VLSI DESIGN METHODOLOGIES FOR DIGITAL SIGNAL PROCESSING ARCHITECTURES

edited by
Magdy A. Bayoumi

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

VLSI DESIGN METHODOLOGIES FOR DIGITAL SIGNAL PROCESSING ARCHITECTURES

THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE

VLSI, COMPUTER ARCHITECTURE AND DIGITAL SIGNAL PROCESSING

Consulting Editor

Jonathan Allen

Other books in the series:

CONNECTIONIST SPEECH RECOGNITION: A Hybrid Appoach, H. Bourlard, N. Morgan
ISBN: 0-7923-9396-1
BiCMOS TECHNOLOGY AND APPLICATIONS, SECOND EDITION, A.R. Alvarez
ISBN: 0-7923-9384-8
TECHNOLOGY CAD-COMPUTER SIMULATION OF IC PROCESSES AND DEVICES, R. Dutton, Z. Yu
ISBN: 0-7923-9379
VHDL '92, THE NEW FEATURES OF THE VHDL HARDWARE DESCRIPTION LANGUAGE, J. Bergé, A. Fonkoua, S. Maginot, J. Rouillard
ISBN: 0-7923-9356-2
APPLICATION DRIVEN SYNTHESIS, F. Catthoor, L. Svenson
ISBN:0-7923-9355-4
ALGORITHMS FOR SYNTHESIS AND TESTING OFASYNCHRONOUS CIRCUITS, L. Lavagno, A. Sangiovanni-Vincentelli
ISBN: 0-7923-9364-3
HOT-CARRIER RELIABILITY OF MOS VLSI CIRCUITS, Y. Leblebici, S. Kang
ISBN: 0-7923-9352-X
MOTION ANALYSIS AND IMAGE SEQUENCE PROCESSING, M. I. Sezan, R. Lagendijk
ISBN: 0-7923-9329-5
HIGH-LEVEL SYNTHESIS FOR REAL-TIME DIGITAL SIGNAL PROCESSING: The Cathedral-II Silicon Compiler, J. Vanhoof, K. van Rompaey, I. Bolsens, G. Gossens, H. DeMan
ISBN: 0-7923-9313-9
SIGMA DELTA MODULATORS: Nonlinear Decoding Algorithms and Stability Analysis, S. Hein, A. Zakhor
ISBN: 0-7923-9309-0
LOGIC SYNTHESIS AND OPTIMIZATION, T. Sasao
ISBN: 0-7923-9308-2
ACOUSTICAL AND ENVIRONMENTAL ROBUSTNESS IN AUTOMATIC SPEECH RECOGNITION, A. Acero
ISBN: 0-7923-9284-1
DESIGN AUTOMATION FOR TIMING-DRIVEN LAYOUT SYNTHESIS, S. S. Sapatnekar, S. Kang
ISBN: 0-7923-9281-7
DIGITAL BiCMOS INTEGRATED CIRCUIT DESIGN, S. H. K. Embadi, A. Bellaouar, M. I. Elmasry
ISBN: 0-7923-9276-0
WAVELET THEORY AND ITS APPLICATIONS, R. K. Young
ISBN: 0-7923-9271-X
VHDL FOR SIMULATION, SYNTHESIS AND FORMAL PROOFS OF HARDWARE, J. Mermet
ISBN: 0-7923-9253-1
ELECTRONIC CAD FRAMEWORKS, T. J. Barnes, D. Harrison, A. R. Newton, R. L. Spickelmier
ISBN: 0-7923-9252-3
ANATOMY OF A SILICON COMPILER, R. W. Brodersen
ISBN: 0-7923-9249-3

VLSI DESIGN METHODOLOGIES FOR DIGITAL SIGNAL PROCESSING ARCHITECTURES

edited by

Magdy A. Bayoumi
The University of Southwestern Louisiana

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

VLSI design methodologies for digital signal processing architectures
/edited by Magdy A. Bayoumi
p. cm. -- (The Kluwer international series in engineering and computer science ; 257)
Includes bibliographical references and index.
ISBN 978-1-4613-6192-3 ISBN 978-1-4615-2762-6 (eBook)
DOI 10.1007/978-1-4615-2762-6
1. Application specific integrated circuits--Design and construction--Data processing. 2. Computer architecture.
3. Silicon compilers. 4. Signal processing--Digital techniques.
I. Bayoumi, Magdy A. II. Series: Kluwer international series in engineering and computer science ; SECS 257
TK7874.6.V57 1994
621.382'2--dc20 93-39842
CIP

Originally published by Kluwer Academic Publishers in 1994
Softcover reprint of the hardcover 1st edition 1994

Printed on acid-free paper.

To Acadiana

With smiling eyes,
you opened your arms for me
I hugged you, I kissed you, and
we are dancing since.

TABLE OF CONTENTS

FOREWORD

Designing VLSI systems represents a challenging task. It is a transformation among different specifications corresponding to different levels of design: abstraction, behavioral, structural and physical. The *behavioral level* describes the functionality of the design. It consists of two components; static and dynamic. The static component describes operations, whereas the dynamic component describes sequencing and timing. The *structural level* contains information about components, control and connectivity. The *physical level* describes the constraints that should be imposed on the floor plan, the placement of components, and the geometry of the design. Constraints of area, speed and power are also applied at this level. To implement such multilevel transformation, a design methodology should be devised, taking into consideration the constraints, limitations and properties of each level. The mapping process between any of these domains is non-isomorphic. A single behavioral component may be transformed into more than one structural component.

Design methodologies are the most recent evolution in the design automation era, which started off with the introduction and subsequent usage of module generation especially for regular structures such as PLA's and memories. A design methodology should offer an integrated design system rather than a set of separate unrelated routines and tools. A general outline of a desired integrated design system is as follows:

* Decide on a certain unified framework for all design levels.
* Derive a design method based on this framework.
* Create a design environment to implement this design method.

An overview of an integrated design methodology is presented in Chapter 1; the rest of the book discusses the following emerging issues:

System Integration: Integrating all the CAD tools at different levels is economically significant. Decisions made during early stages of high level synthesis have great effect on the final VLSI design implementation. A uniform data structure is needed for such integration among various levels (Chapter 1). Global optimization can be achieved with

such integrated systems (Chapter 8). A synthesis manager can be used for coordination and integration (Chapter 4). Several integrated CAD systems have been presented in this book; Sphinx (Chapter 1), HYPER (Chapter 4), MARS (Chapter 5), Hi_Pass (Chapter 6).

Optimization: Most of the subtasks of high-level synthesis are NP-hard. Optimization plays an important role in implementing such systems. Several approaches can be followed such as Heuristic-based algorithms, Graph-theoretical algorithms, and Integer programming. The latter proved to be the most efficient when solving several sub-tasks, simultaneously (Chapter 2). Identifying the intensive optimization tasks is a key parameter to reduce the computational complexity of a design system.

Algorithm Transformation: Algorithm transformation is the first phase in architecture synthesis. Optimizing these types of transformations– retiming pipelining, basic block transformation, control structure transformation, and suboperational level transformation, will shift the emphasis from the traditional level synthesis tasks (such as scheduling, assignment, allocation and module selection) (Chapters 4, 5 & 6).

Impact of Applications: Application requirements play a significant role in deciding the target architecture styles and modules. The main factors are the sampling speed, and the level of required numerical and arithmetic computations (Chapters 4, 6–10). The tasks performed by various CAD tools are highly dependent on the target architectures, their hardware components, and interconnections (Chapter 3). Parallelism is a natural architectural style for high speed sampling rates. It can be realized on the system level by using multiple buses and functional units (Chapter 3), or on the arithmetic level by using multiple operators concurrently (Chapter 6), or by employing non-traditional arithmetic systems (Chapter 9).

Memory Management: DSP and image processing systems involve a lot of data storage and intermediate data manipulation. Such data processing in real-time presents not only computational problems, but also storage problems. Several solutions are discussed in different chapters

such as: Designing high level memory manager (Chapter 7), Automated assignment of variables to multiple memories (Chapter 3), and Interfacing the developed designed system to an existing data base structure and management; e.g: OCT tool (Chapter 6) and CADENCE Edge tool (Chapter 1).

Algorithm Prototyping: As DSP applications get more complicated and involve various components viz., smart sensors, prototyping became an economic necessity. Prototyping is also an efficient vehicle for software development. Two approaches can be followed: first, using a general machine with specific capability for software development (Chapter 10); and, second, using hardware modules for functional representation of an algorithm, i.e. hardware emulation (Chapter 8).

PREFACE

In recent years, Digital Signal Processing (DSP) architectures have gained considerable significance, because of their wide application domains, which range from medium throughput speech, audio and telecommunication systems at the lower end of the spectrum, to image, video and radar processing at the high frequency end. Moreover, state-of-the-art systems for real-time mass communication, such as robotics, machine vision, and satellite systems, represent fast growing application areas. For such complex systems, especially for those designed for consumer electronics market, features such as throughput, area, power consumption and packaging tend to be of utmost importance. Design cycle time, from algorithms to system, has to be reduced, from a few years to weeks, in order to respond to the changing market. These objectives can be achieved by implementing DSP systems in an Application Specific Integrated Circuits (ASICs) paradigm using a comprehensive design methodology.

A design methodology should offer an integrated design system rather than a set of separate unrelated routines and tools. It is focussed around high level synthesis that transform a high level specification (behavioral or functional) into an intermediate implementation. Synthesis technologies have become very popular; due to mainly these reasons, the need to get a correctly working system the first time, the ability to experiment with several alternatives of the design, and the economic factors (such as time to market etc.). In addition, synthesis systems allow designers with limited knowledge, of low level implementation details, to analyze and trade-off between alternative implementations without actually implementing the target architectures.

Design methodologies and environments for DSP architectures and applications are the focus of this book. The emphasis is centered around the emerging issues in this area, which are: system integration, optimization, algorithm transformation, impact of applications, memory management and algorithm prototyping.

The intent of this book is to be informative and to stimulate the reader to get a head start, gain knowledge and participate in the fast evolving field of Application Specific Design Methodology for DSP Architectures. The book can be used as a textbook for research courses in Application Specific Design, VLSI Design Methods, and Silicon Compilers. It can also be used as supplementary text for graduate and senior undergraduate courses in DSP architectures, design, and applications. It can also serve as a material for tutorials and short courses in these topics.

The idea of this book was motivated by a pre-symposium workshop at ISCAS'92, San Diego. The speakers in this workshop were Catherine H. Gebotys, Baher Haroun, Rajeev Jain and myself. The workshop was sponsored by the VLSI Systems and Applications (VSA) Technical committee of the IEEE Circuits and Systems society. I extend my thanks to the speakers at the workshop for supporting the idea of this book, starting from its conception in San Diego. Special thanks to the authors who patiently spent considerable time and effort to have their research work reported in this book. The environment at the Center for Advanced Computer Studies (CACS) has been dynamic, inspiring and supportive for such a project. My sincere thanks and appreciation to my close friends N.A. Ramakrishna and Aakash Tyagi. They have been my students and colleagues for several years; working with them has been a very enjoyable experience. Having Cathy Pomier in CACS is a source of smiles in tense times. She is the person to go to when deadlines are overwhelming. My sincere thanks to Kluwer Academic Publishers for their continuing support, to Bob Holland, the editor and his assistant Rose Luongo for their patience. They have provided me with a friendly communication channel.

Finally, I would like to acknowledge my lovely wife's support and patience. She is still looking forward to my future money-maker book. My interesting kids, Aiman, Walid and Amanda, are always asking me when I will write a #1 Best Seller, so their friends can see it in Wal-Mart and other bookstores in the Acadiana mall.

Magdy A. Bayoumi.

1

Sphinx: A High Level Synthesis System for ASIC Design

Ramakrishna N.A and Magdy A. Bayoumi

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504

Introduction

During the past twenty years, integrated circuit foundries have gained the ability to produce increasingly complex integrated circuits. This has encouraged system designers to design increasingly complex electronic systems - from the board level down to the chip level. Such rapid progress can be partially attributed to two phenomenon - first is the fabrication technology and second, is the development of design methodologies and tools for automated design and design support. Just a decade ago, in the early eighties, designers began to use automatic/semi automatic tools for layout/mask generation, schematic capture, and circuit and logic simulation. A limited amount of high level simulation was also being done, for example, several simulators were built and used for experimenting with high level descriptions in ISPS [1], and other academic and industrial proprietary languages. The tools as such were used primarily as support systems for certain tasks. When the tools failed, the designs could be done manually. However, the rapid changes in technology, the ability to put more devices on a chip, new applications, cost and performance targets have forced the use of comprehensive design systems. Using such systems, the design effort in tasks such as schematic capture, placement, routing, layout generation, simulation, etc., has reduced by at least 50%. The design automation era started off with the introduction and subsequent usage of module generation environments. Hardware

generators for regular structures such as PLAs, ROMs etc, became very popular. Soon after, logic synthesis systems were introduced. In logic synthesis, the automation was the direct translation of truth tables into optimized logic networks. In the recent past, the evolution of design automation technologies has brought about the introduction of synthesis systems - essentially methods and methodologies that transform a high level specification (behavioral or functional) into a reasonable implementation. There are several reasons why synthesis technology has become popular. Amongst them are, the need to get a correctly working system the first time, the ability to experiment with several alternatives of the design without actually implementing the design, and economic factors such as time to market. In addition, synthesis systems allow designers with limited knowledge of low level implementation details to analyze and tradeoff between alternative implementations without actually implementing the target architectures. Ideally, synthesis techniques promise designs that are correct by construction. Though such claims are hard to prove using formal methods, it is widely accepted that synthesis systems arrive at designs that are at least satisficing. Currently push-button technology for automatic design of hardware from high level specifications is far from reality. Designers have to take part in several design decisions during the design process specially at the higher levels, while delegating low level (and well understood) tasks such as layout generation, place and route etc, to robust design tools. In the last decade, there has been significant interest amongst CAD tool developers to design and implement multi-level tools and methodologies that aid in the design of integrated circuits.

The ideal goal of design automation is to be able to derive mask level information from a given behavioral specification. Silicon compilers are software systems that allow us to achieve such goals. The various tasks in a silicon compiler environment can be roughly classified into the following categories.

System level synthesis

High level synthesis

RTL synthesis

Logical level synthesis

Technology mapping

At the highest level of abstraction, systems are specified in terms of their instruction set, constraints that need to be met such as fabrication cost, power consumption etc. System level synthesis deals with transforming such specifications into one or more subsystem descriptions at the algorithmic level. The tasks include partitioning, and sub-system specification. System partitioning in the domain of high level synthesis has now become an important research area.

High level synthesis aims at deriving high quality and correct physical structure from behavioral descriptions. The fast turnaround times achiveable using synthesis tools encourage designers to explore different architectures before deciding upon the final one. The ability to explore for solutions within the design space sometimes results in arriving at architectures that are of an order of magnitude better than those achieved using traditional methods of design. In high level synthesis, the starting point is the behavioral description at the algorithmic level. This description is a precise procedure for the computational solution of a problem. The behavior is specified in terms of the operations and the computation sequences of the inputs to produce the required outputs. A high level programming language such as ISPS [1], Verilog [2], and VHDL [3], Silage [4] can be used to describe the behavior of the architecture to be synthesized. The various tasks associated with the high level synthesis include scheduling of operators to control steps, allocation of operators to operations in the description, assignment of operators to operations, allocation and assignment of memory modules to variables. There are several ways to schedule operators and allocate resources and this causes several tradeoffs in terms of time and area. By providing for more resources, a more parallel implementation can be achieved, however the price to pay is the additional hardware cost. Scheduling, resource allocation and binding, are tasks involved in finding a satisfying solution within the design space.

Register level synthesis deals with the issues involved in the synthesis of the data paths and controllers. The output of the scheduling and allocation stages is a description at the register transfer level and the next step is the register level synthesis. At this level, physical characteristics of the various operators are taken into consideration. Modifications on the initial allocation can also be made based on performance issues. Optimization of memory, buses and interconnect are some of the sub tasks involved in data path synthesis. Controller synthesis involves a transition from behavior to structure. The objective of this task is to synthesize a controller given the schedule and the data path by the scheduler and the data path synthesizer. Both hardwired and microcoded controllers can be synthesized.

The data path synthesis and the controller synthesis phases specify the various building blocks of the design under consideration. It is the task of the logic synthesis system to map the behavior of these building blocks to gate level hardware structures. Technology mapping on the other hand deals with the actual mapping of these gate level structures to standard library cells in a given technology.

Scheduling is an important sub task in the synthesis process. Briefly, scheduling deals with the assigning of operations into control steps. Scheduling affects several aspects of the synthesized design such as the total number of functional units used, the total time of computation, the storage requirements, interconnect requirements etc. In other words, the area of the design and the speed of the design are directly affected by decisions made during the scheduling process. In manual designs, it is quite possible that certain decisions are made which have a rather negative affect on the area and time characteristics of the resulting design. In addition, the inclusion of issues such as pipelining, conditional branching etc, complicate the matters further.The data path synthesis stage follows the scheduling stage. The tasks considered here are functional unit, storage and interconnect allocation and binding. These are complicated tasks and can literally lead to a combinatorial explosion of possible solutions.

The above mentioned areas are some of the issues that can be effectively handled by synthesis programs as opposed to manual design. The effectiveness of placement and routing systems has encouraged designers to develop techniques for automating the higher levels in the design abstraction.

Ever since the power of synthesis techniques has been shown to be effective, there have been dozens of algorithms that have been published. While most of these algorithms cater to individual tasks of the synthesis process, not many integrated methodologies have been presented. There are several challenges that need to be faced when trying to get a working methodology — specially one wherein the constituent methods themselves are complex systems.

DSP and Synthesis

Digital Signal Processing (DSP) systems are an important class of applications in electronic design. Application domains range from medium throughput speech, audio and telecommunication systems at the lower end of the frequency spectrum to image, video and radar processing at the higher frequency end. For such complex systems and specially for those designed for the consumer electronics market, features such as throughput, area, power consumption and packaging tend to be of utmost importance. Design cycle time from algorithm to system has to be reduced from a few years to weeks in order to respond to the changing market. These objectives can be achieved by implementing DSP systems in application specific integrated circuits using computer aided design (CAD) support. Such CAD support should include tools to assist the designer at all levels of the design, from the highest level of abstraction to the physical design level. DSP applications are characterized by intensive arithmetic operations in contrast to control dominated applications. In DSP applications, many arithmetic operations including multiplication, have to be executed in parallel in order to meet the throughput constraints. DSP system implementations can be roughly classified into two classes, compiled silicon and software programmable [5]. In the case of compiled

silicon systems, the hardware is designed to execute the DSP algorithm, while in the case of software programmable, either general purpose DSP systems are used or the given algorithm is compiled to generate code which can run on general purpose DSP systems [6]. In our research, we are concerned with compiled silicon structures. In the next section, we will present an overview of a methodology for the synthesis of application specific integrated circuits.

System Overview

The concept of high level synthesis and the numerous sub tasks associated with it were introduced and described in the previous section. In this section, we will propose a methodology for the synthesis of application specific integrated circuits. The methodology will be integrated into a system we call *Sphinx*. Sphinx provides an integrated set of interacting tools for synthesis of synchronous ASICs. They include behavioral, structural and logic synthesis tools in addition to placement, routing, and low level integration tools. These tools are all interfaced with the Cadence Edge Framework.

Figure 1 shows an overview of the system. Some of the important features of the system include:

- A synthesis oriented description language: Verilog is used as the language to specify the behavior of the algorithm that needs to be synthesized. This specification in then translated into an internal graph representation [7].
- A scheduling subsystem that supports both pipelined and non-pipelined scheduling. Partially bound hardware descriptions are allowed at the flow graph level. This enables the user to pre-bind certain hardware units to nodes in the flow graph. The rest of the nodes can be bound by automatic procedures that are built into the system. In this chapter, we will describe one of the pipelined scheduling algorithms that have been used. In addition to this algorithm, the system also supports other algorithms for pipelined and non-pipelined scheduling.

- A data path synthesis subsystem to generate the data path given a schedule and a module database. Issues such as memory allocation, functional unit allocation and binding, interconnect optimization, register minimization, etc., are considered here.
- A control synthesis subsystem to synthesize the controller for the synthesized data path. Microprogrammed control units are utilized here. The system takes in the schedule information from the scheduler, the data path generated by the data path synthesizer, information regarding the various modules to be used from the module library, and generates the microinstructions and also specifies the architecture of the control unit.
- A netlister that takes in input from the data path synthesis subsystem and the controller subsystem. Using the information such as word size, the netlist describing the connectivity at the signal level and functional unit level is derived. At this level, the netlist will contain all the necessary information that the module maker needs in order to generate the required hardware.
- The Module Generation Environment consists of a set of parameterized module generators developed in SKILL running on Cadence Edge Framework. Using these generators, one can develop modules such as adders, subtracters, multipliers, registers, etc.
- The Cadence Place and Route package is used. Sphinx is integrated with the rest of the Cadence environment.
- A Graphical interface in SKILL has been used because of the ease of programming and the graphics libraries that it supports in conjunction with the Cadence database.
- Sphinx has been designed as an open system — in which tools communicate through machine readable and human readable formats. Additional tools can be easily integrated with the system as long as interface formats are maintained.

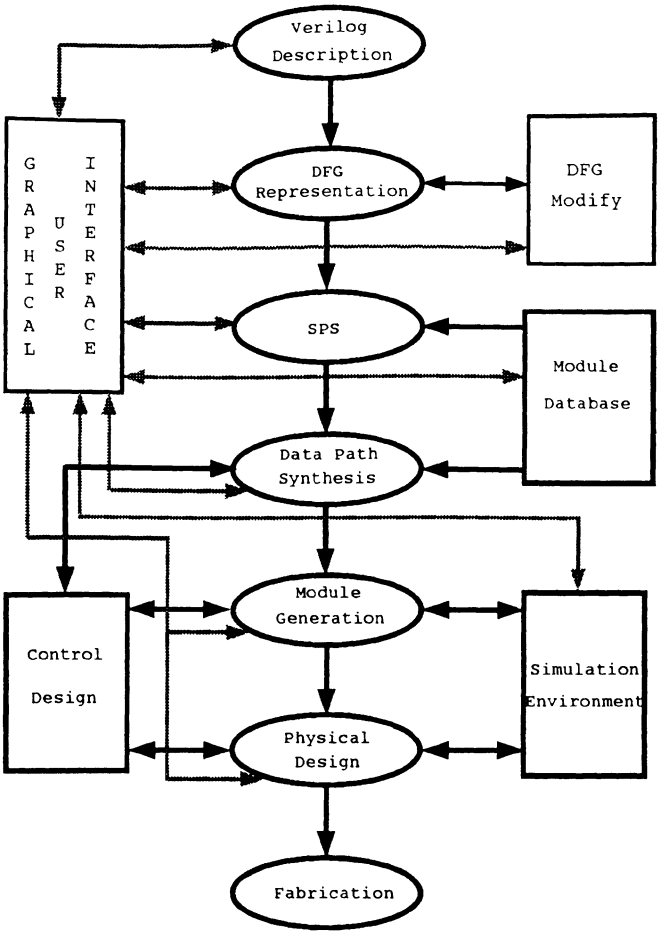


Fig. 1 Sphinx System Overview

Algorithm Representation

The input to the Sphinx system is a description of the algorithm in Verilog, a hardware description language. In addition to being a comprehensive and an easy to learn language, the simulation environment provided with the Cadence Framework makes Verilog the language of

choice for behavioral description. The input algorithm described in Verilog is translated to a data flow graph representation. Traditional compiler techniques using Lex and Yacc tools have been used to develop the translator. The data flow graph nodes represent the operations in the input algorithm and the edges represent the precedence between operations specified in the algorithm. In lieu of a behavioral description, the input algorithm can also be input to the system graphically. An X-windows based data flow graph editor has been designed for this purpose. Using this editor, a user can enter and modify a data flow graph. This editor stores this graphical information in a form suitable for use by other subsystems such as the scheduling subsystem and the controller synthesis unit.

Pipelining and DSP

Due to the nature of DSP applications, computational throughput is a highly desirable feature in DSP architectures. DSP applications require throughputs that are in the range of 10^2 to 10^6 MOPS (Millions of operations per second). Such throughputs are hard to achieve in a non-pipelined uni-processor. It is therefore important for us to exploit ways by which we can increase the throughput to suit the application domain. Two techniques that can be used to achieve concurrency of operations are pipelining and parallelism. Pipelining can improve performance by an order of magnitude with very little additional hardware. Manual design of large pipelined systems is a rather difficult, tedious, and error prone process.

In pipelining [8], each computation task is partitioned into a sequence of sub tasks and each one is executed during one clock cycle. Consecutive tasks are initiated at some regular intervals (latency) which are integral multiples of the clock cycle. Therefore, subtasks of consecutive tasks are executed in parallel and they are overlapped in time on different parts of the architecture.

The throughput of a pipelined system is the number of tasks completely processed by the pipeline per unit time. Therefore for an n-stage

pipeline, the throughput is k/(k + n -1) where, (k + n -1) is the time required to process k tasks.

The speedup of a n-stage pipelined system, processing k tasks, relative to a non-pipelined system is (nk)/(n + k -1). When k >> n, the speedup approaches n, the number of stages in the pipeline. In order to achieve maximum throughput, the number of stages should be kept minimum. In our algorithms, the number of stages is equal to the length of the critical path of the given data flow graph.

Many computer aided design systems have been proposed and implemented for the synthesis of pipelined and non-pipelined architectures. Sehwa [9] is one of the earliest systems proposed for the synthesis of pipelined architectures. In Sehwa, two approaches for scheduling have been presented - performance based scheduling and cost based scheduling. In [10], a force directed approach was used to schedule data paths with an aim to maximize the hardware sharing for a given latency. List scheduling has been used in several systems such as [30]. Our basic approach to scheduling for pipelined systems is along two lines - performance based and cost based scheduling.

Scheduling for Pipelined Architectures

There are two main issues that arise in the scheduling of pipelined systems. Scheduling can be done to achieve one of the following [10]:

to come up with the optimal number of hardware resources for a particular latency (performance constrained scheduling), or
to come up with an optimal latency for given hardware resources (cost constrained scheduling)

We will present algorithms for each of the above. The inputs to our scheduling algorithm are the data flow graph, information about the functional units such as, the operation times, cost (required to assign priorities to operations), latency in case of performance constrained scheduling and the number of operators available in the case of cost constrained scheduling. Several assumptions are made such as - operation

times of functional units are integral multiples of each other. For example, an adder takes 40ns and a multiplier takes 80ns. Time to latch result to output buffer is 20ns. This fixes our clock period to 60ns - the minimum time required for an adder to operate on a set of data and latch the result to the output buffer. Functional units whose operation times are integral multiples of the fastest operator, can be structurally pipelined or non-pipelined. If a functional unit takes x times the time required by the fastest functional unit, it can be assumed to be a x-stage pipeline with a buffer between stages.

Basic Scheduling Schemes

The ASAP (as soon as possible) scheduling scheme is a simple scheduling technique. ASAP scheduling can be performed under the assumption that the number of resources is known and also otherwise. The operations in the flowgraph that have to be scheduled are first sorted topologically — i.e., if operation O_j is constrained to follow O_i due to data/control precedence, then O_j will topologically follow O_i. The operations are taken one at a time in this order and each is scheduled into the earliest control step possible — given its dependence on other operations and limits on usage. ASAP has been used in several systems [11]. ASAP scheduling suffers from the disadvantage that critical path nodes may not be given priority causing non-critical path nodes to have greater priority than critical path nodes. ALAP scheduling on the other hand tends to delay the execution of each node to the latest possible time. Figure 2 shows the ASAP, and ALAP schedule for differential equation example.

ASAP and ALAP schedules are used to determine the time frames for each operation in the data flow graph. The time frame is constructed based on the fact that an operation can be scheduled into a control step between its ASAP and ALAP times. Figure 3 shows the time frame diagram for the ASAP and ALAP schedules shown in Fig. 2 [10].

The width of the box indicates the probability that the node will be eventually placed in a control step. For example, nodes on the critical

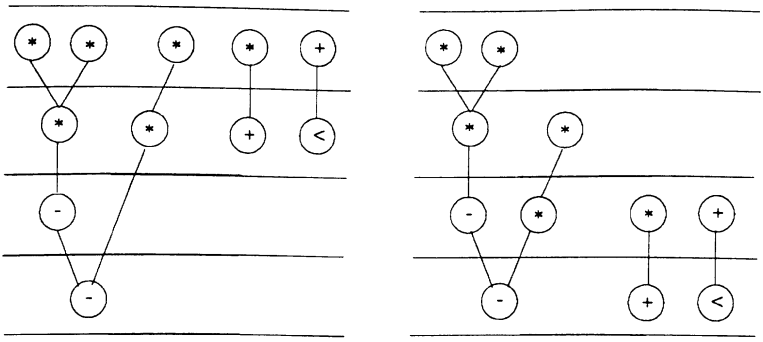


Fig. 2 ASAP and ALAP Schedule for DiffEq

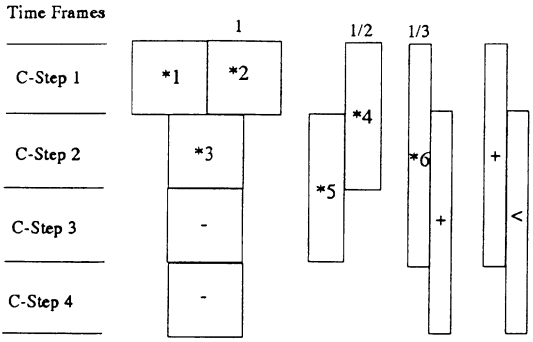


Fig. 3 Time Frame Diagram for the schedule in Fig. 2

path (i.e., same ASAP and ALAP times) will result in a square of area one. Once we have the time frame diagram, we can derive a distribution graph for each of the operations. The distribution graph shows for each control step, how heavily loaded that control step is — given that all schedules are equally likely. It is calculated by finding the earliest and latest control step in which the operation can be scheduled given the time constraints and precedence relations. If an operation can be scheduled in

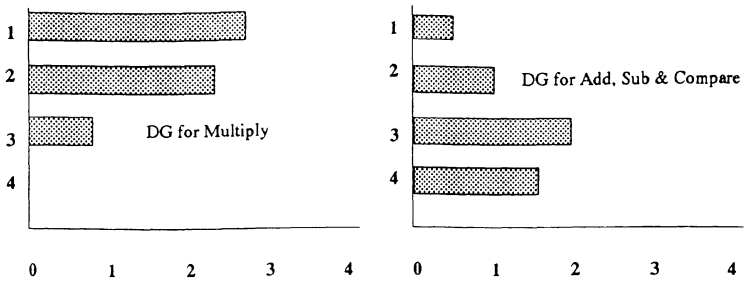


Fig. 4 Initial DGs for the graph shown in Fig. 2

any of C control steps, then 1/C is added to each of the control steps in the graph. For an operation i in the graph,

$$DG(i) = \sum_{i=1}^{N} P(i)$$

where $P(i)$ = probability that the operation is scheduled in the control step i and N is the number of control steps [10]. The DGs derived for the graph in Fig. 2 are shown in Fig. 4.

The first graph shows the distribution of the multiply operations. The second graph combines the distributions of the add, subtract and compare operations.

Algorithm for performance constrained scheduling

In performance constrained scheduling, the scheduler attempts to achieve a schedule with the optimum number of hardware resources for a given latency.' We will illustrate our algorithm using the FIR filter example, whose DFG is shown in Fig. 5.

1. For the given data flow graph (DFG) with N nodes, the ASAP and ALAP times are computed.

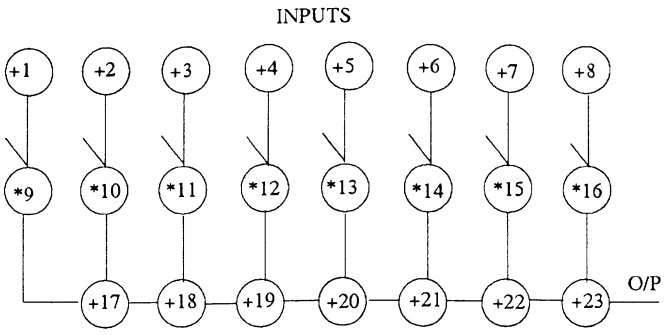


Fig. 5 Flow Graph of a FIR Filter

2. Hopping Distance Computation: The hopping distance of a node is the difference between its ALAP and its ASAP schedule time.

3. The probability factor (PF) is computed for each node.

   a. PF = 1/(hopping distance + 1)

   b. Critical nodes (ASAP = ALAP), will have PF = 1.

4. After the probability computations, the distribution graphs are generated for the different kinds of operators. Given that all possible schedules are likely, the distribution graph shows for each control step how heavily loaded that step is [10]. The distribution graphs are generated based on the fact that for a pipeline with a latency l, the operations scheduled in control steps $i = kl$, (k = (0,1,2, ..) run concurrently. Fig 6 shows the initial distribution graphs for each of the operators. In this example, a latency of 3 was used.

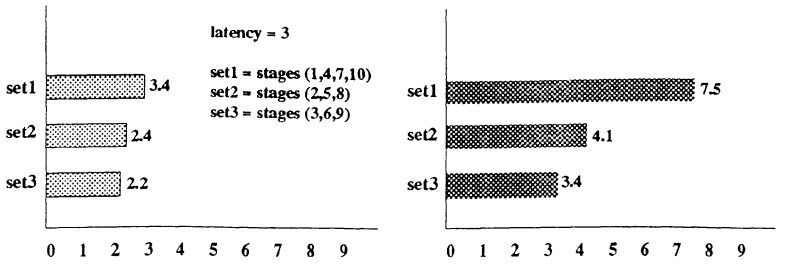


Fig. 6 Initial Distribution Graphs

5. Scheduling of critical path nodes is straightforward. These nodes are assigned to control steps based on either their ASAP or ALAP time which happen to be the same. The non-critical path nodes are scheduled based on a *load distribution balancing* approach. In this approach, the scheduler attempts to assign nodes to control steps so as to balance the load on the operators in all the control steps. The process is described as follows:

   a. Critical path nodes are scheduled according to their ASAP or ALAP times.
   b. All control steps in which all the operators have a PF = 1 are marked off since, no other operation can be scheduled in that control step without increasing the number of operators.
   c. For all other nodes, assignment to control step is based on the following:

      - Each of the nodes is to be assigned a control step in the closed range of its ASAP and ALAP times.
      - It should be assigned in such a way so as to maximize balancing of distribution
      - Priorities can be chosen based upon:

        - priority based on the cost of the functional units
        - priority based on the operation times of the functional units

- priority based on the number of operations of a specific type in the DFG

6. The balancing of the distribution graphs is done by assigning nodes to control steps which contribute the least to the DG. The nodes of a particular type are selected based on one of the following heuristics:

   a. Nodes are hopped in the increasing order of their PF
   b. Nodes are hopped in the decreasing order of their PF

7. After hopping and priority rules are decided upon, nodes are assigned fixed control steps. Assignment is based on the following:

   a. for all control steps that a node can be possibly assigned to (i.e., its freedom), identify the different sets (a set here is the collection of all the stages that should be active concurrently) and determine the set among the identified ones that contribute the least to the DG. Select that set because hopping to one of the stages of that set would result in the improvement of the DG.
   b. after set identification, one of the stages of the set is selected. When a node is assigned a particular control step, its immediate successor's ASAP time is affected and has to be modified.
   c. each of the stages of the set is tried for assignment. The stage which results in minimum modification of the ASAP and ALAP times of its successors and predecessors is chosen for assignment of the node.
   d. once a node is assigned a control step, its PF becomes equal to 1. and the DGs are updated.

8. The whole process is repeated till all nodes have been assigned to a control step.

Figure 7 shows the balanced DG for a latency of 3. As a result of the above procedure, the scheduler has determined that we need a maximum of five adders and three multipliers for the FIR filter.

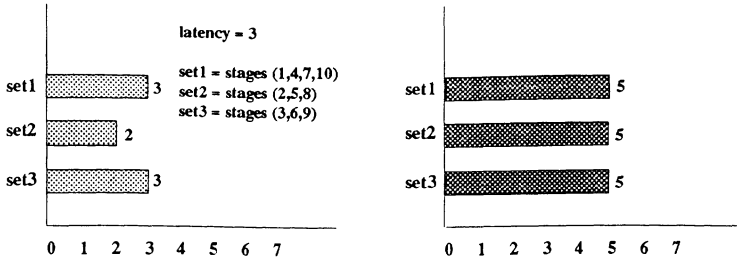


Fig 7 Final Distribution Graph

Figure 8 shows the schedule obtained by for the pipelined version with a latency of 3. The results obtained by our algorithm compare

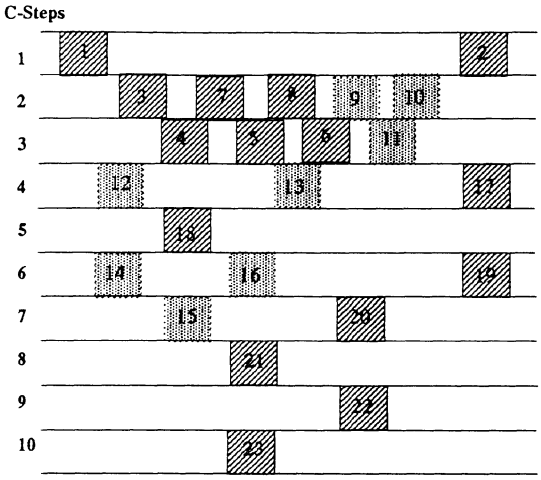


Fig 8 Final Pipeline Schedule

well with published reports such as [9] and [10]. For example, for the pipelined case, assuming that each control step has a duration of 60ns, we can obtain an output every 180ns (for a latency of 3). In the case of both [9] and [10], the output is obtained only after 300ns.

Optimal Latency Computation

In the previous section, an algorithm for performance constrained scheduling was presented. The aim there was to determine a schedule for a given latency using the minimum number of resources. While other systems compromise on the length of the critical path, our algorithm strives to maintain the length of the critical path to its minimum value. In several situations, the number of resources that we can use may be limited. In such cases, we have to determine a schedule which will be most effective for the given resources. The goal is to determine a schedule with the minimum latency because latency affects throughput. For a latency of one, the throughput is maximum, however, all stages of the pipeline are active in every clock cycle thereby curbing any resource sharing. When latency is set to be the length of the critical path, then maximal sharing of resources can take place, however, the throughput reduces drastically. In effect it behaves as a non-pipelined system. With the cost constrained approach which will be described briefly in this section, the minimum value of the latency is determined for the given resource set and critical path length.

The approach is as follows: Let the number of types of resources that are used in the DFG be m. Letr_m represent the number of operators of type$m$ and therefore $(r_1, r_2, \cdots, r_m)$ is the set representing the number of operators of each type in the DFG. Let $(a_1, a_2, \cdots, a_m)$ be the set representing the available number of resources of each type. The aim is to determine a minimum latency such that, given the resources that are available, we can get a feasible schedule. An important constraint on the value of the latency is that it should not exceed the length of the critical path. If l_l exceeds the length of the critical path, then there is no feasible solution using the given hardware resources without stretching the pipe. The upper bound of the latency is equal to the length of the critical path. Let this latency be l_u. If there exists a schedule with latencyl_u using the given hardware resources, thenl_u is the upper bound. If a schedule is not possible, then it means that no feasible schedule is possible while maintaining the number of stages equal to

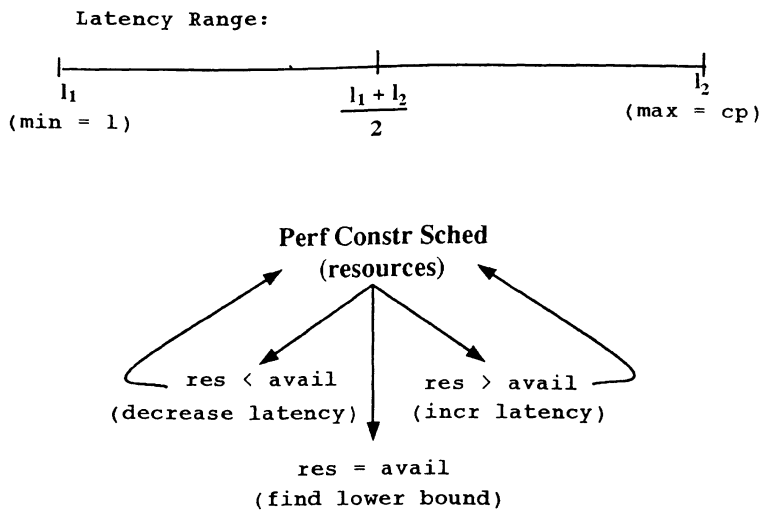


Fig. 9 Optimal Latency Computation

the length of the critical path. In such cases, we can determine the l_u using a divide and conquer approach. Let the range of latencies be $(l_1, l_1 + 1, \cdots, l_2 - 1, l_2)$. Let $l_{new} = (l_1 + l_2)/2$. At this point, we can try to do a performance constrained scheduling (as discussed earlier). As a result of the scheduling, if the number of resources required is less than the available number of resources, then$l_u = l_{new}$. The upper bound of the latency is therefore in the range $(l_1, (l_1 + l_2)/2)$. However, if the number of resources required is greater than the number available, it would mean that the upper bound of the latency would lie in the range $((l_1 + l_2)/2, l_2)$. By taking these new bounds, we can perform the above procedure till we get a schedule with a latency l_{opt}. At some point of time during the search process, we might arrive at a situation wherein we obtain a schedule with a latency that uses the exact number of resources that are available. In such cases, we store the solution as the current best solution, and try and determine if the latency can be further reduced. By decrementing the latency by one each time, a performance constrained scheduling is performed till we reach a point when the number of resources required will be greater than those available. At

that point, we stop the search and the latency at that point is the optimal latency l_{opt}. With a latency less than l_{opt}, we cannot arrive at a schedule with the given number of resources and with a latency greater than l_{opt}, we will always arrive at a schedule.

The procedure is shown in Fig.9 and a program outline is shown in Fig. 9a. Initially `l1 and l2` are equal to 1 and the length of the critical path respectively.

```
/*PCS(lnew) -> Performance Constrained Scheduling
 with latency lnew)
step 1: lnew = (l1 + l1)/2
        PCS(lnew)
     if req < avail /*try to decrease latency */
         l2 = lnew
         l1 = linit
         step 1
     else
     if req > avail /*need more resources, increase lat*/
         l2 = lfinal
         l1 = lnew
         step 1
     else
     if req = avail
         l1 = l1
         l2 = lnew
         (store solution)
         check_for_better_soln
     endif
```

Fig. 9a. Algorithm Outline for Cost Constrained Scheduling

Data Path Synthesis

Data paths consist of functional units, memories or storage elements, and interconnection units which provide for data transfer between functional units and memories. Given a scheduled set of operations, the functional unit and memory allocation problem consists of five sub problems [13]: specification of the data and control flow, mapping operations onto available functional units, assigning values to registers, and providing interconnections between operators and registers using buses and multiplexors. The goal is the minimization of an objective function such as:

* total interconnect length
* total functional units, register, mux, bus cost
* critical path delays

Data path allocation techniques are of two types — iterative/constructive and global. Global techniques find simultaneous solutions to a number of assignments at a time. Iterative/constructive schemes assign elements one at a time. These are more efficient but are less likely to produce optimal solutions.

In our prototype design, a global allocation technique based on the clique partitioning algorithm has been used. In the case of register allocation, the goal was to bind registers to variables in a way such that the lifetimes of those variables bound to the same register did not overlap. The primary objective was to obtain a minimum set of registers. A weighted cluster partitioning algorithm was used to reduce the number of multiplexors needed in the final data path. In this section, we will present an outline of one of the data path synthesis procedures used in the synthesis of pipelined systems. Figure 10a shows an outline of the process.

The data path generation procedure consists of three parts — grouping sharable operations, allocation of interconnect and netlist generation. The result of the data path synthesis is a netlist which provides a list of operators, including registers, multiplexors, and functional units and their interconnection patterns. The primary inputs, outputs and control points are also identified. Figure 10b shows the data path generated for the

```
procedure datapath()
   begin
      read_schedule_output();
      read_module_lib_info();
      find_fu_info();
      /*determine FU type, FU Id, CS*/
      group_stage_operators();
      group_sharable_operators();
      update_pred_list();
      /*nodes are either shared operators
      or unique operators*/
      mux_reg_allocation();
      netlist_generation();
   end
procedure mux_reg_allocation()
   begin
     for each sharable_fu(sfu) of same type
        begin
           if num_nodes(sfu) > 1 then
               create(mux,sfu);
               /*one mux for each input*/
               for each node(sfu)
                  begin
                     cs = timestep(node);
                     pred_list = predecessor_list(node)
                     for each node(pred_list)
                         insert_reg(node, pred);
                         mark_external_ports();
                         update_port_connectivity();
                     endfor
                  end
               endfor
           else
               begin
                     cs = timestep(node);
                     pred_list = predecessor_list(node)
                     for each node(pred_list)
                         insert_reg(node, pred);
                         mark_external_ports();
                         update_port_connectivity();
                     endfor
               end
           endif
        end
     endfor
   end
```

Fig.10a The Datapath Synthesis Process

differential equation example. This is a pipelined datapath for a latency of three.

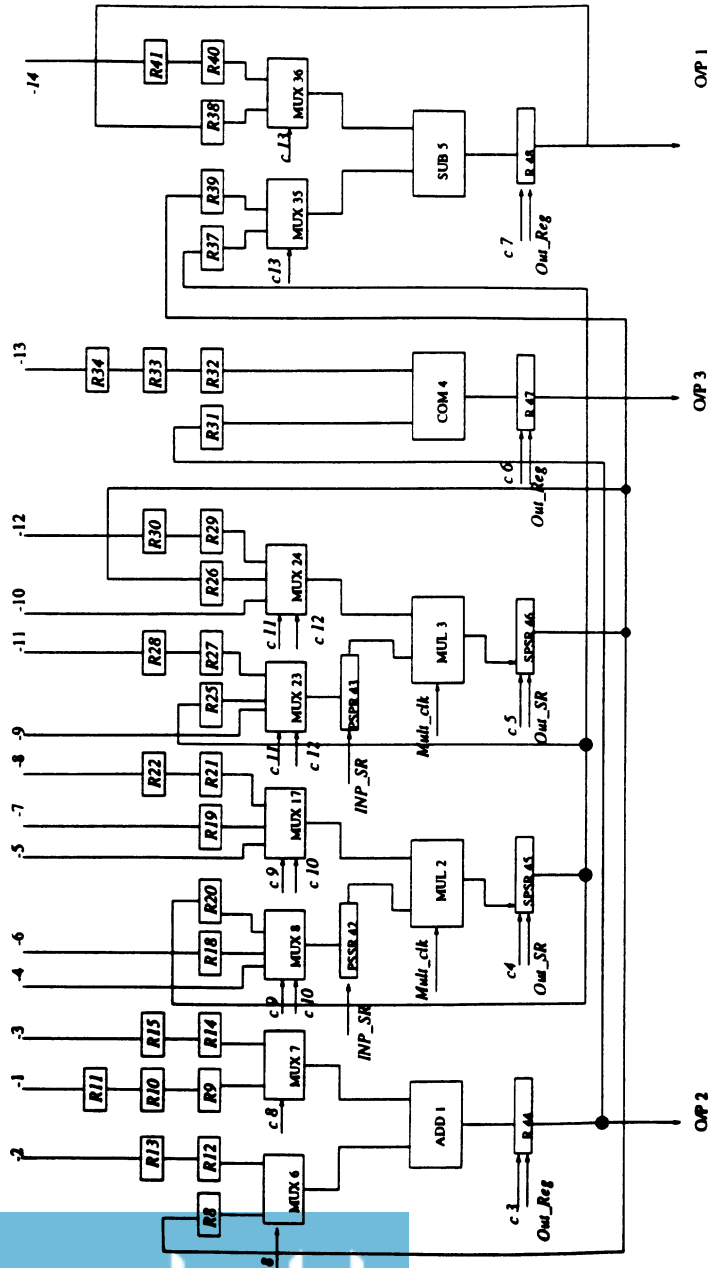


Fig. 10b Data Path Generated for the DiffEq Example

Control Path Synthesis

Control path synthesis is a very important phase in the overall synthesis process. The two basic approaches to control design are hardwired and micorprogrammed control. While hardwired control units might prove to be more area efficient and faster, the generic nature and programmability of microprogrammed control units make them a viable choice, specially in a rapid prototyping environment. Controlling pipelined datapaths is however a complex process using either of these schemes. In a pipelined system, each stage of the pipeline may be working on an independent set of data depending upon the latency. If the pipeline is capable of handling loops and conditionals, the control becomes even more complex.

Microprogrammed control of pipelined structures can be peformed along two lines — one where each microinstruction controls the entire pipeline system simultaneously and the other where data and microinstruction travel together in the pipeline. In [12], these two approaches have been referred to as *time stationary control* (Fig. 11a) and *data stationary control* (Fig 11b).

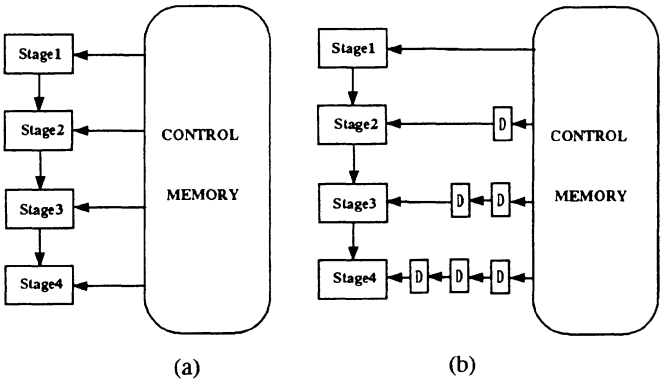


Fig 11. (a) Time Stationary Control
(b) Data Stationary Control

In the case of the time stationary approach, only one microinstruction is active at a given time and it defines the state of the entire machine. In the case of the data stationary control, the datum's entire travel path is defined at the beginning of each clock cycle [12]. Furthermore, there are are multiple microinstructions active at any given time. In our prototype design, we have chosen to use a time stationary approach to the control of the pipelined system. Loops and conditionals are not included in the prototype design.

The control synthesis system follows the scheduling and datapath synthesis. The datapath synthesis phase generates the netlist of the hardware which has to be scheduled according to the schedule generated by the scheduling subsystem. During datapath synthesis, appropriate control points are also identified in conjunction with the module library. The main tasks of the control synthesizer subsystem are: microinstruction format generation, microinstruction generation and specification of the control unit.

Fig 12 shows the format for each microinstruction. In the prototype,

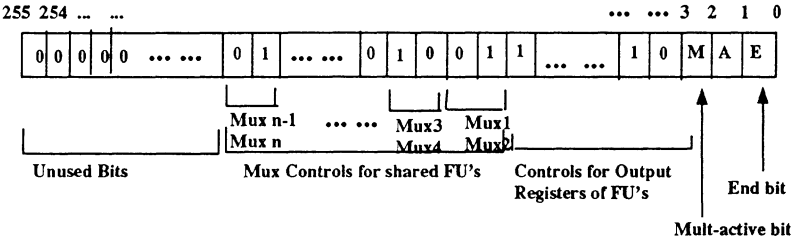


Fig 12. Microinstruction Format

the microinstruction has been designed to have a worst case length of 256 bits. However for smaller designs, this length will be reduced. The microword has bit positions for each of the hardware components in the data path including functional units, registers and multiplexors. Because of the nature of the serial-parallel multipliers used, multipliers require n clock cycles to compute a result where n is the datapath wordlength.

Adders and Subtrators on the other hand, require only one clock cycle to compute a result. This means that multipliers are active for more than one clock cycle for each datum. The microword has a bit position **M** to indicate whether the multiplier is active in any control step. The second bit position **A** indicates whether an adder, subractor or comparator is active in a control step. If both **M** and **A** bits are not set, it means that the particular instruction is a NOP. Each functional unit has an output register (Fig. 10). The microword has bit positions for each of the output registers. In the case of multipliers, since one of the inputs is serial, data travels through a parallel to serial shift register to one of its inputs, and the output which is generated in serial is stored in a serial to parallel shift register (Fig. 10). Control bits for these and the rest of the registers in the datapath are provided in the control word. The sharability of the operators involves certain data routing mechanisms handled by multiplexors. The control word has bit positions for each of these multiplexors in the datapath. All the bit positions are dynamically allocated for each datapath that is synthesized.

Fig. 13 shows the architecture of the microprogrammed control unit. The main building blocks are the micro-control memory, a microprogram counter, microinstruction register and additional circuitry for address generation. In addition there is another control block that does the basic clock routing and event sequencing. The main tasks that are carried out in and by the control unit are:

1. Microinstruction access and loading
2. Appropriate signals are sent to the multiplexors in the datapath for proper functional unit input selection.
3. Signals (clock pulses) for shifting bits in and out of the input and output shift registers (including serial to parallel and parallel to serial registers in the case of multipliers, and the regular parallel in parallel out registers).
4. Control signals to the output registers of the functional units.
5. Control signals to the functional units.

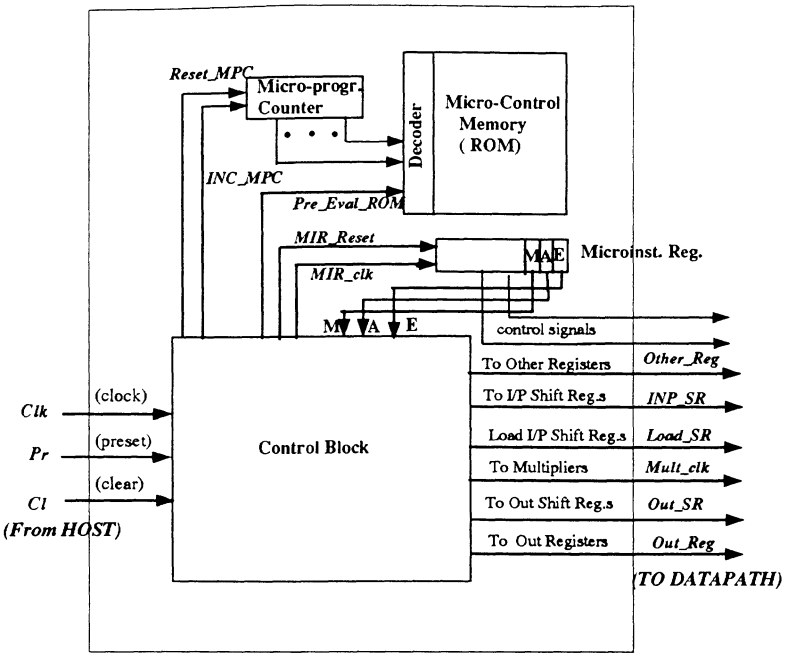


Fig. 13. The Microprogrammed Control Unit

The detailed operation of the control unit is beyond the scope of this chapter. Details are available in [14].

The Physical Design Environment

In Sphinx, the physical design environment (PDE) consists of several interacting tools integrated with the Cadence Edge Framework. Fig. 14 shows the overview of the PDE.

The goal of the PDE is to bring form to the various abstract components manipulated by the higher level tools in the scheduling and data path/control path synthesis phases of the methodology. The PDE has been

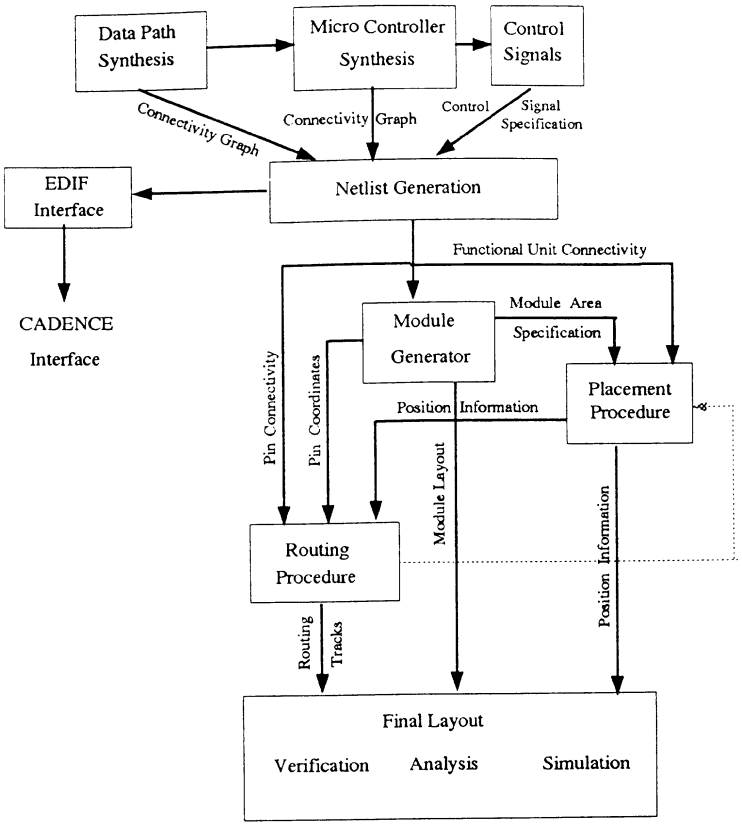


Fig. 14. The Physical Design Environment

implemented in SKILL — a procedural design language. SKILL has been chosen as the medium of implementation because of the ease with which one can access and manipulate the design data. The PDE comprises of a set of parameterized module generators along with a netlister and other associated tools to aid in the generation of the target architecture. In addition to providing for tools that generate the actual layouts for the architecture, the PDE also integrates the rest of the system with the un-

derlying Cadence framework. The core sub tasks include netlisting and module generation.

The Netlister

The Netlister derives its input from the outputs of data-path synthesis and microcontroller synthesis programs, reading them in the form of graphs. Using information on word size, it generates a netlist specification of the system, describing the complete connectivity at the signal level, functional unit by functional unit. It also specifies, for each functional unit, its size and type to aid in the module generation process. The overall process is shown if Fig 15.

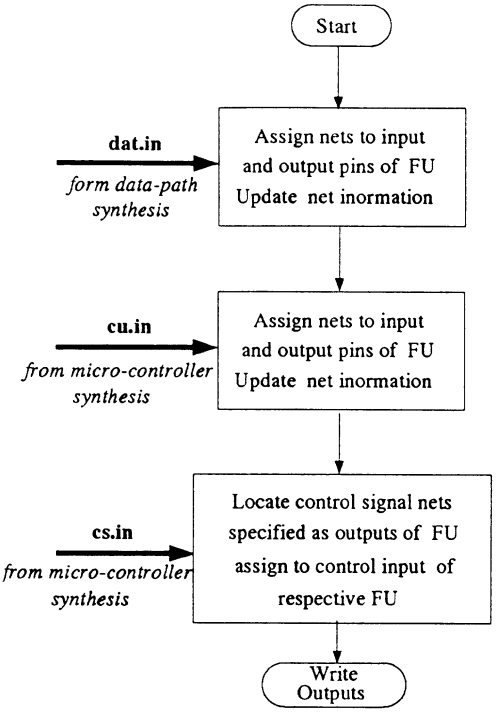


Fig 15. The Netlisting Process

The first step in netlisting is the creation of a net database. The net database contains information for every pin considered. Information stored in the database includes pin bit position, port number, FU number, the driving port and corresponding FU (if it is an input pin). It is assumed that the bit sequence of pins is maintained between the driving and driven terminals (i.e. pin i on a port will be driven by pin i on the driving port).

This net database is searched for existing nets on pins during net assignment. If a pin on FU has a net already assigned to one of the pins it connects to, then the same net name is assigned to this pin, else a new name is created and assigned. Thus if an input pin is not found in the net database, it is assigned a new net name. Then, when the output that drives it is considered, the search will reveal the net associated with the input pin that this output pin drives, and assigned to the output pin. Thus connectivity is ensured over the FU's of the system

The netlisting process consists of three parts. Datapath and control synthesis subsystems provide the netlister with data in the following ascii formats.

```
<FU id #> <FU type #> <# of left inputs>
              (for each left input..)       <from FU> <from port>
                                    <# of right inputs>
              (for each right input.)       <from FU> <from port>
                                    <# of outputs>
              (for each output..)   <# of fanouts for that output>
              ( for each fanout..)           <to FU> <to port>
```

The data specified by the files in the above format provide the netlister with a complete interconnection pattern of the entire data path and control unit. Using these interconnection patterns and information from the module database, the netlister subsystem assigns net names to each terminal of the components defined by the datapath and control unit systems. The output of the netlister is a plain ascii file with each record having the following format:

```
<FU type #> <FU I size>    <FU O size>

                                O          <list of output nets>
                                I          <list of input nets>

                                C          <list of control nets>
```

The output of the netlister is then used by the moduleMaker.

The moduleMaker

The moduleMaker environment is where the abstract descriptions of the datapath and the control unit become a physical reality. Figure 16 shows the structure of the module making environment. At the core of

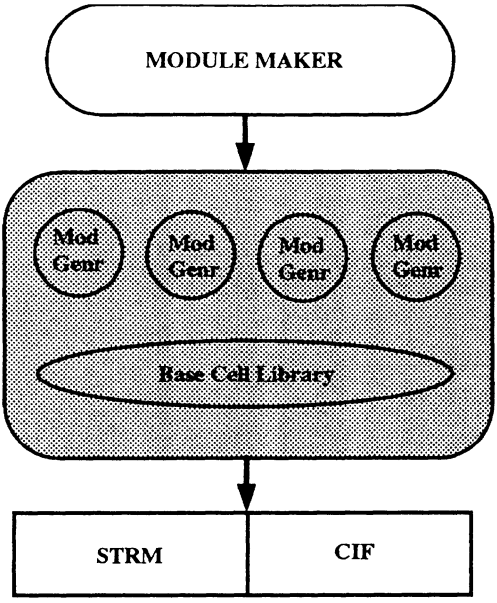


Fig. 16 The Module Making Environment

the environment is a collection of parameterized module generators working in conjunction with a centralized module base cell library. The base cell library consists of primitive cells ranging from simple gates to flip flops to adder cells. The input to the moduleMaking environment is the netlist generated by the Netlister. The moduleMaker parses the netlist file extracting FU identification and size information. It then searches a module mapfile, **fu2name.map** that maps FU identification codes to module names and module generator procedures. The map file has entries that have the following format :

```
FU #    MODULE GEN NAME     MODULE PREFIX     PROCEDURE NAME     COMMENTS
```

The file is formatted with comments in a format easily readable and parsed by IL.

When it locates the FU code that matches the search key, the **module prefix** field is used to create a block for the module. The blockName assigned is generated by concatenating the prefix to an underscore and the size of the functional unit. The moduleMaker then loads the SKILL parameterized module generator specified by **module generator name** and makes a call to the procedure **procedure name**, passing origin coordinates and the module size as parameters. This is repeated for all the functional units specified in the netlist that already do not have existing generated modules. The output of each of the module generators is a Cadence Layout file. Using standard translation techniques, stream and CIF versions of the layouts are also generated. At the end of the module making process, the module maker has accumulated several layouts which are logically interconnected. An example module generator to generate parameterized full adders and a prototype program for the moduleMaker are shown in Appendix A.

Summary

In this chapter, we have presented a menthodology for the synthesis of application specific integrated circuits. The methodology covers the

complete design flow from behavioral specifications in Verilog to the generation of mask level descriptions in CIF/Stream.

In the prototype system, there are tools for supporting different kinds of scheduling and allocation algorithms for the synthesis of non-pipelined and pipelined architectures. These tools facilitate a variety of search space tours. The low end physical design tools have been integrated with the basic Cadence tool set and together they are very versatile. The using of Skill as the design and extension language makes the modifiabilty of the low end physical tools simple.

The methodology is still undergoing constant change with new tools being added at all levels in the methodology. The methodology is being extended to allow system level partitioning, synthesis of multichip modules, and the use of VHDL as a behavioral specification language.

References

1. M.R. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Application," *IEEE Trans. on Computers*, Vol. C-30, no. 1, pp. 24–40, Jan 1981
2. The Verilog Hardware Description Language, Cadence Design Systems, CA
3. IEEE, IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076, 1987
4. silage ref
5. H. DeMan *et al.*, "Cathedral-II: A silicon compiler for digital signal processing", *IEEE Design and Test*, Dec 1986, pp. 13–25
6. E.A. Lee et al., "A Design Environment for DSP," Tech Report, UC Berkeley, 1989
7. M.A. Bayoumi et al, "From Verilog Descriptions to Data Flow Graphs — A Prototype Design," Center for Advanced Computer Studies, May 1992
8. D. Gajski, Silicon Compilation, Addison Wesley, 1988

9. N. Park and A.C. Parker, "Sehwa: A Software Package for the Synthesis of Pipelines from Behavioral Specification,"*IEEE Trans. on CAD*, vol. 7, pp. 356–370, Mar. 1988
10. P.G. Paulin and J.P. Knight, "Force Directed Scheduling for the Behavioral Synthesis of ASICs," *IEEE Trans. on CAD*, pp. 661–679, Jun 1989
11. Robert Walker, Survey of High Level Synthesis, Kluwer Academic Press, 1991
12. P. Kogge, The Architecture of Pipelined Computers, McGraw Hill, 1981
13. M. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions", in *Proc of Design Automation Conference*, pp. 474–480, June 1986
14. A. R. Thipanna *et al.*, "A Microprogrammed Control Unit for the control of pipelined architectures in Sphinx,", CACS Tech Report, Feb 1993.
15. M. A. Bayoumi *et al.*, "Sphinx: A High Level Synthesis System for DSP ASICs," in *Proc. of International Symposium on Circuits and Systems*, May 1992

Appendix A Sample SKILL Programs

Sample module generator

```
;N BIT ADDER parameterized module generator
;design: ajay nath, ramakrishna.
;Modified Mar 1993 for SPHINX
load "/home/u18/SPHINX/module_gen/library"
procedure( n_adder(xloc yloc inputs)
 prog( (blob)
 saveEnv()
 printMessage("Generating %d bit Adder .." inputs)
 layer = getCurrentLayer()
 setWEnvVar("screenGridMultiple" 8)
 setMaster("/home/u18/CellLib_fall91_2micron/base_cells/adder_2 layout")
 for(i 0 (inputs-1)
  instance(xloc+grid(i*17):yloc)
  setEnvVar("pathWidth" 3)
  setLayer("metal2")
   path(xloc+grid(i*17)+3.5:yloc-2 xloc+grid(i*17)+3.5:yloc+grid(11)+2)
;to stretch the input A of the adder on either side..
   sprintf(blob concat("A" i))
   label(xloc+grid(i*17)+2:yloc-4 blob "" "" 2 "")
   label(xloc+grid(i*17)+2:yloc+grid(11)+2 blob "" "" 2 "")
;make pin for abgen
   setLayer("metal2" "pin")
   setEnvVar("labelText" blob)
   setEnvVar("io" "input")
   relRectangle(xloc+grid(i*17)+2:yloc 4 4)
   relRectangle(xloc+grid(i*17)+2:yloc+grid(11)-2 4 4 )
  setLayer("metal2")
;to label A's in the module...
   path(xloc+grid(i*17+11)+3.5:yloc-2 xloc+grid(i*17+11)+3.5:yloc+grid(11)+2)
;to stretch the input B of the adder on either side..
   sprintf(blob concat("B" i))
   label(xloc+grid(i*17+11)+2:yloc-4 blob "" "" 2 "")
   label(xloc+grid(i*17+11)+2:yloc+grid(11)+2 blob "" "" 2 "")
;make pin for abgen
   setLayer("metal2" "pin")
   setEnvVar("labelText" blob)
   setEnvVar("io" "input")
   relRectangle(xloc+grid(i*17+11)+2:yloc 4 4)
   relRectangle(xloc+grid(i*17+11)+2:yloc+grid(11)-2 4 4 )
  setLayer("metal2")
;to label B's in the module...
   path(xloc+grid(i*17+16)+3.5:yloc-2 xloc+grid(i*17+16)+3.5:yloc+grid(11)+2)
;to stretch the input S of the adder on either side..
   sprintf(blob concat("S" i))
```

```
   label(xloc+grid(i*17+16)+2:yloc-4 blob "" "" 2 "")
   label(xloc+grid(i*17+16)+2:yloc+grid(11)+2 blob "" "" 2 "")
;make pin for abgen
   setLayer("metal2" "pin")
   setEnvVar("labelText" blob)
   setEnvVar("io" "output")
   relRectangle(xloc+grid(i*17+16)+2:yloc 4 4)
   relRectangle(xloc+grid(i*17+16)+2:yloc+grid(11)-2 4 4)
  setLayer("metal2")
;to label Sum's in the module...
  if( (i == 0) then
   path(xloc+grid(1)+3.5:yloc-2 xloc+grid(1)+3.5:yloc+grid(11)+2)
   label(xloc+grid(1)+2:yloc-4 "Carry_In" "" "" 2 "")
   label(xloc+grid(1)+2:yloc+grid(11)+2 "Carry_In" "" "" 2 "")
;make pin for abgen
   setLayer("metal2" "pin")
   setEnvVar("io" "input")
   setEnvVar("labelText" "C")
   relRectangle(xloc+grid(1)+2:yloc 4 4 )
   relRectangle(xloc+grid(1)+2:yloc+grid(11)-2 4 4)
  setLayer("metal2")
;to label Carry_In in the module...
  )
  if( (i == inputs-1) then
   path(xloc+grid((inputs-1)*17+15)+3.5:yloc-2
        xloc+grid((inputs-1)*17+15)+3.5:yloc+grid(11)+2)
   label(xloc+grid((inputs-1)*17+14)+2:yloc-4 "Carry_Out" "" "" 2 "")
   label(xloc+grid((inputs-1)*17+14)+2:yloc+grid(11)+2 "Carry_Out" "" "" 2 "")
;make pin for abgen
   setLayer("metal2" "pin")
   setEnvVar("io" "input")
   setEnvVar("labelText" "CO")
   relRectangle(xloc+grid((inputs-1)*17+15)+2:yloc 4 4)
   relRectangle(xloc+grid((inputs-1)*17+15)+2:yloc+grid(11)-2 4 4 )
  setLayer("metal2")
;to label Carry_Out in the module...
  )
;make vdd and gnd pins for the module
   ;setLayer("metal1" "pin")
   ;setEnvVar("io" "input")
   ;setEnvVar("labelText" "vdd!")
   ;relRectangle(xloc+grid(i*17)+2:yloc+grid(11)-6 4 4 )
   ;setEnvVar("labelText" "gnd!")
   ;relRectangle(xloc+grid(i*17)+2:yloc+3 4 4)
  ;setLayer("metal2")
 )
 setMaster("/home/u18/CellLib_fall91_2micron/butt_cells/bot_poly_poly layout")
 for(j 1 (inputs-1)
  instance(xloc+(grid(j*17)-4):yloc)
 )
 sprintf(blob concat(inputs "BIT_ADDER"))
 label(xloc+grid(inputs*17/2)-grid(2):yloc+grid(12)+2 blob "" "" 4 "")
```

```
 label(xloc-6:yloc+grid(ll)+4 "LSB" "" "" 4 "")
 label(xloc+grid(inputs*17):yloc+grid(ll)+4 "MSB" "" "" 4 "")
;make a prboundary bounding box for abgen
 setLayer("prboundary")
 rectangle(xloc:yloc-6 xloc+grid(inputs*17):yloc+grid(ll)+4 )
;change representation type to standard cell
 property("" "Representation/type" "standard")
;now the bounding box, and hence the placement area is known
;save dimensions for returning to moduleMaker
 dimhgt = grid(ll)-2
 dimwid = grid(inputs*17)
 bname = getEnvVar("blockName")
 redraw()
 setLayer(layer)
; open symbol rep for editing
; invoke the pdverify abgen, specifying layers file and rules file
 save()      ;save layout
 graphEdit("%s symbol" bname)
 printMessage("Making Abstract reps ..");
 xt =system("pdverify abgen -l /home/u18/Tools/layers -r
   /home/u18/Tools/Mosistools/ABSTRACT_GENERATOR_RULES_FILE
   %s layout" getEnvVar("blockName"))
 if((xt != 0) then
  printMessage("unsuccessfull abgen")
  ringBell()
  )
 )
;this is where we try and make symbol for the module
 prog( (blob)
 printMessage("Making Symbol ..")
;file to put mapping coordinates to..
 sprintf(mapfpath concat(getEnvVar("blockName") "/symbol.map"))
 outf = outfile(mapfpath)
 printlength = 100
 printlevel = 100
;initlialize the lists
 Alist = nil
 Blist = nil
 Clist = nil
 ilist = nil
 olist = nil
 clist = nil
;gu is the increment unit for symbol to make it a decent size
 gu = .0625
 xloc = 0
 yloc = 0
 width = 0
;decide on the height of the symbol box
 height = 10*gu + inputs*gu
 setEnvVar("fontHeight" .1)
 setEnvVar("labelLayer" "pinlab")
;for all bits make pins Ai Bi and Si
```

```
 for(i 0 (inputs-1)
  sprintf(blob concat("A" i))
  setEnvVar("labelText" blob)
  setEnvVar("io" "input")
  setLayer("device")
  line(xloc+4*gu:yloc xloc+4*gu:yloc+2*gu)
  setLayer("wire" "pin")
  relRectangle(xloc+3.5*gu:yloc+2*gu 1*gu 1*gu)
  label(xloc+3*gu:yloc-2*gu )
;made input pin A
;add location to inputA list
  Alist = append(Alist list(xloc+4*gu yloc+2.5*gu))
  sprintf(blob concat("S" i))
  setEnvVar("labelText" blob)
  setEnvVar("io" "output")
  setLayer("device")
  line(xloc+6*gu:yloc-height xloc+6*gu:yloc-height-2*gu)
  setLayer("wire" "pin")
  relRectangle(xloc+5.5*gu:yloc-height-3*gu 1*gu 1*gu)
  label(xloc+5*gu:yloc-height+2*gu )
;made output pin S
;add location to output list
  olist = append(olist list(xloc+6*gu yloc-height-2.5*gu))
  sprintf(blob concat("B" i))
  setEnvVar("labelText" blob)
  setEnvVar("io" "input")
  setLayer("device")
  line(xloc+8*gu:yloc xloc+8*gu:yloc+2*gu)
  setLayer("wire" "pin")
  relRectangle(xloc+7.5*gu:yloc+2*gu 1*gu 1*gu)
  label(xloc+7*gu:yloc-2*gu blob )
; made input pin B
;add location to inputB list
  Blist = append(Blist list(xloc+8.0001*gu yloc+2.5*gu))
  xloc = xloc + 8*gu
  width = width +8*gu
 )
;make one input list
 ilist = append(Alist Blist)
;place carryin pin and label
 setEnvVar("labelText" "C")
 setEnvVar("io" "input")
 setLayer("device")
 line(0:0 0:2*gu)
 setLayer("wire" "pin")
 relRectangle(-.5*gu:2*gu 1*gu 1*gu)
 label(0:-2*gu )
;append carryin to input list
 ilist = append(ilist list(0.00 2.5*gu))
;place carry out pin and label
 setEnvVar("labelText" "CO")
 setEnvVar("io" "output")
```

```
 setLayer("device")
 line(xloc+4*gu:yloc-height xloc+4*gu:yloc-height-2*gu)
 setLayer("wire" "pin")
 relRectangle(xloc+3.5*gu:yloc-height-3*gu 1*gu 1*gu)
 label(xloc+3*gu:yloc-height+2*gu )
;add carryout to output list
 olist = append(olist list(xloc+4*gu yloc-height+1.5*gu))
;save i/o pin locations to file
 println(olist outf)
 println(ilist outf)
 println(clist outf)
 close(outf)
;make device box
 setLayer("device")
 rectangle(-4*gu:0 xloc+8*gu:yloc-height)
;make bounding box
 setLayer("instance")
 rectangle(-4*gu:3*gu xloc+8*gu:yloc-height-3*gu)
;symbol looks ok .. gotta add some properties
;adding instanceName property
 setEnvVar("labelLayer" "devlab")
 label(xloc/2-4*gu:yloc-height/2 "Adder" )
 setEnvVar("labelLayer" "instname")
 l=label(xloc+8*gu:yloc "[@instanceName]")
 property(l "Label/labelType" "nlpExpr")
 save()
 setLayer(layer)
 setWEnvVar("displayStartLevel" 0)
 setWEnvVar("displayStopLevel" 20)
 fullPlus()
;return module dimensions
 restoreEnv()
 return(list(dimhgt dimwid))
 )
)
```

Listing of moduleMaker

```
;this program is a SKILL/IL program to create modules
;it reads the netlist and the mapping files
;and opens the corresponding hierarchies of modules
;the representations created are layout, abstract, symbol
;design: ajay nath, ramakrishna.
procedure(moduleMaker()
;CAUTION  netlist parsing area starts here (IL)
;specify netlist input and map file names here
 netlist = "/home/u18/SPHINX/work/net.out"
 mapfile = "/home/u18/SPHINX/work/fu2name.map"
 dimfile = "/home/u18/SPHINX/work/module.dim"
 nfile = infile(netlist)
```

```
 if(nfile == nil then
  printf("Cannot open netlist %s  for reading\n" netlist)
  exit(1)
  )
 ;printf("Reading Netlist %s \n" netlist)
 netexpr = lineread(nfile)
 while( (netexpr != nil)
  if( (netexpr != t) then
;parse the fu id and the input and output sizes from the list
   fu_id = car(netexpr)
   fu_is = nth(1 netexpr)
   fu_os = nth(2 netexpr)
;printf("checking for fu %d sizes %d %d \n" fu_id fu_is fu_os)
;get the module generator path for the fu by scanning the mapfile
   modlist = getpath(fu_id)
   modprefix = car(modlist)
   modpath = nth(1 modlist)
   procname = nth(2 modlist)
;call the module generator ( ?? if input size != output size )
;get the heirarchy blockName
   heir=concat("/home/u18/SPHINX/work/" modprefix "_" max(fu_is fu_os))
;get the module generator program name
   modgen = concat("/home/u18/SPHINX/module_gen/" modpath)
   ;printf("Making module %s\n" heir)
;check to see if module exists, else call module generator
   if(((infile(get_string(modgen)) != nil) && (isDir(heir) ==nil)) then
    load(get_string(modgen))
    graphEdit("%s layout" get_string(heir) )
    dimlist = eval( list(procname 0 0 max(fu_is fu_os)))
;module generator returns the dimension of module, append to file
    dfile = outfile(dimfile "a")
    fprintf(dfile "%d %d %d %d %d\n" fu_id fu_os fu_is car(dimlist) cadr(dimlist))
    close(dfile)
   )
  )
  netexpr = lineread(nfile)
 )
 ;printf("Netlist %s read in completely\n" netlist)
)
procedure(getpath(fu_num)
 prog( (mfile mapexpr)
 mfile = infile(mapfile)
 if(mfile == nil then
  printf("Cannot open %s for reading\n" mapfile)
  exit(1)
  )
 ;printf("Reading mapfile %s \n" mapfile)
 mapexpr = lineread(mfile)
 while((mapexpr != nil)
  if( ((mapexpr != t) && (car(mapexpr) == fu_num))
   return(list(nth(2 mapexpr) nth(1 mapexpr) nth(3 mapexpr)))
  )
```

```
 mapexpr = lineread(mfile)
 )
 printf("ERROR : Functional unit %d not found in map\n" fu_num)
 exit(1)
 )
)
```

2

SYNTHESIZING OPTIMAL

APPLICATION-SPECIFIC DSP ARCHITECTURES

Catherine H. Gebotys
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo Ontario N2L 3G1

This chapter will examine previous research on architectural synthesis for DSP systems including a definition of the problem and a description of some approaches currently being used to solve it. Solution approachs to high level synthesis problems are heuristic-based algorithms, graph-theoretical based algorithms, and integer programming based optimizations. The focus of this chapter is on the later approach. After an introduction to integer programming is presented, two IP models are defined which simultaneously solve several important tasks for DSP architectural synthesis. Results are reported and concluding remarks will be made concerning both models and their impact on this important area of research.

Due to emerging low cost digital VLSI technologies, the interest in application-specific architectures has become widespread. Not only can high volume products benefit from VLSI but even medium to low volume DSP applications are economically viable. For example VLSI programmable technologies, such as FPGA, which offer low cost VLSI (zero non-recurring engineering costs) has widened the market. FPGAs are a very popular vehicle for fast prototyping and offer reprogrammability. There are several advantages of application-specific architectures over using instruction-level programmable off the shelf DSP processors. First the architecture itself can be tailored to the

performance requirements of the DSP application. This is unlike off-the-shelf components which may not offer sufficient performance. For example one can significantly reduce latency by adding more functional units (multipliers, accumulators). Furthermore smaller controllers may allow more registers to be put on the VLSI chip thus increasing the performance of DSP application-specific architectures. In order to take advantage of the offerings of new VLSI technologies and obtain significant performance improvements over off-the-shelf processors one needs a tool which can explore a wide range of application-specific architectures. The tool should take into consideration the application performance requirements as well as the constraints of the VLSI technology to obtain the optimal architecture for the chosen VLSI technology. The tool for this task is called a high level synthesis tool or architectural synthesis tool. The input for this synthesis tool is a description of the DSP application in a data flow graph, z-diagram, or language input format. The DSP applications performance requirements must also be specified normally in terms of speed, latency, and throughput. The application constraints would be a description of the timing or frequency of arrival of input signals, requirements on timing of output signals and other I/O signal timing constraints. The technology constraints would be the maximum number of I/O pins per chip, and the maximum area per chip. The output of the architectural synthesizer is a description of the architecture (number of multipliers, ALUs, register, busses, etc) and a schedule which describes how the application is to be performed on the architecture by mapping the operations in the application into control steps.

2.1 INTRODUCTION

In general terms the objective of high level architectural synthesizers is to transform an input algorithm (or behavior) into a hardware architecture that minimizes an area-delay cost function. It is well known that these early decisions made during high level synthesis have the greatest effect on the final VLSI design implementation. It is critical to consider interconnect costs during synthesis since interconnect is seen as the key to high performance architectures. Synthesizers should efficiently produce optimal architectures and handle complex constraints and cost functions. In addition functional pipelining and interfacing to asynchronous and analog processes must be supported during architectural synthesis. The architectural synthesis problem involves several interdependent scheduling and allocation subtasks, all of which must be solved simultaneously in order to provide optimal solutions. The architectural synthesis problem is believed to be NP-hard, since many of its subtasks have been defined as NP-complete. For a thorough review of this problem see[1] .

This chapter examines briefly the previous researched approaches to architectural synthesis including a definition of the problem solved and the approach used to solve it. Solution approachs to high level synthesis problems are heuristic-based algorithms, graph-theoretical based algorithms, and integer programming based optimizations. The focus of this chapter is on the later approach. After an introduction to integer programming is presented, two IP models are defined which simultaneously solve sveral important tasks for DSP architectural synthesis. Results are reported and concluding remarks will be made.

We first define some frequently used terms the reader will find helpful in understanding the function of different subtasks of synthesis. There exist various media for input representation. We will assume the most general (intermediate) form of an input algorithm, a directed acyclic graph (DAG), where the nodes represent the code operations, and the directed edges (arcs) represent the data transfers between code operations. Any algorithm or z-diagram can be represented by a DAG. *Modules* refer to hardware units which will be defined (in functionality) with operations at some later point. *Functional units* refer to digital hardware units (for example an ALU) that perform a defined set of computations on the input data and provide new output data. For example one functional unit may be a 3 cycle pipelined multiplier and another functional unit may be a 2 cycle pipelined multiplier (not pipelined). *Scheduling* refers to the assignment of code operations to a control step. Since processing is synchronized with a global clock, time is an integer value. We use the term control step (cstep) to represent the state of the synthesized architecture where control step 1 is present after the architecture is powered up and initialized. The execution time of the algorithm (Te) is defined as the minimum number of csteps required to execute the input algorithm or DAG on the synthesized architecture. The latency (or delay) is the execution time multiplied by the clock period. Throughput is the number of applications completed in one cstep. *Allocation* is the determination of the number of hardware units such as functional units, registers, and busses. For example, four registers may be allocated, however the variables that are stored in each register have not yet been determined. A schedule may require 3 modules, which may be defined (through binding code operations; addition and multiplication) as 2 adders and one add/multiply functional unit. If the add/multiply functional unit does not exist in the library then 4 functional units (3 adders and 1 multiplier) may be necessary. The number of modules is a lower bound on the number of functional units to be allocated. In general the term resource will refer to functional units, busses, and registers.

Design style defined in [2] refers to the types of functional units, for example an adder or an ALU, to be used in synthesis. For example if one chooses a 115ns clock period and one type of multiplier with a 100ns propagation delay and 20ns delay adder, then one cycle is required by the multiplier and in one cycle four successive additions can be performed (thus four adders can be chained together). However there may exist another multiplier in the library which has a larger latency, 190 ns, but smaller area. Therefore it may be possible to chain the multiplier and adder together, therefore defining a new type of functional unit (which computes (x * a + b) in two clock periods). Most DA tools assume that the clock period is defined before synthesis so that the operational characteristics of the functional unit are known. They also assume that the selection of functional units from the library is done before high level synthesis. However the clock speed, design style selection and scheduling and allocation tasks are highly interdependent.

The output of the architectural synthesizer that we will address are the following:

- total number of control steps, functional units, busses and/or multiplexors, registers and/or register files, memory.
- *scheduling*: code operations to control steps.
- *functional unit allocation and selection*
- *register allocation*
- *interconnect allocation*

One must also determine the type of hardware (type of functional units or memory versus registers) to be used in the final architecture. In some cases the former is done during architectural synthesis. The final schedule and binding produced by the architectural synthesizer can be transformed into a control table for input to a logic synthesizer.

The architectural synthesis problem involves many subtasks such as scheduling (S), resource allocation (A), and resource binding (B). However each of these steps are heavily interdependent. An example of the interdependence between the subtasks is shown in figure 2.1. We will use the term hardware resource to describe the number of registers, functional units, and busses. For example a fixed schedule directly determines the minimum number of functional units and registers (allocation). The subsequent binding of these resources directly determines the minimum number of multiplexors (allocation) required in a multiplexed architecture ($S \Rightarrow A \Leftrightarrow B$). Another design approach which illustrates the interdependence in figure 2.1 is to first perform resource allocation. This allocation will constrain the scheduling and subsequently

constrain the binding (A ⇒ S ⇒ B). It is also easy to see that binding affects scheduling (B ⇒ S). For example operations bound to the same resource cannot be scheduled at the same time.

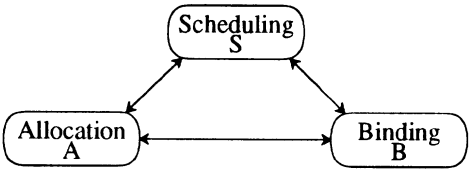


Figure 2.1. Subtask interdependence in architectural synthesis.

The optimal approach to solving architectural synthesis is to simultaneously consider all tasks at the same time. However since this is a very complex approach most researchers have concentrated on one or a limited number of subtasks to be solved simultaneously. We will briefly review research in this field with emphasis on graph theoretical results and integer programming (IP) approaches. We will describe the complexity of these subtasks and overview the iterative/simultaneous approaches to architectural synthesis. More detailed analysis of architectural synthesis material can be found in papers such as[3, 4] .

2.2 PREVIOUS RESEARCH IN HIGH LEVEL SYNTHESIS

In this section we will study the various subtasks associated with architectural synthesis. The graph theoretical problems, their complexity and solutions are discussed for independent and simultaneous solutions of subtasks. The scheduling of a DAG without resource constraints can easily be performed in polynomial time[5] using the well known critical path method, CPM. This algorithm calculates the critical path and the as soon as possible (asap) and as late as possible (alap) control steps[5] for each node of the DAG. This algorithm executes in $O(n^2)$, where n is the number of nodes in the DAG. An example DAG, representing the operations $w=y*z; x=((a+b)-c*d+w)$ and illustrating the asap and alap schedules, are shown in figure 2.2 a) through c). The bottom empty circle is used to ensure that the variables x and w are output at the end of the algorithm. The alap schedule can be calculated for any upper bound on the number of clock periods by incrementing the previous alap csteps by $(T_e^{UB}-T_{CP})$ number of csteps, where T_{CP} stands for the minimum number of csteps in the critical path. The asap schedule obviously is valid for any upper bound on T_e. Therefore this processing needs to be done only once per application (or input algorithm).

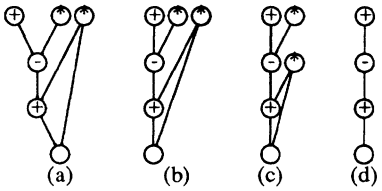


Figure 2.2. DAG (a) and corresponding ASAP (b), ALAP (c), and critical path identification (d).

The asap and alap schedules have not been successfully used for subsequent resource allocation in architectural synthesis because most often they require too many hardware resources. In figure 2.2 the asap and alap requires 3 modules (2 * and 1 +/-) and 2 modules (1 * and 1 +/-) respectively. However these schedules are very important for an initial analysis of the synthesis problem by providing the range of valid control steps (which do not violate any partial order constraints) for each code operation.

Almost all resource allocation in architectural synthesis problems for a fixed schedule have similar structure. We will represent a graph, $G=(V,E)$ as a set of vertices V and edges E. In general the scheduled DAG is transformed into another (conflict or compatability) graph. By further classifying this graph (chordal or interval) one can either solve the problem optimally using a known polynomial time algorithm or heuristically using a similar algorithm. We will use register allocation as an example to illustrate the transformation and solution process that previous research [6] has examined. Not only is register allocation an interesting subtask, but its simple solution for basic blocks presented in this section becomes even more difficult (NP-complete) to solve simultaneously with the scheduling problem.

Although some of the problems presented in this section for general graphs are NP-complete, such as vertex coloring, they can be optimally solved using known algorithms in polynomial times if the graph is of a particular type[7] . It is interesting to note that the same types of characterizations exist in integer programming (IP) and often for the same problems.

We assume that the DAG is scheduled in figure 2.3 a) in four control steps (including the last cstep for the last node whose incident edges are the output variables). Each variable can be represented in an interval representation shown next to the DAG. In the interval representation, the lifetime of each variable is represented by a vertical edge starting at the cstep the variable is defined (output by a code operation) and ending at one cstep before the latest cstep where an

operation uses the variable as input. This interval representation is convenient for register allocation because we have to find sets of variables, such that in each set the lifetimes of the variables are disjoint (or in other words no two lifetimes of the same set have the same cstep). Thus each set represents a register. We will next define the graphs and then define the algorithms.

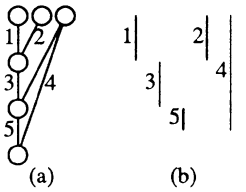


Figure 2.3. Scheduled DAG (a) and the variable lifetimes shown with an interval representation in (b).

The compatibility graph, G^c, is formed from the interval representation. Each edge of the interval representation becomes a vertex of the graph G^c. Edges are formed between all pairs of vertices in G^c whose corresponding variable lifetimes are disjoint (originally called "comparable" vertices[8]). In other words two variable lifetimes are disjoint if there exists no cstep that intersects the lifetime of both variables. The conflict or interval graph[7] , G^i, uses the same definition of vertices as G^c however edges are formed between all pairs of vertices whose variable lifetimes are not disjoint or in other words have overlapping lifetimes (or are "incomparable"). Another characteristic we can observe from these two graphs is that G^C is the complement [1] of G^i.

Register allocation is performed on G^c by a clique partitioning algorithm. Clique partitioning essentially removes edges from G^c so that the remaining graph is a number of disconnected cliques. The algorithm tries to produce a minimum number of disconnected cliques. A clique of a graph G is a maximal complete subgraph. We will use the notation K_x to represent a clique on x nodes. The number of cliques is equivalent to the number of registers. Alternatively the register allocation problem can be solved on graph G^i using vertex coloring. The vertex coloring of the interval graph, can be solved using a polynomial run time algorithm or the left edge algorithm also presented for solving channel routing problems in[8] . The number of colors is equivalent to the

1 The complement of graph G is $\overline{G}$; ($\overline{\overline{G}}=G$).

number of registers. In fact the minimum number of cliques in G^C is equivalent to the minimum number of colors (or independent sets which cover the graph) in G^V. These two algorithms are hence complementary. The clique partitioning approach was first presented in Facet[6] . It was shown in [9] that larger problems could be solved faster than using the interval graphs.

In the presence of conditional code there may be more than one edge used to represent a variable's lifetime. For example a variable defined before a branch on conditional code, but whose last use is at different csteps inside each branch. Thus the graph is no longer an interval graph and one cannot minimize registers in general. REAL[10] heuristically extended the left edge algorithm for conditional resource sharing register allocation. However in[9] specific types of conditional code that formed chordal graphs (of which interval graphs are a subset), were identified thus showing that one could for some cases minimize the number of registers in the presence of conditionals. Minimizing registers in loops, where variable lifetimes are defined on a circle, was also solved by using an arc coloring algorithm in[2] .

Functional unit allocation is complicated by the fact that the mapping of operations to type of functional units may be a one to many mapping. In other words a selection of types of functional units for each operation must be performed. Many synthesis systems reduce this complexity to a one to one mapping, by preselecting the types of functional units, and therefore do not simultaneously select functional units when performing allocation. Facet[6] performs functional unit allocation using the clique partitioning algorithm. The user provides a scheduled DAG and Facet solves each allocation task, including register, functional unit and interconnect allocation, independently using a clique partitioning heuristic algorithm.

MIMOLA[11] uses a integer linear programming model (IP), with branch and bound solver, to obtain the number of functional units required for a fixed schedule. However it could not apply this IP to bind operations to functional units due to its large model size.

The problem of bus allocation with a fixed schedule is also very similar to register and functional unit allocation. The number of data transfers per cstep are used to calculate the number of busses. If one wants to allocated all general busses (multiplexors and busses) there is a problem with using global data broadcasts. A global data broadcast is a transfer of one data value from one source to more than one destination. If one counts the number of distinct sources (accounting for a global data broadcast as one transfer) then this will not account for extra multiplexors which may be required at the inputs of functional units. On the other hand if one counts the data broadcast by the number of destinations then one may overestimate the number of busses. In most synthesis

systems it is assumed that the extra multiplexors required will be substituted later in the design process, and the number of sources for data transfers is counted. Interconnect optimization with a fixed schedule and a fixed number of functional units, [12] for register-transfer file architectures with separate read and write clock phases was examined using a simulated annealing approach.

2.2.1 SOLVING TWO OR MORE SUBTASKS SIMULTANEOUSLY

Scheduling and functional unit allocation were the first two most common subtasks to be considered simultaneously. Previous research[13] for scheduling multiprocessor systems such as list scheduling[14] has had a large impact on the architectural synthesis application. We will use this application to introduce and define the problem. A brief overview the architectural synthesis applications will then be performed. This scheduling and functional unit allocation problem is similar to the precedence constrained scheduling problem formally defined in [13] as:

" A set T of 'tasks' (each assumed to have 'length' 1), a partial order $<$ • on T, a number of 'processors' and an overall 'deadline' $D \varepsilon Z^+$.

Is there a 'schedule' $\sigma: T \rightarrow \{0,1,..,D\}$ such that, for each $i \varepsilon \{0,1,...,D\}$, $|\{ t \varepsilon T : \sigma(t) = i \}| \leq m$, and such that, whenever t $<\bullet$ t', then σ(t) $<$ σ(t')?"

This problem was proved [15] to be NP-complete. The precedence constrained scheduling problem for DAGs with an intree structure [16] were shown to have a polynomial time solution and outtree examples were shown to be NP-complete. This research was the start of a technique called list scheduling[14] which has since been refined for architectural synthesis, such as[17, 18] . An example of an intree DAG is matrix multiplication, where an intree structure has its leaves representing multiplication operations and the rest of the vertices of the tree additions. However with a restricted number of functional units the problem of scheduling this computation is more optimally handled using a multiplier accumulator DAG.

The partial order of the quoted precedence constraint scheduling problem represents a data transfer in the architectural synthesis model. The partial orders can be also represented by arcs in a directed acyclic graph representation of the set of tasks. The extensions to the formal scheduling problem for architectural synthesis include : limited mapping of tasks to processors; timing constraints; and complex task operation such as multicycled or pipelined processors.

Research in mapping algorithms onto multiprocessor structures also examines the precedence constrained scheduling problem[13] . For an infinite number of processors one can schedule a DAG to minimize the makespan or execution time of the algorithm using CPM. In multiprocessor applications it is assumed that each processing node of the DAG requires negligible time compared to the time for communication between processors. Therefore the problem in this research area is to minimize the execution time, where execution time is modeled as a function of the number of communication delays required to perform the algorithm[19] . Other research has shown that if we limit our architecture to two modules (and ignoring the communication delay) then given any DAG we can calculate the minimum execution time[20] . This problem maps into a matching problem in a graph which is the complement of the DAG. The matching problem is to maximize |M|, where $M \subset E$ of a graph, G=(V,E), such that each vertex is incident to at most one edge εM. An example shown in figure 2.4 illustrates a matching, |M| =2, thus providing an optimal schedule of 3 control steps for a 2-processor implementation of the five code operations. In fact a valid schedule could also be obtained using the matching algorithm.

If we increase the number of modules beyond 2 the problem is again NP-complete, since we are then looking for a restricted set of cliques of size less than or equal to the number of modules (>2). It is however interesting to look at this application since it illustrates the limitations of purely graph theoretical approaches to solving complex problems. For example as new complex constraints arise during the design cycle using purely graph theoretical approaches may not be viable due to the difficulty in adjusting these algorithms to the new constraints. We will now briefly review previous research that tries to simultaneously schedule and solve functional allocation tasks.

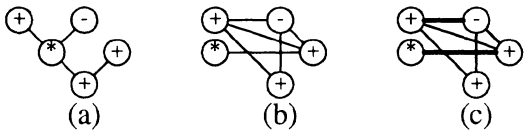


Figure 2.4. Illustration of restricted optimal scheduling for two modules.

Variations of list scheduling techniques are very popular in architectural synthesis as well as multiprocessor compiler design[21] . In general one places an upper bound on or fixes the number of functional units and then schedules operations in a prioritized order. The priority is set by the (alap - asap) value, where a smaller value has a larger priority. Operations are placed in a cstep based upon this priority until all functional units are exhausted. Then operations are placed in the next csteps in the same manner. HAL[18] uses an iterative refinement heuristic algorithm based on force directed list scheduling to perform scheduling and functional unit allocation. Recently extensions to provide heuristics to minimize registers and interconnect have been incorporated. The number of parallel data transfers, using transfers with distinct sources counting as one transfer, were used to heuristically approximate the number of busses. However the exact relationship to number of busses was not defined.

The mathematical approaches to simultaneously solving more than one subtask of the architectural synthesis problem will be outlined now. In these examples the scheduling was simultaneously solved with more than one subtask. However no previous research to our knowledge has tried to simultaneously schedule and allocate busses, only estimates of busses are used to guide the scheduling task. These examples show how the previously studied independent subtasks, such as register allocation for a fixed schedule, now become very difficult to solve simultaneously with the scheduling subtask.

A mixed-integer linear programming (MILP) model in [22] , solves simultaneous scheduling, functional unit and register allocation using a MILP formulation. In addition scheduling is done in real time and both registers and functional units are selected from a library. A nonlinear model was first formed and then linearized by the addition of binary variables. Unfortunately only very small examples could be solved due to the size of the model and the inefficiencies of the branch and bound technique. For example an input algorithm with 4 code operations required 87 variables, of which 46 had to be integers.

One of the first IP models for resource constrained scheduling was presented in [23] . This same model was recently used in a two step methodology in[24] . The IP formulation was solved using a branch and bound algorithm to produce a schedule that minimizes the number of functional units in one step and the sum of the lifetimes of the variables of the DAG (which heuristically minimizes the execution time and in some instances the number of registers) in the second step. Figure 2.5 shows an example where this heuristic fails to minimize the number of registers. Very fast execution times were obtained most likely due to the improved computer technologies available today as compared to 20 years ago. More importantly by using this two step methodology bounds are kept small by incrementally moving across the design space. However the

bounding argument (which sets the previously solved number of functional units as an upper bound for the present optimization with a larger execution time possible) does not necessarily hold in all cases. For example very often as the execution time (or number of control steps) is increased the number of adders may increase at the added benefit of decreasing a more expensive functional unit such as a multiplier. These tight bounds are very important for solving any IP and in particular for branch and bound techniques (they greatly improve the performance). The model was later extended for functional pipelining in[25] and a heuristic partitioning strategy to decrease the size of the input algorithm, however register allocation still was not incorporated.

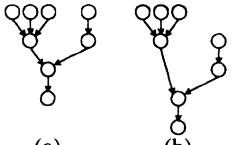


Figure 2.5. An example where sum of the lifetimes of the variables in the DAG does not decrease the number of registers but favors minimum execution time. In (a) Te=4 (minimum), 4 registers, sum = 7 and in (b) Te=5, 3 registers, sum = 8 are required.

A simulated annealing technique presented in [26] solves simultaneous scheduling, functional unit allocation, and register minimization. The formulation includes a calculated number of registers, and an estimate of interconnect in its cost function. Since the cost functions are used to evaluate two dimensional placements (or fixed schedule and functional unit allocation), the number of registers could be calculated using the left edge algorithm. The number of parallel data transfers was used as a heuristic estimate of the number of busses as defined in HAL, however the relationship was not defined. Another part of the cost function was called links, which tried to estimate the number of bus drivers or multiplexor inputs required. Both fast simple and slower more accurate cost functions are used at different stages of the annealing to improve the efficiency of the annealing since many solutions are searched. Running times were achieved comparable to heuristic techniques. However the rate of convergence to a global optimum[27] is exponential. It was stated that new constraints could be added by changes to the cost functions.

A graph theory approach to the simultaneous scheduling and resource (modules and registers) minimization problem has been examined by[28] . A two dimensional placement of the data flow graph where makespan (or

execution time), graph height (number of modules), and modified cutwidth measurement (estimated number of registers) were defined was used to represent the scheduling problem. The problem is that the cutwidth which can be solved easily includes all edges in the graph and we only need the edges representing the variable lifetimes. Thus we need only consider the longest outdegree arc of each node to represent the lifetime of the variable. This is why a heuristic was needed to solve the problem, since minimizing the lifetime defining edge (maximum length of all edges incident to a node) is NP-complete. A heuristic was used to solve this multiprocessor makespan scheduling problem.

Interface constraints are very important for architectural synthesizers, even though few synthesizers[29-31] can handle these simultaneously with allocation subtasks. Not only are these important for supporting interfaces to external environments but they are also necessary for handling local application specific constraints within the synthesized architecture itself. For example timing constraints are required to model functional pipelining or possibly for multicycled operations. The first synthesizer to consider timing constraints was Elf[32] where a timing constraint for a group of operations was specified. This constraint was generally a minimum or maximum execution time to be met. More recently the Carnegie Mellon University synthesis effort has updated the CSTEP scheduler to incorporate minimum and maximum timing [33] constraints. These constraints can be placed between any pair of operations in the algorithm. The list scheduler uses priority values for operations to decide if they must be placed in a certain control step. Timing constraints are checked and if a constraint is about to be violated by an operation not being placed in a control step then the priority value for this operation is modified to prevent the illegal assignment from being made.

Research at Stanford University [34,35] has examined timing constraints for high level scheduling and logic synthesis. They identify a fixed timing constraint and a unknown unbounded timing constraint. It is assumed that module binding and hardware allocation has already been done, and an iterative algorithm for relative scheduling is presented. The feasibility of timing constraints is defined and an algorithm is also presented.

