions or operators in expressions instead of predefined ones.

   The allowed operands in an expression include names, literals, and function calls. In addition, an expression enclosed in parentheses may be an operand in an expression. A literal is either a integer literal, a Boolean literal, a bit literal, or a bit string literal.

8. *Sequential Statements*

Sequential statements shall be the major means for describing the behavior of the component under design. The allowed sequential statements are

- *Signal assignment* statement.
- *Variable assignment* statement.
- *Procedure call* statement.
- *If* statement.
- *Case* statement.
- *Loop* statement.
- *Next* statement.
- *Exit* statement.
- *Return* statement.
- *Null* statement.

Statement labels can be used whenever necessary. A signal assignment statement is considered like a variable assignment statement. Hence, *transport* delay is not supported and the waveform at the right hand side can only consist of one element. In addition, a waveform element is not allowed to have an **after** clause. The *iteration scheme* of a **for** loop must be a *range* of type INTEGER.

9. *Concurrent Statements*

The *process* statement is the only form of concurrent statement allowed in this VHDL subset[3]. A process statement defines an independent sequential

[3]Since the feature of multiple concurrent processes is not supported, the inclusion of the process statement in this VHDL subset is merely for syntactical reasons.

process representing the behavior of the design. The execution of a *process* statement is modeled by the endlessly repetitive execution (an implicit loop) of its sequence of statements. Hence, a process statement is not allowed to have a *sensitivity list*.