Timing constraints and their effects on loops and conditional codes [36] for a logic synthesis environment has been investigated. Asynchronous circuit synthesis in[37] or [38] has also been researched but no datapath is synthesized. Design representation in[33] has researched the use of charts to partition synchronous from asynchronous circuitry and perform partial binding of hardware.

In summary we have briefly discussed the different (locally optimal) approaches to state of the art architectural synthesis. The optimization of independent subtasks (of architectural synthesis) was shown to be limited for

certain cases where the graph (obtained from the scheduled DAG) had a particular structure. It was also shown to be very difficult to extend this approach using graph theory for simultaneous solutions of more than one subtask. The previous integer programming approaches either were too large, and could not be solved, or were formulated to solve only a small part of architectural synthesis. Because of these complexities and the fact that architectural synthesis is most likely NP-hard, many researchers have turned to heuristics. In the next section we will discuss the recent successes in integer programming research. In particular this research involves the study of polyhedral characteristics and their use in the solution of large scale integer programming problems. Secondly we will show that unlike graph theoretical techniques even constraints with no apparent structure can often be solved using these techniques.

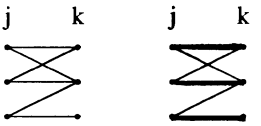
2.3 INTRODUCTION TO INTEGER PROGRAMMING

General integer programming (IP) applications and solution techniques are briefly reviewed in this section. The general formulation techniques for IP, the state of the art solutions of general IP problems, including classical enumerative and heuristic approaches (ie. simulated annealing) and recent successes in polyhedral approaches to solving partially structured IPs are outlined. (The notation for a graph is G=(V,E), where V is the set of vertices and E is the set of edges).

Since many problems from a wide range of applications can be formulated as an IP problem, there is a great deal of interest in trying to efficiently solve integer programming problems. Two of the most important steps in integer programming are preprocessing and model formulation. Both the amount of preprocessing that can be done and the formulation of the model has a great impact on the final IP solution efficiency and accuracy. We will first look at one of the simplest models, the assignment problem, that has many applications. A simple method for formulating constraints that can be represented as logical inferences is discussed next, followed by the definition of disjunctive constraints.

The assignment problem is one of the easiest models to formulate. The variables of the model are binary and each represents the mapping of j elements to k elements. For example figure 2.6a) illustrates a possible mapping choice, where the variables are the edges of the graph, $e_{j,k}$. If $e_{j,k}$ is 1, in the solution, then the assignment of j to k is optimal. Otherwise, if the value is 0, there is no assignment produced by the solution. Although we have used a bipartite graph [2] for illustration this type of assignment or matching is not restricted to these types of graphs alone.

[2] A bipartite graph is a graph with no odd cycle. It can always be partitioned into two groups X and Y (or j,k in figure 2.6).



(a) (b)

Figure 2.6. An assignment problem illustrated by a bipartite graph (G=(V,E)) with two partitions j and k. A solution, $\mathbf{M} \subset E$ is shown in bold in (b).

A perfect matching problem is a set $\mathbf{M} \subset E$ such that each node is incident to *exactly* one edge of M. The binary variables are:x_e=1 if $e \varepsilon M$ or x_e=0 if e is not a member of M. Thus we wish to solve the following optimization problem, where $\delta(u)$ is the set of edges incident to vertex u.

$$Max \ cx$$
$$\sum_{e \ \varepsilon \ \delta(u)} x_e = 1. \ \forall u \varepsilon V, \ x \varepsilon \{0,1\}$$

An example of a matching problem is scheduling tasks of unit duration on a uniprocessor where there are K tasks and J possible control steps during which a task can be performed by the uniprocessor. In this example there is no precedence relationship between the tasks or in other words the tasks are all independent.

The following logic is often very useful in the formulation of an IP model. A representation of logical inferences by mathematical linear inequalities has been examined by [39] and[40] . For example the logical expression or inference $P_1 \Rightarrow P_2$ is equivalent to: 1) $\neg P_1 \vee P_2$[41] and ; 2)1-p_1+$p_2 \geq 1$ or $p_1 - p_2 \leq 0$ [40] , where p_i are binary variables. For example if $p_1 = 1$, then for the inequality to be satisfied, p_2 must also be 1, which is the same as $P_1 \Rightarrow P_2$. Another example is $\neg y_1 \vee \neg y_2 \vee z$ which is equivalent to the mathematical inequality: $y_1+y_2-z \leq 1$.

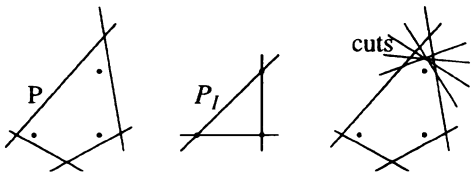
Integer variables can also be used to represent disjunctive constraints[27] or model the activation or deactivation of a continuous variable. For example, y =1 $\Rightarrow L \leq x \leq U$ and y = 0 $\Rightarrow x$=0, can be modeled by the inequality $Ly \leq x \leq Uy$ This represents a disjunctive constraint on x or a (de)activation of a continuous variable x by a binary variable y.

2.3.1 SOLUTION OF UNSTRUCTURED IPs

We will now look at a few general techniques for solving IPs with no apparent structure. These IPs are called unstructured IPs. The first step to solving an IP is to transform the IP into a relaxed linear program (LP) and solve the LP. We transform an IP into a relaxed LP by removing the integrality constraints on the variables and allowing them to be solved as real positive numbers. For example we can replace $x_e \varepsilon \{0,1\}$ with $1 \geq x_e \geq 0$. If we obtain an all integral solution then we have found an optimal solution to our problem. Proof that the solution is globally optimal comes from the duality theory of LPs[27] because we are solving the IP as an LP. In our LP solution if one or more variables are not integral then we have to look for other procedures to solve for the integral variables. This section will address this problem. We will assume that we are solving for binary variables (since any integer variable can be represented by a sum of binary variables).

We will first define some IP terms commonly used. There exists a *bounded polyhedron* for any rational bounded system of linear inequalities. Figure 2.7a) gives an example of a polyhedron defined by its constraints, $Ax \leq b$. We will call the convex hull of integer vectors an *integral polyhedron*. This is also illustrated in figure 2.7b), where the linear inequalities (now called *facets*) intersect at integer values (represented by the dots). These facets are of dimension one less than the dimension of the polyhedron. It has been provn that for any bounded system of rational linear inequalities there exists an integral polyhedron, and in fact the facets are linear combinations of the inequalities defining the polyhedron. Unfortunately for most problems we do not know how to form these linear combinations or in other words we do not know what the facets look like. Furthermore even if we did there may be an exponential number of them. A final term to define is a *cut*. A cut is a valid linear inequality that cuts away fractional values from the existing linear programming fractional solution. For example in figure 2.7c) the lines represent cuts.

General IPs may be difficult to solve [27] due to 1) size of the formulation, 2) weakness of bounds, and 3) speed of the algorithm. For example in 1) the number of variables or constraints may be very large, in 2) the difference of the lower bound and optimal solution of a variables may be great, or in 3) the algorithm for solving the problem may be very slow. Recent success in solving IPs have shown that (in addition to preprocessing) by tightening constraints, or more effectively by using facets,[27] one can dramatically improve the efficiency of solving IPs. We say that one constraint, $\delta x \leq \delta_0$, is tighter, dominates, or is stronger than the other constraint, $\zeta x \leq \zeta_0$, if $\{x \varepsilon R \mid \delta x \leq \delta_0\} \subset \{x \varepsilon R \mid \zeta x \leq \zeta_0\}$, where $x \varepsilon R$, R is the set of real values. In other words let the polyhedron



(a) (b) (c)

Figure 2.7. (a) illustrates a bounded polyhedron, (b) shows the corresponding integral polyhedron, and (c) identifies possible cuts, on the polyhedron of (a).

generated by the first set of constraints be $P^1 = \{x \varepsilon R \mid \delta x \leq \delta_0\}$ and $P^2 = \{x \varepsilon R \mid \zeta x \leq \zeta_0\}$, for the second set of constraints, then $P^1 \subset P^2$. One way to show this is to find a fractional point where $x \varepsilon P^1 \cap \overline{P^2}$, therefore $P^1 \neq P^2$, and show that $P^1 \Rightarrow P^2$. The efficiency of solving the IP is improved due to the fact that tighter models have a smaller set of feasible solutions which must be searched. Branch and bound algorithms can be used to solve IPs in practical times if additionally the model has a small number of variables and tight bounds are known. The most well known general solution techniques for integer programming are the enumerative techniques such as branch and bound or heuristic variations. We will first review one of the oldest techniques for solving IPs, called Gomory's cutting planes algorithm.

Gomory's cutting planes is more interesting from a theoretical point of view than from a practical point of view. Generally Gomory was able to prove that after a finite number of cuts on any bounded polyhedron P, an integral solution can be obtained. He found a general method for obtaining these cuts using the simplex tableau of the LP solution. Unfortunately a very large number of cuts must be generated before an integral solution is found and few researchers use this technique on practical IPs because it takes too long.

The branch and bound method, or variation of it, may be used for a small number of variables (<200). However it is possible that even for small problems the solution may not converge due to the shape of the polyhedron. The bound on the objective function may also be very bad, for example the distance between X^* and X. The objective of the branch and bound technique is to create new LPs by bounding each variable towards integral values. The tree formed, by branching on a variable $x \geq \lceil X^* \rceil$ and $x \leq \lfloor X^* \rfloor$, is expanded only on nodes where the objective function is more optimum. From experience it has been

found that an integral solution may be found quite early yet to finish the algorithm and therefore prove it is a global optimum takes a very large amount of time. Nevertheless it has been widely used for many small problems. Commercial software uses branch and bound techniques and can generally handle up to 200 integer variables[42] .

There exist many heuristic techniques for solving IPs such as greedy algorithms, interchange heuristics, simulated annealing, and others [27] . These techniques tradeoff optimality for efficiency. Tremendous success in solving many engineering problems with simulated annealing has been achieved, even though the convergence to a global optimum is exponential. Since combinatorial optimization problems have many local optima, some heuristic approaches, such as the greedy or interchange algorithm, are often run with random starting points. Simulated annealing is a different approach to avoiding local optima, by allowing the objective value to decrease only occasionally (for a minimization problem), to avoid getting stuck at a shallow local optimum and thus escaping towards another neighborhood with a smaller objective value. A geometry of numbers approach [43] to solve particular IP's that cannot be solved using branch and bound has been researched. Generally IP's with not necessarily a large number of variables but those which exhibit a long needle-like polyhedron were solved using geometrical transformation. Using a quadratic potential function projected on a ellipsoid the recent work of Karmarkar[44] has shown that large sized integer problems known as the satisfiability problems can be solved. However if an objective function is required only a locally optimal solution is possible and there exists no guarantee of finding a solution. Thus this approach seems to be directed towards a problem characterized by a small number of integral optimal solutions.

2.3.2 POLYHEDRAL APPROACHES TO SOLVING STRUCTURED IPs

In general solving an IP problem is NP hard[13] . However, analogous to special graphs in graph theory, there exist special techniques for solving some IPs. Thus all IPs are not equivalent in difficulty in all respects. For example to solve a node packing IP problem on a graph which is claw-free (ie. $\exists$ no $K_{1,3}$ [3]) requires only polynomial time, using Minty's[45] algorithm. This is analogous to the graph theory approaches where polynomial algorithms are known to exist if the graph at hand is of a particular structure (ie. interval graph for polynomial time algorithms that perform node coloring[7]). We say that these IPs have *structure*. Additionally IPs where some constraint has this property are said to

[3] $K_{x,y}$ is a complete bipartite graph with partition x,y.

have *some structure*. In IP we can often obtain good bounds on a particular problem and often solve for integer variables using this structure, even when no known graph theoretical algorithms, heuristics or formulations may exist. But how can we find this structure? We can often do this through proper model formulation.

The research focus over the past 25 years in IP has been to study polyhedra characteristics of a problem and thus define structure which may help in its solution. This was motivated by the desire to obtain tight formulations of the problems rather than adhoc models, since IPs have exhibited extremely erratic performance. A systematic way to obtain these formulations is to analyze facets. Unfortunately there exists no formal method for obtaining facets of a given IP and even if we could find a method to generate all facets, most likely we couldn't solve the LP because there may be an extremely large number of them (possibly exponential). Balas and Padberg[46] have argued that its very useful to find facets or approximation of facets because only a few define optimal points. Also it is known that if one used a branch and bound technique after extracting some facets, the algorithm would generate fewer live nodes [47] and terminate faster. This is mainly due to the better bound obtained from the use of facets. Thus by mapping a problem or subsets of a problem into a well studied class of problems, such as node packing, whose facets are partially characterized one may be able to improve the bounds of the problem and solve for integer variables more efficiently.

Recent research has proven how important facets are. The tremendous success of the use of facetial characteristics is demonstrated with the traveling salesman problem [48] and large sparse unstructured IPs solved by using facets of subproblems in[49] etal. Further research [48,49] has also shown how it is highly advantageous to add facets to the LP until no new ones can be found even before you start to branch and bound.

State of the art solutions of unstructured IP have been researched by [49] using a combination of preprocessing, cutting planes (using knapsack facets of underlying polytopes), and branch and bound techniques to solve sparse 0-1 unstructured IPs of over 2000 variables in reasonable computation times (less than 1 cpu hour). The cutting planes which were facets of the underlying polytope (knapsack inequalities) were extremely useful and successful for exact solution of their class of problems. Their system was completely automatic, and represents state of the art for solving unstructured IPs. When a cut cannot be found a variable is selected to branch on. The definition and characterization of knapsack inequalities is given later in section 2.5.

In 1980, Grotschel[50] demonstrated optimal solution of (over 7,000 integer variable) TSPs in 30 cpu sec to 2 cpu min to show the usefulness of the theoretical research in polyhedral characteristics. In all cases the problems could not be solved using existing branch and bound techniques, thus demonstrating the importance of polyhedral combinatorics in solving large scale optimization problems. In 1980 Padberg[51] solved for 50,000 integer variables of the TSP problem completely automatic to within 0.25% optimality in 30 minutes using automatically generated facets. Unfortunately the number of applications which can be modeled as a traveling salesman problem is not proportional to the large amount of research that this problem has generated. Conversely there are other problems, such as finding the maximum weighted directed cycle in a graph that have a large number of applications, but generated little research. This is also partially true for the node packing problem in a smaller sense as we shall see in section 2.4.

2.4 THE NODE PACKING PROBLEM

There exists a great deal of interest in the node packing problem because of a) the large number of practical applications and b) the stronger structural properties than the general integer programming problem[52] . The node packing problem has also been called vertex packing and the stable set problem. It is also related to other problems in optimization such as the set covering, set packing, anticliques, independent sets, and node covering, [52, 53] which we will not cover in this text. We will first illustrate the relationship between integer programming, graph theory, and node packing, using a simple completely structured problem (that of maximum matching). Secondly we will formally define the problem and then proceed to define the known facets of this problem.

Integer programming and graph theory have many areas of research which overlap. For example figure 2.8a) illustrates a perfect matching problem. Each edge must be assigned a 0 or 1 value to maximize the sum of all edges with the restriction that each vertex is incident to at most one edge with a value 1. Alternatively we can use the Hungarian Method or Kuhn Munkres[54] algorithm to solve for a maximum matching in polynomial time. Alternatively one can solve an IP where constraints correspond to integral facets. In the latter method, we can solve the IP as an LP and be guaranteed to always obtain a solution with integer variables. The second constraint given below can be automatically generated as needed for a particular problem by at most 2n-1 min cut problems on the graph. In other words instead of generating this constraint for all odd sets of vertices we can solve the LP and automatically generate facets to cut away the fractional values and solve for integer variables using the relaxed LP. The complete model for weighted perfect matching is given below, where $\delta(S)$ is the set

of edges, where each edge has one vertex in S and the other vertex in $\bar{S}$.

$$Max\ cx$$
$$\sum_{e\varepsilon\delta(u)} x_e=1,\ \forall u\varepsilon V.$$
$$\sum_{e\varepsilon\delta(S)} x_e\geq 1,\ \forall S\subset V, |S| odd$$
$$x_e\geq 0,\ \forall e$$

The vertex representation of this problem (which we will define in the next section as node packing) is shown in figure 2.8b) where each edge is now a vertex (variable) and edges of this new graph represent adjacent vertices of the matching graph in (a). The graph in figure 2.8b) is a line graph obtained from (a), and it is known that the solution of this problem (node packing) on a line graph[27] can be solved in polynomial time. This example briefly illustrates the relationship between graph theory and integer programming. However this relationship is not true in all cases. For example there exist some problems for which known polynomial algorithms exist (ie. it is well solved) however the associated polyhedron is nontrivial. An example of this is the to find the edge in a node weighted graph which has the maximum sum of its two incident nodes [46] .

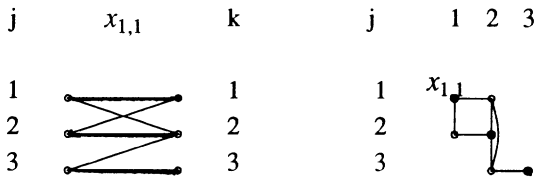


Figure 2.8. The matching problem represented by edge variables in (a) and vertex variables in (b). In the former case one assigns 0 or 1 to edges and in the later case one must solve a node packing problem, by assigning 0 or 1 to vertices.

Like the traveling salesman problem characteristics of the integral facets are partially known[27] for the node packing problem. This problem is more formally stated below in two forms. One form is the graph theoretical view and the second is the mathematical linear system of equations view.

1. In graph theoretical form: Given a graph G =(V,E), maximize $\sum_u c_u x_u$, where c_u is a cost value, such that

$$x_u + x_v \leq 1, \quad \forall (u,v) \in E$$
$$x_u \geq 0, \quad \forall u \in V.$$

2. In linear systems of equations form:

$$\max c^T x, \quad Ax \leq e$$
$$1 \geq x_j \geq 0, \quad \forall j \in N = \{1,...,n\}$$

where A is a m by n matrix (m rows, n columns, with entries of 0 and 1, two 1's per row which identify the two vertices which form an edge of the graph G), c is an arbitrary n-vector, and $e^T = (1,...1)$ is a unit m-vector[52].

If all variable solutions are integral then a globally optimal solution to the problem has been found and we are done. A property unique to the node packing problem is that if not all variables solutions are integral, the variables that are integral remain integral[27] in the optimum solution. Therefore the problem can be decomposed into a smaller problem to solve. However it is also known that this node packing formulation with node edge incidence constraints, generates very poor bounds[47] . Furthermore studies which attempt to use this property to solve the problem have found that in most cases few integer variables are attained[55] . We will discuss the node packing problem using the graph theoretical formulation.

Finding all integral facets for a particular node packing problem is NP-complete. This problem is known as the *stable set polytope* (SSP) problem, using graph theoretical terminology. Nevertheless only integral facets over the region of the minimum objective function are required to obtain integral solutions. We will now define some of these facets.

One known integral facet for the node packing problem is given by $\sum_{u \in K} x_u \leq 1$, *for all K cliques* , where a clique (or maximal complete subgraph) is a subset of nodes K for which there exists an edge in the graph for every pair of nodes in K. There are other facets such as an odd cycle, however we will discuss these since they are not necessary for the IP model to be presented in section 2.7.

2.4.1 EXAMPLE: SCHEDULING DAGS WITH NO RESOURCE CONSTRAINTS

Scheduling nodes of a DAG with no resource constraints is an example of a problem which can be transformed into a node packing problem. Assume that each node of the DAG refers to a task which must be assigned to a time. It is very important to understand this model since it is the basis of the remaining chapter.

Assume we have the following variables $x_{j,k}$, where nodes of the DAG are represented by k and the control steps (units of time) that one must assign these nodes to are represented by j. Assume that all tasks are of unit duration (or require one cstep to complete). Thus if $x_{j,k}$=1 then task k is assigned to (or scheduled at) cstep j. Equation (1) ensures that each task will be assigned to one control step.

$$\sum_j x_{j,k} = 1 \ , \ \forall k. \qquad (1)$$

Inequality (2) , called the precedence constraint, prevents a task, k_1 from being scheduled after task k_2 whenever there is a precedence relationship between these two tasks such that $k_1 <\bullet k_2$. In other words task k_1 must be completed before task k_2 can be started. For example task k_1 produces data which must be used by task k_2. In a DAG representation, the k_i $\forall$i are the vertices of the graph, and an arc is defined using $k_1 <\bullet k_2$ to mean an arc from node k_1 to node k_2 in the DAG.

$$x_{j_1,k_1} + x_{j_2,k_2} \leq 1, \ \forall \ j_2 \leq j_1, k_1 <\bullet k_2 \qquad (2)$$

In order to write this IP problem in the exact form of Ax <= 1, A(0,1) matrix (a node packing problem, where A has at most two 1's per row) the model becomes:

$$Max \ \sum_j \sum_k x_{j,k}$$

$$x_{j_1,k} + x_{j_2,k} \leq 1 \ , \ \forall j_1, j_2, k \qquad (1')$$

$$x_{j_1,k_1} + x_{j_2,k_2} \leq 1, \ \forall \ j_2 \leq j_1, k_1 <\bullet k_2 \qquad (2')$$

The first difference is that equation (1) now becomes an inequality, ≤, instead of an equation, =, with the current cost formulation. Either equation (1) or inequality (1') are equivalent in terms of the set of integer feasible solutions. However they are not equivalent in terms of fractional space as we shall see.

More specifically consider an instance of this scheduling problem for a DAG with two tasks. The DAG has tasks a and b, where $a<\bullet b$ represents an arc from a to b in the DAG and J=5 is the total number of csteps to be considered. Constraints (1') and (2') are used to generate the following A matrix where $Ax \leq 1$ and $x^T=[x_{1,a}, x_{2,a}, x_{3,a}, x_{4,a}, x_{2,b}, x_{3,b}, x_{4,b}, x_{5,b}]$:

		row #	inequality #
A =	1 1 0 0 0 0 0 0	(1.1)	(1')
	1 0 1 0 0 0 0 0	(1.1)	
	1 0 0 1 0 0 0 0	(1.1)	
	0 1 1 0 0 0 0 0	(1.1)	
	0 1 0 1 0 0 0 0	(1.1)	
	0 0 1 1 0 0 0 0	(1.1)	
	0 0 0 0 1 1 0 0	(1.2)	
	...		
	0 1 0 0 1 0 0 0	(2.1)	(2')
	0 0 1 0 1 0 0 0	(2.2)	
	0 0 1 0 0 1 0 0	(2.3)	
	0 0 0 1 1 0 0 0	(2.4)	
	0 0 0 1 0 1 0 0	(2.5)	
	...		

This model, $Ax \leq 1$, can be visualized as the placement of tasks into control steps. The node packing graph for this problem is shown in figure 2.9. We will now describe how this graph is formed. First each variable of the IP model is represented by a node in the node packing graph. Each row generated using inequality (1') and (2') forms an edge of the node packing graph between a node $x_{j,a}$ and $x_{j,b}$. In figure 2.9b) we note that the four nodes, $x_{3,a}, x_{4,a}, x_{2,b}, x_{3,b}$, shown in bold, form a clique, or integral facet. The edges of this clique were formed by row numbers (1.6),(1.7),(2.2),(2.3),(2.4), and (2.5), of matrix A. This clique can be represented by the following inequality: $x_{3,a}+x_{4,a}+x_{2,b}+x_{3,b} \leq 1$. Therefore this inequality can be added to the A matrix. When we continue in this manner and then remove the redundant constraints we obtain the new A' matrix shown below.

		row #	inequality #
A' =	1 1 1 1 0 0 0 0	(1.1)	(1'')
	0 0 0 0 1 1 1 1	(1.2)	
	0 1 1 1 1 0 0 0	(2.1)	(2'')
	0 0 1 1 1 1 0 0	(2.2)	
	0 0 0 1 1 1 1 0	(2.3)	

This system of linear inequalities, $A'x \leq 1$, generates a smaller search space than $Ax \leq 1$. We can generalize this clique facet (precedence constraint) and represent

the new system of inequalities as:

$$\sum_{j} x_{j,k} = 1 \ , \ \forall k \qquad (1'')$$

$$\sum_{\substack{j_2 \le j+C_1-1 \\ j_2 \varepsilon R(k_2)}} x_{j_2,k_2} + \sum_{\substack{j \le j_1 \\ j_1 \varepsilon R(k_1)}} x_{j_1,k_1} \le 1 \qquad (2'')$$

$$\forall k_1 <\bullet\, k_2 \ , \ j \ \varepsilon \ R(k_2) \cap (R(k_1)+(C_1-1))) \ , k_1 \ \varepsilon \ Op(C_1,L_1)$$

A different formulation of the 1-D precedence constraint formulation was presented in [23,24] as $\sum_{j \varepsilon R(k_1)} (j) x_{j,k_1} - \sum_{j \varepsilon R(k_2)} (j) x_{j,k_2} \le -C_1, \ \forall k_1 <\bullet k_2$. We will call this constraint (2*). Even though the set of integer feasible solutions are the same, formulation (2'') is tighter (or forms a smaller search space) than (2*) and the proof is given in[31] . Thus improvements in IP solution efficiency and better bounds on variables are expected with (2'') since it is a tighter formulation[27] . On further analysis of the graph in figure 2.9, one can see that this graph is strictly characterized by cliques completely generated by (1'') and (2''). Thus G is a perfect graph (by definition) and its integral polytope is completely characterized by inequalities (1'') and (2'').

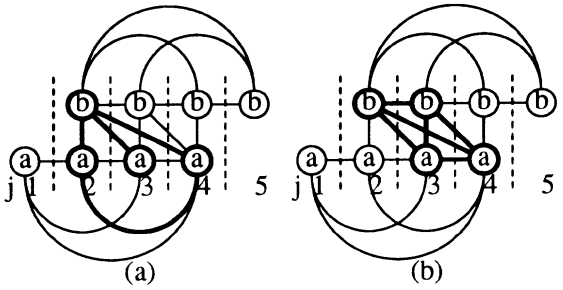


Figure 2.9. Node packing graph for 2 tasks ($a <\bullet b$), showing in bold a clique facet for j=2 in a) and j=3 in b) of inequality (2''). R(a)={1,2,3,4} and R(b)={2,3,4,5}.

2.5 THE KNAPSACK PROBLEM AND OTHER TIGHTENING TECHNIQUES

In many IPs some constraints may fall into the category of knapsack inequalities. By generating known facets of this underlying (knapsack) polytope, one can often tighten the larger polytope represented by all inequalities. This has been very successful as demonstrated by the award winning paper in[49] .

The definition and facet characterization of the knapsack problem will be given in this section.

The knapsack problem is to minimize cx over the polytope P, where

$$P=\{x \mid \sum_{j\varepsilon N} a_j x_j \leq b, 0 \leq x_j \leq 1\}, 0 \leq a_j \leq b, \forall j \varepsilon N$$

and $a_1 \geq a_2 \geq \cdots \geq a_n$. Let $\bar{x}$ be an integer vector in P. Then we can represent this fact by saying that the set $S=\{j \mid \bar{x}_j=1\}$ is independent. Now let C be a minimal dependent set. In other words we say that a x vector is dependent if it is not in P. The dependent set C is minimal if and only if C \ {i} is independent $\forall$ i ε C. The following inequality is valid for P_I : $\sum_{j\varepsilon C} x_j \leq |C|-1$. Given C, we can define E(C)= C $\cup\{k: a_k \geq a_j, \forall j \varepsilon C\}$. Now the following inequality : $\sum_{j\varepsilon E(C)} x_j \leq |C|-1$. is a facet of P_I if and only if at least one of the four conditions are true :

[1] C = N ;

[2] E(C) = N and (i) $C\setminus\{j_1,j_2\}\cup\{1\}$ is independent;

[3] C = E(C) and (ii) $C\setminus\{j_1\}\cup\{p\}, p=min\{j \mid j \varepsilon N \setminus E(C)\}$ is independent;

[4] C $\subset$ E(C) $\subset$ N and (i) and (ii).

2.5.1 EXAMPLE : RESOURCE CONSTRAINTS IN SCHEDULING

Now we will examine an example of the knapsack problem that arises in IP formulations for high level synthesis. The example is a formulation of a resource constraint, specifically bus allocation constraint. Consider the following knapsack inequality: $3\sum_{+} y_+ + \sum_{*} y_* + \sum_{*1} y_{*1} \leq 7$, Let the coefficients of y_k be a_k where k = +, * or *1. Consider the following minimal dependent set, $C \varepsilon \{+_1, +_2, *, *1\}$, $(\sum_{k\varepsilon C} x_C \leq |C|-1)$. For example summing the coefficients of the minimal dependent set we get 3+3+1+1=8 >7 (therefore dependent), and removing any one of these C \ {i} $\forall$ i ε C we get 3+3+1=7 or 3+1+1=5 $\leq$ 7(therefore independent). $y_{+_1}+y_{+_2}+y_*+y_{*1} \leq 3$ We can now prove that the following tighter inequality, where $k \varepsilon E(C)$, is a facet: $\sum_{+} y_+ + y_* + y_{*1} \leq 3$ First we prove that: (1) C\ {j_1,j_2} $\cup$ {1} is independent, ie. $a_+ + a_{*1} + a_{*2} = 3+1+1=5 \leq 7$ Secondly we must prove (2) C\ {j_1} $\cup$ {p}|p:min j $\varepsilon N \setminus E(C)$ is independent, ie. $a_{+2} + a_{*1} + a_{*2} + a_{*p} = 3+1+1+1=6 \leq 7$ This particular facet of the knapsack

inequality will be used in section 2.10 for improving the lower bound on the number of busses. Generally if the a_j's do vary in magnitude (as they do for the EWF a_i's = 1 or 3, and other types of input algorithms with pipelined or multi-cycle operations) these facets are very useful.

We can also use application-specific information to tighten inequalities. For example cut vertices of the DAG can be used in this way. An example of this vertex is a task, k_{sep}, which directly precedes and/or is directly preceded by (<•>) all other tasks in a transitivity-reduced DAG. For example consider a DAG consisting of vertices a,b,c,d, where a<• b, d <• b <• c <• e, a<• e, d <• c. A transitivity reduced DAG would be a<• b <• c <• e, d <• b. Any DAG can be transitivity reduced by applying the following rule as many times as necessary to a DAG until no further arcs are eliminated: if a <• b <• c, a <• c then delete arc a<• c. An example of a tightened inequality will be given next. First assume that we sum all variables representing tasks which may be scheduled at one time,j and set this less than or equal to 2, since we have only 2 processors (each processor can only execute on task at a time). The following inequality does this: $x_{j,k_1}+x_{j,k_2}+x_{j,k_3}+x_{j,k_4}\leq 2$. Now let us assume that k_2 is a cut vertex of the transitivity reduced DAG. If $x_{j,k_2}=1$ in this inequality then we know that all the other tasks cannot also be scheduled at this time because k_2 is a cut vertex. Therefore it would be valid to say $2x_{j,k_2}\leq 2$. Furthermore it would be valid to rewrite the inequality as $x_{j,k_1}+2x_{j,k_2}+x_{j,k_3}+x_{j,k_4}\leq 2$. The new inequality is tighter than the original inequality and thus generates a smaller search space.

2.6 PROBLEM TO BE ADDRESSED

Now we will identify the two major architectural synthesis problems which we present a model for in this chapter. An exact definition for these problems in the context of architectural synthesis for DSP systems is given below. Problem 1 has been called *simultaneous scheduling and allocation of functional units, registers, and busses*. The number of registers and busses are defined for a random topology where functional unit outputs are connected to a set of busses which also connects to inputs of registers. The registers output values on to a different a set of busses which has inputs of functional units connected to it.

> *PROBLEM 1: Produce a schedule, by mapping each code operation to a time (maintaining the partial order among operations) that minimizes a piecewise linear (area and delay) cost function of the number of functional units, registers, busses, and execution time (the total number of csteps required to execute the algorithm on the final architecture).*

Problem 2 has been called *simultaneous scheduling, selection and allocation of functional units* including chaining of operations. Unlike problem 1 this model does select a type of functional unit. This problem is very important in order to support architectures for high speed DSP systems. It is well known that the delay through two successive adders is less than two times the delay through one adder. Thus chaining operations may slightly increase the clock period but overall decrease the latency (total number of control steps) of the application significantly.

> *PROBLEM 2: Produce a schedule, by mapping each code operation to a time and a type of functional unit. A number of different functional units including those which are chained, multicycled, and pipelined are available. The problem is to optimally schedule and allocate such that area and speed requirements are met.*

As discussed in the previous section the basis of both problems is precedence constraint scheduling. Our submodel for solving this subproblem is new and we will prove its advantages over previous research in the following sections. Advances in computers providing faster and larger spaces for mathematical software has also had a great impact on this modeling area. The following algorithmic and complex constraints are also supported by the two IP models.

> *Additional Features to Support for Problems 1 and 2: The following features are to be supported: (1) Interface to analog and asynchronous processes, (2) Minimum and maximum timing constraints, (3) Conditional Code, (4) Functional Pipelining, and (5) Minimize average execution time in presence of interfaces to asynchronous processes.*

Many subtasks of architectural synthesis could not be solved to global optimums using previous algorithms, such as minimizing registers in the presence of general conditional code. Thus not only is the larger problem of simultaneous scheduling and allocation being solved for the first time but many of its subproblems can now for the first time be solved optimally by the IP approach to be presented. Furthermore as we shall demonstrate it is easy to incorporate these above features into our model, whereas it is difficult to make modifications to heuristics in previous synthesizers. We will show in section 2.10 that it is feasible to find global optimal DSP architectures with these features using our IP approach.

2.7 IP MODEL FOR PROBLEM 1

The following section will present the model which is used to solve problem 1, simultaneous scheduling and allocation, preiously defined in section 2.6. First we will define some terminology and then present the model. The following terminology will be used in this section: k = a code operation of the input algorithm (previously called a task or node of the DAG). A partial order between k_1 and k_2 is represented by $k_1 <\bullet k_2$, or in other words code operation k_1 produces (or outputs) data that is used by code operation k_2. The variables of the model are $x_{j,k}$. When $x_{j,k} = 1$, code operation k is assigned to time (cstep) j ($j \varepsilon Z$, set of integers). $j_z \varepsilon R(k_z)$ means that $asap(k_z) \leq j_z \leq alap(k_z)$ (where $asap/alap(k_z)$ is the as soon/late as possible csteps for operation k_z). The variable I_i represents the number of functional units of type i. For simplicity we will assume that $i \varepsilon op(C_z,L_z)$ which means that the type of functional unit i requires C_z csteps to produce output data and can accept new input data every L_z csteps. For example a single cycle type of functional unit is $i \varepsilon op(1,1)$ and a two-cycle pipelined type of functional unit is $i \varepsilon op(2,1)$.

In general the model consists of 5 inequalities. The operation assignment constraint, (1), ensures that each code operation (of the input algorithm) will be assigned to one cstep. The precedence constraint, (2), prevents an operation k_2 from being scheduled after operation k_1 whenever there is a partial order between these operations such that $k_2 <\bullet k_1$. The derivation of this inequality was presented in section 2.4.1. The only difference is now an operation k_1 (task) may require C_i csteps to execute (not a unit cstep). That is why the summation of the first term is different. For many cases this inequality represents a facet of the precedence scheduling problem[1] . The functional unit constraint, (3), ensures that no more than I_i functional units of type i [23, 24] will be required in the solution.

$$\sum_{j \varepsilon R(k)} x_{j,k} = 1, \forall\, k \tag{1}$$

$$\sum_{\substack{j_1 \geq j-(C_1-1) \\ j_1 \varepsilon R(k_1)}} x_{j_1,k_1} + \sum_{\substack{j \geq j_2 \\ j_2 \varepsilon R(k_2)}} x_{j_2,k_2} \leq 1 \tag{2}$$

$$\forall\, k_1 <\bullet k_2,\ j \varepsilon R(k_2) \cap (R(k_1)+C_1-1)$$

$$\sum_{\substack{k \varepsilon i \\ j \varepsilon R(k)}} \sum_{j1=j}^{j1=j+(L-1)} x_{j1,k} \le I_i \ , \ \forall \ j, \ i \ \varepsilon \ op(C,L) \qquad (3)$$

$$\sum_{k_n} \Big(\sum_{\substack{j1 \le j-(C_n-1) \\ j1 \varepsilon R(k_n)}} x_{j1,k_n} + \sum_{\substack{j2>j \\ j2 \varepsilon R(k_e) \\ k_n <\bullet k_e}} x_{j2,k_e} - \sum_{\substack{j3 \le j \\ j3 \varepsilon R(k_e) \\ k_n <\bullet k_e}} x_{j3,k_e} \qquad (4)$$

$$- \sum_{\substack{j4>j-(C_n-1) \\ j4 \varepsilon R(k_n)}} x_{j4,k_n} \Big) \le 2 \ R \ , \ \forall j, \ and \ for \ all$$

maximal sets of arcs $(k_n <\bullet k_e)$ *that cross j each with unique heads* (k_n).

$$\sum_{\substack{k \\ j \varepsilon R(k)}} (In(k)) x_{j,k} + \sum_{\substack{k_1 \varepsilon R(j1) \\ j1=j-(C_1-1)}} (Out(k_1)) x_{j1,k_1} \le B \ , \ \forall \ j \qquad (5)$$

The register allocation constraint ensures that there are no more than R variables whose lifetimes overlap at any cstep. A variable lifetime can be represented by a (lifetime-defining) edge $(k <\bullet k_l)$ between the defining operation, k, and the operation which last used the variable, k_l. However in many algorithms each variable may be input to more than one code operation $(k <\bullet k_e \mid e>1)$, thus it is difficult to determine which k_e should be the lifetime-defining edge. Two properties, transitivity and alap analysis, can be used to decrease the number of edges we must consider for representing a variable lifetime. For example in figure 2.10a), (e=1,2) transitivity requires $(k_1 <\bullet k_2) \Rightarrow (k_2 = k_l)$ and in figure 2.10b) alap analysis requires $(asap(k_2) \ge alap(k_1)) \Rightarrow (k_2 = k_l)$. This preprocessing can be done very fast and is outlined in appendix II. We will now describe how the register allocation constraint (4) can ensure no more than R registers are allocated even with multiple edges representing a variable's lifetime. The following terminology is used: (a) An arc, $k_n <\bullet k_e$ (whose head is k_n and tail is k_e), is said to *cross* cstep j if and only if $R(k_n) \cap \{0,1,...,j\} \neq 0$ and $R(k_e) \cap \{j+1, j+2,...,Te\} \neq 0$; (b) e(n) = the number of arcs $(k_n <\bullet k_e, e \ge 1)$, with head k_n that cross at j (e(n) ≤ e). For the general case where e(n) ≥ 1$\forall n$, constraint (4) is generated, $\prod_n e(n)$ times per cstep, for all maximal sets of arcs that cross j such that no two arcs in a set have the same head. For example if only one head (k_i) has e multiple arcs $(k_i <\bullet k_j, j=1,...,e.)$ that cross at j $(e(i)=e)$, and the rest of the arcs have unique

heads ($e(n)=1 \forall n \neq i$), then (4) is generated e times (once for each k_j). In practise the number of constraints will not be a significant problem, because 1) the computation time for IP problems is not highly sensitive to the number of constraints [27] and 2) most algorithms will have small values of $\prod_n e(n)$ which intersect at the same cstep. The register allocation constraint (4), calculates two times the number of cut edges at each cstep, by dividing time and operations into four quadrants as shown in figure 2.11.

Since we are interested in obtaining an exact measure of the number of busses of a final architecture, we define the number of parallel data transfers (*pdt*) as the maximum number of data transfers that occur at one time (counting transfers with distinct sources as the number of destinations) unlike previous[18, 26] definitions. We constrain each hardware unit (register or functional unit) to have only one bus per input, unlike other heuristics which cannot estimate additional multiplexors required later in the synthesis process. Data broadcasts can be modeled using fixed timing constraints[56] on all pairs of destination operations and selecting one of the destination operations to contribute to the bus count.

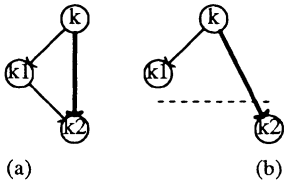


Figure 2.10. Reduction of the number of arcs to be considered for representation of a lifetime of a variable is illustrated above.

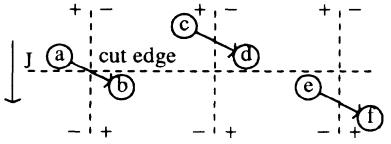


Figure 2.11. Illustration of the register allocation constraint generated for cstep J. This constraint assigns coefficients of +1,-1 to variables according to their location in one of four quadrants. The vertical line partitions tails of arcs from heads of arcs. The csteps are increasing from top to bottom.

The bus allocation constraint, (5), ensures that at each cstep no more than B busses are required to transfer data between functional units and registers. We also use the constraint $(In(i)+Out(i))I_i \leq B$, $\forall i \varepsilon op(1,1)$ to decrease the size of the search space. The defined number of parallel data transfers is exactly equal to the number of busses in an optimal architectural solution 1) for module allocation, 2) allocation of at most 2 types of functional units and in other cases. For example assume we have three (single cycled) functional units of different types (i=3) and all pairs of functional units are scheduled in parallel (but all 3 functional units are never scheduled at once). The *pdt* would be calculated as (2*2+2=) 6, but (3*2+2=) 8 busses are required since each functional unit must have only one bus per input and each functional unit in theory must be able to access any output bus. The *pdt* is exact when not all pairs are scheduled at the same time or when all 3 are scheduled at least once in parallel. Nevertheless since many DSP algorithms have only two types of functional units and in many practical applications the functional units have high utilization, the *pdt* will often be exactly equal to the number of busses for $i \geq 3$. Proof that an architectural solution with B busses, R registers, and I_i functional units of type i ($i \leq 2$) is always guaranteed to exist can be found in[1, 57] . For the first time this provides us with an exact defined relationship between parallel data transfers and the number of busses required in the architecture.

2.7.1 SOLUTION OF PROBLEM 1 USING IP APPROACH

Before we can solve the IP model we need to formulate a cost function to be minimized or maximized. The most logical formulation in DSP synthesis is to minimize the execution time with a constraint on the area or to minimize the area with a constraint on the execution time. Alternatively one could set the area as a constraint and then minimize the execution time in the first optimization phase. Once a minimum execution time is found one then can set this as a constraint and minmize the area in the second optimization phase. This two phase optimization methodology can synthesize very good architectures that not only meet but exceed the application performance requirements. We will first discuss formulation of different cost functions and then proceed to outline the procedure which must be followed in order to solve the IP.

Piecewise linear area cost functions such as: $\sum_i c\_fu(i)\, I_i + c\_bus\, B + c\_reg\, R$ can be formulated. For example assume we have a cost of 10 for the first 5 registers and a cost of 15 for the additional registers[26] after the fifth. The right hand side of the register allocation inequality, (4), becomes $\leq 2(R_1+R_2)$, $R_1^{UB}=5$ and $R_2^{LB}=0$, and part of the cost function becomes $(10\ 15)(R_1\ R_2)^T = c\_reg\ R$. Alternatively the minimization of

execution time can be formulated as follows: Minimize $T_e=\sum_j (j-1)x_{j,k_{end}}$. In this case a special node is added to the DAG, such that all other operations must precede this node, $k_i < \bullet k_{end}, \forall i$. This operation , k_{end}, is used to determine how many csteps it takes to execute the algorithm on the synthesized architecture. Thus the following two approaches can be used first to minimize execution time given constraint on area, and second to minimize area given constraint on execution time. In the later case the constraint on execution time is formulated by setting the as late as possible cstep which the end operation can be assigned to as Te + 1. All other operations are assigned as late as possible values accordingly. In summary the set of cost formulations (6) and corresponding constraints are shown below.

$$\begin{aligned} &Minimize \ \sum_j (j-1)x_{j,k_{end}} \\ &\sum_i c\_fu(i)\, I_i + c\_bus\ B + c\_reg\ R \le Area \\ &Minimize \ \sum_i c\_fu(i)\, I_i + c\_bus\ B + c\_reg\ R \\ &alap(kend) = Te+1 \end{aligned} \tag{6}$$

Before one solves the integer programming problem it is essential to obtain lower bounds on all variables. We use inequalities (1), (2) and $0 \le x_{j,k} \le 1, \forall j,k$ with (3),(4), or (5) and cost functions I_i, R, *or* B to calculate lower bounds for functional units, registers, or busses respectively. Furthermore even upper bounds on the cost or other variables should be set if they are known or found from a previous search. For example if we have found an architecture with area A1 and execution time T1 and want to investigate another architecture with execution time T2 > T1, then the upper bound on the new area A2 is A1, since we are only interested in architectures with smaller area than our current solution.

2.8 IP MODEL FOR PROBLEM 2

The IP model for solving problem 2, simultaneous scheduling, and selection and allocation of functional units, including chaining of operations will be presented next. The following terminology in addition to that used in section 2.7, will be used in this section to describe the model for problem 2. The variables of the model are $x_{j,k,t}$. When $x_{j,k,t} = 1$, code operation k is assigned to time (cstep) j ($j \varepsilon Z$, set of integers) and operation k is assigned to functional unit type t ($t \varepsilon Z$). $t_z \varepsilon T(k_z)$ means that code operation k_z can be implemented by functional unit type t_z. For example a multiplication operation, k, can be implemented by t=1 a two-cycle multiplier and t=2 a pipelined multipler,

$T(k)=\{1,2\}$ (further details will be provided later in the paper concerning the use of t for chained operations). For simplicity we will assume that $t_z \in op(C_z,L_z)$ which means that the type of functional unit t_z requires C_z csteps to produce output data and can accept new input data every L_z csteps. For example a single cycle type of functional unit is $t \varepsilon op(1,1)$ and a two-cycle pipelined type of functional unit is $t \varepsilon op(2,1)$. We use the notation $time(k_1,k_2) \leq or \geq or = T$ to represent $(\sum_j x_{j,k_2} - \sum_j x_{j,k_1} \leq or \geq or = T)$ the maximum, minimum or fixed time constraint between the two operations. Note that $time(k_1,k_2) \leq T$ is equivalent to $time(k_2,k_1) \geq -T$.

The IP model for solving problem 2 will now be presented. The inequalities which comprise the model are shown beside the boxes below. The code operation assignment constraint, (7) , ensures that each code operation of the input algorithm will be assigned to one cstep and one type of functional unit. The type of functional unit constraint, (8) calculates the number of functional units of a particular type to be allocated.

$$\sum_{t \varepsilon T(k)} \sum_{\substack{j \varepsilon R(k) \\ j1=j+(L_1-1)}} x_{j,k,t} = 1, \ \forall k \tag{7}$$

$$\sum_{j1=j} \sum_{\substack{k_1 \\ t_1 \varepsilon T(k_1)}} x_{j1,k_1,t_1} \leq I_{t_1}, \ \forall j, t_1 \tag{8}$$

Next we will discuss the precedence constraints required to model different types of functional units. These constraints are more complex than our previous model because now an operation may be assigned to more than one type of functional unit. Thus depending upon which functional unit it is assigned to, it will have different precedence constraints. The general data precedence constraint, (9), where k_1 is a single cycle or multicycle operation or pipelined operation (not chained), prevents an operation k_1 from being scheduled after operation k_2 whenever $k_1 <\bullet k_2$ and t_1 is not a chained type of functional unit. The precedence constraint is equivalent to the minimum timing constraint $time(k_1,k_2) \geq C_1$, $t_1 \varepsilon T(k_1)$.

$$\sum_{t_1 \varepsilon T(k_1)} \sum_{j_1 \varepsilon R(k_1),\ j-(C_1-1) \leq j_1} x_{j_1,k_1,t_1} + \tag{9}$$
$$\sum_{t_2 \varepsilon T(k_2)} \sum_{\substack{j_2 \leq j \\ j_2 \varepsilon R(k_2)}} x_{j_2,k_2,t_2} \leq 1, \ \forall k_1 <\bullet k_2, \ j \varepsilon R(k_2) \cap (R(k_1)+C_1-1)$$

We will now address the inequalities necessary to support chaining of operations. Consider a single cycle chained type of functional unit, called an adder-adder (aa), composed of two successive additions $+_1<\bullet+_2$. The variables which represent selection of the adder-adder and operation scheduling, are $x_{j,+_1,t_1}$ and $x_{j,+_2,t_2}$, where $t_1=t_{aa1}$ and $t_2=t_{aa2}$ represent the first and the second addition operations in the adder-adder respectively. The following constraint ensures that the adder-adder will execute in one cstep, time$(+_1,+_2)=0$ $\forall$ $+_1<\bullet+_2$, $t_1=t_{aa1}$, $t_2=t_{aa2}$. This (fixed) timing constraint is represented in the IP model as time$(+_1,+_2)\geq 0$ and time$(+_1,+_2)\leq 0$ $\forall$ $t_1=t_{aa1}$, $t_2=t_{aa2}$ along with the following inequalities $x_{j_1,+_1,t_{aa1}}+\sum_{j\neq j_1,\ j\in R(+_2)} x_{j,+_2,t_{aa2}}\leq 1$, $x_{j_1,+_2,t_{aa2}}+\sum_{j\neq j_1,\ j\in R(+_1)} x_{j,+_1,t_{aa1}}\leq 1$ $\forall$ j_1, $+_1<\bullet+_2$. However each addition operation may also be implemented with a one-cycle adder type of functional unit. Therefore the remaining constraints are time$(k_1,+_2)\geq C_1$, $\forall$ $k_1<\bullet+_2$, $t_1\neq t_{aa1}$, and time$(+_1,k_2)\geq C_1$, $\forall$ $+_1<\bullet k_2$, $t_2\neq t_{aa2}$.

Another example of a chained functional unit, called a multiplier-adder (ma), is a multiplication chained with an addition, $*_1<\bullet+_2$, $t_1=t_2=t_{ma}$ (for multiplier-adder type), where $C_{ma}=2$. Thus the following timing constraints are used time$(*_1,+_2)\geq 1$ $\forall$ $*_1<\bullet+_2$, $t_1=t_2=t_{ma}$, time$(*_1,k_2)\geq C_1$ $\forall$ $*_1<\bullet k_2$, $t_2\neq t_{ma}$ (this includes $t_1=t_{ma}$), and time$(k_1,+_2)\geq C_1$ $\forall$ $k_1<\bullet+_2$, $t_1\neq t_{ma}$ *or* $t_2\neq t_{ma}$. The IP approach supports chaining of two or more operations to define any type of functional unit. Unlike previous research, no preprocessing of operations is required in order to simultaneously select chained types of functional units. Partial subset of operations can also be mapped into a larger complex operation. The fact that we have shown the data precedence constraint to be facet-generating (for a subset of the problem[31]) and therefore very tight, is important not only since it forms the basis of our model, but helps to explain why even with complex timing constraints we can still practically and efficiently solve many IP synthesis problems in very good execution times. Register allocation constraints for the new model are obtained by substituting $\sum_{t\in T(k)} x_{j,k,t}$ for $x_{j,k}$ in the constraint (4) of section 2.7.

2.8.1 SOLUTION OF PROBLEM 2 USING IP APPROACH

One can support the same cost functions as described in section 2.7.1. However in addition one can use the fact that some adders will be chained, thus requiring less interconnect than those adders that are not chained. Or perhaps

the single adders have a smaller area than the chained adders. Thus one can separately formulate a model that counts the number of single adders used and the number of chained adders used as shown below.

$$Minimize \quad 2area(chainedadder)I_{aa1} + area(adder)I_{aa2}$$

$$\sum_{kadd} x_{j,kadd,1} \leq I_{aa1}$$

$$\sum_{kadd} x_{j,kadd,2} - \sum_{kadd} x_{j,kadd,1} \leq I_{aa2}$$

Alternatively let us assume that all the adders had the same area and we multiplexed both adder inputs of the adder-adder (so that it could be functionally equivalent to two separate adders). Then the cost function and additional constraint would be:

$$Minimize \quad area(add)I_{add}$$

$$\sum_{kadd\,at} \sum x_{j,kadd,at} \leq I_{add}$$

The same preprocessing is required as discussed in section 2.7.1 before the integer programming problem can be solved. In addition to this, operations in the application which can not be a certain type (due to types of operations which precede or succeed it) are identified and set equal to zero. For example an addition operation which outputs data only to multiplication operations can never be assigned to a type aa1 as previously defined. Therefore we can set $x_{j,add,1} = 0 \; \forall j \varepsilon R(add)$. The execution time and area (with the above exception) can be modeled the same way as in section 2.7. Again lower bounds are computed as described in section 2.7 before the IP problem is solved.

2.9 EXTENSIONS TO IP MODELS: PIPELINING,MUTUAL EXCLUSION,INTERFACES

Our IP model can easily support conditional code for IP model 1 and 2. In model 2 the term $\sum_t x_{j,k,t}$ is substituted for $x_{j,k}$ in the description that follows. Inequality (2) is generated for each set of mutually exclusive code operations or code operations from each possible path generated by conditional branches. Similarly the register allocation constraint (4) is generated from arcs which cross (whose head is executed before the branch and tail is executed after the join), or have a head and/or tail in each path generated by conditional branches. Also extra data precedence constraints may need to be added to prevent conditional code operations from being scheduled before the branch or after the join of the branch.

Fixed, minimum or maximum timing constraints between pairs (or groups) of operations can be incorporated into our IP model. In all examples below, T is the time constraint value measured in number of control steps. We use the notation time(k_1, k_2) $\geq \leq =$ T to represent the minimum $\geq$, maximum $\leq$, or fixed $=$ time constraint between the two operations, where time(k_1,k_2) $= \sum_i (\sum_{j \varepsilon R(k2)} j\, x_{i,j,k2} - \sum_{j \varepsilon R(k1)} j\, x_{i,j,k1})$.

A fixed timing constraint between two operations, k_1 and k_2, forces the scheduled time for operation k_2 to be T control steps after operation k_1 or in other words $\sum_i x_{k_1,j} = \sum_i x_{k_2,(j+T)}, \forall j$. An equivalent representation of this fixed timing constraint using node packing inequalities is shown below: $\sum_i (x_{i,k_1,j_1} + \sum_{\substack{j \neq j_1 \\ j \varepsilon R(k_2)}} x_{i,k_2,j}) \leq 1, \sum_i (x_{i,k_2,j_1} + \sum_{\substack{j \neq j_1 \\ j \varepsilon R(k_1)}} x_{i,k_1,j}) \leq 1, \quad \forall\, j_1,\ time(k_1,k_2)=T.$

The fixed timing constraint is very important for interfacing to analog processes which may input or output synchronous sequential data at a fixed rate, or at a fixed number of control step intervals (T). In addition the minimum and maximum timing constraints of T, which will be presented next, also form facets of the fixed timing constrained scheduling problem.

The minimum and maximum timing constraints are represented by the two inequalities shown below:
$\sum_i (\sum_{\substack{j > j_1 \\ j \varepsilon R(k_1)}} x_{i,k_1,j} + \sum_{\substack{j \leq j_1 + T \\ j \varepsilon R(k_2)}} x_{i,k_2,j}) \leq 1,\ \forall j_1,\ time(k_1,k_2) \geq T.$
$\sum_i (\sum_{\substack{j < j_1 \\ j \varepsilon R(k_1)}} x_{i,k_1,j} + \sum_{\substack{j \geq j_1 + T \\ j \varepsilon R(k_2)}} x_{i,k_2,j}) \leq 1,\ \forall j_1, time(k_1,k_2) \leq T.$ Both these constraints are similar to the data precedence constraints in inequalities (2), except they are extended in a direction by T (where $C_1 - 1 = T$).

Functional pipelining for a fixed initiation rate (defined as the time between successive starts of an input algorithm), l, can be incorporated into our model without additional variables. We define $\lceil J/l \rceil = p$ pipestages, and replace $\sum_i \sum_{\substack{k \\ j \varepsilon R(k)}} x_{i,j,k}$ of (3,4,5) with $\sum_i \sum_{n=1}^{n=p} \sum_{\substack{k \\ (j+nl) \varepsilon R(k)}} x_{i,j+nl,k}$ where addition $(j+nl)$ is modulo l. Thus only variables representing code operations of one pipestage are used. The functional unit allocation and register allocation constraints are generated for l consecutive control steps. If the number of pipestages is less than p

(defined above), then the model must generate these constraints for all j such that $(J - p\ l) \geq j \geq p\ l$. Functional pipelining maintains the node packing structure of the inequalities.

Constraints which ensure that data (output from operation k_{out}) is valid in a register for a least T csteps can be formulated using a dummy operation (k_d) and the constraint time(k_{out},k_d)$\geq T$, where $k_{out}<\bullet k_d$ forms the lifetime of the variable which is included in the modified register allocation constraint. For example this may be important to ensure that output data is valid for access by an external process, that runs at a slower clock rate. Alternatively if input data, arriving from an external process (k_{ext}), is only valid in an input register (or at an input port) for T csteps (after which point it may be overwritten), then constraint time(k_{ext},k_d)$\leq T\ \forall\ k_d$ is used. Interface with analog processes can be formulated with the fixed timing constraints described in the previous section. A disjoint timing constraint is used whenever an operation cannot be scheduled during an interval of time, but it can be scheduled before or after this interval. For example an operation which accesses data from a shared bus that is being used by another process for a large data transfer, would constraint the operation not to be scheduled during this period. Disjoint timing constraints can easily be modeled as : $\sum_t (x_{j,a,t} + \sum_{j+\alpha>j1>j+\beta} x_{j1,b,t}) \leq 1$, $\sum_t (x_{j,b,t} + \sum_{j-\alpha>j1>j-\beta} x_{j1,a,t}) \leq 1$ $\forall\ j$, $time(a,b) \geq \alpha$ *or* $time(a,b) \leq beta$, $\alpha > beta$.

A new approach is presented for modeling the asynchronous interface that minimizes an average execution time, $\overline{T_e}$. We assume that each possible arrival time for input data has an associated probability, obtained from previous simulation studies. For example the probability of data being located in cache (the hit rate) could be 0.85. By modeling only selected high probability arrival times using a mutually exclusive conditional code approach, the average execution time of the input algorithm can be optimized. For example the cost function is *Minimize* $\overline{T_{end}}$ where $\overline{T_{end}} = \sum_i P_i T_{end_i}$, $T_{end_i} = \sum_t \sum_{j \varepsilon R(end_i)} (j-1) x_{j,end_i,t}$ (the execution time assuming data arrives during the ith selected time) and P_i is the probability that the input data arrives at the ith selected time. The code operation end_i marks the completion of the application assuming data from the asynchronous operation arrives at time i. However it is possible that the input data will arrive at a (lower probability) time that was not selected. In these cases the data must be stored in a register until the controller reaches the next selected time at which point it can process the data. The storage of the input data can be modeled using dummy operations and minimum and maximum timing constraints as described previously between selected arrival times. The IP model

for asynchronous interfaces is treated the same way as conditional code. Multiple interfaces to asynchronous operations, even ones that overlap in time, can also be easily modeled. Infinitely bounded asynchronous interfaces are modeled by selecting high probability arrival times and adding an edge from the last selected arrival time to the earliest cstep where all operations are interface dependent.

2.10 SYNTHESIS RESULTS

Architectures for DSP applications, such as the elliptical wave filter (EWF) and the discrete cosine transform (DCT), were synthesized. The EWF has 34 code operations and over 56 arcs in the DAG and the DCT has 42 code operations and over 52 arcs in the DAG. The previously researched EWF was compared with simulated annealing[26] , HAL[18] , and SAW[58] solutions. In the EWF, functional unit and bus allocation constraints were tightened where possible with k_{sep} (equal to the fourth adder, from the top, in the critical path of figure 2.12) and k_{out}. Lifetime defining edges for all variables except two (which required 24 extra register allocation constraints), were found using the transitivity and alap preprocessing. The *, *pl, -, and + refer to the number of two cycle multipliers, pipelined multipliers, single cycle subtractors, and single cycle adders respectively. The IP problems were solved using the branch and bound solver GAMS/ZOOM[42] on a MIPS RC2680.

Table 2.1 shows the results of simultaneous scheduling and allocation of functional units for the DCT example. This example was solved using the two phase methodology outlined in section 2.7.1. The DCT example was quoted in [59] as being too large for an IP approach to synthesis. Results on cpu time in table 2.1 show that our IP approach can solve for many different architectures in under 9 cpu minutes. In phase I the area was minimized given a upper bound on execution time (Te) and fixed initiation rate (l or the number of csteps in between successive initiations of the algorithm). The phase II optimization pass, shown in table 2.2, minimized the execution time (or delay) for the fixed initiation rate, l, and fixed area (value equal to minimized area value of phase I optimization pass). For example with 7 adders, 7 subtractors, 8 multipliers, and l=2 (see row 3 of table 2.1), the execution time was minimized to 9 csteps. The IP problem took only 1.26 cpu seconds to solve and had 358 variables and 482 equations. Each phase II IP approach row in table 2.2 required less than 50 cpu seconds to solve. Unfortunately there were no cpu times quoted for the PLS algorithm [59] . Each architecture for the DCT (represented by a row in table 2.2) required less than 100 cpu seconds in total including phase I and II (except for the last row of table 2.1 which required 471 seconds).

Table 2.1. DCT Synthesis Results (Phase I: Minimize Area)

l	Te	+	-	*	Var	Eqn	CPU (sec)
2	12	7	7	8	355	458	0.24
3	12	5	5	6	355	461	0.45
4	15	4	4	4	397	523	62.3
8	15	2	2	2	397	535	175
12	12	2	2	2	355	497	12.8
13	13	1	1	2	355	497	471

Table 2.2. Comparison of DCT Synthesis Results (Phase II: Minimize Te)

Synthesizer	l	Te	+	-	*
IP approach	**2**	**9**	**7**	**7**	**8**
IP approach	**3**	**9**	**5**	**5**	**6**
PLS[59]	3	10	5	5	6
IP approach	**4**	**10**	**4**	**4**	**4**
PLS[59]	4	11	4	4	4
IP approach	**8**	**11**	**2**	**2**	**2**
PLS[59]	8	12	2	2	2
IP approach	**13**	**13**	**1**	**1**	**2**
PLS[59]	13	16	1	1	2

Two solution techniques, LB and KP, were demonstrated with the IP approach[60] and are shown in table 2.3. The first method, LB, calculates lower bounds on all variables and branch and bounds to obtain a solution. The second method, KP, additionally uses knapsack inequalities to improve the bound on the number of busses before the branch and bound. Table 2.3 shows a comparison of the EWF synthesized solutions with previous research. From the 17 cycle simulated annealing schedule given in [26] , although not specified the eighth row requires 11 busses and the IP solution requires 10 busses. The 17 and 18 cycle IP solutions given in table 2.3 required 0.5 cpu minutes and 3 cpu minutes respectively (using the LB method) where after branch and bounding on I_i,R,B variables, the initial LP provided all $x_{j,k}$ integer solutions. These cpu times are faster than the 2 cpu minutes and 4 cpu minutes respectively quoted by HAL[18] and simulated annealing[26]. HALs EWF synthesis for 19 csteps[18] requires 8 busses and 12 registers (in 6 cpu min), unlike the optimal IP approach with 7 busses and 9 registers. This architecture was synthesized in less than 6 cpu minutes This cpu time included calculation of lower bounds and IP solution for a globally optimal solution.

Table 2.3. EWF Synthesized Architecture Comparisons

Synthesizer	Te	*pl	*	+	R	B	Total CPU minutes LB	KP
IP approach	**21**		**1**	**2**	**9**	**7**	**30**	**6**
SAW[58]	19		2	2	11	9	na	
HAL[18]†	19	1		2	12	8	6	
IP approach	**19**	**1**		**2**	**9**	**7**	**5.8**	**-**
HAL[18]	18	1		3	12	na	4	
IP approach	**18**	**1**		**3**	**10**	**9**	**3**	**-**
IP approach	**18**		**2**	**2**	**10**	**8**	**3**	**0.5**
HAL[18]	17	2		3	12	na	2	
IP approach	**17**	**2**		**3**	**10**	**10**	**0.5**	**-**
IP approach	**17**		**3**	**3**	**10**	**10**	**0.5**	**-**

$$Minimize \quad 50\,I_{+} + 250\,I_{*} + 15\,R + 100\,B + 50\,T_{e} \quad (12)$$

†HAL 6 busses + 2 local busses[18];SAW page 79 in [58];-/na= not applicable
R does not include IN and OUT of EWF, CPUmin. for PC386

Table 2.4. Comparison of CPU Seconds for EWF

Synth	# Code Operations	Te	*pl	+	Var	Eqn	CPU (sec)
IP approach	34	19	1	2	130	160	36
BAKER	34	19	1	2	130	120	600†
IP approach	102	50	1	3	310	407	40

† branch and bound is required. CPU sec on PC386

The area-delay (12) optimized solution for two cycled multipliers (with an upper bound of 18 cycles) is shown in row seven of table 2.3. The LB method solves for a globally optimal schedule and allocation in less than 3 cpu minutes total. By fixing B=7, we were able to extract knapsack facets of this constraint and use it to show (via the infeasible LP) that the bound of 7 busses could be improved. The bus allocation constraint[60] with B=7 is $3\sum_{+} x_{j,+} + \sum_{*} x_{j,*} + \sum_{*} x_{(j-1),*} \leq 7,\ \forall j$. The generalization of one facet of this knapsack inequality is $\sum_{\substack{+ \\ j\varepsilon R(+)}} x_{j,+} + x_{j,*} + x_{(j-1),*^{-1}} \leq 3, \quad \forall *, *^{-1}, \quad j\varepsilon R(*), \quad j-1\varepsilon R(*^{-1})$. Using these knapsack inequalities it required 9 cpu seconds to determine that the new LP was infeasible. We therefore improved the lower bound from 7 to 8 busses and solved the IP problem (using branch and bound) to obtain an all integer solution in a total of 24 cpu sec (0.5 cpu min in total in table

2.3). Other knapsack inequalities were used for the EWF example with Te=21 csteps, whose IP model had 202 variables and 308 constraints. The initial lower bounds were I_{+}=2,I_{*}=1, R=9, B=6. Generalized knapsack inequalities extracted from the bus allocation constraint, were used to created the new LP (with B=6) which was infeasible. So by setting B^{LB} = 7 we could branch and bound to a completely integer solution in 6 minutes total. Without using the knapsack inequalities (ie. B^{LB} =6) we required 30 minutes of branch and bound (see table 2.3) to find the same globally optimal solution. Both examples illustrate the advantages of using knapsack facets for efficiently solving IP problems.

Table 2.4 shows the total cpu seconds required by the IP model for minimizing the area cost function in (12) for functional units alone. In row one the relaxed LP with (integer rounded) lower bounds produced integer solutions in 36 cpu seconds (for model generation and LP execution). We also ran this same instance of the EWF problem using the precedence constraint (2*) of [23,24,61] which required branch and bound [42] to find an integer solution in approximately 10 cpu minutes (see second row, BAKER, in table 2.4). In both cases solutions are globally optimal for this cost function and the same IP solver[42] was used. Row three illustrates how efficiently we can simultaneously schedule and allocate large algorithms such as the EWF which was unrolled three times creating 102 input code operations. Over 300 $x_{j,k}$ variables were solved to integer values in the initial LP in less than one cpu minute. These results illustrate how important good bounds and tight models are for solving integer programs.

To illustrate the IP architectural synthesizer for asynchronous interfaces we have introduced communication with an asynchronous external operation into the EWF example[62] . The asynchronous operation, a, which is external to the synthesized chip, receives input data from the bold addition operation in figure 2.12 (+<•a). This operation must also return data to the filter at some indeterminate cstep (a<•*). The asynchronous operation can at the earliest produce data after two cycles and at the lastest after 4 cycles. An area constraint of 2 adders and 1 two-cycle multiplier was used and the average execution time was minimized. The asynchronous operation, a, can at the earliest produce data after two cycles (with probability 0.6) and at the lastest after 4 cycles (with probability 0.4). The optimal schedule, shown in figure 2.12, used 425 variables, 610 equations, and was synthesized in 190 cpu seconds.

Alternatively the IP approach can minimize chip area with constraints on the average execution time. The area cost function $50I_{t_+} + 250I_{t_*} + 10R + 10B$ (R and B is the number of registers and busses) was minimized. The constraint on the average execution time was T_{e_1}=21 and T_{e_3}=26. The IP approach

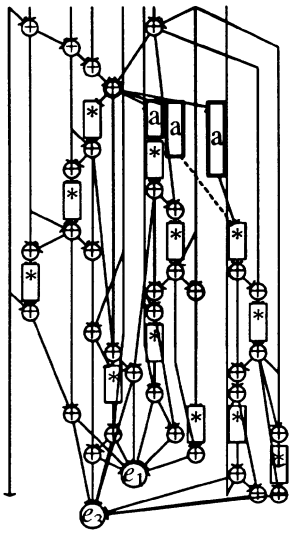


Figure 2.12. Optimized schedule and allocation, with asynchronous interface to a, which minimizes the average execution time (23.8) for P_1=0.6,P_3=0.4. The dashed line represents input buffer allocation for data arriving at the lower probability time.

required 254 variables, 592 inequalities, and synthesized a globally optimal architectures in 8.3 cpu seconds. To illustrate the impact of the model formulation on the efficiency of solving the IP, the data precedence constraints were replaced by a previously researched constraint[23, 61] . This modified IP model had the same number of variables, 469 inequalities, and required 274 cpu seconds to solve (using the same IP solver on the same cpu).

The EWF was also used to demonstrate how IP approach can support automatic selection of chained operations. The following types of functional units were allowed: two-cycle multipliers, two-cycle multiplier-adders, one-cycle adders, and a one-cycle adder-adders. IP approach minimized the area of the architecture by selecting and allocating functional units simultaneously with scheduling operations, given an upper bound of 14 csteps on the total execution time. Figure 2.13 shows the IP approach solution requiring one adder-adder and two multiplier-adders. This architecture was globally optimized in 13.6 cpu seconds, requiring 202 variables and 447 inequalities. Note that the single addition operations shown in figure 2.13 can be implemented with the second

addition in the adder-adder or with the addition in one of the multiplier-adders.

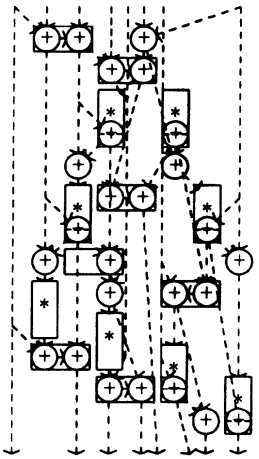


Figure 2.13. Chaining of operations to optimize speed and minimize execution time. Two multiplier-adders and one adder-adder type of functional unit were automatically selected simultaneously with scheduling and allocation.

Figure 2.14 illustrates the area-delay curve for the EWF obtained by examining a different number of clock periods which allowed chaining of operations, and single cycle and multicycle operations. A multiplier with 49Kmil^2 area and delay 375ns, and an adder with 1200mil^2 area and delay 151ns was selected from a library (along with a 5ns delay register). Clock period 1 was chosen to be 156ns so that the multiplier required 3 cycles and the adder required one cycle. Clock period 2 is 190ns so the multiplier requires only two cycles latency. Clock period 3 is 265ns so that two adders can be chained together and perform two additions in one cstep. Also a multiplication followed by an addition operation can be performed in two clock periods. Each point on the area-delay curve required less than 10 cpu seconds to obtain using the IP approach[63] .

2.11 DISCUSSION AND CONCLUSIONS

A DSP methodology based on IP approach to architectural synthesis has been presented. Using polyhedral theory, we have shown that optimal solutions to the simultaneous scheduling and allocation problem can be solved in cpu execution times faster or equivalent to previous research which use heuristic

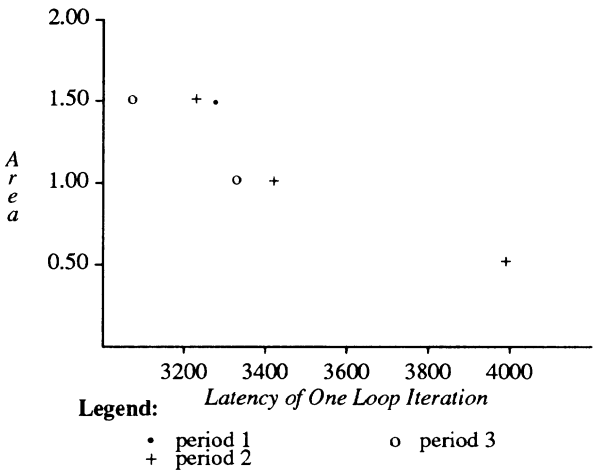


Figure 2.14. The Area vs. Delay curve for the EWF example.

techniques. Furthermore we have also shown that previous heuristics have not obtained the globally optimal architectures that are synthesized using the IP approach. Many other examples have been synthesized with the IP model[1] .

The worst case complexity for the IP problem in theory is exponential. This gives a poor representation of the expected complexity, since it means that there will be some problems which will take a long time to solve. However similar to previous research[27] we have found that most problems are solved very fast and it is expected that the majority will exhibit this behavior. Although we cannot guarantee 0-1 solutions, (the problem is NP-complete) most examples we have optimized provide 0-1 solutions. For the first time our IP model provides tight bounds on the architectural synthesis problem which is extremely important for any synthesis solution technique, including simulated annealing or the use of techniques described in this chapter.

A second approach for dealing with complexity relies upon the designer to select subsets of the model to explore the design possibilities. For example one can first minimize the number of functional units and then secondly minimize the number of registers to explore the architectural design space. There also exist various input algorithm partitioning strategies that have been researched to deal with complexity such as vertical partitioning in [64] or partitioning and pipelining of the algorithm as demonstrated with the matrix multiplication example in this chapter. However more importantly we have demonstrated that over 100

code operations (EWF, table 2.4) can be simultaneously scheduled and allocated in very fast cpu times. Large examples where on average each code operation can be scheduled in any one of 9 possible csteps (DCT, table 2.1 and 2.2) have been solved in practical cpu times. This ability to optimally synthesize large complex algorithms is a significant contribution to the synthesis field. The two phase optimization method has created architectures for DSP applications that are up to 23% faster than previous architectures (see table 2.2).

Currently we have extended our model to include multichip synthesis[65, 66] and retiming[67] . In the future we intend to extend the IP model for simultaneous scheduling, allocation and binding (to minimize the number of multiplexors and the number of inputs to multiplexors). We also plan to conduct further research on the extraction of more facets and the use of this model for interfacing to analog signal processing domains. The IP model presented in this chapter is a significant contribution to the field of high level architectural synthesis. For the first time we have proof that the synthesized architectures are globally optimal. This is important for industry since the early decisions made during architectural exploration have the greatest impact on the final design. Previous synthesizers could at best guarantee a locally optimal synthesized solution, which may not meet design constraints. Finally we have demonstrated that the IP architectural synthesizer can handle input DSP applications with different types of structure, with over 100 code operations and with complex constraints. In summary this research guarantees globally optimal synthesized solutions, synthesizes large input applications in practical execution times, and supports complex constraints and cost functions.

References

1. C.H. Gebotys, "A Global Optimization Approach to Architectural Synthesis of VLSI Digital Synchronous Systems With Analog and Asynchronous Interfaces," *Dept of Electrical & Computer Engineering, Univ of Waterloo, Waterloo Ont, Canada, PhD Thesis*, (July 1991).

2. B. Haroun and M. Elmasry, "Architectural Synthesis for DSP Silicon Compilers," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, **CAD-8**(4)(April 1989).

3. M. McFarland, A. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," *Design Automation Conference*, pp. 330-336 (1988).

4. M. McFarland, A. Parker, and R. Camposano, "The High Level Synthesis of Digital Systems," *Proceedings of IEEE*, **78** pp. 301-318 (1990).

5. L.R. Foulds, *Optimization Techniques: An Introduction*, Springer-Verlag (1981).

6. C. Tseng and D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Transactions on Computer-Aided Design*, pp. 379-395 (1986).

7. M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press (1980).

8. A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignments with Large Apertures," *Proc. 8th Design Automation Workshop*, pp. 155-169 (1971).

9. D.L. Springer and D.E. Thomas, "Exploiting the Special Structure of Conflict and Compatibility Graphs in High Level Synthesis," *Int'l Conf on Computed Aided Design*, pp. 254-257 (1990).

10. F.J. Kurdahi and A.C. Parker, "REAL: A Program for Register allocation," *Design Automation Conference*, pp. 210-215 (1987).

11. P. Marwedel, "A new Synthesis Algorithm for the MIMOLA software system," *Design Automation Conference*, pp. 271-277 (1986).

12. L. Stok, "Interconnect Optimization During Data Path Allocation," *Workshop on High Level Synthesis*, (1989).

13. Garey and Johnson, *Computers and Intractability*, Freeman and Co. (1979).

14. E.G. Coffman, *Computer and Job Shop Scheduling Theory*, J.Wiley and Sons (1976).

15. J.D. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and Systems Science*, **10** pp. 384-393 (1975).

16. P. Brucker, M.R. Garey, and D.S. Johnson, "Scheduling Equal-Length Tasks Under Treelike Precedence Constraints to Minimize Maximum Lateness," *Mathematics of Operations Research*, **2**(3)(August 1977).

17. B.M. Pangrle and D.D. Gajski, "Design Tools for Intelligent Silicon Compilation," *IEEE Transactions on Computer-Aided Design*, **CAD-6**(6) pp. 1098-1112 (1987).

18. P.G. Paulin, "Force Directed Scheduling," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, **8**(6) pp. 661-679 (1989).

19. C.H. Papadimitriou and M. Yannakakis, "Analysis of Parallel Algorithms," *SIAM*, (1990).

20. E.L. Lawler, *Combinatorial Optimization Networks and Matroids*, Holt-Rinehart-Winston (1976).

21. V. Sarkar, *Partitioning and Scheduling Parallel Programs for MultiProcessors*, MIT Press (1989).

22. L. Hafer and A. Parker, "A Formal Method for the Specification, Analysis and Design of Register-Transfer-Level Digital Logic," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, **CAD-2**(1) pp. 4-17 (Jan 1983).

23. K.R. Baker, *Introduction to Sequencing and Scheduling*, John Wiley & Sons (1974).

24. J. Lee, Y. Hsu, and Y. Lin, "A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis," *International Conference on Computed Aided Design*, pp. 20-23 (1989).

25. C. Huang, Y. Chen, Y. Lin, and Y. Hsu, "Data Path Allocation Based on Bipartite Weighted Matching," *Design Automation Conference*, pp. 499-504 (1990).

26. S. Devadas and A.R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, **8**(7)(July 1989).

27. G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, Wiley Interscience (1988).

28. P. Pfahler, "Automated Datapath Synthesis: A Compilation Approach," *Processing and Microprogramming*, **21** pp. 577-584 (1987).

29. J.A. Nestor and D.E. Thomas, "Behavioral Synthesis with Interfaces," *Int'l Conf on Computed Aided Design*, (1986).

30. J.A. Nestor and G. Krishnamoorthy, "SALSA: A New Approach to Scheduling with Timing Constraints," *Int'l Conf on Computed Aided Design*, pp. 262-265 (1990).

31. C.H. Gebotys and M.I. Elmasry, "Optimal Synthesis of High Performance Architectures," *IEEE Journal of Solid-State Circuits*, **27**(3) pp. 389-397 (March 1992).

32. E.F. Girczyc, R.J.A. Buhr, and J.P. Knight, "Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation," *IEEE Transactions on Computer-Aided Design*, **CAD-4**(2) pp. 134-142 (1985).

33. N.D. Dutt and D.D. Gajski, "Designer Controlled Behavioral Synthesis," *Design Automation Conference*, pp. 754-757 (1990).

34. D.C. Ku and G. DeMicheli, "Relative Scheduling Under Timing Constraints," CSL-TR-89-401, Stanford Technical Report (1989a).

35. D.C. Ku and G. DeMicheli, "Optimal Synthesis of Control Logic From Behavioral Specifications," CSL-TR-89-402, Stanford Technical Report (1989b).

36. S. Hayati and A. Parker, "Automatic Production of Controller Specifications from Control and Timing Behavioral Descriptions," *Design Automation Conference*, pp. 75-80 (1989).

37. G. Borriello and E. Detjens, "High-Level Synthesis: Current Status and Future Directions," *Design Automation Conference*, pp. 477-482 (1988).

38. Meng and Brodersen, "Asynchronous Circuit Synthesis," *IEEE Transactions on CAD*, (1989).

39. Grossman, "Mixed Integer NonLinear Programming Techniques for the Synthesis of Engineering Systems," EDRC-6-83-90, Engineering Design Research Center, Carnegie Mellon University (1990).

40. Ra and Grossman, "Relation Between MILP Modelling and Logical Inferences for Chemical Process Synthesis," EDRC-06-87-90, Engineering Design Research Center, Carnegie Mellon University (1990).

41. W.F. Clocksin and C.S. Mellish, *Programming In Prolog*, Springer-Verlag (1984).

42. A. Brooke, D. Kendricke, and A. Meeraus, *GAMS/MINOS Users Manual*, Scientific Press (1988).

43. B. Cook, "," Solving General Integer Programming, Talk at University of Waterloo, Canada (May 1990).

44. N. Karmarkar, "An Interior Point Approach to NP-Complete Problems," *Integer Programming and Combinatorial Optimization*, pp. 351-366 (May 1990).

45. G.J. Minty, "On Maximal Independent Sets of Vertices in a Claw-Free Graph," *Journal of Combinatorial Theory*, **B28** pp. 284-304 (1980).

46. P.L. Hammer, E.L. Johnson, and B.H. Korte, *Annals of Discrete Mathematics 4, Discrete Optimization I,*, North Holland (1979).

47. M.W. Padberg, "Covering, Packing, and Knapsack Problems," pp. 265-287 in *Annals of Discrete Mathematics*, North-Holland (1979).

48. E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooykan, and D.B. Shmoys, *The Travelling Salesman Problem, A Guided Tour of Combinatorial Optimization*, Wiley-Interscience (1985).

49. H. Crowder, E.L. Johnson, and M. Padberg, "Solving Large Scale Zero-One Linear Programming Problems," *Operations Research*, **31**(5) pp. 803-834 (September 1983).

50. M. Grotschel, "On the Symmetric Travelling Salesman Problem: Solution of a 120 City Problem," *Mathematical Programming Study*, **12** pp. 61-77 (1980).

51. M.W. Padberg and S. Hong, "On the Symmetric Travelling Salesman Problem: A computational Study," *Mathematical Programming Studies*, **12** pp. 61-77 (1980).

52. M.W. Padberg, "On the Facial Structure of Set Packing Polyhedra," *Mathematical Programming*, **5** pp. 199-215 (1973).

53. G.L. Nemhauser and L.E. Trotter, "Properties of Vertex Packing and Independence System Polyhedra," *Mathematical Programming*, **6** pp. 48-61 (1974).

54. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, North Holland (1976).

55. Grimmett and Pulleyblank, "Random Near Regular Graphs and the Node Packing Problem," *Operations Research Letters*, **4**(4)(1985).

56. C.H. Gebotys and M.I. Elmasry, "A Global Optimization Approach to Architectural Synthesis," *IEEE International Conference on Computer Aided Design*, pp. 258-261 (1990).

57. C.H. Gebotys and M.I. Elmasry, "Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis," *ACM/IEEE Design Automation Conference*, pp. 2-7 (1991).

58. E.D. Lagnese, "Architectural Partitioning for Systems Level Design of Integrated Circuits," *CMUCAD-89-27, PhD Thesis*, (1989).

59. C-T. Hwang, Y-C. Hsu, and Y.L. Lin, "Scheduling for Functional Pipelining and Loop Winding," *DAC*, pp. 764-769 (1991).

60. C.H. Gebotys and M.I. Elmasry, *Optimal VLSI Architectural Synthesis: Area, Performance, Testability*, Kluwer Academic Press (1992).

61. 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 of IC and Systems*, **10**(4) pp. 464-475 (1991).

62. C.H. Gebotys, "Synthesizing Embedded Speed-Optimized Architectures," *IEEE Custom Integrated Circuits Conference*, (1992).

63. C.H. Gebotys, "Optimal Scheduling and Allocation of Embedded VLSI Chips," *ACM/IEEE Design Automation Conference*, pp. 116-119 (1992).

64. F. Depuydt, G. Goossens, J. Van Meerbergen, F. Catthoor, and H. DeMan, "Scheduling of Large Scale Signal Flow Graphs based on Metric Graph Clustering," *IFIP Conference on High Level Architectural Synthesis and Logic Synthesis*, (1990).

65. C.H. Gebotys, "Optimal Synthesis of Multi-Chip Architectures," *IEEE ICCAD*, (1992).

66. C.H. Gebotys, "Optimal Synthesis of MultiChip Architectures," *Tech.Rept.UW/VLSI 92-01,VLSI Group,Univ.of Waterloo*, (1992).

67. C.H. Gebotys, "Optimal Loop Winding Using An IP Approach," *Tech.Rept.UW/VLSI 92-02,VLSI Group,Univ.of Waterloo*, (1992).

3

Synthesis of Multiple Bus Architectures For DSP Applications

Baher S. Haroun[1] and Mohamed I. Elmasry[2]

Abstract

In this chapter, we present synthesis techniques of parallel VLSI processor architectures with multiple busses and functional units used for DSP applications. The presented architectures and synthesis approach are most suitable for applications with medium sampling rates (few MSamples/Sec) and medium to large storage requirements (tens to thousands of words) such as in single and multiple channel filtering and transform algorithms. Novel synthesis algorithms and architecture support are described for looped execution of regular algorithms which allow multiple address space for looped variables. These synthesis techniques are used in an architectural synthesis tool "SPAID-X" which inputs the behavior specification as a hierarchical signal flow graph representation that support folded loop constructs. The output of the tool is an architectural specification of the data path and the controller. We demonstrate the functionality of SPAID-X and the quality of the resulting architectures on a number of practical DSP applications and show that it produces results that are favorable to other approaches.

1.0 INTRODUCTION

The use of architecture synthesis tools to automate the search for an optimal VLSI architecture from a graph or language description of a signal processing system behavior have seen considerable emphasis in the past few years. The tasks performed by a synthesis tool are highly dependent on the underlying architecture structure and its main hardware components; the data-path composed of the functional units, data storage and their interconnection and the controller or interconnection of controllers. Different architectural styles and models are required for different sampling speeds and applications [1,2]. The main task of synthesis is to allocate the hardware resources and explore the different parallel implementations that executes the described behavior and satisfies the speed and cost constraints. Another important task the synthesis tool environment should provide is the exploration of the described behavior for alternative representations where each representation may induce different requirements on the hardware. These different representations can be explored through behavior preserving transformations. Therefore a synthesis tool for DSP has to provide its user with an environment that provides for:

- A natural behavior representation for DSP applications which is easy to learn and can hide the details of the hardware implementation.

1. ECE Dept., Concordia University, Montreal, Quebec, Canada. H3G 1M8. haroun@vlsi.concordia.ca
2. ECE Dept., University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1.

- Exploration of the different representations of a described behavior.
- An underlying architectural model (or models) suitable for the application and performance requirements.
- Synthesis algorithms and methodology that are used for that architecture exploration, optimal generation and efficient utilization that can take into account technology dependencies of speed, area and power dissipation.

Figure 1 shows the different dependencies of architectural synthesis and interactions.

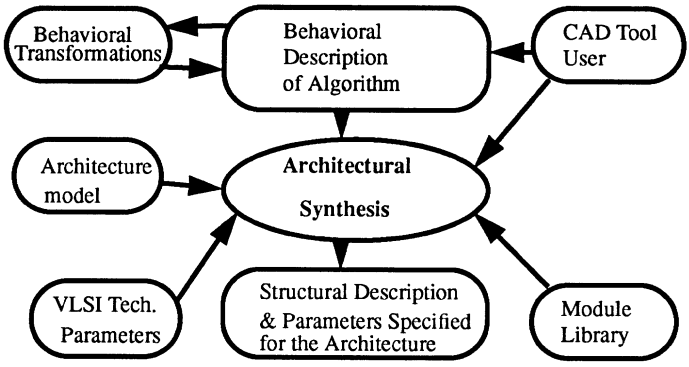


Figure 1: Architectural Synthesis

It is assumed that the structural description output of these tools feeds module generators and layout compilers to produce the final chip layout. In this chapter we elaborate on the above four points in architectural synthesis, specifically concerned with signal processing applications. Our architecture model will be that of the bus based processors. **SPAID-X**, an architectural synthesis tool for **S**ignal **P**rocessing **A**utomated **I**ntegrated-circuit **D**esign with e**X**tended design space, is used and presented as our example synthesis tool.

2.0 ARCHITECTURE MODEL CLASSIFICATIONS

We classify architecture models used in the synthesis of multiplexed architectures into three categories depending on the topology of their final VLSI layout. The method by which the main components of an architecture, the storage units and computation units, are interconnected determines their topology, hence we have the three categories; *random, linear* and *regular topology* architectures.

2.1 From Random to Linear Topology Architectures

To illustrate the differences between random and linear topology architectures, we use a small example of a second order digital filter (Figure 2). The operations of the filter (A1,A2,M1,M2) can be mapped *directly* onto an architecture with two adders and two multipliers and has the same structure of the filter with one to one correspondence of oper-

ations to functional units (adders, multipliers or FUs for short). Although such a direct implementation is as regular as the signal flow graph it represents, the architecture is inefficient especially if loops exist that have long delays between storage (z^{-1} registers).

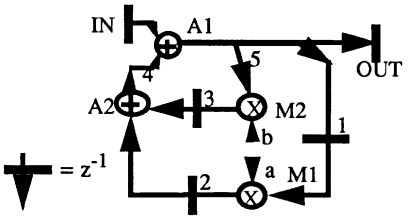


Figure 2: Second-Order Filter Example

To demonstrate the inefficiency, assume that the multiplier takes 40ns and the adder takes 20ns to produce valid data at their outputs, we can easily see that the fastest time that this filter can operate is the time for the closed loop (A2,A1,M2) to finish, which is (40+20+20=80nsec). Hence the maximum throughput for this filter is to have a new sample every 80nsec (neglecting storage time in the z^{-1} pipelined registers as well as communication delays). The adders will only be used 20nsec out of every 80nsec i.e. 25% efficiency, while the multipliers will be 50% efficient.

To reduce the inefficiency of direct implementation, random topology architectures attempt to re-use the FUs and share them between operations. A random topology data-path architecture for the above example filter is shown in Figure 3.a. It has one adder and

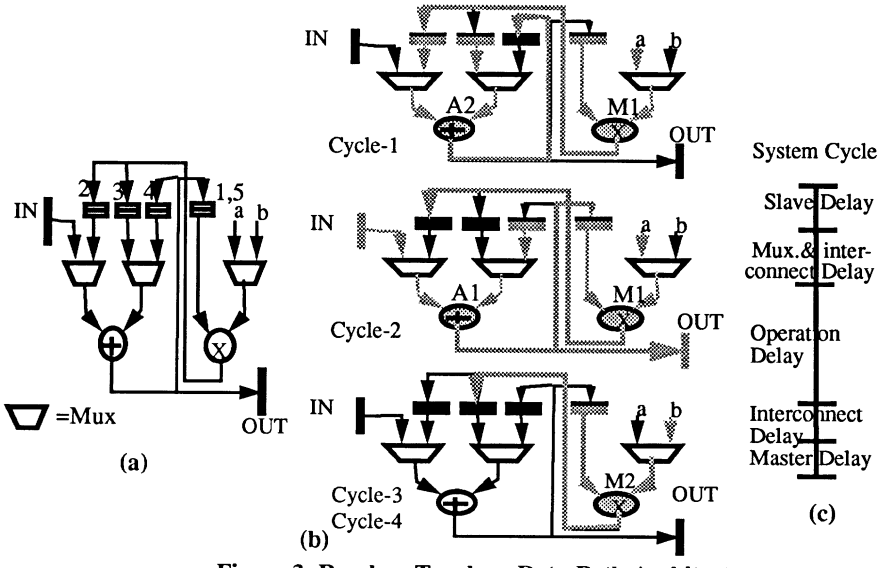


Figure 3: Random Topology Data-Path Architecture

one multiplier. Figure 3.b shows the execution sequence in four cycles where the lightly

shaded lines indicate hardware which is used in executing the operations in that cycle. The multiplier is used 100% of the time, while the adder is used 50% of the time. The architecture can operate on a new sample in slightly more than 80nSec, due to the extra multiplexer, interconnect and pipeline register delay, Figure 3.c. The main feature of this data path is that the interconnection between the FUs, pipeline registers and muxes does not have any specific topology, hence it can be termed to have random topology. This style of interconnection of FUs to the registers can be generalized as shown in Figure 4.a. The **IIS**

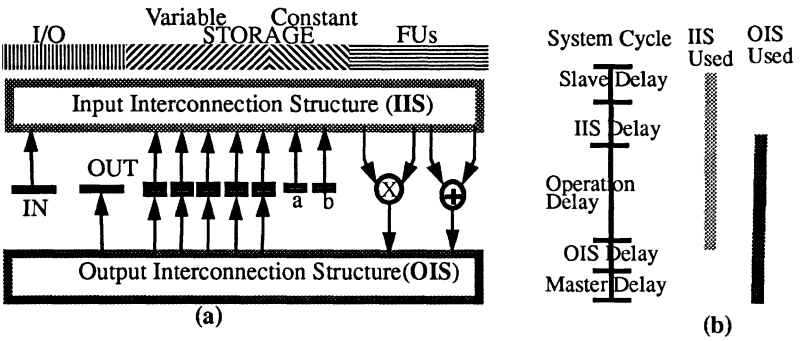


Figure 4: A General Data-path with pipeline registers

and the **OIS** in Figure 4 can be one of three types: a *mux* style, a *tristate* style or a *bus* style structures as shown in Figure 5 a-c. The mux style can result in high fan-in muxes which can require a large interconnection area. The tristate style (Figure 5.b) replaces high fan-in muxes with busses which can reduce the interconnect complexity but can result in extra capacitive bus loading (due to tristate buffers output capacitance) which increases the **IS** delay. The bus style (Figure 5.c) can result in less interconnect area than Figure 5 a, b as this style balances the use of large muxes of the mux style with high capacitive bus loading of the tristate style. The number of busses, NB, can change to increase or decrease

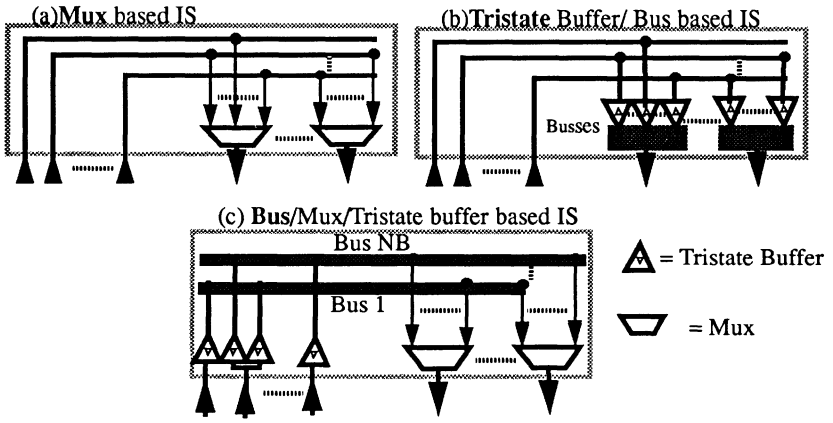


Figure 5: Interconnection Structures (IS)

accessibility of interconnections for data transfers. Smaller muxes are used in the bus style as less number of interconnects (busses) are connected to the muxes, while the busses are loaded by less tristate buffers, as outputs of **IS** can share busses. As more busses are added more parallelism is added to the number of data transfers that can be done at any time to a maximum of the number of destinations or sources connected to the **IS** whichever is smaller. To minimize the cost of the two interconnection structures attempts are made in most synthesis systems using a bus style or tristate style **IS** to further share the busses between the **IIS** and the **OIS**. We notice from Figure 4.b that the **IIS** and **OIS** are used simultaneously in the same cycle which implies that the buses used in the same cycle cannot be further shared between the **IIS** and **OIS**. This restriction can be alleviated, as done in SPAID [1], by moving the slave latches of the pipelined registers through the IIS to the inputs of the FUs. This results in that the **IIS** and the **OIS** do not overlap in their usage time in *any* cycle and hence the two interconnection structures can be merged into one IS as shown in Figure 6.a. Figure 6.b shows that the **IS** is used twice in one system cycle.

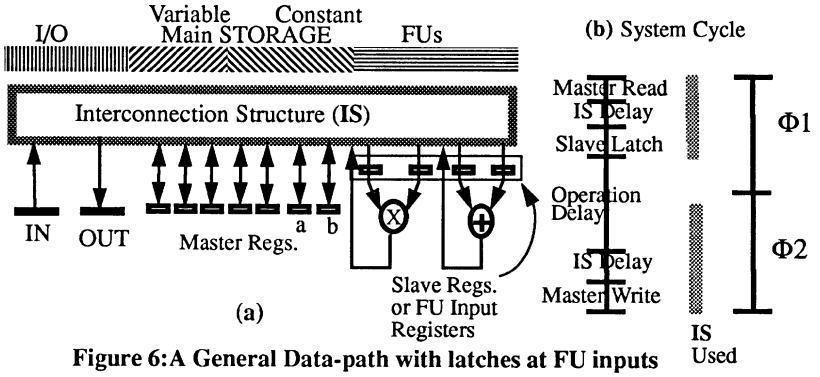


Figure 6:A General Data-path with latches at FU inputs

Hence, a two phase clock is used to define the data transfers on the **IS**. In $\Phi 1$, the data moves from storage to the FU input latches, on $\Phi 2$ the data moves from the output of FUs to the main storage. Input and output ports to the system can be considered as either FUs or as storage.

The topology of the **IS** determines the topology for all the data-path. We show in following sections, that defining a bus style structure for the IS results into a linear topology architecture. One advantage of the architecture in Figure 6 over the one in Figure 4 when using bus style **ISs** is that in the latter, as long an operation is executing on a FU, the **IIS** is used. For operations which use more than one cycle execution time the bus (in the **IIS**) used for that operation cannot be used (shared) to transfer data to other operations during these cycles. For the former architecture this extended use of the busses is reduced to a fraction of the cycle because data is latched at the other end of the connection which leaves the bus free to be used either at the end of the cycle ($\Phi 2$) or in other subsequent cycles. This means that a more efficient interconnection structure can result. Moreover, one can increase some operation's execution time (more than the required delay) for certain operations to reduce the demand on busses at specific clock cycle time which can further reduce the interconnections required.

2.2 Review of Synthesis Tools for Different Topology Data-Paths

Random topology architectures refer to an irregular interconnection between the functional units (FUs), pipeline registers and multiplexers to form *a pipelined data path.* Variable and state storage are mainly done by the pipeline registers, the structure is similar to that of Figure 4, where the interconnection structures (ISs) are personalized to the specific synthesized algorithm. Both mux based and bus based interconnection styles where used. Examples of these architectures can be found in [7-9]. The main reason for the random style of interconnection that result from these synthesis tools is the reduced sharing between the IIS and the OIS because of the overlapping in the their use in one system cycle, as well as that interconnect binding is left at a later stage in the binding process and are not taken initially in the synthesis as constrained resources. Large storage of data was not addressed for these style of data-paths.

Linear topology data-paths are characterized by the bussed interconnection between its functional units with a pre-scheduling assumption that FUs have access to a constrained number of busses. The bus topology results in efficient layout compared to random topology, on the expense of the possible increase of the system clock cycle duration because of bus delay overhead. Data storage is done through register files placed at either FU inputs (CATHERDRAL_II)[2], as multi-ported register files (Theda) [6], or on each bus as in SPAID [3] and ASPS [11]. Large storage of variables were addressed for this style of architectures [2,3,11] by assuming RAMs as FUs during synthesis and explicitly including RAM load/store operations in the behavior specification. Full automation for memory assignment was not achieved as the following tasks were relegated to the user of the synthesis system in case of SPAID and CATHEDRAL-II; a) Distributing the data storage on to separate RAMs. b) Deciding on the mechanism for parallel access of data from the RAMs by the data path and address generation path. c) Specifying the address space and address computation. These tasks are automated in SPAID-X as shown in later sections.

The third category, the *regular topology*, has evolved from the random topology to resolve the large interconnect area that results with random topology approaches and without resorting to the use of busses which may increase cycle time and hence reduce throughput. Regular topology architectures use the regularity inherent in some DSP algorithms to generate similarly regular pipelined architectures with partitioned control. Cathedral-3 [14] is the main example in this architecture model, which uses lowly multiplexed pipelined data-paths that exploit regularity of the data flow graph by partitioning it into clusters and re-grouping compatible ones into sets of operations. Algorithm specific units (ASU) are designed for these sets where chaining of operations is allowed as well as deep pipelining.

In this chapter we present the synthesis techniques for a linear topology multiple bus architecture. The original multiple-bus architecture (SPAID) was presented previously in [3,4]. In the following sections, extensions to that architecture to handle algorithms that require large storage (SPAID-X) are presented. In section 3 , we present the behavioral representation to the tool SPAID-X. In section 4, algorithm transformations which can affect the final architectural outcome of the synthesis tool are discussed. The different aspects of the synthesis tool SPAID-X are detailed in section 5. Finally experiments with the synthesis tool are presented in section 6.

3.0 DSP ALGORITHM REPRESENTATION

Behavioral representation of an algorithm to a synthesis tool has to be closer to the application domain it describes than to the hardware it is supposed to generate. Using flow graph representations is natural for most DSP applications. A graphical representation is easier to learn compared to learning a new language. Hierarchy in a graphical representation results in a block diagram equivalent of the designed system. It has been used for representation for different applications:

- DSP simulators; SPW from Comdisco [18].
- Retargetable compilers for single and multiple programmable DSPs; GABRIEL[19].
- Synthesis of ASICs; SPAID-ACE [17], HDS/SPW [18].

The graphical representation facilitates estimating timing bounds on execution like minimum latency and throughput. Structural transformations can be done on the graphical representation to optimize system timing (CSD coding, retiming etc.) as explained in following sections. We present here a *generalized signal flow graph* representation (GSFG) that is used as an input to the synthesis tool SPAID-X to specify DSP algorithms. The GSFG can be used to represent general single and multi-channel filters, single and multi-rate systems as well as transform such as FFT.

3.1 The GSFG Representation

The nodes of the GSFG are either one of the following:

- Executable Operations, N, can be of any of a number of primitive types; additions, subtractions, comparison, multiply etc. The expressiveness of the GSFG depends on this primitive set. Input and output operations are considered executable.
- Non-executable operations, NX, are constant nodes, source and termination nodes.
- Executable *Block Nodes*, BN, are represented in a lower level of hierarchy by only other block nodes or a *Ground Block Node*.
- *Ground Block Nodes:* GBN, are represented internally by a signal flow graph (SFG) which is formed only of executable operations. A ground block node is also used to represent an inner loop for looped execution, and is annotated by the range of the loop indices (more details in section 3.2).

The edges of the GSFG represent data values and are either:

- Variable edges E which also indicates data dependency and operation precedence.
- Constant edges EC.
- Input/output edges EIO.
- Time (Non data) precedence edges P (forced time precedence between nodes).

Each edge is either *a wrap_edge* or an *inner_edge*. *A wrap_edge* is used for representing looped execution as explained in section 3.2. The *inner_edge* is used to represent all other

edges and is of the form *inner_edge(Source_node, Destination_node, Separator)*. Any edge can be a separator edge which has z^{-w} delay with weight w. *Separator*=1 if a z^{-1} delay register was intended in the original SFG on that edge and the edge is called a Separator edge, otherwise *if Separator*=0, the edge is called a non-Separator edge. Note that an infinite number of outer loop iterations in a DSP system is always implied by a GSFG. An example of a GSFG representation of an elliptic wave digital filter is shown in Figure 7.a. This filter has been used as a benchmark filter for synthesis. This filter contains 8 multiplication and 26 addition operations. It has 68 data edges, a number of which carry identical data. For example addition operation A1, has three output edges all carrying the same output variable from A1.

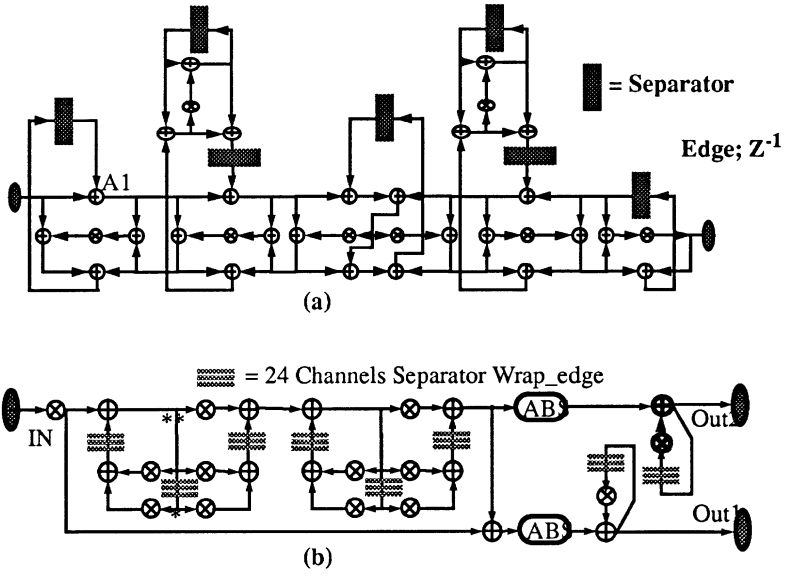


Figure 7: Elliptic Filter SFG (a), 24-Channel Telecommunication Filter (b)

3.2 DSP Algorithm Representation For Looped Execution

Signal processing algorithms such as multi-channel filtering, transforms (FFT, DCT etc.), image convolution algorithms and artificial neural networks processing algorithms, all share the property of being regular algorithms that require **repeating** a finite number of times a specific sequence of operation on a **large** number of different data sets within another infinite time loop (the implicit DSP iteration every sample period).

In SPAID-X, a hierarchical signal flow graph SFG is used to represent a DSP algorithm containing loops. A looped execution of a SFG can be done by folding the original flat SFG onto the inner loop SFG. We use an example of an 8-point FFT shown in Figure-8.a to demonstrate the folded representation. The FFT is folded into a *one butterfly* inner loop as shown in Figure-8-c, or partially folded to a *double butterfly* inner loop as in Figure-8.d.

All instances of the inner loop are indexed with one or multi-dimensional Indices. The use of partially folded loops as inner loops results in more parallel operations to be scheduled which can increase the architecture utilization, hence throughput, on the expense of a longer control sequence for the inner loop.The folded SFG has edges that represent one or more of the edges of the original SFG. Each edge which wraps around the inner loop (*the one or double butterfly* in our example) is represented by a relation: *wrap_edge(Source_node, Destination_node, listof _all_indeces, Separator)*. Where *listof_all_indices* is a list of elements of the form *[Source_index- Destination_Index]* indicating all the edges in the flattened SFG that fold into this wrap edge in the folded representation (Figure 8.c or 8.d). Each index is an ordered set [e.g.(I,J)] indicating the row and column of the inner loop in the original SFG. Constant and input/output edges

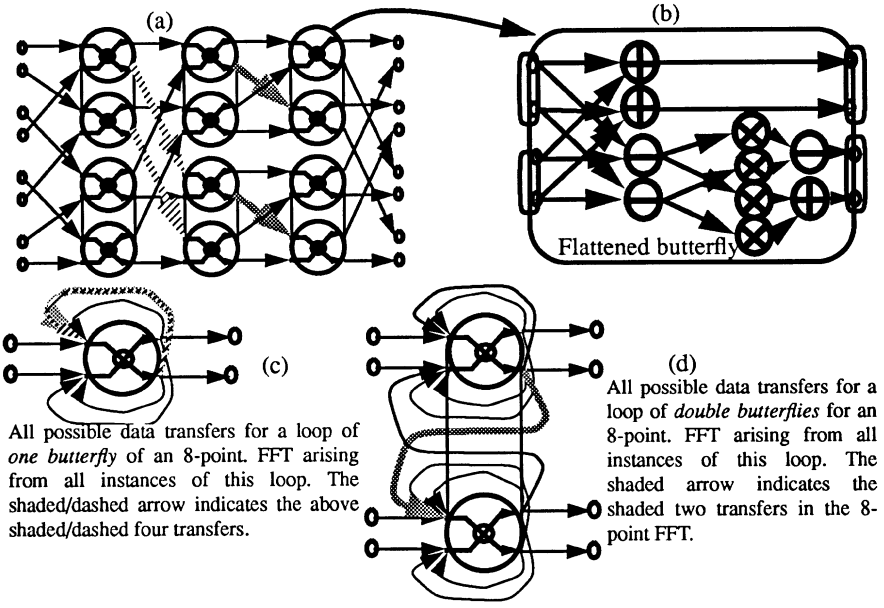


Figure 8: Folded FFT representation

are also defined similar to the variable edges and are of type *wrap* or *inner*. In the looped representations both the hierarchy and folding can be totally or partially flattened in an implementation. These trade-offs are left to the user of the synthesis tool.

In the case of representing multiple channel filters, each *separator* edge of the SFG has a *listof_all_indices* with a length equal to the number of channels and its elements are of the form [I-I] for all channel Indices. An example of a 24 channel telecommunication filter is shown in Figure 7.b.

The folded graph or the SFG, **G**, is used as the data structure in the scheduling and binding phases of our synthesis tool by properly annotating the nodes and edges of **G**. Note that

the folded representation used in SPAID-X does not have any implicit address generation as the Silage description of the FFT in [13].

An important property of the folded graph is that it can have an input to a node (operation) with more than one incident edge. These edges represent data transfers that are mutually exclusive that do not occur at the same time as they occur at different instantiations of the inner loop. Note also that the out-degree of the output of an operation increases quite substantially in the folded graph. This high in-degree and out-degree of the folded graph has ramifications on the synthesis procedure as shown in subsequent sections.

3.3 Bounds from the Signal Flow Graph

Bounds on the total execution time and the architecture can be derived from the SFG description if the sample period of the DSP algorithm is known (T_{sample}) and an estimate is known for the delay in executing each operation on a given functional unit(FU_delay). Typically there can be more than one operation delay depending on the selections done for the functional units. Also for the technology used one can have rough estimates of bus delays, register read/write delay times.

The following are a set of measures that are useful for obtaining bounds on the architecture as well as total execution time:

- Operation $Delay_{Op\_Type}$=(2*Bus_Delay + Read_delay + Write_delay + FU_delay)
- Shortest Execution Time $(T_{exec})_{shortest} = \sum Delay_{op\_type}$ of the longest delay path in the graph with total separator weight=0. $(1/(T_{exec})_{shortest})$ gives the maximum throughput for the given SFG. An architecture exists if $T_{sample} > (T_{exec})_{shortest}$.
- Minimum Execution Time $(T_{exec})_{min} = \sum Delay_{op\_type}$ of the longest delay of a loop/path divided by the total separator weight in that loop/path. $(1/(T_{exec})_{min})$ gives the highest throughput that may be achieved by a retiming and pipelining. (section 4.2).
- Lower bound on the number of functional units (*NFU*) of type (*Type*) that execute operations of different types (*OpType*) and an efficiency (FUEfficiency which is typically between 50-100% in a number of applications) is given by:

$$NFU_{Type} = \left\lceil \sum_{\forall OPType} \left(\frac{NumberofOperations_{OpType} Delay_{OpType}}{T_{sample}} \right) / (FUEfficiency) \right\rceil$$

- A lower bound for the architecture clock cycle T_{cycle} is the minimum $(Delay_{op\_type})$.

The ratio $R=T_{sample}/T_{cycle}$ or $(T_{exec})_{shortest}/T_{cycle}$ can be used to determine the style of the architecture to be used. In SPAID-X, the suitable ratio is from tens to few thousand and NFU less than 10. This is because a high ratio means that the architecture is highly multiplexed and there is enough sequential execution to warrant a bussed architecture. On the other hand, for DSP algorithms where this ratio R is less than 10, a highly pipelined architecture (of random or regular topology) is most suitable as the degree of parallelism is very high at such low ratios and have to be used to achieve the high throughput.

4.0 DSP ALGORITHM TRANSFORMATIONS

For each intended behavior of a DSP algorithm there are more than one SFG representation that have equivalent behavior but different hardware implementations. Transforming one representation of behavior to another is important in a synthesis tool that works as an aid in exploring the design space. Structural transformations are applied on the behavioral description GSFG and do not affect the behavioral intent.

Transformations that can be performed on a SFG have either *local* or a *global* effect on the graph. The goal is to achieve the following:

- Reduce $(T_{exec})_{shortest}$. The delay on the critical path can be reduced by changing the operations done (local transformation) or by inserting separator delay(s) between input and output hence pipelining the behavior (global transformation).
- Reduce $(T_{exec})_{min}$; By moving separator delays around a loop, the delay of the longest path can be distributed on to shorter ones (global transformation).
- Increase the efficiency of utilization of FUs; By re-arranging Separator storage such that operations of one type are not to be executed all towards the end or towards the beginning of the execution sequence (global transformation).

4.1 Global Transformations

Retiming Transformation: it is an optimization technique based on separator register[1] movement. It affects the length of the critical path delay between separator registers as well as the number of separator registers. Retiming was proposed to improve the clock rate of synchronous circuits[20] by minimizing T_{exec} of a given SFG. Retiming can be done by applying cutset transformations on the SFG. A cutset is composed of two types of edges with different directions with respect to the cut. Subtracting a fixed number of separator registers from each edge in the cutset in one direction, and adding the same number of separator register to each edge of the cutset with the other direction, without having negative separator registers, results in no change in the functionality of the GSFG as shown in Figure-9. Applying the cutset locally on one node, one can move a fixed number of delays from the input edges to the output edges of the node. This local register transformation can be put in a set of linear inequalities, with global constraints on path delays, and total latency. Details of the formulation can be found in [3].

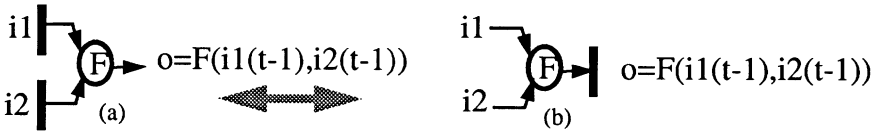


Figure 9: Retiming Transformation

A consequence of retiming, apart from reducing the critical path delay and decreasing the number of data dependencies, is that the order of operation execution can change after

1. A separator register corresponds to a separator edge with z^{-1} delay.

applying one cutset transformation. Referring to Figure 9, operation F in (a) has first precedence and can execute in the first cycle since its data are readily available from the previous iteration execution, while in (b) operation F has last precedence and can only execute after all preceding operations have been executed. This change of precedence can be used to advantage to balance the operation requirements in the different states (clock cycles) of the execution. For example, two addition operations with the output of the first is connected to the input of the other with an edge weight of w=0, can only be done in sequence in one sample period. If a separator register can be placed on that edge with retiming, then the order of execution may be reversed or even can be executed in parallel. This demonstrates the extra degree of freedom that can be introduced by retiming to the scheduling of operations. Another advantage is that the retiming formulation can allow for an extra latency to be inserted between input and output. This is equivalent to pipelining the behavior description level rather than functional pipelining at the hardware level. They are essentially equivalent, but behavioral pipelining can also be done for recursive filters which is not the case when functional pipelining is used.

Unfolding transformation: This is a useful global transformation required to accommodate multi-rate DSP systems [4]. In addition, for single rate systems it can result in better architectures. In this transformation the operations of k consecutive sampling iterations are merged together in one iteration operating on one data vector formed of k consecutive data inputs and producing one output vector of k consecutive outputs. Moreover, unfolding transformations are used to achieve upper bounds on throughputs in single rate DSP algorithms, and in conjunction with local transformations can lead to a scattered look ahead implementation [22] as shown in the next section. In Figure 10-a, a DSP algorithm SFG **H** is represented in an applicative state transition format where **A** is an interconnection of the nodes of **H**, and all edges with w=1 are displayed as feedback path **S**. Inputs **X**

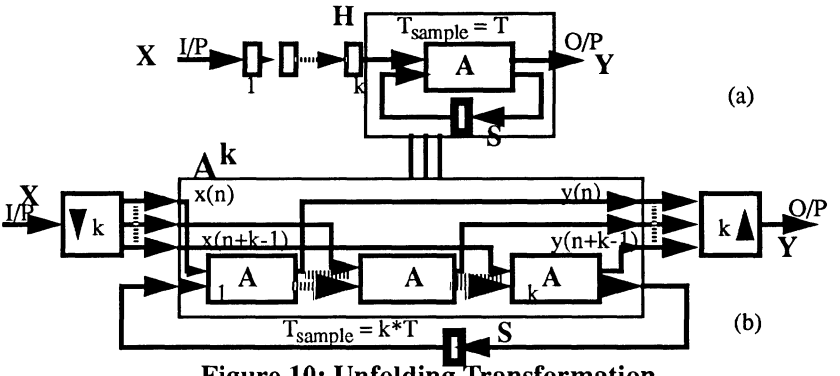


Figure 10: Unfolding Transformation

and outputs **Y** are operating at a rate 1/T. First, an extra weight k is added to the input separator edges increasing its latency by k*T; k an integer (assuming that this latency increase can be tolerated by the external behavior). Figure 10.b defines the unfolding transformation using a decimator at the input and an interpolator at the output and an unfolded functional block $\mathbf{A^k}$ operating on **k** data values at a time. The details of $\mathbf{A^k}$ are also given in Figure 10.b, which is formed by replicating **A** (the GSFG without separator edges) **k** times

and properly adding the edges between the replicas of **A**. Note that the resulting graph of $\mathbf{A^k}$ has **k** times more operations in it but **k** times more time to execute. The extra operations present can allow for better utilization of the architecture and can also allow other local transformation, described in the following section, to have a significant effect on the T_{exec}.

4.2 Local Transformations

A number of local transformations can be applied on few operations of the SFG with an objective to either reducing T_{exec} or eliminating operations from the SFG. This can have an effect on the quality of the architectures synthesized. Local transformations have been used in a number of synthesis systems (SPAID[3], HYPER [5]). The transformations are performed either manually or by using scripts consisting of a sequence of local and global transformations as done in SPAID-X or in an automated probabilistic search to minimize a heuristic cost function as done in HYPER. The important local transformations are summarized in Figure 11 in the following types.

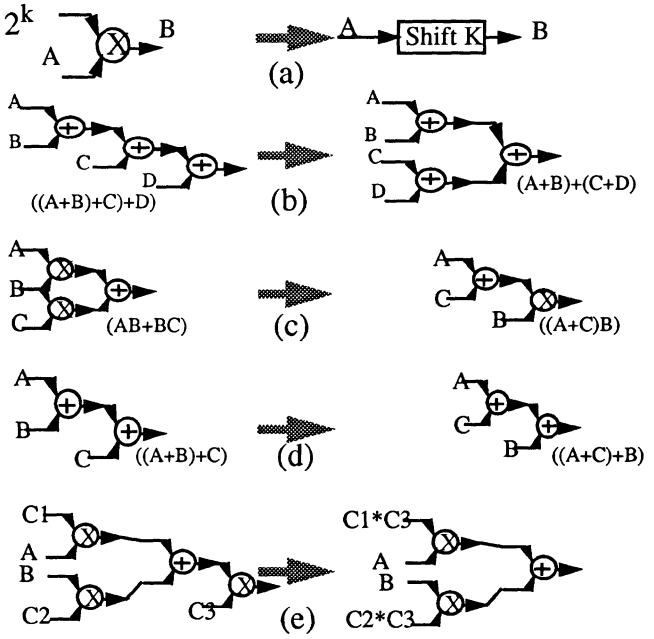


Figure 11: Local Transformations

- Equivalence: Used to replace an operation by a less costly one (or eliminate the operation, e.g. multiply by one) as shown in Figure 11.a. A common example for the equivalence transformations is the elimination of multiplications by constants with shift/add/subtract operations using cannonic signed digit (CSD) representation.

- Association: Used to reduce the level of a group of operations to reduce delay on a critical path as shown in Figure 11.b
- Distribution: Used to reduce the number of operations as shown in Figure 11.c
- Commutation: Used to reduce the delay on the critical path. As shown in Figure 11.d, if B is available at a later clock cycle than C, then exchange B and C.
- Constant Propagation/Back Propagation: Used to reduce the number of multiplications as shown in Figure 11.e.
- Scattered look ahead transformation [22] is used to decrease $(T_{exec})_{min.}$.This transformation is performed by an unfolding transformation then applying a sequence of constant back propagation and commutation transformations as shown in Figure 12.

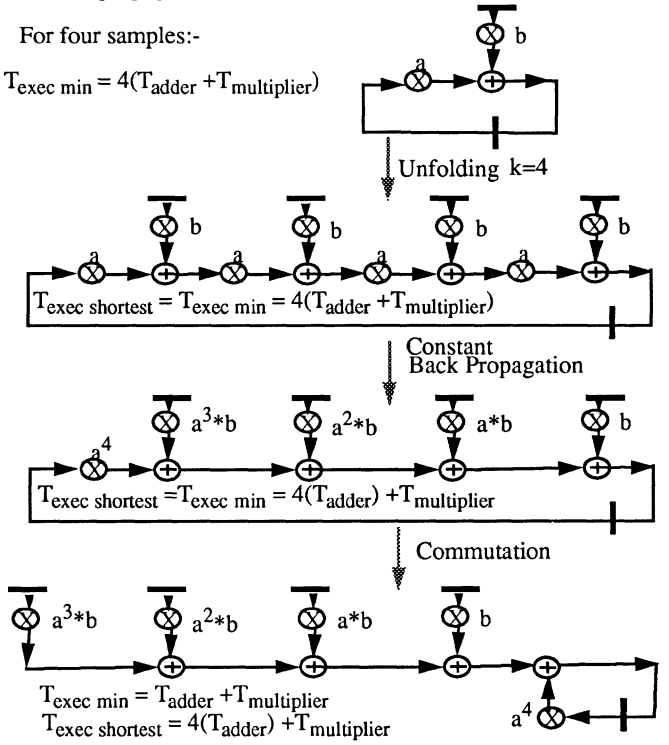


Figure 12: Scattered Look Ahead Transformation

The result is an algorithm with less multiplications that can be pipelined to execute at k times the maximum throughput of the algorithm before applying the scattered look-ahead transformation. Note that in all local transformations, quantization behavior of the signal processing function may change and careful application of these transformations with provisions to increase bit resolution of computations to overcome quantization problems have to be applied.

5.0 SPAID-X MULTI- BUS/FU/MEMORY ARCHITECTURE

The bus based architecture that results from our synthesis approach in SPAID-X is shown in Figure 13. The data-path of the architecture has NB busses, NB register files each with a maximum NRF registers and functional units of different types with multiplicity M_{FU} for each type FU. The NM RAMs are connected to the data path through the memory interface, shown in Figure 13. The data path is based on the model of Figure 6 which uses a two phase clock. On the *read phase*, data are *read* from either the *register files* or from the *memory interface* to the FU input slave registers. On the *write phase*, data are written to the register files, Memory Interface OR to FU input master register. The slave register of the FU double buffered register is only updated on the first phase. The master register allows FUs to write through directly to other FUs without going through the register files.

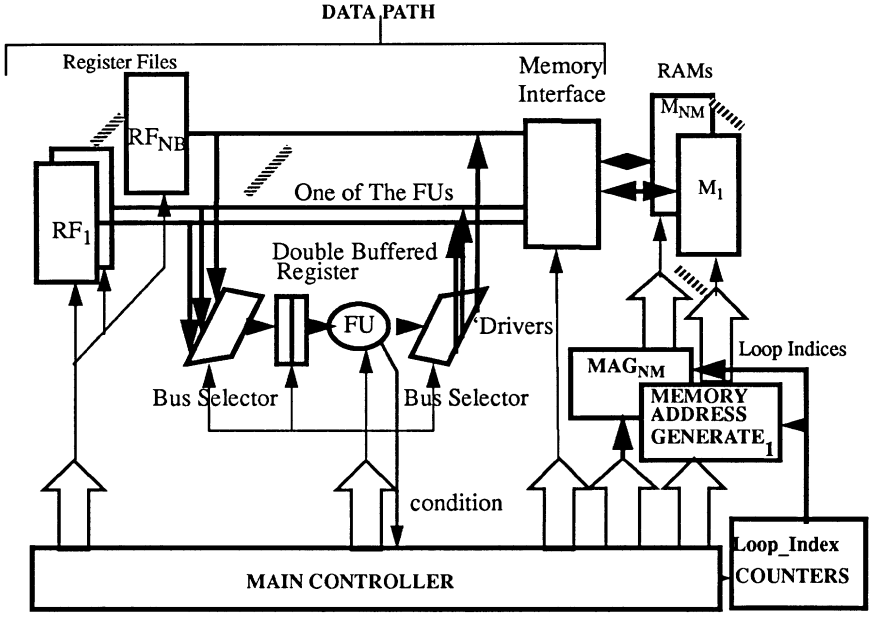
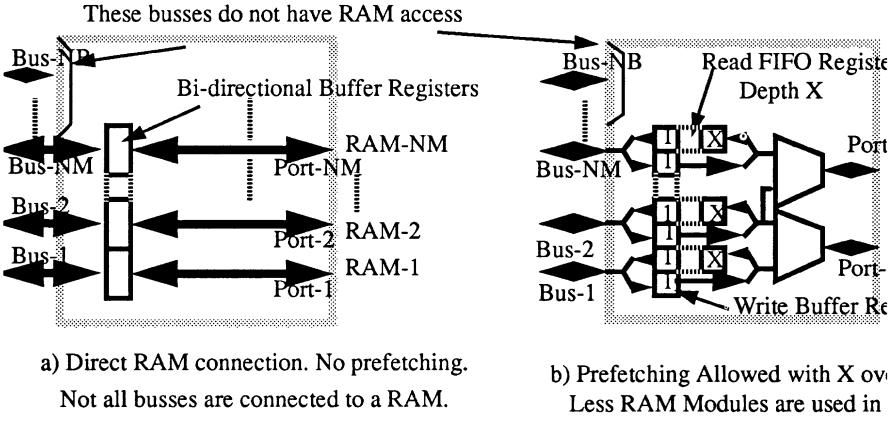


Figure 13: The Multiple Bus/FU/Memory Architecture

The main controller, details of which are presented in section 6.5, issues a control word on the read and the write phases of the system.To support looped execution, *loop_index* counters are controlled by the main controller to change every new inner loop iteration scanning all indices values in a specified order. The main controller executes a control sequence repeatedly until the last iteration of an inner loop. The main control sequence is independent of values of the *loop_index* counters. Each memory (RAM) address generator (MAG) takes a control sub-word from the main control sequence and the *loop-index* counters to produce a memory address on each phase of the system clock, where the RAMs alternate between read and write. The MAG is pipelined at its output to store the address. The control word to the data-path is pipelined by one phase delay to allow for the

MAG address evaluation. The *Memory Interface* can be **either** a direct connection from the buses to the RAMs through a bi-directional buffer and each RAM is connected to only one bus, Figure 14.a, **or** through a bi-directional FIFO (depth X in read path, 1 in write path) and each RAM can be connected to any of the busses, Figure 14.b. Typically, not all busses are connected to the memory as shown in Figure 14.b. The depth of the FIFO, X, is chosen small (1 -3, and is determined by synthesis). The FIFO allows data to be prefetched from any RAM and wait at one of the NB busses. This allows the data-path to have NB parallel RAM accesses at one clock cycle from only NM RAMs (NM<NB). This implies that the high internal parallelism of data transfers in the data-path is maintained while reducing the number of RAM modules required. By reducing the number of RAM modules, better storage efficiency is achieved as well as less MAGs are used which reduces the controller size).



a) Direct RAM connection. No prefetching. Not all busses are connected to a RAM.

b) Prefetching Allowed with X ov Less RAM Modules are used in

Figure 14: Memory Interface Detailed

6.0 SYNTHESIS OF THE MULTIPLE-BUS DATA-PATH

The synthesis procedure can be divided into three main tasks as given in Figure 15. These tasks of, allocation, scheduling and binding are inter-dependent and an optimal synthesis can only be a result of taking all tasks together. Because of the complexity of the synthesis process the task dependency is usually broken at some point and iteration is applied to account for the dependencies. The approach used for SPAID-X was to make an initial allocation assumption for the major parts of the data-path that determine the degree of parallelism in execution which are the number and type of FUs and the number of busses. The reason for such a choice stems from the fact that VLSI layout area and bus loading limit these allocations to only a few (1-10 typically). An exhaustive *or* a branch and bound search for these variables is used because of the limited number of alternatives and hence the search is practical. With such initial allocation, a more detailed scheduling and binding of operations to resources is done. Finally, the allocation is revisited to determine detailed storage allocation as well as bus connections, memory assignment and control. An important aspect of the SPAID-X approach is that the initial allocation of FUs and busses assumes the most flexible interconnection structure (**IS**) similar to Figure 6. For such an

initial architecture, each edge in the SFG is assigned a master register with full access to the **IS** while all FUs have full access to the **IS**. The **IS** is a bus based structure similar to Figure 5.c, but with all sources and destinations fully connected to all the busses. Further optimization of the data path results in the architecture of Figure 13, in spite of fully connected interconnection structure of Figure 6 being the one used in the scheduling. This is a major advantage for this style of architectures since if the number of busses, NB, is increased to be equal to a maximum of the number of FU inputs, this architecture can achieve the same throughput of random topology architectures with the same number of FUs. Because of the use of busses in SPAID-X architecture to regularly connect the components of the data-path, a linear topology layout results. This has obvious VLSI area advantages as was demonstrated in a chip described in [23]. High throughputs can also be achieved if fast busses are used in the VLSI implementation.

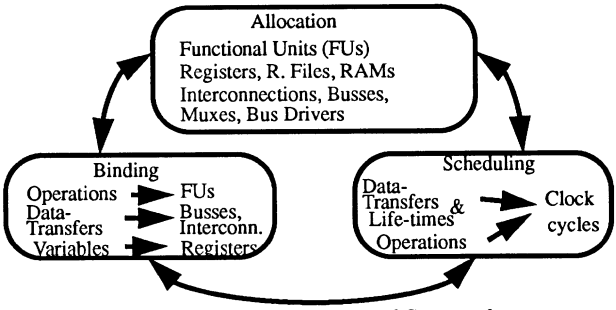


Figure 15: Tasks of Synthesis

6.1 Overview of The Synthesis Procedures

The hierarchical GSFG describing the system is used as the input. At each level of hierarchy the block nodes are sequentially ordered to ensure data precedence. At the level of the ground block nodes, the SFG describing the node is then synthesized, based on the current allocation of FUs and busses, to obtain an execution time T_{exec}. The actual starting and end clock times for each ground block node is computed after all block nodes are synthesized and the sequence of execution of all block nodes is known. If the execution time for the entire GSFG is not satisfied another allocation is tried.

The synthesis procedures in SPAID-X of a SFG for any of the ground block nodes are summarized in Figure 16. An initial SFG specification of an algorithm (or a folded loop) is used as the input. The first step of the SPAID-X synthesis procedure is to partially order the operations, N, of the SFG into sets (*posets*, N^i| i=0 to I) of data independent operations for each poset (i), where $N= \sum N^i$. For each poset N^i, the number of operations that are of different types which can be executed on a functional unit of type FU is N^i_{FU}. Partial ordering is done by default in SPAID-X by an *as late as possible* (ALAP) algorithm. The ALAP makes operations on the critical path appear early in the first posets. A poset *i* is not equivalent to state (or clock cycle) *i* of the architecture, the role of posets is merely to force precedence in the scheduling and binding phase of the synthesis. Any other partial order (or schedule; where a state can be interpreted as a poset index *i*) can be accepted as input, hence other scheduling heuristics such as force directed [9] can be used to deter-

mine the partial order. The posets are used as an input to a list scheduler where operation bindings to clock cycles are then determined as well as their binding to specific FUs while taking into consideration bus availability. The details of the list scheduler are explained in section 6.2. One of the main advantages of using a list scheduler is that it binds one operation at a time while knowing the current status of the architecture. This allows dealing with very complicated FUs with local storage and deep pipelining which other optimal schedulers (such as ILP approaches [12]) could not support efficiently. Such FUs can be required to handle efficiently signal processing algorithms like neural networks [16].

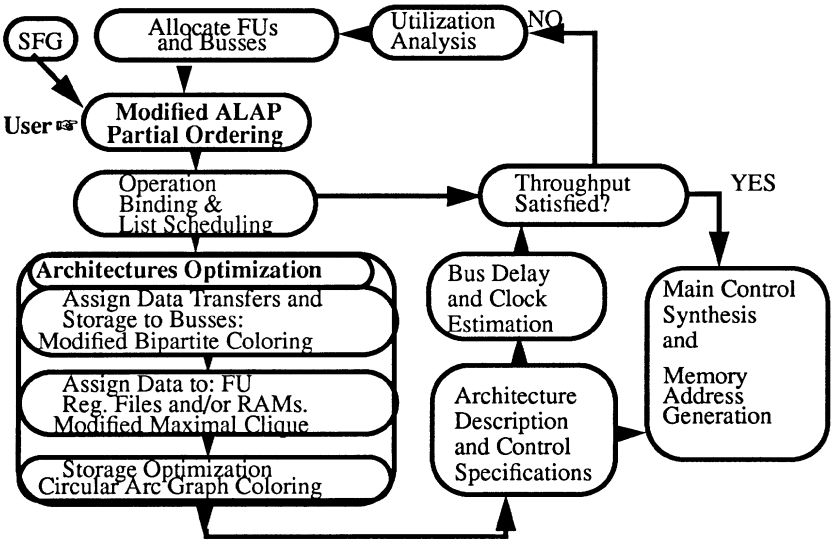


Figure 16: SPAID-X Synthesis Procedures

After the scheduling is done, a throughput assessment is done based on the number of clock cycles required. If the throughput is not satisfied a utilization analysis of the resources can be done and the heavily utilized resources (FUs or busses) are incremented. A more accurate throughput can only be obtained if the system clock cycle is known. Since the system clock cycle depends on the bus loading, the estimate of the clock cycle has to be relegated to after the architecture optimization.

An architecture optimization phase follows the scheduling and binding of operations to FUs. At this phase, the binding of data transfers to busses as well as the memory assignment and storage minimizations are done as explained in section 6.3 and 6.4.

After an architecture search is performed, a full architecture specification can be generated from the scheduling, binding and architecture optimization phases. The full architecture specification includes a controller specification for both the main controller and the MAG of each RAM (section 6.5 and 6.6). Ideally, the controller synthesis should influence the optimizations done in the architecture synthesis. We partially address this issue in SPAID-

X by providing for options in the memory optimization phase to try to minimize the size of the MAGs as will be explained in section 6.7.

6.2 Scheduling of Operations and Bus Transfers

Scheduling and FU selection: Operations are ordered within each poset *such that* the operation that has all required input data available at the earliest clock cycle has first priority *otherwise* priority is to the operations of a type with higher *Busy Time* BT= $(N^i_{FU}/M_{FU})*D_{FU}$, where D_{FU} is the number of clock cycles it takes to execute an operation of that type on the FU. For the operation with highest priority, the binding is done to the selected FU. The selection of the FU for binding is done based on a **best fit** of the data available time (this is similar to bin-packing heuristics) as shown in Figure 17, where four different situations of data available times (A-D) are shown and the FU selected is shown.

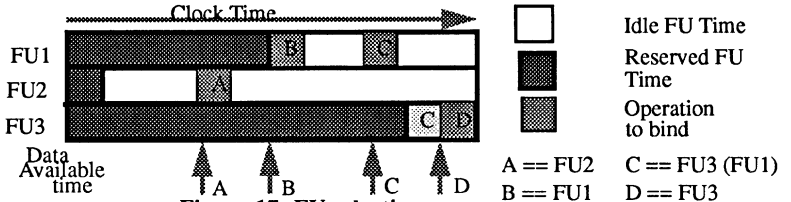


Figure 17: FU selection.

For situation C, if FU1 is selected then a large idle time on FU1 is introduced. The heuristic used in SPAID-X allows for a small postponement of binding the operation if a large idle time can be reclaimed. A fixed ratio (typically 3-10) between idle time and postponed time is used to determine the FU selection.

Bus reservations: Before binding an operation to a selected FU, the input variables to the operations have to be transferred on the busses from storage to the FU inputs. Similarly after the operation is bound the output variables have to be transferred back through the busses to storage. In this binding phase, our assumption is that each edge of the SFG is represented by a master register that has access to all the busses in the system and in turn all FU inputs or outputs have access to all the busses. Therefore, it only matters that any one bus is not reserved and can be used by a variable for transfer at a specific clock cycle. Specific assignments of transfers to specific busses come at a later stage in the architecture optimization. What is only needed at that point is to reserve a maximum of NB different variable transfers at any clock phase.

A complicating factor in bus reservation is that some of the operation nodes in the SFG can have quite a large out-degree (specially for a folded loop graph because of the added wrap_edges). These edges represent variables with *identical* values or *mutually exclusive* values (in case of wrap_edges in folded graphs). Moreover, when more than one edge have the same input of an operation node as a destination then these transfers are *mutually exclusive* either because of conditional execution or of being wrap_edges of a folded graph. This possibility of occurrence of large in-degree or out-degree of nodes has to be taken into account when doing variable transfers to bus reservations and bindings to account for identical or mutually exclusive variables.

For the set of FUs and NB busses assumed, the synthesis proceeds by maintaining a reservation table for all FUs, and for each read and write clock phase a data-transfer reservation list. The data-transfer reservation list is maintained to have the following properties at each read or write system clock phase:

1. Maximum NB data transfers with different values (i.e different source operations).
2. Mutually exclusive transfers count as one transfer.
3. Total number of transfers (identical, mutually exclusive or other-wise) at any clock is bounded by NB + SK. SK is a slack variable with a value up to the maximum out/in degree of the folded graph. SK is a parameter to ensure an architecture optimization that results into only NB register files as discussed in section 6.3.
4. Total number of wrap edges transfers with different values is bounded by NM. NM is chosen to be the desired number of RAM modules connected to the busses. NM =< NB. This is because wrap_edge variables are typically assigned to the RAMs.
5. If a reservation for a bus transfer is attempted at the current cycle and any of the above bounds is reached, the next cycle is tried. If this transfer is for the first input variable to an operation, the operation is postponed. If it is the next input variable or an output variable from the operation, the operation execution time on the FU is stretched by one cycle. This variable operation execution time is only possible in this style of architectures because data is buffered at the input of the FUs which is not the case with random topology architectures.

The procedure for scheduling and binding operations is then briefly: For each member of a *poset* of operations select the FU and find its first available clock time $T_{i\_av.}$. For all edges to which this operation is a destination, reserve a place on the read reservation lists starting at T_{i_av} to get $T_{reserved}$. Bind the operation to the FU starting at $T_{reserved}$. Reserve the FU table and find an output data available clock T_{o_av} from the FU. For all edges to which the operation is a source, reserve a place on the write reservation lists. It is evident that no binding of data transfers to specific busses are done in this scheduling step. The maximum clock cycle count resulting from scheduling all operations in all posets is T_{exec}.

6.3 Data Transfer Assignment to a Minimum Number of Busses

The next step in the synthesis procedure is to bind the data transfers to specific busses. A number of issues can arise in this binding step that can complicate control or result in using more than NB busses. Assigning data transfers to busses is a bipartite coloring problem, where the bipartite edges are the data-transfers corresponding to the edges of the SFG and its nodes are the read nodes (r-nodes) and write nodes (w-nodes) assigned to these transfers in the scheduling. Assigning each edge a color (corresponding to a bus) such that on any r-node or w-node all edges with different data values are assigned a different color (bus) is a solution to the bus assignment problem.

Although it is guaranteed from the scheduling that the maximum number of edges carrying different data values is NB, the degree of the bipartite nodes can be (NB+SK) which may be much larger than NB. In graph theory, the maximum degree of a bipartite graph

determines the number of colors used under the condition that all edges at every node conflict and hence are colored differently. We do not require that condition in our case of bipartite coloring, since the edges of the SFG with the same destination operation (mutually exclusive) or the same source operation (equal value or mutually exclusive) can share the same read or write node of the bipartite graph and can be assigned the same color (bus). Since we guarantee from the scheduling that a maximum number of conflicts at any r(w)-node is NB, we should be able to color the bipartite graph using NB colors. This is not generally guaranteed, since one can easily devise very tight bipartite graphs with only NB conflicts and (NB+SK) degree which require more than NB colors. We also have some desired coloring situations that can arise in hardware, for example, all edges that correspond to constant values may be assigned to one bus so that one ROM can be used. The problem in this case is to color the bipartite so that the following conditions are satisfied:

1. All edges sharing a read node that have their source operation a constant node should be assigned a specific bus (if desired).
2. All edges sharing a read node and having the same destination operation should be assigned one bus (color).
3. All edges having the same source operation should be assigned one bus if possible.
4. Minimum number of busses (colors) have to be used for all data transfers.

Condition 1 ensures that constants are grouped on one bus to minimize constant duplication hence ROM storage. The bus used should be different from other busses used for wrap_edges if constants are to be grouped in a separate ROM. If condition 2 is violated, then for different iterations of the inner loop or for different branches of a conditional execution, the FU executing the destination operation will have to load data in its input from more than one bus. This requires **either** the data-path controller to be iteration/condition dependent **or** a data active bit to be attached to each bus and produced by the MAG generating the address for the valid data in the current iteration. In both cases this results in control overhead, hence it is desirable to eliminate that from occurring. If condition3 is violated, then data duplication will exist in the storage (same value on two busses or more). This is specially critical for wrap_edges since they appear as one edge in the graph but in fact they represent data values in all iteration the edge is active. Data duplication for these edges is magnified by the multiplicity of *list_of_all_indices* associated with that edge which can result in significant storage over head. Condition 3 minimizes this occurrence. The last condition is an optimality condition. Note that these conditions result in a colored bipartite graph which has edges with *same* colors at the *same* r(w)-nodes.

Modified Bipartite Coloring Algorithm: The algorithm used for bus assignment while satisfying the above conditions is described in Figure 18. The terminology used in that algorithm is given in the following. A *non incident color* on a node is a color not assigned to the already colored edges or a color of an edge that does not conflict[1] with current edge. Colors A and B may be the same. Procedure **select** always tries to make A and B the same as a first choice. Procedure **Select Best** can be switched between different heuristics; a)

1. No conflict occurs if the edges have same source node, same destination node or are mutually exclusive in conditional branches.

Ordered (greedy) which skews bus transfers to re-use already used busses. This heuristic results in some busses being barely used which helps further storage optimization (section 6.7) that re-map these transfers into fewer RAMs, b) *Least used bus* which balances bus transfers on all busses which can result in equal register file sizes. c) *Random* selection helps as a comparison heuristic with a &b. Procedure **Augment graph** succeeds if A and B are the same. If A and B are not the same then a graph S that starts at the w(r)-node and contains all A and B colored edges is formed. All edge colors in S are interchanged. If S contains a path to the r(w)-node it *fails* and S is then reduced to SR by deleting all nodes having edges with only one A or one B color. This deletion reduces the size of the S graph and hence reduces the possibility of finding the starting r-node and w-node in a path of S. The algorithm attempts to satisfy the above mentioned conditions in their order.

```
Initially assume number of busses NB
Start: for {next edge in the GSFG with r-node and w-node
          with source operation Src and destination operation Dst} do
     if a separate bus for constants is required
     and if there exists Non-incident colors on r-node
     and one can select a color A from ROM colors allowed
     and one can select a color A from Non-incident colors on r-node
     then {color edge with A, go to start}
else if there exists incident colors IC on r-node of edges with same Dst operation
     and if there exists Non-incident colors on w-node
     and one can select a color A from IC on r-node of edges with same Dst operation
     and one can select a color B from Non-incident colors on w-node
     and if augment graph at w-node interchanging A B succeeds
     then {color edge with A, go to start}
 else if there exists incident colors IC on w-node of edges with same Src operation
     and if there exists Non-incident colors on r-node
     and one can select a color B from IC on w-node of edges with same Src operation
     and one can select a color A from Non-incident colors on r-node
     and if augment graph starting at r-node interchanging A B succeeds
     then {color edge with B, go to start}
 else if there exists Non-incident colors on r-node
     and if there exists Non-incident colors on w-node
     and one can select best color A from Non-incident colors on r-node
     and one can select a color B from Non-incident colors on w-node
     and if augment graph at w-node interchanging A B succeeds
     then {color edge with A, go to start}
else if there exists Non-incident colors on r-node
     and if there exists Non-incident colors on w-node
     and one can select best color A from Non-incident colors on r-node
     and one can select a color B from Non-incident colors on w-node
     and if augment graph starting at r-node interchanging A B succeeds
     then {color edge with B, go to start}
 else Increment number of busses by one then go to start.
```

Figure 18: Modified Bipartite Coloring

Note that the final number of busses used may be greater than NB. In a large number of runs with this modified bipartite algorithm (a number of multi-channel filters, FFTs and

Neural Network algorithms of different sizes) condition 1 and 2 were always satisfied, while condition 3 was violated (duplicated variables (<10%) resulted) in very special cases specifically when SK was high (near the maximum degree of the SFG). In all these cases, reducing SK resulted in satisfying condition 3 without sacrifice in throughput. NB busses is usually achieved, but in few cases SK and NM had to be reduced to achieve NB with little sacrifice (< 5%) in throughput.

This algorithm ensures that at any r-node any two edges can have same color only if they occur in the same iteration or in all iterations and have same source operation. Two edges can also have the same color if they have the same destination operation (it does not matter what their sources are as their iterations are different). ***But*** if edges have the same source operations that are at different iterations and have different destination operations that are at the same iteration and their r-node is the same, ***then*** they have to get different colors for the data to be transferred on different busses at the same time to different FUs. The complexity of this algorithm is worst case $O(|E|^2)$ where E is the number of edges in the folded graph. Our experience is that it runs on the average O(NB.|E|) It has been used with |E| = 5000 with time on Sparc1 of <10 minutes in a PROLOG implementation.

6.4 Data Storage Assignment

Each of the variables of the GSFG given by the set **E** has been assigned a bus in the previous step, the next step in the synthesis is to assign the variables to either the FU input registers, register files or RAMs.

FU input assignment: The maximum number of non-conflicting variables that can be written in a FU input register are first selected. Each of these variables gets written on a bus from a FU output and are written immediately (on that *write phase*) at the assigned FU input register. The algorithm to assign variables to FU input registers is as follows:

1. Form a subset **S** of the variables **E** that are read by a FU input. Initialize the ordered set of variables assigned to that FU input register to **P**={}.
2. Sort **S** on the left edge of the variable life-time. Initialize read time R= zero.
3. Remove the first variable **V** from **S** to get **S'**, if 'write time of **V**' is greater than or equal to R then add **V** to **P** and update R='read time of **V**' else if the 'read time of **V**' is less than R then replace the top (last entry) of **P** with that variable and update R ='read time of **V**' else go-to (4).
4. S = S', if S={} go-to (5) else go-to (3).
5. Go to (1) until all FU input registers are assigned.

Step 3 ensures that all variables with single cycle life times and the maximum *number* of variables with the life time that can use a FU input register are selected. It also ensures that a value written in the FU input register is not over-written by another value before it is consumed by the FU. Subtracting the final **P** from **E** gives the set of variables **ERM**.

RAMs and Register File Assignment: Each of the set of variables **ERM** is further assigned to either a RAM or a register file on each bus as described next. Input and output

variables are assigned to the allocated input and output ports. All wrap edges (variables and constants) are assigned to RAM. *Inner_edges* (variables and constants) are assigned to the register files.

Splitting the data between RAM and register files when dealing with algorithms requiring large data storage is essential for the following reasons:-

1. *Inner_edge* variables (non-Separator) are re-used every iteration, hence they account for a very large number of accesses to storage in the execution of the entire iterations although they use few locations. Assigning them to register files reduces memory accesses which allows prefetching of variables, that in turn can result in merging and reducing the number of parallel RAMs required in the architecture (Prefetch and merge explained later).
2. *Inner_edge* variables can have life-times as short as 2 cycles but *wrap_edge* variables have larger life times typically greater than $T_{exec.}$ Having variables with large life-times facilitates memory interleaving (an issue beyond our scope here). This is specifically interesting for image processing algorithms which require very high storage and speed.

An example of the scheduling, bindings and register file or RAM assignment for the 24 channel telecommunication filter is shown in Figure 19 where the bipartite coloring for the edges of that filter is also shown. The architecture uses a two stage pipelined multiplier,

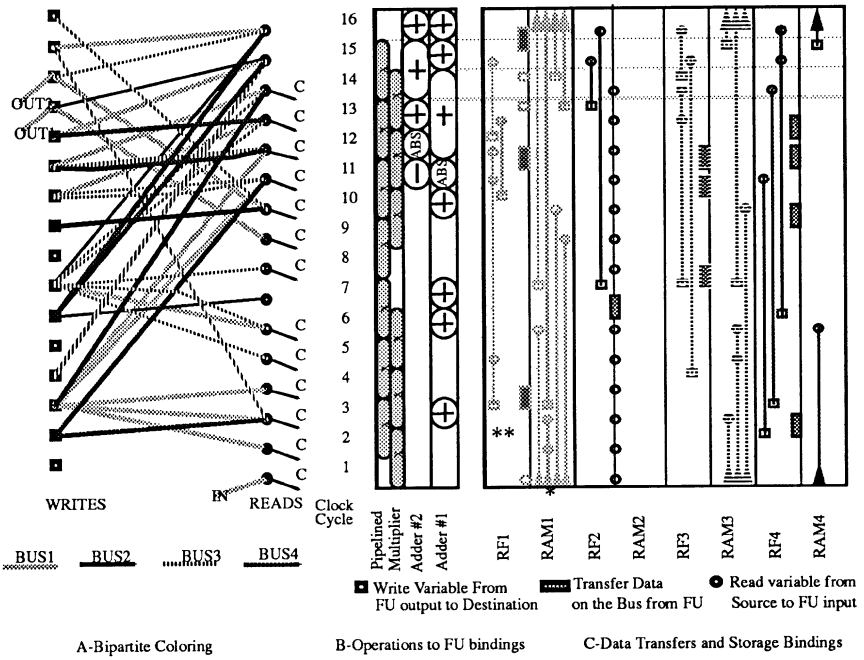


Figure 19:Scheduling and Binding For Telecom Filter

two adder units (one cycle) and four busses. The number of cycles used in one channel iteration is 16, which is repeated 24 times for the different channels. Notice, for example, that the output of the first addition has an out-degree of five, at write clock 3 and read clocks (2,3,4,11,12) all have been assigned to Bus-1. A copy of that variable is assigned to the RF1 (marked ** in Figure 9 and 19) to be used in the same cycle, and another copy to RAM1 (marked *, corresponds to the wrap_edge of Separator type) to be used after 23 other iterations for the rest of the channels. Figure 19.b shows also that an operation can take more than its minimum required time if busses are not available. For example, on adder#1, between clock 12 and 14, an addition takes 3 cycles (rather than one) because of bus contention, as evident from the bipartite graph (Figure 19.a) for these cycles.

Register file optimization: A variable is alive between its write time and its read time. The variables with non overlapping life-times in each RAM or register files are grouped together to re-use a single register storage. The register minimization problem in general is recognized as a circular arc graph (CAG) coloring problem rather than an interval graph coloring as some synthesis tools advocate. The reason for this is that Separator edges or wrap edges need to store the variables from one sample period or iteration to the next. The life-time of these variables have a read clock time which is less or equal to the write clock time during one iteration. The coloring of the CAG is an NP-Complete problem while the interval graph can be colored optimally in polynomial time O(E*log(E)) [21]. For example, in Figure 20 data storages are shown, where variable A (inner edge) does not overlap with register B (Separator or wrap edge). Typical synthesis systems would cut all separator edges and assign each a register and then color the inner edges as an interval graph

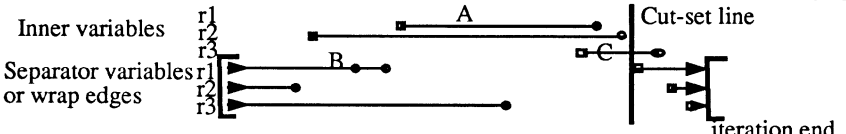


Figure 20: Variable Life times form a CAG

with the left edge algorithm where for this example six hardware registers would be allocated; three for the separator variables and three for the inner variables.

The heuristic used in SPAID-X results in a number of colors which satisfies the best known upper bound for the colors, which is equal to the size of the minimum clique plus the size of the maximum clique of the CAG [24]. The heuristic for coloring is done by finding a minimum clique and cutting it from the CAG (in our example the clique is the single variable C). The remaining graph is an interval graph that starts from the cut-set line and loops around the iteration end and back to the cut-set line. This is colored optimally using the left edge algorithm. The cut-set (variable C) is then colored in a non-conflicting color. This procedure is O(E*logE), but it results for our example into allocating only 3 registers as indicated with the number assignments in Figure 20.

The left edge algorithm can be summarized as follows.

1. Assume that an infinite number of registers are available. Each register I is assigned a variable R^I_{last}=0.
2. Sort variables on the write time into a list L., O(E*LogE).

3. For each variable (in order) in L with write time W and read time R assign a register I such that its $R^I_{last} < W$, update $R^I_{last} = R$.
4. All registers with $R_{last}>0$ are allocated in the hardware.

The procedure for finding the minimum clique, proceeds by finding for each clock cycle the number of registers that overlap and runs in $O(T_{exec}*E)$. The minimum overlap is found in O(E) by a simple search. Then a renaming of reads and writes is done in O(E) to prepare for the left edge coloring.The left-edge coloring is done as above in O(E logE). The cut-set is colored by matching each variable of the cut-set to the set of empty intervals of the colored interval graph. Each member of the set of empty intervals is defined by the first write clock and last read clock of each allocated register in the interval graph. The matching is done by generating a bipartite graph with two sets of nodes, the first are the variables of the cut-set and the other is the set of empty intervals. Its edges define a compatibility relation when the variable fits in the empty interval. A bipartite maximal matching algorithm (using augmenting paths) is used and runs in $O(R*CE^2)$ where CE is the number of variables in the cut-set and R is the number of registers allocated for the interval graph.

Mapping RAM Access to Ports: To demonstrate how prefetch of RAM data works to allow for re-mapping of RAM data transfers from the busses to the ports, we show the RAM data transfers for the FFT inner loop (for a *double butterfly*) before mapping in Figure 21.a where 4 busses with direct interface to RAMs are used. The value of X (depth of FIFO Figure 14.b) is 2 for this example. Figure 21.b shows the port data transfers after mapping bus transfers to two ports connected to two RAMs through memory interface as in Figure 21.b.

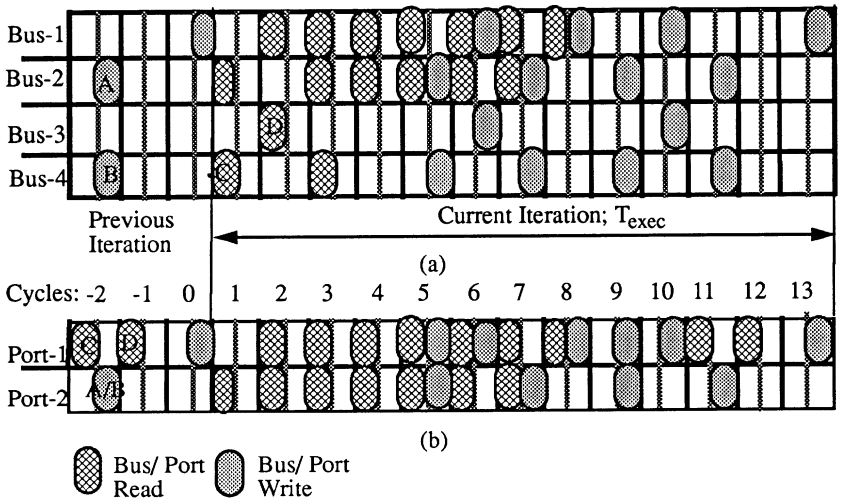


Figure 21: Memory accesses from the busses (a) and re-assignment of the memory accesses through the ports (b).

Note that memory access on bus 3 and 4 are under utilized. All the reads and writes have been re-mapped on empty clocks (example; memory reads C and D in Figure 21) or on clocks which carry equal value data (for example, memory writes A and B in Figure 21) or are mutually exclusive (used in different instances of the inner loop iteration). Note that memory reads C and D have been prefetched in the previous iteration. The data resides on the bus FIFO until it is read from the bus.

The algorithm that is used to perform the re-mapping and merging of the data transfers on NB busses through ports to NM memory modules is given next, where Z is the minimum data life time and a set of ports **Ps** equal in number to the busses is initially assumed:-

1. Sort the busses such that the bus with largest number of memory accesses is first.
2. Sort all variables assigned to memory on each bus according to their read times (smaller read times first).
3. Initialize the FIFO read cycles R^I_{FIFO}, on each bus Bus-I, to a large negative number.
4. For each bus Bus-I (in the sorted order) and for each read time Rd on Bus-I (in the sorted order) do the following:- {

   Find all variables Vs that have the same read time Rd.

   Find the maximum W_{max} of the set Ws of all the write times of Vs.

   Find the first clock X^I_{FIFO} that does not fill the FIFO (with depth X) on bus-I.

   Set a read clock limit RL as the larger value of: X^I_{FIFO}, R^I_{FIFO} and $(W_{max}-T_{exec} + Z)$.

   Select a port **P** from **Ps** until a port is found that satisfies the conditions: (a) an empty or non conflicting write clocks exist for all Ws AND (b) *an empty prefetch read time* **p_rd** exists between RL and Rd inclusive; THEN re-map all variables in V to **P** with new read time **p_rd** and set R^I_{FIFO} = **p_rd** + 1. }
5. The number of ports used is equal to the number of memory modules NM.

The writes clocks with variables that have equal value or are mutually exclusive do not conflict. Also note that the selection of ports is done first on the used ports and if non of these ports satisfies conditions (a) and (b) then an unused port is selected. There are two options provided when selecting a used port*; 1) ordered(greedy)* which selects the first port in a specified order which usually results with less number of ports. 2) *least_used_port* which keeps track of the used ports and selects the one with least number of read and write access, this tends to balance RAM sizes.

The algorithm running time is O(|E|.NB.NM) as |E| variables are re-mapped and all busses and ports may be exhaustively tried. The algorithm is guaranteed to find a port assignment since the worst case is to have no reduction in number of RAMs. Note that in Step 4, in addition to a *an empty prefetch read time* a non-conflicting read time could be used (non-conflicting on read means that data are mutually exclusive, i.e. having the same destination operation in the folded graph). Although this is more flexible and may result in better merging, it requires the main controller (for the memory interface) to be iteration dependent to redirect in each iteration the ports to the appropriate bus.This is disallowed in order to achieve an iteration independent main controller. With this restriction in step 4, for each

clock cycle only the same bus is connected to the same port in all iterations. Note that the resulting merging into 2 RAMs is optimal, in a sense that 17[1] different memory reads in 13 clock cycles cannot be done using only one RAM.

6.5 Main Controller Specification

The main controller shown in Figure 22.a has a control word that is generated every system phase (read phase Φ1 and write phase Φ2 of the system cycle) and consists of the following *fields:*

1. 1. Control for the data-path consisting of: a-Address for each register file. b- Control for each FU input and output bus selector and FU mode. c- Control for memory interface.
2. 2. A control word for each MAG (*MAG_field*).
3. 3- A next address generation (NAG) field consisting of: a- Branch status bit (branch/ increment). b- A next address generation offset index.

The clocking is done with two non overlapping clocks C1 and C2 as shown in Figure 22.b. Level sensitive latches are assumed. A system cycle contains two C1 or C2 clock cycles.

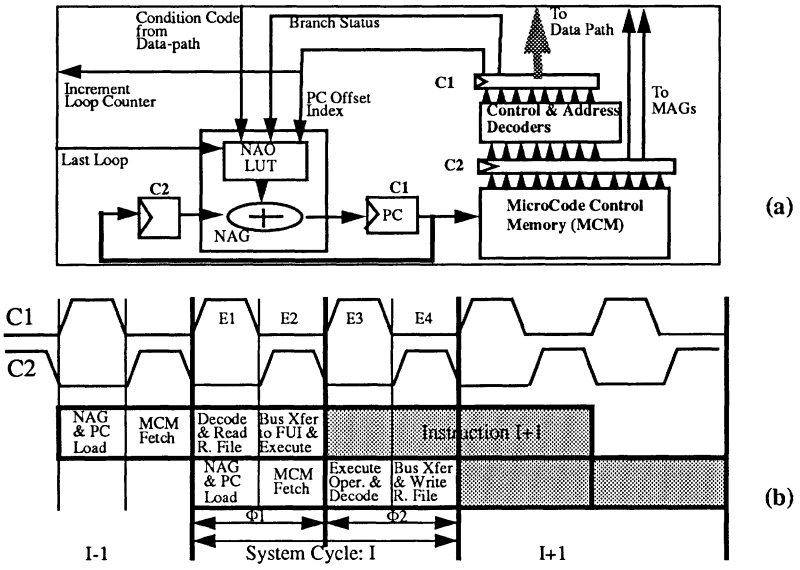


Figure 22: Main Controller (a) and clocking scheme (b)

A system cycle is divided into four events (E1-E4). For the current system cycle executing the current set of data transfers instruction (I), the fetch of that instruction begins in (E3,E4) of system cycle I-1 and the data transfers complete in system cycle I (E1-E4).

1. 16 different reads is more accurate, since 2 of these reads are mutually exclusive.

Instruction I overlaps with instruction I+1 first control word fetch as shown in Figure 22.b. The details of the execution in each event is also shown.

Note that in event E2, the data read from the register files or RAM interface is latched in the FU input registers. The operation starts executing once data are stable on the FU input. For a one cycle operation, the execution spans part of E2, E3 and E4. In E4, the data propagates on the bus and gets written on the falling edge of C2 (end of E4) in the register file, RAM interface or FU input register. For an N cycle operation, an extra N-1 system cycles are added to the execution time of the operation.

For non-conditional or non-looped execution, the next address generation (NAG) block simply increments the current PC. In that case, the next address offset look up table (NAO-LUT) produces an increment value of 1. In the last instruction of a loop and if *Last Loop* is not active, the *branch status* indicates (branch) and the *PC Offset index* indicates an index to the negative branch offset to repeat the loop. If *Last Loop* is active then NAO-LUT produces a branch offset to the code of the next Block Node in the hierarchical GSFG (could be simply an increment of 1). A similar operation is done for conditional execution. When the *branch status* indicates (branch), the *Condition code* and the *PC Offset index* determine the branch offset to the next code. The encoding method for the *PC offset index* depends on the code address mappings in MCM and the details are beyond the scope of this chapter.

The choice of the *MAG_field* encoding can affect the MAG controller size. We use a multi-valued symbolic encoding for this *field.* On the read phase either the destination operation input name (*Dst_name*) or one of the mutually exclusive edge names (*edge_name*) can be used as the symbol. On the write phase either the source operation output name (*Src_name*) or one of the edge names can be used as the encoding symbol. The use of either the operation names or the edge names can result in less number of symbol names depending on the GSFG **G**. Since the number of symbols affect the main controller size as well as the MAG size, we have the option in SPAID-X to generate both.

6.6 MAG Controller Specification

The memory address generation (MAG) for the RAM connected to each port is a combinational logic block. The *input--specification* of the MAG have the following fields as shown in Figure 23:

1. 1-Read/Write (Φ1) bit.
2. 2- Loop index fields.
3. 3-A symbolic MAG field from the main controller (as explained above).

The *output-specification* is the corresponding encoding of the address location of the variable in the RAM. A symbolic encoding (*ram_location*) is generated for the output specification. The MAG can be designed as multiple level logic network, a PLA, or as a user defined specific address arithmetic unit (AAU) that maps the Indices and the MAG field to a RAM address. When such an AAU is used then it is assumed that a specific mapping is available from the variables to an encoded address space. In absence of a user defined

MAG, logic synthesis procedures are used to generate the logic block. The *input-specification* is generated for both the read and write phases for each wrap_edge in G and for each member of the *list_of_all_indices* of that edge.

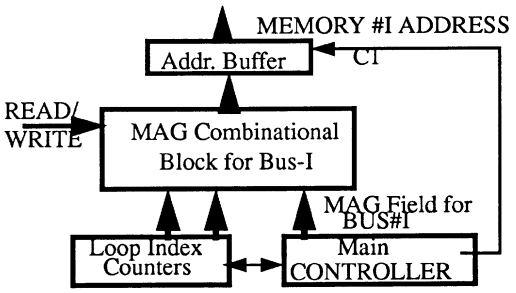


Figure 23:Interface of Memory Address Generation (MAG) for Port-I

MAG Synthesis: The MAG is defined as a list of elements = [input-specification, output specification], one for each read and write operation to the RAM. Each element is detailed into symbolic fields; for example for the FFT example with two Indices:- [[R/W, I_index,J_index,MAG_field], [ram_location]]. We have applied the tool NOVA [15] for optimal encoding of the MAG_field symbolic names. All other symbolic fields were heuristically assigned to ascending bit encoding starting from an all zeros code. The MAG logic is then synthesized using misII [25] to obtain a multi-level logic implementation.

6.7 RAM Optimization

Initially we flatten all the wrap_edges of G. This is done by generating for each wrap_edge and for all members of *list_of_all_indices* of that wrap_edge, a set of variables with different life times (read and write clocks) calculated knowing the sequence in which the Indices are generated and T_{exec}.

RAM optimization does not essentially imply that only the number of locations is minimized. External or embedded fixed module RAMs have typically 2^m locations, therefore the number of locations used have to be compared in that context. Moreover, when a set of data variables **S** with non-overlapping life-times are *grouped* into a location with a symbolic encoding (*ram_location*) then for this *output specification* of the MAG a number of input-specifications with different indices corresponding to the different variables sharing the location are generated. The best grouping of variables does not only depend on decreasing the RAM locations but also on decreasing the MAG size. The MAG size can decrease if large sub-cubes of the input-constraints can be found[14]. Because of such inter-relationship between the variable assignment to location, address encoding and input encoding, the problem of minimizing both the RAM and MAG size is quite complex.

We propose two variations in variable assignments to RAM locations which have resulted in a good improvement in the size of the generated MAG. The two alternatives are: 1) Restrict all variables that have the same source operation output in the graph to be minimized together. 2) Restrict all variables that have the same destination operation input in

the folded graph to be minimized together. If the MAG field symbolic encoding uses Src_-name and Dst_name then these restrictions ensure that there is more likelihood that the *input-specification* with the same Src_name or Dst_name would share the same location (*output-specification*) which reduces the number of product terms in MAG after logic minimization. The procedure for memory minimization that takes these restrictions into account is based on the previous left-edge algorithm. The procedure in addition to checking for no-overlap of the life-time of variables that share a *ram_location*, it also checks for these variables for a common source operation (*src restricted heuristic*) or a common destination operation (*dst restricted heuristic*) to allow merging them in one *ram_location*.

It is also possible to define a memory location assignment and an encoding for the variables based on a single memory space assumption. For an architecture with more than one RAM, the same memory space is mapped on all RAMs. This results in redundant memory locations but it is assumed that such an assignment has a corresponding MAG architecture with efficient size. This is true especially for large address space RAMs as has been used in the Cathedral tools. Nevertheless, automating this step efficiently remains a point of future research.

Re-Synthesis of port mapping and code assignments: We have found that when the difference between the encoded cover resulting from the NOVA encoding is much larger (> 50%) than the multi-valued cover that there is always room for improvement in the size of the MAG by changing the heuristics in three areas:-

1. The port assignment heuristics (*greedy* or *least_used*),
2. The MAG field symbolic name choice (*edge_name, Src/Dst_name*),.
3. RAM location symbolic assignment heuristic (*Src or Dst restricted*).

Moreover the number of heuristic choices for NOVA itself and the other symbolic fields is large which presents another dimension for re-synthesis. In Section 7, we show some results for the application of these heuristics.

7.0 EXPERIMENTS USING SPAID-X

7.1 Single and Multi-Channel Filters

Multi-Channel filters:- SPAID-X synthesis results are first demonstrated for the synthesis of multi-channel filters. The first example, which is a 24 channel filter from an industrial telecommunication application, is shown in Figure 7.b. The second example is the elliptic filter bench mark shown in Figure 7.a which was modified to work on 24 channels.

For 24 channels, the unfolded filter is 24 parallel filters. Hence, the folding is trivial and results in the same filter with only the separator edges being folded and replaced with wrap_edges with a proper *list_of_all_indeces*. We synthesized both filters for a large number of architectures.The architectures synthesized are based on using a pipelined multiplier with a delay of 2 cycles and initiation every cycle and an adder that executes an addition in one cycle as our FUs.

		SPAID_X									
	ADDER/SUB/ABS	1	1	1	1	2	2	2	3	3	3
	MULTIPLIER	1	1	1	1	1	1	2	2	2	2
	BUSSES *	1	2	3	4	4	5	6/4	4	6/5	7/5
ELLIPTIC	RAM Distribution**	7	3,4	3,4	2,5	2,5	2,5	1,3,3	2,5	1,3,3	1,3,3
	# of RAMs/Ports	1	2	2	2	2	2	3	2	3	3
	Registers in RegFiles	9	11	12	13	9	11	11	7	11	8
	Mux.Ips/Bus Drivers	4/2	7/5	10/5	12/7	18/11	19/14	27/18	26/16	33/20	31/21
	Sample Period	68	34	31	29	20	18	18	20	17	16
TELCOM	RAM Distribution	8	6,2	7,2	5,3	5,3	5,3	5,3	5,3	5,3	4,2,2
	# of RAMs/Ports	1/1	2/2	2/2	2/2	3/2	3/2	3/2	3/2	4/2	4/3
	Registers in RegFiles	11	11	13	12	9	8	10	9	7	7
	Mux.Ips/Bus Drivers	4/2	8/4	10/6	11/6	16/10	14/10	21/14	26/11	26/16	26/16
	Sample Period	50	25	21	19	16	16	15	14	13	13

TABLE 1. Multi-Channel Filter Architecture Search.

Table 1 shows a large assembly of results obtained for these filter. In this table, for the number of busses row (marked *), the second number indicates the maximum number of busses used for the telecom filter while the RAM distribution row (marked **) the number of words per RAM *per channel* of filter is shown. Note that the row with the number of ports shows that a reduction of the number of RAMs by prefetching is always achieved. We have used SK values for these filters between 3 and 7 to get best results. In some instances, where SK is large, an extra bus above NB would be required to find a bipartite coloring. We used NM between 2 and 4 to achieve the above results. For NM=2, more cycles (1-3) are sometimes added to the T_{exec} of the case with NM=NB. The number of ports achieved is always less than or equal to NM+1. For example, the elliptic filter with 24 channels and a data-path with 2 adders, one multiplier and four busses, two ports are used each connected to a RAM. The number of locations in RAM1=120, and in RAM2=48 which is the minimum. The size of the MAG for RAM-1 was 119 product terms and for RAM2 was 8 product terms. For all the examples shown in Table 1 the minimum number of storage locations were achieved. This means that no data duplication was present.

Design space search: In order to appreciate the design space search provided by a synthesis tool like SPAID-X, Figure 24 demonstrates the Area versus T_{exec}, Area versus Area*T and $A*T^2$, trade offs for the above two 24 channel filters. The assumptions are that the multiplier area is five times the adder area, the adder five times that of a register in the register files and a register in the register files costs twice a memory location in the RAM. The figures are normalized to the minimum architecture. In spite both filters having largely different structures, the same optimum architecture (in AT^2 sense) with 2 adders, one pipe-

lined multiplier and 4 busses resulted. With actual areas and delays, such a design space

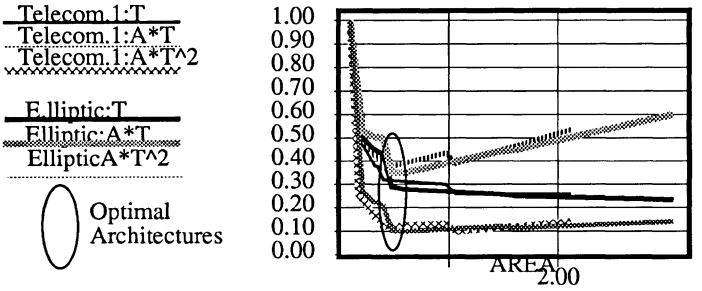


Figure 24: Area-Execution time Trade-offs in 24-channel Filters

search can actually be used to decide on the best architecture for a core data-path to be added and supported in a VLSI design library.

Unfolding transformations: An unfolding of k=2 was applied to the telecom filter. For an architecture with one adder, one multiplier and four busses, the unfolding transformation reduced the execution time from 38 cycles to 35 cycles for two consecutive iterations. Appreciable T_{exec} reductions were not achieved without the use of subsequent local transformations. For the elliptic filter, for the architecture with 3 adders, 2 multipliers and 6 busses, unfolding (k=2) and local transformations resulted in a 15 cycle solution.

Single channel comparisons with random topology architectures: Although the results shown in Table 1 are for the best architectures for a multiple channel filter, the results for a single channel filter are very similar. As other work did not consider multiple channel implementation, we compare only our inner loop T_{exec} (corresponds to one channel execution time) in Table 1 with others. With ALPS[12], which optimally schedules the operations, we obtain identical results for all reported architectures except that for an architecture with 1 multiplier 1 Adder and 4 busses we get 29 clock cycles and ALPS 28 cycles. ALPS achieves 17 cycles for 2 adders and 2 multipliers, we get 17 for 3 adders and 2 multipliers and 18 with 2 adders, 1 multiplier and 5 busses. The other aspects of the design such as the number of mux inputs, tri-state outputs or the number of registers, is very difficult to compare since the numbers mean different things for different architectures. For example, the number of registers in the register files in the SPAID-X architecture correspond to a RAM location[1] while in the random topology architectures these registers are pipeline registers[2], which probably have more than twice the area of the equivalent RAM storage for the same number of bits. Another factor affecting the storage area is the FU input registers. To give an indication of how storage performance of a SPAID-X architecture compares with a random topology one, we compare the architecture for a 2 adder, one multiplier, 19 cycle solution by HAL [9] to the SPAID-X architecture (without the memory interface) for a single channel elliptic filter. In HAL, 12 pipeline reg-

1. The register files in SPAID-X are implemented such that each bit storage requires only one latch. If a large number of registers are assigned to a file a fast RAM structure is used.

2. In the pipeline register case each bit of storage requires two latches. A separate input and output connection is required for the register.

isters were used. In SPAID-X, 11 register file locations are used in addition to five FU input registers. If one takes a RAM location to be one half a pipeline register area and a FU input register to be equal in area to a pipeline register, then the storage area is better for SPAID-X (by about 1.5 pipeline registers).The SPAID-X topology is shown in Figure 25. A major aspect of comparison in a bus based architecture is the number of busses and the

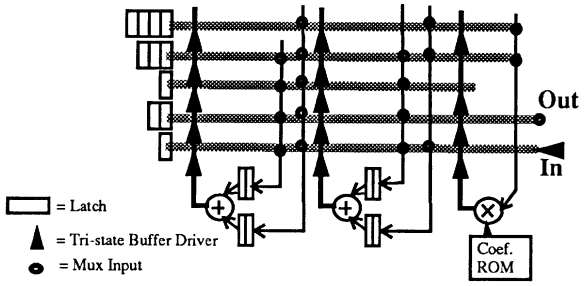


Figure 25: SPAID-X Linear Topology Architecture For a Single Channel Elliptic Filter Example

maximum bus loading. In SPAID-X, five busses are used with a maximum loading of 5 tri-state driver outputs and 6 inputs, in HAL 6 busses are used with a maximum loading of 7 tri-state outputs and one input. Typically, a tri-state output contributes much more than an input (3-10 times the capacitive load). Also, the maximum fan-out from a FU or register in SPAID-X is 5 while in HAL it is 7. These loadings reflect on the maximum clock speed. The total number of tri-state drivers in SPAID-X is 19, while in HAL it is 26. HAL uses an extra bus, while SPAID-X requires 5 muxes with 19 inputs. Registers are grouped into register files for the SPAID-X architecture, this reduces storage area but may add to the register access time as compared with the HAL architecture. Using the number of global nets (each global net has the full word-width with a fan-out of two or more) to determine the regularity of the architecture; the HAL architecture uses on top of the 6 busses 7 global nets while the SPAID-X architecture does not have any global net other than the 5 busses.

7.2 A Stereo Hi-Fi Filter Chip Implementation

In order to assess an architecture efficient layout, an actual layout implementation is essential. A VLSI chip was designed for a stereo HiFi wave digital filter using the SPAID-X architecture [23]. Equivalence transformations were used on that filter to replace all constant multiplications with CSD shift/add/subtract operations. With this transformation all multiplications were eliminated. A design space search was carried out, and based on the filter specification a 20-bit data path with two adder/subtracter FUs and one barrel shifter (with limited number of shift counts) were found to be sufficient in a two bus system. The amount of storage on each register file which is connected to each bus required for the data-path to support a number of different filter orders and sampling rates, was found to be 16 words. Hence, a total of (2*16*20=640 bits) storage is required. The I/O ports were required for interleaved input and output of both channels (16 bits). The chip was implemented in a 3-micron technology using pre-charged busses and static RAMs for

the register files. The clock frequency for this architecture was 14MHz. The data-path was bit sliced, where the two busses were routed over the cells. Full abutment was possible for all the data-path, which resulted in a very compact design. The chip floor plan is shown in Figure 26, and gives an indication of the regularity of the multiple bus architectures with an active area less than 10 mm^2.,

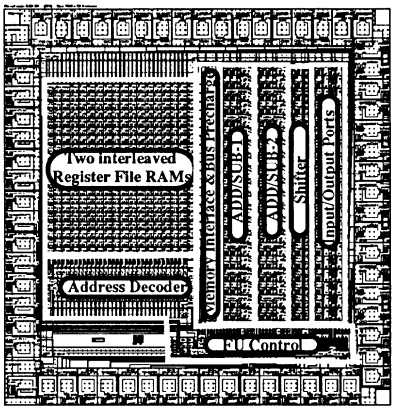


Figure 26: Chip Floor Plan For a Two Bus Data-Path for a Stereo Hi-Fi Filter Chip

7.3 The FFT Example

This example demonstrates the handling of folded loops by SPAID-X. We also show that the size of the MAGs is reasonable for small/medium FFT sizes (up to 64 points). The folding was done by a PROLOG program that automatically generates the folded graph of any N-Point FFT. All the results shown are for the case of the double butterfly folded graph. The total number of cycles to execute the FFT is T_{total}=(N/4)*ln2 (N)*Texec, where N is the FFT order. For a 256 point complex FFT, CATHEDRAL_II [13] produces an architecture that runs for 16,903 cycles using 2 multipliers with accumulators, 3 separate RAMs and 4 AAU (address arithmetic units for RAM address computation). In SPAID-X, for a data path with 2 adders and one multiplier, 2 RAMS for the storage and one (RAM or ROM) for the sine table we obtain T_{exec}= 13 and T_{total}= 64*8*13cycles=6,656 cycles. The hamming window is done in SPAID-X in 2*N cycles, which is done in conjunction with inputting the input-vector. The total execution time for an 256-FFT with windowing is 7,168 cycles.

Table 2 gives detailed results on the size of the RAMs and controller size of MAG. For this table, two adders, one multiplier, four busses, one ROM, and two RAMs are used. The value of SK=3, NM=2 and the resulting T_{exec}=13cycles. The number of product terms after using NOVA with default heuristic for state assignment is shown. Best results are shown after using different heuristics in port selection and symbolic encoding. From Table 2, it is evident that the number of storage locations used is more than the minimum (2*N). For the 16-point FFT the extra storage is <50% and <25% for the others.

To show the effect of re-synthesis on reducing the MAG size, the 16 point-FFT example was re-synthesized with the option in RAM minimization (*src restricted*) allowing only the variables with same source node to share a location. The numbers between parenthesis

			MAG of RAM-1		MAG of RAM-2		MAG of ROM	
FFT size	RAM1 #locs.	RAM2 #locs.	NOVA# PT	misII gates|area	NOVA #PT	misII gates|area	NOVA #PTs	misII gates|area
16	29	20(16)	92(115)	198| 517	18(76)	28| 70	27(27)	35| 88
32	47	32	291	697| 1755	31	46| 109	54	100| 264
64	95	64	702	1453| 3056	33	44| 100	94	215| 503
128	187	128	1715	N/A	50	N/A	158	N/A

TABLE 2. : FFT RAM Storage and MAG size

indicate the numbers before reduction due to re-synthesis. In table 2, the size of the MAGs after applying the default "script" with misII and mapping to "mcnc.genlib". From the size of the MAGs it is clear that this technique is viable for small/medium size FFTs. For larger FFTs, the NOVA/misII encoding and synthesis is quite inefficient and better approaches should be devised.

8.0 Conclusions

In this chapter, we presented an architecture and synthesis methodology useful for signal processing algorithms requiring large data storage. Our approach is novel in terms of addressing both the issue of the memory hierarchy in high level synthesis with automated assignment of variables to multiple memories as well as data-path architecture synthesis with optimal throughputs. It was shown that the multiple bus architectures have excellent layout regularity properties for VLSI implementation and that algorithms with large storage are most suitable for this style of architecture, especially if the sampling rate is lower than the architecture clock. Through a number of practical examples, the design space search by using SPAID-X was shown to be useful to determine a range of architectural choices. The design space for a number of similar applications can also be used to determine an optimal core data-path module that can be added to the current library, as we demonstrated with the Hi-Fi stereo data-path.

The memory address encoding problem was attempted and a number of factors were shown to affect the design of memory generation. Examples showing these interactions were given and some heuristics were shown to obtain good results. This problem of mixed data-path and control synthesis is still under research.

We have also briefly introduced a number of behavior transformations that are essential for any signal processing implementation environment. It was shown that by applying such transformations better throughputs and efficient architectures can be realized.

The design experience gained from implementing the multiple bus architecture, has made us conclude that register file access time and bus delay over-heads become a significant factor in determining the system speed. Fortunately, implementation techniques exist for speeding up these delays by applying pre-charging and fast sensing techniques which have been applied in a number of commercial processors that are predominantly bus based architectures. An advantage of the multiple bus architecture is that large fan-out (global) networks other than the busses do not exist, which is not the case with random topology architecture. Large fan-out networks can result in large interconnection area over head as well as difficult estimation of the delays (and delay compensation) prior to floor-plan assignments. When a large number of variables are handled in random topology architectures, the number of global networks explode resulting in less area efficient layouts. On the other hand, for small storage, highly pipelined algorithms requiring sampling rates close to the minimum achievable clock rate in that technology, then the random topology architectures are most suitable.

The multiple bus architecture was also used in neural network digital implementations, which was not presented in this chapter and is beyond its scope. Details of the issues involved could be found in [16]. It suffices to mention here that the multiple bus architecture was used in conjunction with complex, deeply pipelined FUs, that closely resemble random topology style data-paths, for such an application in order to achieve high efficiency of utilization and throughput as well as handling the large storage required by these algorithms. Another factor that comes to the advantage of using busses, is the use of advanced technologies and techniques, such as BiCMOS and low logic swing, which allow having fast delays on highly capacitively loaded busses with low power dissipation.

References

1. F. Catthoor, H. J. De Man, "Application Specific Architectural Methodologies for High Throughput Digital Signal and Image Processing", IEEE Tr. on ASSP, Vol 38, pp339-349, Feb. 1990.
2. D. Lanneer et al, Synthesis of medium and high rate signal processing applications with the new Cathedral environment", ACM/IEEE HLS workshop, 1989.
3. B. Haroun, M.I. Elmasry, "Architectural Synthesis for DSP Silicon Compilers. IEEE Tr. CAD, vol.8, p431-447., April 1989.
4. Baher Haroun "VLSI Architectural Synthesis for DSP Custom Applications" Ph.D Thesis. University of Waterloo, 1989.
5. M. Potkonjak J. Rabaey, Optimizing Resources Using Transformations, Proc. ICCAD-91, pp. 88-91. Nov. 1991.
6. Lin et al, " Efficient Microcode arrangement and Controller Synthesis for ASICS" Proc. of ICCAD, p38-41, Nov. 1991.
7. N. Park, A. Parker " Sehwa: A software package for synthesis of pipelines" IEEE Tr. CAD, Mar. 1988, pp.356-370.

8. B.M. Pangrle, D. Gajski," State synthesis and connectivity binding", Proc. of ICCAD-86, pp210-214.

9. P.Paulin, J. Knight "Force Directed Scheduling. for ASICS", IEEE Tr. CAD, vol-8 June 1989, pp661-679.

10. F.D. Lanneer, et al., "Synthesis of the Pitch Extractor FFT and 5th Order Elliptic filter on Cathedral " ACM/IEEE HLS workshop, 1989

11. M. Breternitz Jr., J. P. Shen, "Architecture Synthesis of High Performance Application Specific Processors", Proc of 27th DAC, pp.542-548, June 1990.

12. C. Hwang, J. Lee, Y Hsu, "A formal approach for scheduling in High Level Synthesis" IEEE Tr.CAD 1991, pp464-476

13. G. Goosens , J. Vandewalle, H. Deman, "Loop optimization in Register transfer scheduling for DSP"Proc. DAC 89, pp826-821.

14. S. Note et al. "Cathedral-III: Architecture Driven HLS for High Throughput DSP Applications" Proc. 28 th DAC, pp597-602. 1991.

15. T.Villa A.S.Vincentelli"NOVA: State Assignment of FSM for Optimal Two-level Logic implementation" Proc. DAC89p327-332.

16. B. Haroun, E. Torbey, "Synthesis of Multiple bus/Functional Unit Architectures Implementing Neural Networks", Proc. ICCD-92, Oct. 1992, to appear.

17. O. Buset, M. I. Elmasry "ACE: A hierarchical Graphical Interface for Architectural Synthesis" DAC-89, IEE/ACM, Jul 1989, pp 537-542.

18. "Signal Processing Work System SPW and Hardware Design System" Comdisco Systems, Inc.

19. E. Lee et al, " Gabriel: A Design Environment for DSP", IEE tr. ASSP, vol. 37, pp 1751-1762, Nov. 1989.

20. F. Rose, C. Leiserson, J. Saxe "Optimizing Synchronous Circuitry by retiming", Proc. of Caltech Conf. on VLSI, pp 41-67 Computer Science Press, 1983.

21. M. Golumbic "Algorithmic Graph Theory and Perfect Graphs", Academic Press, 1980.

22. K.K. Parhi, D.G. Messerschmitt " Pipeline Interleaving and Parallelism in recursive Filters Part-1: Pipelining Using Scattered Look ahead and decomposition" IEEE tr. ASSP, Vol. 37, pp1099-1117, July 1989.

23. B. Haroun et al. "VLSI Arch. Synthesis and Implementation of HIFI Digital Filters", Proc. CCVLSi, pp.107-114, Oct. 1989.

24. A. Tucker "Coloring a family of Circular Arc Graphs" SIAM. J. of Applied Math. 29, pp493-502, 1975.

25. R. Rudell et al. "MIS: A multiple level logic optimization system", IEEE Tr. on CAD, pp.10062-1081, Nov. 1987.

4

Exploring The Algorithmic Design Space using High Level Synthesis

Miodrag Potkonjak
C&C Research Laboratories, NEC USA, Princeton

Jan Rabaey
Dept. of EECS, University of California, Berkeley

HYPER is a third generation high level synthesis system, targeted at numerically intensive applications. By shifting the emphasis from the traditional high level synthesis tasks (such as scheduling, assignment, allocation and module selection) to the domain of optimizing transformations, new avenues for high level synthesis are opened. One of the most exciting among them, which probably has the largest impact on the quality of the design, is to select and optimize the algorithms for a given application.

The paper starts with a brief overview of the HYPER system, with special stress on the optimizing transformation methodology. After this, the paper concentrates on the exploration of the algorithmic design space. We show how HYPER can be used to guide and conduct a proper algorithmic selection process and how transformations in HYPER can improve the performance or cost of real life applications with orders of magnitude.

INTRODUCTION

Motivation: Why yet another higher level synthesis system?

High level synthesis has emerged as one of the most exciting and most challenging areas in CAD. It is both eagerly and reluctantly expected by the design community. Eagerly since it promises an increase in productivity, it opens new venues for designers and it relieves them of many mundane and tedious tasks, which are currently undertaken manually; reluctantly since it dictates a dramatic and drastic change in the way the design process is conducted. We have currently reached the point, where high level synthesis has to prove itself as one the most important and most influential CAD tools in design, or will slowly disappear as an exciting, but also dead end academic research avenue.

While, of course, its destiny will ultimately be determined by the effectiveness and efficiency of the tools in industrial designs and their viability to compete with the manual design process, it is very instructive to look at the genealogy of high level synthesis. This approach enables us to fully leverage on already achieved successes and achievements and avoid some of the dangerous pitfalls, which prevented it from becoming a standard engineering tool at present.

The identification of different generations in technological developments is an uncertain undertaking and a classification according to different criteria appears often to be more natural. However, it seems that until recently there have been two major generations of high level synthesis systems. The first generation was developed by the researchers from the computer architecture community [Bar73]. Although initially targeted at RTL level synthesis, the focus switched later to high level synthesis. Many important results were obtained and almost all high level synthesis tasks were precisely defined and outlined; these include allocation, assignment, scheduling, module selection, partitioning, transformations and interface design. Early researchers in the field accurately identified many most relevant issues and proposed the first approaches to address them. For example, the first papers already stressed the importance of design space exploration, which just recently has come again into the attention of both the high level synthesis and design communities. However, the supporting algorithmic and computing environments were of insufficient capability to support overly ambitious design tasks, such as microprocessor design. In the last phase of the first generation, great hopes were based on the use of artificial intelligence techniques. The success was

limited, however. At that point, a majority of researchers in high level synthesis field reached a consensus, that, without dramatic improvements in knowledge acquisition and manipulation techniques, automatic competitive design of general purpose computers is unlikely. This conclusion lead to the second generation of high level synthesis, which targets more restricted design areas and synthesis problems.

The second generation is characterized by a more focused approach and greater attention to the solving of a few high level synthesis tasks. More than a hundred high level synthesis systems have been reported in the CAD, DSP, communication and circuits and systems literature [Wal91]. Despite the large number of systems and the variety of the application domains and the employed optimization techniques, there are striking similarities among many of them: the focus of the synthesis process is almost exclusively on the scheduling, allocation and assignment tasks, although more recently partitioning and module selection have been drawing attention as well.

While providing a clear insight in the nature of the problems and producing a wide variety of excellent solutions, the idea of focusing exclusively on isolated problems generates some monumental pitfalls. Tools were tested on a small set of academic benchmarks and optimized to outperform previously published approaches ("the fifth order elliptical filter syndrome"). No attempts are made to validate the approaches and to assess their potential on real life examples. As a result, a clear vision of the actual cost of a design was lost. The cost of a design is rarely determined by the number of execution units (the "primary" optimization goal), but most often is set by the "secondary" resources, such as registers, interconnect, control and input/output. Although background memory quite often dominates the cost of a system [DeM92, Pat90], it was only recently addressed in a few papers. Finally, while hierarchy has long been accepted as one of the keys to successful system design, it has only drawn marginal attention in the synthesis community.

Compared with the effort spent on individual tasks, scant attention was paid to tool integration. We refer here not so much to software integration, but mainly to the way tools interact and communicate with each other and with the environment. Information feedback to the designer is limited, if existing. The relationship of high level synthesis with respect to the overall system design problem is ill understood or not addressed at all. Designers have, however, made clear that synthesis only makes sense when seen in a system context and in co-existence with other compilation tools, such as available or under development for programmable hardware and heterogeneous hardware/software platforms [Pau92]. Finally, the organization of current synthesis tools is such that they offer scarce opportunity to address the requirements of

the next generation of CAD tools, such as computer aided algorithm selection and design

In depth reviews of the state of the art in high level synthesis can be found in [McF90, Wol91b]. For an inspirational and fresh look at the current trends in high level synthesis and in CAD in general, please refer to [DeM92].

Notwithstanding the above observations, several systems have been designed, which transcend those problems and are successful in designing complex pilot designs. A number of them are currently making it to the commercial arena. The response from the design community is still muted, however. Even if a system succeeds to proceed smoothly from a behavioral level description of an application to a layout, designers complain that the system does not offer any added value: at most, it performs a number of tasks they knew how to do anyhow and it does not offer any help in the most demanding design problems, such as the exploration of the architectural space. So, while the feasibility proof of concept has been demonstrated, the establishment of high level synthesis as a dominant or attractive design option is still pending. If high level synthesis wants to avoid the destiny of other research fields, which generated great expectations and even greater disappointments, it is essential that a number of those observations are successfully addressed.

The HYPER high level synthesis system tries to address some of those issues and is based on two major thrusts:

1) The only way high level synthesis can get an edge over current design approaches, is by identifying and treating optimization intensive tasks, which are difficult (or even impossible) to address manually due to the computational and logical complexity. We believe that optimizing transformations fall into that category. They help to transcend the limitations of the initial specification and can result in more dramatic reductions of the implementation cost (be it area, speed or power) than any other synthesis task. Closely related to transformations is the estimation task, which helps to plot the design space and serves as an essential feedback to the designer. Finally, transformations are the key link to the next layer in the design synthesis process, being algorithm design and selection. The latter is the main topic of this paper. It is important to notice that the use of algorithm selection and optimization use has not been previously addressed in the high level synthesis literature.

2) The only other high level synthesis task, which can have even more dramatic influence on the overall quality of a design, is, surprisingly, the way the high level synthesis tools are organized and integrated. Although it is easy to notice that all high (and low) level synthesis tasks are interrelated, the majority of current day high level synthesis systems concentrate heavily on

individual tasks. Integrating these tools in an effective manner requires a global view on the goal to be achieved, which can only be offered by estimations. The HYPER synthesis manager has been described elsewhere in a detail [Rab90, Pot91a] and is out of the scope of this paper.

Paper Organization

The rest of the paper is organized in the following way. First, the HYPER system and its main components are briefly surveyed. Next, transformations and their use in high level synthesis and in the HYPER system in particular are discussed. Following section contains the most important material in the paper: how transformations can be used to efficiently and effectively select the best computational structure for a given behavioral description and set of goals and constraints. The detailed study of a real life example is presented in this section.

Finally, we conclude by outlining the issues which we consider as the most important directions in high level synthesis in general and in HYPER in particular.

HYPER

Real time applications in areas such as terrestrial, mobile and satellite communications, speech, image and video processing, radar, sonar and computerized tomography often require high performance dedicated datapaths. The structure of the datapath is strongly correlated to the structure of the computations. The controller, if existing at all, is small and contains at most a few tens of states. Pipelining is often used to improve throughput performance.

The synthesis of this type of architecture is an involved, meticulous and cumbersome process, which implies a strong need for sophisticated CAD tool support. The HYPER system addresses exactly this class of numerically intensive algorithms [Rab91]. A detailed description of the HYPER system and its toolset is given elsewhere [Rab91, Pot91a, Pot92b, Chu92, Hoa92]. We will limit our discussion to a presentation of the overall strategy and organization and a review of the transformational techniques.

The software organization of HYPER is shown in Figure 1. The input to the system is a description of application in a behavioral language, called Silage [Hil92]. Silage is an applicative signal-flow language which is particularly effective for the specification of digital signal processing algorithms. The Silage description is translated into an intermediate CDFG (control data

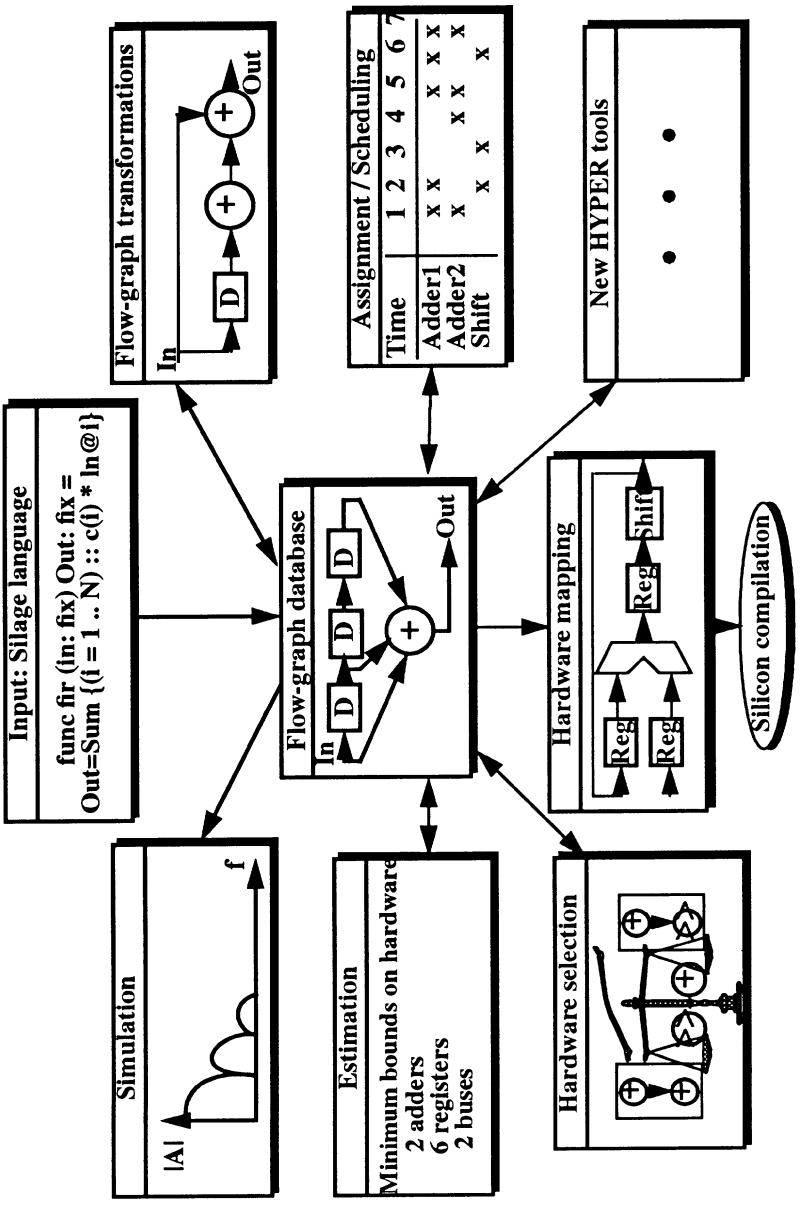


FIGURE 1: Overview of The HYPER System

flow graph) format, which serves as the central repository, on which all synthesis tools are operating and all results are annotated. The tool library includes tasks, such as simulation, hardware module selection, transformations, allocation, assignment, scheduling and background memory management. The single most important future of HYPER is the synthesis manager program, which supports the exploration of the design space using classical high level synthesis tools, transformations and a set of estimation and feedback information tools. The synthesis manager uses a single global design quality measure, called the resource utilization, to direct the design space exploration. It also guides an ergonomic user interface, which facilitates user interactivity.

Once a final result is obtained, HYPER maps the annotated CDFG into an architectural description, which is translated into layout using a silicon compiler. Currently, the Lager IV [Bro92] is used, but a VHDL interface to commercial RTL synthesis environments is provided as well.

Transformations in HYPER

Transformations: What? Why? How?

As we already mentioned, the single most important feature of HYPER is the synthesis manager program. Although, the synthesis manager is capable of achieving implementations, which are optimal or close to optimal (according to the min-bounds predicted by the estimations), its power to compete with and outperform a human designer heavily depends on transformations. Transformations are alternations of the structure of a computation such that the behavior (the relationship between output and input data) is preserved. The impetus behind the application of transformations is the hope that the transformed computational structure might result in an implementation with improved performance. As mentioned above, the performance measure has many dimensions such as area, throughput, power, testability, fault-tolerance and I/O requirements.

A more insightful definition is that transformations are elegant algebraic identities and correctness preserving reorganizations of the computational flow. This definition, as we will show later, is an excellent starting point to answer a number of important questions about transformations, and their role in high level synthesis. Those questions include the following:

(1) Which transformations to implement and support in a high level synthesis system?

It has been recognized for a long time in the compiler community, and now becomes obvious in high level synthesis community too, that implementing and supporting a large set of transformations is a rewarding, but also tedious and overly meticulous effort [Wol92]. Precise answers to a number of complex issues have to be addressed, such as when and where to apply a transformation, how transformations interact, what side-effects are generated and what the global effects are of a locally applied transformation. Large transformation sets also impose numerous complex requirements on the data structures used for storing the CDFG.

(2) How complete is the set of already implemented transformations?

More than a hundred transformations have been proposed in the compiler community alone. It is more than likely that the number of transformations, which can be defined for high level synthesis, is substantially higher, this due to the additional degrees of freedom, the multi-dimensionality of the cost function and the demand for high quality solutions. The nature of computation in an ASIC design is, furthermore, such that significantly more parallelism is generally available, compared to general purpose computation. This results from the fact that most ASICs (such as real time systems) operate on a semi-infinite stream of data and temporal parallelism is hence readily available.

Taking into account all the mentioned points, the important question when considering the introduction of a new transformation is one of diminishing return: do its benefits outweight the algorithmic and software implementation efforts?

(3) What are the limits on the effectiveness of transformations?

As already demonstrated in several compilers and a few high level synthesis systems and illustrated extensively later in the paper, it is not rare that a transformation can result in an order of magnitude improvement in performance. The excitement about this spectacular improvement immediately triggers the following question: what is the real limit on the power of a transformation or set of transformations for a given algorithm. While several techniques have been developed to estimate the effect of the application of other high level synthesis tasks, only simplistic, overly optimistic or pessimistic techniques have been proposed for transformations. It is our belief that no major progress on this difficult and important topic can be expected in the immediate future.

Transformations in High Level Synthesis

The current state-of-the-art in transformations can be traced over several application areas: operational research, software compilers and in particular software compilers for parallel computers, CAD, DSP ASIC and systolic array design, theoretical computer science, and numerical analysis and algebra.

A detailed overview of the transformations and their use in the mentioned fields can be found in [Pot91a]. We will restrict ourselves to an overview of the application of transformations in the high level synthesis domain.

Applying transformations in the high level synthesis process was first proposed in the late seventies [Sno78, Cas80, Mir79, McF83]. The most comprehensive sets of transformations are described in [Tri87, Wal89, Har89a, Bha90, Rab91]. Flamel [Tri87] uses five block level transformations (essentially 2 variants of loop merging and 3 forms of loop unrolling) followed by a greedy application of height reduction and constant folding. The goal is the minimization of the implementation time under area constraints, which is significantly simpler task than the problem where the hardware cost is optimized while time is the constraint. The reason for this is that in the former formulation a hierarchical problem can be solved optimally by optimizing each of the sub-functions individually. Although the optimization strategy is simple and at least partially greedy, impressive experimental improvements are achieved on 15 small examples.

The System Architect's Workbench (SAW) [Wal89] uses in-line expansion, dead code elimination, four types of selects (if-then-else transformations where the code is moved across boundaries imposed by the control structure) as well as pipelining mainly in order to support behavioral and structural partitioning. SPAID [Har89b] proposes the independent use of retiming (and pipelining as the special case of retiming), interleaving, replacement of multiplication by a constant with add/shifts and algebraic transformations. Both SAW and SPAID are interactive frameworks where designer manually explores the design trade-offs using predefined transformation mechanisms. Hi-PASS, a CAD system for DSP architecture synthesis, efficiently uses bit level retiming in designs with no hardware sharing [Dun92].

Gyrczyc [Gyr84] was one of the first to discuss the effect of loop folding. Several high level synthesis researches combine a few transformations (most often loop unrolling, software pipelining and pipelining) with scheduling. For example, Devadas [Dev89] combines scheduling with dynamic partial unwinding (unrolling) and functional pipelining with allocation, assignment, and scheduling using simulated annealing algorithm. Goossens

successfully treats [Goo90] both single and nested loop folding (software pipelining) during scheduling. Hwang combines functional pipelining and loop winding with scheduling [Hwa91].

Besides the use of transformations in high level synthesis systems for computationally intensive computations and microprocessors, their successful application in the design of control dominated machines has recently been reported [Wol91a].

Pipelining is by far the most often applied transformation [Par88, Pau89]. Hartley combines pipelining and tree-high reduction (a special case of associativity), but his technique is applicable only to examples without feedbacks [Har89b]. Although pipelining is very powerful, it is not a transformation in the strict sense, since it increases the algorithm latency. Its application domain is also often limited to non-recursive algorithms [Mes88].

Mechanisms of Transformations

Although transformations can be used to optimize a wide variety of objective functions using an unlimited number of optimization approaches, transformations belong normally to a small set of classes.

HYPER currently uses three types of transformations, according to their scope:

(1) suboperational level transformations
(2) basic block transformations
(3) control structure transformations.

The first type is based on the application of knowledge about the relationship between the various operations in their hardware implementation, as well of internal structure of the computation. Very few transformations of this type are addressed in literature [Mas87, Dal89], and the most important transformation of this type in the DSP ASIC domain is substitution of multiplication by add/shifts. This transformation has been used regularly in HYPER, often with a significant improvements in both area and speed. When substituting multiplications by add/shifts, HYPER uses the Canonical Signed Digit representation to minimize the number of newly introduced operations.

The basic block transformations do not alter the control structure of a program and consist mainly of algebraic manipulations. This is in contrast with the third class, which modifies the structure and dependence of control operations such as conditionals, loops and subroutines. Since the number of combinations for both of them is unlimited, the number of transformations

can be arbitrarily large. So, finding an effective and efficient subset is an important decision during the development of a high level synthesis system.

Basic Block Transformations The majority of the numerically intensive computations can be interpreted within an algebraic structure, called a *field*. A field is a relatively simple structure, consisting of two basic operations defined with the aid of a few axioms. Those operations are most often called *addition* (denoted by +) and *multiplication* (denoted by *). In addition to the fact the field is closed for both addition and multiplication (i.e. for every a, b in the set, both c = a + b and d = a * b are in the set.), the following properties are, per definition, valid:

(1) Both addition and multiplication are associative operations.
(2) Both addition and multiplication are commutative operations.
(3) The distributive law is valid (i.e. a * (b + c) = a * b + a * c).
(4) Both addition and multiplication have identity elements. These elements, usually denoted by 0 and 1, are such that for any element a in the set the following statements are valid: a + 0 = a and a * 1 = a.
(5) Both addition and multiplication have inverse elements. (For each element a in the set, there exist elements b and c in the set, such that a + b = 0, and a * c = 1.)

Although a field is simple structure, it is also very powerful. A field is a set in which one can do "additions", subtractions", "multiplications", and "divisions". The most commonly used instances of fields are: R, the set of real numbers, C, the set of complex numbers, and Q the set of rational numbers. The vast majority of the computations in DSP and other numerically intensive applications are performed in some field.

The application of algebraic transformations for the optimization of performance measure almost always translates into a computationally intractable optimization problem. To address this computational problem, a novel probabilistic sampling algorithm was introduced in HYPER. It uses the five mentioned axioms as basic moves. A crucial element of any optimization problem is the qualification of the cost function, which is both accurate and easy to compute. While this is relatively simple when the goal is to minimize the time, it is substantially more complex when the area is the measure to be minimized. An experimentally derived and statistically verified objective function has, therefore, been introduced to correlate the properties of a DCFG with the area of the final implementation. To simplify the number of moves,

the associativity and inverse element law have been combined into a *generalized associativity* move.

One should be aware of some important side-effects, related to algebraic transformations such as associativity and distributivity. As a result of the finite word length representations, those transformations can result in deviating or even non-correct solutions, when applied on actual hardware. A careful analysis of the obtained result (most often using fixed point simulation) is hence necessary.

The associativity, commutativity, distributivity, identity and inverse element laws are sufficient to transform any computation in a field into any possible, equivalent form. However, as even the high school algebraic experience implies, the explicit use of some composite transformations (algebraic theorems) can greatly help to achieve better structures faster. This is the reason why the basic transformation set is often augmented with other transformations, such as factoring and common subexpression elimination and replication, in compilers and high level synthesis tools. We are currently investigating which small set of algebraic theorems is amenable as a sufficient basis for transformations.

Closely related to the above is the observation that the goal of the majority of the computations is not just to compute one output, but a set of outputs. This makes the identification of redundant and unnecessary computations non-trivial. To handle those situations, a few more transformations come in useful. The most important transformations in this class are the common subexpression elimination and replication, constant propagation, dead code elimination and strength reduction [Aho77]. The treatment of common subexpresions is a difficult and involved problem, where the decision to replicate of eliminate a common subexpression is based on a trade-off between throughput and resource utilization [Pot92b]. While the optimal treatment of constant propagation and dead code elimination is an undecidable problem in the general case, a greedy algorithm often works well in DSP practice. An optimal solution is obtained when no conditionals are present. HYPER applies the greedy algorithm after the application of any other transformation to remove the dead code and the manifest expressions, generated by those manipulations. Strength reduction is often advocated in conjunction with loop transformations and address calculation, where it is often generalized to value numbering [Wai84].

At the border between basic block and control structure transformations exists another important transformation, called *retiming*. Retiming is a conceptually simple, yet powerful transformation, which has been successfully

applied in several CAD areas. It is formally defined as the distributivity property of the delay operator over most other operators: when D is defined as the delay operator, the statement D(a) * D(b) is equivalent to D(a * b) and vice versa (where * is an arbitrary operation). More informally, the retiming transformation moves delays in a CDFG such that a particular objective function is optimized. Most often either the critical path or area of the implementation is targeted. When the former case, the Leiserson-Saxe retiming algorithm generates the optimum solution in polynomial time [Lei91]. However, when area minimization is the goal, the optimization problem is NP-complete [Pot90]. Retiming is naturally defined in the form of a basic move (being a distributivity move), it operates on the same level of granularity as the algebraic transformations and acts as an enabling and disabling transformation for those algebraic transformations. HYPER therefore treats retiming using the same framework, optimization algorithm and objective function. A detailed description of the treatment of retiming in HYPER is presented in [Pot91b].

Closely associated with retiming is the *pipelining* transformation, probably the most popular high level synthesis transformation. Although pipelining is not a transformation in a strict sense as the temporal relationship between the outputs and inputs is changed, it is often used due to its capability to reduce the critical path in a graph with no feedback. Pipelining is easily related to retiming by the following observation: "Pipelining with N stages is equivalent to retiming where the number of delays on all inputs or all outputs, but not both, is increased by N".

HYPER provides separate optimization mechanisms for pipelining with a fixed and a flexible number of stages, and for various performance goals, such as throughput, area and power.

Control Structure Transformations Control structure transformations restructure the control flow of a computational graph, as implied by loop, conditional and subroutine structures. In this area, most of the attention has been devoted to the exploration of loop transformations, as it is widely recognized that this is where most of the parallelism is embodied. Therefore, this section will solely concentrate on this subject.

As in the case of the basic block transformations, we tried to identify a small set of "axiomatic" loop transformations, from which all others (or at least the majority of them) can be constructed. The two major issues in the composition and the transformation of a control structure are (i) the partitioning of the program into control sections and (ii) the determination of the temporal relationship between those units. A simple study indicates that the

following transformations are sufficient to support the partitioning part: loop unfolding (and its inverse transformation, loop folding), loop fusion (and its inverse, called loop fission), loop blocking, loop permutation, loop retiming and invariant code motion.The ordering of the partitioned units is effectively supported by the software loop pipelining and loop interleaving transformations. A detailed description of all the mentioned transformations can be found in several standard compiler references [Aho77, Fis88]. It appears that the small set of transformations, mentioned above, is sufficient to support the wide spectrum of loop transformation, proposed in compiler literature.

At present, HYPER supports only partial and full loop unfolding, loop expansion and software retiming and pipelining. The development, implementation and support of other control structure transformations is under investigation.

Transformation Ordering

From previous applications in the areas of software compilation, logic synthesis and high levels synthesis, it was learned that the ordering of transformations is often the key factor in achieving their full potential. At each step of the optimization process, the system must select a transformation which is both legal and desirable. While the application of individual transformations has received a lot of attention and has yielded impressive results, barely any progress has been achieved in the area of transformation ordering. By far the most popular and widely used approach is a static ordering where the order of transformations is given a priori, most often in the form of a script. Script development is almost always based on experience and extensive experimentation. This method has at least two drawbacks: it is a time consuming process which involves a lot of experimentation and the quality of the solution is strongly dependent upon the match between the application characteristics and the underlying strategy of the script. For well defined application areas, such as logic synthesis or linear digital signal processing, this approach has proven to be very effective.

Another, often advocated, approach, is the "generate and test" technique [Wol91a]: explore all possible combinations of transformations. Although this method can be augmented with suitable implicit enumeration techniques, its obvious drawback is a large run time, exponential in the number of transformations. Another interesting approach, proposed recently, is to use a mathematical theory behind the ordering of some transformations. However, this method is limited only to a class of loop transformations [Wol91a].

HYPER supports two modes for operation ordering at present: the user guided mode and, the above mentioned, optimal scripts. In the user guided mode, the synthesis manager provides information to the user with respect to the bottlenecks in the design, their nature and the estimated effect of a transformation. A detailed description of this mode is given in [Rab90].

For several important sets of transformations and application domains, we achieved in defining simple and optimal transformation orderings, such that a particular cost function is minimized (or maximized) in polynomial time. This was, for instance, the case for the important class of linear computations over a set of seven transformations (distributivity, associativity, commutativity, inverse element law, identity element law, dead code elimination and constant propagation) [Pot92b]. Even when unfolding is added to this class, we proved that an optimal script can be defined [Pot92b]. Currently, we are addressing the design of optimal scripts for other application areas and even wider sets of transformations.

What Next in Transformations?

After been studied for several centuries in mathematics and a few decades in the compiler domain, transformations are continuing to be an important research topic in those areas. Considering the considerably shorter history of transformations in high level synthesis, it is certain that they will be a topic of intense study for some time to come.

In the area of numerically intensive applications, we expect that loop transformations supported by a loop dependency analysis (also called LCD) will be the primary tool to achieve an extra order of magnitude of performance improvement. The current probabilistic optimization algorithms, implemented in HYPER to support basic block transformations, can be sped up when supplemented with heuristics to target directly critical parts of the restructured CDFG. Increasingly background memory and I/O management and design will be supported and optimized using transformations, which directly target this type of resources.

Numerically intensive applications are being merged increasingly with non-numerical applications. A set of axiomatic and theorems based transformations, which target non-numerical transformation will therefore become important. Important elements of this class are transformations related to full and partial ordering, set membership and string manipulation. We expect that transformations which target control dominated designs will gain popularity and prove their effectiveness.

The ordering of transformations and an even more close integration and combination with other high level synthesis and software compiler tasks [Gue92] will have a significant impact on the field. Finally, and we believe most importantly, transformations will evolve and serve as the basis for of a new field of algorithm design, selection and tuning. In the next section, we will demonstrate how the combination of algorithm selection and optimizing transformations can produce results which cannot be anticipated by even advanced human designers.

ALGORITHM SELECTION AND PERFORMANCE IMPROVEMENT USING TRANSFORMATIONS

Motivation

Even a superficial, brief survey of the algorithm design research in several important application areas (such as numerical analysis, DSP, communication and information retrieval) reveals that a large variety of algorithms exist to address a single problem.

Consider, for example, the problem of sorting. In addition to the conceptually simple selection, insertion and bubble sort, there exist several more subtle algorithms, such as quick sort and Batcher bitonic sort [Knu73]. For common matrix transformations, such as fast Fourier transform (FFT) and discrete cosine transform (DCT), which are often used in audio and video processing and many other digital processing areas, there exist dozens of other fast algorithms [Nus82, Rao90]. A variety of algorithms exist to solve a Toeplitz set of equations, which occurs in a large number of signal processing applications such as for instance speech coding. The most popular ones are the Levinson-Durbin, Trench, Barlekamp-Massey and Barlekamp-Massey recursive algorithms [Bla85].

In some cases, there is an obvious superiority of one algorithm over others in terms of performance, in particular if one is interested only in asymptotic computational complexity [Gar79]. More often the optimality of an algorithm depends upon additional constraints. Although the selection of the right algorithm for a targeted task has a major impact on the final quality of the design, this area is currently more art than science or engineering and designers almost exclusively rely on intuition instead of accurate quantitative procedures. This intuitive approach is however bound to break down.

Progresses in technology have increased the application complexity and have made parallel processing a more viable alternative. Selecting the right algorithm within this setting is becoming increasingly more difficult for a human designer.

Algorithm Selection and Tuning Using Transformations

Transformations are an efficient and powerful tool for the enhancement and exploration of parallelism. It has been shown in previous section they can have a dramatic influence on the quality of an implementation. However, their effect is obviously constrained by the algorithm's computational structure. For example, it is simple to construct program instances, which are not amenable at all to transformations. Therefore, the only tools with more potential than transformations are located in the area of algorithm design and selection.

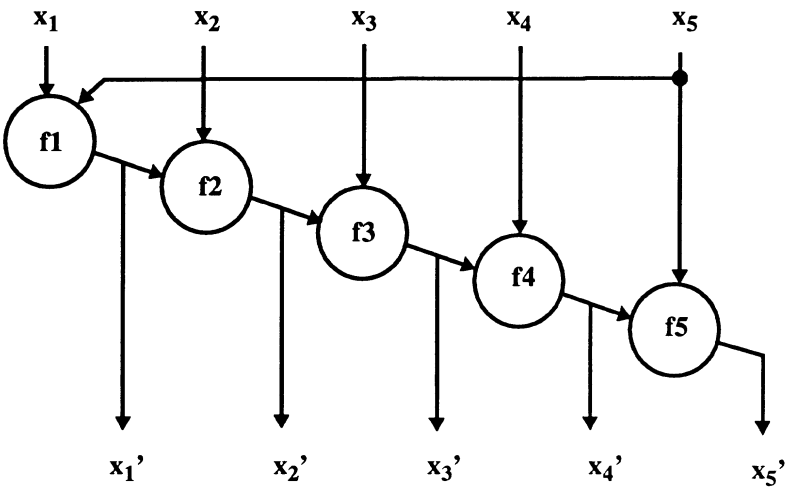
An obvious goal in this area is the automatic generation of algorithms (which adhere to a given set of specifications), such that the resulting computational graph can be easily transformed into a final implementation with maximal performance or minimal cost. It is very likely that this goal will remain elusive for years to come. However, it appears that there exist several ways to do efficient algorithm design for restricted application domains.

For instance, the algorithms used in wide classes of applications have a substantial freedom in their computational structure. This makes them prime candidates for a search to determine, which structures possess the most parallelism or hardware potential. Consider for instance the class of algorithms, which address the iterative solution of sets of equations. Examples of such algorithms are shown in Figure 2 and Figure 3. They all have the following structure:

$$x(t+1) = f(x(t)), \quad t = 0, 1, \quad \ldots \tag{1}$$

where each $\boldsymbol{x}(t)$ is an n-dimensional vector and $f(.)$ is some function which has as domain and range some subset of the n-dimensional space. If the sequence $\{\boldsymbol{x}(t)\}$ generated by the above iteration converges to a *point* $\boldsymbol{x}^*$, and if the function $f(.)$ is continuous, then $\boldsymbol{x}^*$ is a fixed point of f, which satisfies the relationship $\boldsymbol{x}^* = f(\boldsymbol{x}^*)$. For example, iterative methods are often used for the solution of sparse system of equations, or for the maximization and minimization of a function by search for the zeroes of the derivatives.

When all the components of $\boldsymbol{x}$ are updated simultaneously, the method is often called the Jacoby or Gauss-Jacoby iteration. An alternative approach



x1' = f1 (x5, x1)
x2' = f2 (x1', x2)
x3' = f3 (x2', x3)
x4' = f4 (x3', x4)
x5' = f5 (x4', x5)

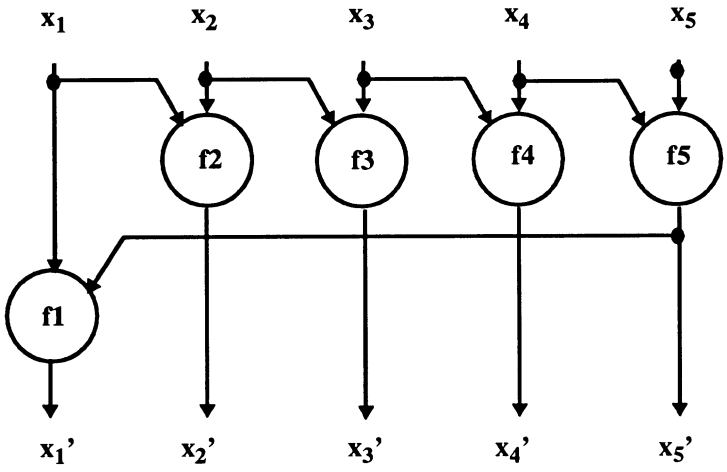
FIGURE 2: Parallelism of Gauss-Seidel iterations: The initial updating order.

is to update one equation at a time. Equation (1) can then be expressed in the following form:

$$x_i(t+1) = f_i(x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_n(t)), \quad i = 1, \ldots, n \quad (2)$$

This type of iteration is often called the Gauss-Seidel iteration and is illustrated in Figure 2 for one particular problem instance. Since the Gauss-Seidel algorithm incorporates the most recent information at each step [Ort90], it often converges faster than Gauss-Jacoby iterations. It is easily recognized that the Gauss-Seidel algorithm, due to its recursive nature, is not well suited for pipelining. Furthermore, no parallelism is available. On the other hand, it is very amenable to transformations or manipulations which

address the fast execution of recursive programs to the fast implementation of recursive program transformation. This observation can be substantiated in the following way:



$$x_2' = f2\ (x_1, x_2)$$
$$x_3' = f3\ (x_2, x_3)$$
$$x_4' = f4\ (x_3, x_4)$$
$$x_5' = f5\ (x_4, x_5)$$
$$x_1' = f1\ (x_5', x_1)$$

FIGURE 3: Parallelism of Gauss-Seidel iterations: The increase in parallelism and the reduction of the critical path due to the change in the updating order.

In the Gauss-Seidel iterative algorithm, the order of updating is not fixed. Instead of starting from x_1 and proceeding forward, we can permute the updating order, as shown in Figure 3. Of course, this results in a new algorithm with different convergence properties. Nevertheless, for many sets of equations, all Gauss-Seidel algorithms will converge to the same fixed point after a number of iterations. By analyzing the speed of convergence, the performance and the amount of available parallelism, we can, for a particular application instance, construct the algorithm, which is the best suited for implementation. For example, it is easy to see that the algorithm of Figure 3 has a significantly shorter critical path than the algorithm in Figure 2 and will

result in a substantially faster implementation. Besides the updating order, other aspects of the Gauss-Seidel algorithm are prone for optimization as well. For instance, one can consider the effect on both performance and convergence of the updating more than one component of x at a time.

Similar algorithm design selection techniques can be used in a variety of other numerically intensive algorithms. They include algorithms for the solution of ordinary differential equations (such as the Runge-Kutta adaptive size step methods, the Bulirsh-Stoer method, and the Predictor-Corrector technique), partial differential equation solvers and various minimization and maximization algorithms. One especially interesting domain is the class of probabilistic optimization methods.

From the above, it becomes obvious that, in order to be effective and realistic, an algorithmic designer needs a set of tools, which can help him to predict or analyze the speed of convergence (and other performance properties, such as the numerical behavior) and the implementation properties of a proposed algorithm. While simulation (or closed form analysis for some application classes) is the preferred medium for the former task, tools like HYPER with its estimations and transformations can help to predict and analyze the latter.

For many common tasks, as already mentioned, there exists a variety of sophisticated algorithms. The structure and the parameters of those algorithms were developed, most often manually by a human designer, in an attempt to optimize one particular (and often simplistic) objective function. This results in algorithms which perform well with respect to that parameter, but ignore all other dimensions of the implementation cost. A typical example of such an algorithm is the FFT, which was developed to minimize the number of multiplications in the DFT. This was a worthwhile goal at that time, but progresses in implementation technology have made this cost function less critical, while factors such as regularity and interconnect structure have gained importance.

We propose the following procedure, which helps a designer to select the most promising algorithm for a given problem instance over a set of goals and constraints.

Given a set of potentially useful algorithms $\wp$, the first step is to identify the so called *non-inferior* algorithms. An algorithm A is called inferior to an algorithm B, if all of the following criteria are fulfilled:

(1) Algorithm B uses fewer operations of all types than algorithm A;

(2) Algorithm B has a shorter critical path and shorter iteration bound than algorithm A;

(3) According to all other available lower bound and statistical estimations, algorithm B is always superior to the algorithm A. To evaluate these properties, we use the estimation techniques available in HYPER.

An algorithm A is then called non-inferior, if it is not inferior with respect to all algorithms in $\wp$.

For all non-inferior algorithms, we use branch and bound (B&B) techniques to select the best one for the given set of goals and constraints. During B&B, the implementation cost is estimated using the lower bound estimation techniques, available in HYPER. To reduce the impact of the initial algorithm specification, the algorithms are optimized and tuned over a set of transformations. This set of transformations is restricted to those transformations, which do not alter or degrade the numerical behavior of the original algorithm. Since HYPER employs very fast estimation and synthesis techniques, usually one afternoon is sufficient for the process of algorithm selection and tuning.

Proof of Concept Example

In this section, the potentials and the impact of the proposed algorithm selection technique will be demonstrated using an eight order Avenhaus Bandpass Filter as a driver example. It will be shown how the proper algorithm selection can result in dramatic improvements in either throughputs or area. Another important point, which is conveyed in this section is that transformations are an integral part of the algorithm selection process and that ignoring this point can result in inferior solutions or wrong decisions. Finally, it is demonstrated that algorithm selection is a complex process and that the results most often deviate dramatically from intuitive insights!

The importance of filters is illustrated by the quote from "Fast Algorithms for Digital Signal processing", by R.E. Blahut [Bla85]. He says: "The most important task of digital signal processing is the task of filtering a long sequence of numbers, and the most important device is digital filter." While there are many important families of non-linear filters, including homomorphic, morphological, nonlinear mean, order statistics and higher order Volterra filters, linear filters are the most widely used, because of their well studied and understood properties and their inherent simplicity, which in many situations guarantees satisfactory performance for a minimal cost. The widespread use of linear filters was a major factor in selecting them as the driver example in this paper.

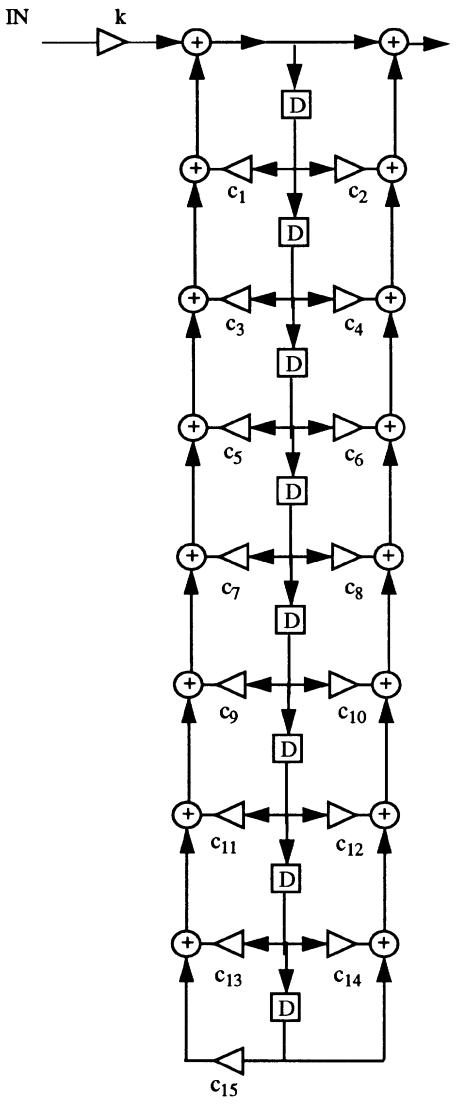
The selection of the appropriate filter structure for a specified frequency response has long been recognized as one of the crucial factors determining almost all filter characteristics such as numerical stability and implementation complexity [Cro75]. As a result, many sophisticated filter structures have been proposed. While the numerical side of the filter behavioral is well understood, the implementation complexity issue have been rarely or only marginally addressed.

The filter under study in this paper was first presented by Avenhaus [Ave72] and has often been used in the digital filter design, analysis and research. For example, Crochiere and Oppenheim [Cro75] presented in an depth discussion of several digital filter structures, which can implement the required frequency response. They compared the structures according to their statistical coefficient word length, the required number of multiplies and adds, the total number of operations and the amount of the parallelism and serialism. Although their analysis and presentation is prototype example of excellency in research, we will show that the presented measures are far from being sufficient to select a proper structure for ASIC implementation. Actually, our analysis of the example implicates that this manually conducted procedure often leads to misleading conclusions.

We consider the following five structures of the Avenhaus filter, designed by Crochiere and Oppenheim [Cro75]:

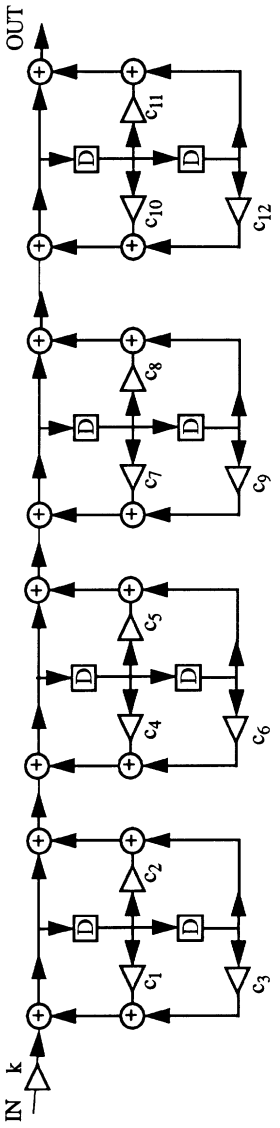
(i) direct-form II;
(ii) cascade form composed of direct form II sections;
(iii) parallel form;
(iv) continued fraction expansion structure type 1B;
(v) Gray and Markel's ladder structure.

The corresponding structures with the appropriate coefficient values for the filter at hand are shown in Figures 4 throught 8.. The first four structures correspond to different representations of the same transfer function. The first three of them (direct-form II, cascade and parallel form) are well known and were considered very early in the filter design literature [Opp89]. The direct form corresponds to the representation of the transfer function as a ratio of polynomials, the cascade form represents the transfer function as a product of second order polynomial ratios, while the parallel and continued fraction forms correspond to partial and continued fraction expansion respectively. The continued fraction representation was first proposed by Mitra and Sherwood [Mit72, Mit73]. The fifth structure is proposed by Gray and Markel and



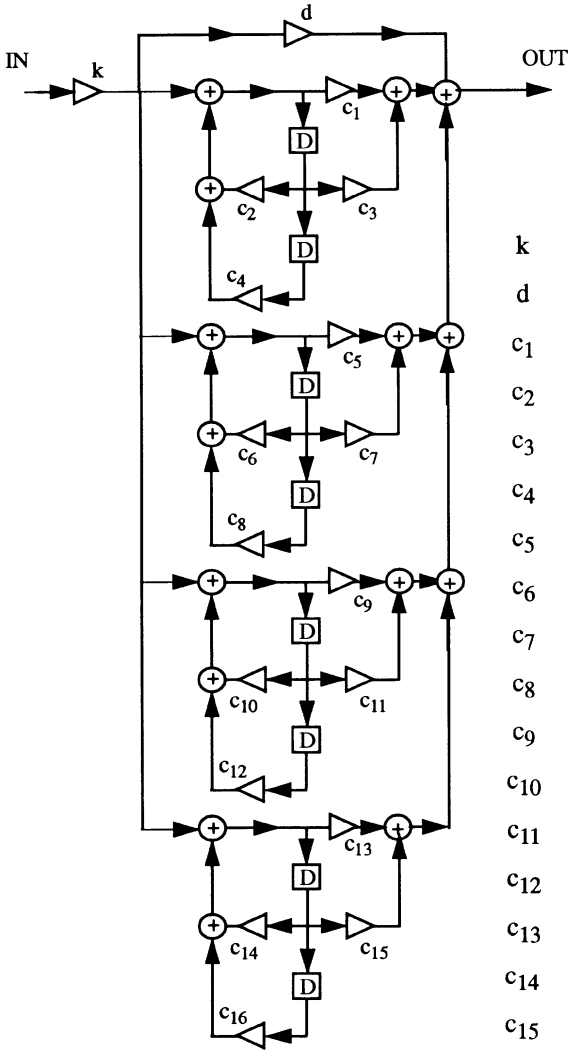
k	=	0.005656462366
c_1	=	1.479757145
c_2	=	-1.387818861
c_3	=	-4.548129731
c_4	=	3.990834175
c_5	=	4.352241822
c_6	=	-3.801090558
c_7	=	-6.802822262
c_8	=	5.978059098
c_9	=	4.094876357
c_{10}	=	-3.801090558
c_{11}	=	-4.026604318
c_{12}	=	3.990834175
c_{13}	=	-0.7833447265
c_{14}	=	-1.387818861
c_{15}	=	1.232007442

FIGURE 4: Direct-form II Structure



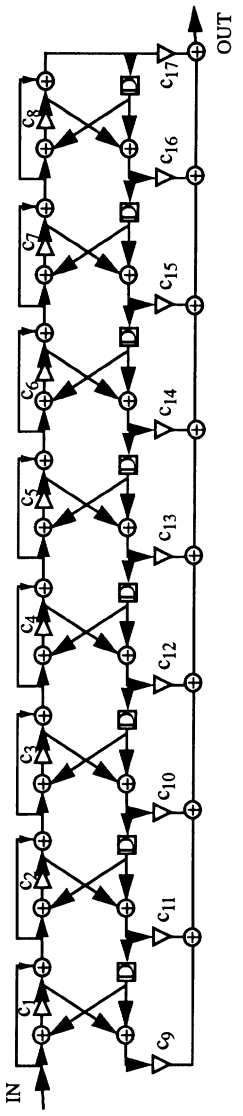
k	=	0.005656462366
c_1	=	0.2855823838
c_2	=	0.4512591755
c_3	=	-0.9116860618
c_4	=	0.4457342317
c_5	=	-1.09869455
c_6	=	-0.913078673
c_7	=	0.196901665
c_8	=	-0.0099665157
c_9	=	-0.9695353354
c_{10}	=	0.5515388641
c_{11}	=	-0.7304169706
c_{12}	=	-0.9705899009

FIGURE 5: Cascade-form (direct-form II sections) Structure



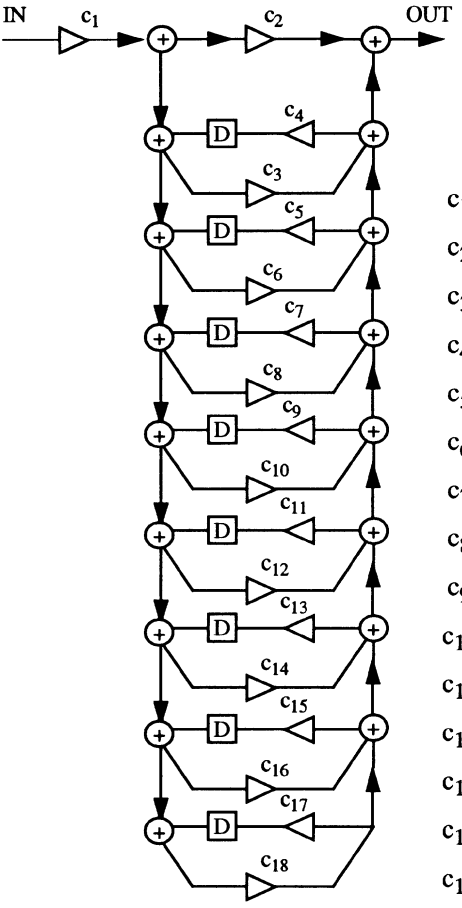
k	=	0.10000
d	=	0.07220910762
c_1	=	0.2034508401
c_2	=	0.2855823838
c_3	=	0.9026252904
c_4	=	-0.9116860616
c_5	=	0.182701196
c_6	=	0.4457342317
c_7	=	-0.9457352763
c_8	=	-0.913078673
c_9	=	-0.2052671178
c_{10}	=	0.196901665
c_{11}	=	-0.2413054652
c_{12}	=	-0.9695353354
c_{13}	=	-0.1965294023
c_{14}		0.5515388641
c_{15}		0.298888793
c_{16}		-0.9705899009

FIGURE 6: Parallel-form structure.



c1	=	0.7833447265
c2	=	-0.1885428229
c3	=	0.9843741951
c4	=	-0.1920962931
c5	=	0.9936376827
c6	=	-0.1890252997
c7	=	0.9914182038
c8	=	-0.1964790194
c9	=	0.005656462366
c10	=	0.0002916124024
c11	=	-0.001699657657
c12	=	-0.06980454614
c13	=	0.203123313
c14	=	0.01061747099
c15	=	-0.01641846145
c16	=	-1.031187912
c17	=	0.8446328267

FIGURE 7: Ladder structure by Gray and Markel.



c_1 = 0.0002828231183
c_2 = 25.53154355
c_3 = -0.2117906668
c_4 = 1.017015349
c_5 = 0.1736820909
c_6 = -12.7837888
c_7 = 0.4649449432
c_8 = -1.606209516
c_9 = -0.2767502193
c_{10} = 7.610142205
c_{11} = 3.939247317
c_{12} = -0.4301416747
c_{13} = -0.9905202523
c_{14} = 0.6779657016
c_{15} = -21.82668197
c_{16} = -0.4193825017
c_{17} = 0.2481853405
c_{18} = 0.4028556869

FIGURE 8: Continued-fraction expansion structure type 1B.

is based on an exploration of the relationship between two port networks in the analog circuit and digital filtering theory [Gra73].

In addition to presented structures, there exists a great variety of others which, of course, can also be used for the realization of the required frequency response. The most popular among them, often cited for competitive characteristics, are the wave digital filters [Fet86], the state-space digital filters [Mul76], the orthogonal [Rao84] and the multi-variable digital lattice filters [Vai85] filters (which are a generalization of both Gray-Markel's and Rao-Kailaths digital filters) and finally the systolizable IIR digital filters [Lei90].

We limited our analysis to the five mentioned structures, since our goal is not to find the "ultimate" structure for the implementation of the Avenhaus eight order bandpass filter over all structures ever proposed, but to present a procedure to classify arbitrary computational structures given a set of constraints and implementation goals. Actually, the results will indicate that no such "ultimate" exists, as the optimality of a structure strongly depends upon the goals and implementation constraints

Experimental Results

All five Avenhaus structures were described using the Silage language and passed to the HYPER environment. They were simulated in both double floating point and fixed point precision in all phases of the design process to ensure that the obtained structures were in agreement with the required frequency response.

structure	number of multiplications	number of additions	statistical word length
direct form II	16	16	21
cascade	13	16	12
parallel	18	16	11
continuous fraction	18	16	23
ladder	17	32	14

Table 1: Number of operations and Statistical Word Length.

Table 1 (assembled using data from [Cro75]) shows the number of multiplications and addition and the statistical word length for all five forms. The

statistical word length is the minimum number of bits needed, such that the resulting structure still meets the frequency domain specifications (as determined by a statistical analysis). We simulated all five examples, and indeed, all of them produced the required frequency response. A small correction was needed for the direct form II, as, due to a typographical error, two coefficients were interchanged in the Crochiere and Oppenheim paper.

Table 2 shows the number of operations in the five structures after the multiplication strength reduction, which substitutes all multiplications by a proper combinations of shifts and adds. Generally speaking, of course, there is a correlation between the number of multiplications and the word length in the initial form and the number of shift-operations after the expansion. However, only a precise analysis can determine the final effect of the application of the multiplication strength reduction.

structure	total	additions	shifts	subtractions	transfers
direct form II	215	58	103	46	7
cascade	94	31	40	18	4
parallel	113	33	51	24	4
continuous fraction	205	55	106	43	-
ladder	116	35	49	31	-

Table 2: Number of operations after Multiplication Expansion. Note that all structures have one input operation in addition to the listed Operations. The transfer operations are register-register transfers, needed for the implementation of delays.

For example, although the direct form has both fewer multiplications (16 vs. 18) and a shorter word length (21 vs. 23) than the continuous fraction, the latter eventually requires fewer shifts (103 vs. 92) and a smaller number of operations (215 vs. 178). From this table, it already becomes clear that, even when implementing the filter on a general purpose, single processor computer, the selection of the right algorithm is extremely important (for instance because of the wide span over the number of operations: 94 versus 215). This is even more true when considering ASIC implementations.

Table 3 shows the length of the critical path for all five forms in the initial format and after the application of retiming and pipelining. All examples are implemented using the word lengths as indicated in Table 1. We decided

to use only those two transformations, because they never alter the bit width requirements. Although HYPER has more than 20 other transformations, many of them may alter the numerical behavior and hence the required data representation. Distributivity is an example of such a transformation. During the optimization for the critical path, we used the Leiserson-Saxe retiming algorithm for both retiming and pipelining [Lei83,Pot92a]. During the optimization for area, retiming and pipelining for resource utilization was used [Pot91b, Pot92b]. As will be demonstrated below, those few transformations had already profound effects on the final results.and

structure	initial	retimed	pipelined
direct form II	980	686	686
cascade	527	341	279
parallel	609	522	261
continuous fraction	2332	1908	1908
ladder	2835	2835	630

Table 3: Critical path for the Avenhaus Eight Order Bandpass Filter (in μsec - for a 2 μm technology).

In the initial structures, the ratio in critical paths between the fastest (cascade) and the slowest (ladder) structures equals 5.4. When no extra latency is allowed, this ratio becomes even higher (8.3) as the cascade structure is very amenable to the retiming transformation, while this is clearly not the case for the ladder filter. If we allow the introduction of one pipeline stage, the parallel form becomes the fastest structure. It is important to notice that, when throughput is the major concern, other filter structures can be conceived, which result in even smaller critical paths. For example, the recently proposed maximally fast implementation of linear computations [Pot92b] reduces the length of the critical path to only 174 nanoseconds without introducing any latency. This is the improvement by the factor of 16.3 over the original ladder structure. When additional latency is allowed, this factor can be improved to arbitrarily high levels [Pot92b].

Table 4 shows the area of the final implementation for the five structures for six different sampling periods. For all five forms, results are shown for both the original and the transformed structures. Note that the feasibility of acheiving the required performance is a function of the selected structure and the applied transformations. It can be observed that the ratio between the larg-

est and smallest implementations is even higher than the mentioned performance ratios.

For example, for a sampling period of 1 micro-second, only the direct, cascade and parallel forms are feasible alternatives. The ratio of the implementation areas between the direct form and the cascade form equals 12.9. This ratio is improved to 15.1 for the retimed cascade form (retiming was done using the retiming for resource utilization [Pot91b]). Interestingly enough, there is not a single instance were pipelining improved the area requirements. This is the consequence of the fact that in all designs, except for the fastest implementations, the major part of the area cost is located in the registers and not the execution units or the interconnect. Most often, pipelining increases registers requirements even more. Also, due to recursive nature of the filters, the effectiveness of pipelining is limited.

structure	sampling period (in μsec)					
	5	**4**	**3**	**2**	**1**	**0.5**
df II I	20.32	20.32	22.88	26.36	92.49	-
df II PR	19.86	19.86	21.75	30.65	53.98	-
cascade I	6.63	6.63	6.63	6.63	8.97	-
cascade P	5.98	5.98	5.98	5.98	6.11	11.36
parallel I	6.71	6.71	6.71	6.71	8.67	-
parallel R	5.92	5.92	5.92	5.92	7.16	-
parallel P	5.92	5.92	5.92	5.92	7.16	13.42
con frac I	14.65	22.67	-	-	-	-
con fra RP	13.56	19.54	27.19	-	-	-
ladder RI	5.73	7.38	-	-	-	-
ladder P	5.73	7.38	10.77	16.87	-	-

Table 4: Implementation Area for six different throughput requirements. I - initial structure; R - retimed structure; P - pipelined structure.

Table 5 shows the critical path for all five forms (original and transformed) when the statistical word length information is not used and all structures are implemented with 23 bit data and coefficients. All forms simulate

well for this word length. It can be observed that the critical path increased for all structures, except for the continuous expansion form, which needs 23 bits in any case. For example, the critical path of the 23 bit ladder filter (5300 μsec) is 20 times higher than what was obtained for the optimal cascade or parallel structures with an optimized word length (261 μsec). The obvious and important conclusion is that the importance of word length optimization can not be overstated and is a crucial component during the optimization for both area and throughput. Preliminary studies also indicate that the importance of word length optimization is even higher when power and testability are considered. Interestingly enough, until recently, very little attention was paid to this important issue, which often has a dramatic influence on the quality of the final implementation.

Looking again at Table 4, we see that, depending upon the required throughput, the minimal area is obtained by different structures. For the fastest designs (designs where throughput rate is higher than 1 MHz), the pipelined cascade (when additional latency is allowed) and the parallel forms are the smallest solutions. For medium speeds (sampling period between 1 and 4 μs) the parallel form achieves the smallest area. Finally and most surprisingly, when the throughput requirements are the least strict, the ladder form is the most economical implementation. Notice that ladder form does neither have the smallest bit width nor the fewest number of operations. However, its regular and balanced structure requires few registers and few interconnects, which results in a slightly smaller area than other implementations.

structure	initial	retimed	pipelined
direct form II	1060	742	742
cascade	1113	795	689
parallel	1484	1431	689
continuous fraction	2332	1908	1908
ladder	5300	5300	1272

Table 5: Critical Path for 23 bit Word Lengths (for all structures).

It is very important to observe that the previous analysis does not indicate that any form is a-priory superior in terms of speed or area. The results depend strongly upon the required frequency response, the applied transformations, the objective function (for instance power or testability) or the per-

formance parameters (for instance signal to noise ratio or overflow behavior). For instance, the conclusions might be different for a 7th order elliptical low-pass filter. Our point is that for a given set of goals and constraints, only a quantitative analysis such as proposed in this paper, can provide a more qualified view, which can help designers in making the proper decisions during the design process.

Finally, it is worth mentioning that all results presented in this section, if not otherwise stated, were obtained using the HYPER system. All results were generated and analyzed in a time span of two hours, which demonstrates that efficiency of HYPER is high enough to address the algorithm selection and tuning problem.

FUTURE DIRECTIONS IN HYPER

It has been a proven wisdom that the quality of an answer is almost always proportional to the number of questions it rises. At least according to this criteria, the HYPER approach has been highly successful. Design experience with HYPER have lead to a number of exciting novel design approaches and research venues. For instance, it was realized that high level synthesis techniques and optimizations can have a major impact on the power consumption of an implementation. This result is especially interesting given the current interest in portable consumer, computation and communication devices. A low power design methodology has been developed and implemented in HYPER [Cha92]. We have furthermore realized that the synthesis techniques, used in HYPER (especially the estimations, transformations and instruction and hardware selection) are amenable to other architectural styles as well. This has lead to the development of a a number of related synthesis environments for field programmable data path architectures[Che92] and multi-processor programmable DSP's [Hoang92]. We are currently investigating how HYPER can be adapted to address the retargetable compilation problem for programmable DSP's. Finally, using similar concepts as described in this paper, it can be easily seen that HYPER can be extended to address the architectural design space exploration and performance analysis problems. Efforts in this direction are under way.

Both transformations and estimations (which were not discussed here due to lack of space) are research areas in the early stages of development. Their impact in domains such high level and system level synthesis will increase with the development of novel and more powerful techniques. These results might be reflected into more traditional domains, such as retargetable

compilers and compilers for massively parallel machines. It is our belief that tools such as HYPER can act as a seed for widely used, industrial strength synthesis environments. Finally, we are convinced that computer aided algorithm selection and tuning is one of techniques, which will help the designer to manage and conceive ever more complex systems and applications.

REFERENCES:

[Aho77] A.V. Aho, J.D. Ullman: "Principles of Compiler Design", Addison-Wesley Publishing Co., Reading, MA, 1977.

[Ave72] E. Avenhaus: "On the design of digital filters with coefficients of limited word length", IEEE Trans. on Audio and Electroacoustics, Vol. 20, pp. 206-212, 1972.

[Bar73] M. Barbacci: "Automatic Exploration of the Design Space for Register Transfer (RT) Systems", *Ph.D. Thesis*, Dept. of CS, Carnegie-Mellon University, November 1973.

[Bha90] J. Bhaskar, H. Lee, "An Optimizer for Hardware Synthesis", IEEE Design and Test of Computers, Oct. 1990.

[Bla85] R. E. Blahut, "Fast Algorithms for Digital Signal Processing", Addison-Wesley Publishing Company, 1985.

[Bro92] R.W. Brodersen, ed.: "Anatomy of a Silicon Compiler", Kluwer Academic Publishers, Boston, MA, 1992.

[Cam91] R. Camposano, R.A. Walker: "A Survey of high-level synthesis systems", Boston: Kluwer Academic, Norwell, Mass., 1991.

[Cas80] A.E. Casavant, D.D. Gajski, D.J. Kuck: "Automatic design with dependence graphs", 17th ACM/IEEE Design Automation Conference, pp. 506-515, 1980.

[Cha92] A.P. Chandrakasan, at. al.: "Hyper-LP: A Design System for Power Minimization using Architectural Transformations", IEEE International Conference on Computer-Aided Design, Santa Clara, CA, paper 6c.3, November 1992.

[Che92] D.C. Chen, at. al.: "An Integrated System for Rapid Prototyping of High Performance Algorithm Specific Data Paths", IEEE ASAP-92, pp. 134-148, 1992.

[Chu92] C-M. Chu, J. Rabaey: "Harware Selection and Clustering in the HYPER Synthesis System", IEEE EDAC-92, Brusses, Belgium, pp. 176-180, 1992.

[Cro75] R.E. Crochiere, A. V. Oppenheim: "Analysis of Linear Networks", Proceeding of the IEEE, Vol. 63, No. 4, pp. 581-595, 1975.

[Dal89] W.J. Dally: "Micro-Optimization of Floating point Operations" Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III), SIGARCH Computer Architecture News, Vol. 17, No. 2, pp. 283-289, 1989.

[DeM92] H. De Man: "Design Technology Research for the Nineties: More of the Same?", Proceedings of the European Design Automation Conference, Hamburg, Germany, pp. 592-596, 1992.

[Dev89] S. Devadas, A.R. Newton: "Algorithms for Hardware Allocation in Data Path Synthesis", *IEEE Transaction on CAD*, Vol 8, No 7, pp. 768-781, 1989.

[Dun92] P. Duncan, S. Swamy, S. Sprouse, D. Potasz, R. Jain, N. Gafter, W. Cammack, Y. Wong, W. Gass: "HI-PASS: A Computer Aided Synthesis System for Fully Parallel Digital Signal Processing ASIC's" to appear in Proceedings of the International Conf. on Acoustic, Speech & Signal Processing, March 1992, San Francisco.

[Fet86] A. Fettwies: "Wave Digital Filters: Theory and Computation Methods", Proceedings of the IEEE, Vol. 74, No. 2, pp. 270-327, 1986.

[Fis88] C.N. Fischer, R.J. LeBlanc, Jr.: "Crafting a Compiler", The Benjamin/Cummings Publishing Co. Inc., Menlo Park, CA, 1988.

[Gar79] M.R. Garey, D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H.Freeman and company, New York, 1979.

[Goo90] G. Goossens, J. Rabaey, J. Vandewalle, H. De Man: "An Efficient Microcode Compiler for Application Specific DSP Processors", IEEE Transaction on CAD of integrated circuits and systems, Vol. 9, No. 9, pp. 925-937, 1990.

[Gra73] A.H. Gray, J.D. Markel: "Digital lattice and ladder filter synthesis", IEEE Trans. on Audio Electroacoustic, Vol. 21, pp. 491-500, 1973.

[Gue92] L. Guera: Personal Communications, 1992.

[Gyr84] E. Gyrczyc: "Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Description", *Ph.D. Thesis*, Carleton University, 1984.

[Har89a] R. Hartley, A. Casavant: "Tree-height Minimization in Pipelined Architectures", IEEE IC CAD, pp.112-115, 1989.

[Har89b] B.S. Haroun, M.I. Elmasry: "Architectural Synthesis for DSP Silicon Compilers", IEEE Transaction on CAD for IC, Vol. 8, No. 4, pp. 431-447, 1989.

[Hen89] J.L. Henessy, D.A. Patterson: "Computer architecture: a quantitative approach", San Mateo, Calif.: Morgan Kaufman Publishers, 1989.

[Hil92] P. Hilfinger, J. Rabaey: "DSP Specification Using the Silage Language", in "Anatomy of a Silicon Compiler", ed. by R.W. Brodersen, pp. 199-220, Kluwer Academic Publishers, Boston, MA, 1992.

[Hoa92] P. Hoang, J. Rabaey: "Hierarchical Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput", IEEE ASAP-92, pp. 21-36

[Hwa91] C.-T. Hwang, Y.-C. Hsu, Y.-L. Lin: "Scheduling for Functional Pipelining and Loop Winding", ACM/IEEE Design Automation Conference, San Francisco, pp. 764-769, 1991

[Lei90] S.-M. Lei, K. Yao: "A Class of Systolizable IIR Digital Filters and Its Design for Proper scaling and Minimum Output Roundoff Noise", IEEE Trans. Circuits and Systems, Vol. 37, No. 10, pp. 1217-1230, 1990.]

[Lei83] C.E. Leiserson, F.M. Rose, J.B. Saxe, "Optimizing synchronous circuits by retiming", Proceedings of Third Conference on VLSI, pp. 23-36, Computer Science Press, 1983.

[Lei91] C.E. Leiserson, J.B. Saxe: "Retiming Synchronous Circuitry", Algorithmica, Vol. 6, pp. 5-35, 1991

[Knu73] D.E. Knuth: "The art of computer programming, Vol. 3: Sorting and searching", Addison-Wesley, Reading, M

[Mas87] H. Massalin: "Superoptimizer: A look at the Smallest Program", Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), SIGPLAN Notices, Vol. 22, No. 10, pp. 122-127, 1987.

[McF83] M.C. McFarland, A.C. Parker: "An Abstract Model of Behavior for Hardware Descriptions", IEEE Transaction on Computers, Vol. 32, No. 7, pp. 621-636, 1983.

[McF90] M.C. McFarland, A.C. Parker, R. Camposano: "The High-Level Synthesis of Digital Systems"", Proceedings of the IEEE, Vol. 78, No. 2, pp. 301-317., February 1990.

[Mes88] D. Messerschmitt, "Breaking The Recursive Bottleneck", in *Performance Limits in Communication Theory and Practice,* Kluwer Academic Publishers, 1988.

[Mir79] G.S. Miranker: "The use of conflicts in the translation and optimization of hardware description languages", Ph. D. dissertation, MIT, May 1979.

[Mit72] S.K. Mitra, R.J. Sherwood: "Canonic realization of digital filters using the continued fraction expansion", IEEE Trans. on Audio Electroacoustics", Vol. 21, pp. 185-194, 1972.

[Mit73] S.K. Mitra, R.J. Sherwood: "Digital ladder networks", IEEE Trans. on Audio Electroacoustics", Vol. 21, pp. 185-194, 1972.

[Mul76] C.T. Mullis, R.A. Roberts: "Synthesis of minimum roundoff noise fixed-point digital filters", IEEE Trans. Circuits and Systems, Vol. 23, No. 9, pp. 551-561, 1976.

[Nus82] H.J. Nussbaumer: "Fast Fourier transform and convolution algorithms", 2nd edition, Springer-Verlag, Berlin, Germany, 1982.

[Opp89] A.V. Oppenheim, R.W. Shafer: "Discrete-time Signal Processing", Prentice Hall, Englewood Cliffs, NJ, 1989.

[Ort90] J.M. Ortega: "Numerical analysis: a second course", Philadelphia: Society for Industrial and Applied Mathematics, 1990.

[Par88] N. Park, A.C. Parker: "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Trans. on CAD*, Vol 7, No. 3, pp. 356-370, 1988.

[Pat89] D.A. Patterson, J.L. Henessy: "Computer architecture: a quantitative approach", San Mateo, Calif. : Morgan Kaufman Publishers, 1989.

[Pau89] P.G. Paulin, J.P. Knight: "Force -Directed Scheduling for the Behavioral Synthesis of ASIC", *IEEE Transaction on CAD*, Vol 8., No 6, pp. 661-679, 1989.

[Pau92] P.G. Paulin: "DSP Design Tool Requirements for the Nineties: An Industrial Perspective", Submitted to the High Level Synthesis Workshop.

[Pit90] I. Pitas, A.N. Venetsanopoulos: "Nonlinear Digital Filters", Kluwer Academic Publishers, Boston, MA, 1990.

[Pot91a] M. Potkonjak: "Algorithms for High Level Synthesis Resource Utilization Based Approach", *Ph.D. Thesis*, University of California at Berkeley, 1991.

[Pot91b] M. Potkonjak, J. Rabaey: "Optimizing the Resource Utilization Using Transformations", *Proc. IEEE ICCAD Conference*, Santa Clara, November 1991.

[Pot92a] M. Potkonjak, J. Rabaey: "Pipelining: Just Another Transformation", IEEE ASAP-92, pp. 163-175, 1992.

[Pot92b] M. Potkonjak, J. Rabaey: "Maximally Fast and Arbitrarily Fast Implementation of Linear Computations", IEEE International Conference on Computer-Aided Design, Santa Clara, CA, paper 6c.4, November 1992.

[Rab90] J. Rabaey, and M. Potkonjak, "Resource Driven Synthesis in the HYPER system," *ISCAS-90*, vol. 4, pp. 2592-2595, New Orleans, LA, May 1990.

[Rab91] J. Rabaey, C. Chu, P. Hoang, M. Potkonjak: "Fast Prototyping of Data Path Intensive Architecture", *IEEE Design and Test*, Vol. 8, No. 2, pp. 40-51, 1991.

[Rao84] S.K. Rao, T. Kailath: "Orthogonal digital filters for VLSI implementation", IEEE Trans. Circuits and Systems, Vol. 31, No. 11, pp. 933-945, 1984.

[Rao90] K.R. Rao, P. Yip: "Discrete cosine transform: algorithms, advantages, applications", Academic Press, Boston, MA, 1990.

[Sno78] E.A. Snow, D.P. Siewiorek, D.E. Thomas: "A technology-relative computer-aided design system: abstract definition, transformations and design tradeoffs", 15th ACM/ IEEE Design Automation Conference, pp. 220-226, 1978.

[Tri87] H. Trickey: "Flamel: A high-Level Hardware Compiler", IEEE Transaction on CAD, Vol. 6, No. 2, pp. 259-269, 1987.

[Vai85] P.P. Vaidyanathan, S.K. Mitra: "A general family of multivariable digital lattice filters", IEEE Trans. Circuits and Systems, Vol. 32, No. 12, 1985.

[Wai84] W.M. Waite, G.Goos: Compiler construction, New York: Springer-Verlag, 1984.

[Wal89] R.A. Walker, D.E. Thomas: "Behavioral Transformation for Algorithmic Level IC Design" *IEEE Trans. on CAD*, Vol 8. No.10, pp. 1115-1127, 1989.

[Wal91] R.A. Walker, R. Camposano: "A Survey of High-Level Synthesis Systems", Kluwer Academic Publishers, Boston, Ma, 1991.

[Wol91a] W. Wolf, A. Takach, T-C. Lee: "Architectural Optimization Methods for Control Dominated Machines." in W. Wolfe R. Camposano: "High-Level VLSI Synthesis", Kluwer Academic Publishers, pp. 231-254., 1991.

[Wol91b] M. Wolfe: "The Tiny Loop Restructuring Research Tool", 1991 International Conf. on Parallel Processing, Vol. II, pp. 46-53, 1991.

5

The MARS High-Level DSP Synthesis System

Ching-Yi Wang and Keshab K. Parhi

Department of Electrical Engineering
University of Minnesota
Minneapolis, MN 55455

ABSTRACT

This paper addresses high-level synthesis methodologies for dedicated digital signal processing (DSP) architectures used in the Minnesota ARchitecture Synthesis (MARS) design system. We present new concurrent scheduling and resource allocation algorithms which exploit inter-iteration and intra-iteration precedence constraints. These novel algorithms implicitly perform algorithmic transformations such as pipelining and retiming, and produce solutions which are as good as or better than those previously published. Previous synthesis systems have focused on DSP algorithms which have single or lumped delays in the recursive loops. In contrast, MARS is capable of generating valid architectures for algorithms which have randomly distributed delays. MARS exploits these delays to produce more efficient architectures and allows our system to be more general. We are able to synthesize architectures which meet the iteration bound of any algorithm by unfolding the original data flow graph.

This research was supported in parts by Texas Instruments, the office of Naval Research under contract number N00014-91-J-1008, and the Army Research Office under contract number DAAL03-90-G-0063.

1. INTRODUCTION

High level synthesis of dedicated architectures for real-time digital signal processing (DSP) systems is becoming a more common and crucial task because many applications are requiring higher sample rates which can only be implemented by dedicated architectures. Even low to moderate sample rate systems are implemented with dedicated architectures to meet low power and area requirements. The objective of high level architecture synthesis is to design a valid architecture from an algorithmic description while using realistic technology constraints. The resultant architecture must maintain the original functionality while meeting the speed requirement and minimizing the area. Although architecture synthesis is becoming more common and applied more widely, it is still a difficult and NP-complete problem. Therefore many heuristic approaches have been proposed [1-8]; however, the quality of the performance of each heuristic can differ from one benchmark to another. The main task in high-level synthesis is the scheduling of algorithmic operations to iteration time partitions and the allocation of hardware operators to implement the operations. DSP algorithms are repetitive in nature; therefore, most scheduling algorithms attempt to exploit concurrency among the algorithmic operations to reduce the iteration period and/or the amount of hardware needed.

Some previous synthesis approaches include the simplest technique known as 'As Soon As Possible' (ASAP) scheduling and allocation where the algorithmic operations are ordered according to their precedence constraints and are then scheduled from the first iteration time partition to the last [1]. The iteration time partition is defined to be the time step at which a task is executed modulo the iteration period. For example, if the iteration period is 16 units, then there are 16 iteration time partitions (0 through 15). If a task is scheduled for time step 18, then the task is assigned to time partition 2 (which is 18 modulo 16). ASAP and similar methods generally do not produce very good results because they use a global one-time ordering of the nodes. A more complex class of algorithms utilize list scheduling techniques. Operations are first sorted into an ordered list according to some local priority function. The sorted operations are then iteratively scheduled into an iteration time partition until the hardware resources are exhausted. Then operations are re-ordered and scheduled into the next time partition [2]. Another technique is force directed scheduling where force values for all operations at all feasible time partitions are calculated. Then the operations with the least force values at a time partition are scheduled [3]. More recently, integer linear programming techniques have been applied to the synthesis problem [4]. This paper presents the methodologies for high level synthesis of dedicated DSP architectures using the Minnesota ARchitecture Synthesis (MARS) system. MARS is capable of producing results as good as or better than previous synthesis systems [9,10].

Previous synthesis systems have concentrated on simple DSP algorithms where all recursive loops contain single or lumped delays. To design better DSP architectures for real-time applications, one needs to consider a wider range of algorithms. Little work has been performed in the area of randomly distributed delays within recursive loops [11]. MARS can synthesize valid architectures for simple algorithms with lumped loop delays as well as complex algorithms with randomly distributed delays. In some cases the resultant design cannot meet the required sample rate unless an algorithmic transformation is applied. If the algorithmic description requires a fractional iteration period (e.g., $T = \frac{3}{2}$ units), all previous synthesis systems will generate an architecture with an iteration period of 2 units. MARS is able to apply an *unfolding* transformation and design an architecture which processes multiple iterations over a longer iteration period (e.g. 2 iterations over $T = 3$ units) [12]. Thus the architecture is able to achieve the required sample rate.

Multiprocessor schedules can be non-overlapped, fully-static overlapped, or cyclo-static overlapped. A non-overlapped schedule contains all operations for a single iteration of the algorithm within one iteration period. A fully-static overlapped schedule contains operations of multiple iterations of the algorithm within one iteration period [12]. A cyclo-static overlapped schedule contains operations of multiple iterations of the algorithm within one iteration period; and within the multiple iterations, a different processor can be used to start each iteration [13]. Depending on the original algorithm, MARS will generate the proper type of schedule which best satisfies the algorithm's precedence constraints.

In this paper, we represent the DSP algorithm using a synchronous data-flow graph (DFG) model [14]. Within this model, each node in the DFG represents an algorithmic operation and all arcs represent communication links between the operations. Any arc $U \rightarrow V$ with i delays (where i is any non-negative integer) implies that the result of the l-th iteration of U is used to execute the $(l+i)$-th iteration of V. The arcs with delays dictate the inter-iteration precedence constraints and represent concurrency between iterations. The arcs without delays represent the intra-iteration precedence constraints and dictate the concurrency between operations of the same iteration.

The organization of this paper is as follows. Section 2 presents a novel *iterative loop based* concurrent scheduling and resource allocation algorithm for operations located within recursive loops (recursive nodes). Section 3 presents the scheduling and resource allocation algorithm for operations not found in recursive loops (non-recursive nodes). In section 4, we present an algorithmic transformation technique used in MARS to achieve the maximum sample rate for certain cases. Section 5 describes the way MARS converts the final schedule into an architecture. We present results of MARS in section 6, show a simple example with a set of input/output files in section 7, and draw conclusions in section 8.

2. ITERATIVE LOOP BASED, CONCURRENT SCHEDULING AND RESOURCE ALLOCATION ALGORITHM FOR RECURSIVE NODES

The first step of any synthesis system is scheduling and resource allocation; therefore, we first present our iterative loop based concurrent scheduling and resource allocation algorithm. To improve the quality of the schedule and reduce the number of processors, MARS implicitly retimes and pipelines the flow graph as it schedules the DFG [15]. The synthesis of a high sample rate system is restricted by the recursive sections of any system. Recursion negates the most obvious ways of improving the performance of the final architecture. This is because the computational latency associated with the feed-back loops limits the opportunities for pipelining and/or parallel processing. The non-recursive sections are less restrictive because one can always place latches across any feed-forward cutset (at the expense of greater latency) to achieve the desired level of pipelining. Loops cannot be pipelined to any arbitrary level by simply inserting latches because the pipelining latches would change the number of delays in the loops and, hence, the original functionality of the DFG. Therefore, the recursive loops set the maximum sample rate for an architecture. In this section we describe MARS's major steps of scheduling and resource allocation for recursive nodes.

2.1. Step 1: Loop Search

The data input file to MARS specifies the operations, the DFG, the technology constraints for each processor type, and a user defined iteration period. From this information and before initiating the loop search, MARS generates some preliminary information to be used throughout the rest of the program. First MARS locates

the maximum computation time, T_{cmax}, of all operations within the DFG:

$$T_{cmax} = \left\{ MAX\left[T_{c_i}\right] : \text{for } all\ i \right\}.$$

where T_{c_i} = the computation time for operation type i.

If T_{cmax} is greater than the iteration period, MARS sets the iteration period equal to T_{cmax}. This is to satisfy the constraint that the operation with the greatest computation time must complete within one iteration period. Next create the precedence graph of all recursive and non-recursive nodes by removing any arc which contains delays.

Definition 2.1: The *precedence graph height* (PGH) value represents a relative location of a node within the precedence graph as referenced from the bottom of the precedence graph.

All nodes of the precedence graph are assigned PGH values. MARS determines the PGH values by first assigning a value 0 to all nodes without any successor nodes (i.e., the nodes at the bottom of the precedence graph). Then all of their immediate predecessors will be assigned a PGH value of 1. PGH values continue to increase until the top level nodes are reached. For example consider a node which has a PGH value of c, each of its immediate predecessor nodes will be assigned the value c + 1 iff all other successors of this node have PGH values less than or equal to c. Consider a section of a precedence graph as shown in Fig. 1, we see that nodes 8 and 2 do not have any successor nodes; therefore, their PGH values are 0. Nodes 7 and 3 have PGH values equal to 1, since they are the immediate predecessors of nodes 8 and 2 respectively. Node 4 will have a PGH value of 2. Now, to calculate the PGH value for node 5, we first see that node 7 has a PGH value of 1; therefore, the PGH value assigned to node 5 will initially be equal to 2. However, node 4 has a PGH value of 2. Therefore the PGH value assigned to node 5 will be 3.

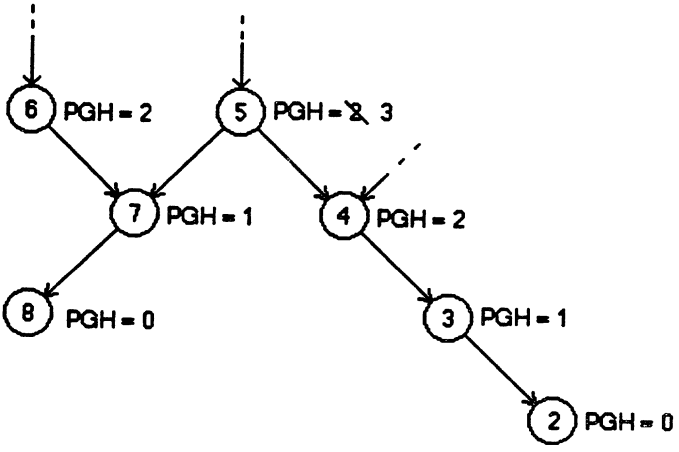


Figure 1: An example to show the calculation of PGH values. Node 5 was initially set to 2, but node 4 has a PGH = 2; therefore, node 5 has a PGH = 3.

Now we may begin the search for all loops within the flow graph. When a loop is located, MARS calculates the loop's loop bound [16] as follows:

$$T_{LB_j} = \frac{T_{L_j}}{D_{L_j}}.$$

where T_{L_j} = the loop computation time of loop j and
D_{L_j} = the number of loop delays within loop j.

One can calculate a lower bound on the iteration period from the loop bound values by locating the maximum loop bound. This lower bound is known as the iteration bound [12,13,16]. The complexity for obtaining all of the loops is linear in the number of nodes plus edges; however, in the worst case the number of loops can be exponential in the number of nodes [17]. Next, MARS orders the loops by decreasing *criticalness* or loop bound. Ties are broken by giving preference to the loop which contains a node with a larger PGH value. Now MARS searches the ordered set of loops for a subset of loops, L, such that all recursive nodes can be found in L. MARS first places all loops, whose loop bounds are equal to the iteration bound, into L. All nodes found in L are considered to be covered. Next, MARS choses a loop to be a member of L if the loop contains the greatest number of uncovered nodes. Other methods may be applied to minimize the number of loops in L. For each loop with a single or lumped delay(s) in L, break the loop at the delay(s) such that all nodes are ordered in decreasing PGH values. For loops with distributed delays, break the loop at each delay to create a set of *loop sections* . Each loop section will be manipulated as a separate loop, but they are still considered as part of the original loop (see Fig. 2). Here we see that loop L_1 in Fig. 2(a) contains two distributed delays. MARS breaks loop L_1 at the delays to create 2 loop sections, as shown in Fig. 2(b). The two loop sections are still considered part of loop L_1. The nodes of each loop section are also ordered by decreasing PGH values.

Example 2.1: Consider the 4-stage pipelined normalized lattice filter as shown in Fig. 3(a) which consists of 15 multiply and 11 add operations [18].We are assuming a multiply has a computation time of 2 units and an add has a computation time of 1 unit. The critical path computation time of this filter is 10 units. This filter contains a recursive section which has 8 recursive nodes (5 multiply and 3 add operations), and a non-recursive section which has 18 non-recursive nodes (10 multiply and 8 add operations). There are 3 loops in this filter:

L_1: 23-25-22-19-*4D* Loop bound = $\frac{3}{2}$

L_2: 20-21-*4D*-26-25-22-19-*4D* Loop bound = $\frac{9}{8}$

L_3: 24-21-*4D* Loop bound = $\frac{3}{4}$

MARS breaks loop L_2 into 2 loop sections:

L_{21}: 26-25-22-19-*4D*
L_{22}: 20-21-*4D*

Therefore the total number of loops and loop sections required to schedule the recursive nodes is 4: L_1, L_{21}, L_{22}, and L_3. •

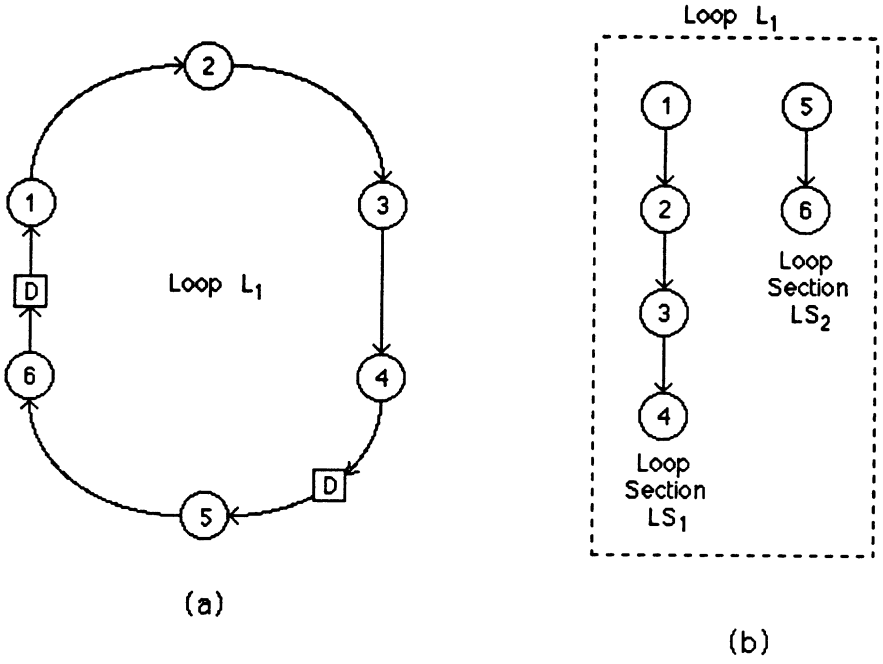


Figure 2: **(a)** This shows a loop, L_1, which has two distributed delays within the loop. **(b)** Here L_1 is broken into two loop sections, LS_1 and LS_2. MARS created the loop sections from loop L_1 by breaking the loop at the arcs which contained a delay.

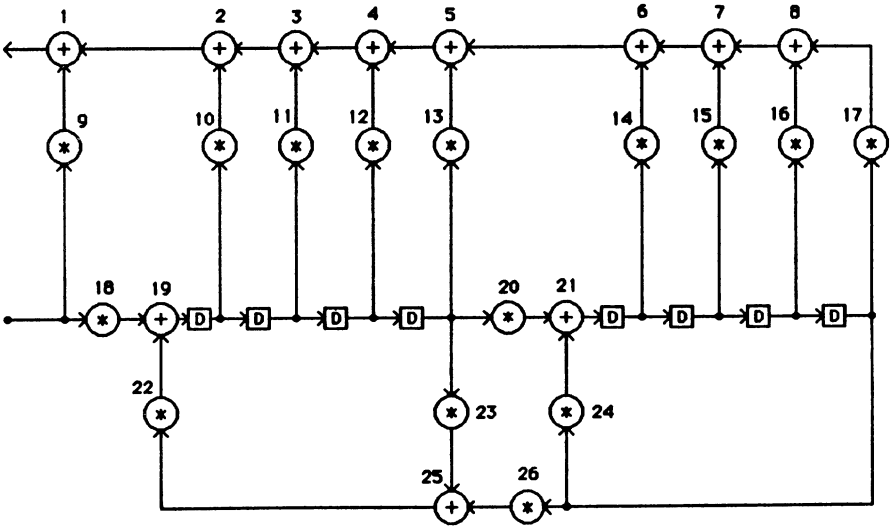


Figure 3(a): The DFG of a 4-stage pipelined, normalized lattice filter. T_A = 1 unit, T_M = 2 units, the iteration bound = 1.5 units, and T_{CRIT} = 10 units.

Example 2.2: Consider the 5th-order wave digital elliptic filter benchmark as shown in Fig. 3(b) which consists of 8 multiplication and 26 addition operations [1,21]. We are assuming the same parameters as in example 2.1. The critical path computation time of this filter is 17 units. This is a completely recursive system that contains 45 loops. MARS only requires 5 loops to schedule all of the nodes:

L_1: 2-7-16-17-21-15-8-10-6-3-5-1-4-*D*	Loop bound = 16
L_2: 7-16-17-21-15-8-10-6-9-11-14-12-13-*D*	Loop bound = 16
L_3: 24-17-22-18-23-27-25-26-28-31-29-30-*D*	Loop bound =15
L_4: 24-17-22-18-23-27-25-32-34-33-*D*	Loop bound = 13
L_5: 16-17-21-15-20-19-*D*	Loop bound = 7

•

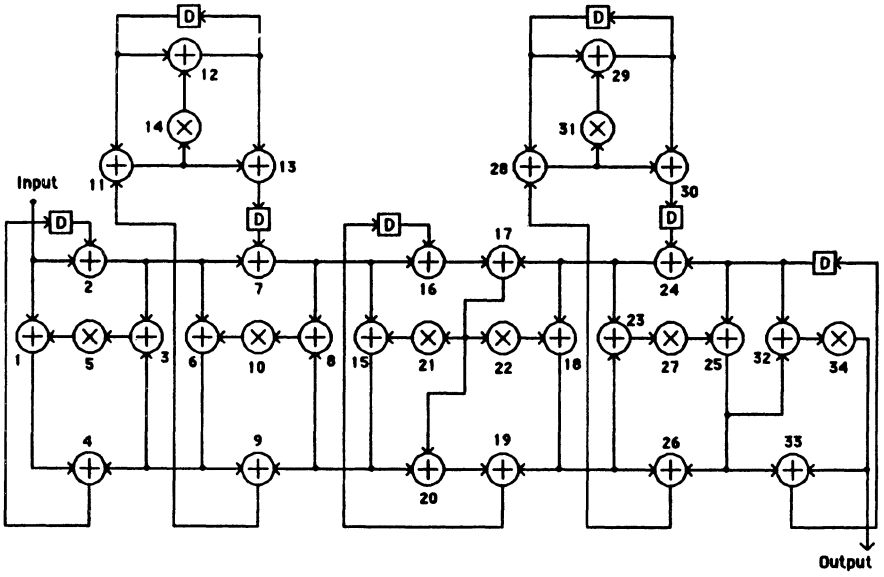


Figure 3(b): The DFG of the 5th-order wave digital elliptic filter. T_A = 1 unit, T_M = 2 units, the iteration bound = 16 units, and T_{CRIT} = 17 units.

2.2. Step 2: Initial Schedule

The iteration bound is the lower bound on the iteration period for any flow graph. If the iteration bound is greater than the iteration period, then we set the iteration period equal to the iteration bound. Next calculate the lower bound for the number of processors required for each operation type by the following:

$$(Lower\ Bound)_U = \left\lceil \frac{N_U * T_U}{P_U * T} \right\rceil$$

where N_U = number of U type operation,
T_U = computation time of a U type operation,
P_U = pipelining level of a U type processor and
T = iteration period.

Now create the schedule matrices, one matrix for each operation type. For example, consider a DFG consisting of addition operations and multiplication operations. There will be 2 schedule matrices: one for addition and one for multiplication operations (see examples 2.3 and 2.4). Rows of a schedule matrix represent iteration time partitions, and columns represent loops and loop sections. Each loop or loop section is assigned to a set of columns such that the first set of columns corresponds to the most critical loop and the last set to the least critical loop. We do this to reflect the order of the loops within L onto the schedule matrices. MARS then creates an initial schedule by first scheduling the nodes of the most critical loop and then by iteratively scheduling the other loops. The algorithm MARS uses to choose the next loop to be scheduled is as follows:

```
while (more unscheduled loops exist in set L) {
    Place into set L_p all unscheduled loops which contain at least one previously scheduled node.

    If (set L_p is the null set) {
        Place into set L_p all unscheduled loops which have an intra-iteration precedence constraint with a previously scheduled node.

        If (set L_p is the null set)
            Place all unscheduled loops into set L_p.
    }

    Locate loop L_s within set L_p such that L_s has the largest loop bound or equivalently, L_s is the most critical loop of set L_p.

    Schedule the unscheduled nodes of loop L_s into the schedule matrices.

    Clear set L_p
}
```

When scheduling a loop, MARS adds the unscheduled nodes to the schedule matrix while maintaining the intra-iteration precedence constraints. The scheduling of the DFG's most critical loop is the easiest because MARS has not yet scheduled any nodes. Therefore, starting from time 0, MARS schedules the nodes of the most critical loop. Loops which contain previously scheduled nodes must be scheduled in a manner which maintains the proper precedence constraints. By locating the positions of the previously scheduled nodes within the loop L_s, MARS has a reference point to begin adding the unscheduled nodes into the matrices. Loops which only have a precedence constraint to a previously scheduled node may have multiple constraints to previously scheduled nodes. Before MARS can begin the scheduling of the nodes of such loops, it must locate the proper constraint which will ensure that the other constraints will be satisfied. By choosing the tightest precedence constraint, which minimizes the difference in PGH values, as the reference point to begin the scheduling process for the nodes of loop L_s, MARS is able to satisfy all other precedence constraints. Thus we connect an unscheduled node with the largest PGH value to a previously scheduled node. Note that the minimum difference in PGH values directly corresponds to the tightest constraint between the scheduled loops and loop L_s.

The loops which are completely independent of the previously scheduled nodes have no initial reference point from which to start the scheduling of their nodes. Because these loops are independent, they can be scheduled from any starting point. Therefore MARS begins the scheduling of such loops from a starting time determined by:

$$\textit{Starting time partition for } L_s = PGH_{\max} - PGH_{top}.$$

where $PGH_{\max}$ = maximum PGH value of the precedence graph and
PGH_{top} = largest PGH value within loop L_s.

In some cases when MARS schedules a loop, the loop may "wrap" around the matrices. Nodes of such loops are represented with a superscript of $\pm l$. The resultant value of such nodes in the n-th iteration will be used to compute the $(n \pm l)$-th iteration.

Definition 2.2: The *loop flexibility* available for each member of set L defines the number of iteration time partitions that nodes of a loop may be shifted before violating the inter-iteration precedence constraint. MARS calculates the loop flexibility for each loop of L by the following equation:

$$F = T * D_L - T_L$$

where F = the flexibility,
T = iteration period,
D_L = number of loop delays and
T_L = loop computation time.

Definition 2.3: A *frozen* node is defined as a node which belongs to a loop with zero flexibility.

Property 2.1: Frozen nodes cannot be shifted to another time partition without violating an intra- or inter-iteration precedence constraint.

For every loop with zero flexibility, MARS marks all nodes of that loop as frozen nodes. Next, MARS determines if the lower bound on the number of processors for each operation type is capable of processing all of the frozen nodes assigned to each time partition. If not, MARS will allocate more processors to handle the frozen nodes.

Example 2.3: The filter of example 2.1 has an iteration bound equal to 1.5 units. Since this is less than the multiply computation time, the iteration period is set to 2. The lower bound on the number of multiplication and addition processors for all nodes are equal to 8 and 6 respectively. We also assume the add processors are pipelined by 1 stage and the multiply processor by 2 stages. Fig. 4(a) shows the initial schedule of the filter. Note that the nodes which are marked by -1 and -2 represent the operations of the (n-1)-st and the (n-2)-nd iterations. The flexibility for loops L_1, L_2, and L_3 are 2, 7, and 5 respectively. •

Example 2.4: The filter of example 2.2 has an iteration bound of 16 units. Assume we use an iteration period equal to the iteration bound, the lower bound on the number of multiplication and addition processors are equal to 1 and 2 respectfully. We also assume the same technology constraints as in example 2.3. Fig. 4(b) shows the initial schedule of the filter. Note that the nodes which are marked by -1 represent the operations of the previous iteration. All nodes of loops L_1 and L_2 are frozen because their loop bounds equal the iteration period. Therefore the loop flexibility for each loop equals 0. The flexibility for loops L_3, L_4 and L_5 are 1, 3, and 9 respectively. •

MULTIPLICATION

time step	L_1			L_{21}			L_{22}			L_3		
0	23			26				20^{-1}			24^{-1}	
1		22^{-1}										

ADDITION

time step	L_1			L_{21}			L_{22}			L_3		
0		25^{-1}							21^{-2}			
1			19^{-2}									

Figure 4(a): The initial schedule for the recursive nodes of the lattice filter of Fig 3(a). Note that this is also the final conflict free schedule for the recursive nodes of the lattice filter.

2.3. Step 3: Resolve Conflicts

The basic concept which MARS uses to reduce the total number of processors is shown in Fig. 5. We see that MARS divides the initial schedule matrices into two halves. The upper half represents earlier time partitions; the lower half represents later time partitions. The initial schedule contains only one iteration and minimal overlap of the neighboring iterations. MARS will "stretch" or shift the schedule in two directions to increase the overlap between multiple iterations and to reduce the number of nodes involved in each shift. We can increase the overlap by exploiting the inter-iteration precedence constraints and, therefore, help generate better schedules. Previous precedence graph based scheduling methods did not consider the inter-iteration precedence constraints. Furthermore, by ordering the schedule matrices by decreasing order of hardware area cost, MARS is able to reduce the overall area by attempting to fully utilize the allocated processors of the more expensive operation types. MARS attempts to minimize the number of allocated area expensive processors, by using the loop flexibility initially on the more expensive operation types.

Definition 2.4: A *scheduling conflict* occurs at an iteration time partition when more nodes are scheduled at that time partition than processors are available. MARS recognizes two types of conflicts: *primary conflicts* and *secondary conflicts*. A primary conflict is a conflict between nodes which are members of loops with zero flexibility. Some nodes involved in secondary conflicts belong to loops with positive flexibility.

	Add					Multiply				
time step	L_1	L_2	L_3	L_4	L_5	L_1	L_2	L_3	L_4	L_5
0	2	13^{-1}	30^{-1}							
1	7									
2	16		24							
3	17									
4						21		22		
5										
6	15		18							
7	8		23		20					
8					19	10		27		
9										
10	6		25							
11	3	9	26	32						
12		11	28			5			34	
13							14	23		
14	1			33						
15	4	12	29							

Figure 4(b): The initial schedule of the elliptic filter of Fig 3(b). Note that all nodes are recursive nodes in this example.

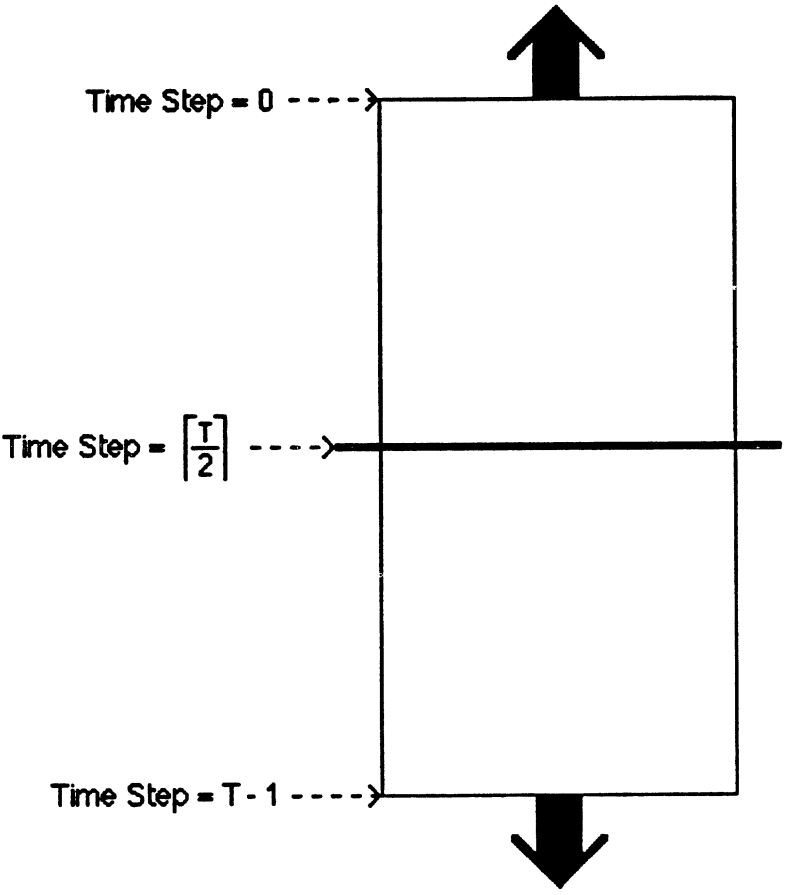


Figure 5: A conceptual view of how MARS "stretches" the schedule to resolve conflicts and to reduce the number of allocated processors.

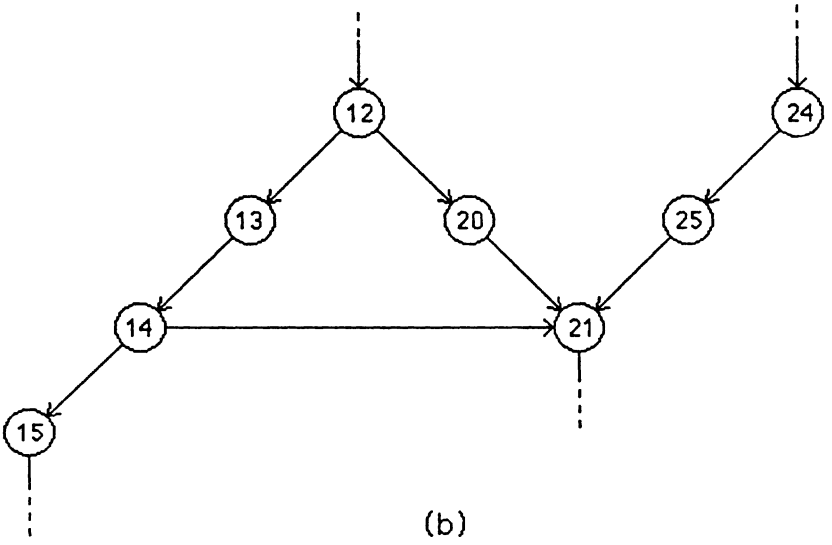
MARS begins with the most expensive operation's schedule matrix and locates a time partition with a non-primary scheduling conflict. Then MARS must choose a node for shifting to resolve the conflict. First MARS searches all of the nodes involved in the non-primary conflict for all non-frozen nodes. Of the non-frozen nodes, MARS chooses the node which belongs to the more critical loop.

Definition 2.5: *Hidden flexibility* is the flexibility between a node and its immediate predecessor or successor. This flexibility is only available to that node and not to the loop.

For example, consider the partial schedule matrix of Fig. 6(a) and its corresponding partial precedence graph as seen in Fig. 6(b). We see that the intra-iteration precedence constraint from node 12 to node 20 is satisfied. We also note that if node 20 were scheduled at time partition 6, the precedence constraint will still be satisfied. Therefore, there is hidden flexibility available to node 20.

Time Step	L_1	L_2	L_3
⋮	⋮	⋮	⋮
5	12		
6	13		24
7	14	20	25
8	15	21	
⋮	⋮	⋮	⋮
F:	0	0	1

(a)



(b)

Figure 6: (**a**) This is a partial schedule segment to demonstrate the idea of hidden flexibility. F = the loop flexibility for each loop. (**b**) This is the corresponding precedence graph for the partial schedule segment of Fig. 6(a).

Property 2.2: Any node which is a member of a loop with zero flexibility is only *temporarily frozen* iff there is hidden flexibility available to it.

If MARS cannot locate a non-frozen node, it searches the nodes involved in the non-primary conflict for all temporarily frozen nodes. From this set of nodes, MARS chooses the node which belongs to the more critical loop. If all nodes are frozen (i.e., a primary conflict), MARS allocates another processor.

If the selected node lies in the time partition range of 0 to $\lceil T/2 \rceil - 1$ (T is the iteration period), MARS attempts to shift the node and any predecessors to an earlier time partition.

Definition 2.6: *Wrapped nodes* are nodes which are scheduled at time steps which are either negative or greater than or equal to T (i.e., $t < 0$, or $t \geq T$). *Non-wrapped nodes* are nodes which are scheduled at time steps equal to the time partitions (i.e., $0 \leq t \leq T-1$). Wrapped nodes are identified in the schedule with a superscript of $\pm l$.

If the node is a non-wrapped node, MARS tries to exploit the hidden flexibility available to the node or to one of its predecessors before attempting the shift using the loop flexibility. For wrapped nodes with the "$-l$" superscript, only the hidden flexibility shift to an earlier time partition is considered at this point. A shift of the node and its predecessors to new time partitions should not create any primary conflicts. Therefore, if shifting to an earlier time partition is not possible, MARS attempts the shift to a later time partition by first trying to exploit the hidden flexibility available to the node or to one of its successors. If no hidden flexibility exists, MARS attempts to shift the node and its successors by using the loop flexibility. Again, the shift should not create any primary conflicts.

For nodes which lie in the time partition range of $\lceil T/2 \rceil$ to $T - 1$, MARS first attempts to exploit any hidden flexibility of the node or one of its predecessors for shifting to an earlier time partition. If this does not resolve the conflict, it attempts to use the hidden flexibility available to the node or one of its successors for shifting to a later time partition. Should the conflict still remain unresolved and if the node is a non-wrapped node MARS will use the loop flexibility to shift the node and its successors to a later time partition. Otherwise, MARS will attempt to shift the node and its predecessors to an earlier time partition. MARS does this because the wrapped nodes with the "$+l$" superscripts should continue to be shifted to earlier time partitions. If MARS cannot resolve the conflict after attempting to shift all non-frozen and temporarily frozen nodes, MARS allocates a new processor. MARS repeats the process of locating a conflict and then resolving it until a conflict-free schedule is generated for the recursive nodes.

Example 2.5: Continuing the previous filter of example 2.3, we see that the initial schedule is already conflict free; therefore the final, conflict-free schedule is the same initial schedule as shown in Fig. 4(a). •

Example 2.6: Continuing the previous filter example 2.4, we see that MARS is capable of generating a valid conflict-free schedule as shown in Fig. 7. Note that with the iteration period equal to the iteration bound, a third addition processor was allocated due to the tight intra-iteration precedence constraints. •

time step	A_1	A_2	A_3	M
0	2	13^{-1}	29^{-1}	
1	7	33^{-1}	30^{-1}	
2	16		24	
3	17			
4				21
5				22
6	15			
7	8	20	18	
8		19	23	10
9				27
10	6			
11	3	9	25	
12	32	11	26	5
13			28	14
14	1			31
15	4	12		34

Figure 7: The final conflict free schedule for the elliptic filter of Fig. 3(b). Note that with an iteration period equal to the iteration bound, a third processor was allocated because of the tight precedence constraints.

3. SCHEDULING AND RESOURCE ALLOCATION FOR NON-RECURSIVE NODES

Because non-recursive sections are more flexible than recursive sections (one can always pipeline the non-recursive sections at the feed-forward cutsets at the expense of latency), MARS performs the scheduling and resource allocation for non-recursive nodes after all recursive nodes have been scheduled and all conflicts among the recursive nodes have been resolved. Non-recursive paths do not impose limits on achieving the desired iteration period except that the iteration period has to be greater than or equal to the maximum computation time of all operations within the DFG. Most previous critical path scheduling systems were based on earliest and latest execution time; however, for MARS the latest execution time partition can be as large as required. MARS reduces the number of pipeline delays by satisfying as many intra-iteration precedence constraints as possible and by exploiting the inter-iteration precedence constraints. Since all recursive systems must have internal algorithmic delays, MARS reduces the number of pipelining stages by first using these internal delays through implicit retiming before introducing any pipeline stages. At this point, MARS considers all recursive nodes to be frozen. They will not be involved in any shifts when conflicts are being resolved. In this section, we describe MARS's major steps for scheduling and resource allocation of the non-recursive nodes.

3.1. Step 1: Calculate Minimum Number of Processors

The scheduling of non-recursive nodes begins with the identification of all non-recursive nodes of the flow graph and the number of unused time partitions for each allocated processor. MARS also checks if the iteration period is valid in the same manner as described in section 2.1. With this information, the exact number of additional processors can be calculated by:

$$(Processor)_U = \left\lceil \frac{\left(N_U - TS_U\right) * T_U}{P_U * T} \right\rceil$$

where N_U = number of type U operations within the non-recursive section,
TS_U = number of available time partitions in the type U processors,
T_U = computation time of the type U operation,
P_u = pipelining level of the type U operation and
T = iteration period.

Example 3.1: Consider the filter of example 2.1. We have already shown the scheduling of the recursive nodes. Now we schedule the non-recursive nodes. From example 2.5, we see that there are 11 time partitions available for multiply operations and 9 time partitions available for add operations. There are 18 non-recursive nodes: 10 multiplication and 8 addition operations. MARS determined that this filter will not require any new processors. Therefore the number of processors required will equal the lower bound of 8 hardware multiplication operators and 6 hardware addition operators. •

Example 3.2: Consider the 16-point FIR filter shown in Fig. 8. This example has been studied extensively [3,5,7]. We are assuming the same parameters as in the earlier examples. This simple filter consists of 8 multiplication and 15 addition operations and has a critical path computation time of 10 units. All operations are in the feed-forward paths; therefore there is no recursive section in this filter. In this example, the iteration period is assumed to be 3 units, and MARS determined that it will require exactly 5 addition and 3 multiplication hardware operators. •

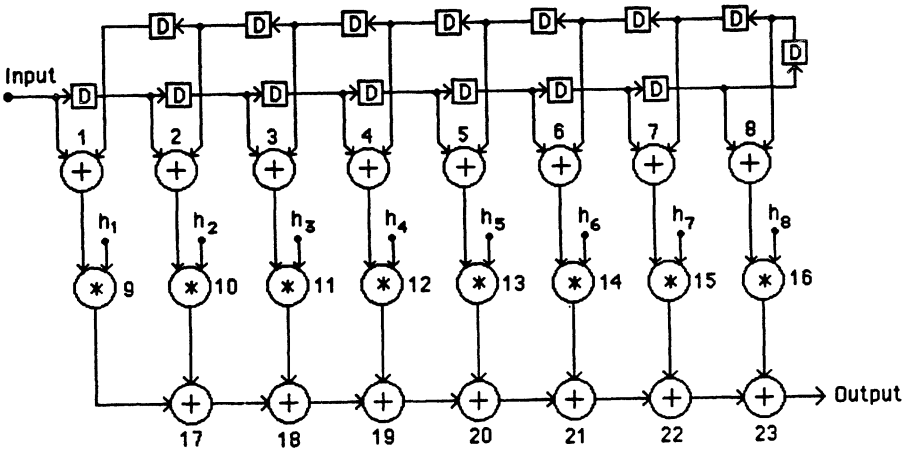


Figure 8: The DFG of a 16-point FIR filter. T_A = 1 unit, T_M = 2 units, and T_{CRIT} = 10 units.

3.2. Step 2: Locate Feed-Forward Paths

MARS locates all feed-forward paths which only consist of non-recursive nodes and calculates their path computation time. If a path begins with or ends on a recursive node, the recursive node is added to the path only as a starting or ending reference point. The recursive node computation time is not included in the path computation time. For paths which do not begin or end with a recursive node, search for a precedence constraint to a recursive node. There are three cases to consider: 1) all non-recursive nodes are predecessors of the recursive nodes; 2) all non-recursive nodes are successors of recursive nodes, and 3) some non-recursive nodes are predecessors and some are successors of recursive nodes. For the first two cases, MARS flags the precedence constraint which minimizes the difference in PGH values between the recursive and non-recursive node (see Fig. 9(a) and 9(b)). Fig. 9(a) shows an example where all non-recursive nodes are predecessors of the recursive nodes. We can see that the precedence constraints of node 7 to node 2, node 8 to node 4, and node 8 to node 5 have a difference of PGH values of 1, 1, and 2 respectively. Therefore MARS flags the precedence constraint of node 7 to node 2. One can easily see that by referencing the unscheduled nodes from the flagged precedence constraint, all other precedence constraints are satisfied. Fig. 9(b) shows an example where all non-recursive nodes are successors of the recursive nodes. From this figure, we can easily see that the flagged precedence constraint will satisfy all other precedence constraints. These flagged precedence constraints will be used as reference points when MARS begins the scheduling of the nodes that belong to the path.

The third case can have two possible configurations. If the non-recursive path contains a precedence constraint to a recursive node and later along the path has a precedence constraint from a recursive node (see Fig 9(c)), MARS breaks the path into 2 paths to maintain the proper precedence constraint. Here we see that this example can only satisfy all intra-iteration precedence constraints if the path is split into two separate paths. One path will end on a recursive node and the other will begin with a recursive node as shown in Fig. 9(d). Note that the precedence constraint from node 7 to node 8 is still satisfied. The other possible configuration

(where a precedence constraint from a recursive node to the path occurs first and is then followed by a precedence constraint from the path to a recursive node) generates a loop and therefore it will not occur. Next MARS sorts the paths by decreasing path computation time and for ties, MARS gives preference to the path which contains a node with a larger PGH value. From this ordered set of paths, MARS locates a subset of paths, FP, such that FP contains all of the non-recursive nodes. The choice of a path to be included in set FP is identical to the technique described the section 2.1.

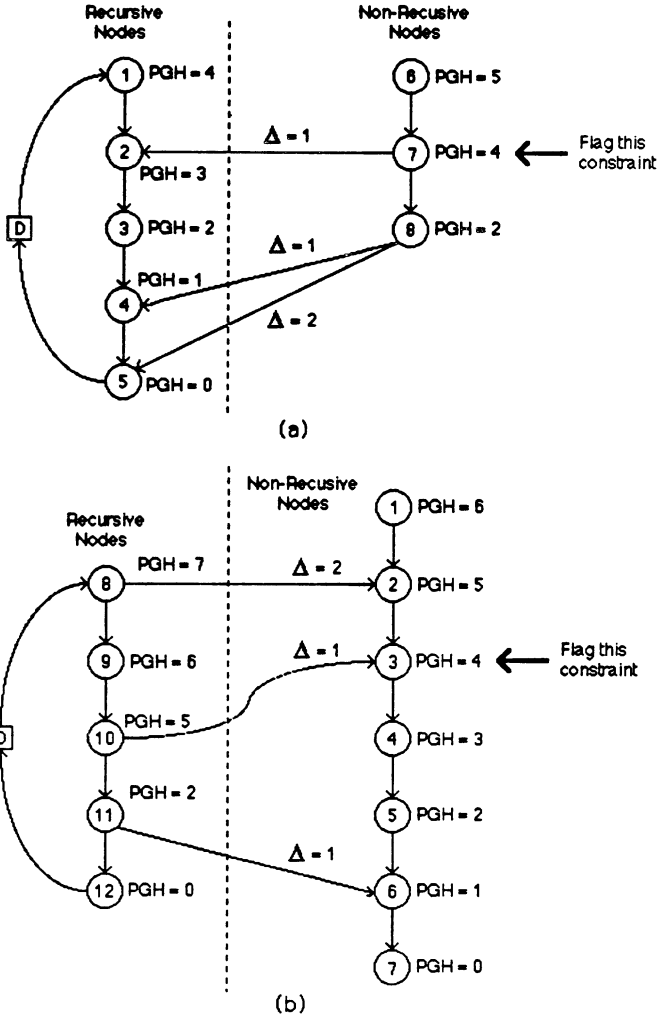
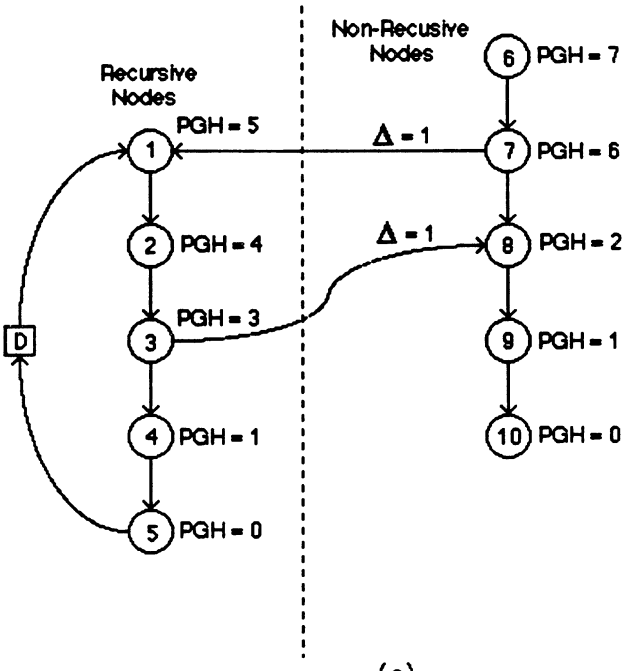


Figure 9: (**a**) A simple example where all non-recursive nodes are predecessors of the recursive nodes. The flagged precedence constraint is from node 7 to node 2 with a difference of PGH values (Δ) = 1. (**b**) A simple example where all non-recursive nodes are successors of the recursive nodes. The flagged precedence constraint is from node 10 to node 3 with a difference of PGH values (Δ) = 1.



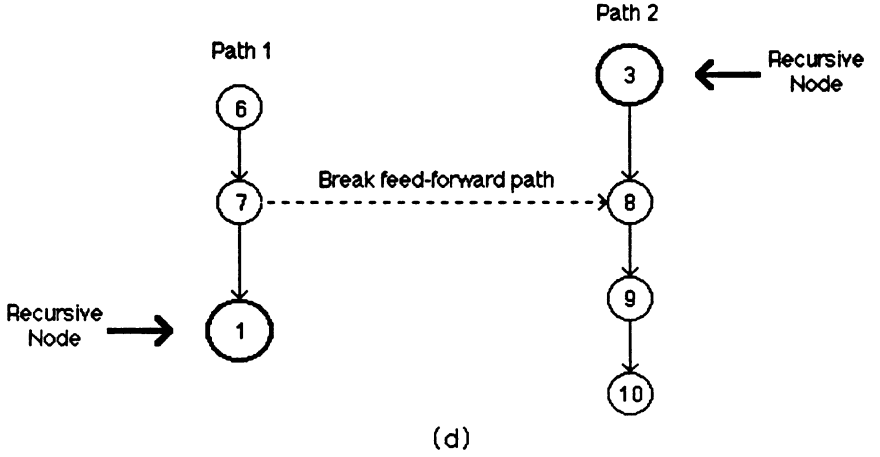


Figure 9: (c) A simple example where a non-recursive nodes has a precedence constraint to a recursive node and then later in the path, a non-recursive node has a precedence constraint from a recursive node. (d) For cases as shown in Fig. 8(c), MARS breaks the path into two separate paths as shown here. One path will end on a recursive node, node 1, and the other path will begin with a recursive node, node 3. Note that the precedence constraint from node 7 to node 8 will also be satisfied.

Example 3.3: The filter of example 2.1 contains 10 feed-forward paths which only contain non-recursive nodes:

$P_{1:}$	17-8-7-6-5-4-3-2-1	$T_{P1} = 10$
$P_{2:}$	16-8-7-6-5-4-3-2-1	$T_{P2} = 10$
$P_{3:}$	15-7-6-5-4-3-2-1	$T_{P3} = 9$
$P_{4:}$	14-6-5-4-3-2-1	$T_{P4} = 8$
$P_{5:}$	13-5-4-3-2-1	$T_{P5} = 7$
$P_{6:}$	12-4-3-2-1	$T_{P6} = 6$
$P_{7:}$	11-3-2-1	$T_{P7} = 5$
$P_{8:}$	10-2-1	$T_{P8} = 4$
$P_{9:}$	9-1	$T_{P9} = 3$
$P_{10:}$	18-19*	$T_{P10} = 2$

Example 3.4: The filter of example 3.2 contains 8 feed-forward paths:

$P_{1:}$	1-9-17-18-19-20-21-22-23	$T_{P1} = 10$
$P_{2:}$	2-10-17-18-19-20-21-22-23	$T_{P1} = 10$
$P_{3:}$	3-11-18-19-20-21-22-23	$T_{P1} = 9$
$P_{4:}$	4-12-19-20-21-22-23	$T_{P1} = 8$
$P_{5:}$	5-13-20-21-22-23	$T_{P1} = 7$
$P_{6:}$	6-14-21-22-23	$T_{P1} = 6$
$P_{7:}$	7-15-22-23	$T_{P1} = 5$
$P_{8:}$	8-16-23	$T_{P1} = 4$

3.3. Step 3: Create an Initial Schedule

To create an initial schedule for the non-recursive nodes, MARS assigns a set of columns of the schedule matrices to each path and the rows represent iteration time partitions. For non-recursive nodes, the paths are assigned to columns in reverse order. Instead of mapping the first set of columns to the most critical path (as in section 2.2), MARS assigns the first set of columns to the least critical path. The last set of columns corresponds to the most critical path. We do this because when MARS resolves conflicts, we want MARS to first exploit the intra-iteration precedence constraints before introducing any pipeline stages. When resolving conflicts, the less critical paths are chosen first since these paths have greater flexibility. At this point, each path is assigned a set of columns in which to schedule it's nodes, and the order of scheduling of the paths does not affect the column assignment. Now MARS begins the process of building the initial schedule for the non-recursive nodes. MARS searches the set FP for all paths which begin with or end on a recursive node and places them into set FP_r. From this subset of paths, MARS locates and schedules the paths in an iterative manner as described below:

while (more unscheduled paths exist in set FP_r) {

Place into set FP_p all unscheduled paths which contain at least one previously scheduled non-recursive node.

If (FP_p is the null set)

Place all unscheduled paths of set FP_r into set FP_p.

Locate the path FP_s which has the largest path computation time or equivalently, the most critical path within set FP_p. We schedule the most critical path first to minimize the number of unscheduled nodes. (Note:

Each path has already been assigned a set of columns and the order of the path scheduling does not affect the column assignment.)

Insert the unscheduled nodes of path FP_s into the schedule matrices.

Clear set FP_p

}

After all paths which begin with or end on a recursive node have been scheduled, MARS iteratively schedules the remaining paths of FP in a similar manner as described in section 2.2, except for some minor changes. The modified algorithm is as follows:

while (more unscheduled paths exists in set FP) {

Place into set FP_p all unscheduled paths which contain at least one previously scheduled non-recursive node.

If (set FP_p is the null set) {

Place into set FP_p all unscheduled paths which have a flagged precedence constraint.

If (set FP_p is the null set) {

Place into set FP_p all unscheduled paths which have an intra-iteration precedence constraint with a previously scheduled non-recursive node.

If (set FP_p is the null set)

Place all unscheduled paths into set FP_p.

}

}

Locate path FP_s within set FP_p such that FP_s is the most critical path of set FP_P

Insert the unscheduled nodes of path FP_s in to the schedule matrices.

Clear the set FP_p

}

When the initial schedule is complete, MARS resolves all conflicts in the same manner as in section 2.3, except all paths will have infinite flexibility. The infinite flexibility allowed for each path removes any limitations on the number of pipelined stages allowed. Although the paths have infinite flexibility, MARS only exploits as little flexibility as needed to construct a valid schedule and this minimizes the number of pipeline stages added implicitly to the DFG. This approach minimizes the number of interconnection registers. The final conflict free schedule generated by MARS will be either a non-overlapped or a fully-static overlapped schedule depending on the inter-iteration precedence constraints.

Example 3.5: Continuing with the 4-stage pipelined lattice filter of example 3.3, MARS creates an initial schedule from the feed-forward paths as shown in Fig. 10(a). Note that from this figure and Fig. 4(a), there exists a conflict in the multiply schedule matrix at time partition 0. There are 5 non-nonrecursive nodes and 4 recursive nodes scheduled at that time partition, but the number of multiplication processors allocated is 8. Node 10 is the node chosen to be shifted to resolve the conflict. After the conflict is resolved, we see that no more conflicts exist and the

final conflict-free schedule for non-recursive nodes is shown in Fig. 10(b). The final hardware operator count for this example equals the lower bound of 8 multiplication operators and 6 addition operators. •

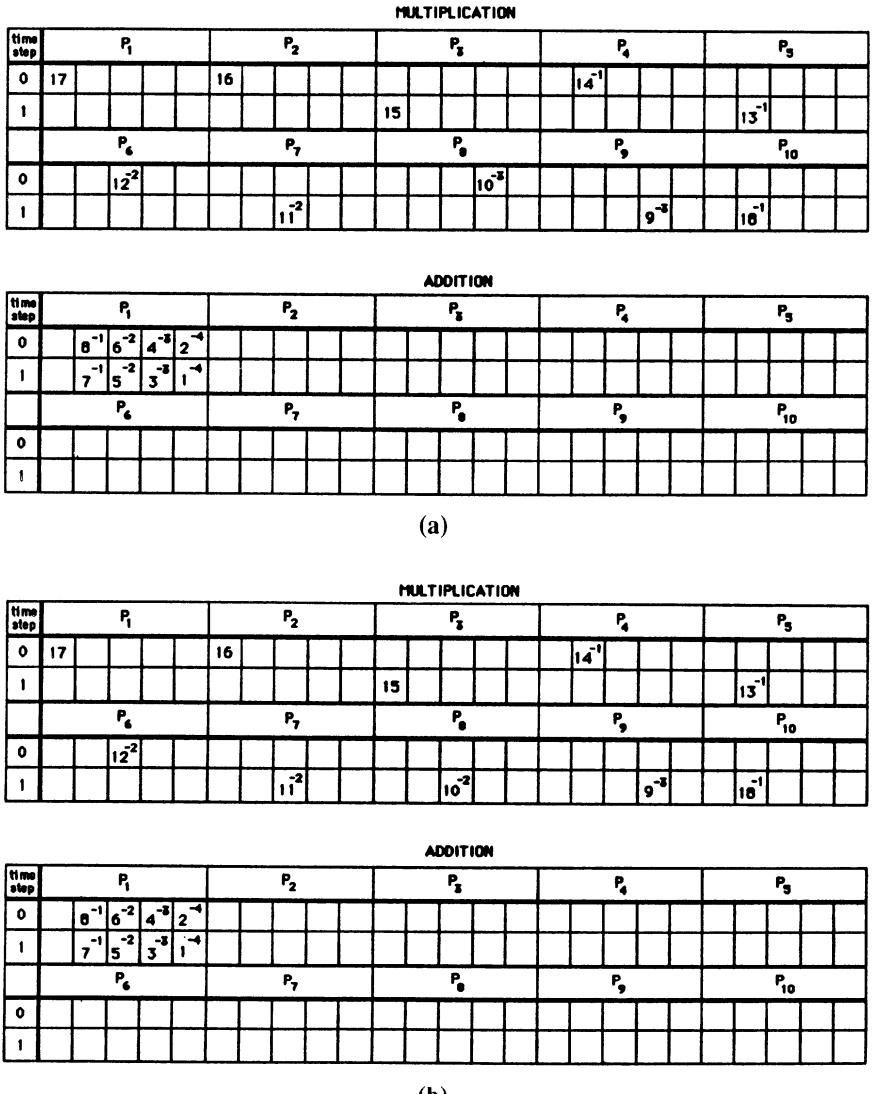


Figure 10: (a) The initial schedule for the non-recursive nodes of the lattice filter of Fig. 3(a). (b) The final conflict free schedule for the non-recursive nodes of the lattice filter of Fig. 3(a).

Example 3.6: Continuing with the FIR filter of example 3.4, MARS creates an initial schedule from the feed-forward paths as shown in Fig. 11. Note that because all nodes of this example are non-recursive nodes, no conflict will occur and the initial schedule is the final schedule. The final hardware operator count for this example equals the exact number required of 3 multiplication operators and 5 addition operators. •

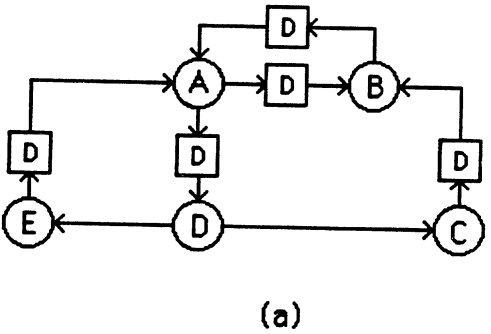
time step	A_1	A_2	A_3	A_4	A_5	M_1	M_2	M_3
0	1	2	3	17^{-1}	20^{-2}	15^{-1}	16^{-1}	ϕ
1	4	5	6	18^{-1}	21^{-2}	9	10	11
2	7	8	23^{-3}	19^{-1}	22^{-2}	12	13	14

Figure 11: This is the initial and final conflict free schedule of the FIR filter of Fig. 8. Note that all nodes of this filter are non-recursive nodes.

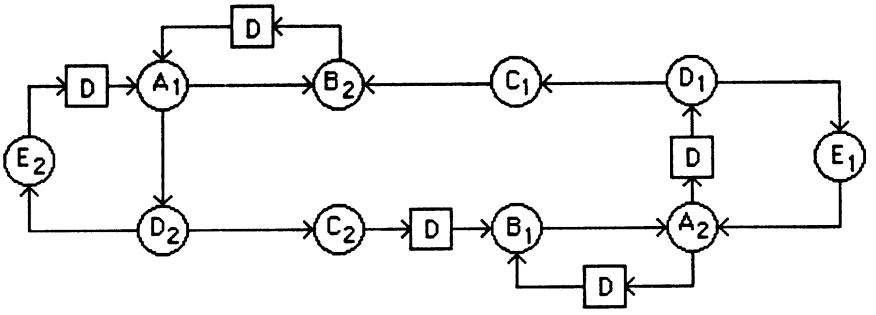
4. ALGORITHMIC TRANSFORMATIONS

There are cases where the iteration bound can be a fractional value (e.g. 1.5) or be less than a node's computation time. To synthesize an architecture which is able to achieve the maximum sample rate (or equivalently, to have an iteration period equal to the iteration bound), we need to perform algorithmic transformations. MARS utilizes an unfolding algorithm to help generate the proper architecture [12,19]. The unfolding process creates a new DFG which has an increased number of iterations but maintains the original functionality. Due to the increase number of iterations, the iteration period will increase; however the iteration bound will remain the same. MARS generates fully-static overlapped schedules for DFG's which contain multiple iterations.

Example 4.1: Consider the general DFG shown in Fig. 12(a). The computation times of nodes A, B, C, D, and E are 20, 5 10, 10 and 2 units, respectively. Assume all nodes will be scheduled to some general purpose processor. The iteration bound for this DFG is 16 units, which is less than the computation time of node A. Therefore, MARS applies the unfolding algorithm to generate a 2-unfolded version of the DFG (see Fig. 12(b)). The unfolded DFG contains 2 iterations and the iteration period for the DFG is now 32 (which is greater than the largest computation time). Note how the delays are distributed randomly in the recursive loops. The initial schedule generated by MARS is shown in Fig. 12(c) and the final conflict-free schedule is shown in Fig. 12(d). We can see that MARS is capable of scheduling this DFG to the iteration bound using 3 processors. •

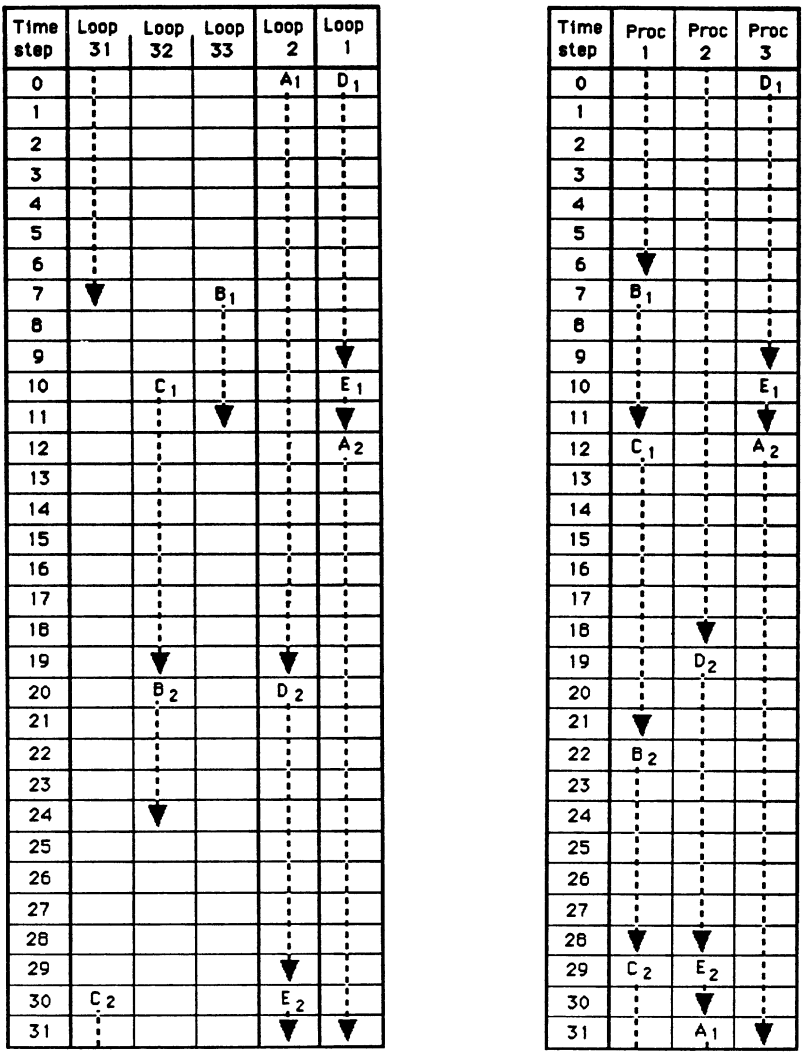


(a)



(b)

Figure 12: (a) A general DFG with three loops. The computation times of nodes A, B, C, D, and E are respectively: 20, 5, 10, 10, and 2 units. The iteration bound = T_{CRIT} = 16 units. (b) The 2-unfolded equivalent DFG of Fig. 12(a). Note that the DFG consists of two iterations of the original DFG and that even though *TCRIT* = 32 units, the iteration bound = 16 units.



(c) **(d)**

Figure 12: (c) The initial schedule of the DFG of Fig. 12(b). (d) The final conflict free schedule of the DFG of Fig. 12(b).

5. DATA PATH SYNTHESIS

Given a final schedule and the number of processors, MARS can construct the data paths which connect the processors, generate the control circuitry required to sample the signals, and determine the number of latches for intra-processors and inter-processor communication. The final results are hardwired control based architectures. For a more detailed description of automatic data path generation, see [20]. From the final schedule matrices, we can generate a retimed and pipelined DFG which has the same functionality as the original DFG.

Example 5.1: As we saw in the 4-stage pipelined lattice filter of example 3.5, the final conflict free schedule for both recursive and non-recursive nodes with an iteration period of 2 units will require 8 hardware multiplication operators and 6 hardware addition operators as shown in Fig. 13. From the final conflict free schedule, we can also derive the implicitly retimed and pipelined DFG as shown in Fig. 14. In this example we see that the new DFG required 3 pipelining stages. Fig. 15 shows the final hardware implementation of the filter with all of the control circuits, hardware operators, and latches in place. •

Example 5.2: As we saw in the 5th-order wave digital elliptic filter of example 2.6, the final conflict free schedule for the recursive nodes with an iteration period of 16 units will require 1 hardware multiplication operator and 3 hardware addition operators as shown in Fig. 7. From the final conflict free schedule, we can also derive the implicitly retimed and pipelined DFG as shown in Fig. 16. Fig. 17 shows the final hardware implementation of the filter with all of the control circuits, hardware operators, and latches in place. •

Example 5.3: As we saw in the 16-point FIR filter of example 3.6, the final conflict free schedule for the non-recursive nodes will require 3 hardware multiplication operators and 5 hardware addition operators as shown in Fig. 11. From the final conflict free schedule, we can also derive the implicitly retimed and pipelined DFG as shown in Fig. 18. Here we see that the new DFG required 3 pipelining stages. Fig. 19 shows the final hardware implementation of the filter with all of the control circuits, hardware operators, and latches in place. •

time step	M1	M2	M3	M4	M5	M6	M7	M8
0	23	26	20^{-1}	24^{-1}	17	16	14^{-1}	12^{-2}
1	22^{-1}	18^{-1}	15	13^{-1}	11^{-2}	10^{-2}	9^{-3}	ϕ

time step	A1	A2	A3	A4	A5	A6
0	25^{-1}	21^{-2}	8^{-1}	6^{-2}	4^{-3}	2^{-4}
1	19^{-2}	ϕ	7^{-1}	5^{-2}	3^{-3}	1^{-4}

Figure 13: The final conflict free combined schedule of recursive and non-recursive nodes of the lattice filter of Fig. 3(a).

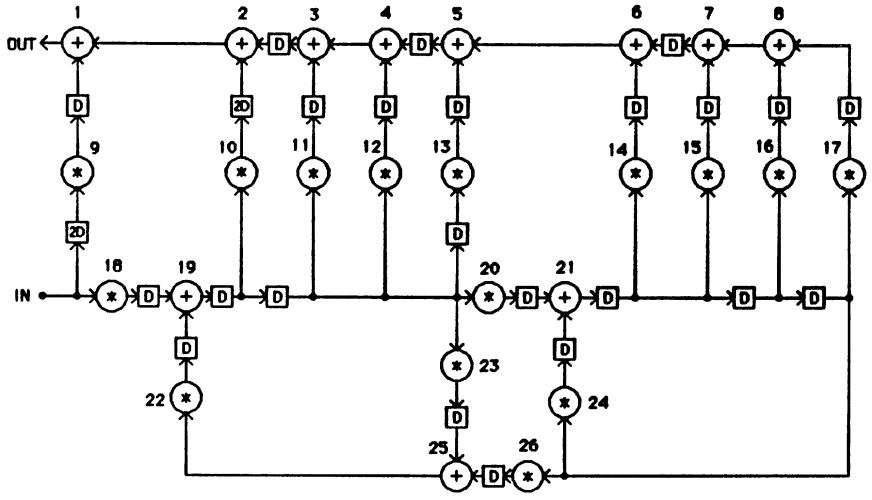


Figure 14: The retimed and pipelined DFG of the lattice filter of Fig. 3(a) as derived from the combined schedule generated by MARS and shown in Fig. 13. Note that the number of pipelined stages = 3.

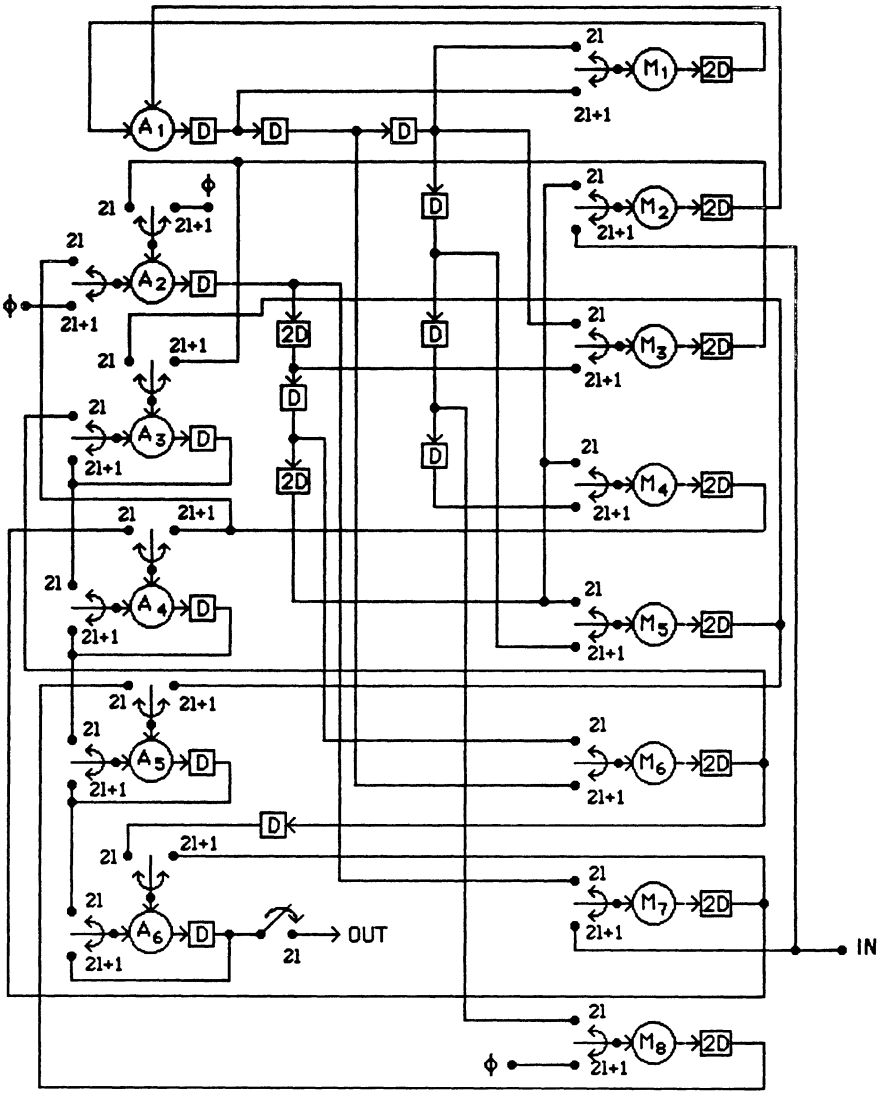


Figure 15: The final hardware architecture of the lattice filter of Fig. 3(a) as generated by MARS.

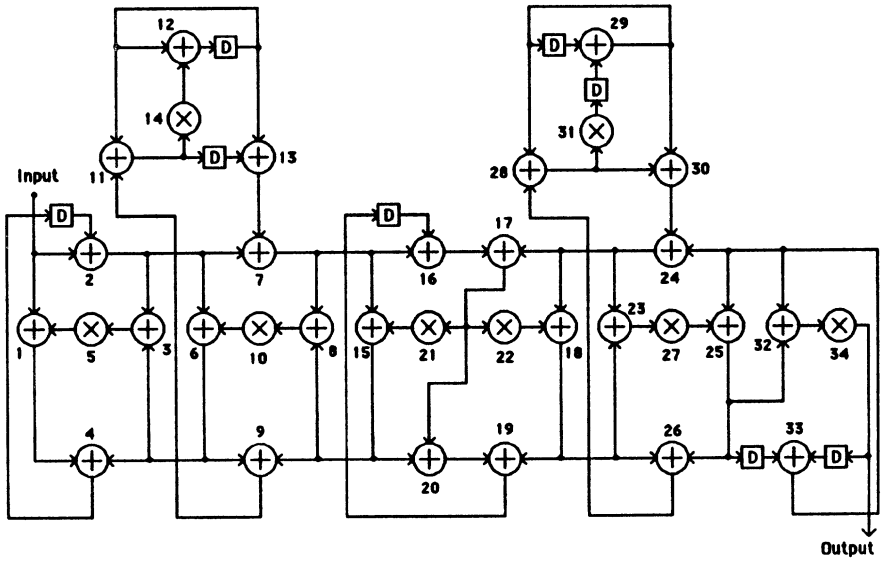


Figure 16: The retimed DFG of the elliptic filter of Fig. 3(b) as derived from the final schedule generated by MARS and shown in Fig. 4(a). Note that there are no pipelining stages because the filter is completely recursive.

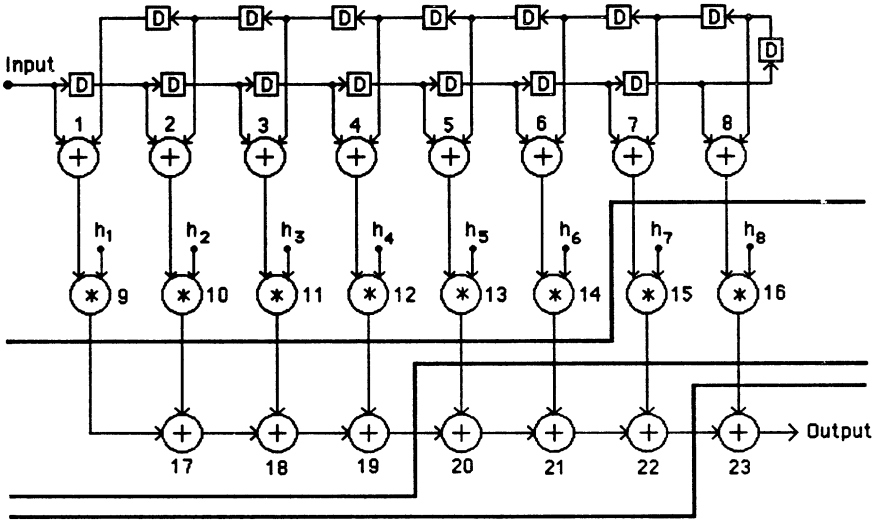


Figure 18: The retimed and pipelined DFG of the FIR filter of Fig. 8 as derived from the combined schedule generated by MARS and shown in Fig. 11. Note that the number of pipelined stages = 3.

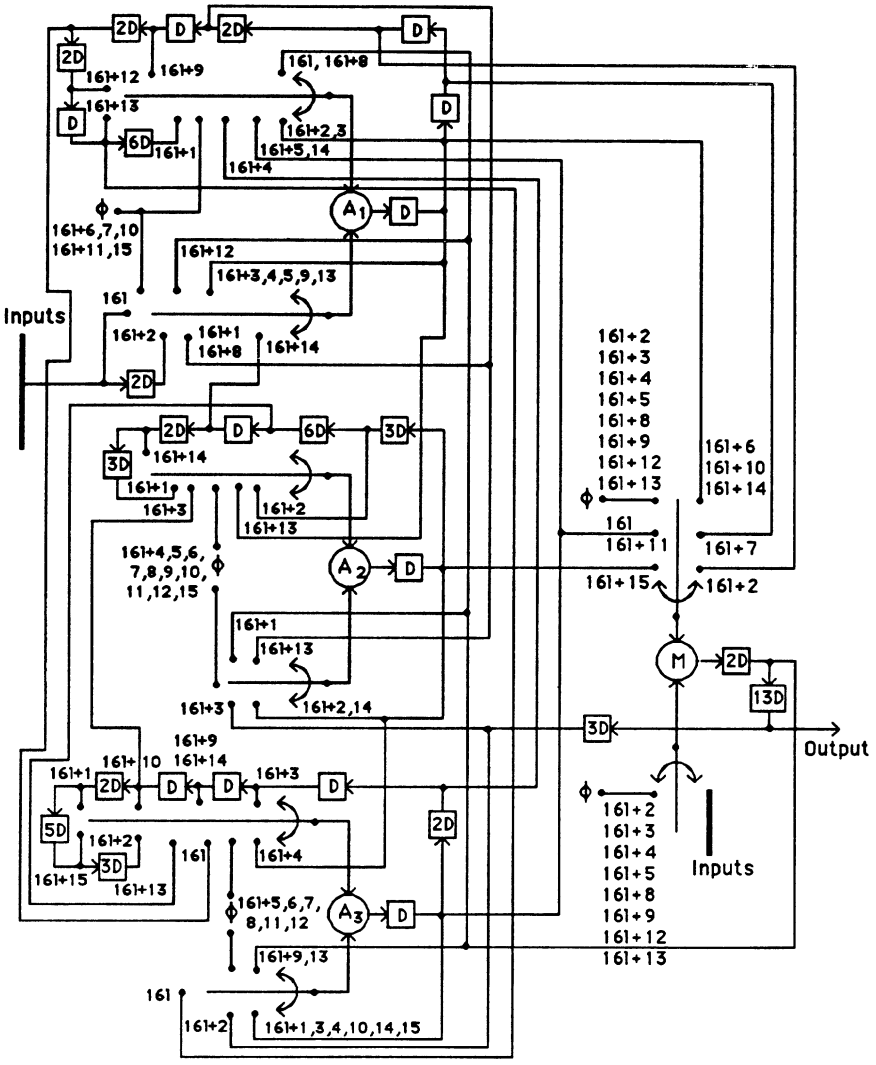


Figure 17: The final hardware architecture of the elliptic filter of Fig. 3(b) as generated by MARS.

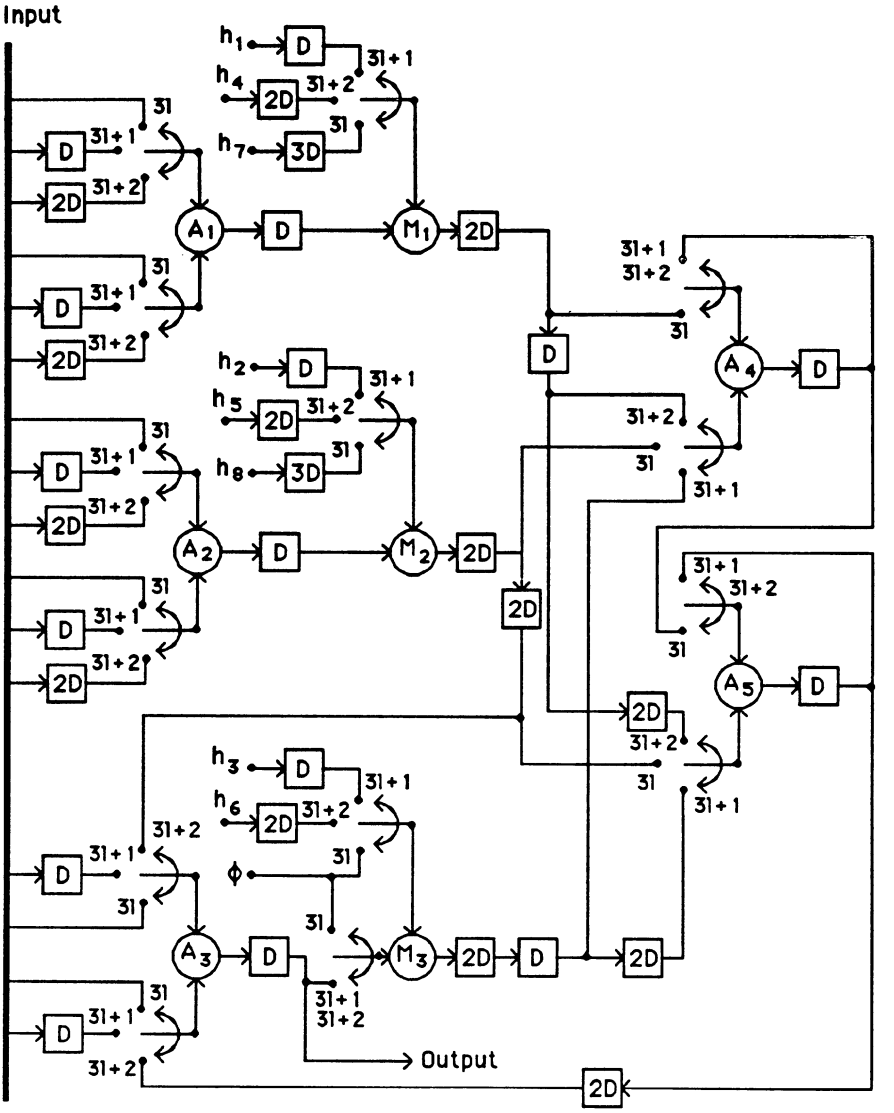


Figure 19: The final hardware architecture of the FIR filter of Fig. 8 as generated by MARS.

6. RESULTS

We now present some results of our scheduling and resource allocation algorithm for a set of benchmarks. These results demonstrate the performance of MARS. They will show how MARS performs on completely recursive systems, completely non-recursive systems, and on general data-flow graphs. In all of our tests, we assumed the availability of one stage pipelined adders and two stage pipelined multipliers. Table 1 summarizes a few of our benchmark results. For each filter type we show the iteration bound T_∞, the critical path time T_{cr} and the iteration period used to schedule the filter T. The table also shows the types of operations and the total number of each type in the original DFG, the lower bound on the number of processors required for the given iteration period, and the actual number of processors required for the given iteration period. We also show the total number of loops in the DFG, the number of loops MARS needed to cover all recursive nodes, and the number of loops and loop sections MARS used to complete the scheduling, and the cpu times in seconds need to generate a valid schedule when MARS is executed on a SUN Sparc2 workstation.

As we look at the results in Table 1, one can see that MARS is capable of generating valid schedules for iteration periods equal to the iteration bound. Most other synthesis systems are limited to an iteration period equal to the critical path time. The FIR filter benchmark is completely non-recursive and the results show that the final number of processors required is always equal to the lower bound. The all-pole lattice filter shows that even though a filter may have many loops (58), only a small subset of the loops is required to schedule all of the nodes (4). We also see from this example and the elliptic filter example (both of which are completely recursive) that the recursive section of a filter is the most restrictive portion of any DFG. The 4-stage and 8-stage pipelined lattice filters show that generating a valid schedule for an iteration period equal to the iteration bound may not always be achieved due to the technology constraints. By using the unfolding transformation we see that a schedule with an iteration period equal to the iteration bound can be achieved. The 2-unfolded example shows an iteration period equal to 3 units. The iteration bound is achieved because this filter contains two iterations within the iteration period; therefore, the iteration period for one iteration = $\frac{3}{2}$ or 1.5. A similar example is shown for the 4-unfolded example. In this example, the number of iterations is four and the iteration period = 3. Therefore the iteration period for a single iteration = $\frac{3}{4}$ or 0.75.

When compared to previous works, MARS produces better or as good results. In the worst case, the scheduling complexity is exponential; however Table 1 shows that MARS can schedule the filters in less than exponential time. The largest example has 168 operations and only took 0.866 cpu seconds to generate a valid schedule with an iteration period equal to the iteration bound.

Table 1: Summary of the results from the MARS design system's scheduling and resource allocation algorithms for uniform architectures. T_{INF} is the iteration bound, T_{cr} is the critical path time, and T is the iteration period for the unfolding factor number of iterations.

Filter Type	T_∞	T_{cr}	T	algorithmic		Lower bound		hardware		Total # loops	# covering loops	# loops/ loop sections	cpu time
				mult	add	mult	add	mult	add				
16-point FIR	2	10	2	8	15	4	8	4	8	0	0	0	0.082
	2	10	3	8	15	3	5	3	5	0	0	0	0.099
2nd-order biquad	4	5	4	4	4	1	1	1	1	2	2	2	0.132
2-cascaded biquad	4	7	4	8	8	2	2	2	2	4	4	4	0.149
All-pole lattice	8	16	8	4	11	1	2	2	3	58	4	7	0.083
	8	16	9	4	11	1	2	1	2	58	4	7	0.100
5th order wave digital elliptic	16	17	16	8	26	1	2	1	3	45	5	5	0.167
	16	17	17	8	26	1	2	1	2	45	5	5	0.167
	16	17	28	8	26	1	1	1	1	45	5	5	0.233
4-stage pipelined lattice	1.5	10	2	15	11	8	6	8	6	3	3	4	0.333
	1.5	10	5	15	11	3	3	3	3	3	3	4	0.399
8-stage pipelined lattice	0.75	18	2	23	19	12	10	12	10	3	3	4	0.232
	0.75	18	10	23	19	3	2	3	2	3	3	4	0.366
2-unfolded 4-stage pipelined lattice	1.5	10	3	30	22	10	8	10	8	6	6	8	0.299
	1.5	10	5	30	22	6	5	6	5	6	6	8	0.299
4-unfolded 8-stage pipelined lattice	0.75	18	3	92	76	31	26	31	26	12	12	16	0.866
	0.75	18	4	92	76	23	19	23	19	12	12	16	1.482
LMS adaptive	9	9	9	11	10	2	2	4	4	10	5	5	0.167
	9	9	12	11	10	2	2	2	2	10	5	5	0.217

7. THE MARS SYSTEM

We have implemented our algorithms into the MARS system. In this section we present the input and output files of a second-order biquad filter as shown in Fig. 20(a). This filter contains both a recursive section and a non-recursive section and the iteration bound equals 4 units and T_{CRIT} =5 units. Fig. 20(b) shows the input description file of the filter. Each line which begins with the character "N" represents a node and contains the node characteristics using the following syntax: N (node computation time) (pipelining level) (processor type number). If the last

field is 0 MARS uses the node computation time to determine the processor type to which the node is assigned. As each node line is read, MARS assigns a node number which represents the node within MARS. The numbering is sequential. Any other information beyond the last field is considered to be comments. Each line that begins with the character "E" describes an edge with the following syntax: (starting node number) (ending node number) (number of delays). An optional line begins with the character "T" which represents the user defined iteration period. If this line is not present, it is initially assumed to be zero. Fig. 20(c) contains the output file. This file consists of four sections:

1) A table of the nodes and their relevant information.
2) The final processor counts for each processor type.
3) The final conflict free schedule for recursive nodes.
4) The final conflict free schedule for non-recursive nodes.

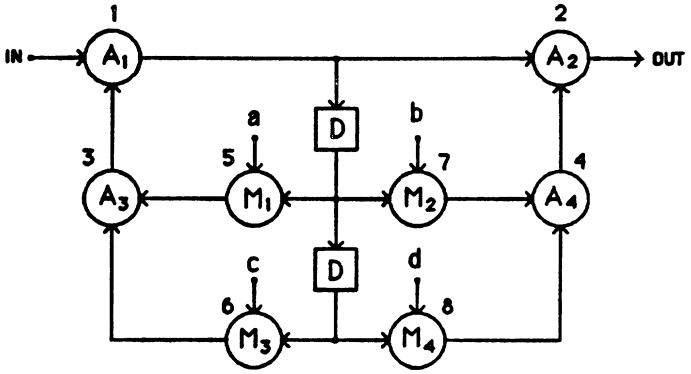


Figure 20: **(a)** The DFG of a second order biquad filter. T_A = 1 unit, T_M = 2 units, the iteration bound = 4 units, and T_{CRIT} = 5 units.

```
N 1 1 0          !A1  1  1
N 1 1 0          !A2  2  1
N 1 1 0          !A3  3  1
N 1 1 0          !A4  4  1
N 2 2 0          !M1  5  2
N 2 2 0          !M2  6  2
N 2 2 0          !M3  7  2
N 2 2 0          !M4  8  2
E 0 1 0
E 1 2 0
E 3 1 0
E 1 5 1
E 1 6 2
E 1 7 1
E 1 8 2
E 4 2 0
E 5 3 0
E 6 3 0
E 7 4 0
E 8 4 0
E 2 0 0
T 4
```

Figure 20: **(b)** The input file for the biquad filter of Fig. 20(a) which describes the node parameters and the topology of the filter.

```
Node #   delay   processor   time partition   real time
    1       1         2              0             4
    2       1         2              2             6
    3       1         2              3             3
    4       1         2              1             5
    5       2         1              2             2
    6       2         1              1             1
    7       2         1              0             4
    8       2         1              3             3

The number of required processors for this DFG
scheduled with an iteration period of 4:
Processor # 1 needs 1 instances
Processor # 2 needs 1 instances

This is the conflict free schedule for Recursive nodes

      0:  5       0
      1:  0       0
      2:  0       0
      3:  0       6 ( 1)

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

      0:  0       0
      1:  0       0
      2:  3       0
      3:  1       0

This is the conflict free schedule for Non-Recursive nodes

      0:  0       0       0     *    0       0       0
      1:  0       0       0     *    0       8       0
      2:  0       7       0     *    0       0       0
      3:  0       0       0     *    0       0       0

     &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

      0:  0       0       0     *    4 (-1)  0       0
      1:  0       0       0     *    2 (-1)  0       0
      2:  0       0       0     *    0       0       0
      3:  0       0       0     *    0       0       0
```

Figure 20: (c) The output of the scheduling program of MARS which describes the final conflict free schedules for recursive nodes and non-recursive nodes.

8. CONCLUSIONS

Using graph based approaches and incremental refinement steps, we have developed a new concurrent scheduling and resource allocation scheme and have incorporated it into the MARS design system. Our approach exploits concurrency among the iterations through the use of inter-iteration precedence constraints Previous work only addressed the synthesis of simple algorithms which contained single or lumped delays. MARS is not limited to such algorithms. It is capable of performing scheduling and allocation for more general data-flow graphs. Synthesis and issues dealing with such algorithms were not explored extensively before. We have eliminated the requirement for preprocessing algorithms for retiming and software pipelining because our approach implicitly retimes and pipelines flow graphs as it schedules. Future research is directed towards extending MARS for time-constrained scheduling of data-flow graphs with non-uniform implementation styles. The extension of the loop scheduling algorithm for resource-constrained scheduling is also a topic of further study.

9. REFERENCES

[1] P. DeWilde, *et al*, "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms", in *VLSI and Modern Signal Processing*, (edited by Kung, Whitehouse, and Kailath), Chapter 15, Prentice Hall, 1985.

[2] J. Rabaey, *et al*, "Resource Driven Synthesis in Hyper System", *Proc. of 1990 IEEE ISCAS*, pp. 2592-2595.

[3] P.G. Paulin and J.P. Knight, "Force Directed Scheduling for the Behavioral Synthesis of of ASIC's", *IEEE Trans. on Computer Aided Design of IC's*, June 1989.

[4] C.T. Hwang, J.H. Lee, Y.C. Hsu, and Y.L. Lin, "A Formal Approach to the scheduling Problem in High Level Synthesis", *IEEE Transactions on CAD*, vol. 10, April 1991, pp. 464-475.

[5] K.S. Hwang, *et al*, "Scheduling and Hardware Sharing in Pipelined Data Paths", *Proc. of the IEEE Int. Conf. on Computer Aided Design*, 1989, pp. 24-27.

[6] M.C. McFarland, A. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, Feb. 1990, pp. 310-318.

[7] N. Park and A.C. Parker, "Sehwa: A Software Package for the Synthesis of Pipelines from Behavioral Specifications", *IEEE Trans. on Computer Aided Design*, March 1988, pp. 356-370.

[8] H. DeMan, *et al*, "Cathedral II: A Silicon Compiler for Digital Signal Processing", *IEEE Design and Test*, December 1986, pp. 13-25.

[9] C.-Y. Wang and K.K.Parhi, "Dedicated DSP Architecture Synthesis Using the MARS Design System", *Proc. of the IEEE ICASSP*, May 1991, Toronto, pp. 1253-1256.

[10] C.-Y. Wang and K.K. Parhi, "High Level DSP Synthesis Using the MARS Design System", *Proc. of the IEEE ISCAS*, May 1992, San Diego.

[11] D.J. Wang and Y.H. Hu, "Optimal Scheduling of Linear Recurrence Equations on a Multiprocessor Array", *Proc. of the IEEE ICASSP*, May 1991, Toronto, pp. 1581-1584.

[12] K.K. Parhi and D.G. Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding", *IEEE Trans. on Computers*, Feb. 1991, pp. 178-195.

[13] D.A. Schwartz and T.P. Barnwell, III, "Cyclo-static Solutions: Optimal Multiprocessor Realizations of Recursive Algorithms", *VLSI Signal Processing II*, IEEE Press, 1986.

[14] K.K. Parhi, "Algorithm Transformation Techniques for Concurrent Processors", *Proceedings of the IEEE, Special Issue on Supercomputer Technology*, Dec. 1989, pp. 1879-1895.

[15] C.E. Leiserson and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming", *3rd Caltech Conference on VLSI*, pp. 87-116.

[16] M. Renfors and Y. Neuvo, "The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints", *IEEE Trans. on Circuits and Systems*, March 1981, pp. 196-202.

[17] E.M. Reingold, *et al*, "Combinatorial Algorithms - Theory and Practice", Prentice-Hall, Inc., 1977, pp. 348-353.

[18] J.-G. Chung and K.K. Parhi, "Design of Pipelined Lattice IIR Digital Filters", *Proc. of the 25th Asilomar Conf. on Signals, Systems, and Computers*, Nov. 1991, pp. 1021-1025.

[19] K.K. Parhi and C.-Y. Wang, "Digit-Serial DSP Architectures", *Proc. of the 1990 IEEE Application Specific Array Processors*, Princeton, Sept. 1990, pp. 341-351, (see also K.K. Parhi, *IEEE Trans. on Circuits and Systems*, April 1991).

[20] K.K. Parhi, C.-Y. Wang, and A.Brown "Synthesis of Control Circuits in Folded Pipelined DSP Architectures", *IEEE Journal of Solid-State Circuits*, vol 27, January, 1992, pp. 29-43.

[21] Benchmarks for the Fifth International Workshop on High-Level Synthesis, 1991.

6

HIGH PERFORMANCE ARCHITECTURE SYNTHESIS SYSTEM

Phillip Duncan, Shobana Swamy, Steve Sprouse, Daniel Potasz, Rajeev Jain
UCLA, Los Angeles, California

William Cammack, Neal Gafter, Yiwan Wong, Wanda Gass
Texas Instruments, Dallas, Texas

The High Performance Architecture Synthesis System (Hi-PASS) is a CAD system for DSP architecture synthesis. Hi-PASS generates maximally parallel architectures to support DSP applications where the sample rate is too high for time sharing of hardware to be feasible. Such applications cannot be implemented on a single DSP microprocessor. Real-time execution demands either a multiprocessor implementation or dedicated hardware where multiple operations are happening concurrently. It is the latter solution which Hi-PASS generates using a combination of compiler and optimization techniques.

This high-level synthesis tool converts a C code description of the DSP algorithm into an RTL description of the design. Hi-PASS provides a link between the algorithm development environment and the VLSI design environment which will reduce the overall design time of complex VLSIs. High-level synthesis will enable quick evaluation of feasibility and investigate architectural trade-offs early in the design cycle.

INTRODUCTION

Hi-PASS consists of four modules which are integrated using the OCT database developed at UC Berkeley. The symbolic interpreter converts the C code description of the algorithm into a dataflow graph. The Heuristic Search Optimization System (HOPS) optimizes the arithmetic and logic operations in the flowgraph. Bit-level retiming is used to minimize the critical path and increase the clock rate. Finally, an RTL description is generated for Lager (a silicon assembly program) or in VHDL.

The input to the Hi-PASS system is a C subroutine which describes the processing of data on a sample by sample basis. The C code is parsed by a C compiler front-end and then symbolically interpreted. The symbolic interpreter removes all control flow and symbolically executes all operations which can be computed at compile time. The output of the interpreter is straight-line, single assignment C code which represents the full parallelism and the data dependencies of the algorithm. Each variable in the new version of the algorithm is assigned a bit length to minimize error propagation and hardware utilization. To further reduce hardware utilization, all multiplications and divisions which have a constant as one of the operands are converted to an equivalent form of hardwired shift and add operations.

A data flow graph representation of the algorithm is then constructed using the OCT database as a data structure. Two different programs are used to minimize the flow graph. The Heuristic Search Optimization System (HOPS) minimizes the number of operators in the graph and modifies the type of operators to reduce the hardware requirements. The Retiming program adds pipeline registers and repositions registers to reduce the critical path which improves the overall clock rate of the circuit. Both HOPS and Retiming read the data flow graph from the OCT database and then write the optimized version of the graph back to OCT.

Because the sample rate of the system requires that there is no resource sharing, no scheduling is required. Each operator in the data flow graph is mapped into a unique instance of hardware. Therefore, the optimized data flow graph is equivalent to the structural representation of the algorithm. There are two versions of data path synthesis supported by Hi-PASS. One program in Hi-PASS maps the data flow graph into the standard cell library of the Lager system. Another program in Hi-PASS converts the data flow graph into a behavioral description of the RTL architecture using VHDL.

C FRONT END

The goal of Hi-PASS is to link the software development environment with the hardware synthesis environment so that software and hardware development becomes an interactive process. The C programming language is a well established standard and is often used to develop a computer simulation of the algorithm in the first step of the design process. By using the C language as the input to Hi-PASS, the computer simulation of the system can directly drive the synthesis process.

While synthesis based on the C language has several benefits, it also has some limitations. One of these limitations is the sequential constraints of program execution which are not inherent in the algorithm. The first step in the synthesis process removes the sequential nature of the C program and converts the algorithm to its inherently parallel form. To fully exploit the parallelism of the algorithm, all of the data independent control flow is removed and the data dependencies are identified. In this process, a data dependency graph is constructed to describe the algorithm.

Standard compiler optimization techniques can be applied to reduce the amount of computations, such as dead code removal and constant propagation. Given that a maximally parallel datapath is the target architecture, other transformations can be applied to reduce the hardware. For example, division and multiplication by a constant can be converted into a series of shifts and adds.

Another restriction of C is that data types are limited to a fixed number of bits as implemented on a microprocessor (i.e. 32 bits). An important step in architecture synthesis is the selection of bit widths for all variables and their effect on the accuracy of the algorithm. Therefore, in generating the dependency graph, all data elements are assigned a bit width value based on the resolution of the input variables.

SYMBOLIC INTERPRETATION

The first step in the synthesis system is the symbolic interpretation of a C program which describes the DSP application, for example an IIR filter. The purpose of symbolic interpretation is to remove control flow and other constructs that would not map directly onto hardware. The output of the interpreter is straight-line code that assigns a value to each variable only once.

This straight-line code corresponds directly to a data flow graph of the application.

The C program is expected to contain a separate function which describes the operations that must be executed on each data sample. The C function is executed symbolically, with the initial value of each input parameter being set to its name. For example, the input variable X is set to the expression "X" and input variable Y would be set to "Y". When the interpreter encounters an assignment statement, such as **X = X + Y**, the variable X would then contain the expression "X + Y", indicating that X contains the sum of the original values of X and Y. Executing this assignment a second time (for example from within a loop) would result in X being given the value "X+Y+Y".

Complications arise when the program uses arrays, pointers, or data-dependent control flow (such as **if** and **for** constructs). An algorithm has been developed for interpreting programs with these constructs in contexts that appear most frequently in DSP applications.

When the interpreter reaches the **return** statement of the function, every variable has a symbolic representation of its final value in terms of the initial values. The interpreter then outputs a new C program that contains assignments to the program variables - each variable is assigned the final value that was computed during interpretation.

Each subexpression output is computed into a temporary variable so that no output assignment statement has more than one operator on its right-hand side. Because the interpreter has computed the final values of the variables, and because no temporary variable is used more than once, each variable is assigned to at most once in the resulting program.

The output of the interpreter is a C program. This allows the original code to be tested against the interpreted code during development of the interpreter. A post processing program is then used to translate this restricted form of C into the OCT flowgraph format.

BIT WIDTH ASSIGNMENT

As mentioned earlier, use of the C language as a front end to synthesis restricts data types to be a fixed number of bits. For example, many C compilers treat integers has having 32 bits, shorts as having 16 bits, and

character variables as having 8 bits. These bit width assignments are convenient for program execution on microprocessors, but can cause creation of excessive hardware during synthesis. Thus, Hi-PASS provides a method for analyzing and minimizing C variable bit widths based on user requirements.

Bit width assignment occurs in two phases. In the first phase, the C code produced by the symbolic interpreter is analyzed and default bit widths (based on the C compiler implementation) are assigned to all global variables, static variables, and variables used as parameters to the interpreted C function. These variables and their associated bit widths are printed in a report file in order to allow the user to customize the widths according to his specific requirements.

The report file is then used as a starting point for final bit width analysis in the second phase. During this phase, the user specified widths for the global variables, static variables, and C function parameters are propagated throughout the C function. Other variables are minimized according to the context in which they are used. For example, the C statement

int test1 = y > 100;

would assign test1 a width of 32 bits. This is unnecessary, however, since the result of the comparison operation will always be either true (1) or false (0). Thus, test1 is assigned a width of 1 during the second phase.

After all bit width assignments have been made, the resulting code is transformed into an Intermediate Programming Language representation (IPL). The IPL language is similar to C, but provides syntactic structures for specifying bit widths in variable declarations. For example, declaration of two, 16 bit variables would be written as follows:

INT<16> var1, var2;

The IPL code is then fed to a translation program which converts the code into a data flow graph representation, which is in turn stored in an OCT database to be used by other programs in the synthesis flow.

DATAPATH OPTIMIZATION

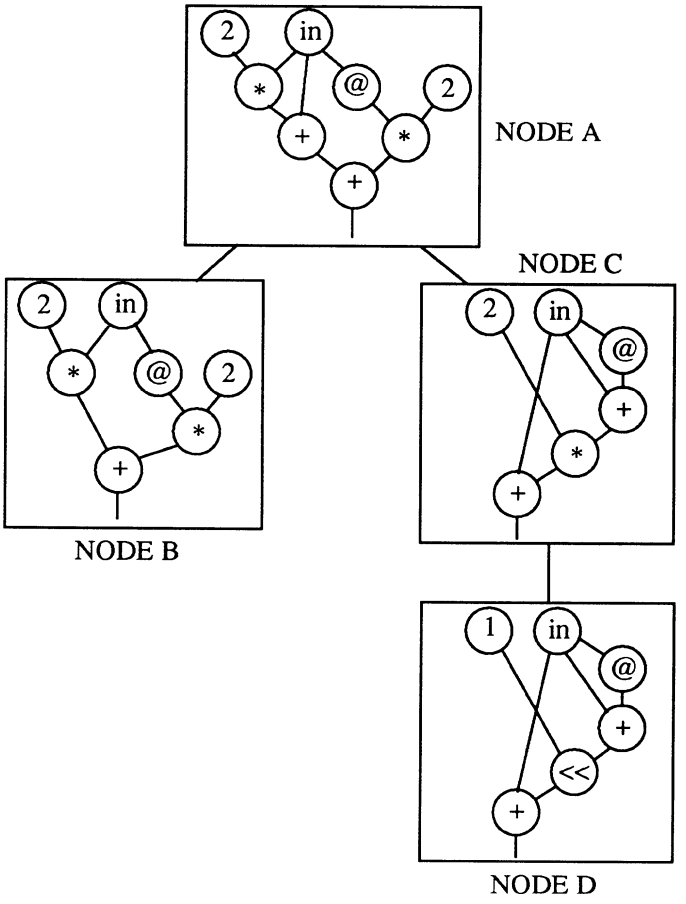
Since the high throughput DSP applications being targeted by the Hi-

PASS system require real-time processing at very high data rates and consist of arithmetic intensive operations, a key problem associated with the architecture design issue is the optimization of the datapath. Previous work focuses on optimization of time-shared datapaths where each physical functional unit processes multiple algorithmic operations. A primary measure of optimization quality for time-shared datapaths is the degree of resource utilization. In contrast, the Hi-PASS system focuses strictly on optimizing the area and critical path for the fully parallel architectures to achieve the highest possible throughput with no resource sharing overhead.

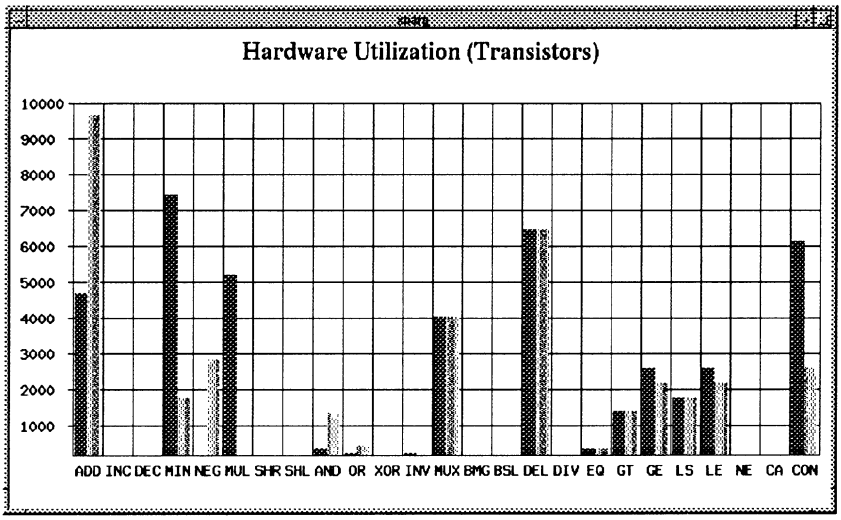
The Heuristic Search Optimization System (HOPS) is a datapath optimization system developed to handle flow graphs consisting of arbitrary combinations of arithmetic and logic operations. It produces functionally equivalent flow graphs that better meet the high performance requirements of the targeted applications. One of the keys to obtaining high performance datapaths is making maximum use of all opportunities to optimize the arithmetic portions of the system. Much work has been published on methods to optimize single or multi-level gate logic, but in general these methods fail to produce adequate results when the system to be optimized includes a high percentage of arithmetic elements.

The basic concept of HOPS is to generate a search tree of functionally equivalent graphs using a set of arithmetic and logic transformations. A heuristic search strategy is then employed to direct the traversal of the search tree using a cost function based on estimates of area and speed. Several search strategies have been investigated, including branch and bound, A*, and best first. HOPS is the first system to propose search techniques developed within the artificial intelligence community to optimize fully parallel DSP architectures. New techniques have also been developed for pruning redundant sections of the search tree. The first figure below represents a small search tree example. The solution found by the search is Node D, which reduces the hardware to two adds and a fixed shift versus two adds and two multiplies for Node A.

The number of occurrences of each type of operator is computed both before and after the datapath optimization. The second figure below shows a hardware utilization graph in which the module types are shown on the X axis and the number of occurrences is shown on the Y axis (dark bars show the results before optimization and the light bars show the results after optimization). The figure shows that for the given application, the number of ADD modules increased, but the number of MUL modules was reduced to zero.



Example Search Tree



Hardware Utilization Graph

RETIMING

The throughput rates for the applications being targeted by the Hi-PASS system demand that the critical path of the architecture be optimized to yield very short inter-register delay times. It is a well known fact in the design of DSP architectures that this goal can be achieved by pipelining (inserting extra registers into the computation paths). This technique can reduce the critical path length and thus reduce the clock period. A more general form of pipelining is known as retiming. Retiming allows existing registers to be moved within the circuit as well as adding extra registers. In order to achieve the high performance goals of Hi-PASS, an automatic retiming tool has been implemented.

In addition to implementing the retiming methods presented by Leiserson, an important extension to the retiming model has been developed for Hi-PASS. It is sometimes necessary to pipeline bit-parallel arithmetic operators at the bit level by inserting registers in operator ripple paths. To accomplish this using Leiserson's retiming module requires each n-bit operator to be expanded into n distinct nodes. Since the number of nodes in the graph grows

by a factor of n, the execution time of the retiming algorithms increases dramatically. The extension to Leiserson's model allows bit-level retiming to be performed without increasing the number of nodes by a factor of n.

SYNTHESIS OUTPUT

The OCT flowgraph representation consists only of functional operators and represents a register transfer level (RTL) description. This description can be output in two different forms. The first form is a structure master view (SMV) description based on the standard cell library of the Lager IV silicon assembler. The second form is a VHDL description of the datapath that can be synthesized into a gate-level netlist using commercially available tools.

Interface To Lager

The interface to the Lager IV silicon assembly tools is implemented in a program called flow2smv. This program generates an OCT structure master view from the OCT flow graph produced by Hi-PASS. The structure master view created by flow2smv is a netlist of physical macrocells that have a one-to-one correspondence with the cells in the OCT flowgraph library. The purpose of the macrocell library is to provide encapsulation for the underlying standard cell library. For instance, the Lager IV standard cell library does not include a subtract cell, so the subtract physical macrocell encapsulates adder and inverter standard cells.

There are several steps in addition to library mapping which must be incorporated into flow2smv. Several global signals are not present in the OCT flow graph description because they are not related to the logical functionality of the design. These signals - power, ground, clock, and reset - must be added to the SMV for physical implementation. A global reset line is added to all registers as a means of forcing the sequential circuits into a known state for testing purposes.

After adding global signals, buffer trees are generated for any signal lines that drive high loads. Currently any signal with a fanout greater than three is buffered. Typically, the clock signal requires the most buffering, but is treated no differently than ordinary data signals. A tree is generated using single strength standard cell buffers at all levels and a branching factor of three throughput.

Finally, input/output latches and pad drivers are optionally added to produce a chip core that can be dropped in a pad ring.

Interface To VHDL

In order to utilize commercially available tools for gate-level synthesis, Hi-PASS provides facilities for converting the optimized and retimed flow-graphs into VHDL representations through the use of a Flow2VHDL (Flow Graph to VHDL) translator. This program allows creation of behavioral VHDL, structural VHDL, C, and C++ models.

The behavioral VHDL models produced by Flow2VHDL are similar in form to the straight-line C programs produced by the symbolic interpreter, in which each line consists of a single assignment and signal operation. Flow2VHDL, however, adds clock and reset signals to the design and connects them to each register. The reset signal is utilized in the VHDL in order to initialize all of the register values during the first clock cycle.

The structural VHDL models are block level descriptions of the design, in which each flow graph operation is constrained to its own hierarchical block. Each hierarchical block contains a behavioral description for the block's operation based on the previously computed input and output bit widths. For example, a VHDL block library produced by Flow2VHDL may contain behavioral code for a 4-bit adder, 6-bit adder, etc. These blocks can be individually synthesized by commercial tools and re-used on multiple designs.

Production of C and C++ code provides a fast method for verification of the synthesis process without having to simulate the code with a VHDL simulator. The C programs can be compiled on any standard C compiler and the results can be compared with those of the original C program. The C++ code allows simulation of the program based on the bit width assignment for each variable in order to verify that no underflow or overflow conditions have been introduced during the synthesis process.

SUMMARY

Hi-PASS is an integrated toolset that provides automatic design flow from a C language description of a DSP function down to an ASIC layout. It synthesizes maximally parallel DSP architectures for applications where the

sample rates are close to the achievable clock rate in the given technology. CAD tools have been developed to aid in both the reduction of hardware and the reduction of critical path delays. Significant contributions have been made in the areas of symbolic interpretation, datapath optimization, and retiming.

References

1. P. Duncan et al. "Hi-PASS: A Computer-Aided Synthesis System For Maximally Parallel Digital Signal Processing ASICs". *Proc. International Conference on Acoustics, Speech and Signal Processing*, 1992.

2. P. Duncan et al., "Experiments With The Hi-PASS Synthesis System", *Proc. IEEE International Symposium on Circuits and Systems*, 1992.

3. P. Moore, "The General Structure of OCT", *Oct Tools Distribution 3.0*, Electronics Research Laboratory, UC Berkeley, 1989.

4. N. Park, "SEHWA: A Program for Synthesis of Pipelines", *23rd Design Automation Conference*, 1986.

5. J. Rabaey, M. Potkonjak, P. Hoang, and C. Chu, "HYPER - Design Synthesis for High Performance Real Time Systems", *Proc. IEEE International Symposium on Circuits and Systems*, May 1990.

6. C. Shung et al., "An Integrated CAD System for Algorithm Specific IC Design", *IEEE Transactions on CAD*, May 1991.

7. C.E. Leiserson et al., "Optimizing Synchronous Circuitry", *3rd Caltech Conference on VLSI*, pp. 87-116, 1983.

8. R. Jain, et al., "FIRGEN: A CAD system for Automatic Layout Generation of High-Performance FIR Filters", *IEEE 1990 Custom Integrated Circuits Conference*, 1990.

7

Modeling Data Flow and Control Flow for DSP System Synthesis *

Michaël F.X.B. van Swaaij **Frank H.M. Franssen**
Francky V.M. Catthoor **Hugo J. De Man**

IMEC, Kapeldreef 75,B-3001 Leuven, Belgium

Abstract

A data flow and control flow model is presented for use in high level synthesis of efficient time multiplexed architectures targeted towards real-time DSP systems. The model is an extension to the polyhedral models used in array synthesis techniques. The model features a mathematical description of dependencies between individual operations and signal instances of multi-dimensional signals for algorithms that can be described by Conditional Affine Recurrence Equations. It allows for a generalization of high level control flow transformations and their steering by efficacious optimization methods. The inherent amenity of this type of model for these tasks is motivated by examples. Important tasks in high level synthesis that can exploit this model are memory management for time multiplexed architectures and non-linear transformations for array architectures, but also other tasks may benefit. The former task will be described in more detail in this paper.
Performance figures of a CAD tool implementing the model extraction demonstrates the feasibility of this approach for the envisaged application domain.

*Research partially supported by ESPRIT programs BRA 3280 and 3281 of the EC.

1 Introduction

State-of-the-art systems for real-time Digital Signal Processing (DSP) can be found in a variety of technical domains, ranging from consumer goods (e.g. video/audio components, mobile telephone) to means of production and mass communication (e.g. robotics, machine vision, satellite systems). Depending on the application, different systems and different performance criteria are applied. In consumer goods, for example, manufacturing costs and power dissipation are usually important performance issues. In production goods, important performance criteria can be processing speed and also flexibility.
In many of these cases, advanced systems have to be specially designed for a given application to meet the performance criteria under current technology constraints. Therefore, Application Specific architectures need to be synthesized.

A primary task in Application Specific (AS) architecture synthesis is the extraction of the data flow from a given algorithm description. The term *data flow* is defined here as the combination of operations and dependencies between them that define the algorithm. *Control flow* is defined as a (partial) ordering of computations meeting the restrictions of their dependencies. A computation is the execution of an operation on its operands.

Data flow extraction is one of the first tasks in any design trajectory from algorithm to architecture realization. A data flow combined with performance requirements and system constraints forms a complete specification of a systems component. Formalization of optimization tasks in the synthesis process can only be done when data flow and control flow are modeled properly. Formalization is necessary to make globally optimal design decisions, and also to be more independent from the way an algorithm has been specified.
The synthesis of an application specific time multiplexed architecture for DSP applications is a complex task in which the performance criteria have to be met without compromising the specified behavior or system constraints. One way of dealing with such a complex task is to split it up into subproblems that are solved in a certain order with possible iterations over sets of subproblems. The method of problem partitioning will depend on the selected target domain of applications and the target architectural style, in order to retain efficiency. The *Cathedral* synthesis systems are examples of how this approach can be used to automate architecture synthesis [11, 9, 10]. The target domain of applications for the Cathedral systems have the following algorithmic properties:

- Hierarchically nested loops.
- Locally and globally nested conditions.
- Recursive signal dependencies.
- Fixed and varying I/O rates.
- Large multi-dimensional signals.

These properties are commonly found in e.g. video, image processing, telecommunication and audio applications. The amount of hardware time multiplexing that can be used depends on the timing constraints given in a system specification. Different performance specifications and different types of algorithms will lead to different *styles* of architectures that implement a given system most efficiently. These styles differ in e.g the level of hardware multiplexing, the controller structures (micro-coded vs hard-wired), memory organization (single vs multiple) and the data path composition (general purpose units vs specialized heavily pipelined hardware). In the Cathedral system philosophy, different styles of architectures are best synthesized by different compilers, each tuned towards a particular style [11, 9]. The same applies for some other signal processing oriented synthesis systems as FACE [8] and Phideo/Pyramid [31]. In these synthesis systems an important task that should be performed early in the design trajectory is the definition of a management scheme for data storage and retrieval in systems. This is called *High Level Memory Management.* The novel data flow and control flow model that is presented in this paper is demonstrated by its use in the High Level Memory Management task of system synthesis.

1.1 Modeling data flow and control flow

The model, which is used in a synthesis process for capturing the data flow and control flow, reflects an interpretation of the given algorithmic specification. If, for example, loop iterators in the algorithmic specification are interpreted as signals which are to be implemented as any other signal in the architecture, then these iterators will be modeled as any other signal in the data flow model. The same observation holds for control flow modeling. If, for example, loop structures in the algorithm are interpreted as control flow specifications, then the data flow model will feature loop structure specification, which allows for control flow modeling directly on the data flow model.

At least two factors play a role in determining the interpretation of algorithmic specifications. The first factor is the *target style of architecture* for synthesizing architectures from algorithm specifications. The second is the *type of optimizations* which are foreseen in the synthesis process.

The style of architecture determines to a large extent the most efficient way of implementing certain algorithmic features. For example, in a heavily time-multiplexed architecture, the operations described in a loop will be sequentially executed on the same hardware. In this case it seems quite natural to implement the loop iterator as an actual signal, which can then be used to index multi-dimensional signals inside the loop body. In other architectural styles, like systolic or regular arrays (RA's) [23, 9]), the operations within a loop body may be executed in parallel on separate hardware. Loop iterators are for this style of architecture not seen as signals to be implemented by the hardware.

The type of optimizations performed in the synthesis process determine which aspects of an algorithmic specification are to be taken as a fixed spec-

ification and which aspects may be changed. Depending on the 'level' of a synthesis task this may range from a strict implementation of each and every signal and operation, to an implementation with only the specified input-output behavior. The 'lower' the level of optimization the more 'literal' the specification will be implemented. For example, low level scheduling will closely follow the loop structure given in the algorithm, while this loop structure is part of the algorithm that is to be optimized in High Level Memory Management (HLMM) for time multiplexed architectures [59, 13, 30, 57]. This is also the case in high level transformations for regular array synthesis [10].

1.2 Data- and control flow modeling for HLMM

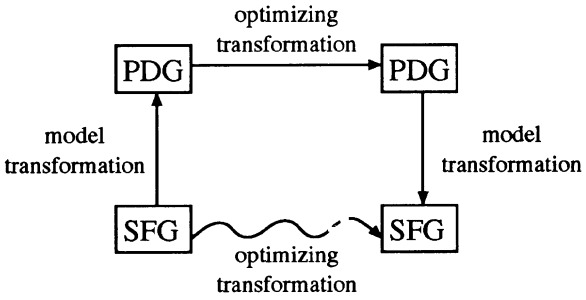
In real-time signal processing systems large quantities of data are processed. These data are most often expressed as multi-dimensional signals. Processing large multi-dimensional signals in real-time poses not only computational problems but also storage problems for background memory [49, 58]. In studies of memory management methods [58, 13, 59] it has been recognized that the efficiency of a chosen storage scheme is mainly dependent on the relative ordering of the execution of computations, which from now will be called *ordering of computations*. The task of organizing the multi-dimensional signals in background memories is crucial in the synthesis of efficient AS architectures. Therefore, it should precede data path synthesis. The task of high-level memory management (HLMM) in time multiplexed architecture synthesis is to devise an efficient scheme for multi-dimensional signal storage and retrieval. It will only produce constraints on the ordering of computations related to multi-dimensional signals. This is different from schedulers that fix absolute cycle instances [26, 37].

In order to allow for maximal flexibility in control flow optimization, none of the possible control flows should be excluded in advance from the search space in the memory management task, except those incoherent with a given data flow. In other words, the syntactical structure of an algorithm description ***should not be used*** to limit the set of possible control flows. Nevertheless, this structure is often used in architecture synthesis methods to derive a control flow, even for non-procedurally described algorithms. It is widely regarded as a part of the algorithm specification that has to be implemented instead of seeing it for what it is: a syntactical convenient way to describe a data flow.

Information on the structure of the data flow is in HLMM as important as in the synthesis of regular array architectures. Regularity can be exploited to derive read-write sequences of signal instances that minimize storage and addressing costs. Furthermore, the lower complexity of the data flow model obtained by exploiting regularity is essential in optimizing global memory managements costs for very large multi-dimensional signals. The treatment of individual signal instances is in these cases a practical obstacle. The same is true for specifying control flow. Examples of this can be found in [58, 57, 30, 52].

2 Conspectus

Based on the requirements discussed in the previous sections, a specification of a novel data flow and control flow model is discussed in section 4. The traditional data flow and control flow model for architecture synthesis for real-time signal processing algorithms is the Signal Flow Graph (SFG) [22, 23]. In section 5 it will be shown that this model does not satisfy the requirements of HLMM for time multiplexed architectures.
The principles of a model that complies with the given specifications are discussed in section 6 and 7. The proposed model exploits for its basic elements the vast body of theory, techniques and methods present in the fields of combinatorial optimization, array architecture design and parallelizing/vectorizing compilers. These basic elements are used in a structure that is tuned to high level architecture synthesis for signal processing algorithms.
It will be shown that specifications indicate the need for *control flow independent data modeling.* The data flow is given solely in terms of operations consuming and producing signal instances and not in terms of loop structures and iterators. A signal instance is a single signal of a multi-dimensional signal. Control flow is defined in terms of placement of the operations in a common space and an ordering vector defined over that space. Control flow transformations can then be easily expressed as a re-indexing (repositioning) of operations in the common space and/or as a change in the ordering vector. In section 7.5 it will be shown that control flow optimizations for HLMM are easier to steer and express in the proposed model than in a SFG model.



In this paper it is proposed to replace the direct SFG optimizing transformation by a SFG to PDG (Polyhedral Dependency Graph) model transformation, followed by an optimizing PDG transformation after which the result is transformed back to the SFG model.
Section 8 discusses extensions to the model to accommodate non-affine index functions. The use of the PDG model for HLMM is demonstrated for a real-life application in section 9. Performance figures for a prototype CAD tool for model extraction are given in section 10.

The introduction of this modeling approach to the field of high level synthesis results in the formalization of control flow transformation, which thereby

generalizes traditional loop transformations. As a result, control flow optimization techniques can be devised that are more general, controllable and efficacious than those based on syntactical structure transformations. Although the concept of the type of control flow specification used in this model has been known in array synthesis and parallelizing/vectorizing compilers, it has never been linked to an appropriate data flow model which enables to steer control flow transformations based on dependency analysis. In this paper it is motivated that this novel approach changes the way synthesis researchers can incorporate the effect of control flow transformations and is a real step forward in coping with the complexity of the exploration of control flow alterations.

3 The algorithm model

The algorithm model that is used in this paper is a set of Conditional Affine Recurrence Equations (CARE's). This model follows the single assignment principle. A CARE can be formally defined as:

$$\{x \in \mathcal{Z}^{n_k} \mid C_k x \geq c_k\}: \quad U_k(A_k x + b_k) = f_k[\ldots, V_k^i(A_k^i x + b_k^i), \ldots] \qquad (1)$$

with:

- k is the identification number of the CARE.
- $\mathcal{Z}^n$ is the set of integer points in an n dimensional Euclidean space $\mathcal{R}^n$ with $\mathcal{R}$ the set of reals.i
- $\{x \in \mathcal{Z}^{n_k} \mid C_k x \geq c_k\}$ defines the set of integer vectors with dimension n_k over which the CARE is defined. C_k is a $m \times n_k$ matrix and c_k a $m \times 1$ vector.
- U_k is a multi-dimensional signal which is (partially) being defined by the CARE.
- $I_k(x) = A_k x + b_k$ is an affine mapping $\mathcal{Z}^{n_k} \rightarrow \mathcal{Z}^{n_{U_k}}$ with n_{U_k} the dimension over which U_k is defined. A_k is a $n_{U_k} \times n_k$ matrix and b_k is a $n_{U_k} \times 1$ vector. $I_k(x)$ is an *index function.*
- $f_k[]$ is the operation performed by the CARE. Its function may be dependent on the value of its operands.
- V_k^i is the i-th operand of the CARE with label k.
- $I_k^i(x) = A_k^i x + b_k^i$ is the affine mapping $\mathcal{Z}^{n_k} \rightarrow \mathcal{Z}^{n_{V_k^i}}$ with $n_{V_k^i}$ the dimension over which V_k^i is defined.
- All matrices and vectors have rational coefficients.

Note that this model covers only a subset of the characteristics of the algorithms in the target application domain, as specified in section 1. Extensions to the proposed data flow and control flow model to cover the remaining characteristics are discussed in section 8.

4 Requirements for a novel data flow and control flow model

The following model requirements can be extracted from the discussion of the synthesis tasks in section 1.2:

1. Efficient modeling of the *exact* dependencies between *individual* operations, operands and defined signal instances in Conditional Affine Recurrence Equations (CARE's) (see eq. 1, section 3).
2. Modeling of the *structure* of the data flow.
3. Concise control flow specification, based on the structure of the data flow.
4. Modeling in terms of expressions which can be directly used in mathematical analysis and optimization methods.

The dependencies referred to in item 1 are those which would result after completely 'unrolling' an applicative algorithmic description. Computations and their dependencies made on behalf of signal indexing via *loop iterators* or *conditions on loop iterators* are *not* meant by the first item. These computations determine the *structure* of the data flow. Optimization and specification of control flow is often based on this structure, as explained in section 1.2.

If dependencies between individual operations and signal instances are to be modeled then they are to be referenced as well. This means that there must be basic elements in the model, which are somehow associated with individual operations and signal instances, to which to refer. Furthermore, referencing must be done in a mathematical way. Assuming that these basic elements exist, the structure of a data flow can be represented by mathematical relations between the basic elements. Control flow based on structure is then equivalent to control flow based on mathematical relations between model elements. These requirements lead to the following model specification:

A: Signal *instances* and *individual* operations must be associated with model-elements which can be related to each other by mathematical means. The relation between model elements are called dependencies.

B: The structure of a data flow must be expressed in terms of the model elements in a mathematical way.

C: Control flow specification is based on relations between model elements.

Item A is illustrated in figure 1. Figure 2 shows how, according to specifications, instances of recurrence equations are related to each other by use of the model elements. Note that both occurrences of the signal instance $a[5]$ are associated with the same model-element. A different signal instance, e.g. $a[10]$, would be associated with a different model-element. The 'model-elements' correspond to *vectors*. Each signal instance is associated with a vector that is unique to all vectors associated with signal instances with the same name. Likewise,

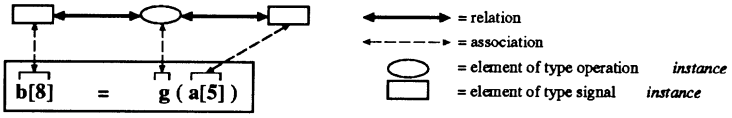


Figure 1: Model elements and their relations and associations.

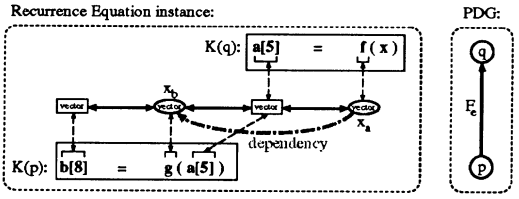


Figure 2: Recurrence equation relations.

individual operations are associated with a vector that is unique to all vectors associated with individual operations from the same recurrence equation. Because of the single assignment principle, the relations between vectors are unique as well: a specific individual operation uses specific signal instances as operands to define another specific signal instance. The structure of a data flow can now be modeled by collecting vectors into sets, defined by polytopes (section 6), and by collecting vector-vector relationships into affine functions (see figure 3 and section 7). Control flow can then be specified in terms of functions on vectors which assign to each vector in a polytope a number (or vector) that represents the ordering of the computations associated with that vector (section 6.3).

5 Related work

5.0.1 SFG models

Recently, methods have been proposed to alter loop structures in algorithm descriptions within the context of AS architecture synthesis (e.g. [57, 59, 31]). The used transformations are based on those applied in optimizing software compilers [36, 39]. It has been found difficult to formalize these transforma-

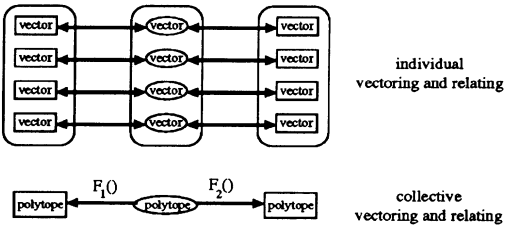


Figure 3: Capturing data flow structure.

tions in terms of the *global* optimization criteria to be used. The reason for the difficulties in the formalization of transformations, leading to these suboptimal methods, is the following. By deriving control flow directly from the syntactical structure of an algorithm description, this structure determines both data flow and control flow. The *global* relation between syntactical structure, data flow and control flow can be quite intricate. Therefore, if *local* transformations are expressed in terms of syntactical structure then both the data flow directed constraints on these transformations as well as the quantitative effects of them on implementation aspects, like storage efficiency and potential parallelism, will be hard to express (especially mathematically). The SFG model, as used in most architecture synthesis systems, is based on the syntactical structure of the algorithm specification and is therefore not amenable to supporting the control flow optimizations as described in this paper.
Another way in which the SFG model does not comply with the proposed specifications is that it does not specify relations between signal instances and single operations. This is because nodes in a SFG represent sets of operations, while the arcs are simple dependencies that do not take this into account. An example is given in section 7.5.

Sets of linear inequalities are an integral part of the definition of CAREs (eq. 1). Consequently, data flow analysis and modification as well as control flow specification will be expressed in terms of these sets. Therefore, sets of inequalities and *operations on sets of inequalities* must be fully supported by the data- and control flow model. This is clearly not the case for a traditional SFG.

Many tasks in regular array synthesis require an *exact* knowledge of the dependencies between multi-dimensional signal instances. These dependencies should be explicitly expressed in a data flow model. They are only *implicitly* expressed in a SFG by a web of operations, iterators and signals formed by iterator-, condition- and index constructions. The SFG model is, therefore, not suited to support optimization tasks that use these dependencies in optimality criteria and/or alter them as part of their task.

5.0.2 Stream model

The Phideo compiler [30, 31] uses a stream model for data flow descriptions. A stream is defined by mapping a multi-dimensional signal to the *time* axis by use of a linear function. The offset of the stream on the time axis is controlled by a scheduler. Dependencies between streams are resolved by symbolic simulation, or by solving an ILP problem as discussed in e.g. [40]. Streams of this kind are naturally occurring in the target application domain of the Phideo compiler, which consists mainly of video applications.
The drawbacks of the method are the mapping to a time axis, the lack of exact dependency descriptions and the size parameter dependent complexity. The mapping of a signal to the time axis by a linear function leads to a fragmented stream if the domain to be mapped is not rectangular. Fragments are treated individually, which means that the model complexity is dependent on the size

parameters of the signals involved. This is extremely undesirable. Furthermore, the model does not explicitly model individual dependencies between signals and operations, which leads to a worst-case dependency characterization. This severly limits the model's applicability in global control flow optimization.

5.0.3 Polyhedral models

The different algorithm models, such as CAREs and UREs, can be regarded as different data flow models for regular- or systolic array synthesis. Most array synthesis systems that use these models (a.o. [12, 41, 33]) expect the algorithm specification to be given according to the model. If not, then nested loop programs, like Fortran do-loops, are converted into the appropriate model (e.g. [6]). These models are closely related to the model proposed in this paper. They also associate vectors to operations and collect these vectors in polyhedrals. However, in most RA data flow models the operations from all recurrence equations which appear in *the same loop body* are in a fixed way associated with vectors from *the same space*. One of the aspects in which the model proposed in this paper differs from the other models is that vector associations are more flexible. This is needed because fixed associations unnecessarily restrict control flow optimizations. Furthermore, relations between operations and signal instances are not explicitly defined in data flow models derived from algorithm specifications. These relations are needed if arbitrary affine index functions on multi-dimensional signals are allowed in algorithms consisting of several depending nested loop structures. Advanced control flow optimization techniques [53, 54] need this information to be explicitly modeled.

The type of RA synthesis tasks which have been investigated up till now in array synthesis did not prompt the development of a data flow model beyond an exact copy of an algorithm model. The only exception to this might be the Dependence Graph (DG) [23, 28, 2]. Each node in this graph represents a single operation and each arc represents a single dependency. The size of the graph depends on the size of the problem. Clearly, this is not a very convenient data flow model when problem sizes are large, as they often are in real-time signal processing. Furthermore, useful information encapsulated by a recurrence equation on common characteristics of sets of operations and dependencies is lost in a DG. This complicates most synthesis tasks.

5.0.4 Parallel computation models

The problem of dependency analysis has been addressed extensively in optimizing compilers for parallel/vector computers [36, 39, 55]. Initially the goal of these compilers was *not* to explicitly describe the existing dependencies, but to indicate the operations that have no dependencies. This information describes the parallelism available in an algorithm, which is used to maximize the parallel execution of operations. The data flow models and optimization strategies developed for these compilers are related to a sequential algorithm model, which is different from the non-sequential CARE model.

Recent work in this field shows an increasing interest in the overhead associated with parallel computation, like load balancing and data communication

[38],[1] and [21]. When these factors are taken into account, then the problems to be solved by parallelizing/vectorizing compilers become closely related to those of array synthesis methods that map algorithms to fixed Single Instruction Multiple Data (SIMD) machines. In fact, an efficient array architecture can be regarded as a perfectly balanced parallel computer with 100% resource utilization. As a consequence of these problem similarities, data flow and control flow modeling are also modeled in an increasingly similar way. In [38], sets of Uniform Recurrence Equations (UREs) are modeled by use of a Reduced Dependence Graph (RDG), a widely used model in systolic array synthesis [23, 63, 43]. Also in [61] UREs are used along with unimodular transformations to formalize 3 types of loop transformations. Since UREs are a subset of CAREs, this model is too limited. A rudimentary data flow model capable of expressing a limited subset of UREs is described in [1]. The limitation is in the shape of the domains of computation that are considered (rectangular and hexagonal).
However, publications in this field indicate a rapid development of the data- and control flow models in the direction of the model presented in this paper. Developments in this field should be much closer watched than they have been up till now in the high-level-synthesis community.

6 The principles of the proposed model

Interpretation of specification

The key to the model specified in the previous section is the assignment of vectors and the extraction of structure information from an initial algorithmic specification. This assumes a certain interpretation of an *initial specification* by an applicative algorithm description. In this paper the following is proposed:

1. A set of nested loop iterators defines a *lattice* $\{x \in \mathcal{Z}^n\}$ in an Euclidean n-space.

2. Inequalities on iterators imposed by conditional expressions and loop boundaries define *sets* of points belonging to the lattice.

3. Loop- and condition structures do *not* specify or imply a control flow.

The first two parts of the interpretation allow for an efficient modeling of data flow. The last part allows for a control flow specification that is independent from the loop- and condition structures by which the data flow is initially specified.

Restrictions on modeling

1. The data flow modeling is only exact when signals are indexed by affine functions of iterators and conditions are affine inequalities on iterators.

2. Control flow is specified by an affine function on vectors, associated with operations, in a single Euclidean n-space.

The first restriction poses problems in HLMM, where data-dependent and non-linear indexing and conditioning are frequently encountered. In [14] proposals are made for extending the model to feature correct data flow modeling under these types of indexing/conditioning. The second restriction has not yet found to of influence in practical examples of target domain applications.
In the target domain of HLMM for lightly multiplexed architectures many optimal storage schemes have found to be a concatenation of relatively large pieces of signal sets which can are addressed in an affine way [58]. The model provides the means to re-index these pieces individually, after which a single affine ordering function suffices.

The different parts of the interpretation and the usefulness of the model under the given restrictions in control flow optimization tasks will be motivated and illustrated in the remainder of this section.

Algorithmic descriptions
Two different types of algorithmic descriptions will be used, in addition to the CARE description:

- An applicative description, i.e. *implying no ordering of operations*, except for those inferred by the data dependencies.
  This type of description is a syntactical short hand notation for specifying *nothing more* than a dependency graph. The iteration syntax is defined by: '(index-name: range) ::' and a unit step size.
  (similar to the Silage-language style [22] and related to a SFG/DSFG [24])
- A procedural description, i.e. *implying a sequential ordering of operations.*
  The iteration syntax is defined as: 'for index-name = range do .. od' and a unit step size.
  (similar to the Fortran style)

6.1 Loop iterators and lattices

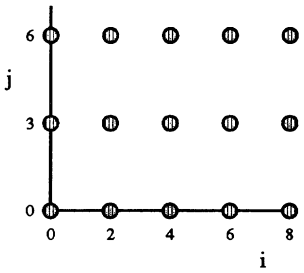
A loop iterator x represents an element of $\mathcal{Z}$. A set of n nested loops define an n-tuple of loop iterators $(x_1, \ldots, x_n)$. This n-tuple represents a *vector* or *point* in an Euclidean n-space [25]. The set of vectors which can be represented by n loop iterators from a set of n nested loops is a subset of $\mathcal{Z}^n$. Loop iterators of loops which are not nested are not part of the same tuple defining a vector.

The space set up by loop iterators has been given various names in literature, a.o.. *index space* and *iteration space*. Each recurrence equation has its own Euclidean space associated with it. Because a recurrence equation is specified by a node in a SFG, the space is called a *Node space.*

The lattice specified by a set of n nested loop iterators can be different from $\mathcal{Z}^n$ if a 'step' is associated with each iterator. For example:

$(i : \ldots)$ step 2 ::
  $(j : \ldots)$ step 3 ::

defines lattice:⟶



Only the first quadrant is shown for this example. In general, a lattice may be defined as $\{y \in \mathcal{R}^m : y = Ax, x \in \mathcal{Z}^n\}$ by a set of nested loops, with A a $m \times n$ matrix with integer coefficients.

In the rest of this paper it is assumed that no steps are specified with loops. Therefore, lattices are always $\mathcal{Z}^n$. Extensions for lattices different from $\mathcal{Z}^n$ are considered in [14].

6.2 Loop boundaries, conditions and sets of lattice points

A recurrence equation in an applicative algorithm description is defined within a set of nested loops and, possibly, under a set of conditions. The loop boundaries and conditions define a region, or *domain*, of the lattice set up by the loop iterators on which the recurrence equation is defined. Consider the following example:

$(i : 1..10)$ ::
  $(j : 1..i)$ ::
    "recurrence equation"

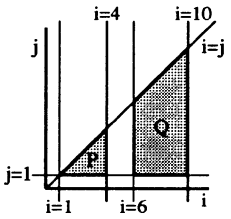
defines domain:⟶

$i, j \in \mathcal{Z}$:
$i \geq 1$ and
$i \leq 10$ and
$j \geq 1$ and
$j \leq i$

In general, a loop definition $(x : b_1..b_2)$ in which b_1 and b_2 are affine expressions of other loop iterators and constants, defines two inequalities: $x \geq b_1$ and $x \leq b_2$. In the following example a condition is added to the recurrence equation:

$(i : 1..10)$ ::
  $(j : 1..i)$ ::
    $\ldots = if\ (i \neq 5)\ \ldots;$

defines domain:⟶



The domain over which a recurrence equation is defined is specified by the *union* of inequalities defined by the loops and by the conditions belonging to that recurrence equation. Different recurrence equations from the same loop body can therefore be defined over different domains.

The convex grey regions P and Q are the *polytopes* which make up the complete domain over which the recurrence equation is defined. A polytope is a bounded *polyhedron*. A polyhedron is a set of points that satisfy a finite

number of linear inequalities [34]. The points that are considered here are all elements of $\mathcal{Z}^n$. Therefore, in the context of the data flow model, a polytope is a set of elements of $\mathcal{Z}^n$ that satisfy a finite number of linear inequalities. For example:

$$P = \{\begin{pmatrix} i \\ j \end{pmatrix} = x \in \mathcal{Z}^2 : Cx \geq c\} \text{ with } C = \begin{bmatrix} 1 & -1 \\ 0 & 1 \\ -1 & 0 \end{bmatrix} \text{ and } c = \begin{bmatrix} 0 \\ 1 \\ -4 \end{bmatrix}$$

The inequalities which bound P: $i - j \geq 0$, $j \geq 1$ and $-i \geq -4$ are given by the matrix expression $Cx \geq c$. Issues concerning the extraction of polytopes such as how to obtain a set of non redundant inequalities and using *extreme points* as an alternative description of a polytope are dealt with in [53].

Note that in equation 1 (section 3) the domain of a CARE is given by a single polytope $C_k x \geq c_k$. Two CARE's are therefore needed to express the recurrence equation of this example, one for each of the polytopes.

The importance of using polytopes to indicate sets of points is that it is precise, concise and allows for easy dependency analysis without symbolic simulation. Furthermore, many powerful mathematical methods and techniques are available for handling polytopes [34, 46]. These methods and techniques can be readily used in the analysis and optimization of problems which rely on polytopes for their modeling.

Definition 1 Operation Placement *is the association of vectors* $\in \mathcal{Z}^n$ *with operations of a recurrence equation, where the vectors are defined in the Euclidean n-space set up by the n nested loops in which a recurrence equation is defined.*

Example 1

$$(i : 1..n) ::$$
$$a[i] = f(a[i-1]);$$

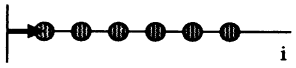
The set of vectors defined by the loop boundaries is given by $\{i \in \mathcal{Z} : i \geq 1 \wedge i \leq n\}$. For each vector in this set, a specific operation is performed on a specific operand and assigned to a specific signal instance. This assignment of vectors to specific operations is depicted by the placement of dots on a line in example 1. The dots represent operations, and their position on the line represents their corresponding vector in the Euclidean 1-space. For example, operation $f(a[3])$ is placed on point (4). The arrows in the figure represent *dependencies* between the operations defined by the recurrence equation. Note that the dependency belonging to the assignment $a[1] = f(a[0])$ is not present in the figure. This is because the operation which assigns $a[0]$ its value is not defined by the given CARE, therefore it has not been assigned a vector in the Euclidean 1-space belonging to that CARE.

6.3 Control flow specification by ordering vectors

Example 1 describes a data flow but not yet a control flow. A *procedural interpretation* of the Silage code in that example would lead to the control flow depicted in the following figure:

Example 2

```
for r=1 to n
  a[r] = f(a[r-1])
```



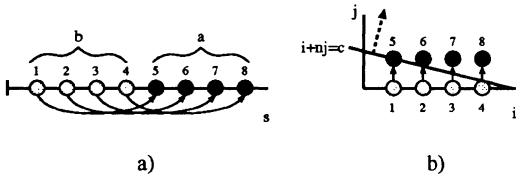
The placement of the operations by the CARE is also given in this figure. A control flow, which is a partial ordering of computations, can be described by an ordering of the corresponding operations. If to each operation a unique vector is assigned, then an *ordering can be defined by a function $T()$ of these vectors.* In example 2, the function $T(i) = i$ would specify the ordering given in the procedural description.
Note that this function defines a *relative ordering* of operations and *not* a schedule in terms of cycles. Schedules are orderings on the execution of operations in terms of an absolute time axis. Scheduling is done later on in the synthesis trajectory. The function indicates that the computation $a[1]$ precedes that of $a[2]$, or $a[1] \prec a[2]$, since $T(1) < T(2)$. The same relative ordering would be given by e.g. $T(i) = 2i$ or $T(i) = 10i$. Because $T(i)$ is linear in i, it can be represented by a vector. The bold arrow in the example 2 indicates the *ordering vector* corresponding to $T(i) = i$. The inner product of this vector with a vector assigned to an operations will give the operation's relative order.

Clearly, there is usually more than one way in which a certain control flow can be specified by a placement of operations and an ordering vector. Consider the following example:

Example 3

$$
\begin{aligned}
&(k : 1..n) :: \\
&\quad b[k] = g(c[k]); \\
&(l : 1..n) :: \\
&\quad a[l] = f(b[l]);
\end{aligned}
$$

Suppose we want to define a control flow for this data flow which corresponds to a procedural interpretation of the applicative code: $b[1] \prec \ldots \prec b[n] \prec a[1] \prec \ldots \prec a[n]$. Figure 4 gives a procedural code with this control flow. The total set of operations is defined over two different recurrence equations. They are therefore initially placed in two different Node spaces. If a *global* relative order is to be defined by means of operation placement and a *single* ordering vector, then all operations should be placed in a *common Node space.* Figure 4 shows just two of many possible ways to do this. In figure 4a, all signal instances are placed within a common 1-dimensional node space 'z1' with ordering function



a) b)

```
for r=1 to n
do
  b[r] = g(c[r]);
od;
for r=1 to n
do
  a[r] = f(b[r]);
od;
```

Figure 4: Two different ways of expressing the same control flow (n=8).

$T_{z1}(s) = s$. The same ordering can be obtained by an alternative placement and ordering function $T_{z2}(i,j) = i + nj$, as shown in figure 4b. This ordering is also described by the procedural code on the right.

Definition 2 *If the ordering vector is represented by a row vector* Π, *then the operations on two points* p, q *in the common node space have an execution ordering given by* Πp *and* Πq. *If* $\Pi p < \Pi q$ *then* p *will be executed before* q. *The constraints on* Π *are: if there is a dependency* d_{pq} *between point* p *and* q *then* Π *must be chosen such that* $\Pi d_{pq} \geq 0$.

The line $i + nj = c$ in figure 4 represents the *equi-ordering* line, i.e. operations on this line are ordered at the same ordering point. The order increases in the direction of the ordering vector Π, which is perpendicular to this line: $\Pi(i\ j)^T = i + nj = c$. It is clear that for a given placement only a small set of relevant linear orderings exist.

Note that the concept of using a vector to *schedule* operations which are placed in an Euclidean space has been used extensively in systolic array synthesis [33, 41] and is known in parallelizing/vectorizing compilers [48].

6.4 Formalizing control flow transformations

The formalization of control flow transformations by use of the placement and ordering vector definitions will be illustrated by showing how a number of well known loop transformations [57, 59, 18, 39, 36] can be elegantly described by this technique. The impact of the formalization of these transformations on HLMM is demonstrated here by use of a much simplified memory cost model. In the examples used in this section, the memory cost is modeled as being directly proportional to the maximum number of signal instances to be stored. A more complete overview of the aspects that determine memory cost is given in section 9.

6.4.1 Loop merging/fusion

Loop merging is, as all loop transformations, defined in terms of procedural code. Loop merging applied to the procedural code of figure 4 can be expressed as follows.

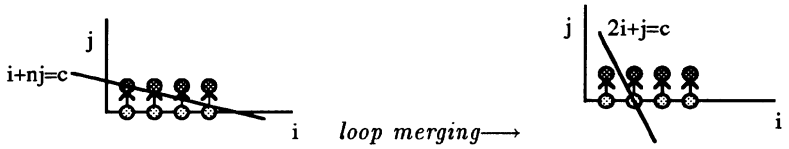


Figure 5: Loop merging represented as a change in ordering vector.

Example 4

```
for r=1 to n
do
  b[r] = g(c[r]);
od;
for r=1 to n
do
  a[r] = f(b[r]);
od;
```

loop merging⟶

```
for r=1 to n
do
  b[r] = g(c[r]);
  a[r] = f(b[r]);
od;
```

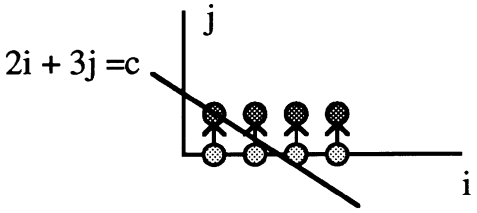
The same transformation of control flow can be obtained by changing the ordering vector in figure 4b from $\Pi_1 = (1\ n)$ to $\Pi_2 = (2\ 1)$, as shown in figure 5. The importance of this transformation in example 4 is that the number of signal instances of $b[]$ to be stored has been reduced. For Π_1 a maximum of as much as n signal instances have to be stored, while for Π_2 this is only 1 signal instance. These signal storage figures can be easily extracted by determining the maximum number of dependencies that cross the equi-ordering hyperplane at any position in space.

Loop fission or iterator splitting is the inverse transformation to loop merging, and can be described by the inverse ordering vector transformation.

6.4.2 Loop folding/winding/pipelining

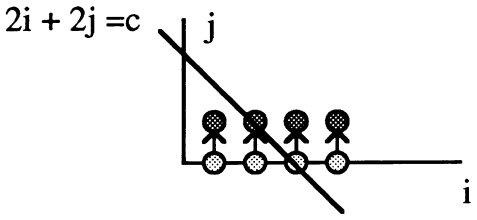
Loop folding is a control flow transformation used in schedulers and compilers to introduce functional pipelining [18, 16, 39]. It allows operations from a sequence of loop iterations to be executed in parallel. Although this type of optimization has only recently been introduced for schedulers in high level synthesis, it has been the basic principle of systolic array synthesis since its conception. It shouldn't be surprising then that this transformation can also be easily described in the proposed control flow model. In compilers a procedural model is used but without real optimization with a clear objective function.

Consider again the example of figure 4b. Loop folding is performed, just as loop merging, by changing the ordering vector. Figure 6 shows an ordering vector change from $\Pi_1 = (1\ n)$ to $\Pi_3 = (2\ 3)$ for the given placement of operations. The corresponding procedural code is given in the same figure. Setting $\Pi_4 = (2\ 2)$ corresponds to an ordering in which both $g()$ and $f()$ operations may be executed in parallel and two signal instances $b[p]$ and $b[p+1]$ can be accessed at one time point (figure 7). In a similar way various degrees




```
b[1] = g(c[1]);
for p=1 to n-1
do
  b[p+1] = g(c[p+1]);
  a[p] = f(b[p]);
od;
a[n] = f(b[n]);
```

Figure 6: Loop folding represented as a change in ordering vector.




```
b[1] = g(c[1]);
for p=1 to n-1
doparallel
  a[p] = f(b[p]);
  b[p+1] = g(c[p+1]);
odparallel;
a[n] = f(b[n]);
```

Figure 7: Loop folding with operations which are executed in parallel.

of folding can be selected by setting different Π vectors. Note that one more signal instance must be stored than before folding.

6.4.3 Loop migration/swapping/switching/interchanging

This transformation changes the nesting of loops in a procedural description or procedurally interpreted description of an algorithm.

Example 5

$(i : 1..n) ::$
  $(j : 1..m) ::$
    $a[i,j] = f(a[i-1,j]);$

procedural interpretation $\longrightarrow$

```
for p=1 to n
do
  for q=1 to m
  do
    a[p,q] = f(a[p-1,q]);
  od;
od;
```

The control flow belonging to the procedural interpretation of example 5 is described by the procedural code and by figure 8a. The ordering vector $\Pi_1 = (m\ 1)$ corresponds to the procedural interpretation of the Silage code. The ordering indicated by Π_1 causes a maximum of m signal instances to be stored. Loop migration is now performed by defining $\Pi_2 = (1\ n)$. Figure 8b shows the new ordering vector. Note that for $N = max(n,m)$ the vectors $\Pi_1' = (N\ 1)$ and $\Pi_2' = (1\ N)$ would define the same orderings as Π_1 and Π_2 respectively. In a *procedural* description this transformation corresponds to changing the loop hierarchy:

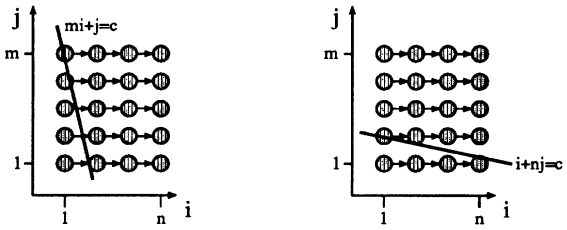


Figure 8: Loop migration represented as a change in ordering vector.

```
for q=1 to m                                    for p=1 to n
do                                              do
  for p=1 to n                                    for q=1 to m
  do                        Loop migration ⟶      do
    a[p,q] = f(a[p-1,q]);                           a[p,q] = f(a[p-1,q]);
  od;                                             od;
od;                                             od;
```

The ordering indicated by $\Pi_2 = (1\ n)$ requires only a single storage location instead of m locations before migration.

6.4.4 The steering of control flow transformations

Some of the loop transformations described in this section, like loop folding, are control flow transformations which can be described by a simple transformation of the ordering vector. For regular array synthesis, the selection of an optimal *scheduling* vector has been the subject of much research [33, 45, 42, 62, 64]. For HLMM this research is still in its initial phase. It is clear though from the examples that the effect on the storage cost can be orders of magnitude for applications with multi-dimensional signal processing with large iterator bounds. Ordering vector selection for HLMM will be discussed in section 9.

Other transformations, like loop merging and loop splitting/unwinding, require a transformation of the operation placement given by the CARE's. The steering of this type of transformation is discussed in [51, 53, 54].

6.5 The efficiency of the grouping of operations

The way in which vectors are assigned to operations, and vectors are clustered in polytopes constitutes a certain grouping of operations. This grouping is done according to the specified CARE's. Although this is a compact way of expressing information, it is not necessarily efficient in the synthesis process. In the loop transformation examples, the operations could be group-wise placed to obtain the desired control flows. In [53, 54] it is shown that polytopes sometimes have to be split in order to allow for more optimal placements. A method for polytope splitting is given in [53]. The proposed grouping of operations is only efficient for control flow optimization tasks in which this grouping does not have to be drastically changed.

7 The data flow model

The data flow model is based on the algorithm model for RA architecture synthesis: CARE's. The goal of the model is to represent the data flow in a way that is as independent as possible from the way in which the algorithm has been expressed in CARE's. Specifically, independence of absolute loop boundary values, loop structures and absolute inequality values is obtained. In this way the designer or synthesis system is presented with a data flow model in which these degrees of freedom are made explicit and can be used while preserving behavior.

It is assumed that the operations defined by the CARE's are *atomic operations*. If an atomic operation is composed of a set of more basic operations then these are not individually present in the model. The decomposition of atomic operations is only performed at the transition of abstract operations to hardware operators. From now on an 'operation' indicates an 'atomic operation'.

The conversion from a certain algorithm model used for specification to CARE's is often ambiguous. This is also the case for the conversion from Silage to CARE's. This conversion incorporates much of the interpretation made by the synthesis system of the algorithmic specification. It results in a certain choice of atomic operations and a choice in how to deal with the discrepancy in expressive power between the two models (see [53]). In order not to complicate the explanation of the model in the following section, it will be assumed that a recurrence equation in Silage corresponds to a CARE, and that its domain of definition can be defined by linear inequalities. Details on how to deal with algorithmic descriptions that do not satisfy these conditions are given in [53].

There are two basic types of dependencies in the data flow model:

- the dependency between an operation and the signal instance which it defines.
- the dependency between an operation and a signal instance which it uses as operand.

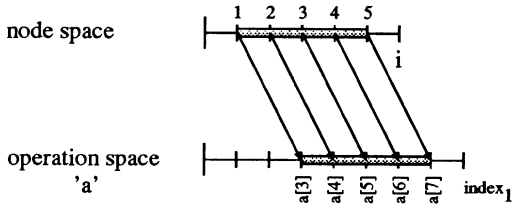
The representation of these two dependencies in the data flow model and their extraction from an initial algorithmic specification is discussed in the following two sections.

7.1 The operation space

The value of any signal *instance* is defined only *once* by a specific operation, because the algorithmic specification is assumed to be in single assignment code. Therefore, there exists a one-to-one relationship between operations in the node space and signal instances. This relationship, or dependency, can be modeled by an affine function between points in the node space (representing the operations) and points in the *operation space* of the signal. The points in the operation space are assigned to signal instances. Each operation space is therefore associated with a single specific multi-dimensional signal given in the code. The i-th coefficient in a vector in the operation space corresponds to the i-th index of the signal being defined by the recurrence equation.

Example 6

$(i : 1..5) ::$
$\quad a[i+2] = f(\ldots);$



The affine dependency between the spaces is specified by the multi-dimensional expression used to index the left hand side signal in a recurrence equation. This expression is the affine transformation which maps the polytopes in the node space to polytopes in the operation space. The result is that by applying a simple base transformation to the node space polytopes an exact description is obtained of the operation to signal instance dependencies.

The affine transformation can be expressed as $y = Ax + b$, with A an $m \times n$ matrix and the number of indices of the signal being defined is m and the dimension of the node space is n. Note that a correct signal assignment code may result in a singular A. However, there always exists an *equivalent* transformation with a non-singular A if :

1. the algorithmic description can be represented by CARE's (single assignment).

2. the dimension of the node space polytope and operation space polytope are equal.

$(i : 1..5) ::$
$\quad (j : 1..5) ::$
$\quad\quad \mathbf{a[i,i]} = \text{if } (i == j) \; -> f(\ldots);$

$A = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$

equivalent code:

$(i : 1..5) ::$
$\quad (j : 1..5) ::$
$\quad\quad \mathbf{a[i,j]} = \text{if } (i == j) \; -> f(\ldots);$

$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

In [53] the details are discussed on how to obtain a non-singular equivalent of A for all types of singularities. How to check the given code for correct use of the single assignment principle is also discussed in [53]. Note that the second criterion is easily satisfied by extending the dimensions of one of the spaces.

If the criteria for non-singularization of A are satisfied, then there is a link between node space and operation space polytopes which can be represented by an invertible affine function:

Node space $\underset{\longleftarrow f^{-1}(x)}{f(x) \longrightarrow}$ Operation space

7.2 Composite operations

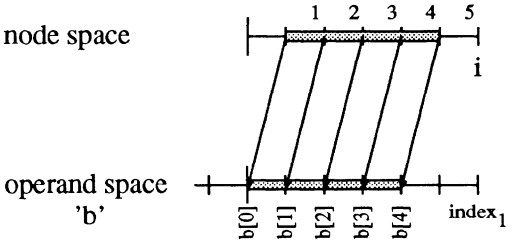
Some operations are composed of a set of other operations. Associating all the operations in a set with a single vector in a node space may not allow for enough flexibility in control flow specification (see next section). An easy way to make the composing operations 'visible' in the model is to *extend the dimension of the node space.* If a single operation is composed of k operations , then k dimensions can be added to the node space. The composing operations are placed on orthogonal vectors in this space. This will allow for any possible internal schedule to be described by a linear function. In this way, also control flow transformations like retiming can be described in the model and, more importantly, be combined with all other kinds of control flow transformations in a single global optimization problem. An example of this would be a composite operation consisting of a multiplication and addition. In order to increase hardware utilization, each of the components may be placed in perpendicular subspaces. In this way an ordering vector may be chosen such that the components are not get the same order, allowing for a pipelined use of the corresponding hardware units.

7.3 The operand space

Each operation represented in the node space uses certain signal instances as operands. This is *not* necessarily a one-to-one relation, since a set of operations may use the same operand. Therefore, the affine function which maps node space polytopes to operand spaces may be singular. Each operand space is associated with a certain multi-dimensional signal. For each operand of an operation, a separate operand space is derived. The i-th coefficient in a vector in the operand space corresponds to the i-th index of a signal being used by the recurrence equation as an operand.

Example 7

$(i : 1..5) ::$
$\quad \ldots = f(b[i-1]);$



The mapping process is almost identical to that of mapping to operation spaces, except for an extra polytope projection which can be performed as a result of broadcasting. In broadcasting, a single signal instance is used as operand by multiple operations. This many to one mapping results in a reduction of the dimension of the polytope on which the operations are defined when it is mapped to the operand space of the operand signal. Details can be found in [53].

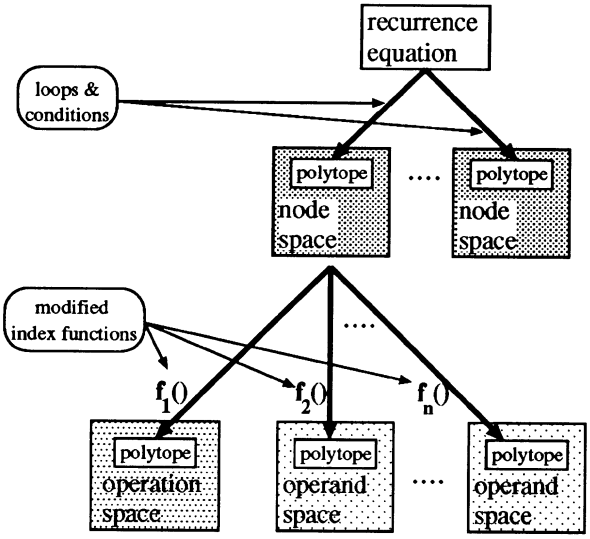


Figure 9: Summary of relationships for a single recurrence equation.

The link between node space and operand space polytopes can be represented by an affine function:

$$\text{Node space } f(x) \longrightarrow \text{Operand space}$$

7.4 The affine polyhedral dependency graph

The structure of spaces, polytopes and dependency-functions described in the previous sections is summarized in figure 9 for a *single* recurrence equation. A single recurrence equation may be associated with multiple polytopes, each in their own node space. Dependencies between operations are derived via the operation and operand spaces. Each polytope in a node space is represented by a node in a graph. The dependencies between the operations are specified by arcs between the polytopes to which they are associated. Each arc is associated with a dependency function, which expresses the dependencies between individual operations in a collective way.
The inverse dependencies are frequently needed in optimization tasks. For example, the inverse dependency is needed to find out the actual flow of data, which involves a mapping from operand space to node space. [53] deals with this problem.

Figure 10 gives an example of a recurrence equation with internal dependencies. Figure 10a shows the different spaces and their polytopes. Since both the operation space and operand space are defined for signal 'a', there is a one-to-one relationship between vectors assigned to signal instances of 'a' in

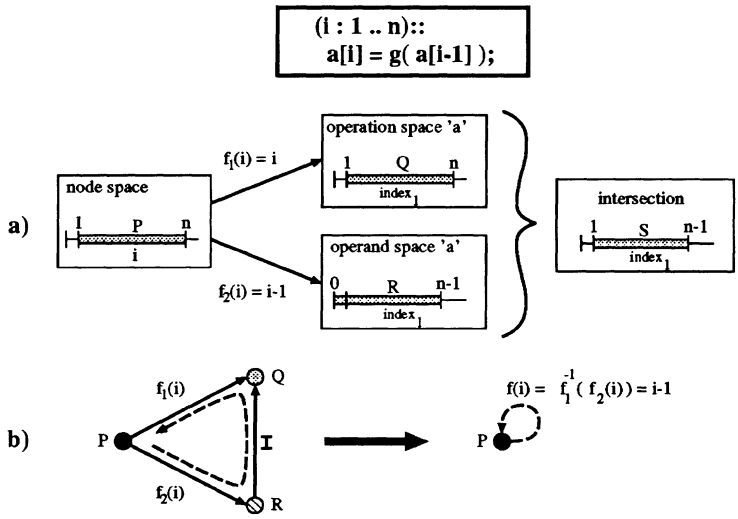


Figure 10: Example of recurrence equation with internal dependencies.

the two spaces. This is expressed in figure 10b by the identity function I. The identity relation between operation and operand spaces of the same multi-dimensional signal makes it possible to check dependencies between recurrence equations by checking whether the polytopes defined in the spaces have a non empty intersection. In figure 10 Q and R have a non empty intersection S. The internal dependency function of P can be easily extracted by computing $f(i) = f_2(f_1^{-1}(i))$, which follows from the path of the dashed arrow in figure 10b. The dependency function $f(i)$ is defined by going from node space to operand space to operation space to node space. The reason for this choice is that node space to operand space mapping functions often do not have an inverse.

Figure 11 extends the example by adding an extra operand. This is an example of how two recurrence equations, described in different loops, can be related to each other via functions between their node space polytopes. A more complex example can be found in section 9.

7.5 Comparison to the SFG model

The loop folding transformation as described in section 6.4, figure 7, in terms of the proposed model is now described in terms of a SFG [24]. The goal is to show that both *measurable* optimization criteria as well as transformations are much harder to describe in terms of the elements of the SFG model than in terms of the elements of the proposed model. Figure 12 shows the SFG belonging to the result of loop merging on example 4. The circles indicate SFG nodes. The dots denote indexing of multi dimensional signals by other signals in the SFG. Note that relations between signal instances are not explicitly defined in the SFG. Signal instances are referenced by indexing but are not explicitly

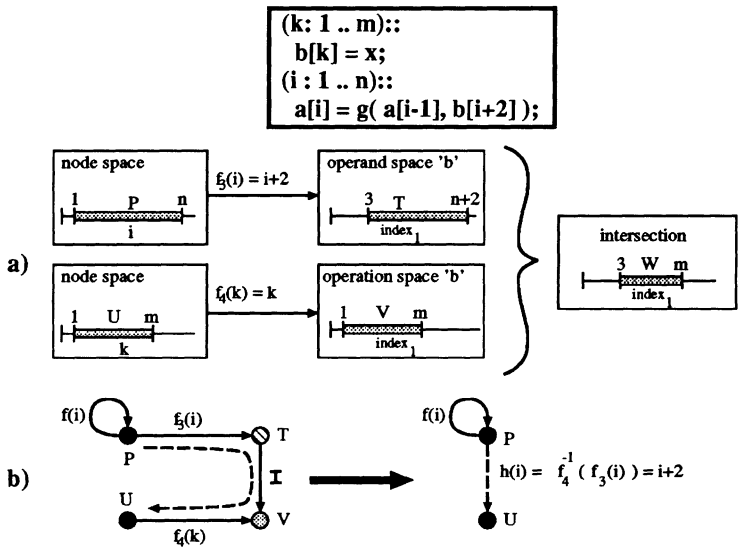


Figure 11: Extended example with relations between recurrence equations.

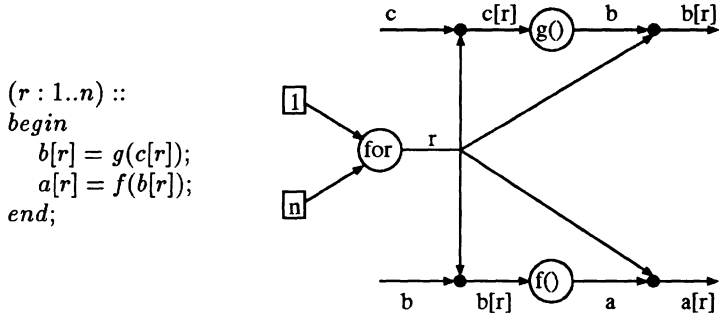


Figure 12: A Silage code and its SFG

$b[1] = g(c[1]);$
$(r : 1..n-1) ::$
begin
$\quad a[r] = f(b[r]);$
$\quad b[r+1] = g(c[r+1]);$
end;
$a[n] = f(b[n]);$

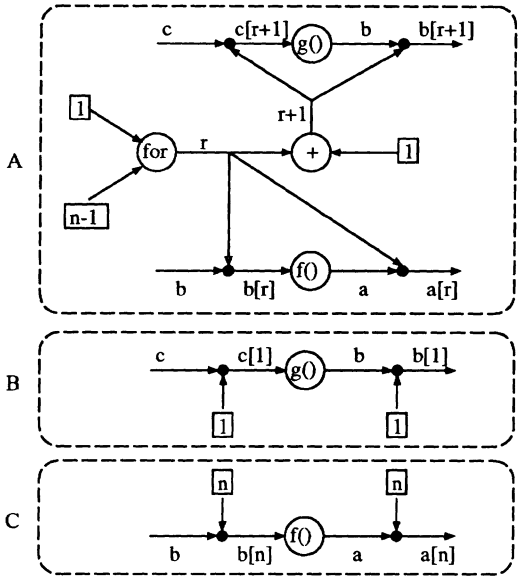
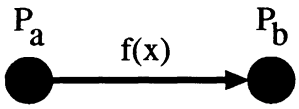


Figure 13: The Silage code after loop folding and its SFG

related. For example, the signal instances $b[r]$ resulting from the node $g()$ are not related in the SFG to the instances $b[r]$ used by $f()$. They are 'recreated' by separate indexing. Relations between signal instances and single operations must be derived by symbolic simulation or analysis.
The major problem with SFG is, therefore, that nodes and signals in the SFG represent *sets* of individual operations and signal instances, while the arcs in the graph are related to *lines* of code and signal *names*.
This mismatch of model elements prohibits the use of measurable entities based on mathematically defined relations.
The transformation on the SFG leads to the creation of 3 subgraphs (figure 13). Figure 14 shows the loop folding transformation in terms of the proposed model. What in the proposed model is obtained by a simple change of vector, is obtained in the SFG by a complete structure transformation. A measure like the maximum distance in ordering between two depending operations is easily extracted in the proposed model. For each of the two extreme points p of P_a the expression $\Pi(p-f(p))/\Delta$ is evaluated. The constant Δ is equal to the minimum ordering distance for a given Π. In this example $\Delta = 2$. The maximum of the two is the required measure. Extracting this measure from the SFG requires the evaluation of the relations between the new indexing signals. Since both structure and signals of the SFG have changed, previous evaluations and analysis are not reusable. Only by symbolic analysis or simulation, requiring enumeration of iterator values, this maximum can be found.



$$\text{inequalities of } P_a = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} x \geq \begin{bmatrix} 0 \\ -n \\ 1 \\ -1 \end{bmatrix}$$

$$\text{inequalities of } P_b = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} x \geq \begin{bmatrix} 0 \\ -n \\ 0 \\ 0 \end{bmatrix}$$

$$f(x) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

merging expressed by ordering vector: $\Pi = (1\ n)$
loop folding by changing vector to: $\Pi = (2\ 2)$

Figure 14: Loop folding as a change of ordering vector in the proposed model

8 Extensions for non-affine index functions

In section 1 the characteristics of the target domain of DSP algorithms is outlined. Clearly, a model based on CARE's covers a large part of these characteristics, but certain types of indexing functions are not supported. Among these the most important are: non-linear index functions and data dependent index functions. These functions can be created by conditional statements or direct by non linear computations. See [14] for examples.

Note that little attention has been given to modeling index functions because they are affine in the context of CARE's. However, non-linear and data dependent functions need an efficient model that blends in with the proposed data flow and control flow model.
In the context of the proposed model, a convenient way to model indexing functions is to couple their m-dimensional output values to their n-dimensional operands into $m+n$ dimensional vectors. For example, the function $f(i) = i+2$ can be represented by the vectors $\begin{pmatrix} i+2 \\ i \end{pmatrix}$. This is equivalent to plotting $f(i)$ on an axis against i on another axis. But instead of looking at this plot in terms of a curve, it can be considered as a set of points in a *value-iterator* space. Such a set can be modeled by a polytope, because the range of the iterators is bounded. A less straightforward use of this technique is demonstrated in figures 15 and 16. Figure 15 shows the values of the function $f(x) = x \bmod 4$ plotted against x. This piece wise linear function can be modeled by a sin-

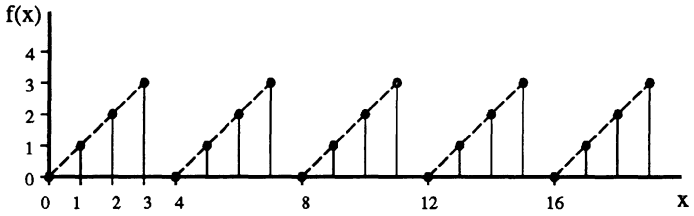


Figure 15: A modulo indexing function : $f(x) = x \bmod 4$.

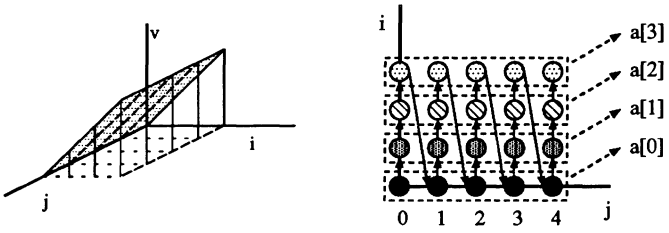


Figure 16: Modeling the function $f(x) = x \bmod 4$ by a polyhedral and an ordering function.

gle polyhedral and an ordering function as shown in figure 16. The function's values have been distributed over a 3 dimensional space set up by the axis i,j and the 'value-axis' $v = f(i,j)$. The relation between x and i,j is given by the simple ordering function $x = i + 4j$ and the bounds $0 \leq i \leq 3$. The bounds form a part of the polytope definition. The ordering function and the polytope together model the indexing of a signal $a[f(x)]$ as indicated by the arrows in the figure.
Note that the order of indexing has been given by the way in which the indexing function has been specified, namely by means of a modulo on an increasing x. This order may be changed, if the data dependencies allow this, in favor of a more efficient control flow. Other indexing orderings can be easily expressed by changing the corresponding ordering function.
It is clear that by modeling non linear index functions by polytopes and ordering functions, the optimization of control flow has become even more independent from the way the initial algorithm has been specified. The technique presented here can be easily extended to multi-dimensional indexing signals.

The polytope in figure 16 has a dimension lower than the space in which it is defined. This is because the function is manifest. It is known which *single* indexing value is produced for each element of i,j. In data dependent indexing, this value may be unknown during architecture synthesis.

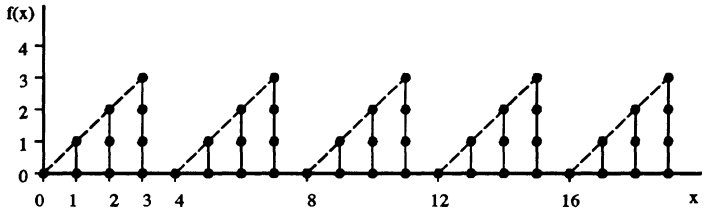


Figure 17: A *data dependent* modulo indexing function.

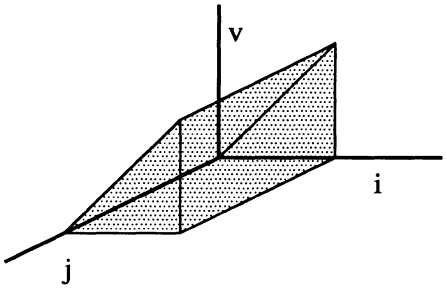


Figure 18: Modeling a data dependent indexing function by a single polytope.

Example 8
$p[-1] = 3;$
$(x : 0..19) ::$
begin
$\quad p[x] = p[x-1] + 1 \ mod \ 4;$
$\quad f[x] = input[x] \ mod \ p[x];$
$\quad \ldots = g(a[f[x]]);$
end;

In example 8 signal a is indexed by a data dependent function $f[x]$. The index function is depicted in figure 17. Clearly, the indexing values lie within a range. If this is the case, then to each i, j pair one or more values are coupled. This means that the polytope's dimension will increase in the direction of the v axis. This is shown in figure 18 for example 8. The faces of the polytope indicate the bounds of the index value ranges. Note that by using this model, ranges can vary for different i, j pairs. More details of this powerful index function modeling method can be found in [14].

9 Application of the model to High Level Memory Management

The use of the model in regular array synthesis, except for the relatively known tasks of localization and scheduling/assignment, is discussed in [51, 53]. The use

of the model in HLMM for optimization and analysis of memory size, memory access characteristics and port count is briefly discussed in this section. More details can be found in [14].

9.1 The memory architecture model

Most memory architectures in real-life applications in the target application domain, described in section 1, can be classified by a two level hierarchical memory system. The individual modules are categorized as follows:

- *foreground memories* are used for the short term storage of intermediate signals or signals with a large number of accesses compared to their life-times. These memories are typically considered as a part of data-paths. They are characterized by fast accessibility, i.e. they don't have a separate read and write cycle.
- *background memories* are used for long term signal storage, mainly due to use of large multi-dimensional signals. They are characterized by separate read and write cycles or multi-cycle accesses. Depending on the required access schemes different type of memories are possible.

In general, each memory has a number of *memory ports*. The type of ports can be read, write or combined read and write. With each port addresses are associated to access a specific signal instance from memory. Due to the large number of multi-dimensional data involved in the application domain, combined with the high throughput demand, the communication between memories and data-paths is of major importance to arrive at an efficient architecture. If high-level data-path mapping (HLDPM) [35] would be performed without taking into account a possible communication bottleneck, the timing constraints resulting from this mapping could give rise to the allocation of an excessive number of memory ports or large buffer spaces. In order to avoid this, we believe it is necessary to allocate first the minimal number of memory fields and memory ports to satisfy the algorithmic requirements. The latter are then used as constraints for the data-path allocation [15]. The best solution would be to handle both issues simultaneously, but we believe that this is impossible given the complexity of this task for realistic applications. Therefore, we have chosen to split up the tasks into HLMM and HLDPM. Practical experiments indicate that this leads to acceptable data paths, while the memory cost can be reduced significantly. Therefore, the HLMM task is the first task in the Cathedral synthesis scripts.

The architectural model for the background memories consist of a number of distributed (dedicated) bulk memories of different types (pointer based or random addressed, single or multi port, static or dynamic) which communicate via (dedicated) ports and interconnections with the data-paths. Note that after HLMM the final number of interconnections is not solely related to the number of ports but also to the data-path composition, which is unknown at this stage. However, the necessary bandwidth is fixed by the HLMM task. The architectural model as defined for the HLMM task is depicted in figure 19. The architectural model for the address generation includes dedicated address logic, whenever it can be motivated, which in general can be possibly shared by a

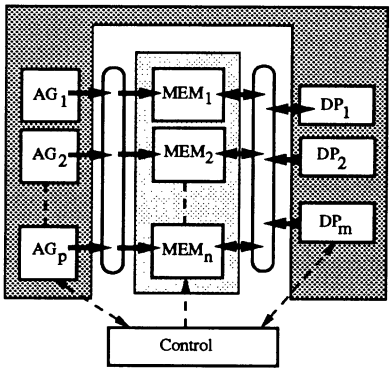


Figure 19: The memory architectural model.

number of memory ports. However, (large) part of the address generation will be performed based on arithmetic operations which can be multiplexed on the data-paths together with the rest of the signal flow operations.

9.2 Control flow transformations for HLMM

The proposed data flow modeling allows for the analysis of groups of signal instances instead of individual ones. However, multi-dimensional signals are not grouped beforehand in the initial data flow. In fact, the final grouping of individual signal instances will be a result of the optimization process to be performed on the data flow. A first complexity reducing step in HLMM is an initial pruning of the polyhedral dependency graph. All intermediate signals, which are sure to be consumed directly after their production, do not have to be stored in background memory. The corresponding polytopes and dependencies can be pruned from the initial graph. The pruned graph is used in the remaining optimization steps.

The solution space for HLMM is a multi-dimensional common node space, along with linear ordering vectors.

Creation of the common node space requires the placement of individual node spaces into a single common node space. Given the common node space, various linear orderings form the solution space. The total cost related to storage is dependent on *all* of the following 3 elements.

Memory size: Memory size is related to the maximum number of signal instances to be stored at any point in time for a given ordering of computations. Both the placement of computations of signal instances in an algorithmic space as well as the choice of linear ordering function have an influence on the memory size.

Due to the choice a certain ordering vector the number of intermediate signals which are alive at one time point might be strongly reduced. As a result it can be decided to use a type of foreground memory to store these signals. This has already been illustrated in section 6.4, for example by showing the effect of loop merging on the number of signal instances to be stored (example 4).

Optimization of the internal organization of individual memory modules is another task. Different groups of signals with their access scheme have to be allocated to memory locations in such a way that the total number of memory words is minimized.

Distribution of memory accesses: The distribution of accesses over the available cycles is an optimization task in memory management. The distribution of memory accesses is related to the distribution of dependencies in the algorithmic space after placement of computations. An uneven distribution of dependencies in combination with a linear ordering function will give rise to an uneven distribution of memory accesses and therefore inefficient utilization of allocated memory ports. For a given ordering vector, the relative ordering of accesses is fixed for sets of signals. However, the exact coupling of which signal is assigned to which memory port still has to be determined. This is called port assignment. In principle the number of ports should be high enough to satisfy the bandwidth constraint given as the maximal number of simultaneous access operations at a single time point.

Complexity of address calculations: Address calculation complexity is influenced by the uniformity of the dependency distribution after placement of computations. Regularization of the data flow is a well known topic in array synthesis. Regarding multiplexed architectural styles, compared to array synthesis there is not a constraint on the mapping of space-time dependencies into a regular structure of physical interconnections, i.e. busses and registers. However, there is much similarity with optimizing the regularity of space-time dependencies, because this regularity does allow a highly repetitive access pattern of signal instances.

The effectiveness of this type of regularization has already been reported in [60] where it is illustrated on a Levinson-Durbin algorithm. The reason for this optimization is that the regularity within these signal instance clusters is exploited, leading to a small number of different indexing patterns in the complete n-dimensional common index space.
A small number of different indexing patterns is desirable for the following reasons.

- Different indexing patterns will lead to increasing control flow complexity,
- A large variety of index patterns reduces the compatibility of the data flow operations related to indexing. This compatibility is desired for multiplexing these parts of the data flow on a minimal amount of hardware.

From the above, it can already be concluded that the proposed model can be used to derive all aspects of storage related cost. Note that expressing a partial ordering by means of a linear ordering vector may in some cases put constraints on the solution space whenever directly applied to polytopes extracted from the initial data flow specification. Hence, the straightforward use of polytopes combined with the linear ordering already excludes some orderings. This is not always desirable and it may lower the efficiency of the derived memory organization. Therefore, these constrains should be detected and alleviated. This can be done by reformation of the polytopes [14].

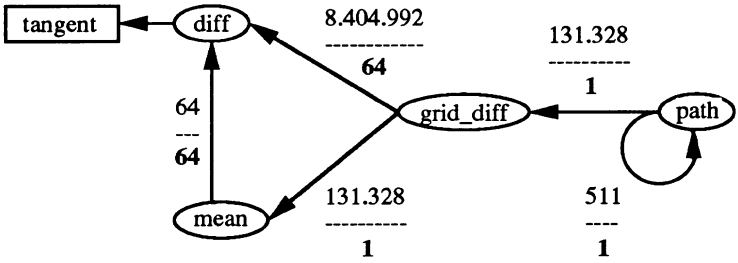


Figure 20: Simplification of the CRD algorithm with node space to node space signal dependencies.

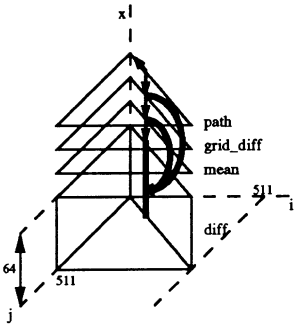
9.3 Example of optimizing control flow for HLMM

In this section, the theory of control flow modeling and transformation will be applied in the design of a memory management scheme for a simplified Contour Regularity Detector (CRD) algorithm [27]. This algorithm is used in a robot vision application where robust contour tracing has to be performed on complex images corrupted by the presence of noisy [56].

Example 9

```
func main(tangent : word[4096])path : bool[][] =
begin
  (i : 0..511) ::
  (j : i..511) ::
  begin
    (k : 0..63) :: diff[i][j][k] = f1(tangent[], i, j, k);
    mean[i][j] =f2(diff[i][j][]);
  end;
  (i : 0..511) ::
  (j : i..511) ::
    grid_diff[i][j] = f3(diff[i][j][], mean[i][j]);
  (i : 0..511) ::
  (j : i..511) ::
    path[i][j] = f4(path[i − 1][j − 1], grid_diff[i][j]);
end;
```

The input of the algorithm is a multi-dimensional signal *tangent* with 4096 signal instances. This large signal is stored in memory external to the system implementing the algorithm. The task of HLMM is to optimize the memory organization resulting from signal creation within the algorithm and therefore within the system implementing the algorithm. To understand the dependencies between signals expressed by this abstract algorithm, consider the following line of code extracted from it: $grid\_diff[i][j] = f3(diff[i][j][], mean[i][j])$. This line indicates that signal instances $grid\_diff[i][j]$ depend for their creation on signal instances $mean[i][j]$ and $diff[i][j][]$, using some function $f3()$. The empty brackets in $diff[i][j][]$ denote *all* 64 signal instances of $diff$ for a given i, j. This means that each signal instance $grid\_diff[i][j]$ depends on a




```
for p=0 to 511
do
 for q=0 to 511-p
 do
  for k=0 to 63
  do
   diff[q][p+q][k] = f1(tangent[],q,p+q,k);
  od
  mean[q][p+q] = f2(diff[q][p+q][]);
  grid_diff[q][p+q] =
      f3(diff[q][p+q][],mean[q][p+q]);
  path[q][p+q] =
      f4(path[q-1][p+q-1],grid_diff[q][p+q]);
 od;
od;
```

Figure 21: Polytope placement with the control-flow-equivalent procedural code for ordering vector $\Pi = (-34237\ 34304\ 1)$ in the i, j, x space.

total of 65 other signal instances.
A direct procedural interpretation of the code of example 9 leads to memory requirements of over 8 Mega signal instances, which equals for this application over 40 Mbit of memory. These memory requirements can be drastically reduced by optimizing the control flow of the algorithm. This optimization task can be expressed in terms of an optimal placement of the polytopes involved and an optimal selection of the linear ordering vector, using the proposed model. Extraction of the polytopes and their dependencies according to the presented model, leads to the inter-node-space dependencies as indicated by the arcs in the graph of figure 20. How to select a placement is discussed in [53, 54]. The result of the placement, which is in fact a series of affine polyhedral transformations, can be found in figure 21. In this figure, the polytopes, but not their placement, correspond to those extracted from the initial applicative code. Their dependencies are indicated for a single i, j-pair by arrows. The bold arrows indicate the dependencies on 64 signal instances $diff[i][j][]$. It is clear from the way the polytopes have been placed that minimizing dependency lengths is an important placement criterion. Notice the dependencies within the polytope of *path* which are the only ones which are not parallel to the x-axis.

The selection of an ordering vector is done by successively adding constraints to the ordering vector. Each constraint has the form of a linear equation, set up by a vector inner product $\Pi v = c$, in which v denotes a direction corresponding to a preferred ordering of operations. The constant c is a result of the previously given constraints and the placement of the polytopes. The optimal ordering vector for the placement depicted in figure 21, in terms of minimizing memory size requirements, is determined by 3 linearly independent equations: $\Pi(0\ 0\ 1)^T = 1$, $\Pi(1\ 1\ 0)^T = 64 + 3$ and $\Pi(0\ 1\ 0)^T = 512 * 67$. The unique solution is $\Pi = (-34237\ 34304\ 1)$. A procedural code with a control

test vehicle	signal inst.	n-dim. signal inst.	node space pol.	poly-topes	CPU time (s)
CRD	±136M	55	69	227	20
CRD-out	±68M	26	20	66	10
APP	–	2	11	44	18.6
QRJ	–	7	5	17	5.2
Hough	±78	5	5	15	1.3
ex. 9	±9M	4	4	9	<1
Lev-Dur	±0.26M	4	11	17	1

Table 1: Performance of model extraction tool.

flow equivalent to this placement and ordering vector is also given in figure 21. Notice the differences in loop structure and indexing of signals compared to the applicative code of figure 20. It is difficult to fully describe the applied polyhedral transformations in terms of classic loop transformations. It is clear that some sort of loop merging has occurred, but the re-indexing of signals in this example is not covered by any of those loop transformations. In the graph of figure 20, each arc carries two numbers. The numbers in the graph indicate memory requirements in terms of the number of signal instances. Those above the dashed lines apply when the code of example 9 is interpreted procedurally. Those below the dashed lines correspond to the optimized control flow. Note the extraordinary reduction in total memory requirements from 8.5M to 67 signal instances.
In this example only a single task of HLMM, namely minimization of storage locations, has been discussed. Other, equally important, tasks have been mentioned in section 9.2, namely initial pruning, optimal port selection, in-place storage reduction and address calculation optimization.

10 A CAD tool

The evaluation carried out in this section has two aspects. In this section it will be demonstrated that the extraction of the information present in the model does not pose a practical problem for typical algorithms in the application domain. Information extraction can theoretically be a problem, because the removal of redundant inequalities from a polytope and the computation of extreme points have exponential complexity in the number of dimensions and faces [53].

The techniques used in the core routines of this tool are mainly based on results from polyhedral theory, as described in e.g. [34], and on conventional graph theory. CPU complexity is linear in the number of dependencies, but exponential in the dimensionality of the node spaces. However, in most real-life signal processing algorithms this dimensionality is low enough to ensure fast model extraction. Table 1 presents some performance results of the model

extraction tool, implemented in C++, for a DECstation 3100. The first test vehicle is the complete version of the (CRD) algorithm [27]. This test vehicle has been chosen to demonstrate that even for the complete algorithm the model extraction requires little CPU time. However, techniques are available to prune the original SFG, leading to a substantial complexity reduction [13]. The second test vehicle corresponds to the output of this pruning task in HLMM when performed on the complete CRD algorithm. The Algebraic Path Problem stands for a collection of algebraic algorithms [3]. QRJ is an algorithm for QR decomposition by Givens orthogonalization ([17], algorithm 6.3-1, pp. 156). See [49] for details on the Hough transform. The core of the Levinson-Durbin algorithm [59] is the last test vehicle.

The table shows that more polytopes lead to a longer execution time for model extraction, as would be expected. However, the structure of the algorithm also influences the total execution time during the construction of the affine polyhedral dependency graph at node space level. The complexity of the structure of the algorithm is related to the complexity of the graph being built. Algorithms with complex multi-dimensional dependencies between polytopes, like the APP-algorithm, require more model extraction time then those with more polytopes but lower complexity, like CRD-out.

The important conclusion drawn from the table is that model extraction does not pose any complexity problems for practical examples within the proposed specifications and restrictions.

11 Conclusions

In this paper it has been shown that formalization of high level control flow transformations in architecture synthesis requires a suitable formal modeling of both data flow and control flow. The proposed model has the following properties:

- The data flow model allows control flow selection that is independent from the syntactical structure of an algorithm description.
- Control flow alternatives are expressed as a combination of operation placement in a common node space and the selection of a linear ordering function in this space.
- It exhibits amenity to a more formal and general approach to control flow optimization than conventional syntactical structure transformations.

The proposed model has strong links with the polyhedral models used in many array architecture synthesis methods [12, 41, 33]. The main features of the model are the mathematical descriptions of both individual dependencies between operations and signals as well as the structure of these dependencies. These features allow for efficacious steering methods [53, 54] for control flow transformations, which can outperform transformation methods based on alternative models, such as the SFG or stream model, in the given target domain.

In this way, relatively well known control flow transformations, like loop transformations [31, 57, 59], can be easily generalized. Furthermore, important characteristics of a specified control flow, like those of signal lifes, can be extracted in a systematic way. This has been exemplified by applying the model in the context of high level memory management for time multiplexed architectures targeted towards real-time DSP applications. It has been indicated how three important memory costs (storage size, access distribution, addressing complexity) can be expressed, and therefore optimized, in terms of the proposed model. This also applies to other optimization criteria in architecture synthesis, e.g. control complexity, which can be modeled and partially optimized at a high level. Apart from high level memory management, the model has been applied to regular array architecture synthesis [50, 51, 53].

Without extending the basic model, the exact data flow dependencies for algorithms that cannot be written in the form of Conditional Affine Recurrence Equations (CARE's) can only be approximated. Although CARE's cover a large percentage of algorithms in the envisaged application domain, extensions to the model are needed, especially for high level memory management for time multiplexed architectures. These extensions have been briefly outlined in this paper. A more detailed description can be found in [14].

Performance figures of a CAD tool implementing the model extraction demonstrates the feasibility of this approach for the envisaged application domain.

References

[1] S.G. Abraham, D.E. Hudak. "Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic", *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 318-328, 1991.

[2] J. Annevelink, P. Dewilde. "Hifi: A functional design system for VLSI processing arrays", *IEEE International Conference on Systolic Arrays*, pp. 413-452, 1988.

[3] A. Benaini, P. Quinton, Y. Robert, Y. Saouter, B. Tourancheau. "Synthesis of a new systolic architecture for the algebraic path problem", *Science of Computer Programming*, 15, pp.135-158, 1990.

[4] P. Bertolazzi, C. Guerra, S. Salza. "A systematic approach to the design of modular systolic arrays", *IEEE International Conference on Systolic Arrays*, pp. 453-463, 1988.

[5] S.H. Bokhari. "Assignment problems in parallel and distributed computing", Kluwer Academic Publishers, Norwell, Massachusetts, 1987.

[6] J. Bu. "Systematic design of regular VLSI processor arrays", Ph.D. dissertation, Delft University of Technology, Dept. of E. E., May 1990.

[7] P.R. Cappello. "Space time transformation of cellular algorithms", *Systolic Signal Processing Systems*, E.E. Swartzlander editor, Dekker inc, New York, 1987, pp. 161-207.

[8] A.E. Casavant et al. "A synthesis environment for designing DSP systems", *IEEE Design and Test*, pp. 35-44, April 1989.

[9] F. Catthoor, H. De Man. "Application-specific architectural methodologies for high-throughput digital signal and image processing", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol.37, no.2, pp.176-192, February 1990.

[10] F. Catthoor, M. van Swaaij, J. Rosseel, H. De Man. "Array design methodologies for real-time signal processing in the Cathedral IV synthesis environment", *Algorithms and Parallel VLSI Architectures II*, P. Quinton et al. editors, Elsevier Science Publ., 1992., pp.211-221.

[11] H. De Man, F. Catthoor, G. Goossens, J. Van Meerbergen, J. Rabaey, J. Huisken. "Architecture-driven synthesis techniques for mapping digital signal processing algorithms into silicon", *special issue on comp. -aided design of Proc. of the IEEE*, vol.78, no.2, pp.319-335, Feb. 1990.

[12] V. van Dongen, P. Quinton. "Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic arrays", *IEEE International Conference on Systolic Arrays*, pp. 473-482, 1988.

[13] F.H.M. Franssen. "Algorithmic and architectural study of a character outline rasterization application", *European Community Basic Research Action 3280 (NANA)*, report IMEC/y2m12/2.1/4, 1991.

[14] F.H.M. Franssen. "High level memory management in the Cathedral III synthesis environment", *European Community Basic Research Action 3280 (NANA)*, report IMEC/y3m6/2.1/6, 1992.

[15] W. Geurts, F. Catthoor, H. De Man. "Time constrained allocation and assignment techniques for high throughput signal processing", *ACM/IEEE Design Automation Conference*, 29th, June 1992.

[16] E.F. Girczyc. "Loop winding – a data flow approach to functional pipelining", *IEEE International Symposium on Circuits and Systems*, pp. 382-385, 1987.

[17] G.H. Golub, C.F. Van Loan. "Matrix computations", John Hopkins University Press, Baltimore MA, 1989.

[18] G. Goossens, J. Vandewalle, H. De Man. "Loop optimization in register-transfer scheduling for DSP-systems", *ACM/IEEE Design Automation Conference*, 26th, pp. 826-831, 1989.

[19] D. Grant, P.B. Denyer, I. Finlay. "Synthesis of Address Generators", *IEEE International Conference on Computer-Aided Design*, pp. 116-119, Nov. 1989.

[20] D. Grant, P.B. Denyer. "Address Generation for Array Access, Based on Modulus M Counters", *European Design Automation Conference*, , Feb. 1991.

[21] P. Havlak, K. Kennedy. "An implementation of interprocedural bounded regular section analysis", *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350-360, 1991.

[22] P. Hilfinger, J. Rabaey, D. Genin, C. Scheers, H. De Man. "DSP Specification using the SILAGE Language", *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp.1057-1060, 1990.

[23] S.Y. Kung. "VLSI array processors", Prentice Hall, Englewood Cliffs, N.J., 1988.

[24] D.Lanneer, F.Catthoor, G.Goossens, M.Pauwels, J.Van Meerbergen, H.De Man. "Open-ended System for High-Level Synthesis of Flexible Signal Processors", *European Design Automation Conference*, pp. 272-276, March 1990.

[25] S.R. Lay. "Convex sets and their applications", John Wiley & Sons, New York, N.Y., 1982.

[26] J-H. Lee, Y-C. Hsu, Y-L. Lin. "A new ILP formulation for the scheduling problem in data-path synthesis", *IEEE International Conference on Computer-Aided Design*, pp.20-23, 1989.

[27] C.Y. Lee, F. Catthoor, H. De Man, "Real-Time Regularity Detection for Robot Vision Using a Customized Architectural Approach", *IEEE International Symposium on Circuits and Systems*, New Orleans, May 1990.

[28] P.S. Lewis, S.Y. Kung. "Dependence graph based design of systolic arrays for the Algebraic Path Problem", *Proc. 12^{th} Asilomar Conference on Signals, Systems and Comput.*, nov 1987., pp. 13-18.

[29] G-J. Li, B.W. Wah. "The design of optimal systolic arrays", *IEEE Transactions on Computers*, pp. 66-77, 1985.

[30] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney. "Memory Synthesis for High Speed DSP Applications", Proc. CICC, San Diego, May 1991, pp 11.7.1-4.

[31] P. Lippens, J. van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, O. McArdle. "PHIDEO: a silicon compiler for high speed algorithms", *European Design Automation Conference*, pp.436-441, 1991.

[32] T.H. Matheiss, D.S. Rubin. "A survey and comparison of methods for finding all vertices of convex polyhedral sets", *Mathematics of Operations Research*, vol. 5, no. 2, pp. 167-185, 1980.

[33] D.I. Moldovan. "Advis: a software package for the design of systolic arrays", *IEEE International Conference on Computer Design*, pp. 158-164, 1984.

[34] G.L. Nemhauser, L.A. Wolsey. "Integer and Combinatorial Optimization", John Wiley & Sons, New York, N.Y., 1988.

[35] S. Note, W. Geurts, F. Catthoor, H. De Man. "Cathedral III : architecture driven high-level synthesis for high throughput DSP applications", *ACM/IEEE Design Automation Conference*, 28th, June 1991.

[36] D.A. Padua, M.J. Wolfe. "Advanced compiler optimizations for supercomputers", *Communications of the ACM* , vol. 29, no. 12, pp. 1184-1201, 1986.

[37] P.G. Paulin, J.P. Knight, E. Girczyc. "HAL: a multi-paradigm approach to automatic data path synthesis", *ACM/IEEE Design Automation Conference*, 23rd, pp.263-270, 1986.

[38] J-K. Peir, R. Cytron. "Minimum distance: a method for partitioning recurrences for multiprocessors", *IEEE Transactions on Computers*, vol. 38, no. 8, pp. 1203-1211, 1989.

[39] C.D. Polychronopoulos. "Compiler optimizations for enhancing parallelism and their impact on architecture design", *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 991-1004, 1988.

[40] W. Pugh, D. Wonnacott. "Eliminating false data dependencies using the Omega test", to appear at *SIGPLAN PLDI*, 1992.

[41] P. Quinton, V. van Dongen. "The mapping of linear recurrence equations on regular arrays", *Journal of VLSI Signal Processing*, vol. 1, pp. 95-113, 1989.

[42] S.V. Rajopadhye, R.M. Fujimoto. "Systolic array design by static analysis of program dependencies", *Parallel Architectures and Languages Europe*, J. de Bakker, A.J. Nyman, and P.C. Treleaven editors, Springer-Verlag, 1987, pp. 295-310.

[43] S.K. Rao, T. Kailath. "Architecture design for regular iterative algorithms", *Systolic Signal Processing Systems*, E.E. Swartzlander editor, Dekker inc, New York, 1987, pp. 209-297.

[44] Y. Robert, D. Trystram. "Systolic solution of the Algebraic Path Problem", *Systolic arrays*, ed. by W. Moore et al., Adam Hilger, Bristol, 1987., pp. 171-180

[45] J. Rosseel, F. Catthoor, H. De Man. "Extensions to linear mapping on regular arrays with complex processing elements", *IEEE International Conference on Application Specific Array Processors*, pp. 156-167, 1990.

[46] A. Schrijver. "Theory of linear and integer programming", John Wiley & Sons, New York, N.Y., 1986.

[47] W. Shang, J.A.B. Fortes. "On the optimality of linear schedules", *Journal of VLSI Signal Processing*, no. 1, pp. 209-220, 1989.

[48] J-P. Sheu, C-Y. Chang. "Synthesizing nested loop algorithms using nonlinear transformation method", *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 304-317, 1991.

[49] M.F.X.B. van Swaaij, F.V.M. Catthoor, H.J. De Man. "Deriving ASIC Architectures for the Hough Transform", *Parallel Computing*, no. 16, pp. 113-121, 1990.

[50] M.F.X.B. van Swaaij, J. Rosseel, F.V.M. Catthoor, H.J. De Man. "Synthesis of ASIC Regular Arrays for real-time image processing systems", *Journal of VLSI Signal Processing*, Special issue on CAD, 3, pp. 183-192, 1991.

[51] M.F.X.B. van Swaaij, F.V.M. Catthoor, H.J. De Man. "Signal analysis and signal transformations for ASIC regular array architecture synthesis", *Algorithms and Parallel VLSI Architectures II*, P. Quinton et al. editors, Elsevier Science Publ., 1992., pp. 223-229.

[52] M.F.X.B. van Swaaij, F.H.M. Franssen, F.V.M. Catthoor, H.J. De Man. "Modeling data flow and control flow for high level memory management", *European Design Automation Conference*, pp. 8-13,1992.

[53] M.F.X.B. van Swaaij. "Data flow geometry: exploiting regularity in system-level synthesis", internal IMEC report, 1992.

[54] M.F.X.B. van Swaaij, F.H.M. Franssen, F.V.M. Catthoor, H.J. De Man. "Automating high level control flow transformations for DSP memory management", To be published in the proceedings of *IEEE Workshop on VLSI signal processing*, 1992.

[55] N. Tawbi. "Parallelisation automatique: estimation des durees d'execution et allocation statique de processeur", These de doctorat, Universite Paris VI, Institut Blaise Pascal, Laboratoire MASI, Septembre 1991.

[56] L. Van Gool, J. Vagenmans, A. Oosterlinck. "Regularity detection as a strategy in object modelling and recognition", *SPIE Applications of Artificial Intelligence*, Vol. 1095, pp. 138-149, 1989.

[57] J. Vanhoof, I. Bolsens, H. De Man. "Compiling multi-dimensional data streams into distributed DSP ASIC memory", *IEEE International Conference on Computer-Aided Design*, pp. 272-275, 1991.

[58] I. Verbauwhede, F. Catthoor, J. Vandewalle, H. De Man. "Background Memory Synthesis for Algebraic Algorithms on Multi-Processor DSP Chips", *IFIP international Conference on VLSI*, pp. 209-218, 1989.

[59] I. Verbauwhede, F. Catthoor, J. Vandewalle, H. De Man. "In-place memory management of Algebraic Algorithms on Application Specific processors", *Algorithms and Parallel VLSI Architectures*, E. Deprettere et al. editors, Elsevier Science Publ., 1991., vol. B, pp. 353-362.

[60] I. Verbauwhede. "VLSI design methodologies for application specific cryptographic and algebraic systems", *Ph.D thesis, Katholieke Universiteit Leuven, Faculteit Toegepaste Wetenschappen*, July, 1991.

[61] M.E. Wolf, M.S. Lam. "A loop transformation theory and an algorithm to maximize parallelism", *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452-471, 1991.

[62] Y. Wong, J-M. Delosme. "Optimal systolic implementations of N-dimensional recurrences", *IEEE International Conference on Computer Design*, pp. 618-621, 1985.

[63] Y. Yaacoby, P.R. Cappello. "Scheduling a system of affine recurrence equations onto a systolic array", *IEEE International Conference on Systolic Arrays*, pp. 373-382, 1988.

[64] X. Zhong, I. Wong, S.V. Rajopadhye. "Bounds on the number of linear allocation functions", *VLSI signal processing IV*, H. Moscovitz et al. editors, IEEE press, 1990., pp. 85-94.

8

AUTOMATIC SYNTHESIS OF VISION AUTOMATA

**Bertrand ZAVIDOVIQUE, Christian FORTUNEL,
Georges QUENOT, Abdelhakim SAFIR,
Jocelyn SEROT, François VERDIER**

A - INTRODUCTION

We are very thankfull to Marc ECCHER, Etienne ALLARD, Thierry BOMMARD and Serge DACIC whose ideas were used in this work and who contributed greatly during the early phases of this research. We are also very thankfull to the other dozen of researchers that, over the years, built the HECATE emulator.

A.1 - PROBLEM OVERVIEW

The Système de Perception (SP) research laboratory of the Etablissement Technique Central de l'Armement (ETCA) embarked, several years ago, on a vast project to develop an environment that would perform the automatic synthesis of vision automata.

Designing highly integrated (VLSI) vision automata may be considered a major challenge for the future due to the increasing complexity of integrated systems and the necessity to integrate more and more functionalities in systems, while minimizing the design effort that is repetitive and time consuming (thus expensive). There are a certain number of generic problems, described below, that are associated with the automatic synthesis of such automata.

Increased functional diversity

The need to build systems that are robust and capable of autonomous operation requires that one multiplies the number of sensors installed on a robot (for example, two cameras for stereoscopic vision), one diversifies the nature of sensors (for example acoustic and optical sensors) and one specializes the functionalities of each sensor to the mission at hand. Such approach fits within the paradigm of Active Perception that relies on the definition of mechanisms to actively control perception parameters such as the field of view angle, the focusing distance or the viewing orientation, and addresses the generic problem of focusing attention in space and time.

This demand for increased performance requirements in tomorrow's robotic systems, and the specification of more and more stringent constraints for the realization of target systems that forces solutions to be machine dependent, lead to a diversification of image processing (and more generally perception) functions that goes beyond the traditional design cycle and the current

integration technologies, both ill-adapted to satisfy these new needs. Several factors are responsible for this inadequacy.

Algorithmic rigidity

The various topological structures of associating hardware resources (vector, grid, systolic, pyramid, hypercube, etc.) and the resources themselves (shared memory, pipelined programmable processors, interconnection network blocks [Saf87]) all lead, in their architectural association, to different, often mutually incompatible, algorithmic solutions. In fact, given an algorithm, it is very difficult to extract the topological regularity and the processing concurrency needed to fit a particular computing topology, without requiring a reformulation of the procedure [Anders65,Berns66,Kuck77]. This is particularly true if one considers that data structures used for low-level processing (close to the pictures) have nothing in common with generated attribute lists used by the decision making process that follows the recognition phase. In the end, one needs to rewrite the solution whenever one changes the granularity (processor size), the topology or any other parameter of the architecture.

Lack of a description formalism

At the algorithmic level (specification), the lack of a description formalism for Image Processing (I.P.) applications (proof of some underlying immaturity in the field) renders the conception and validation of algorithmic solutions extremely hard. This is accentuated by the ever increasing complexity of tasks that are demanded of vision processes (proliferation of sophisticated sensors, elaborate ways to process their data by taking into account noise and uncertainty, and new techniques that associate heterogeneous (sonar / visual) sensors). This lack of a description formalism results from the fact that Image Processing applications remain problem oriented and machine dependent. This leads to artificial partitions of designing tasks, a priori decompositions of algorithms and arbitrary allocations of hardware resources. These heuristic methods of decomposing and mapping algorithms onto an architecture produce resource allocations that are, at best, sub-optimal; the new complexity of algorithms make them totally inefficient. So, to overcome the difficulties araised by the future multiple designs of tailored functions, we need a single formalism to describe both algorithms and specific architectures. This is our so-called Emulation-Synthesis approach for generating automatically vision automata.

Difficulty in validating complete automata

At the system level (definition), a major part (approximately 50% [Curtis88]) of all application-specific integrated circuits (ASIC) that are designed and verified in isolation do not work properly when placed inside their host environment (we have not seen yet any objective indications that the situation is changing). This is mainly due to the difficulty in simulating, fully and accurately, the complete system. This difficulty arises from the fact that simulation models for the various parts of the system are non-existent or incompatible (for example, functional behaviors of complex sub-systems are

often incomplete). Also, some modules (black box type) only provide a functional specification, whereas some others (library components) permit simulation down to the signal level.

Complexity of synthesis processes

On an implementation level (fabricating), current approaches to the architectural synthesis problem tend to separate its various phases (such as data-path synthesis or floor plan layout) into independent processes with their own tools and techniques or limit the architectural solution to a predefined type. This prevents achievement of optimal mapping solutions by not considering the problem in its globality. It is clear that this independence between the tools cannot subsist if efficient mappings that satisfy multiple constraints are desired.

Higher integration needs

Current trends in the area of robot perception that go in the direction of smart sensors, will require the integration of complete vision automata into very small dimensions that go beyond the reach of current VLSI technologies. The inexistence of tools sophisticated enough to accommodate these constraints (volume, low power consumption, heat dissipation), will necessitate the design or redesign of specific algorithmic solutions. The economic impact of such extraneous work (already happening) creates a need for conceiving new tools that systematically and automatically map algorithmic solutions onto the hardware, given some architectural or technological constraints.

Need for a global and homogenous solution

So a global approach to the automatic design of VLSI architectures, that takes into account the complete algorithmic solution and the dependency between the various phases of the compilation process, becomes more and more necessary. Solutions are required that can automatically map algorithms (specified by their functional behavior on incoming data flows) onto target architectures (specified by their topological structures for transforming and communicating data flows).

A.2 - PREVIOUS ATTEMPTS

Some attempts have been made to avoid the algorithmic rigidity of specialized hardware. Languages can provide software constructs or models that allow problem specific parallelism or pipelining. For example, languages such as OCCAM feature the ability to handle parallelism, especially when they support the micromachine of general processors (transputers); they could be solutions for a better emulation of algorithms, but their semantics are still too restricted.

Conversely if we are trying to avoid regular predetermined structures, the state-of-the-art silicon compiler allows relative flexibility in the definition of structures. For example some work at Berkeley [Pope84] has focused on a simple standard processing element with specifiable hardware features such as

word size and memory capacity. Only the topological association of such elements and a small part of the sequencing is specific to the application.

A solution restricted to the sequencing problem has been studied at the Georgia Institute of Technology that is based on the data-flow paradigm. Since 1986 this paradigm has also been under trial at Stanford and Berkeley.

But despite all these attempts, the flexibility of silicon compilers for material structure is still very limited: the systematic transformation of applications is formulated through conventional programming languages [Back78] and the weakness of their expressive power has contributed to the restriction of design models to a certain class of target architectures. Typical examples are Syco [Jerr86] or MacPitts [South83] which have microprocessor target architectures.

A.3 - PROPOSED SOLUTION

Description of our solution

To overcome the difficulty created by the future multiple designs of tailored vision functions, we have attempted a so-called "functional approach" which aims at describing both algorithms and special architectures through the same formalism. This study lead to two different realizations:

- The building of an image processing emulator (HECATE - MIMD school of specialized image processing modules) to support most currently known algorithms on the fly.
- The design of a specialized highly parallel architecture (DATA-FLOW FUNCTIONAL COMPUTER, DFFC), based on a highly connected network of a custom processor, handling the execution of Real Time Image Processing Algorithms expressed as Directed Data-Flow Graphs (DAG).

The methodology presented here, conceived to rapidly and efficiently design embedded vision systems that satisfy stringent constraints, relies on the decomposition of a given algorithm into a set of functional primitives, and the emulation of the image processing algorithm prior to its automatic integration into hardware parts. The major interest of this so-called functional approach is that it aims at describing both algorithms and special architectures through a unique formalism and permits the automatic transcription of Image Processing algorithms into specialized automatons. This principle is essential: one description can be used equivalently to specify algorithms or the machines executing them. This, in essence, has the effect of fusing the phases of high level image processing algorithm specification, design and hardware implementation.

This approach can be viewed as a three-step process that is clustered along a so-called "mother-architecture" hardware emulator (either Hecate or the DATA-FLOW FUNCTIONAL COMPUTER):

- The **emulation phase** that provides a functional representation of an image processing algorithm. It consists of designing the specific algorithm and controlling its emulation onto the hardware to validate

its functional behavior. This phase can be divided into two steps iteratively executed:

- The **functional description** of an application algorithm.
- The **hardware emulation** of the algorithm onto the emulator. The programmer modifies at this time the algorithm until it performs as needed.

- The **diagnostic phase** that extracts and translates emulation resources needed while running the algorithm (a solution to the application problem) into a final vision automaton. This phase generates a data-flow representation at the register-transfer level (RTL).
- The **synthesis phase** that integrates any physical constraints necessary to the satisfaction of the requirements specified by the particular application and provides a near optimal solution (if one exists) in terms of a list of interconnected constructible units (definition of VLSI chip set masks). This phase, that in effect reduces the automaton generated by the diagnostic phase into a hardware machine, can be subdivided into many interdependent steps. These steps solve the problems of data-path synthesis, control synthesis, floor-plan layout and multi-chip partitioning.

The **functional description** uses the functional approach to specify algorithms in the form of graphs of primitive operators. The interest of this representation is that it is the unique representation employed by the system, whether we want to describe what the system does functionally (specification) or we want to build a solution (definition).

The **hardware emulation** consists of executing the specified algorithm in real time (or close to real time if resources need to be time-shared) and, in an iterative manner, modify an algorithm until it performs as one desires. Let us note that the satisfaction of the user results from a series of adjustments where parameters are modified, and one sees their effects, in real time, on the images being analyzed. This approach truly permits non-specialists to use the full power of highly complex emulators.

The **diagnostic** stage is a process that partly executes during the emulation by "spying" on resources (data flows, operators) to determine which ones are used and when. It consists in collecting traces of all hardware and software activities, during the emulation of a given algorithm. Such traces answer two purposes: optimizing on-line the algorithm implementation, and providing basic information in view of straight integration from this very implementation (exact copy of traces). For example, if a complex ALU is used to do an addition, it will record the fact that an addition was performed. Once the emulation is completed, it generates a directed graph that represents a photographic memory of the execution of the algorithm on the emulator. This graph is then transformed in a way that is better adapted to the global optimization integration steps that follow.

The previous steps (functional description, hardware emulation and diagnostic) rely heavily on an **intuitive programming environment** that allows non-expert users to specify algorithms, parameterize functions and to control the emulation through modifications of the algorithm. In the case of the Data-

Flow Functional Computer emulator, this environment has become an integral part of the machine and is omnipresent.

The integration phase performs the scheduling and allocation of operations (specified by the graph) onto available operators. This phase is the most important, and from a computing resource point of view, the most time-consuming, as the quality of the solution (the way the constraints are satisfied) depends on its efficiency. The originality of the approach resides in the fact that the problem is partitioned (this is necessary to reduce the complexity of finding a solution in the design space that meets all constraints) in two independent sub-problems: **scheduling** and **allocation**. Scheduling determines which operations should be performed at each clock cycle. Allocation assigns operations to available hardware resources. This allows to take into account low level constraints of realization (such as placement area, routing length or complexity, performance, thermal dissipation or grounding path) during the entire process of searching for a global optimum.

Strengths of our solution

The strength and originality of our solution resides in the following aspects:

- At the algorithmic level, the automatic design of specific image processing automata is done from emulation results in the form of data flow graphs that are defined in terms of a "general purpose" machine for Image Processing or so-called "mother-machine".
- At the system implementation level, the tasks of operation scheduling and allocation involved in the synthesis process, provide a globally optimized solution of the data path and the controller by considering low level realization constraints of all sorts.

Thus, by combining the conception of an algorithm with its emulation and avoiding complex simulations, development time bottlenecks and formal verification, the proposed method generates truly integrated solutions.

This functional approach, a priori less dependent on a particular language, except that it is constrained by the application domain (Image Processing), uses the definition of macro-cells that present the user with many facets. It is, in some way, similar to traditional VLSI design techniques and attempts to reduce into one the different phases of high level image processing algorithm specification, design and hardware implementation.

We partially avoid the predefinition of any regular architecture or any type of fixed data path by taking advantage of an interesting property of Image Processing applications: they can be structured according to image features that need to be extracted (edges, regions, points of interest, velocity fields, colours, etc.) or according to decision-making methods that are used (thresholding, statistics classifications, Bayesian techniques, structural methods, tree parsing with heuristics, etc.) [Serf85,Gall86], which are representative of favoured data movements. For instance looking for "regions" into an image leads to grouping of pixels; this class of operator induces hierarchically organized communications between neighbouring objects.

These variables (pixels, edges, regions, etc.) can be associated through operators that favour specific data structures and flows (list, tree, pyramid, etc.) and find efficient implementations in hardware. Above all we focus on the functional properties of data flow models and delay as much as possible taking into account the parameters of the physical layout that will instantiate the architecture.

Weakness of our solution

This approach has a major drawback: we need to build a (reconfigurable) machine that achieves sufficient computing power to allow for the emulation of complex algorithms, at least for the given class of applications that are of interest. Let us mention that if in our case, it is Image Processing, it could as well be networking (for emulating different protocols) or information retrieval (for emulating various update mechanisms). This point was addressed by the successful completion of two data flow emulators (Hecate and the Data-Flow Functional Computer). It turns out that low level Image Processing algorithms fit exactly on data-flow architectures because their intrinsic nature is functional.

A.4 - CHAPTER ORGANIZATION

The chapter will follow (section B) with an analysis of the requirements that need to be satisfied in order to allow for an efficient designing of algorithms. Then we present the concepts of a functional description of algorithms (section C) that provide our solution its power and efficiency. Then we follow with a description of the concepts of emulation and rapid prototyping (section D), where as a related discussion about the main goal of hardware realization and software implementations for this project, we will evaluate challenges facing emulation methodologies. The description of the two emulation systems we have built (Hecate in section E and the DATA-FLOW FUNCTIONAL COMPUTER in section F) will then be done, before presenting the programming environment suggested by an efficient emulation and diagnostic (section G). We will show how studies about the emulator control and about concepts in domains of emulation, simulation test and design, have led to demonstrate the effective power of Object Oriented Programming tools for VLSI design; in particular the fact they support variable grain complex distributed processors aiming at direct and fast VLSI circuit design. We will also state reasons for incorporating the Hardware Modelling concept as a basic component of a powerful design methodology. Finally the synthesis phase and the tools that are associated with it (section H) are detailed before concluding (section I) on the approach.

B - REQUIREMENTS FOR EFFICIENT DESIGNING

B.1 - DIFFICULTIES IN DESIGNING ARCHITECTURES

Complexity of vision automatons

The domain of machine vision (not limited to structured environments) has recently started to be investigated from a proliferating number of research

disciplines (physiology, physics, applied mathematics, etc.) but also from varied application fields (robotics, non destructive testing, quality control, etc.). This is motivated by the difficulty of interpreting images and conceiving algorithms that actually solve real problems. The inability of researchers to structure, categorize and organize the acquired knowledge has lead to the definition of a large and heterogeneous collection of techniques that often work on disparate data structures and cannot be assembled in straightforward manners. This heterogeneity and lack of formalization generate, when it comes to building a system, the creation of ad-hoc, highly complex vision automatons. Such complexity, or lack of regularity, renders very difficult the task of designing corresponding needed architectures that meet the processing requirements.

Need for specialized architectures

The tremendous amount of data (0.75 MegaByte for a 512x512 RVB image) that need to be processed at high frequencies (most often 30 Hertz and more) renders any classical Von Neuman's architecture inadequate to meet real time system constraints of most applications. But the large quantity of data and the locality of many operations suggest the decomposition of an algorithmic solution into multiple partial solutions that can be executed in parallel on multiple processors. This has lead to the definition of two major architectural and programming models ([Flynn72): Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) execution schemes, both based on tightly interconnected arrays of processors. These architectures, whether they are complete systems or add-ons, are generally much more complex than conventional information processors (mostly derived from the Von Neuman's computing paradigm). It is beyond the scope of the subject to detail such architectures which most often lack descriptive generality; so we refer to [Duff81], [Offen85] and [Dan81] for tentative classifications.

But trying to multiply the number of processors cannot be done without difficulty. One needs to solve technological complexities, control local and global communications schemes or determine process concurrencies. In fact, the control of parallel machines lies more in the domain of Operational Research, independently of any heuristics that may be used to take advantage of an application regularity (for example array, list, pyramid or graph organization [Berg87], [Min86], [Nagi78]). More generally, data movements tend to exhibit strong randomness, as in region growing techniques where movements are function of pixel values and cannot be predicted in advance. More problematic is the fact that corresponding data paths end up taking different shapes for each envisioned solution.

So, designing vision architectures faces many obstacles:

- There are theoretical and practical problems to controlling parallel architectures.
- State of the art in vision algorithmic is hard to delimit.
- Depending of the task at hand, abstract data types tend to vary and no consensus can be established to define a common language.

- Due to the diversity of schools, it is a major challenge to reduce vision to a set of primitive tool boxes that could be standardized and reused (linear, differential, morphological, minimization operations).

Mapping algorithms onto hardware

Recent developments in Very Large Scale Integration technologies offer many opportunities to build cost-effective and powerful tailored integrated circuits. Unfortunately, some large scale computational problems are created:

- In which way should problems be specified ?
- What kind of computational models should be used ?
- Which kinds of integrated circuits should be designed ?

On one hand, a technologically possible solution is the systematic building of special purpose computers. Right now, it is possible to design special purpose circuits for dedicated applications; thereby allowing for the creation of rather complex automaton on a single chip. For example, VLSI special-purpose chips can be obtained for regular applications such as convolution or matrix computations [Mong85]. Such an approach is worthwhile for dedicated applications, as it provides the most efficient solution that meets constraints of performance given a small silicon substrate. The problem is that no changes can be done to a given specific circuit during the lifetime of the application. On the other hand, traditional processors are available in a broad range of applications; But, as powerful they may be now (Alpha or C40 processors), they remain rather inefficient for most real time applications in image processing. Between these two extremes, many kinds of very different architectures can be conceived. One aspect is certain: the tailoring / programmability trade-off seems to remain (the more tailored an architecture is made, the less programmable it becomes).

Inadequacy of current design process

VLSI design activities remain fully subordinate to the simulation bottleneck: a circuit is not safely built unless all aspects of its functioning have been simulated. If we take into account the increased complexity of vision automatons that need to be built, this implies large development cycles. Obviously, as the amount of configurations that need to be tested increases (unfortunately in an exponential fashion), the time it takes to validate a design proportionally grows, and cannot be expected to remain small. At some point, this time will grow too long, even with the availability of new unexpectingly powerful workstations; circuit validation will necessarily demand supercomputing power. So paradoxically, as technological developments provide more and more room for integrating very complex automatons on a single VLSI, and demand from industrial and military applications is finally proliferating (triggering the release of significant funds to support them), we may witness a tapering off, possibly a slow down, in the number of ASIC designs due to the limited capabilities of current tools to design, simulate and test effectively the new complex designs that are wanted; in a word, due to the difficulties in verifying the correctness of a solution.

It is clear that the validation of a solution today is hampered by the presence of the debugging step whose role is not only to verify that the circuit works electronically as it should, but also that the circuit performs the function it should. This multiplies and mixes the problems and considerably augments the time it takes to ensure that everything is correct (if there is a problem, is it due to a failure of some electronic component or is it due to logical misconception?). It certainly would be desirable to separate, as much as possible, both aspects.

Within the application domain of interest, such as advanced signal or image processing, where the specificity of algorithmic solutions is necessary to satisfy system constraints, algorithms need to be modified quite frequently, during the design process obviously, but also during the entire lifetime of the system to provide new or enhanced functionality. This is particularly true of our application domain where algorithms are poorly characterized and may functionally fail at some latter date due to changing operative conditions. If we consider the highly competitive market of today that allows very small profit margins, it is critical to reduce the time spent designing a new product or modifying an already existing one, so as to do it as quickly as possible or make due with lower development budgets.

It is clear that with current design processes, it becomes very difficult to concurrently design and test reliable tailored algorithmic solutions that solve a particular problem, and the integrated version of their corresponding target architectures.

B.2 - EFFICIENT DESIGNING

So, when confronted with building real vision architectures, we are faced with two major problems:

- Defining an algorithmic solution to the problem.
- Constructing a specialized architecture that efficiently implements the solution.

This forces us to take two simultaneous looks at the problem:

- An image processing point of view that aims at validating an algorithmic solution.
- An integrate and test point of view to verify that the generated circuits function electronically as specified.

Rapid prototyping

By definition, the design process is iterative. The fact we are conceiving something that is new indicates that some type of search in the solution space will be necessary. But, as we have said, the life time of a product or its many instances (versions) will justify many design cycles. It then becomes critical to minimize each cycle time. What one needs to show at each iteration, is that there exists or does not exist a functional solution given the new specifications of the problem. Let us emphasize that the term "functional" carries two important meanings. The first meaning, which has lead to the definition of APPLICATIVE languages, refers to the fact that the basic

concept is function application. So, algorithms must be specified in terms of function input and outputs . The second meaning, closely related to the lack of internal states in the functional model, concerns the fact that the solution must work according to the specifications, regardless of input/output frequencies. Timing problems are always a major source of delays, especially in under real-time constraints. What we need then, is an environment that permits, what we call, "rapid prototyping" and that allows a designer to quickly evaluate a solution for a set of specifications that is not always clearly defined and tend to evolve continuously. Such an environment addresses the first point of view in that it allows validation of an algorithmic solution from the functional point of view. Having reached that step, it would be desirable to take the functional description and transform it, as directly as possible, into a physical machine (specialized architecture).

Silicon compilation

The second point of view deals with the straight integration into silicon of specific algorithmic solutions. If we look at the state of the art, attempts are mainly concerned with integrating very basic operators (convolvers, edge detectors, median filters) commonly used by many classes of vision problems (tracking, motion detection, stereoscopic vision), but solving in no way a real problem. We are far from being able to systematically integrate complete procedures (whether they need to be real time or not). The reason is that the solution must rely on the design of a specific architecture for which one can rarely extract a regularity that can be exploited throughout all facets of the architecture (data paths, controllers, layout). Unfortunately, this complexity is expected to increase as applications demand more; it must be dealt with. Although the study of architectures is expected to bring new solutions, the architectural complexities of algorithms will not be solved by considering some aspects of the solution, or a few constraints that seem important; they must all be taken into account at once. This calls for a compilation process.

On top of these architectural and integration problems, comes an even more complex problem which is the validation of the resulting architecture. Note that the validation process takes the circuit as input and determines (verifies) its functionality, whereas the design process does the opposite. The real problem here is to build a representation that can be shared by both the algorithmics and architectural (integration) domains. If such representation were available, it would permit switching from one domain to the other, and would greatly facilitate the compliance with both types of functional and electronic constraints. Finding a functionally and electronically correct solution would be rendered much more feasible.

Efficient designing

Lacking the definition of a global theory, one can summarize the previously expressed needs by stating that **the validation of any particular image processing application must rely on extensive testing through many iterations of rapid prototyping (emulation phase), before one considers a hardware implementation dedicated to a**

specific architecture by means of a compilation process capable of handling intrinsic complexities (synthesis phase).

In this way, we propose to avoid complex simulations and development time bottlenecks, by instead generating integrated solutions, without the need for some highly specialized and non-conventional architecture design tools.

C - FUNCTIONAL DESCRIPTION OF ALGORITHMS

C.1 - FUNCTIONAL DECOMPOSITION

Decomposing algorithms into functions

The actual tendency in Image Processing for designing algorithms let us perceive an unavoidable methodology organized around low level elementary functions which provide supports of elementary knowledge to higher level functions susceptible of solving problems having rich semantics. This approach encourages the separation of knowledge into multiple domains and ways to utilise them. In this context, one can look at a complex problem, such as an identification task, in terms of decomposing (analysing) it into elementary entities that can function independently or cooperatively. These entities have for mission the communication, to higher semantic levels, of elementary pieces of information on which, the identification of an object can be done, a detailed analysis of a component can be performed, or a system reconfiguration can be decided. This formalism is the one known as Functional Decomposition.

The introduced formalism preaches the reduction of a complex activity (image processing interpretation task) under the form of a cooperation between multiple primitive operations, each one dedicated to extract part of the needed information. By primitive operations, we mean for instance:

- Preprocessing and noise cleaning.
- Segmentation into regions (groups of pixels having similar characteristics) or edges (dual property of regions).
- Syntactic analysis or pattern matching that compute, for example, a similarity coefficient between segmentation primitives.
- Motion analysis that determines the most likely temporal variation of certain parameters inside an image or provides a region of interest where to point the camera.
- Relaxation process which iteratively improves a segmentation task.
- Decision process which changes the association between a group of primitive operations.

The cooperative aspect between these various primitive processes is fundamental because it is the guarantee that a successful analysis will result given a complex scene to interpret. A robust process is one where a set of primitive operations actively cooperate by exchanging working or decision parameters. For example, a movement detection process may alternate with segmentation processes to provide a robust tracking of objects that stop and go. A major aspect of this decomposition methodology is that it expresses each extraction of information as a well characterized process that possesses a

well specified functionality. The goal is to permit, at the operator level, an adaptation of the primitive operation to the environment (for instance: the contrast of the image or its dynamics), and at a higher level, an adaptation of association strategies between primitives depending on the type of analyzed images, both leading to usage rules. Following this decomposition approach, we can propose a classification of operators:

- Linear signal processing techniques: they transform a signal into another one, more significant, through filtering, convolution or projection into another space (frequency, coocurrences, shape).
- Statistical methods: they gather global characteristics associated with features of an image.
- Syntactic methods: they extract primitives (such as arcs of circle, segments, angles) and determine whether their relative arrangements can be recognized by specific grammars (in the widest sense of the word). These approaches make the foundation for recognition techniques.
- Morphological techniques: they use set theory to define every image feature as the result of an association of set operations.
- Dynamic programming, etc.

Although this classification is somewhat arbitrary, it reveals several classes of operators that can easily be chained to build up complex solutions for almost any image processing problem. If it is clear that each operator is a candidate to integration into a specialized piece of hardware, the a priori total lack of knowledge about dependencies between operators, given that each algorithm will exhibit its own dependencies, is a significant obstacle to building a generic and performant architecture.

Mapping of functions into hardware

The task of implementing an algorithmic solution onto a machine is refered to as the **mapping** of the algorithm onto the architecture. In effect, one tries to map functional resources (for instance: compute this convolution) specified by the algorithm onto available computing resources (for instance: this multiplier, this adder and this accumulator). In a generic way, the finding of a solution always consists of determining the best trade-off between space and time (if there is not enough space to lay down all the algorithm, buffer and reuse some resources and iterate).

If this task is rendered tremendously difficult, and needs to be done by an architecture specialist, it is because **algorithm descriptions and computer descriptions belong, so far, to two spaces which remain completely distinct, without any formal way to map one onto the other one**. This semantic gap makes algorithm implementation difficult, inefficient and unsafe.

Moreover, the lack of an appropriate description may prevent identification of parallel processes. To reach the specified high level computational rates, one must exploit the potential parallelism which is intrinsic in any algorithm (this is particularly true in image processing where operators are applied independently on all extracted features). Special purpose architectures always

realize parallel associations of elementary computational units. Unfortunately, this can increase tremendously the software complexity. What good is a powerful computer if no-one but its designers can program it ?

Let us note that, within our application field, conventional parallel programming languages are bound to be inefficient because they suppose either a highly regular target architecture (which is very suboptimal, given that a unique topology cannot remain efficient during all computations phases of an algorithm that starts with some signal processing and ends with symbolic analysis), or fail to specify the parallel nature of computations expressed in the form of data dependencies. Interestingly enough, a major work of the optimizing phase of a compiler is to detect data dependencies so blocks of code can be sequenced more efficiently. This is unfortunate because, at the time of conception, the designer typically knows where data dependencies exist.

So, if one simply wishes to succeed in mapping efficiently an algorithm into an architecture, one needs to take advantage of the way people conceive algorithms (functional decomposition methodology) and provide a simple programming form that captures the intentions of the programmer.

Let us describe here recent evolutions in conceiving integrated circuits that were a source of inspiration in selecting our approach. One way people create special purpose circuits is by assembling standard cells. The assembly process is controlled by a set of rules that ensure that the behavior of each component is not modified by associated cells. This is very similar to the functional decomposition approach previously described where one defines a set of primitive operators and one assembles them to create complex algorithms. One also uses data typing to verify that the associations are meaningful; for example: corresponding data types between output and input parameters. There are strong similarities between the two approaches that indicate that finding a common representation is not only possible, but would be beneficial to both domains in providing a unifying view: a set of primitive operators and their corresponding assembly rules.

C.2 - DIRECTED DATA FLOW GRAPHS: A SPECIFICATION FORMALISM

As we have seen, image processing algorithms can be described as specific combination of various basic functions. In other words, any algorithm can be described by a directed functional graph, where nodes represent selected primitive functions (operators) and links between nodes represent data dependencies (token flows). Note that we can describe equivalently hardware modules where nodes become computing resources and data flows become communication channels.

So this directed graph becomes a privileged medium for specifying an algorithm but also for permitting an efficient mapping of its functional description onto an architecture. As we will see, the DATA-FLOW FUNCTIONAL COMPUTER emulator is a data flow architecture that permits efficient mapping of data flow graphs. It is this kind of equivalency

that we seek to exploit: matching the semantics of algorithms with the semantics of the computer behavior.

A major advantage of the graph representation is its recursive nature: each operator specified in a graph can itself be another graph defined somewhere else. This permits a hierarchical representation of an algorithm which lets the programmer focus his attention to the level of details that is of interest to him. But more importantly as far as we are concerned, it allows the specification of operators to very fine levels of detail, thereby reaching the hardware level if necessary. This suggests to use a multi-facet approach where the programmer will manipulate independent multiple views of the same operator, depending of its motive. This will be detailed later in the programming interface section.

C.3 - FUNCTIONAL PROGRAMMING

The functional decomposition approach leads to a corresponding programming approach, based on the FP formalism originally defined by Backus [Back78]. This approach is motivated by the natural duality that can be exhibited between directed data-flow graphs and functional expressions. In fact, if a graph is a convenient way for an IP programmer to specify an algorithm, the corresponding FP expression is better adapted to computer manipulations. This approach then provides a great expressive power due to the manipulations that one will be able to perform either on the graph itself, or on the FP program [Will82].
Functional programs, as pure applicative formalisms completely eliminates the notion of variable. This is fundamental because it implies that they can execute on a target architecture without the need for a global controller to transfer data form/to a shared memory (this resulting in well-known bottleneck problems).

In order to efficiently couple the functional programming concept with the data-flow model, we defined a dialect <O,F,P> of Backus' FP in which:

- O is the set of basic objects. Objects are typed (e.g. PIXEL is the type corresponding to an 8-bit value) and structured (e.g. the following sequence: < P_1,...,P_n> is a LINE of PIXELS).
- P is the set of predefined functions. This will correspond to the set of primitives relevant to our application field and that have a known implementation on the target architecture.
- F is the set of functional forms. Only two functional forms are needed to provide an equivalence with the data-flow graph constructors:
  - The composition form, defined by: $f \circ g : x = f : (g : x)$
  - The construction form, defined by: $[f_1,...,f_n] : x = (f_1:x,...,f_n:x)$

Our dialect also allows a function to have more that one input object and to provide more than one output object. By doing so, the transformation between a functional expression and a data-flow-graph is straight forward, the composition functional form corresponding to serial placement of nodes, and the construction to parallel placement.

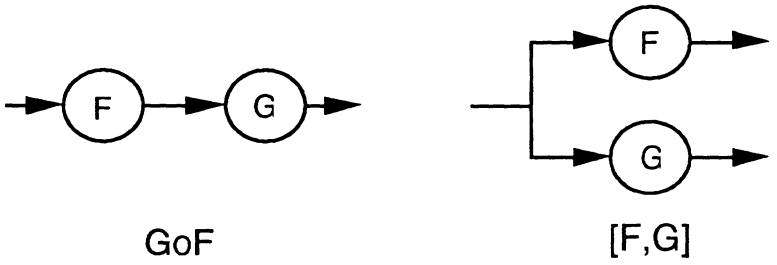


Figure 1. Equivalence between Functional Forms and Dataflow Graph Constructors

The functional programming approach presents several advantages over traditional languages:

- A program in a functional language is an expression representing a function. Program evolutions are simply the result of applying functions to their arguments.
- The syntax is very simple and can be used by non specialists. There are just a few functional forms such as the composition or the construction. The semantics of these expressions are clear.
- There is no control mechanism; therefore no corresponding automatons.
- The definition of a program is hierarchical by nature. Functions are built out of functions. In this way arbitrarily complex programs can be constructed.
- The hierarchical decomposition describes perfectly resources that may be found at the material level. If a convolver is available, then there is no need to go to a lower level of description. If it is not available, then one needs to decompose that particular function until all primitive functions have been matched onto a hardware resource.
- It expresses naturally data dependencies. This implies that such representation is well adapted at mapping expressions onto computational units: allocating them to proper resources, balancing the load between resources, and validating computations.
- The functional decomposition of a program can be achieved in two ways: we can either use a top-down or a bottom-up approach. This means that depending on the ultimate goal, we may obtain several functional expressions of a given algorithm, each one providing a different view (behavior, implementation) of the algorithm. This is particularly useful for emulating a program.

Many image processing operators can be expressed simply with this formalism. Examples include linear filtering, edge detection or the Hough transform. Following is a program for performing a basic edge detection and the corresponding data-flow graph:

```
VIDEO INPUT X:FRAME(PIXEL);
ASYNC INPUT T:PIXEL;
VIDEO OUTPUTY:FRAME(PIXEL);

DEF d1 = abs @ sub @ (X, D1@ X);
DEF d2 = abs @ sub @ (X, D2 @ X);
DEF d3 = abs @ sub @ (X, D3 @ X);
DEF d4 = abs @ sub @ (X, D4 @ X);

DEF max1 = scale @ max @ (d1, d2):
DEF max2 = max @ (d3, d4);
DEF tmp = max @ (max1, max2);

LET Y = thr @ (tmp, T);
```

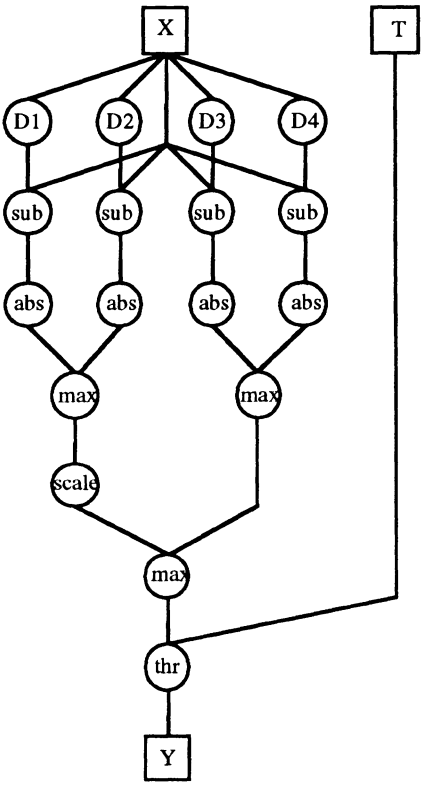


Figure 2. Example of a functional description and the data-flow graph

Expressing algorithms in this functional programming paradigm is both easy and efficient:

- Programs are easy to read because the syntax is straightforward and there is a low number of functional forms.
- The hierarchical nature of the descriptions allow to examine part of an algorithm independently of lower or higher level definitions.

C.4 - DATA FLOW COMPUTING PARADIGM

Functional Programming language is a practical and efficient mean to express image processing algorithms. However, this language is very distant from the traditional Von Neumann model of computation; this suggests that some novel architecture needs to be designed in order to execute efficiently functional programs.

Before we can consider the mapping of algorithms into special purpose hardware structures specified by a set of technical constraints, we need to get a description of the activities of an algorithm inside the space-time dimension. What we have up to now is only its static (space dimension) description. It

does not provide any information about the dynamics of executing a program: we need an execution model.

We already stated the equivalency between the functional description of an algorithm and its associated directed data flow graphs. An execution model is conveniently provided by the **data flow paradigm**. If one considers that the problem of mapping algorithms to architectures is due to the semantic gap between programming languages and the architecture behaviors, one would expect that this equivalency would greatly simplify the mapping.

There has been considerable interest for the Data Flow paradigm in recent years [Denn80][Arv86][Davis82][Vegd84], and data flow graph languages have been proposed [Ack79][Gajs82]. The basic mechanism is simple: to enable function executions, arguments must be present at all input links (edges) of the node. Then, the specified function is applied, and places its result object on the output edge. The availability of an argument on an edge is represented by the presence of a token. Execution can easily be simulated on a graphical interface, and one can see the program executing itself by observing the ways tokens are consumed and produced.

The critical factor that permits an efficient mapping between an algorithm and an architecture is the fact that one can augment this graph with technological constraints such as computing time or data bit lengths; something not feasible with the functional description. This means that we are now able, because of the equivalency between the functional description and the data flow graph, to express algorithms at a high level of abstraction (when we are interested in their behavior) and at a technological level (when we need to consider built-in data dependencies and implementation constraints). The power of this approach resides in the capability to be able to move from one view to the other, thereby taking all aspects of the problem into account: one would expect the best possible implementation of the algorithm.

To summarize, let us reiterate the fact that a program, in the Functional Programming paradigm, is an expression representing a function. Program evolutions are simply the result of applying functions to their arguments. So to represent program evolutions, we only need:

- functions (primitives or user-defined),
- dependences between functions (functional forms), and
- data objects on which functions operate as arguments.

Our Functional Programming formalism permits then an easier, safer and more readable writing of programs because of:

- the lack of control specification (implicitly defined at the data structure level),
- the ability to check for correctness of program semantics thanks to strong data typing,
- the ability to use data flow graphs to efficiently describe needed operators, and to study space-time trade-offs needed for the construction of parallel machines, and,
- the ability to hierarchically describe programs from the lowest to the highest levels of abstraction.

D - EMULATION

D.1 - DEALING WITH CIRCUIT COMPLEXITY

Behavioural simulation

Behavioural simulation has been proposed as a solution to the problem of overall circuit complexity. It represents an attempt at unifying the representation of operated structures inside all CAD tools for VLSI, by making explicit the equivalency that exists between both privileged description modes:

- Structural: the hierarchical (relational) description of components with respect to each other.
- Behavioural: the algorithmic (functional) description of a component.

Such descriptions are used alternatively to simulate each element of a circuit. The precision in calculating signals, fronts or delays from the structural description of a circuit (up to gates or transistors) is always better than the precision derived from a few parameters in a behavioral description, because they include very rough commuting or throughput timings. For instance, implementing typical values from constructors into a VHDL model of a complex microprocessor makes a significant difference with the structural model; so much in fact that some behavior may not make sense any more. The narrow errors that are now tolerated in integrated circuit designing, renders this mixed simulation a "faked miracle solution" [Tier88, Siva82, Widd88]. This problem of loosing precision at the frontier between structural and behavioral descriptions seems to refer to a more general problem of identifying a formal equivalency between an architecture (declarative nature) and a sequential language (procedural nature). A typical example of this equivalency is found when one is trying to characterize a logical circuit at the floor planning step by establishing a relationship between the logical expression (behavior) of the circuit and its microelectronic geometrical structure [Rueh84, Paris89]. As one can see, the general problem is to embed correctly an a priori non dimensional problem, the algorithmics adequacy, into a dimensional description, the architecture. Languages such as VHDL are good attempts at solving this problem. But as one can see, the need to rapidly design complex VLSI, triggered by shorter product lifes, generates some complexity design constraints that are poorly addressed by the concepts of hybrid (behavioral-structural) simulation.

Hardware modelling

Hardware modelling [Widd88] constitutes an attempt at validating in hardware subparts of circuits, and simulating the system in its whole. Such methodology advocates the direct use of some emulation hardware inside a complex simulation. There are two reasons to this: to dramatically cut down the simulation time, and to incrementally integrate critical submodules. In so doing, it is expected that the complexity that must be managed to make the circuit can be greatly reduced by avoiding modelling components, but instead by building the corresponding hardware module. This methodology would contribute also to better formalize the generation of complex testing patterns, in accordance with the expected behavior of the whole chip. This latter general

question, of testing a chip to detect hardware failures and validating its intended functioning, remains a critical and major step during the circuit design. So hardware modelling tries to get closer to the real device environment during the simulation phase, by substituting to a long and tedious definition of test stimuli, a real component that delivers precisely the same stimuli as the one in the final product. This approach raises significant software and hardware implementation problems. The interface between emulated and simulation hardwares is carried out by the synchronized storage of input/output patterns. But this technique does not scale easily. First, one is easily limited by the total number of signals that the simulators will support; thereby limiting the emulation to only a few parts of the global architecture. Second, one must precisely emulate the circuit in a manner similar to its normal mode of operation: this includes proper timings and synchronized signals. Transferring test patterns between the simulation software and the emulated hardware will then require the real time control of rapid and complex architectures (the ones being simulated and the simulator itself). Third, as in most cases, the control device is not originally included among the CAD software, so it is uneasy to graft, and prevent any generalization of the method. Finally, incorporating the ability to emulate components within a VLSI system can only result in drastic architectural constraints about the features of each emulated elements, such as limiting the grain size or predetermining the connectivity of a network; in a word, anticipating some particular architecture that was not intended.

So, if the "hardware modelling" concept appears to be a valid approach to dealing with the increasing complexity of circuits, its effectiveness may actually be limited in practice. If it settles adequately the problem of dedicated architectures in view of emulation, in the domain of I.P. these techniques would lead to software and hardware architectures which remain beyond today's technology possibility in terms of CAD environments.

D.2 - EMULATION OPERATOR SELECTION

It is then worth embedding the "hardware modelling" into a more general frame, the one of emulation, using dedicated hardware resources to execute an algorithm with conditions identical to the one encountered by the target system.

The effectiveness of such emulation relies heavily on the choice of selected operators that support the emulation. They must, in particular, be sufficiently general so that, by setting a few internal parameters, a different behavior may be achieved. More importantly, the base of operators that must be built to emulate algorithms must remain small to allow for the construction of the emulator itself. In the case of IP, this is possible because we have identified a set of primitive operators that can be used to construct new algorithms.

The choice of basic operators is done by a systematic characterization of image processing algorithms. It consists in, first decomposing known algorithms into their elementary computational structures, second detecting the commonality of such structures between various algorithms, and then generalizing such structures through their parameterization. It is then possible

to obtain very powerful sets of operators that are candidate for construction. Typical operators include filtering, convolution, statistics or segmentation operators. They are described further in the section about emulators.

D.3 - EMULATORS

The need for some emulation support hardware lead to two parallel experimentation projects (described next), the first one serving as a learning tool for the second one:

- First, the HECATE image processing emulator was designed (MIMD school of specialized modules) to support the implementation of most common IP primitives in real time (25 images per second) and to generate, from emulation results, the architectural synthesis of an IP algorithm into its integrated realisation (VLSI or integrated circuits).
- Second, the DATA-FLOW FUNCTIONAL COMPUTER used a much more regular architecture based on one custom designed VLSI processor. Its construction, based on a three-dimensional homogeneous network of processors, makes the complexity much more manageable.

E - HECATE EMULATOR

E.1 - MOTIVATIONS

We describe here the HECATE emulator (acronym for "Hôte Emulateur pour la Conception d'Automates Embarquables" - Emulator Host for the Design of Target Automatons). When the project started, there were two major objectives:

- To build some emulation (integrates the notion of real time execution) hardware, capable of running a large number of image processing algorithms. Given the formalism of Functional Decomposition, it became clear that the success of the solution depended on identifying a set of powerful, general operators that could be reused in as many algorithms as possible. These operators are presented below.
- To use the emulator as a test-bench for various algorithms for which an integrated realization (VLSI circuit) is considered (embedded application). Provided one can capture and describe an execution trace, the emulator machine would furnish a precise timing analysis of the target vision algorithm, thereby allowing to skip the more classical and approximated simulation phase, and would permit its direct integration by exploiting resource information derived from emulation.

Let us stress upon two major benefits:

- First, it provides a programming environment for the study of any image processing algorithm. Many powerful operators are available that can be used to emulate algorithms in real time, or near real time if there is not enough hardware resources.
- Second, it enables the rapid prototyping of algorithms by providing an interactive interface where parameters of each operator can be modified at will while it is executing thanks to adequate display facilities. This allows the programmer to immediately view the effect of a parameter change and permits a quick and efficient specification of any algorithm.

E.2 - GLOBAL ARCHITECTURE

From a functional point of view, each vision process is a set of concurrent communications sub-processes dedicated either to low level feature extraction or to high level decision analysis. Features can be pixel attributes (luminance, hue, saturation), multispectral vectors (colour), textural attributes, edge points (spatial differences), regions (spatial continuity), spatio-temporal differences, or motion vectors. Each feature favours particular data structures (data representation), data flows (data communication) and operators (data transformation). Decision techniques range from statistical decision (principal component analysis, k-nearest neighbours) to symbolic processing such as matching (elastic matching through dynamic programming) inference parsing, and symbolic reasoning such as prediction/verification paradigms or relaxation processes. All these primitive tools cooperate to perform robust recognition tasks by controlling each other and exchanging appropriate data structures.

From a hardware point of view, Hecate is composed of a set of parameterized hardware operators powerful enough to emulate a broad range of real-time I.P. algorithms [Wolf86, ZSF91, Ecch86] (see figure 3). It is a functionally reconfigurable assembly of operators.

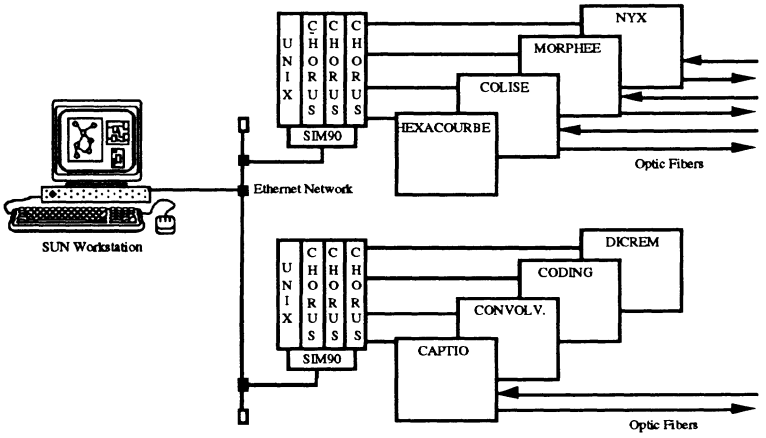


Figure 3. Global architecture of the HECATE emulator.

The control of the machine is different from traditional approaches: no sequential programming language or predefined model of synchronization is available. It is programmed through a graphical environment (described in section G) which embodies concepts for intuitive programming [Coster85, Rosen76].

E.3 - OPERATORS

The HECATE emulator is a set of highly specialized and reconfigurable image processing modules, themselves comprising many operators belonging to the same functional processing class.

This machine obeys the so-called functional decomposition paradigm for pattern recognition that defines a series of transformation mechanisms (operators) starting with image acquisition and feature extraction processes, followed by higher and higher level cognitive computations. Through the systematic characterization of image processing algorithms that consists in decomposing known algorithms into their elementary computational structures, identifying structures that are common among multiple algorithms, and then generalizing these structures through their parameterization, a set of very powerful operators was defined and implemented [ZSF91, Ecch86].

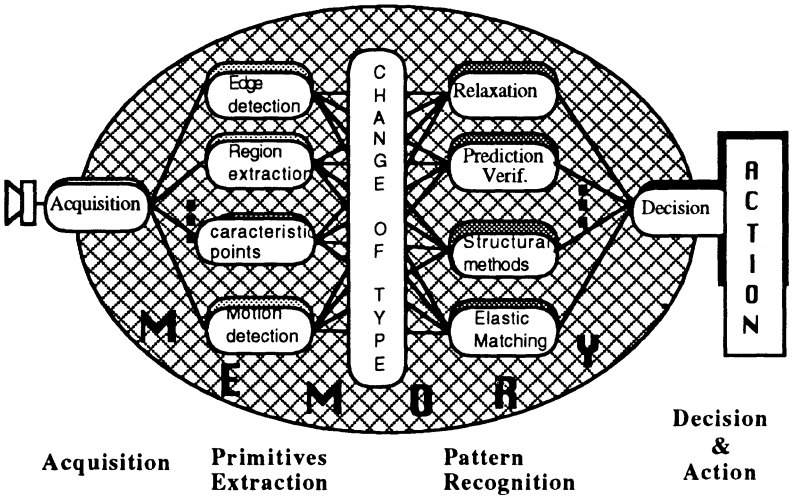


Figure 4. Functional synoptic of the HECATE emulator.

Most of the operators are built with ***MSI*** chips including some ***DGP***s and custom gate array and standard cell circuits. The wiring is mostly done in wrapping except for generic units such as address computation units; as there is only one instance of each operator, it does not economically make sense to realize printed circuit board versions of them once an operator has been debugged. All operators are realized with extended quadruple eurocard format. Video buses are interconnected between modules by wires to increase intermodulation immunity, or by optical fiber cables to minimize transfer delays.

The set of modules is interconnected by very high speed buses that carry many data types (such as images, histograms or lists of regions) depending on the functionality of each module that produces or consumes the information. The interconnection network is hierarchical: an external bus for global control, and many internal buses within each module.

The control of the whole machine is operated by a host computer that provides the user with a programming interface hiding all details of implementation of an operator.

CAPTIO

CAPTIO is the acquisition and preprocessing module. It acquires digital or analog images (camera, photodiode array) from a scene and formats them appropriately for the requesting module. It performs the adaptation of signal levels between multiple sensors and the correction of sensor defects. After conversion, preprocessing operators includes linear combinations of images, thresholding, lookup conversion and temporal accumulation. This module contains the following resources:

- Two independent digitizers: each one selects an input from four analog and one digital signals. Gain can be adjusted automatically and manually.
- 32 x 4 conversion lookups to implement any point operation on 4 x 8-bit images simultaneously.
- One four image input adder to perform transformations on colour images.
- One recursive adder to perform temporal accumulation.
- One image comparator.
- 128 conversion lookups selectable by line number to correct for offset variations (typical of linear sensors).
- One comparator between two images.
- A colour visualization circuit.
- Two independent region of interest extractors.

DICREM

This module performs structural elastic matching on data sequences (any information coded as a list of attributes). The matching between two sequences (sentences) is based on a dynamic programming technique used in speech recognition that quantifies their global resemblance using only local similarities between attributes. The architecture of the module was defined to fit within the framework of functional decomposition. It offers several features specifically designed to take advantage of image data structures:

- Images can be used as resemblance matrices. This module can then be used to extract and link contour points.
- The graph obtained after execution that provides the optimal associations between reference and unknown word attributes can be parsed in both directions.
- Attributes used in computing resemblance coefficients can be attribute vectors.

The architecture is structured around a multitask environment where several processors cooperate to perform all calculations. One processor generates the resemblance matrix that calculates similarities between reference and unknown features of each sentence. Another processor generates all possible transitions from a node in the matrix. Then a series of elementary processors (TMS 320) compute, in a distributed fashion, the optimal path (best matching between

reference and unknown utterance) in the matrix. With a 32x16 matrix, one hundred references from a dictionary can be compared per second.

CODING MODULE

As we have just seen, the DICREM modules compares one dimensional entities. To adapt this technique to images, one needs to transform objects into one dimensional shapes. This module first codes a contours into a Freeman chain. Then it performs some polygonal approximation and encodes the chain into a representation that is translation and orientation independent. This module is tightly interconnected between CAPTIO where it gather images, and DICREM where it serves to construct a dictionary of reference shapes or encode an unknown shape.

GENERALIZED CONVOLVER

The GENERALIZED CONVOLVER (inner product of APL) generalizes the definition of the convolution expressed as a sum of multiplications by allowing the two involved operations (sum and multiplication) to become parameters. Typical extensions involve logical operations (or, and) and comparison operations (minimum, maximum). A convolution operation is computed at each pixel cycle.

The traditional convolution, in the context of image processing, defines an application domain in the form of a neighbourhood. The size of the neighbourhood is nine elements and can then represent a 3x3 window. Another generalization of this module concerns the way the nine points are selected: they can be any points in a 16x9 neighbourhood around the center pixel.

The true power of the module is given by two custom circuits called the "local shaper" that constructs 9 parallel data flows and the "generalized convolver" that computes a function of these 9 data flows.

COLISE

The COLISE module is devoted to the segmentation of images into regions and edges [Ecch92]. It includes many atomic operators whose assembly permits the implementation of many sophisticated functions: linear filtering, non-linear filtering, adaptative kernel convolution, or region labelling and fusion. The approach is similar to the one encountered in the design of RISC processors: to identify the minimal set of basic operators that lead to an efficient implementation. Because the objective here is to manipulate segmentation primitives, the trade-off is between the instruction set (definition of operators) and the communications (interconnections between operators) so that the largest class of applications is covered.

The great functional power of Colise results from the definition of execution (operators) and association (communication) primitives. Three types of operators are defined:

- Simple operators:

    - Unary operators: thresholding, log(x).
    - Binary operators: add, subtract, multiply, logical-and, logical-or, minimum, maximum.
    - Ranking, dynamic thresholding
- Parallel operators: they cover the same operations but either operate on neighbourhoods or accept different images.
- Global operators to gather global statistics on the dynamic range of an image or average size of a region.

Using these basic operators, the module is capable of implementing very sophisticated operators. Among them, real time region labelling algorithms can be performed using a generalized version of the traditional connected component labelling technique. Instead of the classical inverted L mask to analyse an image, one uses a mask that can have a variable number of pixels, sparsely located around the center pixel (variable shape, size and topology). Then one explicitely defines fusion decisions that must be taken based on attribute and label values of each pixel in the fusion mask. The algorithm works on grey scale values and uses thresholded comparison operations to determine whether two pixels are the same or not. Such an approach is quite powerful because it permits fusion of textured regions (merge if there is a similar pixel in the neighborhood) or fusions in privileged directions (by defining a spatialy non symetric decision table).

HEXACOURBE

Curves are frequently used in image processing; in fact one rarely find an image processing application without an histogram or a projection. This module was designed to suport two main functionalities:

- Generation of curves (histograms within a window, and horizontal and vertical projections), and
- Analysis of curves (extraction of extremum values, and ordering of values).

The module contains twelve specialized circuits dedicated to the generation of curves. Depending on the image size, 2 to 6 proprietary COURBE circuits are used (6 for a 512x512 image). Each circuit computes either an histogram or the horizontal and vertical projections. Each of these two operations require a different controller that is loaded into the XILINX gatearray associated with each COURBE circuit. The analysis of each curve is systematically performed for each image and is computed during the vertical retrace interval. Real time operation is supported by a 2-stage (image level) pipeline organization.

NYX

Because it is unlikely that the object of interest in a scene will not present itself in a referenced orientation, an efficient way of recognizing a shape is to rotate it in the image plane and then match it with a model by simple difference. Typically this leads to very robust techniques because the matching function is well characterized. The NYX module is capable of geometrically transforming an image in real time (CORDIC algorithm). Zooming,

translation, centering, rotation and windowing operations can be done in real time on an incoming data flow.

MORPHEE

Last, but not the least, the module MORPHEE provides all the memory necessary for emulation purposes. It is important to realize that all previous modules do not have any local memory and they execute all their operations on the fly. This approch fails in two cases:

- when data formats between two sequential operators are incompatible (interlaced, non interlaced), and
- when not enough operating resources are available to permit a real time implementation of an algorithm (which is the case with most algorithms).

One must then use temporary storage either to convert data structures into appropriate formats, or emulate algorithms in near real time.

This module provides large storage capabilities and arranges its memory into physical blocks (4 MByte banks) for local parallel accesses. But to other modules, it partitions its memory into logical blocks of arbitrary sizes. It is a shared resource between other HECATE modules. The versatility of this module comes from its 6 address generators that support multiple data type transformations. Such transformations are necessary due to the ever changing nature of data as it is passes from an image into a symbolic entity; for example, a list of points of interests become a polygonal shape when on tries to connect them. Each module can then process images in a format best adapted to its inner (input, output, operative) wired functions.

The primary usage of this memory is to provide temporary storage, either to allow for the emulation of an algorithm, or to synchronize data flows between two operators. In any case, it has the effect of delaying data transfers, thereby reducing emulation speed.

Architectural trials

It should be noted that, besides providing specific image processing functionalities, each module was designed to test and validate a different architectural approach better adapted to the problem at hand. For instance, CAPTIO uses specialized hardware modules communicating through a highly interconnected network made of multiplexors circuits (MUX). DICREM uses multiple TMS 320 processors connected to a ring network. HEXACOURBE is built with many Xilinx circuits. And COLISE is a high level (at the level of segmentation operators) mixed VLIW and data-flow machine to which is atached a corresponding assembly language. All these architectural trials were a major source of inspiration when conceiving the processor for the DATA-FLOW FUNCTIONAL COMPUTER, and explain entirely its functional efficiency.

E.4 - CONTROL OF HECATE

The controller of HECATE is a multiprocessor system, where each processor is dedicated to a specific functional task. Each module gets a dedicated real time controller. The user interacts with the machine through a graphical interface (detailled later) residing on a workstation. This central unit dialogs with two UNIX (for transparence reasons) processors over an ethernet network and the TCP/IP protocol. Each UNIX processor transmits to each module controller the list of tasks that must be executed over the VME bus.

Each module necessitates a separate processor to ensure proper configuration and real time control. The high level (synchronization with other controllers and the HECATE interface, conversion of task requests into a series of data transferts to the module) controller is done by a server accessible to all. It uses a real time CHORUS kernel running on a Motorola microprocessor. The low level (data transfers with the module) controller provides all input/output interfaces with the module, and uses a SIM90 frame (multi 68000 processor system built by TRT supporting a VME bus and a specialized high bandwidth input/output bus).

One of the main features of the controller is that it allows the programmer to change parameters of any operator while an algorithm is executing. So the programmer sees effects of his modifications right on the screen. This provides an efficient way to conceive and adjust an algorithm in the same way one adjusts the contrast button on a television set without having a precise knowledge of what contrast is.

E.5 - USE

The construction of the HECATE emulator occurred over an 8 year period and ended in June 1991. Not all previously described modules were entirely finished except COLISE; in some cases some operators were not implemented as originally intended, or were not tested. The project was stopped, primarily for support reasons due to a lack of permanent staff that could maintain appropriate knowledge on the various modules. Some of the first modules, such as DICREM, were working sporadically towards the end. So when the DATA-FLOW FUNCTIONAL COMPUTER became available, development stopped. The HECATE project nevertheless provided most of the concepts found in the second generation DATA-FLOW FUNCTIONAL COMPUTER emulator, whose realisation has gained in regularity, homogeneity, and integrability.

F - FUNCTIONAL COMPUTER EMULATOR

F.1 - MOTIVATIONS

In signal and low level image processing systems, the data-flow paradigm may provide the framework for solving many problems. Because data flow expressions of algorithms permit an exact representation of parallelism, an effective simulation and synthesis can be made. Thanks to its natural expressivity, especially for fine granularity (atomic data flows), the designers usually thinks spontaneously of such systems in terms of functional blocks [Allart88]. Moreover, the functional model offers a significant advantage for

signal processing: the natural expression of concurrency in the description simply because data-flows dictate that a computation be performed when data is available. Therefore, as long as the integrity of data-flows is preserved (data dependencies are kept identical), all implementations of data-flow descriptions will produce the same functional results. Thus the synthesized system is always functionaly equivalent to the one described as input. This fact, which garantees correctness of the solution, primarily lead us to implement a data-flow architecture in our second version of an emulator.

The functional decomposition principles of programming and the Data Flow computing model presented previously became the basis for designing this new architecture. As we have already indicated, the difficulty of mapping algorithms onto a machine results from the differences between programming and execution models. To reduce, hopefully eliminate, such semantic gap, our approach has been to build an architecture that is a tridimensional graph, for which the mapping of a graph of operations into a graph of operators becomes a straightforward task.

The DATA-FLOW FUNCTIONAL COMPUTER [Quenot92] profited from previous experiences about integrating image processing basic routines. The resources that were integrated onto the custom processor that composes the machine were selected based on the knowledge acquired during the realization of HECATE. In particular, by carefully selecting and agencing computing resources on the chip, many operators that used to employ several printed circuit boards are now simply realized with a few processors (for example: histograms and projections). Also a major difference in designing the machine was to build a regular structure made of identical processors; if this facilitates the construction of a compact machine, above all it facilitates the mapping problem because one only needs to choose a processor in terms of its location in the network (not its type). One nevertheless need to select the operation it will perform.

Let us note that the functionality information about a processor will easily permit to derive a target automaton by eliminating processors that only route data.

F.2 - ARCHITECTURE

The core of our Data Flow Functional Computer (DFFC) is a network integrating two different kinds of processors, respectively dedicated to low-level and high-level image processing functions (figure 5).

The network is interfaced with a SUN Sparc host workstation through a specific controller, which performs the load and test functionalities along with low bandwith input/output facilities.

The low level network is composed of a mesh-connected special purpose integrated circuit called the Data Flow Functional Processor (DFFP) [Quenot91]. Each DFFP has been designed to be mesh-connected in a 3D network through 6 input-output ports (practical networks can house from 64 to 1024 DFFPs in a 8x8xN topology. Each port is made up of 10

bidirectional lines (9 for data transfer and 1 for acknowledgement). A full programmable cross-bar interconnection network allows the processing unit to fetch data from any port. Tokens can be output on each port at a frequency of 25 MHz.

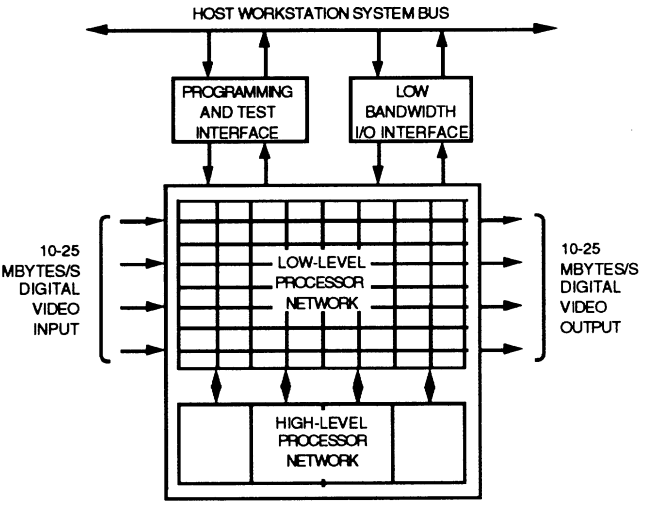


Figure 5. Architecture of the Data Flow Functional Computer.

F.3 - LOW LEVEL PROCESSORS

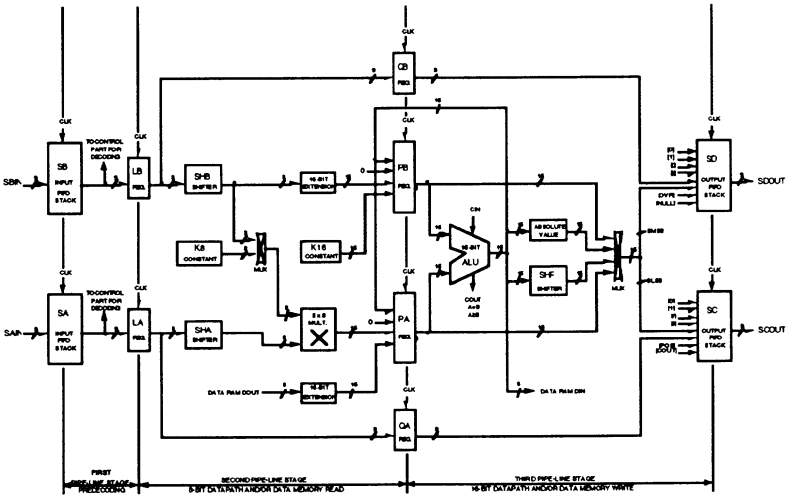


Figure 6. Block diagram of the Data-Flow Functional Processor.

Each circuit contains a 256x9-bit words data RAM. It can be used as a dual port RAM operator, as a 256 word FIFO for buffering, or as a local memory

for specific operators such as histogramming. The core of the pipelined processing block includes an 8x8 multiplier and a 16-bit ALU.

The control part has been designed on the basis of a programmable state-machine using a 64x32-bit word program RAM. Following the Data Flow computing model, the execution of each operation is fully data-driven. It is controlled by a dynamic firing rule that evaluates token avaibility on each input port and room avaibility on each output port. If execution is validated, the operation specified by the state machine is performed. Many data flow operators can be implemented using this aproach such as arithmetic operations (add, multiply), logical combinaison (and, or), reordering (line and pixel shifts), and global statistics (summation, histogramming). Two processors have been integrated into a single chip.

F.4 - HIGH LEVEL PROCESSORS

The execution of high-level functions is handled by networks of T800 Transputers. This general purpose processor which offers 4 serial communication links is very well suited to implement complex functions in our data flow architecture, as long as algorithms respect the data flow paradigm; that is, any output flow may only depend on input flows, and the rate at which tokens can be accepted satisfy real time constraints. Provided that a program in a Transputer node consumes and produces tokens from its serial links in a way similar to the DFFPs, the first constraint is easily met by implementing the operator as a single transputer process. To solve the second one we introduced a buffering mechanism using two upstream and downstream synchronizing operators. This mechanism can be used to couple any low (fast) and high (slow) level operators [Serot91].

F.5 - PROGRAMMING

As our goal is to couple each primitive of the language with a data-flow operator, a library has to be designed for each type of processor. These libraries, implementing the most commonly used primitives in the field of image processing, will mainly save end users from writing their own low-level basic operators using the DFP assembler (exactly as normal users do not have to write machine language code on classical computers). For high level operators, users will have the choice of either resorting to the library or to writing their own specific operators using a C source template. A common specific module, merged at link-time, makes the definition of such an operator independent of the actual network configuration and of any parallel routing process
on the transputer node. Programmers view of the libraries is limited to a global operator database gathering public descriptions of primitives. Each entry in this database consists of a prototype, typing operator input and output sets, a list of external parameters and their default value along with a brief description of the operator functionality.The libraries may be easily extended to suit user's specific needs at any stage of application development. This approach has the advantage of introducing a software hierarchy that maximises code reusability and therefore greatly reduces the programming

complexity. It also provides an homogeneous software interface for different architectures. The following figure illustrates the programming environment.

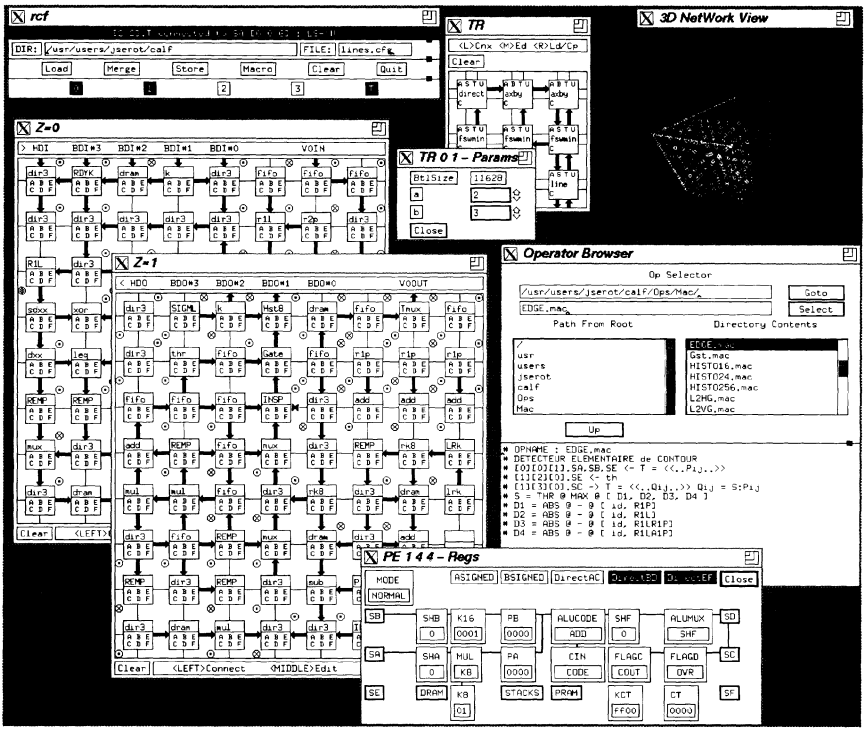


Figure 7. Data-flow Functional Computer Programming Environment

An algorithm to be implemented is first expressed as a combination of primitives using functional forms in a FP-like syntax. The resulting functional expression is then converted into an equivalent operator data-flow graph by means of a FP compiler . A translator handles the conversion of this operator data-flow graph into a loadable processor graph, that is the conversion of public instances of the primitives into their actual implementation on the processor(s). This may involve, for example, translation from external parameters to processor registers and/or macro-operators expansion. The resulting data-flow graph, where each node corresponds to an operator implementable on a physical processor has then to be mapped onto the network. This can be done by means of an automatic place and route algorithm or manually thanks to an interactive graphic tool (Figure 7). In general, a manual placement is more efficient but an automatic, even sub-optimal, placement allows fast code prototyping. This mapping stage has to take advantage of the fact that each processor can be used simultaneously as an operating and routing element. The network configuration file generated by the mapper is finally loaded from the host into the computer through the network controler. Since then it may be executed in real-time using video input and output flows or in ``remote

mode" using controller low-bandwith input/output facilities. All of the programming tools are integrated within an interactive, user-friendly graphic interface running under the X window system, making the programmation of the DFFC easy and efficient.

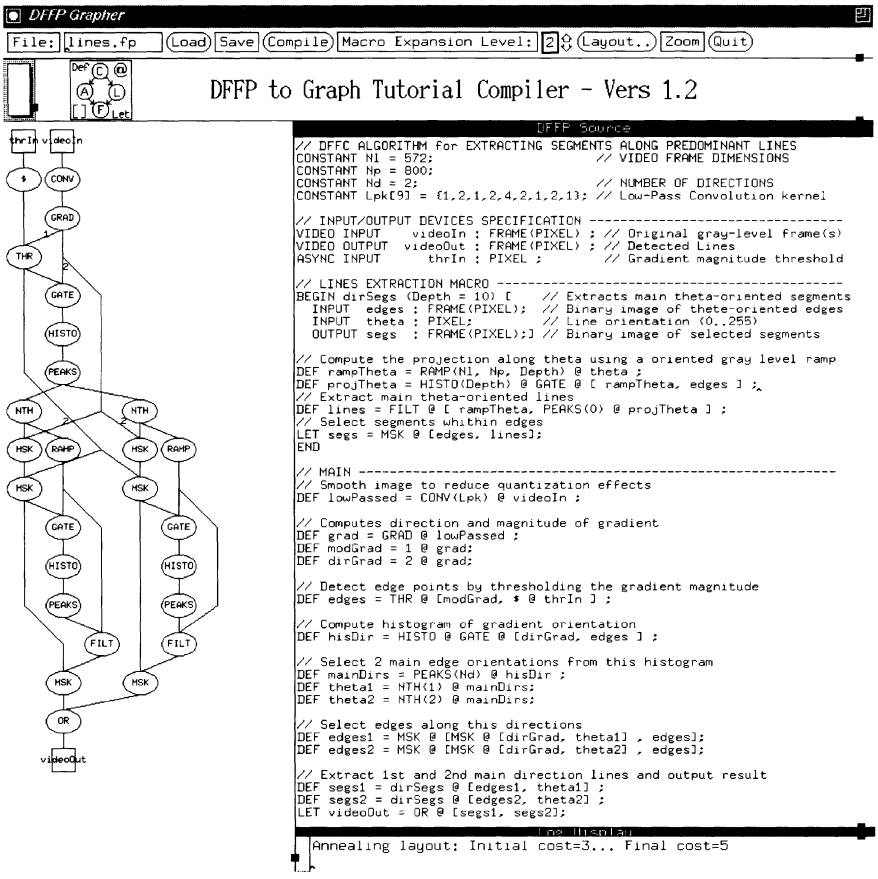


Figure 8. Example of a functional algorithm

Figure 8 illustrates the implementation of an algorithm for the extracting majoritary sets of parallel lines, based upon the computation of the histogram of gradient direction and a partial Radon transform. The source code shows an input/output specification part, a functional description of the algorithm itself, and a instanciation part. The specification part allocates physical input/ouput devices. VIDEO devices correspond to digital video sub-systems while ASYNC devices represents controler low-bandwidth ports. The functional description consists of one or more independent functional blocks (BEGIN..END). Each block combines primitive functions (upper-case) and user-defined functions or blocks (lower-case) to build outputs from inputs. The DEF definition statement enables a function result to be used more than once within the same block. It can also be used to split a complex

function into subsequent expressions. The 1 (2, ..) pseudo-operators are function output selectors. From a classical programming language point of view, each block is a function declaring inputs, outputs and making use of local variables. From a data-flow graph point of view, a block is a stand-alone subgraph, which can be freely replicated. Moreover, it can be shared among source files, precompiled and inserted at link-time, or even included within the libraries (provided that the programmer writes its data-base external specification). Finally, the instanciation part (LET) associates block(s) functional inputs/outputs with the previously declared physical devices. This application can be easily extented in order to take into account more directions. As described it uses nearly 200 DFPs and 2 transputers, and has been executed in real-time on 25 Hz sequences of 800x572 images.

G - PROGRAMMING ENVIRONMENT & DIAGNOSTIC

G.1 - INTUITIVE INTERFACE

We have at our disposal two emulators configured as sets of electronic operators, embedded into flexible reconfiguration networks. The complexity of these emulators (especially HECATE) is such that it becomes unrealistic for any human operator to understand precisely how they work. On top of this, many resource allocation problems are so complex, that the determination of a solution can only be considered with computer tools. It is essential to provide an interface to the user that handles the complexity of the machine and that gives him a view belonging to his application domain (image processing in our case).

The environment that was developed attempts to provide an answer, at least partially, to the open problem of human interface. For example:

- How designers are to input design specifications and constraints?
- How is the system to output results?
- What decisions need to be made by the designers?

It is a user-friendly system that displays, during execution, operational resources and their organization. This is done hierarchically, with at the high level the application, and at the bottom level its structural decomposition into hardware cells implemented by the emulator.

The interface is meant to be used by a programmer that can generate a functional description of the algorithm needed to solve a problem. At this level, he can concentrate entirely on the solution to his problem, irrespective of other aspects of the solution. A person not expert in image processing is able to handle the machine resources needed by the application (recognition target tracking, etc.). But on the other side, a hardware designer is able to concentrate on the architecture problem by looking at the architecture facet of the solution. Such programming environment fills the gap between the behavioral description of an algorithm (the functionality that must be implemented) and its structural description (the architecture that will implement it).

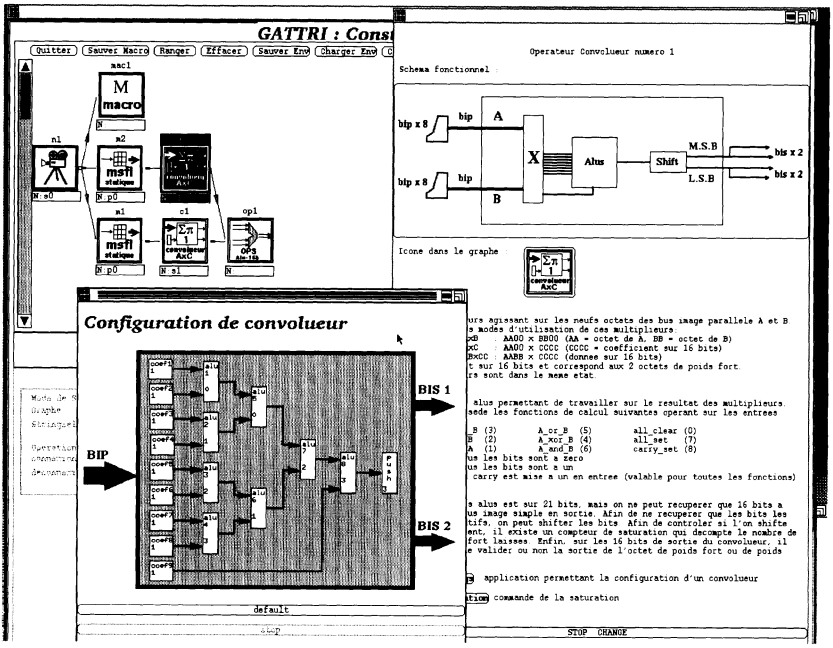


Figure 9. Programming interface for the HECATE emulator.

G.2 - MULTIFACET PROGRAMMING ENVIRONMENT

The environment we have built offers the programmer four separate and independent facets that result from the many ways in which an application may be viewed:

- Definition of a functionaly correct solution (programming): the system provides both a friendly access to the machine (graphical programming environment) and some guidelines on how to use the available operators of the application domain (hypertext paradigm interface). The programming environment offers tools to manipulate high level entities such as image processing programs and implement a given algorithm without having to worry about the configuration of the emulator.
- Control of the emulation of an algorithm (emulation): the environment supports a real time and distributed operating system to manage and control all resources of the emulator. The specification of an algorithm must be transformed into a form executable by the emulator. Debugging of an algorithm supposes that one can set new values of parameters while emulation is taking place, and that the effect of such changes can be seen instantaneously on output screens.
- Analysis of the execution of an algorithm (diagnostic): once a program has been written, and in view of the ultimate goal of building a target architecture to execute it, it becomes necessary to monitor resource

usage as a program is emulated. The idea is to evaluate integration possibilities (outlook) from the very first steps of algorithmic debugging. This is accomplished by spying on which resource has been used and when. It generates a trace of the execution in terms of hardware operators, described at various levels of details. The goal of this diagnostic process is to prefigure (preview), independently of any actual control of the target machine, its organization structure in terms of hardware components and their temporal sequencing. This allows the programmer to modify his algorithm and come up with a solution that has all the chances of meeting realization constraints.

- Design of a target architecture (synthesis): once a solution is functionaly correct and the rough evaluation of its hardware resources is satisfactory, the environment provides integration tools to propose an integrated version of the solution that satisfies realization constraints of heat dissipation, power consumption or volume.

G.3 - MODELLING

The control of the emulator is divided into two levels:

- A low level control that sets all internal parameters of each operator. It is distributed over all processors and handles communications between microcontrollers installed on each module. It can be viewed as an extension of hardware services and provides a functional view of each hardware operator.
- A high level control that provides a uniform and coherent view of each microcontroller independantly of its location on the network. It is based on a message passing architecture and leads to consider each operator as an object, following the "object oriented programming" paradigm.

In the high level environment, we are modelling hardware operators and image processing primitives that can be run on the emulator with "objects" that associate data structures and the methods to manipulate them. These objects correspond both to operators in the image processing sense as well as material resources. Such objects can either represent complex image processing operators, subroutines or entire algorithms. Such modeling attempts to reflect the functional decomposition encountered in image processing.

G.4 - DATA ORGANIZATION: HIERARCHY & GRAPHS

Operators, at any level (both structural and functional), are represented as objects. Each operator communicates with another one by sending and receiving messages. A hierarchy of operators is established where one progressively translates an image processing operator into an elaborate set of interconnected hardwired operators. The natural recursivity of this representation permits a homogeneous vision of a program down to its lowest level operators, following the paradigm of functional decomposition already presented.

A graph is a data structure that stores (relational) associations of entities through links; it is more complex than lists or trees. The purpose of this

organisation is to allow the user to browse inside a database by means of relations existing between the various objects in the system. Such structures are quasi universal in data processing. For VLSI design, graphs are the main support in structural building from circuit to board cells; The basic I.P. process description is the graph of operators. This choice of representation for both image processing algorithms and their hardware implementations comes from the I.P. decomposition principle which brings out functional primitive data movements and operators, that are split recursively into more basic entities (convolvers, multiple adders...). In both projects, the operator graph is a key object upon which all control is applied.

G.5 - OPERATORS & MACROS

Hardware operator objects are virtualized as Le LISP objects. Le LISP is a symbolic dialect and accepts object oriented features. It was chosen for its performance, easiness of portability to a new system and facility of adapting the environment to a different aplication. As an example, Aïda, an UIMS like language, is built on top of the object kernel of Le LISP, Leyx. Objects designed with Aïda are strongly structured and obey the object oriented programming paradigm: they react to methods and they are true encapsulated entities including data and programs. Their structure adapts to more general rules of virtual objects in circuit design: they offer inputs, outputs and an explicit internal structure in the case of composite objects (i.e. MACROS = objects including other objects).

Operator inputs and outputs are also encapsulated into autonomous objects featuring data flows with connect-disconnect methods, flow type compatibilities. The reason why pins have been considered autonomously under the form of objects, stands in the efficiency of managing locally multiple coherence controls at the very level of connections. It is not worth assigning this task to operator objects: as is, several differently flow-typed pins (Image, neighborhood, pixels...) are modelled and interconnected according to precise, yet simple, rules.

Thanks to pins, objects interconnect with each other to constitute graphs. Graphs can be cyclic; if they are, the differences show in the graphic display, and in the control of an hardware operator loop at a very low level. Object graphs, say MACROS, represent either directly implementable processes, or virtual processes. "Virtual" stands here for "unable to be implemented directly", i.e. a process to be transformed or sliced, using intermediate added memory or operative units. Graphs represent also some fuzzier entities such as blocks of code in some common language or DSP instruction files. Thus macros are composite objects, in the hierarchical frame sense (Minsky, Bobrow), which incorporate other objects. The structure field is the computer link (pointer) between external and internal aspects of a macro, under the form of object and connection list. One of the main interests of macros is the functional resemblance between their interface and external aspects to those of atomic operators. As a consequence, whether they are composite or not, all objects are handled in an homogeneous way by the various system tools.

G.6 - COMPORTMENTAL LAYERS & FACETS

Multifacet operators

As already mentioned, the internal objects structure obey the paradigm of object oriented programming in being given facets perspectives, or frame slots (Bobrow, Minsky, Jay, Sussman) to express abilities to integrate informations about particular uses, depending on actual necessities. For instance, multiple news and facets allow an operator to be considered a structure of command words at the hardware level or a graphic interface to access its own programming or documentation. Let us recall that "facet" refers to the many facets of a diamond that reflect light in multiple directions. In computer science, a facet corresponds to a functionally specialized interface which communicates with objects and manipulates specific data of them, exactly as different programs would operate on some data. As soon as a considered object is involved in some activity, it is oriented towards another functional interface.

Typical facets are:

- Graphic interfaces: managing mainly graphic data and the object representation on a display, under the form of interactive boxes sensitive to the mouse. On top of that, this facet direct user's actions towards other object facets, thus providing a standard channel for reactivity between this object and the programmer.
- Documentation: offering texts for explanation, schematics or technical data about the operator hardware design. Links between graphic facets and command facets are also present to provide on line help while emulating an algorithm on the architecture.
- Specification: describing the operator structure as a net list of components, transistors, gates and other layout masks.
- Control (command): gathering orders and transmitting them to the very low hardware level to program hardware operators from their modelization, or control the emulation.

It should be obvious how facets help implementing complex functionalities while allowing for some systematic code structuring, writing and debugging. Then facets may cooperate in building some complex functionalities and participate in relations of the "customer-server" type. This happens for instance in the graphical facet which interogates other facets of a given operator depending on services requested by the user.

Multifacet graphs

By adding facets to graphs, modelling relies on a double structural hierarchy that allows to manipulate independently of parallel programs, VLSI outlines, hyper texts structures, themselves built of graphs. Atomic elements of these structures host the non reducable and unsplittable information about operators, and determine an autonomy of organization independent of the containing super structure. Thus, by modelling parallel processes in IP under the double form of operator graphs and hirarchies of individual operators, independent code can be written and organized among the various facets to

plot the hardware, document the functionality on line, or to extract a net-list representation of the operator.

The benefits of this double structural hierarchy can be used, for example, to perform symbolic image processing. The I.P. symbolic features of objects are assigned to a specific slot which can store affinity or association properties between operators (for instance: neighborhood + convolution ==> homogeneous transform) or features specific to sophisticated processing (smoothing while preserving contours). The key is to realize that deducting global features associated with an operator graph is made easier by using the connectivity between operators and the local properties stored in the operators themselves [ZSF92].

Another benefit of this structure is to provide a friendly interface to a database. To accomodate the complexity of the emulator, the user specialist in IP is very much in need of a friendly interface since he is not supposed to be aware of the machine inner software and hardware. Such a system furnishes all needed information about the operators, their functionalities, or the way their parameters must be set, through slots named "interactive documentation". The activation of the slot "documentation" for a given operator leads to an hypertext structure. This text structure allows the user to either browse the documentation or to search for specific information about a macro.

G.7 - NOVEL COMPUTING SYSTEM

The distribution of activities among the various graph operators is an important aspect in the way the emulator is controled. It makes possible the implementation of interpreted commands at the level of hardware operators. Such interpretation is done through the graphic interface. It is a straight implementation of the "reactivity" paradigm of the Smalltalk language. Execution orders (command facet) specified by the user are translated into a series of messages for the controller of the operator or into orders for the internal configuration of objects. The same process can be applied for connections at a much lower level: when the user connects two operators by their "pins", the interface facet of these pins take into account the link, locally verify the consistency of the connection, and eventually, generates orders back to the user. This kind of interpretation mechanism truly implements the notion of "rendered service" through its multiple facets. These facets can be the visual representation of an object on the screen through which the user gathers documentation, functional descriptions or control parameters

This kind of direct handling of operators, connections and graphs presents two advantages:

- The interactive design of I.P. programs.
- The tuning of object parameters while being directly "plugged" into the hardware.

One is therefore able to process, in real time, images in a way that is immediatly explicite to the user. In so doing, we create a sort of virtual "visualization driver", truly equivalent to a hardware driver, that can be instantiated on any node of the graph (application program) being conceived,

and permits to watch the effects of a parameter tuning or a connection-disconnection operation between the operators of a graph. This way of manipulation corresponds to our goal of considering the system as a true CAD software tool as well as a specialized simulation tool for image processing, except that processing and execution monitoring are emulated in real time.

This, of course, can only be done as long as enough resources are available on the emulator to permit the execution of the algorithm in real time. If the emulator lacks resources to permit a real time emulation of the algorithm (as will necessarily be the case with complex algorithms), one has to use notions of program compilation and execution capable of handling such limited resources.

To accomodate this kind of situation, one had to define a process of partitioning processing virtual graphs into sub-graphs. For each sub-graph, enough resources are available and execution can proceed in real time. A reduced form of the interpretation principle previously described can be applied where the effect of modifying a parameter will only take effect when the corresponding sub-graph is executed. One must then precompile a program by decomposing it into several sub-graphs whose execution are done sequentially using the emulator resources. This process uses the image memory module of the emulator as a way to temporarily store image flows. This concept, of distributing the activity associated with a graph, allows a smooth transition between the classical entirely sequential Von-Neuman execution and generalized parallel execution where one varies the "granularity" of operations that can be implemented by the emulator. Indeed, by identifying "slices" of interconnected and executable operators in the graphs and by storing intermediate results generated by these computational slices, we are partitionning the computational load in a way that greatly increases execution concurrency and takes advantage of the parallel execution ability on the emulator.

The slicing of operators, briefly speaking, is done by searching the graph for high level operators and evaluate the resources configuration of the emulator as its resources gets allocated. When a conflict occurs between operators resources (operator already allocated by a node in the graph), one determines in which other branch of the program execution can be pursued. This process is repeated until a global execution scheme has been found that minimizes timing constraints (execution as short as possible) while maximizing memory usage (Morphée module in Hecate). Tools are available that measure performance expectation of a particular implementation in terms of communication bandwith, memory usage and operator efficiency on the target system. Once a satisfactory solution has been obtained, emulation is done by sequencing parallel branches in the graph through their control facett.

The search for a "good" partitionning solution is mainly heuristical because of technological constraints on operators (clocking scheme, data volume to store in memory), the topology of the virtual graph we need to emulate or the type of needed operators. Since a global and optimized management of parallel command is out of reach at this level because the problem is too complex,

and since it is better done at the circuit synthesis level if really required (see section H), we are implementing this methodology of overlapping graph branches because it provides the following aspects:

- This method is not, most certainly, optimal in term of parallel implementation but it works and gives a non-degraded functional aspect to the global system, specialy with "large" processes.
- Above all, it is evolutive. The environment in which this slicing is done is a high level one where all the information about performances of an implementation are easily available and manageable.

Another integrated concept, that results from the formalism of object-oriented databases and behavioral facets, is the combined fonctional/behavioral simulation. Behavioral facets can be implemented for macros as well as for operators; because they all are regular objects. The behavioral facet of a macro contains text (for example sequential VHDL code) that describes the simulation code for that macro. On the other side, behavioral facets for operators hold either sequential code, or traditional simulation code (descriptions in terms of gates, transistors, registers, etc.), or both. So, depending on how one stimulates the behavioral facet of a macro (behavioral description or VLSI structure), one determines by the same token the simulation mode of the macro in a transparent way.

Let us remind here that the real significance does not rely in the specific implementation of the combined fonctional/behavioral simulation aspect of a macro, but in the fact that it appears in our formalism only as specific facet, a simulation mechanism, that could be extended with any other simulation mechanism. Consequently, the concept of an object database where one separates and controls the activities of each object seems even more powerful as it provides straightforward ways to implement each of its activites.

G.8. - DIAGNOSTIC

Let us recall that the goal of our two systems (Hecate and the DATA-FLOW FUNCTIONAL COMPUTER) is both, the real time emulation of complex image processing processes, but also the integration under constraints of VLSI circuits that become hardware implementations of emulated algorithms. What really counts is the system aspect that such a conception tool can present under its two aspects: emulation and integration. Diagnostics is precisely the set of techniques that formalize and materialize information transfers between these two domains.

The formalism previously described takes another dimension when one realizes that it supports, in a natural way, diagnostics functions. In the same way that programming and emulation tools previously defined provide user's help, symbolic programming or coherence verification by relying on the multiple facetts of operator objects, diagnostics represents another similar behavioral facet that extracts VLSI information for these objects.

Given a graph built either by the tool that builds image processing programs or by the compiler, the diagnostic system interprets links and VLSI facets for objects in the graph to generate a net-list of structural features, in a way

similar to any CAD system. This mechanism is quite powerful and flexible because, similarly to the way behavioral facets describe simulation or emulation behaviors of objects, VLSI facets can be implemented in terms of microelectronic structures (circuit, transistor or mask net-lists) or in terms of behavior (VHDL or HILO code). Depending on these options, one obtains code that is directly usable by a routing tool or code that can be resimulated locally where a more detailed look is necessary. Let us emphasize that this characteristic of dual functional/behavior environment is a "natural" consequence of the ability to manipulate and exchange the various facets of objects in the database.

G.9. - A NOVEL PROGRAMMING ENVIRONMENT

From a methodology point of view, we believe that a major interest of our approach is its proof of the practical power and ability of object-oriented programming techniques to formalize very complex information processing systems such as the HECATE machine. In our case, the techniques themselves used to manage the object-oriented database, solve directly the problems of diagnostic of VLSI circuits, interactive and delayed control or on-line documentation. But moreover, the fact that several concepts about hardware description languages or emulation can actually be described under a unique formalism, leads us to believe that their software implementation is very close to what we have done. More importantly, the concept of efficient emulation through rapid prototyping of complex integrated circuits the way it has been implemented in the Image Processing domain leads us also to believe that this solution is a good one and that the necessary tools are already available.

H - SYNTHESIS

H.1 - PROBLEM

Task partitionning

Having obtained a data flow description of the algorithm (vision automaton), we now want to generate from it a target architecture. This final phase of our process consists of automatically synthesizing and integrating such description into a target architecture. This phase can be decomposed into two tasks:

- Data path synthesis which itself can be further subdivided into operation scheduling, operator allocation and operator assignment (or operator binding).
- Control part synthesis which is further composed of micro-instruction scheduling, state assignment or boolean optimization depending on the choice of a specific controller strycture.

In many systems these two parts (data path and control synthesis) are synthesized in two successive steps. The first step provides a data path layout from a behavioral description of the algorithm. The second one provides a control part description from the data path and associated CDFG using a

control state graph. Since these two steps are strongly interdependent such artificial splitting hampers the quality of the solution.

Before describing our way of solving this problem, let us describe in a more detailed fashion, the purpose of each phase of the synthesis process, which assumes a synchronous machine.

Data path synthesis

Operation scheduling determines the time sequencing of operations. Given a data flow graph, the goal is to decide which operations (add, multiply, etc.) will be performed at each cycle. This is done by manipulating the graph so as to optimize the performance of the system; knowing there is always a trade-off between execution time (depth of the graph) and computation area (width of the graph). The operator allocation phase selects the number and types of hardware resources needed (for example: 1 ALU, 2 adders and 1 multiplier) and the performances of each operator (for example: a 12 bit adder with look ahead carry unit). The operator assignment step determines on which hardware resource (operator) each operation of the graph will be implemented. This step affects the types of interconnections (bus, registers and multiplexors) between operators; therefore the goal is to globally minimize the number of multiplexors and registers, and the length of interconnections (by typically varying the number of registers). Once all processing resources have been selected, the floorplanning step grossly places all modules on a two dimensional grid (bus locations are typically not laid down at this stage).

Having selected all data path components of the architecture, one needs to control it (control is the process by which one activates the right module at the right time; for example: load a register, select a multiplexor, program an ALU).

Control-part synthesis

Once the data-path is set, we know exactly which resource need to be selected, how (values of the control signals) and when (which time slot). The description of the behavior is then given by an intruction list. From this list we can extract the logical equations of all output signals. At this point, many implementations are allowed and the synthesis system have to choose one of them depending on specific features to be integrate (existence of many branching schemes, subroutines, etc ...) and on specific constraints to respect (area, speed, etc ...). From these implementations, two models can be generalized (classified by their degree of optimization) :

- Microprogrammed controllers,
- PLA-based controllers.

The first one is a classical model where each instruction is located in a specific address of a ROM and a sequencing operator (generally a counter) validate one of them at each cycle. Some external hardware could be added in order to implement specific behaviors (address stack for subroutines, next-address multiplexor for conditional branchings, etc ...). The second model is the generalized implementation of a finite state machine where a

combinatorial block generate instructions (primary outputs to data path) and next state from conditional inputs (primary inputs from data path) and present state. Many sequencing schemes can be implemented with this model.

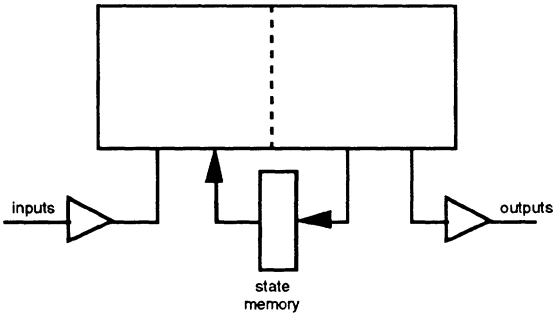


Figure 10. Structure of a finite state machine

For each model, some optimizations can be made :

- Microcode compaction for microprogrammed controllers [Lands80]
- State assignment and boolean minimization for PLA-based controllers [Ama] [Paul89a] [Villa90]

A lot of work has been done on PLA-based controller synthesis due to their attractive characteristics : Layout regularity, relative design simplicity, testability, etc ... These works concern optimal state assignment (choice of the correct boolean state code minimizing the amount of hardware needed to implement output equations) and finite state machine decomposition (reducing the total area and critical paths). A brief survey of these tasks is given bellow :

- Optimal state assignment : In a PLA-based controller, each output is computed in a sum-of-product way. A PLA consists of two logical planes (AND-plane and OR-plane) where each intermediate signal (product-term) is an AND-combination of the inputs. Every boolean function given in a tabular way can be implemented in a PLA : Each input combination generate a product-term (active only when this combination is applied to the PLA) and each output is an OR-combination of the product-terms.

i1 i2 i3 i4	o1 o2
0 0 1 0	1 1
0 1 0 1	1 0
0 1 1 1	0 0
1 1 0 0	0 1
1 0 0 0	1 1

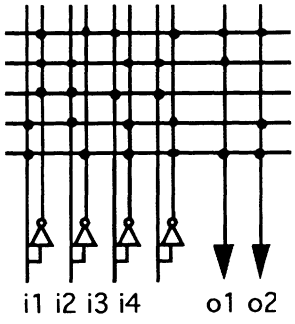


Figure 11. Tabular description and PLA implementation of a boolean function

One can see that each input combination generate a product-term and the complexity (area) of a PLA is directly proportional to the number of product-terms. A PLA-based finite state machine is nothing but a logical function where inputs are the conditional signals generated by the data-path and the state codes of the controller (inputs of the AND-plane) and outputs are instruction and next state codes (outputs of the OR-plane). This implementation is the generalized model of a MEALY finite state machine (outputs are generated by transitions in the state graph). Given a symbolic description of a MEALY machine (the states are represented by symbolic values in place of boolean codes), the optimal state assignment problem consists of choosing the proper boolean code for each state minimizing the number of product-terms.

- Decomposition of finite state machines : The goal of this task is to reduce the total area of a controller by decompose a unique machine into one or more submachines. The techniques used are based on detection of "factors" which are, in fact, subroutines and on implementation of these factors on independant submachines. Some experiments show that these optimization reduce both the total area of the controller and the critical paths (speed up the controller).

After optimization of the controller and the data-path, the netlist of operators we get can be given to classical placement and routing tools.

The figure below summarizes the organization between the various tasks that compose the synthesis process. It should be clear that such process is, before all, a global optimization problem. As a result, the generation of the data path and control components of the the target machine is done by means of global simulated annealing techniques that permit the simultaneous integration of many criteria.

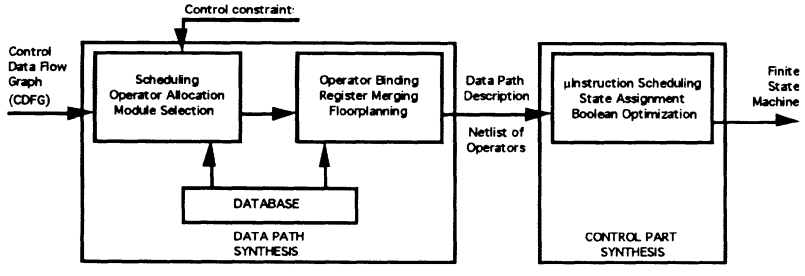


Figure 12. Phases of the synthesis process.

Previous work

Much work has been done on data path synthesis alone [Park86] and control part synthesis once the data path has been laid down [Nagle82]. The

techniques used for control part synthesis are based mainly on microcode optimization for microprogrammed controllers and on boolean optimization for finite state machine controllers. Taking into account earlier control constraints used by the high level synthesis process (such as the control part area or the number of control signals) would make possible a greater optimization of the control part.

Some attemps of integrating control considerations into data path synthesis goes back to ELF [Gircz84]. In this system, the optimization algorithm, which is based on graph grammar, uses a cost function with a control signals complexity measure to estimate control overhead, but only in terms of control interconnection complexity; its estimation of the real cost of the final control part design (ie surface, performance) is too rough. In Yorkstown Silicon Compiler [Camp87], structural synthesis is mainly oriented towards minimizing control states, but it is not clear how the data path constraints are handled. One of the most recent systems is CHIPPE [Brew90], which takes into account a number of constraints. Its underlying method is based on the iterative refinement of an architecture not targeted to a specific type of applications (unlike CATHEDRAL II [Goos88] or SYCO [Varin87]). Some control aspects are taken into account during scheduling, but they are limited to delay considerations.

Our approach

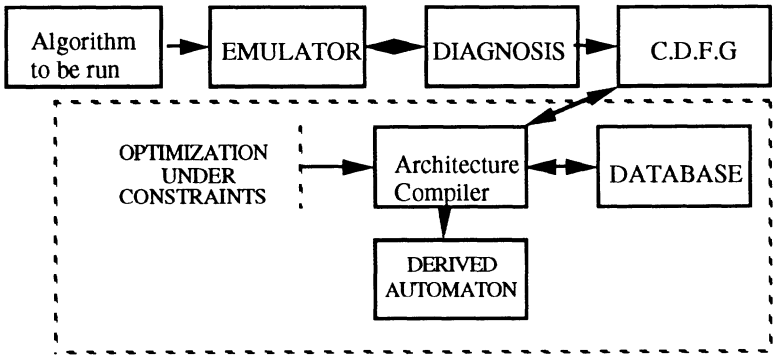


Figure 13. System Overview

Our approach [Verd87] has been to enable the integration of global control aspects (for example the total area of the hardware modules needed to implement the control part) by using the concept of **regularity** of a CDFG during scheduling. This measure is borrowed from a statistical estimation of the complexity (the diversity) of data transfers in the final design. Intuitively, if we consider that what is costly in terms of hardware resources are registers and multiplexors, a regular architecture will tend to use a few number of them and will use them all the time (at each cycle). Ultimately, if all resources were used all the time, control would be reduced to the selection of operations performed by the ALUs (multiplexing would disappear). We have been able to verify our conjecture that "the more regular a CDFG is, the simpler its

control part becomes"; meaning that the more regular the CDFG is, the less expensive (in term of multiplexed inputs and interconnected registers) the obtained solution becomes. Additionaly, we have identified a measure of "CDFG regularity" that can be injected into the operation scheduling process for better results.

Usually the cost function minimized by the scheduling process in isolation is set in terms of the number of operators (multipliers, adders, ...) for area computation and in terms of the number of machine-cycles for performance computation. With our approach, we have been able to show that we obtain several solutions with the same cost function in terms of area and performance. But these solutions, despite the fact that they get to the same area and level of performance, are still different because it is not exactly the same schedule. So, what does make the difference ? The difference is regularity! The cost of a scheduled graph is completely defined by its number of operators, number of machine-cycles and its regularity measure. In fact, when the scheduling process is performed, it acts directly on the regularity of the graph. We have therefore found the relevant features of a CDFG to "qualify" what we call "regularity" and to combine this information into a statistical variable. Taking into account such information about control complexity has several consequences that we will describe later.

In any way, it should be clear that by simplifying the control part of a design during the scheduling process, one gets a better starting point for the binding process between operators and hardware resources.

H.2 - SYSTEM DESCRIPTION

Some previous work at our laboratory on Data Path Synthesis [Saf91] led to the decomposition of this phase into two independent algorithms :

- Algorithm I simultaneously performs operation scheduling, operator allocation and module selection using a simulated annealing technique that we enhanced by a pseudo-deterministic control technique that searches in a "realistic" design space.
- Algorithm II is a global optimization algorithm which simultaneously performs operator binding and register merging, while taking into account interconnecting costs and floorplanning. The search for a good solution, which accounts for low level physical information (floorplanning), is made possible by this global optimization algorithm which also uses a simulated annealing technique improved by means of a stochastic technique that includes some notions of risk (area versus interconnection length).

The search in the design space is guided by the global minimization of a cost function that includes application constraints and system characteristics as follows:

- For algorithm I : Cost = F(surface, performance, power, ...)
- For algorithm II: Cost = F(floorplanning surface, number of registers, interconnect length, number of input multiplexors, ...)

Taking into account new constraints is made possible simply by adding them into the cost function being minimized.

H.3 - PROBLEM REPRESENTATION AND DEFINITIONS

The Optimization Algorithm

We present here the optimization algorithm used by the scheduling process. The reader not familiar with the Simulated Annealing process can refer to [Laar88]. Particularities of our simulated annealing algorithm are as follows:

- The architecture that we seek to synthesize is represented by a Control Data Flow Graph (CDFG) which is used as the initial solution to the algorithm. The random neighboring solution is obtained by applying a random move (transformation) on the current solution.
- These transformations concern local structures in the graph. They are:
  - Isolated nodes.
  - Branches which are sequences of successive nodes such that each node is directly dependent on the previous one belonging to an adjacent step (machine-cycle), and has at most one child node.
- Transformations are of two type:
  - Node or branch moves: A randomly selected node (or branch) is shifted-up or shifted-down. A shift-up (down) means that the node (or each node of a selected branch) is shifted from its own machine-cycle (step) to the one above (or below).
  - A module is selected in the database to implement an operation (this module differs from the preceding one by its performance, area, etc.). The database contains technology-independent descriptions of operators.
  - All these transformations respect data dependencies.
- The cost of a solution is a measure of its quality. The cost function is a weighted sum of the total area of the architecture (the number of modules multiplied by their area) and the total performance (the number of steps in the graph multiplied by the delay of the operators). Weighting of the various parameters allows the optimization to be biased in a particular direction in the design space (area or performance).
- The schedule of the temperature, determined empirically, is as follows: $T_k = T_{init} \cdot a^k$, where the parameter a depends on the size of the problem and k is the number of moves performed since the beginning.

The simulated annealing method is well suited for this problem (the transformation from a high level behavioral description to a structural one) due to the very large amount of data when image processing algorithms are considered (our system is mainly oriented towards this kind of applications), and to the complex engineering knowledge needed to explore the design space (the stochastic search performs a good optimization without complex rules or heuristics).

Definitions

To explain the originality of our approach, we first give some necessary definitions. A **scheduled graph** is an acyclic, directed and multistage graph where each node is assigned to a specific time slot. A Control Data Flow Graph (CDFG) is a scheduled graph. The only representation of the architecture we have is a CDFG. Presently the graph has to be connected (if the graph is not connected, we consider each sub-graph independently).

Any **node** of the CDFG represents a boolean or arithmetic operation. Such node has as many links as the number of operands in the underlying operation. The node type is identical to the one of the operation. During scheduling, each node is temporarily assigned to a specific machine-cycle or step.

An **arc** (or **link**) of the CDFG represents an interconnection network (bus, multiplexors,...) allowing data transfers between different operators. The arc length between two nodes in the CDFG is defined as the number of machine-cycles between these two nodes.

A **motif** in the graph is an item representing both the length of an arc and its associated two nodes (the source and the destination respectively) (see an example in figure 14). Our data path model is a register-mux-operator model so each motif reflects a data transfer between two operations.

The motifs are classified according to their type and value. The type of a motif is given by the one of the source and destination nodes for an associated arc. Its value is the distance (or length in terms of the number of machine-cycles) between the two nodes. We will use the following notation: L^{i}_{tstd} is the **length of the motif** i, where t_s is the source type and t_d the destination type.

The **regularity** of a scheduled graph is computed from a statistical measure of the diversity of its motifs. It is the inverse of diversity, where diversity is a weighted sum over all standard deviation of the motifs.

Mean of motifs lengths of type $_{tstd}$:

$$m_{tstd} = S^{tstd}_{i=1} (L^{i}_{tstd}) / N_{tstd}$$

Standard deviation of motifs lengths of type $_{tstd}$:

$$s_{tstd}^2 = S^{Ntstd}_{i=1} (L^{i}_{tstd})^2 / N_{tstd} - (m_{tstd})^2$$

The graph regularity is given by:

$$R = 1 / (S^{i\Œ ts, j\Œ td} (s_{ij} P_i P_j))$$

These weights (P_i & P_j) increase the minimization of the diversity of data transfers between the more frequently used operations in the graph.

Let us illustrate these computations through some simple examples. In the figure below, the lengths of the two motifs are: $L^{z}_{+x} = 2$ and $L^{x}_{++} = 1$.

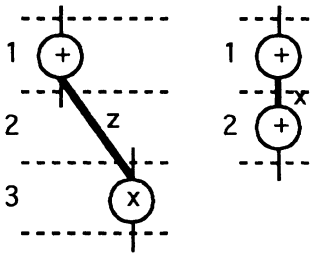


Figure 14. Two motifs "+x" (left) and "++" (right)

Solution

Our approach is based on the use of a simple heuristic: the more regular a sheduled graph is (in terms of its data transfers and the number of operations in a given machine-cycle), the simpler its associated control (in terms of area, number of control signals) is. One can find an illustration of this conjecture with systolic architectures where the control part is reduced to a simple clock distribution. This is illustrated by a simple case (figure 15) where it is easy to see the impact of increased regularity on the control and data path complexity. The increase of the interconnectivity (bus, registers, multiplexors) in the irregular example (the less regular graph) leads to an increase of the control complexity (more control signals) and area.

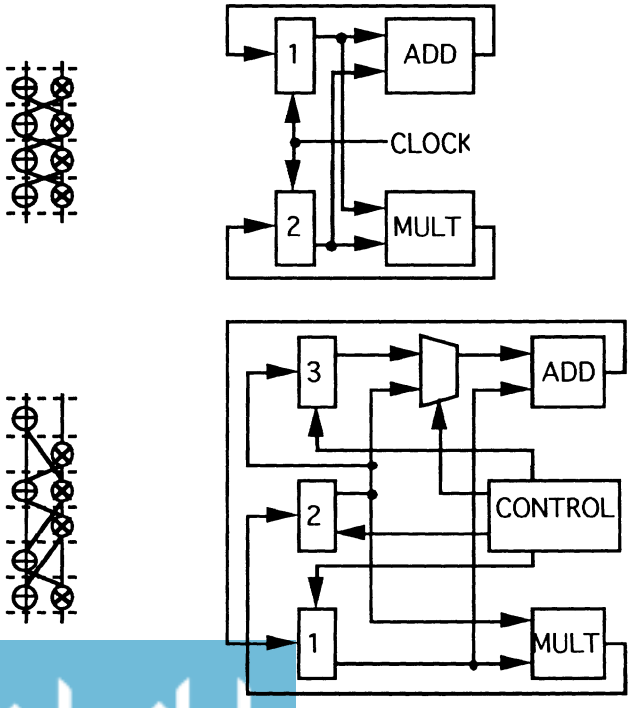


Figure 15. Regular (top) and irregular (bottom) graphs with their corresponding implementations.

Notice that, as mentioned in [Paul89b], the configuration with the lowest global cost is not necessarily the one with the minimum number of registers. Therefore, in the scheduling process, measuring only the number of registers is irrelevant for having an indication on control complexity. And it seems difficult, to say the least, to know the number of multiplexors prior to the binding process (before the assignment of registers and operators).

The potential regularity of a scheduled graph depends on the topology or nature of the algorithm to be implemented. Thus the regularity that can be obtained with a particular graph is bounded to some fixed value.

The problem is to find a way to quantify the regularity of a scheduled graph. The solution we have selected consists of decomposing a graph into a set of motifs, and to make a statistical analysis of these motifs. The advantages of such an approach are as follows:

- A motif gives an indication about the number of data transfers between two nodes - thus between two corresponding operators (data transfers are the most costly in terms of needed control signals).
- The use of statistics inherently compresses data; Whence easier handling of a large amount of characteristics needed in these processes and computation speed up.

These "motifs" are being typed and any type is taken from a bounded set (number of different operations = T) so we have T^2 types of motifs where the standard deviation has to be expressed. Moreover, the total number of arcs in the graph is well known and constant (it never appears or disappears a data transfer during the scheduling process) so it is possible to use classical statistical measures (such as mean and standard deviation) in order to evaluate the graph regularity. For each type-set of motif, we compute a standard deviation of the values of each motif that indicates the global regularity of the graph.

H.5 - RESULTS

Scheduling Process

We list below some of the results we have obtained when applying our method to the fifth order wave filter example. Starting with the same constraints concerning data path area and performance, we obtained two different solutions with the same number of operators and machine cycles but with different regularities; Indeed one of these solutions was found by a scheduling process taking into account the regularity constraint while the other one was found without any regularity constraint. The scheduled graphs (figure 16) have the same basic characteristics: number of adders: 3, number of multipliers: 2, number of machine cycles: 16. The respectives computed regularities (the higher the number is, the more regular the graph is) are:

- For the more regular : Regularity = 1 / 33
- For the other graph : Regularity = 1 / 67.3

The CPU time required for the scheduling process was about 1 min 50 sec. on a Sun SparcStation 2 with or without regularity computation.

Binding Process

Our target structure for the control part of the architecture is a simple microprogrammed controller. The columns in the instruction memory represent control signals for each module in the data-path (adders and multipliers if needed, registers, multiplexors) while each line represents a machine cycle. The estimation of the control complexity (control cost) is nothing but the instruction memory area (total number of bits in the memory). We choose this control structure because of its simplicity since our goal is not to implement the actual control part synthesis but to take into account control constraints during the data-path synthesis. Including control structures such as branching and sequence breaks do not alter the method since in this case only the number of motifs is modified.

	Control area	Mux Inputs	Registers	Iterations	CPU time
Regular	728	43.2	9	3830	41.0sec
Irregular	934	51.8	11	4058	45.5 sec
Gain	22%	16.6%	18%	5.6%	10%

It is clear that taking into account the regularity of the data-flow graph during scheduling allow the binding process to converge towards a better solution.

H.6 - PERSPECTIVES

The relevance of our proposed method is anticipation: indeed, with this method, the impact of scheduling on control part synthesis is taken into account even before the binding process for data path synthesis is performed. This anticipation has been made possible through the use of statistics that measure the global characteristics of regularity for a scheduled graph, i.e., the CDFG.

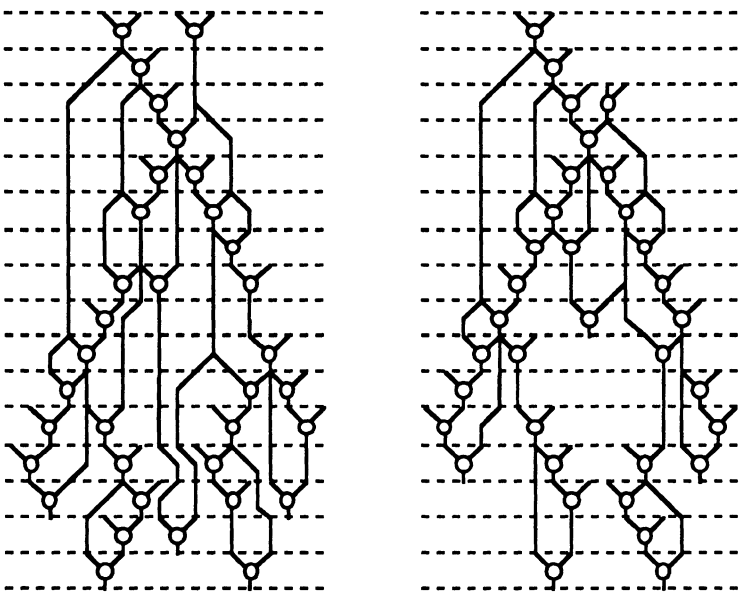


Figure 16. The irregular scheduled graph (left) and the regular one (right)

On a realistic enough example such as the fifth order wave digital filter, we have obtained a significant difference between the regularity of the two scheduled graphs (for exactly the same number of operators: 3 adders, 2 multipliers) and a decrease of 22 per cent of the control part area. Therefore, we have shown that at this level of synthesis (the scheduling phase), we cannot neglect this regularity measure.

The loss of "distinguishability" due to the use of statistics presents no drawbacks, since at this level of the synthesis only global characteristics are needed. In fact, handling more precise information related to control complexity (for instance doing the binding and the scheduling process simultaneously) would be tedious and would not improve design quality. This especially true when one thinks of the other issues such as partitioning, optimal number of clock phases, multicycling and chaining of operators, etc., not to mention testability, fault tolerance and correctness (formal methods, simulation, ...).

Our goal is not to design the control part while performing the data path part, but to try to subject the data path to the control constraints. In doing so, we think we have found a way for a better solution to the actual design of the control part. One possible extension of this proposed method is to facilitate the selection of a control implementation model based on regularity characteristics determined from the very beginning (i.e. the scheduled graph or CDFG). These rules may define the controller type, and this can be done before the run of algorithm II (binding process and floorplanning) for data path synthesis. Another extension is to use the CDFG regularity to estimate the number of product-terms of a PLA-based controller during the scheduling

phase. Other clues indicated in [Mal85] about the performance and testability of many different controller implementations can be added.

We have found a way, in addition to avoid simultaneously performing the binding process with scheduling, by adding a "regularity measure" to the cost function, to obtain a scheduled graph as good as the one we would have obtained while performing simultaneously the binding process and the scheduling. We achieved a better understanding of what improves a given scheduled graph in terms of control complexity and how the binding process affects scheduling. In sum, the added term of regularity measure minimizes the control complexity (through data transfer dispersion) as much as the binding process does, but with far less computational complexity.

I - CONCLUSION

In this analysis, we have tried to show that the complexity of future systems demands that new and radical methods be defined to allow for efficient designing and testing. We have presented an approach that provides an equivalence between the specification of an algorithm and its implementation on a particular architecture: if an algorithm can be specified, then it can be implemented, and vice versa. We have chosen a Functional Programming formalism as a means to express computations that must be performed, and a Data Flow execution model to implement the program. The specification phase is linked closely to the emulation of an algorithm to prototype and validate its functionality; after all, the only way to make sure the solution can be implemented in hardware is to build one. This is quite different from simulating the solution as one is never quite sure the physical system will behave identically to the simulated one. For complex systems, simulation can be quite time consuming and severely limits the number of iterations that can be performed to debug the system. On the other side, emulation at normal operative speed, provides an instant feedback that generates a truly interactive debugging environment. Another side benefits of the equivalence between specification and hardware implementability is that, as a programer, you optimize simultaneously and transparently the algorithmic solution and its implementation on a target machine. This is a departure from classical approaches where one designs an algorithm independently from the team that will adapt it to the target machine. It is clear that this way of doing is both wasteful in human resources (two teams instead of one) but also in terms of algorithmics efficiency as the original thinking that went into the design of the algorithm and that lead to a carefuly organized solution, is completely lost. Let us also add that the real time prototyping of an algorithm can be economically beneficial if one realizes the solution does not perform as was expected; the ability to wastefully experiment provides a truly grandiose dimension to ensuring that the seeked solution is really what is needed.

Once the programmer is satisfied that the algorithm solves his problem and that at least one hardware solution exists, the building of a target system that matches physical constraints is done automatically by using a description, provided by a diagnostic module, of resources actually used during the execution. In so doing, we separate constraints associated with a problem into two types: those that deal with the functionality of the solution (emulation),

and those that deal with the final system implementing the solution (automatic generation of target system). The diagnostic phase is by far the most difficult one to handle. Typically the emulator will be built from general modules offering multiple functionalities. At execution time some, but not all, of its functionalities will be used (ex: add with an ALU). So it is clear that, to avoid wasteful implementations, the ability to match the constraints of the target system will be facilitated by the fact we will try to integrate only needed resources. If in the general case it is very difficult to separate resources used for the purpose of emulation from those truly needed by the solution, we have shown that the diagnostic phase can in fact be eliminated with the proper specification formalism: a dependence data flow graph. With such a graph, two approaches can be used to generate a target architecture: compilation, where, in a traditionnal manner, one integrates all resources specified in the graph, and synthesis, where one manipulates the graph (addition / removal of operations) to obtain the most efficient architecture.

Today, we are validating our approach with a massively parallel emulator, the Fonctional Calculator, which is made of 1024 identical data flow processors. We have emulated many algorithms, from traditional low level convolution filters to high level target tracking techniques. Of all phases, only the synthesis phase remains to be validated with the construction of one or several VLSI circuits. We have selected a "reflex operator" that detects dominant line segments inside an image as an application example. It has already been emulated on the Fonctional Calculator (3 weeks of effort using the standard library) and we have therefore verified that it is functionally correct; its construction in the form of some VLSI can now be considered positively.

Let us finish by saying that when the project was started, we anticipated a radicaly new way on conceiving and constructing machines. Although we have not yet reached our ultimate goal, we have shown that our approach is valid and will lead very soon to our first image processing hardwired operator, automaticaly built from specification. Such an approach is, we believe, the only one that can lead quickly to the validation and construction of complex new designs meeting stringent realization constraints.

BIBLIOGRAPHY

[Ack79] W. B. Ackerman: "Data flow languages", AFICS conf. proc., vol. 48, 1979 NCC, New York, june 1979.

[Allart88] E. Allart, B. Zavidovique: "Image processing VLSI design through functional match between algorithm and architecture", 1988 IEEE International symposium on circuits and systems - Espoo, Finlande - 7 / 9 juin 1988.

[Ama87] R. Amann, U.G. Baitinger: "New State Assignment Algorithms for Finite State Machines Using Counters and Multiple-PLA/ROM Structures", IEEE Intl. Conference on Computer Aided Design, Nov. 1987.

[Anders65] J.P. Anderson : "Program structures for parallel processing", com. on ACM 8, 12, pp 786-788, December 1965.

[Arv86] K.P. Arvind, D.E. CULLER: "DataFlow Architectures". Annual Review in Computer Science, Vol 1, Palo Alto CA, Annual Reviews Inc, 1986.
[Back78] J. Backus: "Can Programming be liberated from the Von Newman Style? A Functional style and its Algebra of Programs".Com.of ACM, volume 21, number 8, August 1978.
[Berg87] C. Berge. "Hypergraphes combinatoire des ensembles finis" - Gauthier-Villars BORDAS - Paris 1987
[Berns66] A.J. Bernstein ."Analysis of Programs for Parallel Processing", IEEE. Trans. Computers, Vol. EC-15, No 5,1966.
[Brew90] F. Brewer, D. Gajski: "CHIPPE: A System for Constraint Driven Behavioral Synthesis". IEEE Trans. on CAD, Vol. 9, No. 7,1990.
[Camp87] R. Camposano, J.T.J Eijndhoven: "Combined Synthesis of Control Logic and Data Path". IEEE ICCAD, 1987.
[Coster85] M. Coster, J.L. Cherman. "Précis d'analyse d'images" 1985 - Editions du CNRS
[Curtis88] L. Curtis Widdoes and Holly Stump. "HARDWARE MODELING." VLSI System Design , pp. 30-38, July 1988.
[Dan81] P. E. Danielson and S. Levialdi. "Computer architectures for pictorial information systems". IEEE Computer, pp. 53--67, Nov. 1981.
[Davis82] A.L. Davis, R.M. KEller: "DataFlow Program Graphs". IEEE Computer, Feb 82.
[Denn80] J. B. Dennis: "Data flow supercomputers". Computer, Vol 13, Nov 80
[Duff81] M. J. B. Duff and S. Levialdi. "Languages and architectures for image processing". Ed. by the authors, Academic Press 1981.
[Ecch86] M. Eccher and B. Zavidovique. "COLISE real-time region detector based on algorithmic decomposition". 20th ASILOMAR Conf. on Signals, Systems and Computers. Nov. 1986, Pacific Grove CA, USA.
[Ecch92] M. Eccher. "Architecture parallèle dédiée à l'etude d'automates de vision en temps réel". Ph.D Thesis. Univ. of Besançon. 1992
[Flynn72] M. J. Flynn. "Some Computer Organisations and their Effectiveness", IEEE Trans. on Computers, Vol. 21, No 9, pp 948-960, 1972.
[Gajs82] D.D. Gajski, D.A. Padua, D. Kuck: "A second opinion on data flow machines and languages", Computer, February 1982.
[Gall86] E. Gallesio, V. Serfaty, L. Bol and B. Zavidovique. SPIE86 - Applic.of AI 4 - vol, n°657 pp 26-33 . April 1986. Innsbruck, Autriche.
[Gircz84] E.F. Girczyc, J.P Knight: "An Ada to Standard Cell Hardware Compiler based on Graph Grammars and Scheduling". IEEE ICCD, 1984.
[Goos88] G. Goosens et al.: "Optimization Based Synthesis of Multiprocessor chips for Digital Signal Processing with CATHEDRAL II". Intl. Workshop on Logic and Architecture Synthesis for Silicon Compilers, Grenoble, FRANCE 1988.
[Jerr86] A.Jerraya, P.Varinot , R.Jamier and B.Courtois, "Principles of the SYCO. Compiler", 23rd DAC 1986 p 715.
[Kuck77] D.J Kuck, D.H Lawrie and A.Sameh: "High Speed Computer and Algorithms organization". Academic Press. 1977
[Laar88] P.J.M. Laarhoven, E.H.L. Arts: "Simulated Annealing: Theory and Applications". D. Reidel Publishing Company, 1988.

[Lands80] D. Landskov, S. Davidson, B. Shriver: "Local Microcode Compaction Techniques". Computing Surveys, Vol. 12, No. 3, Sept. 1980.
[Mal85] Y.K. Malaiya. "Options in Control Implementation". IEEE ICCD, Oct.1985.
[Min86] M. Minoux, G. Bartnik. "Graphes, algorithmes, logiciels - Dunof informatique" - Bordas Paris 1986 - ISBN 2-04-016470-1
[Mong85] C .Mongenet, "Une méthode de conception d'algorithmes systoliques, résultats théoriques et réalisation", These, INPL, Mai 1985.
[Nagi78] M. Nagi. "A tutorial and bibliographical survey on graphs grammars" - International Workshop on graphs, grammars and their applications to computer Science and Biology - Oct/Nov 78 Bad Honneff - West Germany
[Nagle82] A.W. Nagle, R. Cloutier, A.C. Parker: "Synthesis of Hardware for the Control of Digital Systems". IEEE Trans. on Computer-Aided Design, Vol. 1, No. 4, Oct. 1982.
[Offen85] R. J. Offen. "VLSI Image Processing". Collins Professional Technical Books. 1985, U.K.
[Paris89] N. Paris. "Développement d'outils de conception de circuits intégrés et application à la réalisation d'une architecture de visualisation" - Université Paris XI - Orsay 29 Mai 1989
[Park86] A.C. Parker, J. Pizaro, M. Mlinar: "MAHA: A Program for Data Path Synthesis". 23th ACM/IEEE Design Automation Conference; 1986.
[Paul89a] P.G. Paulin: "Horizontal Partitioning of PLA-based Finite State Machines". 26th ACM/IEEE Design Automation Conference, 1989.
[Paul89b] P.G. Paulin: "Algorithms for High Level Synthesis with Area and Interconnect constraints". EURO ASIC'89, Grenoble, FRANCE, 1989.
[Pope84] S. Pope, J. Rabeay and R.W. Brodersen, "Automated design of signal processors using macrocells," in VLSI Signal Processing, IEEE Press, NY.
[Quenot91] G.M. Quenot, B. Zavidovique: "A Data-Flow Processor for real Time Inage Processing". IEEE Custom Integrated Circuits Conference, May 1991, San Diego CAL
[Quenot92] G. M. Quenot, B. Zavidovique: "The ETCA Massively Parallel Data-flow Computer for Real time Image Processing". To appear in IEEE Intl Conf on Computer Design, 11/14 Oct 92, Cambridge MA.
[Rosen76] A. Rosenfeld, A.C. Kak. "Digital Image Processing" - Academic Press 1976 - Duff and Levialdi "Languages and Architectures for image processing"
[Rueh84] A.E. Ruehli, G. Ditlow. "Circuit analysis, logic simulation, Design verification for VLSI" - Invited paper p 183 VLSI Technology and Design O.G. Folberth, W.D. Grobman - IEEE Press 1984
[Saf87] A. Safir, P. Kajfasz and B. Zavidovique. " 'Fresh Connection' a self reconfigurable network with associative capabilities" 21st Annual Asilomar Conference on Signals, Systems and Computers, November 1987.
[Saf91] A. Safir, B. Zavidovique. "A Solution to The High Level Synthesis Problem". Intl. Journal of Computer Aided VLSI Design. Vol. 3, No. 1, 1991.
[Serf85] V. Serfaty-Dutron and B. Zavidovique. "Programming Facilities in Image Processing", Compint *85*, pp. 804-806, Sept. 1985, Montreal, CANADA.

[Serot91] J. Serot, G. Quenot: "Real Time Image Processing using Functional Programing on a Data-flow Architecture". Intl Workshop on Computer Architecture and Machine Perception (CAMP91), Dec 1991.
[Siva82] S.C. Siva, J.A. Covington. "Modular description, Simulations, Synthesis usin g DDL" - 19th Design Automation Conference, Paper 21.2 - ACM-IEEE Caesars Palace - Juin 82, LAS VEGAS, NA
[South83] J.R Southard: "MacPitts: An Approach to Silicon Compilation", Computer , Dec. 1983.
[Varin87] P. Varinot: "Compilation de Silicium: Application à la Compilation de Partie Contrôle". PhD. Thesis, Institut National Polytechnique, Grenoble, FRANCE 1987.
[Vegd84] S.R. Vegdahl: "A survey of proposed architectures for the execution of functional languages", IEEE Trans Computers 33(12), 1984.
[Verd92] F. Verdier, A. Safir, B. Zavidovique: "A High Level Synthesis Algorithm Including Control Constraints". 8th EUROMICRO Conference, 14/17 Sept. 1992, Paris, FRANCE.
[Villa90] T. Villa, A. Sangiovanni-Vincentelli: "NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementation". IEEE Trans. on Computer-Aided Design, Vol. 9, No. 9, Sept. 1990.
[Widd88] L. C. Widdoes Jr, H. Strump. "Hardware modelling" - VLSI systems Design July 1988
[Will82] J.H. Williams: "On the development of the algebra of functional programs", ACM transactions on programming languages and systems, Volume 4, N°4, october 1982.
[Wolf86] Wayne Wolf. "An Object-Oriented, Procedural Database for VLSI Chip Planning", 23rd DAC 1986.
[ZSF91] Bertrand Zavidovique, Véronique Serfaty, Christian Fortunel: "A mechanism to capture expertise". IEEE Software, Nov. 1991, pp 37-50.

9

ARCHITECTURES AND BUILDING BLOCKS FOR DATA STREAM DSP PROCESSORS

G.A. Jullien, Director

VLSI Research Group, University of Windsor

Ontario, Canada N9B 3P4

INTRODUCTION

This chapter is concerned with building real-time digital signal processing systems for high throughput, data stream processors. These systems push silicon technology towards its limits, and the architectures are typically massively parallel, exploiting heavy pipelining. The highest throughput will be reached when each output bit is connected to a pipeline latch. These arrays are often referred to as bit-level systolic arrays [1, 2]. Our target applications include real-time video processing and radar return processing, but many other applications can be found in such diverse fields as machine vision and communications systems.

There is a major difference between these architectures and the current architectures of DSP chips. DSP chips are essentially outgrowths of the early micro-controllers, in which a standard Von-Neuman or Harvard architecture core contains a single (or a few) general purpose ALU's containing fast multipliers. In this case, fast implies a low critical path between the input and output data of the multiplier. In bit-level pipelined systems, the important requirement is to match the data rate of the input stream to the clock rate of the bit-level latches. We are able to control the complexity of the switching circuitry driving each latched bit in order to match it to the required

data rate. This is not normally a concern with high speed designs, i.e. the circuit is designed to run as fast as possible and then it is applied to the application. Recent results, for example, have shown that even mature CMOS technologies are capable of synchronous pipelined arithmetic rates in the hundreds of MHz range [3]. Such techniques have recently been extended to asynchronous high speed CMOS architectures [4] with quoted speeds in the same range. In general purpose computational systems, there is normally a leveling factor based on the need to synchronize (either with clocks, or hand-shaking) disparate computational elements, such as those found in current DSP chips. For data stream architectures, however, that can operate as systolic arrays, the synchronizing requirement is very straight forward; the disadvantage is the limited use for such special purpose architectures. With the advent of ASIC technology, silicon foundries, and the wide spread use of advanced software for fast custom design, it is quite possible to consider the use of such special systems for even small production runs. Data stream high throughput DSP systems are such a target group.

This chapter is concerned with constructing integrated systems for arithmetic intensive digital signal processing (e.g. linear filtering), using bit-level systolic array concepts [1, 5]. In such architectures, the internal pipeline rates are matched to the signal data rates, and every bit (or group of bits) is pipelined. This is in contrast to the design of cascaded combinational circuits for arithmetic processors, where the goal is to reduce the critical path time through the cascade [6]. In particular, we will show that judicious use of new number theoretic mapping techniques can improve the synchronizing problem, by allowing most of the calculations to proceed as bit-level independent data streams. We will also examine the use of such a design approach to two other computational problems; in fact, our starting point will be a simple bit-level pipelined adder, and we will also consider the use of these techniques in the design of redundant arithmetic units for general purpose DSP ALU operation.

With regard to speed requirements, data throughput rates are dependent upon the signal bandwidth. For example, audio and modem data transmission rates are in the range of tens to hundreds of KHz and standard video and some radar system data rates are in the tens of MHz range. We see that the throughput rate for systolic solutions in these application areas are at least an order of magnitude lower than the speeds recently reported for mature CMOS technologies. Even uncompressed HDTV data rates (in the range of 100MHz) are a factor of 5 lower than reported speeds [3]. It is

therefore useful to consider trading off speed for greater functionality within each pipeline stage, and reaping the benefits of reduced area and power consumption. This chapter, therefore, also explores these possibilities, by combining a TSPC pipeline [7] with dense multiple output NFET blocks based on minimized binary trees; we term such blocks *switching trees* [8]. We also present preliminary fabrication results and techniques for on-the-fly module generation of the bit-level blocks.

DATA STREAM PROCESSING

Real-time DSP computation is normally performed on data streams. The data is fed to the processor one sample at a time, at the natural data rate of the generating system. An example might be the processing of video data from a CCD camera. If the camera has a sensor containing 10^6 pixels, and produces 10 complete frame scans every second, then the natural output of an analog to digital converter, connected to the CCD analog output, is 10M samples per second. If the DSP processing algorithm is an image enhancement filters that retains the original image size, then the output rate of the processor is the same as the input rate. If we are extracting information from the data (e.g. defects in the image) then the output rate may be much lower than the data rate. Other examples of different input and output rate are in interpolation and decimation procedures. An example of an interpolation procedure is the generation of a high rate data from the 44.1KHz sample rate used to store audio signals in compact disc format. For this application we interpolate up to a much higher data rate (say 16 times oversampling) prior to analog conversion, in order to reduce the complexity of the analog filter used to recover the base-band audio signal. The precision, repeatability, and mathematical flexibility of a digital system is to be preferred over an equivalent analog system (an anti-aliasing filter with a sharp cut-off at about 20KHz).

In this chapter we will assume that the processing system operates at the same input and output data rate. In this case we can use bit-level systolic arrays as our basic architectural tool, where processors operate at the individual bit-level and each output bit is latched.

DSP CORE VS SYSTOLIC ARRAY

In order to examine the fundamental difference in building systems with programmable rather than fixed array architectures, we will take

the example of two architectures used to build similar DSP one-dimensional FIR filters.

Our starting point is a DSP core [9] that is used as a building block within an ASIC (Application Specific Integrated Circuit) chip. A block diagram of a typical core is shown in Fig. 1. This particular core has been presented by Baji [10], more recent examples of complete chips also use similar architectural styles (e.g. [11]).

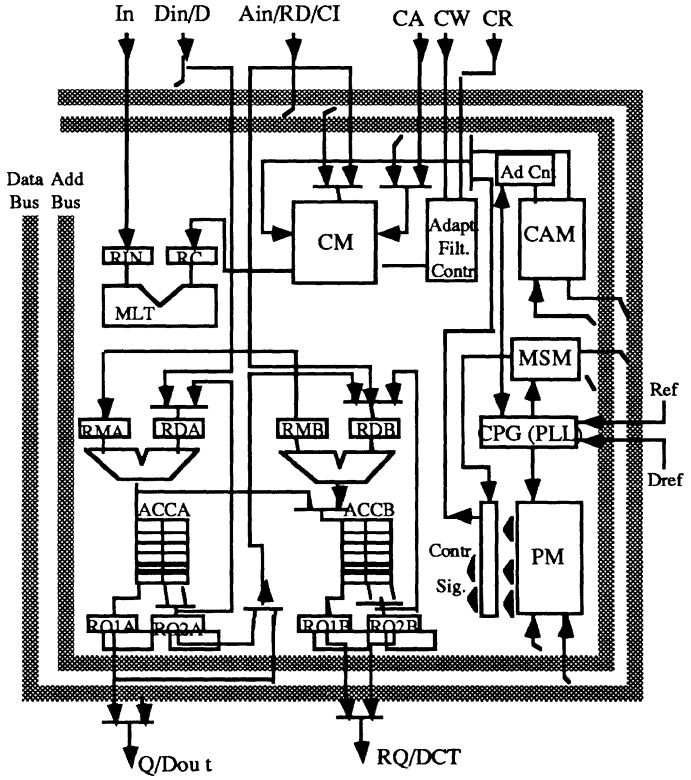
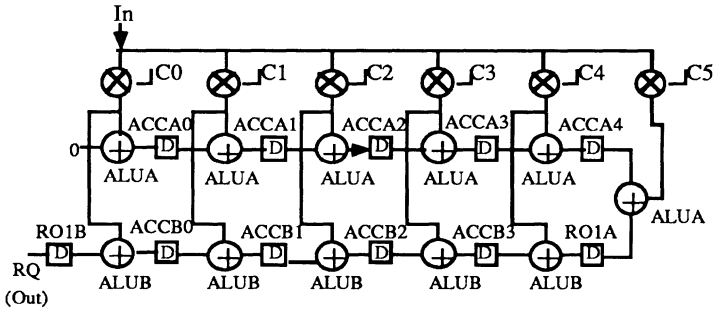


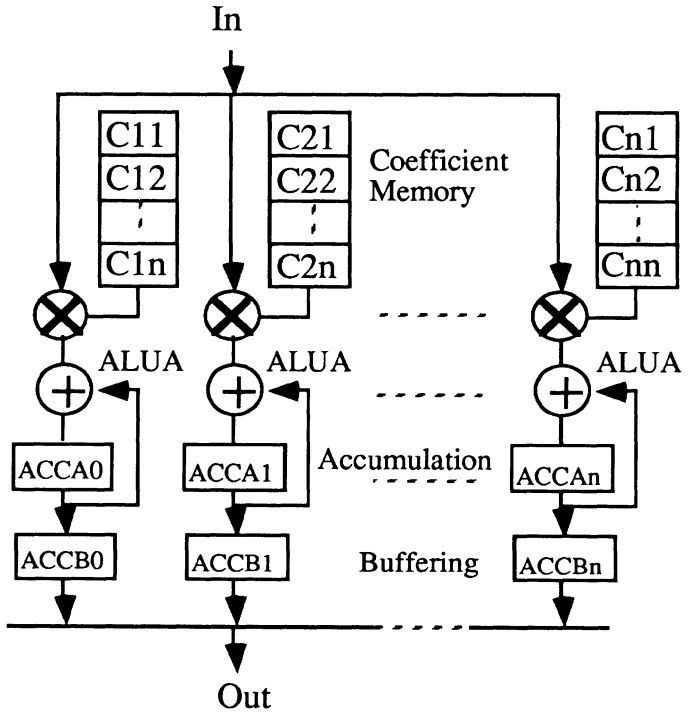
Fig. 1 DSP Core

This core processor has a reconfigurable high speed data path supporting several multiply/accumulate functions including 16-tap linear-phase transversal filtering, high-speed adaptive filtering, and high speed discrete cosine transform (DCT). The DSP core consists of an 8x8 modified Booth parallel multiplier (MLT), two 12-bit arithmetic logic units (ALUA and ALUB), two sets of 2x8x12-bit accumulator arrays (ACCA and ACCB), 8-bit x 16-word coefficient memory (CM), and a 4-bit x 64-word coefficient address memory (CAM). A programmable CMOS phase-locked loop circuit is

provided as a clock pulse generator for high speed operation. It has a 40-bit x 8-word microprogram memory. To implement a linear-phase FIR filter, the core will be configured as shown in Fig. 2a. One tap of the filter is processed in 10 ns. The core will be configured as shown in Fig. 2b to implement a discrete cosine transform (DCT), one point DCT will be processed in 160 ns.



(a)



(b)

Fig. 2 DSP Core configured as (a) FIR filter; (b) DCT

It is clear that the limited resources of the core unit are re-used several times within each computational cycle. Even if we build sufficient processors to have dedicated hardware for each multiplier and adder in the filter or DCT architecture, the data will still have to be routed into the arithmetic units via the bus structure. Typically, the ALU will be built to accommodate a suitable wordlength for a variety of problems that may be programmed into the core; this may be overkill, however, for some problems. A typical use of such a core is in a shared resource mode, in which arithmetic and register resources are allocated according to some suitable strategy [12]. Although this represents a flexible solution (we may program many different algorithms onto a single generic structure, it does not represent the optimal mapping onto a specific algorithm. The very fact that we have bus connections to allow the sharing of resources, flies in the face of an algorithm that maps into a locally connected systolic array.

The buses that are used to move data around the chip, and the connected registers and processing elements have a combined large parasitic capacitance that has to receive charge from the power supply in order to register two complete changes of state (e.g. '0'→'1'→'0'). This charge transfer translates into power dissipation as $C_{bus}V_{DD}^2 f$, where C_{bus} is the total bus and connected modules parasitic capacitance, V_{DD} is the power supply voltage and f is the frequency at which this transition occurs. In large systems, the bus capacitance can be many tens (if not hundreds) of pico farads. For a 100 pF bus experiencing a logic cycle (that completely charges and discharges the bus capacitance) every 20ns on average, the power dissipation is 125mW (the power can be reduced by using low voltage swings on the bus and using sense amplifiers on each connected module to detect the changes).

This dissipation occurs every time there is a complete logic transition between two modules connected to the bus. Contrast this to the situation where the connections between modules can be hardwired and the modules can be placed adjacent to each other on the chip. For the same equivalent speed of operation we can a) slow down the rate of logic transfer (many connections are replacing a single bus) and b) the parasitic capacitance is drastically reduced. These two effects taken together can considerably reduce the power dissipation during a complete logic transition.

An example of a limited, but much more efficient architecture is the bit-level systolic array correlator shown in Fig. 3 . The array operates with single bit coefficients, but may be expanded to operate on multi-

bit words. The individual cell is basically a gated full adder with the functionality shown in eqn. (1).

$$\begin{aligned} x_{out} &= x_{in} \\ a_{out} &= a_{in} \\ y_{out} &= y_{in} \oplus (a_{in} \wedge x_{in}) \oplus c_{in} \\ c_{out} &= (y_{in} \wedge c_{in}) \vee (y_{in} \wedge a_{in} \wedge x_{in}) \vee (c_{in} \wedge a_{in} \wedge x_{in}) \end{aligned} \tag{1}$$

The latches are used to align the data wavefront in order that the array computes the bit-level correlation of eqn. (2).

$$\begin{aligned} y_j &= \sum_{i=0}^{N-1} a_i \wedge x_{j+i} \\ j &\in \{0,1,2,\ldots,N-1\} \end{aligned} \tag{2}$$

The systolic array is based on a recursive definition of eqn. (2) that produces the structure shown in Fig. 3 (a).

The complete recursive computation is shown in eqn. (3).

$$\begin{aligned} \psi_j^{[i][0]} &= 0 \\ \psi_j^{[i][k+1]} &= \psi_j^{[i][k]} + \left(a_i \wedge \left[x_{j+i}^{[k]} \cdot 2^k\right]\right) \\ k &\in \{0,1,2,\ldots,B-1\} \\ y_j^{[0]} &= 0 \\ y_j^{[i+1]} &= y_j^{[i]} + \psi_j^{[i][B]} \\ y_j &= y_j^{[N1]} \\ j &\in \{0,1,2,\ldots,N-1\} \end{aligned} \tag{3}$$

We see that the implementation consists of repeating the same basic cell (a simple gated full adder) and constructing an array that observes the recursive decomposition of the original algorithm. In terms of silicon metrics: we have removed the bus structure by removing the flexibility of being able to program the architecture to implement different algorithms; we have reduced the parasitic capacitance connected to the data nets between modules; we have replaced the registers in the DSP core by distributed single-bit latches throughout the chip. The throughput rate is now dependent upon the rate at which a gated full adder can generate stable latched output bits.

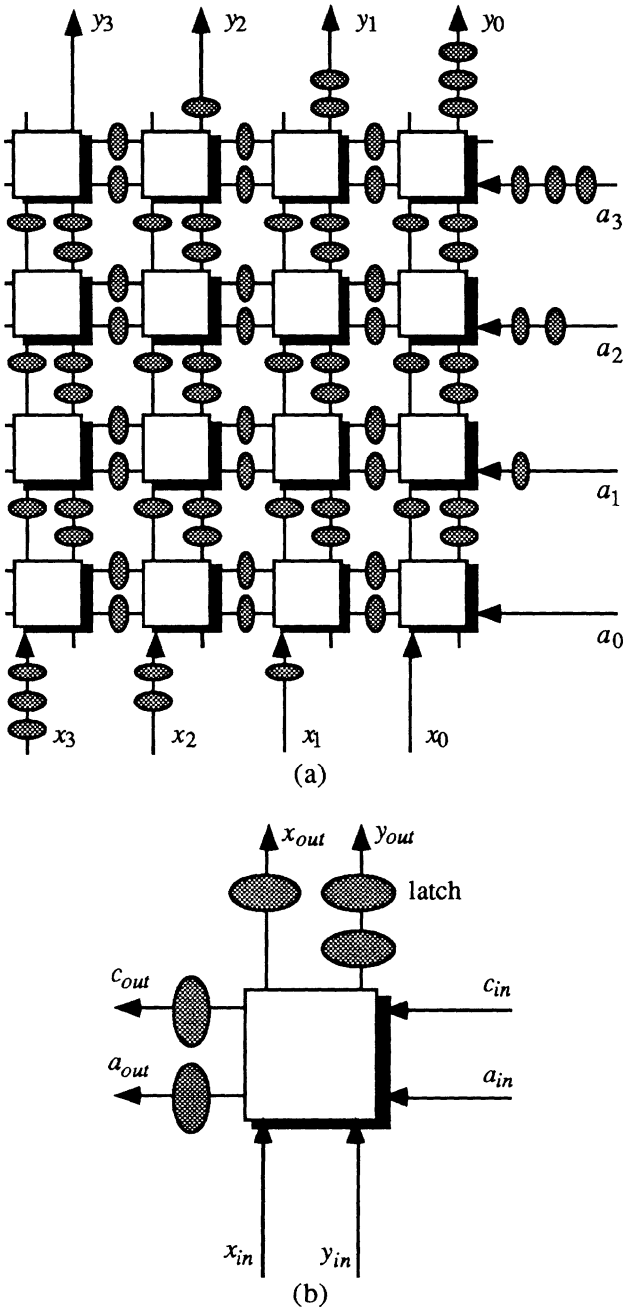


Fig. 3 Bit-level Systolic Correlator (a) 4×4 array (b) cell

This will be considerably greater than the throughput rate afforded by the DSP core.

In reducing the capacitance of a single net (i.e. a bit line on one of the buses) we have also reduced the current 'spike' that will occur as the bus is charged. This current is $C_{bus} \frac{dv_{bus}}{dt}$ and can be many tens of milliamps for large capacitance loads. If we take the value of a 100pF bus load and a 1v/ns switching edge, the current is 100mA. We will, of course, obtain large total charging current values for the systolic array, but there is a chance of skewing the clocks to even out the charging current. This need only be over a few nanoseconds; not sufficient to slow down the array because of timing races.

ARCHITECTURE AND ARITHMETIC

The systolic correlator uses full adders implementing 2's complement binary arithmetic. The bit-level decomposition of algorithms involving massive arithmetic computations, however, will change considerably based on the form of number representation and corresponding arithmetic operations required.

We give the following two examples of architectures based on different arithmetic (and representation schemes). The first is an architecture using a form of redundant arithmetic; the second is an architecture using modulo arithmetic.

DSP ALU Using Redundant Arithmetic

As an example of the synergism associated with architectures and arithmetic, we briefly review the structure of a DSP ALU using redundant arithmetic [13]. In the application discussed here, the bit parallel systolic concept has been used to perform combined multiply-accumulate, divide and square root. The circuit is highly regular, requires only minimal control and can be reconfigured on each cycle. The execution time for each operation is the same. The combination of redundancy and pipelining results in a throughput independent of the wordsize of the array.

The radix 2 SRT division method and the analogous square root algorithm have been used as the basis of the architecture. The use of a redundant number system limits the carry propagation to adjacent cells and allows most significant redundant digit (MSD) first computation.

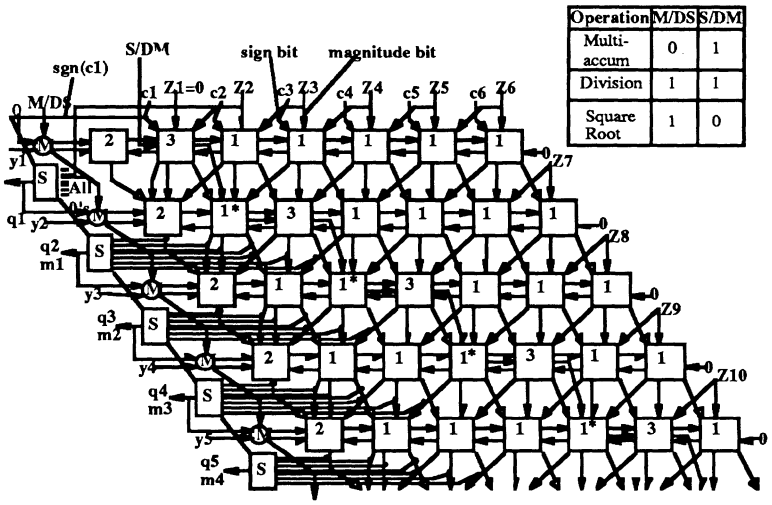
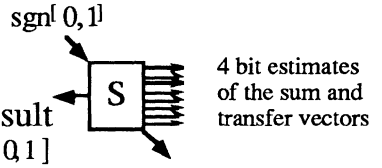


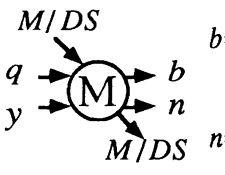
Fig. 4 DSP ALU using redundant arithmetic

The restriction of the carry ripple to the adjacent pipelined stage is the key to the use of the architecture with pipelined feedback. If we use the binary arithmetic element of Fig. 3 feedback can only take place after the most significant bit (MSB) has been calculated, this implies a pipeline latency of B clock pulses, where B is the wordsize. Special algorithmic techniques can be used to allow a maximum throughput rate using standard pipelined binary arithmetic, e.g. by manipulating the transfer function for a recursive filter design [14], but the flexibility of MSB first calculations is very attractive. The architecture shown in Fig. 4 produces high throughput rates independent of the wordsize. Redundancy in the representation permits a degree of choice in selecting result digits and hence a degree of error is permitted in accumulating the results. The architecture is implemented by arrays of four types of adder cells, with control lines to allow the architecture to perform the different operations. The individual blocks used in the ALU are shown in Fig. 5 along with their arithmetic functions. The basic algorithm for multiply-accumulate, division and square root is as given by eqn. (4).

$$Z_j = 2Z_{j-1} - \begin{cases} -\left(\frac{y_j}{4}x - m_{j-2}\right) & \text{if mult / acc} \\ q_{j-1}D & \text{if division} \\ q_{j-1}(Q_{j-2} + q_{j-1}2^{-j}) & \text{if square root} \end{cases} \qquad (4)$$

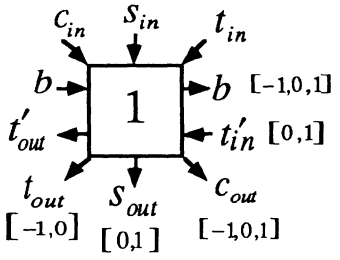
$$\text{for } j = 1,2,3,\ldots,k$$





$$b=\begin{cases} q & \text{if } M/DS=1 \\ -y & \text{if } M/DS=0 \end{cases}$$

$$n=\begin{cases} 0 & \text{if } M/DS=1 \\ -q & \text{if } M/DS=0 \end{cases}$$

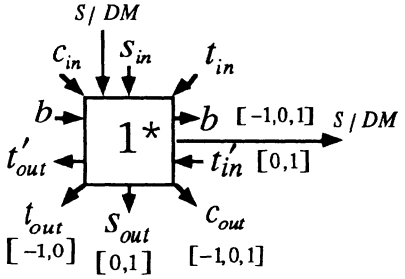


$$p = -b \bullet c_{in}$$

$$2t'_{out} + w = p + s_{in} + t_{in}$$

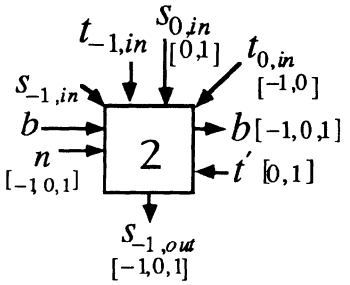
$$2t_{out} + s_{out} = w + t'_{in}$$

$$c_{out} = c_{in}$$



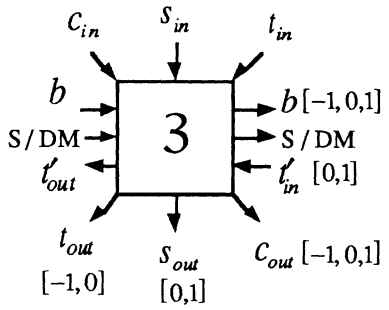
Same as cell type 1 except

$$c_{out}=\begin{cases} c_{in} & \text{if } S/DM=1 \\ b & \text{if } S/DM=0 \end{cases}$$



$$s_{-1,out}=2\left(t_{-1,in}+s_{-1,in}\right)+s_{0,in}+t_{0,in}+t'+n$$

Fig. 5a Cell types 1, 1*, 2, S, and M used in DSP ALU



$$2t'_{out} + w = p + s_{in} + t_{in}$$

$$2t_{out} + s_{out} = w + t'_{in}$$

$$c_{out} = c_{in}$$

$$p = \begin{cases} -b \bullet b & \text{S/DM} = 1 \\ -b \bullet c_{in} & \text{S/DM} = 0 \end{cases}$$

Fig. 5b Cell Type 3 used in DSP ALU

The original implementation used logic gates to build the functional blocks, but, in a later section, we will use a form of latched dynamic logic to illustrate efficient techniques for pipelined circuit design in which the complexity of the switching functions does not depend directly on Boolean decompositions into logic gates.

FINITE RING INNER PRODUCT STRUCTURES

The previous section illustrated that the number representation and corresponding arithmetic can improve the architecture of the pipelined DSP block. The most beneficial structural representation for feed forward calculations (e.g. inner product computations) can be obtained by mapping the original integer calculations to a direct product ring where calculations are carried out by parallel and completely independent small dynamic range ALUs. There is a conversion overhead associated with this technique, but the resulting main computational structure is benign as far as testing, clocking and fault detection are concerned.

Number theoretic architectures have traditionally been based on the Residue Number System (RNS), but the disadvantages of RNS techniques (non-homogeneous data conversion architectures) outweigh the advantages of carry free computation. A recently introduced approach, based on a polynomial ring mapping strategy, removes some of the RNS mapping problems [15]; however, unlike the algebraic integer mapping procedure [16], this new technique allows simple, error-free, mapping of incoming integer streams, and homogeneous conversion architectures at the output. The main body of the computation is performed in identically replicated linear bit-level pipelines; this has important ramifications in terms of fault

tolerance and testability when implemented in dense technologies, such as WSI and ULSI.

RNS Systems

In RNS systems we deal with rings, or fields, that are used for the actual implementation and rings that are isomorphic to direct products of implementation rings or extensions of them. A given digital signal processing algorithm is mapped from real or complex integer arithmetic to the implementation rings, the computation is carried out there, and the result is then mapped back to obtain the final answer.

Let m be a positive integer. We denote by $R(m)$ the ring of integers modulo m, i.e.

$$R(m) = \{S:\oplus_m, \otimes_m\};\ S = \{0,1,..,m-1\} \tag{5}$$

Where we use the notation $a \oplus_m b$ and $a \otimes_m b$ to imply the residue reduction of a and b modulo m within addition and multiplication. We can extend the notion of addition and multiplication from the elements of S to all of the integers. If R_1 and R_2 are any two rings then we can define the cross-product ring $R_1 \times R_2$ as the set of pairs $(s_1, s_2) \in S_1 \times S_2$, with addition and multiplication defined component wise, i.e. by

$$\begin{aligned}(a_1,a_2) \oplus_{R_1 \times R_2} (b_1,b_2) &= (a_1 \oplus_{R_1} b_1, a_2 \oplus_{R_2} b_2) \\ (a_1,a_2) \otimes_{R_1 \times R_2} (b_1,b_2) &= (a_1 \otimes_{R_1} b_1, a_2 \otimes_{R_2} b_2)\end{aligned} \tag{6}$$

The isomorphism ($\equiv$) between $R(M)$ and the direct product of $\{R(m_k)\}$ means that calculations over $R(M)$ can be effectively carried out over each $R(m_k)$, independently and in parallel. A final mapping to $R(M)$ is performed at the end of a chain of calculations. We have therefore broken down a calculation set in a large dynamic range, M, to a set of L calculations set in small dynamic ranges given by the $\{m_k\}$. This is the main advantage of using the RNS over a conventional weighted value numbering system (e.g. binary).

The final mapping is found from the CRT:

$$X = \sum_{k=1}^{L} \left\{ \hat{m}_k \otimes_M \left[x_k \otimes_{m_k} (\hat{m}_k)^{-1} \right] \right\} \tag{7}$$

with $\hat{m}_k = M/m_k$, $X \in R(M)$, $x_k \in R(m_k)$ and $(\bullet)^{-1}$ the multiplicative inverse operator. We have also used the notation Σ_M to indicate summation over the ring $R(M)$.

Polynomial rings and quotient rings

We let $R[X]$ denote the ring of polynomials in the indeterminate:

$$X{:}R[X] = \left\{ \sum_{k=0}^{n} a_k X^k : a_k \in R, n \leq 0 \right\} \tag{8}$$

If $X_1, X_2, \ldots X_S$ are indeterminates then we define the ring $R[X_1, X_2, \ldots, X_S]$ to be the ring of multivariate polynomials in the indeterminates. We use polynomial rings, where the base ring R, is a modular ring, $R(M)$, and we write $R_M[X_1, X_2, \ldots, X_S]$ in place of $R(M)[X_1, X_2, \ldots, X_S]$.

For a given polynomial $g(X) \in R\{X\}$ we consider the set $(g(X))$ of all (polynomial) multiples of $g(X)$. This set is called the 'ideal' generated by the polynomial $g(X)$ in the ring $R[X]$. The quotient ring $R[X]/g(X)$ is then defined to consist of all elements of the form $f(X)+(g(X))$, with $f(X) \in R[X]$. The more usual way of considering the quotient ring is to consider sums and products of polynomials reduced according to the equation $g(X) = 0$, that is, to consider the remainder after division by $g(X)$.

We apply polynomial ring mapping by letting indeterminates represent various powers of 2 in the binary representation of the data samples. This allows the data to be expressed as polynomials with small coefficients. These coefficients are then mapped to a direct product ring consisting of many copies of Z_M (the ring of integers modulo m) as factors. The direct product repeats the factor Z_M many times, so that the same prime divisors of M are used repeatedly, thus obviating the need for additional, larger primes.

In order to be able to perform useful computations, the modulus, M, has to be able to contain the coefficients of result polynomials. Multiplication will be the major problem in coefficient growth, and we assume that the algorithm is arranged so that only single cascades of multipliers are used prior to the application of mapping circuitry. We can further decompose M, to allow the use of very small rings, by the application of a RNS. The mathematical derivations are somewhat tedious, and the reader is referred to a more complete description in [17]

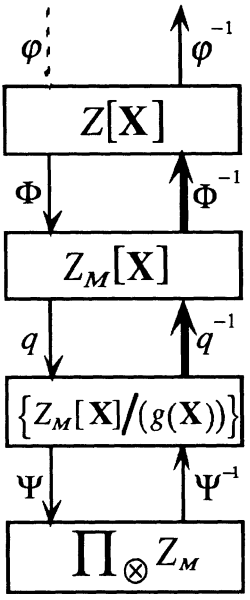


Fig. 6 Mapping procedure

The elements of the mapping procedure are shown in Fig. 6; the method has the following advantages:

1) There are no quantization problems. The data, either real or complex, are assumed to be of a given fixed bit length. No approximations or scaling are used in encoding the data; this is a major advantage compared to the algebraic integer approach [16].

2) The polynomials used are of a general nature, so that no restrictions are placed on the prime divisors of the moduli, except in the case of a QRNS representation of complex data [18] in which case the condition is the usual one of $p \equiv 1 (\mathrm{mod}\, 4)$ for prime divisors p of the modulus M.

3) The same small moduli can be used many times, which allows VLSI implementations of systems which can process data of a large bit length, using direct products of many copies of modular rings with small moduli.

4) Encoding is a simple matter of diverting the bits of the input data to the proper channels. Decoding is only complicated insofar as the Chinese Remainder Theorem is used, and even then only for a limited number of small moduli. Scaling, if used in decoding, is

simplified by the ring structures used; certain monomials can be ignored as they represent insignificant digits.

As an example of the mapping let us write complex input integers as polynomials in the variables W, X, Y and Z, where $W=2$, $X=4$, $Y=16$, and $Z=256$. With this notation, any positive integer $<2^{16}$ can be written in a unique fashion as a sum:

$$\sum_{i_1,i_2,i_3,i_4 \in \{0,1\}} a_{i_1 i_2 i_3 i_4} W^{i_1} X^{i_2} Y^{i_3} Z^{i_4} \tag{9}$$

with the coefficients equal to 0 or 1. Similarly, any negative integer $>2^{16}$ can be written in the same form with coefficients 0 or -1 (note that the use of 0 and ± 1 implies a signed bit representation of the coefficients). We will use a modulus $M=105$, and introduce a further decomposition using an RNS moduli set $\{3,5,7\}$. In order to represent the complex operator, j we add an additional indeterminate, T; we use the polynomial $T(T^2+1)=0$ to define the mapping [19]. The amazing feature about this mapping is that the complex operator and the bit operators are interchangeable, allowing a variety of binary representations of complex numbers to be simply mapped to the direct product ring. This map is performed by evaluating each of the five variables W, X, Y, Z, and T at each of the three roots 0, +1 and -1. This results in $3^5 = 243$ results for each of the moduli 3, 5, and 7. The map (a tensor product) is very simple, consisting of nothing more difficult than sign changes and additions.

A simplified mapping structure is shown in Fig. 7 for 3 indeterminates (one for the complex operator). The mapping blocks simply consist of modulo adder/subtractors with constant multipliers. The arithmetic is computed modulo 3, 5 or 7 so that the complexity of the block is related to 6 input bits and 3 output bits. This, in fact, is the maximum complexity of any block in the system including the hardware for computing the appropriate DSP algorithm.

The DSP algorithm is inserted into the independent streams shown by the dotted horizontal line in Fig. 7.

A 'Fast Algorithm' Example

The technique requires an inverse mapping each time a scaling operation is required, which limits its usefulness to feedforward algorithms with sparse scaling requirements.

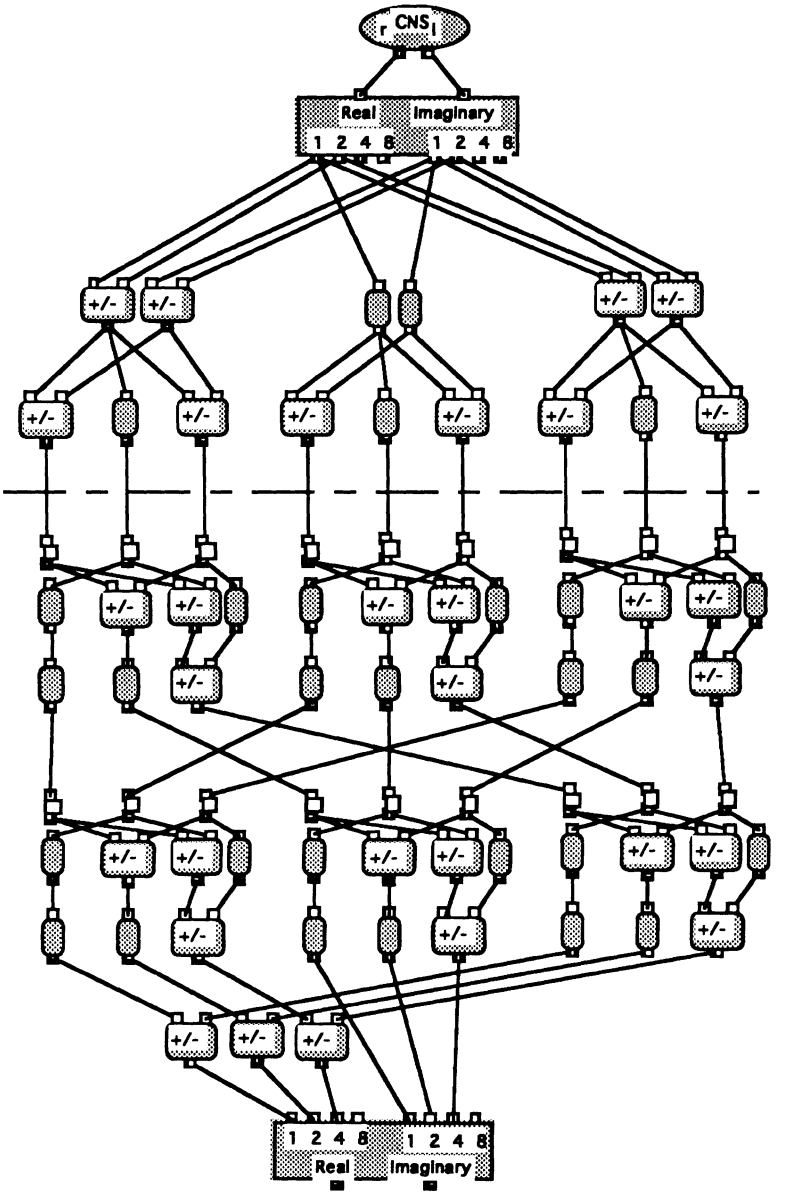


Fig. 7 Polynomial ring mapping structure

Scaling is required if the dynamic range of the integer computation is likely to be overflowed, and this occurs for cascades of multiplication operations. We normally look for inner product type computations, where a single cascade multiplication is embedded in many addition operations. Typical algorithms that possess this property are FIR

filters, direct form transforms (not, in general, 'fast algorithms') and matrix multiplications. In some circumstances it is possible to produce 'fast' architectures that still allow a single cascade multiplication structure. An example of a 15 sample Discrete Cosine Transform that supports such a single multiplication cascade is shown in Fig. 8. The strip in the centre contains the multiplications, all other operations being additions or subtractions [20]. The multiplications are given in Table 1, where $c_i = \cos(i/30)$.

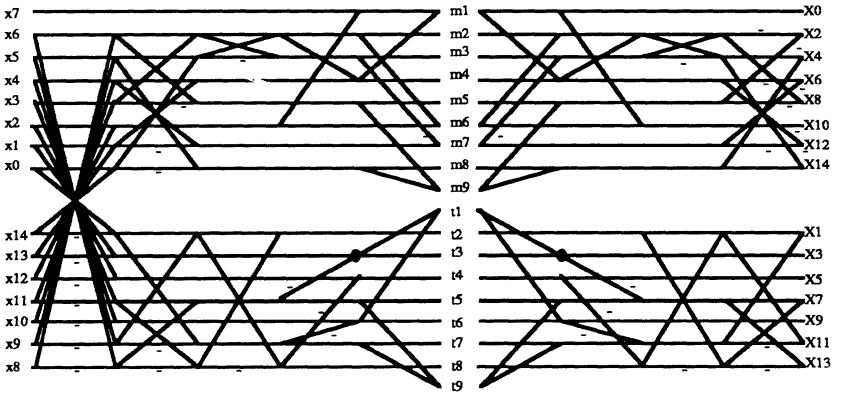


Fig. 8 Structure of 15 sample DCT with a single multiplication cascade

$m_1 = 1$	$m_2 = 1.875$	$m_3 = 0.75(c_6 + c_{12})$
$m_4 = -1.25$	$m_5 = 0.5(c_4 + c_{14} - c_2 - c_8)$	$m_6 = -1.5$
$m_7 = -0.5(c_6 + c_{12})$	$m_8 = -\frac{1}{2}(c_4 + c_{14} + c_2 + c_8)$	$m_9 = 0.5(c_2 + c_8)$
$t_1 = -c_3$	$t_2 = \frac{1}{4}(c_1 + c_{11} + c_7 - c_{13})$	$t_3 = c_3 + c_9$
$t_4 = -c_5$	$t_5 = -1.5(c_3 + c_9)$	$t_6 = c_3 - c_9$
$t_7 = 1.5(c_9 - c_3)$	$t_8 = 1.25c_5$	$t_9 = 1.5c_3$

Table 1 Multiplier Coefficients for Fig. 8

This structure is now replicated at every parallel data stream on Fig. 7 each computations over a 3-bit finite ring using a maximum

complexity block of 6-bit input and 3-bit output. The structure of the parallel DCT computation is shown in Fig. 9. The rectangular blocks are the input and output mapping arrays,.

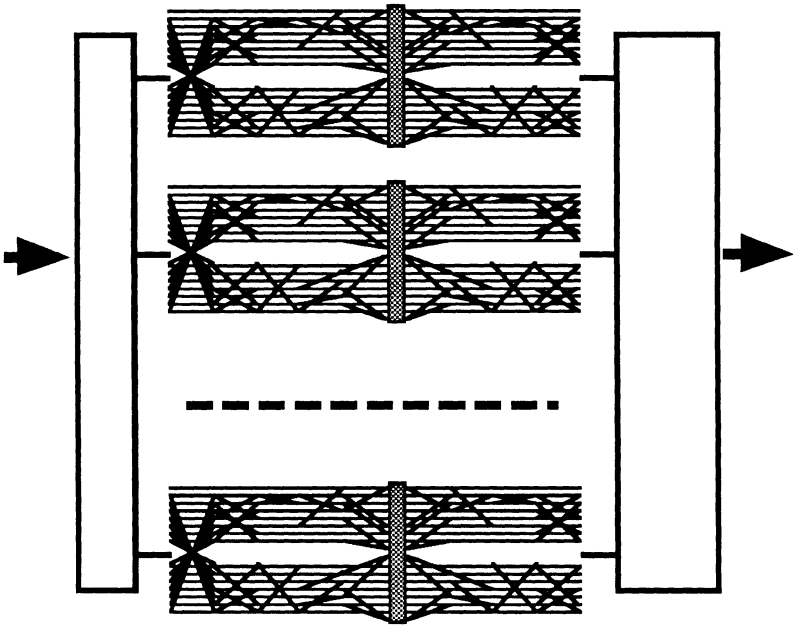


Fig. 9 Parallel DCT implementation

The other example of the use of 'fast' algorithms is in the implementation of number theoretic transforms (NTTs) [21] where the number range growth is only based on the properties of the convolution of the two sequences, not on the way in which the NTT is computed.

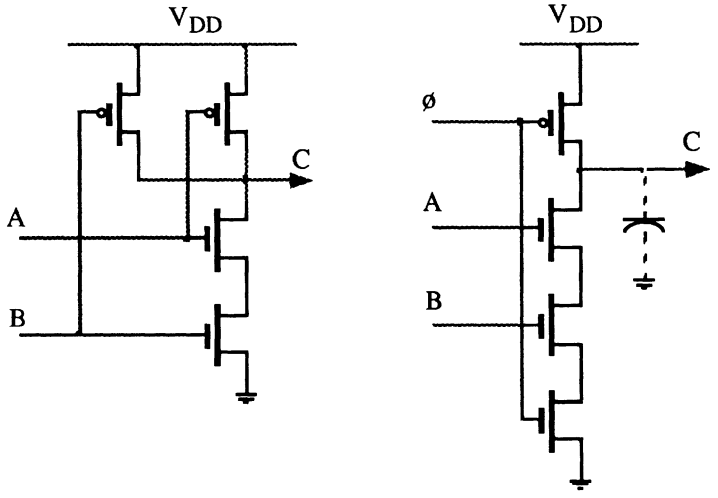
Notes on Implementation

It is important to reflect on the complexity of logic gate implementation of general modulo computations versus binary, or some modification, such as redundant arithmetic. Binary weighted magnitude implementations tend to decompose quite readily to full adders or some modification to the full adder structure. Residue computations do not naturally decompose this way, and most work on RNS implementation, for example, has concentrated on the use of look-up tables for the switching functions, where there is no need for explicit logic gate decomposisitions [22]. We will follow this procedure, but also use the results we obtain for the binary type structures discussed earlier.

BUILDING BLOCKS

The key to efficient implementation of bit-level systolic arrays, is in the minimization of the logic/latch structure used for each processor. Early implementations of bit-level systolic arrays used standard static CMOS logic with static D-latches for the pipeline storage [23]. This is not an efficient method for designing arrays in which there may be a ratio as low as 1:3 between latches and logic gates. The switching blocks (represented by the small number of logic gates connected between latches) may be realized by dynamic trees embedded in a dynamic logic structure such as domino [24] or cascode voltage switched logic [25] and the latches can also use dynamic logic principles.

The difference between static and dynamic logic is illustrated in Fig. 10 for a 2-input NAND gate. The static gate uses complementary p-channel and n-channel blocks to ensure that a conducting path, either to ground or V_{DD}, exists for valid input logic states. The dynamic gate uses the parasitic capacitance at the output node to temporarily store a pre-charge (high logic level for $\phi = 0v$); when the gate evaluates ($\phi = +5v$) this charge either remains high ('1' output) or is discharged ('0' output) depending upon the state of the middle two n-channel transistors.



Static Logic Dynamic Logic

Fig. 10 A 2-input NAND gate

The advantage with dynamic logic is that quite complex gates can be built, using only n-channel blocks, with much faster evaluation times than equivalent static blocks. It is not unusual to see complete full adders, for example, implemented as a single complex n-channel block in dynamic logic, where an equivalent static logic implementation is normally decomposed into cascades of simpler logic gates. The disadvantages with dynamic logic are the temporary nature of the logic output, the need for a clock to time the pre-charge and evaluate phases (static logic behaves like the ideal combinational gate), and circuit theoretical problems associated with charge sharing between nodes during evaluation, and potential race conditions between cascades of dynamic gates [26].

The concept of temporary charge storage is directly applicable to pipelined systems where simple two-phase gates provide the latching and isolation functions required. Fig. 11 shows a very simple implementation of this function using two n-channel transistors and a non-overlapping two-phase clock, the temporary charge storage is on the parasitic capacitance present on the 'wire' connecting the two transistors..

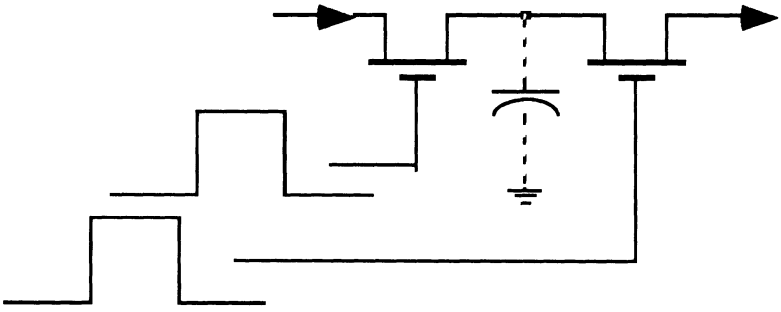


Fig. 11 Simple Master/Slave dynamic pipeline latch

Symmetrical drive capabilities can be obtained, in CMOS, by using transmission gates (parallel n and p-channel transistors) with complementary gate drive signals; the increase in hardware is considerable, though still less than an equivalent static gate implementation. There is no problem with the temporary nature of the latch storage, because data are continually flowing through the latch at cycle times many orders of magnitude lower than the charge storage time.

Recently published work has shown that it is possible to obtain extremely high pipeline rates (over 200MHz, for example, in a mature 3μ CMOS technology) by combining simple logic blocks with single phase clocked dynamic circuitry [3, 7, 27]. The trade-off in this approach is throughput rate versus latency (number of pipeline stages required). The ideal use for very high speed pipelines is in locally pipelined arithmetic units, where the local clock rate is much higher than the input data rate; bit-serial implementations are ideal target architectures. In the case of bit-level systolic arrays, the clock rate is the same as the rate of the data stream, and we can therefore develop a circuit approach where there is a close match between the maximum throughput of the circuitry and the data stream rate. This is a particularly appropriate approach if a good hardware/speed trade-off is the result. We now discuss specific dynamic logic building blocks that are inherently suited to the implementation of bit-level systolic arrays, and which allow general use over the diverse example areas presented so far.

PIPELINED SWITCHING TREES

We embed a minimized complex NFET logic block (we will refer to this as a *switching tree*) in a TSPC master/slave latch, as shown in Fig. 12.

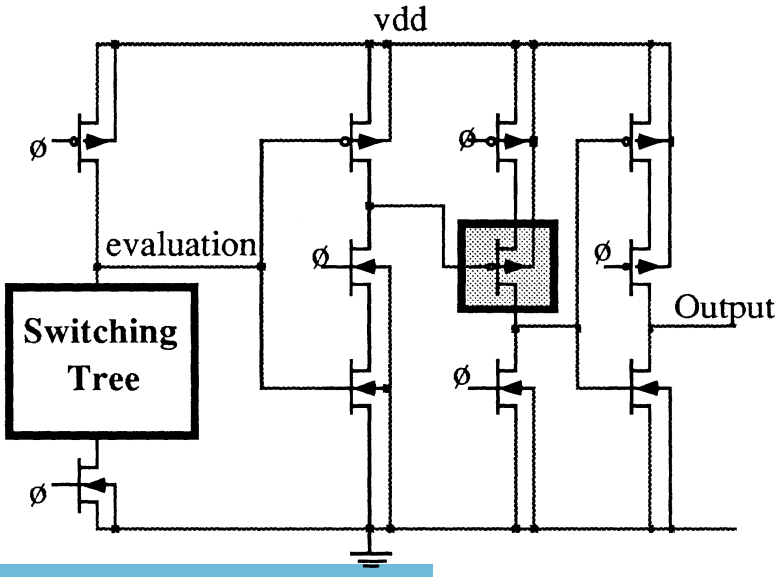


Fig. 12 Embedded tree

Note that the p-channel logic block (highlighted) is restricted to a single PFET (inverter), operating as a slave latch. Our approach is to build the logic for each stage entirely within the NFET block; this provides the most area efficient implementation of a given logic function, and allows the use of an asymmetrical clock.

The tree is designed as an n-dimensional ROM (binary tree) where n is the number of input variables, as shown in Fig. 13. The notation represents transistors whose gates are driven by the true logic input as arcs, ╲ ; the other arc, ╱, represents transistors whose gates are driven by the complement of the logic input. By removing selected transistors from the bottom of the tree, we can implement any arbitrary truth table. Viewing the tree in this way allows tree height reduction using higher order decoders rather than the single inverter decoders required for the n-dimensional ROM.

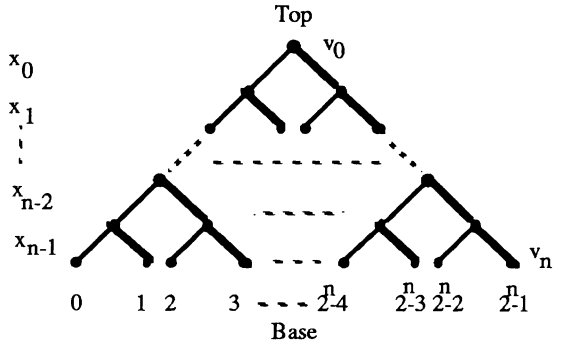


Fig. 13 A Full Binary Tree

A full binary tree possesses interesting qualities as far as a series chain discharge block in dynamic logic is concerned. In the full tree we see that, for stable logic inputs, only a single series path connects the top node to one of the bottom nodes, and the capacitance at every node in each of the possible series paths is only 3 source/drain capacitances in parallel.

The minimization technique is not based on Boolean algebraic concepts, as with most reported dynamic logic designs, e.g. [25], but rather on the application of two graph reduction rules [8]. This approach is useful in that it allows a well established relationship between reduced tree structure and silicon layout that is essential for both hand custom layout and module generation approaches for complex multiple output trees. It also allows efficient circuit decomposition based on treating the block as a ROM, that may be hidden with a pure Boolean algebraic approach.

GRAPH BASED REDUCTION

In order to present the two simple rules used in the minimization procedure, the following definitions are given, based on Fig. 13:

A tree represented by a graph can be denoted as, $G=\{X,V\}$, where X is a set of edges (n-channel transistors) $\{x_{i,j}\}$, and V is the vertex set of nodes $\{v_{i,j}\}$. An edge $x_{i,j}$, consists of elements $(ct, v_{k,l})$, where i and k are tree levels, $j \in [0, 2^{i+1}-1]$, $l \in [0, 2^{k}-1]$, and connection type $ct \in \{T, F, W\}$. The inputs to the tree are $g_i \in \{0,1\}$. If $g_i = 1$, then the path takes edge T if it is present. If $g_i = 0$, the path takes edge F if it is present. W represents an arc which is a wire, or link, connection, and is only present following the successful application of a reduction rule.

A path, $P_{(i,j),(k,l)}$, is the connection from node $v_{i,j}$ to node $v_{k,l}$, constructed by edges. A full path connects node $v_{0,0}$ to node $v_{n,l}$, where n is the height of the tree. A switching tree is the reduction of a unique set of full paths that describe a logical function. A tree is characterized by two sets of full paths, a true set in which an edge T or F is present at the n level, and a complement set in which an edge T or F is removed at the nth level. A truth table is mapped onto a full tree by removing a sub-set of edges $\in \{x_{n,j}\}, j \in [0, 2^{n+1}-1]$, from a full tree based on the set of zeros in the truth table.

Graph Reduction Rules

The following two rules are used in the graph reduction technique [8].

Rule:1 Merging of shared sub-trees

If paths from $v_{i,j}$ to $v_{n,l}$ and from $v_{i,k}$ to $v_{n,m}$, where $j,k \in [0, 2^{i}-1]$, $l,m \in [0, 2^{n}-1]$, contain an identical set of edges, starting at a node at level p, those nodes where the matching occurs in both sequences can be merged. Furthermore, if $k = j$, and $i - p = 1$, then the edges from node $v_{i,j}$ to nodes $v_{p,l}$ and $v_{p,m}$ can be replaced by a link edge.

Rule:2 Deletion of Common Edges

Consider a set of edge paths, X_1, connecting a node $v_{i,j}$ with a node at level n, and a set of edge paths, X_2, also connecting the node $v_{i,j}$ with a node at level n. Path X_1 follows the T edge from node $v_{i,j}$, and

path X_2 follows the F, edge from node $v_{i,j}$. If X_2 covers X_1, then the first edge in X_1 has $ct = W$.

Rule 1 provides for the greatest reduction in the number of nodes by merging common subtrees. Rule 2 replaces transistor links between nodes with wire links. When merging occurs, however, accidental paths through the tree may be created which can produce false results. We call these accidental paths *sneak paths* after the same phenomenon in switch matrices. Rule 2 provides an important reduction mechanism, when the truth table contains *don't care* states. These states are set to either a 1 or 0 to facilitate tree decomposition. Rule 2 sets these states to force one half of a subtree to be a subset of the other half so that the transistor link leading to the subset may be replaced by a wire link. States in the covering half of the subtree that match those in the subset will always be taken care of by the subset. Thus, their effect on the output is unimportant. This may allow the use of Rule 1 in the lower portions of the switching tree (below the common edge).

Example Results

As an example of the use of the graph reduction technique, consider constructing trees for a 4-bit binary adder:

$$Z = X + Y + C = 2^4 C_4 + \sum_{i=0}^{3} 2^i S_i \text{ where } X, Y \in \{0,1,2,\ldots,15\}$$

and $C \in \{0,1\}$. The starting point is a set of lists that define the truth table for each of the 5 output bits based on the 9 input bits. We therefore produce 5 binary trees, each of height 9, and each programmed from a truth table with 512 states for the output bit. We now apply the two reduction rules and finally produce the trees shown in Fig. 14. Only the S_3 and C_4 trees are merged, even though the other trees were merged by the software; this was a final decision based on layout considerations.

There are some points to note in this design:

1 Although the number of inputs is 9, the maximum tree height, because of the decomposition properties of the adder structure, has been reduced to 6.

2 The explicit wire, W, connection type is hidden because the tree heights have been compressed by assigning different input variables to the same row.

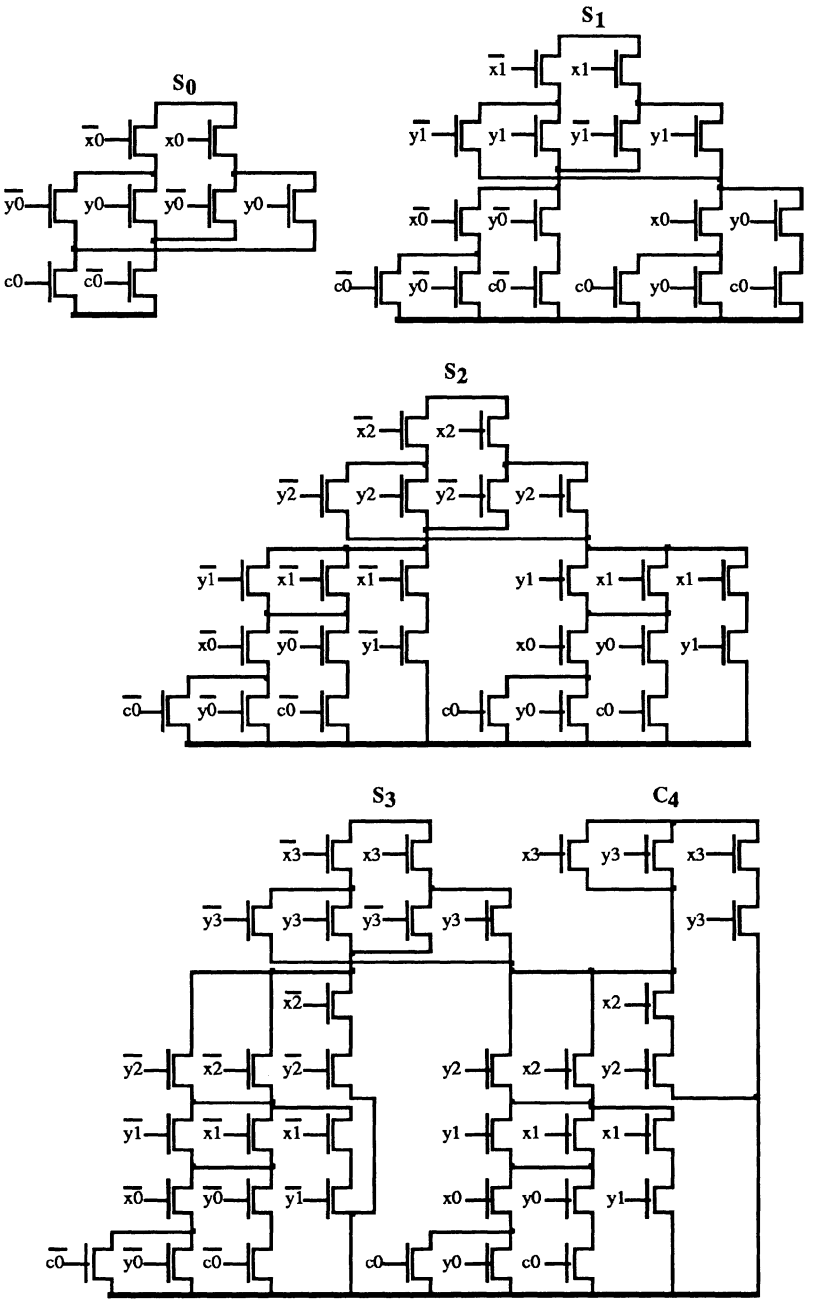


Fig. 14 Switching trees for a 4-bit adder

3 The ordering of the variables in the original tree structure can have an affect on the reduction in terms of numbers of transistors and locality of interconnections. For multiple output trees that are to be merged (as in the original specifications for this example) the ordering should be consistent for all of the individual trees, and this constraint tends to minimize the effect of variable ordering over the entire multiple tree reduction. For this example, the ordering is $\{x_3, y_3, x_2, y_2, x_1, y_1, x_0, y_0, c_0\}$.

4 The maximum net capacitance has been increased from 3, for the original binary tree, to 7 for the net that merges the C_4 and S_3 trees; however, other nets have reduced capacitance. The gate load is shown in Table 2. The general trend is an increase in load as we move down the tree but the gate load of the original binary tree is reduced by orders of magnitude at the bottom.

Variable	True	Comp-lement
x_3	3	1
y_3	4	2
x_2	3	3
y_2	4	4
x_1	5	5
y_1	6	6
x_0	4	4
y_0	8	8
c_0	7	7

Table 2 Gate load for the 4-bit Adder

5 It is tempting to apply standard Boolean decomposition theory to the problem, but this misses the point that graph based reduction allows a direct link between the original problem description and the implementation on silicon. For this particular example, the movement of carries between single bit (full) adders is not a

consideration, whereas typical Boolean decompositions of multi-bit adders are centred on the carry manipulation problem.

CIRCUIT CONSIDERATIONS

Worst case test

In order to invoke worst case conditions, we drive single tree paths to both provide maximum charge sharing effects and maximum pull down delay. A typical test circuit is shown in Fig. 15 for a full 4-high binary tree. Worst case node capacitance loads are used for specific designs.

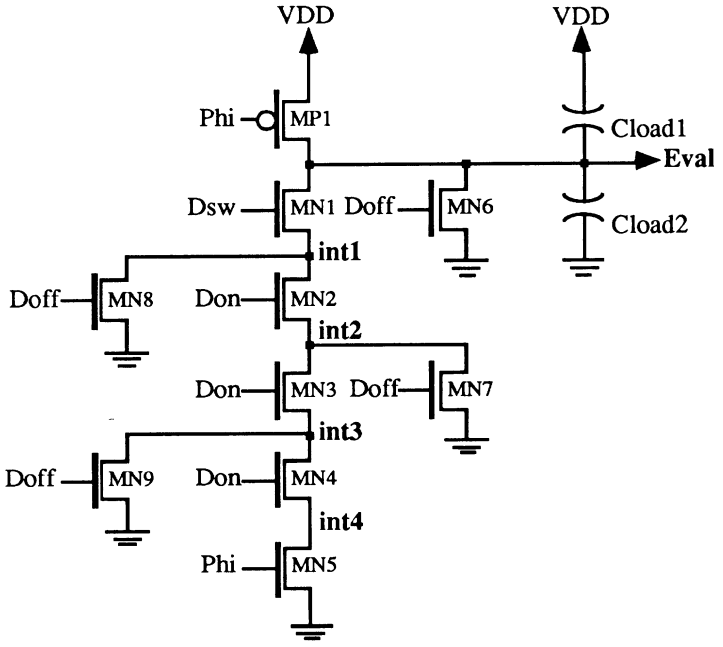


Fig. 15 Worst Case Test Circuit

Since we assume that a switching tree block is always driven from the output of a TSPC latch (pipelined system), the well known worst case charge sharing condition will not occur (where evaluation takes place immediately after a complete tree discharge). Inputs can only change state at the beginning of the precharge cycle, and must remain constant during both the precharge and evaluate cycle.

Test results for a full binary tree path, for our target 3μ DLM CMOS process, indicate that a tree height of 6 yields acceptable charge sharing droop with pull-down times in the region of 20ns. Fig. 16 shows worst case results for a charge share cycle followed by a pull-down cycle for a 6-high tree.

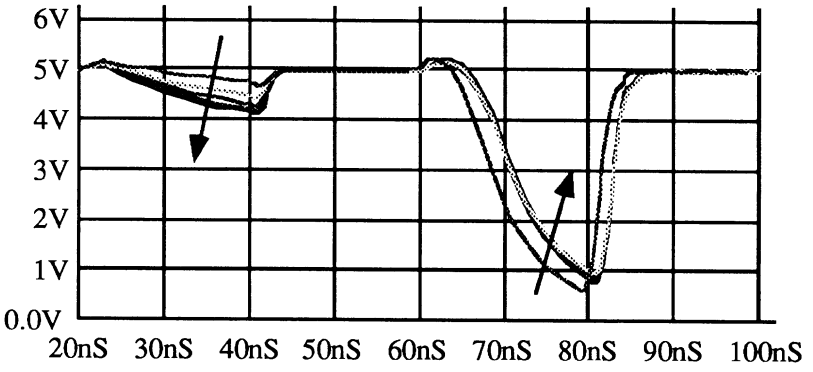


Fig. 16 Worst case test results for a full binary tree path

The arrows on Fig. 16 indicate a movement of a 7 transistor load (which corresponds to the maximum drain load experienced by the 4-bit adder example) towards the bottom of the tree; we see that large loads at the bottom of the tree have the worst case effect.

Reduction of Charge Sharing

We can use the standard technique of internal tree p-channel pull-up transistors to reduce the charge sharing effect. In the case of pipelined blocks, however, a single pull-up is often sufficient, and since we are pulling up an internal node directly, we can trade a reduction in precharge time for an increase in evaluate time, without reducing the throughput rate of the pipeline. Because we are only using a single pFET inverter for the p-logic block of the TSPC latch, we have the flexibility of adjusting the precharge/evaluate duty cycle without being concerned about the effect on the pFET slave latch; i.e. the timing limitations are governed by the nFET latch circuitry.

Fig. 17 shows the effect of increasing the evaluate/precharge duty cycle and applying a small pull-up to either node 5 or 4. In the case of no pull-up pFET, the tree has a reduced precharge of internal node capacitance, and so experiences a large charge sharing effect. With even a small pull-up pFET (6μ width) there is a sufficient precharge of the large capacitance node at drain 5 to essentially remove any

effect of charge sharing. The increased evaluation time ensures that the worst case pull down falls well below the TSPC latch input signal noise margin. A single pull-up transistor will also be effective if it is within one, or two, transistors of the large capacitance node. The constant voltage input effect is clear in the almost identical waveform based on a pull-up at node 4 (a much more lightly loaded node).

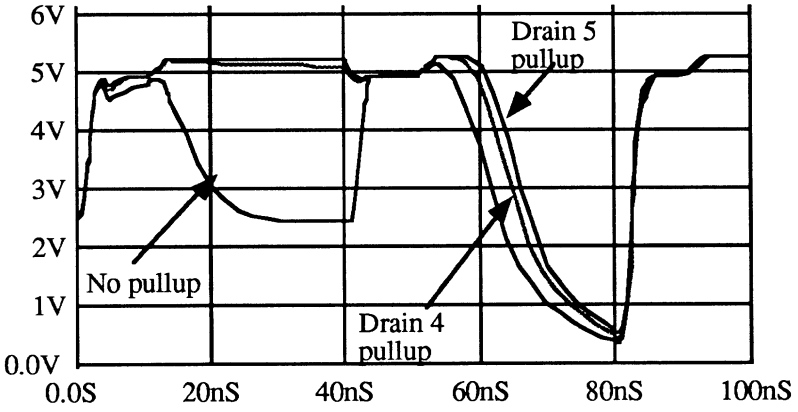


Fig. 17 Increasing evaluate/precharge ratio

Using these techniques we can guarantee almost constant pull-down delay characteristics of pipelined blocks by restricting logic blocks to tree heights, n_{max}, and thus maintain maximum performance of the pipelined arithmetic system. We maintain constant block height by either decomposing the computational function, at the functional level, into such blocks, or by applying tree height reduction steps directly to a minimized tree/latch structure, where $n > n_{max}$.

Reduction of Tree Height

We can reduce tree height by *pulling out* input decoders. The decoders are built using domino logic by simply removing part of the original binary tree, inverting the removed section, and cascading with the remaining tree structure. The Domino decoders use the same single clock signal as the TSPC latch. We can also separate the lower parts of the trees and build the decoders in the p-channel part of the master/slave latch.

Fig. 18 demonstrates pulling out a 2:4 decoder from the top of the S_3 / C_4 tree in our example 4-bit adder. Note that although the single decoder can be shared between the two trees, the C_4 tree only needs to

use the z_3 decoder output since the other transistors are already at a single transistor height.

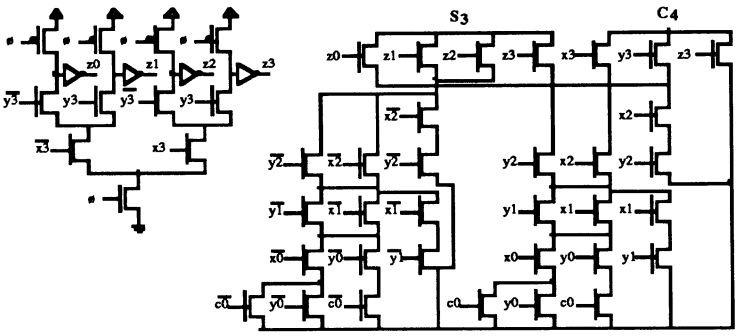


Fig. 18 Pulling out a 2:4 decoder at the top of the S_3 / C_4 tree

Decreasing Pull-Down Time by Sizing

A significant reduction in pull-down delay can be obtained by sizing the transistors [24]. This is a complex issue when looking at the interaction between pull-down delay and charge sharing. We can use an approximate analytical technique [28] that obtains very close to optimal results while allowing both on-the-fly calculations and algebraic manipulations for module generator applications.

In this technique, the discharge delay of the evaluation node E is given by Elmore [29];

$$T_D = \sum_{i=0}^{N} R_i \sum_{j=i}^{N} C_j = \sum_{i=0}^{N} T_{Di} \qquad (10)$$

The analytical technique uses an observation that, for close to optimum results, the time constants of eqn. (10) are almost equal:

$$T_{D0} \approx T_{D1} \approx T_{D2} \approx \ldots\ldots \approx T_{DN} \qquad (11)$$

A typical result of the optimum sizing, using an iterative algorithm, versus the closed form technique, is shown in Fig. 19 for a 6-high tree (with ground switch).

Using the optimum sizing profile in the previous test tree (with appropriately sized load transistors) and increasing the width of the precharge and pull-up transistors to twice and four times their previous size, we obtain the results shown in Fig. 20.

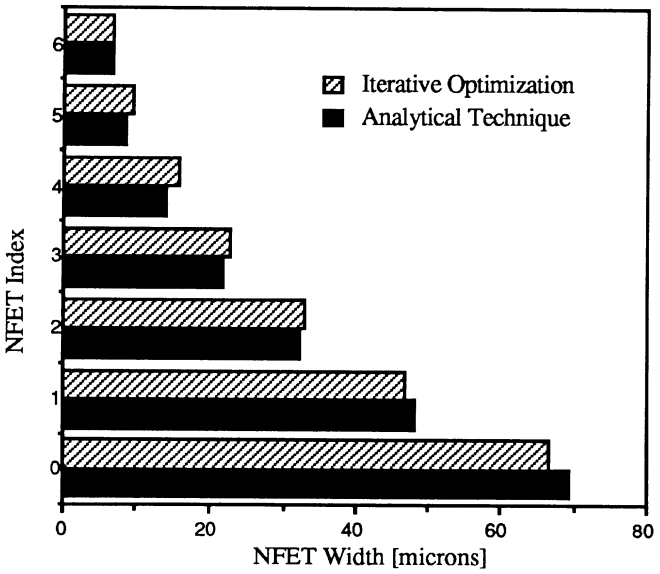


Fig. 19 NFET Sizing Profile

In this example the pull down time has been decreased by about a factor of 2, measured from the start of the evaluate phase to the 1v level. There is, however, about a 4 times increase in power dissipation measured over the two test clock cycles. This will increase to about 8 times if the clock speed is doubled to take advantage of the pull-down decrease. Clearly the option to size the transistor profile will have to be considered in light of the data stream throughput requirements, since the tradeoffs are quite severe.

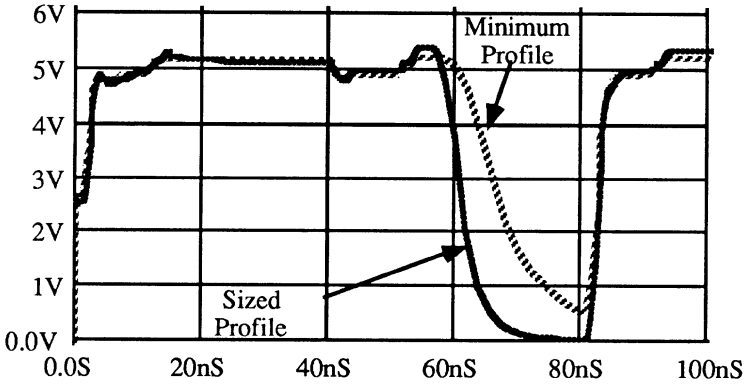


Fig. 20 Comparison between sized and minimum profile

FABRICATION RESULTS

The 4-Bit pipelined adder cell was fabricated on a test chip using a very mature 3μ DLM p-well CMOS technology [30]; all transistors in the NFET block were 5.4μ wide with a channel length of 3μ. In order to find worst case performance limits, we did not use either decoder tree height reduction, or internal node pull-up pFETs. The duty cycle between evaluate and precharge was 1:1. The cell (see the micrograph in Fig. 21) was one of five test cells on the chip.

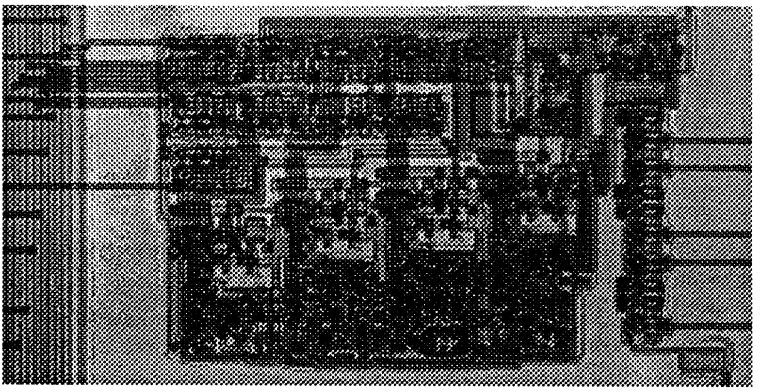


Fig. 21 Photomicrograph of 4-bit adder test cell

The cell reached a throughput rate of 40 Mhz at room temperature. The rate dropped to about 30MHz when the chip was heated with a dryer heat source. SPICE simulation results of the supply rail current demand, at an operating speed of 40 Mhz, show a peak dynamic current of approximately 6 milliamps with an average current demand of approximately 1 milliamp, leading to an average power dissipation of 5 mW. It is interesting that this power consumption is slightly less than the power requirement of the pipelined gated full adder, described earlier, built using static/transmission gate logic with dynamic latches. The 4-bit adder represents at least three times the switching complexity of the gated full adder, and gives a demonstration of the low power operation of reasonably high performance circuitry using the switching trees approach.

FURTHER EXAMPLES OF SWITCHING TREE BLOCKS

In this section we use the Switching Tree approach for the other two architectural examples presented earlier.

Redundant Arithmetic Block

As an example of the cascode minimization of a complex pipelined switching block, consider the construction of the cell in Fig. 5b. The merged trees are shown in Fig. 22.

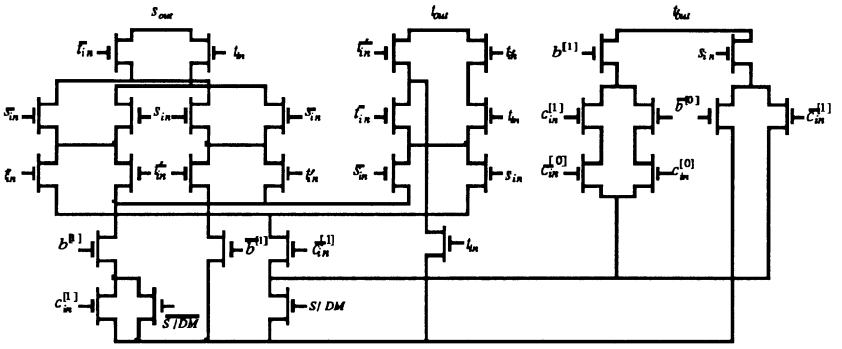


Fig. 22 Cell 3 Switching Tree Implementation for DSP ALU

In this case the 8-bit input tree reduces to a maximum height of 5. The maximum drain load is 5 drains, and the block pipelines at over 30MHz with no pull-up transistors. The charge sharing is quite acceptable down to a precharge pulse width of 10ns.

Modulo Arithmetic Block

Here we consider a Mod-7 multiplier as representative of the maximum complexity that will be required by the small ring polynomial mapping architecture discussed earlier.

The minimized tree is shown in Fig. 23. The order of the inputs (from the top) are $\{B_2, B_1, A_2, A_1, A_0, B_0\}$. This ordering was determined to be the best, based on a limited search, for minimizing interconnection lengths with a close to optimum reduction in number of transistors. We will see, in the next section, that minimizing the number of transistors is not necessarily the best criterion for optimization. The don't care states resulting from the fact that there are only 7 valid states in an 8 state system (3-bits), are used to help reduce the tree structure and to provide, as far as possible, local interconnections rather than cross connections. The merging of the three original trees is quite evident in Fig. 23. The complexity of the multiplication operation is seen in the preservation of the original tree structure at the top of the trees (this is not in evidence, for example, in Fig. 22). This minimized tree structure is ideal for the *pulling out of decoders* technique mentioned earlier.

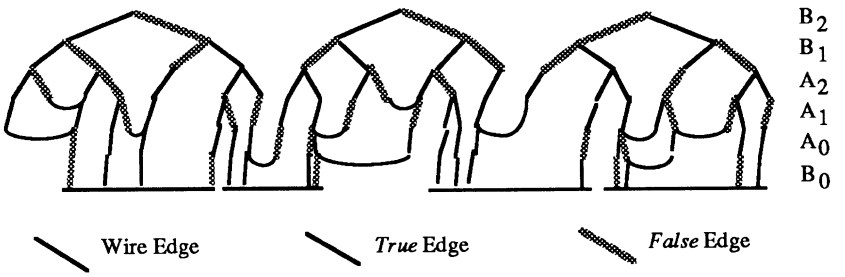


Fig. 23 Minimized tree for Mod 7 multiplication

SWITCHING TREE MODULE GENERATOR

In this section we discuss a module generator suitable for on-the-fly cell generation for arbitrary switching tree designs. This is essential for a design automation procedure, since it is impractical to pre-design a cell library based on the wide variety of truth table requirements that may have to be met in a typical arithmetic computational setting.

The following approach is very similar to the gate matrix or PLA concept, where a two dimensional array of transistors is generated based on a transistor network mapped from a minimized Boolean function. In our case the network is a direct mapping from a minimized binary tree. In order to illustrate the procedure we have developed, we will use the previous example of Mod 7 multiplication.

Placement Mapping

Our approach to the module generator is a direct mapping of tree primitives to layout primitives, using a matrix layout approach. The matrix of primitives for the Mod 7 multiplier is shown in Fig. 24. The levels corresponds to rows on the tree, and each level has two rows; one for the *True* edges and the other for the *False* edges. The position of the rows alternates between adjacent levels; this is to accommodate the inverters that are used to drive the complement gate signals.

The mapping of primitives to the matrix is performed by either filling, or leaving empty, the table positions shown in Fig. 24. The Wire and transistor primitives are direct mappings from the switching tree, the shorting primitives are used to connect gate signals, propagated on metal 2 lines to polysilicon transistor gate lines.

The metal 2 and polysilicon lines run horizontally across each row in the matrix with the metal 2 lines directly on top of the polysilicon lines. By shorting the metal 2 to the polysilicon at several places across the row (ideally near a transistor gate) we can eliminate the time constants associated with the large resistivity of polysilicon and transistor gate capacitances. We use space in the table to place the shorting primitives. Because these primitives are offset from the centre of the metal 2/polysilicon lines, they have two possible vertical directions; both directions have been used in Fig. 24.

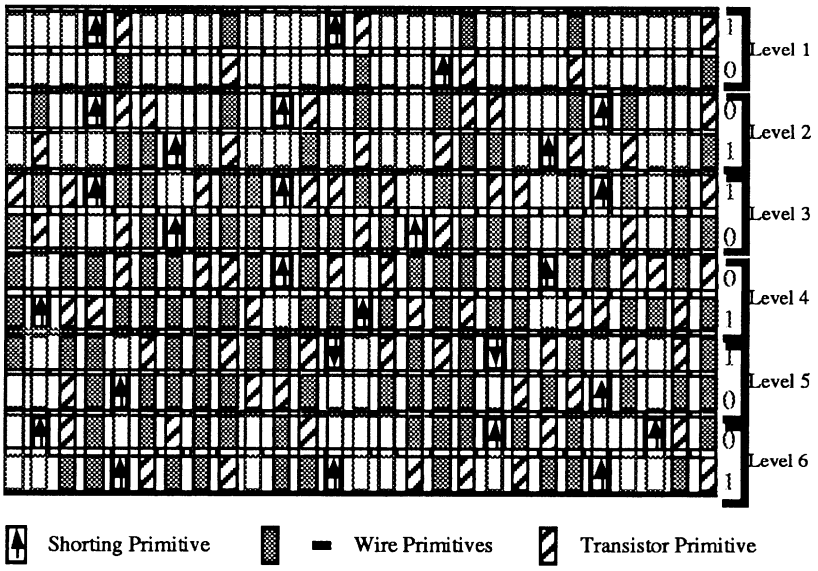


Fig. 24 Table of primitives for the Mod 7 multiplier tree

Placement Algorithm The algorithm used to map the tree edges to the matrix primitives is given below:

1) Start at the top of the right hand tree; and map to the right most column in the matrix.

2) Move towards the bottom of the tree, taking either right hand edges or single merged edges, mapping the edges (vertical wire links or transistors) to matrix primitives in the column. Place horizontal wire matrix primitives if a previously mapped edge (in the right hand adjacent matrix column) is connected to the currently mapped edge. The path will terminate when either a left hand link is reached, or when the bottom of the tree is reached.

3) Move to the left until the first unplaced left hand edge, at any vertical position, is reached. Terminate the algorithm if all edges have been placed.

4) Repeat from 2), mapping to a new column to the left of the previous column.

At the termination of the algorithm, the matrix is examined for suitable placement of shorting primitives. This is a somewhat heuristic procedure since there is a trade-off between reducing the resistance of the signal path to each transistor gate, and the extra capacitance load of the shorting primitive. There is often limited space for the shorting primitives, particularly near the dense central rows. We can see, from Fig. 24., that shorting primitives have been able to be placed within a short distance of every two or three transistors on a row; this will change with the particular function being implemented.

Observations

We see that the area required by the tree edge placement is given by:

$$A = 2 \bullet \Phi H_{Row} \bullet \Theta W_{Col} \qquad (12)$$

where:

Φ is the number of input lines to the switching tree (6 in the Mod 7 multiplier example)

H_{Row} is the height of each row

Θ is the number of separate columns required in the placement mapping

W_{Col} is the width of each column

Notably absent from eqn. (12) is the number of transistors. Although there will be a correlation between minimizing transistors and minimizing Θ., the only direct requirement for minimizing transistors is to reduce the number of series transistors in the critical path of the tree (the path that has the maximum number of transistors between the ground plane and the evaluation node). In the Mod 7 multiplier example, the critical path is Φ.

Floor Plan and Layout

Fig. 25 shows the floor plan and final layout of the Mod 7 multiplier, using the 3μ DLM p-well CMOS process [30]. The transistor block contains the matrix of primitives mapped from the switching tree, and also the metal2/polysilicon signal wires. Note that the figures have been rotated by 90°. The inverters are formed by p-channel and n-channel strips, separated by the tree matrix. The matrix also includes the ground switch transistors, and the input clock signal to the switches is buffered by an inverter at the end of the inverter strip. The latch primitives are full custom layouts, and the clock signal to the latches is also buffered at the bottom of the latch column.

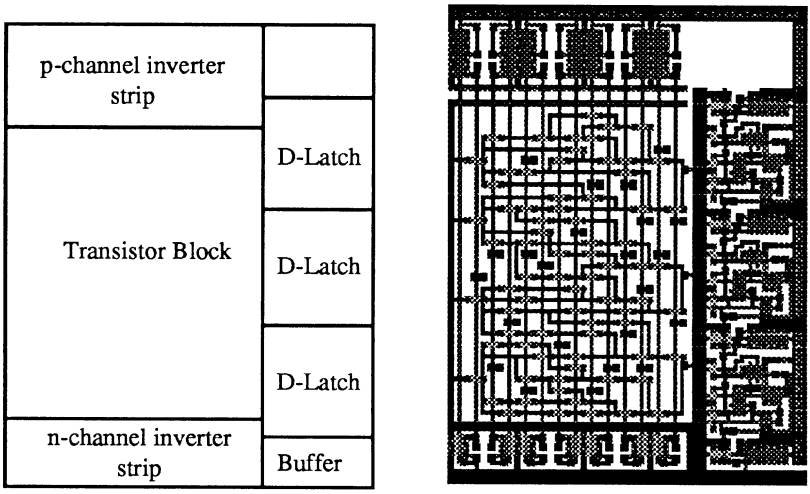


Fig. 25 Floor plan and layout for the Mod 7 multiplier

The transistor array governs the height of this particular example cell, but often the latches control the size, particularly for smaller numbers of inputs, or when there is a greater decomposition of the switching function (e.g. multi-bit binary adders). For such a cell, the area is now controlled only by the number of input bits (width) and number of output bits (height). For these low area switching functions, generating an optimal solution to the tree minimization is often not important; what is more important is control over cross connections (the simple algorithm in the previous section only works with a planar tree mapping) and it is a better choice to increase Φ if a planar tree is the result.

Comparison Study

Based on the statement, in the previous section, concerning layout area that is independent of switching function, it is useful to compare the automatic layout versus full custom hand layout, and also to compare the *Switching Tree* approach to alternative switching function implementations. We will use the Mod 7 multiplier as the example for layout comparison, and we choose a TSPC PLA design as the comparison architecture. The 3μ CMOS fabrication process is used for all comparison designs.

Hand Layout The approach for the hand layout is to map the trees individually and to surround the resulting latched trees with the appropriate buffers. The transistor sizes for the trees are based on the minimum width of the drain/source area afforded by the design rules. The channel widths are also set at this value (5.4μ). The floor plan is shown in Fig. 26.

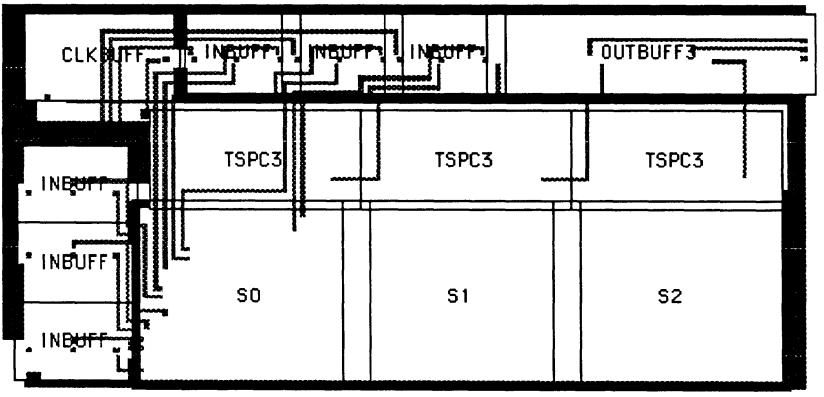


Fig. 26 Floor Plan for the Hand-Layout Design

TSPC PLA Design The PLA design uses the same TSPC latch structure as the switching tree approach. Because the PLA contains separate AND and OR planes, these have been incorporated into the n-channel block and p-channel block of the latch. This provides the ability to pipeline at the rate of the slowest of either block, rather than requiring a single evaluation of the complete structure or separating the planes into different latches. Channel widths are the same as for the *Switching Tree* designs. The floor plan is shown in Fig. 27 along with the multiplier core layout.

The truth table for the Mod 7 multiplier does not decompose sufficiently to allow folding of the core, as seen in Fig. 27.

Area Comparison The 3 designs are compared in Fig. 28. It is clear that the switching tree design has much lower area than the equivalent PLA design, but more surprising is the fact that the automatically generated tree design has lower area than the hand layout. Clearly the mapping of merged trees to a matrix placement is very efficient.

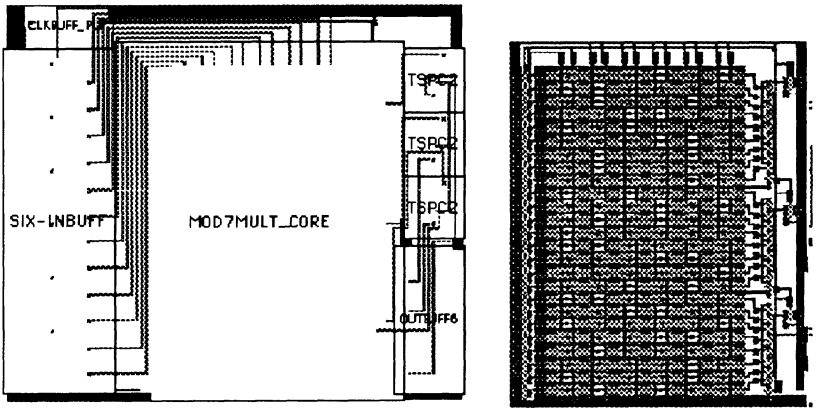


Fig. 27 PLA design: Floor plan and Mod 7 multiplier core

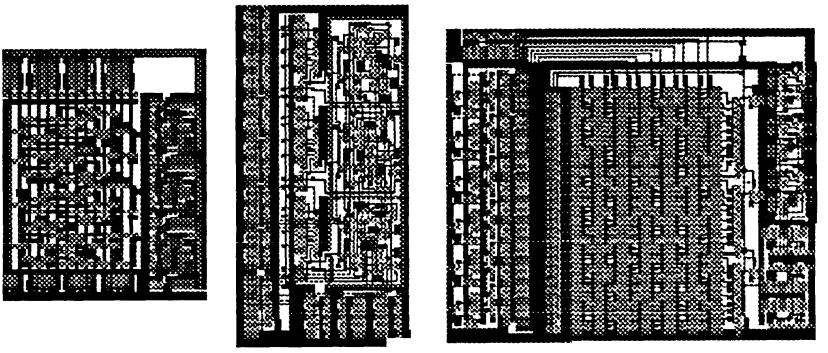


Fig. 28. Area comparison of the three designs

A more useful comparison is between the core areas of the three designs, since this will eliminate differences in the design of support circuitry (buffers, latches etc.). This comparison is given in Table 3.

Speed and Power Comparison This study has been conducted using mask extracted SPICE files, with level 3 models based on tuning from many fabrication experiments. The two switching tree designs perform almost identically, and so we provide a single result for the two designs. Comparison results are shown in Table 4. The power and peak current measurements are taken at a 40MHz throughput rate

Design	Synthesized Switching Tree	Hand-Layout Switching Tree	PLA
Core Area	252μ ×204μ = 0.0514 mm^2	163μ ×458μ = 0.0747 mm^2	363μ ×457μ = 0.1659 mm^2
Relative %	100%	145%	323%

Table 3 Core Area Comparison

Design	Switching Tree	PLA
Maximum Throughput Rate	50MHz	70MHz
Peak Current at 40MHz	2.85mA	8.36mA
Average Dissipation at 40MHz	3.45mW	15.4mW

Table 4 Speed and Power Comparison

We note that the PLA is able to operate at almost 50% higher throughput rates than the switching tree design; the trade-off, however, is the almost 5 times increase in power dissipation and 3 times increase in the peak current spike. This latter result can be as important as the power dissipation result, since the current spike is effectively

multiplied by the number of cells on the chip for perfectly synchronized clocking (no skew between clocks arriving at the cells). This also speaks for producing architectures that allow clocks to be skewed, and the number theoretic techniques, described in the first part of the paper, are directly suitable for such skewed clocking, since the computations are carried out in independent pipelines.

CONCLUSIONS

In this chapter we have discussed the role of special architectures and dynamic logic building blocks for the construction of data stream DSP processing systems. These architectures are optimized for computing on high bandwidth streams of data, where the input and output data stream at the same rate. We have discussed three different implementation procedures based on standard binary arithmetic, a form of redundant arithmetic for MSB first calculations and finally a massive parallel architecture based on number theoretic techniques for feedforward DSP algorithms. In particular we have concentrated on a recently introduced polynomial ring mapping technique which allows large dynamic range computations to be performed using massively parallel small finite ring computational elements. The technique allows direct mapping of bits of either a purely real or multiplexed bit coded complex number to a set of independent rings; to illustrate the technique we have demonstrated that large dynamic range computations can be performed by independent residue computations with the smallest usable odd relatively prime moduli of 3, 5 and 7. Although the use of such small rings in a traditional *RNS* system would yield an inadequate computational dynamic range, the new technique allows usefully large dynamic range computations with such moduli.

We have proposed the use of a building block based on dynamic logic with true single-phase clocked latches (*Pipelined Switching Trees*). We have also discussed a synthesis procedure for pipelined switching trees based on a mapping of the switching tree to a 2-dimensional matrix of layout primitives. We demonstrate that the mapping procedure is area efficiency by a comparison study with a hand-layout of the same circuit and also a PLA implementation. The synthesized layout is smaller than the hand layout (using a different decomposition procedure) and the switching tree is 3 times smaller than the PLA core. In SPICE simulations, the pipelined switching tree consumes only 20% of the power required by the PLA running at the same frequency. The PLA design, as expected, is able to run at throughput

rates that are 50% higher than the switching tree. Fabricated test cells of pipelined switching trees have been proven to operate at the 40MHz bandwidth of the output drivers.

REFERENCES

1. McCanny, J. V. and J. G. McWhirter. "Optimized Bit Level Systolic Array for Convolution." *IEE Proceedings, Pt. G.* 131 6, October 632-637, 1984.

2. Jullien, G. A. "Bit-Level Systolic Arrays for High Speed DSP." Advances in VLSI Signal Processing. 1993 Ablex Publishing. (In Print)

3. Yuan, J. and C. Svennson. "High-Speed CMOS Circuit Technique." *IEEE. J. Solid-State Circuits.* vol. 24 pp. 62-70, 1989.

4. McAuley, A. J. "Dynamic Asynchronous Logic for High-Speed CMOS Systems." *IEEE J. Solid-State Circuits.* 27 3 382-388, 1992.

5. Taheri, M., G. A. Jullien and W. C. Miller. "High Speed Signal Processing Using Systolic Arrays Over Finite Rings." *IEEE Trans. Selected Areas in Comm.* 6 3 1988.

6. Song, P. J. and G. DeMicheli. "Circuits and Architecture Trade-offs for High Speed Multiplication." *IEEE J. Solid-State Circ.* 26 9 pp. 1184-1198, 1991.

7. Afghahi, M. and C. Svensson. "A Unified Single-Phase Clocking Scheme for VLSI Systems." *IEEE J. Solid-State Circuits.* 25 Feb 225-233, 1990.

8. Jullien, G. A., W. C. Miller, R. Grondin, Z. Wang, D. Zhang, L. Del Pup and S. Bizzan. "WoodChuck: A Low-Level Synthesizer for Dynamic Pipelined DSP Arithmetic Logic Blocks." *IEEE International Symposium on Circuits and Systems.* 1 pp. 176-179, 1992.

9. Jullien, G. A. and M. A. Bayoumi. "A Review of VLSI Technologies in Digital Signal Processing." *Proceedings of the 1991 IEEE Int. Symp. on Circuits and Systems,.* (***Invited***) pp. 2347-2350, 1991.

10. Baji, T. and e. al. "A 20-ns CMOS Micro-DSP Core for Video-Signal Processing." *IEEE J. of Solid-State Circuits.* Vol. 23, No. 5, pp.1203-1211, 1988.

11. Goto, J., K. Ando, T. Inoue, M. Yamashina, H. Yamada and T. Enomoto. "250-MHz BiCMOS Super-High-Speed Video Signal Processor (S-VSP) ULSI." *IEEE J. Solid-State Circuits.* Vol. 26 No. 12 pp. 1876-1884, 1991.

12. Gebotys, C. H. and M. I. Elmasry. "Optimal Synthesis of High-Performance Architectures." *IEEE Journ. Solid-State Circuits.* Vol. 27, No. 3, pp. 389-397., 1992.

13. MacQuillan, S.E. and McCanny, J.V. "A VLSI Architecture for Multiplication, Division and Square Root." *IEEE International Symposium on Acoustics, Speech and Signal Processing.* pp. 1205-1208, 1992.

14. Hatamian, M. and K. K. Parhi. "An 85-MHz Fourth-Order Programmable IIR Digital Filter Chip." *IEEE Journ. Solid-State Circuits.* Vol. 27, No. 2, pp. 175-183., 1992.

15. Wigley, N. M. and G. A. Jullien. "On Moduli Replication for Residue Arithmetic Computations of Complex Inner Products." *IEEE Trans. Comp.* (In Print) 1990.

16. Games, R. A. "An Algorithm for Complex Approximations in Z[e2πi/8]." *IEEE Trans. Inform. Th.* IT-32 603-607, 1986.

17. Wigley, N. M. and G. A. Jullien. "Large Dynamic Range Computations Over Small Finite Rings." *IEEE Trans. Computers.* In Print 1992.

18. Jullien, G. A., R. Krishnan and W. C. Miller. "Complex digital signal processing over finite fields." *IEEE Transactions on Circuits and Systems.* CAS-34 4 pp 365-337., 1987.

19. Wigley, N. M. and G. A. Jullien. "A Flexible Modulus Residue Number System for Complex Digital Signal Processing." *IEE Electronic Letters.* 27 16 pp.1436-1438., 1991.

20. Wang, Z., G. A. Jullien and W. C. Miller. "Algorithms for Length 15 and 30 Discrete Cosine Transforms." *1991 Asilomar Conference on Circuits Systems and Computers.* November pp. 111-115, 1991.

21. Jullien, G. A. "Number Theoretic Techniques in Digital Signal Processing." Advances in Electronics and Electron Physics. 1991 Academic Press Inc. pp. 69-163.

22. Soderstrand, M. A., W. K. Jenkins, G. A. Jullien and F. J. Taylor. "Residue Number System Arithmetic: Modern Applications in Digital Signal Processing." . 1986.

23. Corry, A. and K. Patel. "Architecture of a CMOS Correlator." *Int. Conf. on Circuits and Systems*. 522-525, 1983.

24. Shoji, M. "FET Scaling in domino CMOS Gates." *IEEE. J. Solid-State Circuits*. vol. 20 pp. 1067-1071, 1985.

25. Chu, M. K. and D. I. Pulfrey. "Design procedures for differential cascode-Voltage switch circuits." *IEEE Trans. Solid-State Circuits*. vol.SC-21 no.6 pp. 1082-1087, 1986.

26. Shoji, M. "CMOS Digital Circuit Technology." 1988 Prentice Hall Inc.

27. Yuan, J. and C. Svensson. "Pushing the Limits of Standard CMOS." *IEEE Spectrum*, February, pp. 52-53

28. Bizzan, S., G. A. Jullien and W. C. Miller. "Analytical Approach to Sizing NFET Chains." *IEE Electronics Letters*. (In Print), 1992.

29. Elmore, W. C. "The transit response of damped linear networks with particular regard to wideband amplifiers." *J. Appl. Phys*. 19, No. 1 Jan. 55-63, 1948.

30. Canadian Microelectronics Corporation. "Guide to the Integrated Circuit Implementation Services of the Canadian Microelectronics Corporation." . 1986.

ACKNOWLEDGMENTS

The author acknowledges financial support from the Natural Sciences and Engineering Research Council of Canada, the Micronet Network of Centres of Excellence, and the fabrication and equipment loan programme of the Canadian Microelectronics Corporation, to carry out some of the work described in this chapter. The author is also indebted to Mr. R. Grondin, and Mr. L. Del Pup for the switching tree software package, the comparison study data and fabrication results.

10

A General Purpose Xputer Architecture derived from DSP and Image Processing

A. Ast, R. W. Hartenstein, H. Reinig, K. Schmidt, M. Weber
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, W-6750 Kaiserslautern, Germany

ABSTRACT

This paper illustrates a novel class of computational devices called Xputers, which are by up to several orders of magnitude more efficient than the von Neumann paradigm of computers. The paper shows how the new paradigm is partly derived from accelerating features of image processors and digital signal processors, and it illustrates xputer execution mechanisms and associated programming techniques by means of simple algorithm examples.

1. PREFACE

The xputer paradigm with its new *data-procedural* basic execution mechanisms and all its impacts on technology platforms and basic architectural building blocks, and, on application support techniques like languages, compilers and programming techniques is a major step away from the familiar world of traditional computing based control-driven procedural von-Neumann-based models. The main differences are:

- the ALU of an Xputer is **reconfigurable** (soft) such, that it does not really have a fixed instruction set, nor a hardwired instruction format
- that's why (procedural) data sequencing is needed, since instruction sequencing is not feasible: a **data counter** is used instead of a program counter
- this leads to a fundamentally new machine paradigm and a new programming paradigm

Algorithm	6800 (16 MHz) / millisec	MoM2 (10 MHz) / millisec	Acceleration factor
CMOS Design Rule Check	91330.20	39.0300	2340
Digital Filter	9126.40	29.4400	310
Lee Routing: seek S	42.50	0.0625	680
wavefront	70.00	0.3750	186
backtracking	23.25	0.1250	186

a)

Algorithm	Data Manipulation	Addressing	Control
CMOS Design Rule Check	7 %	93 %	<1 %
Digital Filter	28 %	58 %	14 %
Lee Routing: seek S	14 %	74 %	12 %
wavefront	6 %	92 %	6 %
backtracking	17 %	67 %	17 %

b)

Fig. 1: Performance Analysis: a) MoM-2 acceleration factors compared to Motorola 6800, b) overhead analysis on DEC VAX-11/750

Solutions for all this lead to a novel interdisciplinary approach such, that a reader usually is not completely familiar with all the backgrounds needed by the reader. To achieve a more detailed comprehensibility of all fundamentals and relevant aspects, such as basic execution mechanisms, architectural elements, new programming paradigms, compilation techniques, as well as the feasibility of the high efficiency, a book of several hundred pages would be needed. Since only limited space is available for this paper, major parts of it are organized more as an illustration. The sales pitch which sometimes seems to be visible in the presentation is motivated by our desire to convince other researchers, that the principles of the experimental hardware and software illustrated here, point out a way to a highly promising new R&D area worth to invest major efforts in investigating all its relevant aspects.

2. INTRODUCTION

For quite a number of commercially important applications extremely high throughput is needed at very low hardware cost. Often these goals cannot be met by using the von Neumann paradigm nor by ASIC design. In such cases even parallel computer systems [19] [41] or dataflow machines [8] do not meet these goals because of massive parallelization overhead (in addition to von Neumann

overhead) and other problems. Supercomputers, digital signal processors and image processors provide interesting high performance features, but do not well fit to general purpose application. We would like to propose a new computational approach: the *Xputer* machine paradigm. The term xputer clearly distinguishes his *data-procedural* execution model from the *control-procedural* model of von Neumann computers.

This paper first briefly illustrates the new machine paradigm and its basic execution mechanisms and then illustrates programming and execution of some example algorithms on the MoM xputer architecture.

For algorithms with regular data dependencies the Xputer paradigm is by several orders of magnitude more efficient than the von Neumann paradigm. Even for unstructured spaghetti-type sources the Xputer paradigm is at least half an order of magnitude more efficient. The high efficiency of Xputers has several reasons:

- Their data-procedural operational principles cope much better with most kinds of overhead and bottlenecks being typical to the von Neumann machine paradigm:
    - address computation overhead
    - control flow overhead,
    - ALU (multiplexing) bottleneck
    - processor-to-memory communication bottleneck
- Xputers support some compiled fine granularity parallelism inside their reconfigurable ALU (rALU).
- *A smart register file* with a *smart memory interface* contributes to further reduction of memory bandwidth requirements.
- Xputers are highly compiler-friendly by supporting more efficient optimizing compilation techniques, than possible for compilers for computers.

Commercial exploitation of Xputers is now becoming feasible by the progress and commercial availability of modern field-programmable technology [7].

Fast turn-around ASIC design. Recently field-programmable gate arrays have become available, which are compatible to particular real (mask-programmable) gate arrays. Due to code compatibility the personalization code of a field-programmable version can be easily translated into that of a real gate array (being faster and of higher integration density). Such conversions are carried out by *retargeting* software (e. g. [31]), which also provides an efficient bridge between computational paradigms and ASIC design. Compared to conventional ASIC this has the benefit, that simulation is replaced by execution being several orders of

magnitude more efficient. Recently considerable attention has turned over to the topic of retargeting. This indicates that the high significance of retargeting for the future trends has been widely recognized.

3. XPUTERS: WHY AND HOW?

This section first discusses the inefficiency of von Neumann processors, resulting from overhead phenomena and then briefly summarizes architectural acceleration features known as a remedy in image processing and DSP. Later in this paper it will be shown, that these features are important ingredients to (non-von-Neumann) Xputer architectures.

3.1. The von Neumann machine, its bottlenecks and overhead phenomena.

For improved comprehensibility this paper several times illustrates the differences to basic execution mechanisms of computers. These differences are of interest, since the principles of the machine platform has a strong impact on overall system performance, as well as on the efficiency of system programming and application development. Computers have the following bottlenecks and overhead phenomena:

- the ALU bottleneck
- accessing overhead
    - direct (address computation overhead)
    - indirect (data restructuring relocation overhead)
- control flow overhead
- processor-to-memory communication indigestion

The ALU bottleneck. Although is provides a rich repertory of operators, the hardwired ALU of computers is a narrow bandwidth device which mainly can carry out only a single simple operation at a time using only three or less operands. This lack of parallelism we call the ALU bottleneck. It is caused by a multiplexer, the decoder of which is driven by the hardwired instruction sequencer. Such an ALU has a fixed hardwired instruction. Microprogrammable von Neumann machines are nested machines, where the inner machine has a hardwired (micro) instruction set which exhibits the same typical von Neumann bottlenecks and overhead phenomena. This means, that microprogramming does not provide an escape from inefficiency.

Accessing overhead. The von Neumann needs CPU time not only for direct data manipulation, but also to compute direct data addresses (direct accessing overhead) and sometimes also to relocate data for rearrangement of data

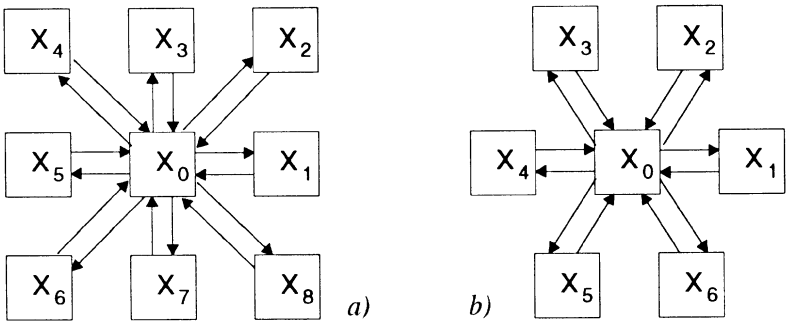


Fig. 2: Illustrating cellular machine use in image processing: array of PEs (Processing Elements) [35]): a) square array, b) hexagonal array.

structures (indirect accessing overhead). This is an important issue, since on a von Neumann platform the percentage of CPU time needed for address computation may be very high (up to 93%, see Fig. 1 b).

Control Flow Overhead. Many of the instructions within a (machine-)program are control instructions. These instructions are causing control flow overhead. This will be explained later (also see section 5.).

Processor-to-memory communication indigestion. The processor-to-memory communication channel is a performance bottleneck not only per se, but also by the high communication requirements due to the above overhead and bottleneck phenomena (memory access cycles for excessive instruction fetch, to save intermediate data values, to execute accessing overhead and control flow overhead). In parallel computer systems further performance degradation (per processor) stems from inter-processor communication overhead and other negative effects.

3.2. Cellular Architectures

Cellular architectures have successfully evolved as specialized image processors (Fig. 6), which efficiently support pixel operations, where the new value x_0' of the current center pixel is computed from its previous value x_0 and the values x_1 through x_8 of all its neighbor pixels (data dependencies: see Fig. 2 a), or, x_1 through x_6 (Fig. 2 b), respectively:

$$x_0' = f(x_0, x_1, \ldots, x_8) \qquad \textit{hexagonal kernel:} \quad x_0' = f(x_0, x_1, \ldots, x_6) \qquad (1)$$

A pixel and all its nearest neighbors are held by a 3-by-3 set of registers, often called *cellular registers* or *kernel*, which operates as a 3-by-3 window onto the

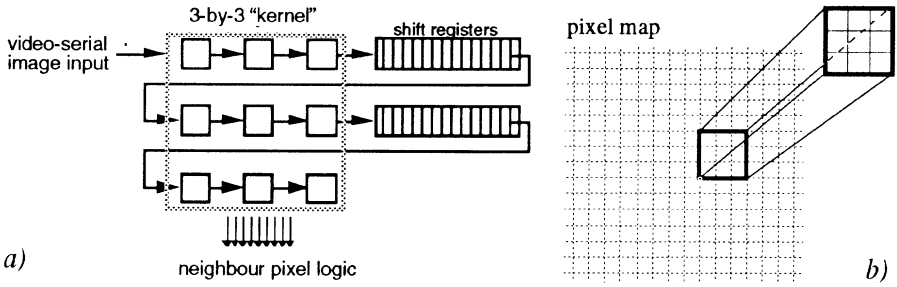


Fig. 3: 3-by-3 kernel: b) a window onto the pixel map, a) its video-serial implementation.

pixel map of the image (Fig. 3 b). (A hexagonal kernel has only 7 registers, see Fig. 2 b). Shift registers used as a delay support on-line processing of video-sequenced image pixels (Fig. 3 a).

The term *cellular (logic) machine* is used for earlier machines in which a single processing element PE is used to operate sequentially on all pixel values x_0 through x_8 [35]. Modern architectures (mostly called *cellular array* s) use a tightly coupled array of 9 PEs running in parallel (Fig. 2: each pixel x_i has its own PE_i) to speedup throughput.

A number of cellular logic machine concepts using a 3-by-3 kernel have been published in the 50s ([2], [21], [5], and [40]), or patented [10], respectively. The first implemented cellular logic machine is Cellscan [32] (based on the patent of Golay), followed by Glopr (Golay LOgic PRocessor) [33], BIP (Binary Image Processor) [11]. Due to the immature state of the art in technology all three of them have only a single PE for serial operation. Glopr is similar to the Cellscan, but uses a hexagonal cellular register kernel and features electrically alterable length of the shift registers ranging from 32x32 and 64x64 up to 128x128 pixels. BIP, using a 3-by-3 kernel, has been commercialized as part of the Grafix1 system for character recognition.

Later on *cellular arrays* have been developed: diff3 [12] for automatic microscope for human blood analysis has been the first cellular logic machine produced in larger quantities. The PEs operate by lookup tables holding the complete set of Golay primitives. The TAS (Texture Analysis System, [29]) is similar to the diff3 and has been commercialized by Leitz (Germany) in 1979.

Modern image processors are RAM-based, i. e. work on primary memory: the shift register is replaced by an address generator. Most of them are only pseudo-cellular architectures, where mainly only algorithms and register file reflect cellularity. The PHP (Preston-Herron Processor, [18], Fig. 4) loads the 3x3 kernel

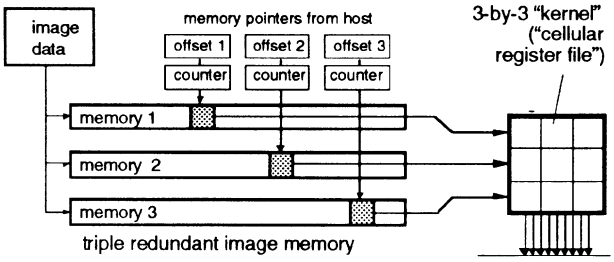


Fig. 4: Cellular register file of the PHP [18]).

from 3 RAM modules which hold identical (redundant) data to provide higher processor to memory communication bandwidth. The addresses to select the pixels are generated by simple counters. The TRO (TRiakis Operator [34]) is an improved version of the PHP having more memory modules and larger lookup tables which enable it perform 3-dimensional cellular transforms.

Picap1 (PICture Array Processor, [23], Fig. 3) combines properties of Glopr and BIP by including a template matching unit and a numerical convolver. Data transfer to the kernel may be serially; or, in parallel from nine image memories holding identical data. Picap2 [22] is 20 times faster and has a 20 times larger memory. The very powerful Cytocomputer [39] consists of a pipeline of 80 Cellscan-like processors with cellular registers. The DIP-1 (Delft Image Processor [9]) combines a 3-by-3 cellular kernel with some traditional computing features: additional two general-purpose ALUs and its multiplier are a step towards being a general purpose processor. Cellular architectures have influenced the development of the first Xputer architecture (MoM-1, formerly called PISA [17]).

The AIS-5000 [42] is a 1-dimensional or scan line processor, using a 1-by-n window instead of a 3-by-3 window. This architecture is a n-by-1 array with n between 128 and 1024. The PE array provides a PE for each column of the image. The memory associated to each PE holds an entire column and has space to store all intermediate results. Fig. 5 summarizes the features of the architectures described above.

High performance of image processors is achieved by minimizing both, direct addressing overhead (address computation) and indirect addressing overhead (rearrangement of data blocks) through hardware-supported windowing, so that for each pixel operation the right data are immediately available at the right time at the right place (in the kernel). In sequential cellular logic machines the 'right time' is achieved by waiting loops (shift registers) for data (Fig. 3 a), such that no

acronym (name), (published)	array size	Mega pixops /sec.	host	maker	features and application
Cellscan, ([32], 1963)	64 x 64	0.004	stand alone	Perkin-Elmer	first implem. of Golays patent[10]; 3x3 kernel; operates serially on neighbors; application: hematology
Glopr (Golay LOgic PRocessor), ([33], 1971)	32 x 32, 64 x 64, 128 x 128	0.3	Varian 620i	Perkin-Elmer	like Cellscan; performs operations based on all Golay primitives; hexagonal kernel is processed in parallel; application: hematology, gen. purp. image processing
BIP (Binary Image Processor), ([11], 1971)	programmable	0.3	PDP-11	Information Int'l Inc	template matching (9 templ. in parallel) on a hexagonal or 3x3 kernel; commercial application: character recognition
diff3 ([12], 1980)	64 x 64	40	Data General Nova-4	Coulter Electronics	first mass produced cellular architecture; 8 PEs in parallel; ROM f. Golay primitives appl.: automatic microsc. blood analysis
TAS (Texture Analysis System), ([29], 1979)	256x256	approx. 50	LSI-11	Leitz	similar to diff3 but faster; application: general purpose image processing
PHP (Preston-Herron Processor), ([18], 1982)	programmable (within limits)	1.5	Perkin-Elmer 3230	Perkin-Elmer	based on diff3, for 2-dimensional cellular transforms; 16 PEs in parallel; memory instead of shift registers
TRO (TRiakis Operator), ([34], 1983)	programmable (within limits)	1.5	Perkin-Elmer 3230	Perkin-Elmer	based on PHP, but 3-dimensional cellular operations
Picap1 (PICture Array Processor), ([23], 1973)	64 x 64 4-bit grey level	0.8		Univ. Linköping	combines Glopr and Bip, 3x3 kernel; application: fingerprint analysis
Picap2, ([22], 1982)	512 x 512	16	Syst.Eng. Labs 77/35	Univ. Linköping	20x faster Picap1; multiprocessor system: 7 PUs; application: general purpose image processing
Cytocomputer, ([39], 1978)	512 x programmable	120	PDP-11 or VAX 11	ERIM*	pipeline of 80 Cellscan-like PEs; each with 3x3 kernel; application: air force automatic target detection
DIP-1 (Delft Image Processor), ([9], 1981)	256 x programmable	0.66	HP 1000	Univ. Delft	combines cellular logic hardw. w. gen. purpose ALUs: applic.: medical microscopy and industrial materials inspection
MoM-1 (Map-oriented Machine), ([15], 1988)	programmable kernel, max. 5 x 5	0,8	ELTEC image processor	Univ. Kaiserslautern	data-procedural general purpose processor with scan window & reconfigurable ALU using programmable logic ICs;
AIS-5000 [42]	programmable: 1x128 - 1x1024				scan line processor, massively parallel

Fig. 5: Summary of features of cellular machines

address is needed. RAM-based image processors reach this goal by multiple address generators (Fig. 4) running in parallel with other hardware and by redundant or interleaved memory: obviously the kernel organization directly points out an efficient storage scheme for redundant or interleaved memory use.

3.3. Digital Signal Processors (DSPs).

Some digital signal processors (DSPs) provide hardware support to accelerate addressing. For example, the Texas Instruments TMS 320C25 [3] has an additional register arithmetic unit (ARAU) for address computation, supporting auto-increment, auto-decrement addressing. and bit-reversal addressing (useful for fast fourier transforms (FFT)). A *Repeatcounter* supports linear address sequences such, that for repetitive use of the same statement does not require repeated instruction fetch (to avoid control flow overhead).

The DSP Motorola 56000 [28] uses an address generation unit (AGU) consisting of two address ALUs and several registers, to automatically generate addresses for two operands in parallel. Each ALU is capable to update an address registers in a single machine cycle. This update operation is performed by one of the following add operations: add, two's complement add, increment by one, decrement by one, reverse-carry add, and modulo add. The latter is a special addressing mode useful to built circular buffers, or for sequential addressing within multiple tables or arrays.

The addressing feature of a DSP, in a wider sense, is comparable to that of vector processors, if we neglect the address dispatcher for interleaved memories in the latter. These address sequences can be used for linear addressing, modulo addressing, and bit-reverse addressing. These address sequences are generated automatically without use of the main ALU, reducing addressing overhead.

4. INTRODUCING the XPUTER

Main stream high level *control-procedural* programming and compilation techniques are heavily influenced by the underlying von Neumann machine paradigm. Most programmers with more or less awareness need a von-Neumann-like abstract machine model as a guideline to derive executable notations from algorithms, and, to understand compilation issues. Also programming and compilation techniques for Xputers need such an underlying model, which, however, is a *data-procedural* machine paradigm, also called *data sequencing paradigm..*

This section introduces and illustrates the basic machine principles [19]. Then the MoM-4 architecture is described, which later will be used as a vehicle to illustrate execution mechanisms via simple algorithm examples. Other examples will illustrate MoPL-3, a data-procedural programming language. This paper also tries to show the reasons of the good performance results having been obtained experimentally (e. g. see Fig. 1 a).

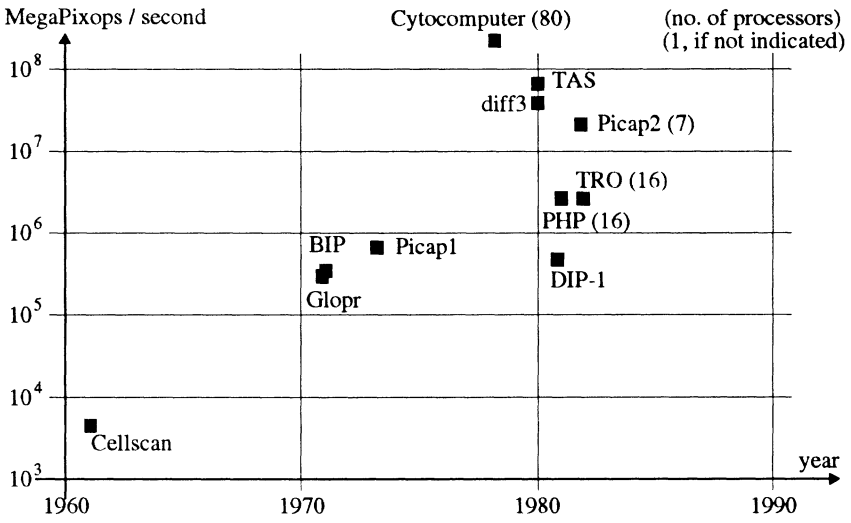


Fig. 6: Evolution of cellular machines for image processing.

4.1. Xputer Machine Principles.

Fig. 7 b illustrates the basic xputer architecture principles. The key difference to computers is, that data sequencer and a reconfigurable ALU replace computers' program store, instruction sequencer and the hardwired ALU (this view is simplified). For operator selection instead of the sequencer another unit is used, which we call *residual control.*

Smart Register File. Due to their higher flexibility (in contrast to computers) xputers may have completely different processor-to-memory interfaces which efficiently support the exploitation of parallelism within the rALU. Such an interface we call a *scan cache.* It implements a hardwired window to a number of adjacent locations in the memory space. Its size is adjustable at run time. Such a scan window may be 'placed' onto a particular location in memory under control a data sequencer. The scan cache is a generalization of the 3-by-3 or hexagonal kernel used by cellular or pseudo-cellular image processors (for a survey see section 3.2.). A sequence of locations we call a *scan path* or *scan pattern* (for examples see later).

The Data Sequencer. The hardwired data sequencer features a rich and flexible repertory of *scan patterns.* for moving scan caches along scan paths within memory space (e. g. see Fig. 7 c). Address sequences needed are generated by

hardwired *address generator*s having a powerful repertory of generic address sequences. For more details see later. After having received a scan pattern code a data sequencer runs in parallel to the rest of the hardware without stealing memory cycles. This accelerates xputer operation, since it avoids performance degradation by addressing overhead.

Similar acceleration features are known from instruction sets with auto-increment features, from a digital signal processor with a bit reversal addressing feature [28] (see section 3.3.), and from DMA controllers with very simple linear address sequences, mainly as needed for block transfers. The above control-procedural processor [28] even has an auto-increment feature for instruction iteration, where even instruction fetch iteration is suppressed (a pseudo-data-procedural mode, which avoids control flow overhead). For the xputer, however, the data sequencer is general purpose device covering the entire domain of generic scan paths, which directly maps the rich repertory of generic interconnect patterns (see [37] et al.) from space into time to obtain wide varieties of scan patterns, like e. g. video scan sequences, shuffle sequences, butterfly sequences, trellis sequences, data-driven sequences, even nested sequences, and many others. Instead of being a special feature it is an essential for xputers: the basis of the general purpose machine paradigm.

But we did not find publications of any address generators Xputers avoid a major part of such overhead by data auto sequencing and residual control, which mainly are effective in operation iterations and in local branching.

Reconfigurable ALU. Xputers (Fig. 7a) have a reconfigurable ALU (rALU), partly using the technology of field-programmable logic. Fig. 7a shows an example: the rALU of the MoM-4 Xputer architecture. The four smart register files called scan caches are explained later (lower left side in Fig. 7a). The MoM-4 rALU has a repertory of hardwired operator subnets (see lower right side in Fig. 7a). Within the field-programmable part of the rALU additional operators needed for a particular application may be compiled by logic synthesis techniques (upper right in Fig. 7a) A global interconnect-programmable structure (centre in Fig. 7a) is the basis of connecting these operators to form one or more problem-specific compound operators, what will be illustrated later by a simple algorithm implementation example.

rALU Configuration is no Microprogramming. Also microprogrammable von Neumann processors have a kind of reconfigurable ALU which, however, is highly bus-oriented. Buses are a major source of overhead [16], especially in microprogram execution, where buses reach extremely high switching rates at run time. The intension of rALU use in xputers, however, is compound operator configuration at compile time (downloaded at loading time) as much as possible,

so that path switching activities at are minimized and the underlying organizational overhead is pushed into compile time to save the much more precious run time.

Compound Operators. The rALU may be configured such a way, that one or more sets of parallel data paths form powerful compound operators which need only a single basic clock cycle to be executed. This rALU uses no fixed instruction set: compound operators are user-defined. Since their combinational machine code is loaded directly into the rALU, xputers do not have a program store nor an instruction sequencer. Instead a data sequencer is used which steps through the data memory to access the operands via register files called *data scan caches*. Xputers operate data-driven but unlike data flow machines, they feature deterministic principles of operation called *data sequencing*.

Summary of Xputer Principles. The fundamental operational principles of Xputers are based on *data auto sequencing* mechanisms with only *sparse control*, so that Xputers are deterministically *data-driven* (in contrast to data flow machines, which are indeterministically data-driven by arbitration and thus are not debuggable). Xputer hardware supports *fine granularity parallelism* (parallelism below instruction set level: at data path or gate level) in such a way that internal communication mechanisms are more simple than known from parallel computer systems.

5. THE MoM XPUTER ARCHITECTURE

To use a practical and comprehensible example for illustration of the novel task of the compiler a simple algorithm example implementation on the MoM Xputer architecture will be used. This MoM (Map-oriented Machine) has a two-dimensional memory organization and uses some extra features which further support optimization efforts of the compiler: the concept of the scan cache (a smart register file: featuring some hardwired smartness).

Scan Cache. The MoM-4 has 4 scan caches (Fig. 7b and 2c) operating as two-dimensional windows, adjustable at run time (some size examples in Fig. 7c) up to a maximum size (5 by 5). Each cache can be used to read, write, or read and write data from and to the data memory. A similar scan cache concept has been used earlier ([41], [39]), but never as an essential of a machine paradigm [15], [19] like the xputer.

Parallel Data Sequences. Since the MoM-4 has *multiple scan caches*, several such data scan caches may be connected to the same compound operator, so that they may run in parallel (e. g. see Fig. 10c and line (33) thru (36) in Fig. 17)

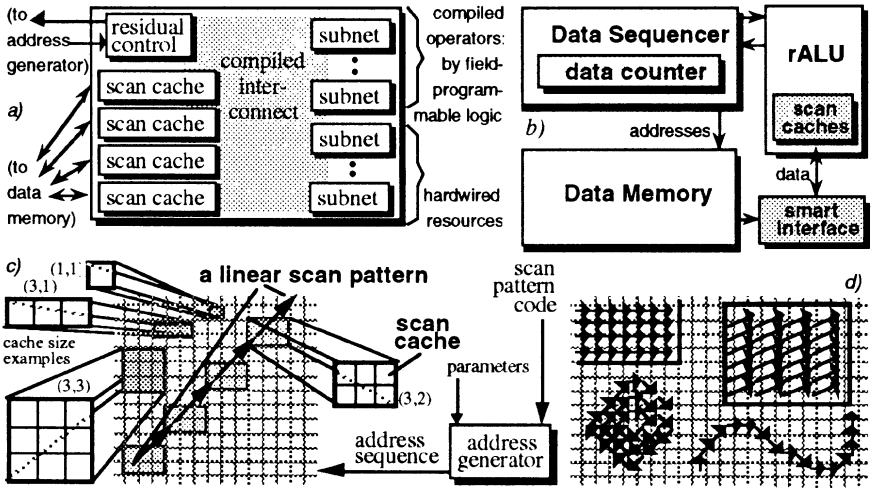
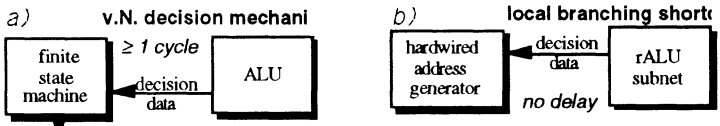


Fig. 7: Basic structures of Xputers and the MoM architecture: a) reconfigurable ALU (rALU) of the MoM, b) basic structure of Xputers, c) MoM cache size examples (left side) and a scan pattern example, d) a few other scan pattern examples.

communicate with each other through the rALU (Fig. 10e). By redundant multiple memory (like in [19] and [24]) or interleaved memory use such parallel data sequencing provides a substantial throughput improvement.

Hardwired Data Address Generator. For the sequencer of the MoM a parameter-driven powerful hardwired address generator has been developed [14]. Examples of such data scan patterns are single steps as well as longer generic scan sequences, such as *video scan sequences*, *shuffle sequences*, *butterfly sequences*, *trellis sequences* and others (for a very few examples see Fig. 7d). The data scan patterns can be adjusted in parameter registers of the data sequencer or they can be evoked by the decision data feedback loop from the r-ALU (Fig. 8b). With this feedback loop data-dependent cache movements can be performed (e. g. lower left scan pattern in Fig. 7d).

Residual Control. Also *tagged control words* having been inserted sparsely into the data memory map (Fig. 8c) can be recognized and decoded within the rALU to derive suitable decision data to select the next scan pattern. This kind of control we call *residual control*, because the decoder within the rALU is called only upon request and does not steal cycles from primary memory. By this means a sequence of several data scan patterns can be executed. Also residual control os a source of improved efficiency, what will be illustrated in section 5.1.



c)

kind of escape	caused by:	source of control word
normal escape	end of scan flag	TCW•>RCW
delimiter escape	TCW encounter	TCW
off-limits escape	address overflow	RCW
conditional escape	decision data	rALU
event escape	external event flag	RCW

terminology:
TCW tagged control word
TAW tagged address word
RCW remote control word
RAW remote address word

Fig. 8: MoM Decision mechanisms: b) branching hardware more efficient than a) von Neumann branching, c) type of escapes from scan patterns

Efficient Branching Mechanisms. The MoM architecture provides branching mechanisms which are more efficient than those known from von Neumann architectures. Von-Neumann-type branching requires one or more control accesses to primary memory, because only after the decision the next control state is known (Fig. 8). Depending on the kind of loop exit control code the number of memory accesses may be higher, even if no address computation is involved (which would cost further memory address cycles). For a number of cases such as e.g. nearest neighbor transitions in data memory space (in curve following, for example).

The MoM architecture provides more efficient branching modes, where usually no control action at all is needed, what saves memory cycles (Fig. 8b). For data-dependent scan patterns this is achieved by direct manipulation of the least significant data address bits by decision data bits. Because this decision data bypasses the sequencer we call this mechanism a *local branching short-cut*.. Other branching modes (called *escape* modes) also avoiding control flow overhead, are handled within the address generator (see next paragraph). For instance, lines (46), (48), (50), (56), (58), and (60) in Fig. 20 (`while` and `until` clauses) refer to the 4 array limit parameter registers (within the address generator) specifying the 4 borders of the array `PixMap` (compare Fig. 19). Fig. 8c lists all types of escapes available. For more details see section 6.2.3.

5.1. Illustration of the MoM execution mechanism

The following algorithm execution example demonstrates the essentials of the Xputer execution mechanism and illustration the task of the new kind of compilers needed [41]: a kind of fine granularity scheduling of caches, rALU subnets, and of data words. Fig. 9a shows the algorithm in a textual high level

language notation, Fig. 9b its graphical representation: a signal flow graph (SFG). Fig. 9d illustrates, how this algorithm is executed on the MoM Xputer architecture. The upper side of Fig. 9d shows the scan cache (format: 1 by 4 words), the rALU subnet for the compound operator (also compare Fig. 9a and Fig. 9b) and the interconnect between cache and subnet. The register inside the rALU subnet saves memory accesses, because the intermediate operands c(0) through c(7) do not need to move to memory. The bottom of Fig. 9d shows the data map (location of operands in memory.

The execution will run as follows. Starting at the left end of the data map The scan cache. The scan cache scans the data map area from left end (location shown in Fig. 9d) to the right end (shaded rectangle at the right of the data map in Fig. 9d). The sequence of arrows below the data map shows the scan pattern having.

Note, that no control action is needed because the *auto-execute* mode, where, whenever the scan cache is placed somewhere in the memory space (i. e. at each step of a scan), two things are carried out automatically (i.e. without having been called explicitly): the movement of data between scan cache and memory (*auto-xfer* mode) as well as the application of the marked rALU subnet to the variables held by the scan cache (what we call *auto-apply* mode). Note that by access mode tags only a minimum of memory semi cycles is carried out: read-only tags for all 4 words by this example. In our example 8 steps (x width=1, y with = 0) are carried out (Fig. 9d shows initial and final cache locations). From this example the task of the compiler may be summarized:

- define a data map (storage scheme)
- select a scan cache size and format
- define a compound operator and its scan cache interconnect
- select a suitable scan pattern available from address generators
- for linkage select a TCW and place it into the data map

At the end of the above data sequence example the cache finds a *tagged control word* (TCW) which then is decoded (right side of the map in Fig. 9d) to change the state of the *residual control logic* to select further actions of the Xputer. This sparse TCW insertion into data maps we call *sparse control*. Note that the control state changes occur only after many data operations (driven by the data sequencer).

5.2. A Multi-Cache Example

Fig. 10 shows an algorithm implementation example, a 16 point constant geometry FFT, where three scan caches run in parallel. Fig. 10 a shows the signal flow graph and the storage scheme (the grid in the background). The 16 input data points are stored in the leftmost column. Weights w are stored in every second column, where each second memory location is empty (for regularity reasons).

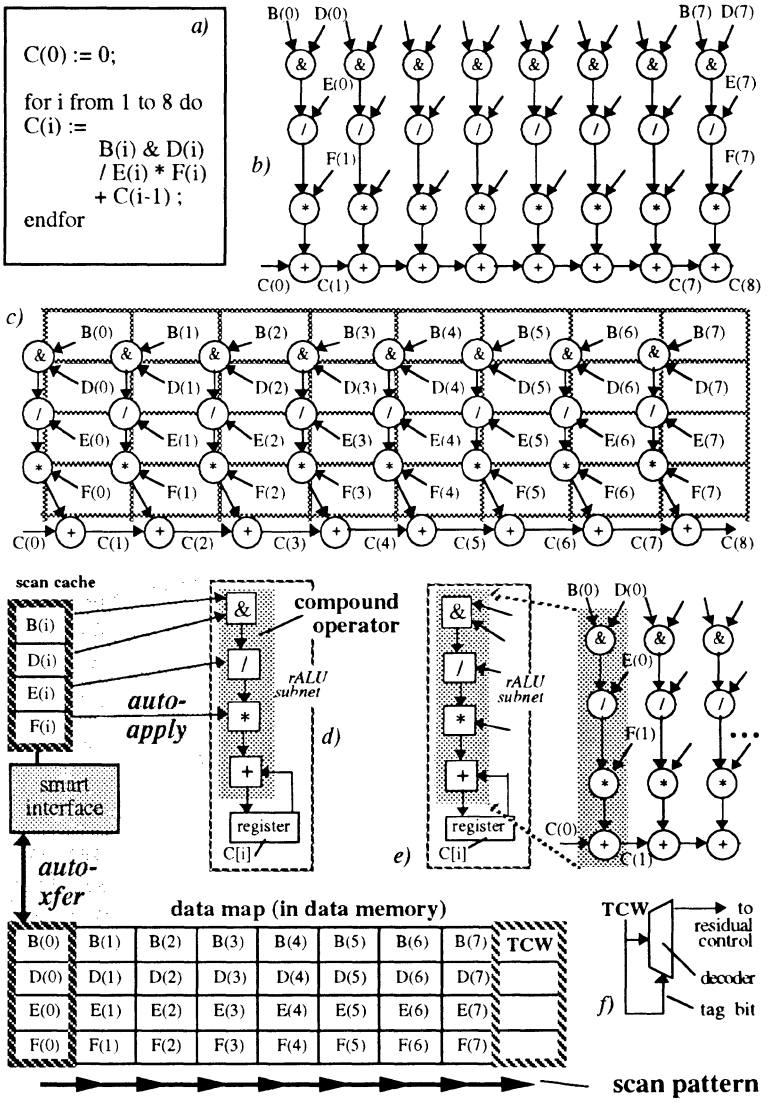


Fig. 9: Simple systolizable algorithm MoM execution example illustrating the compilation task: a) textual algorithm specification, b) graphic version of specification: signal flow graph (SFG), c) deriving a data map from SFG, e) deriving a compound operator for rALU from SFG, d) deriving scan cache size, rALU interconnect and scan pattern (also illustrating auto-apply and auto-xfer operation: needed for data-procedural machine principles)

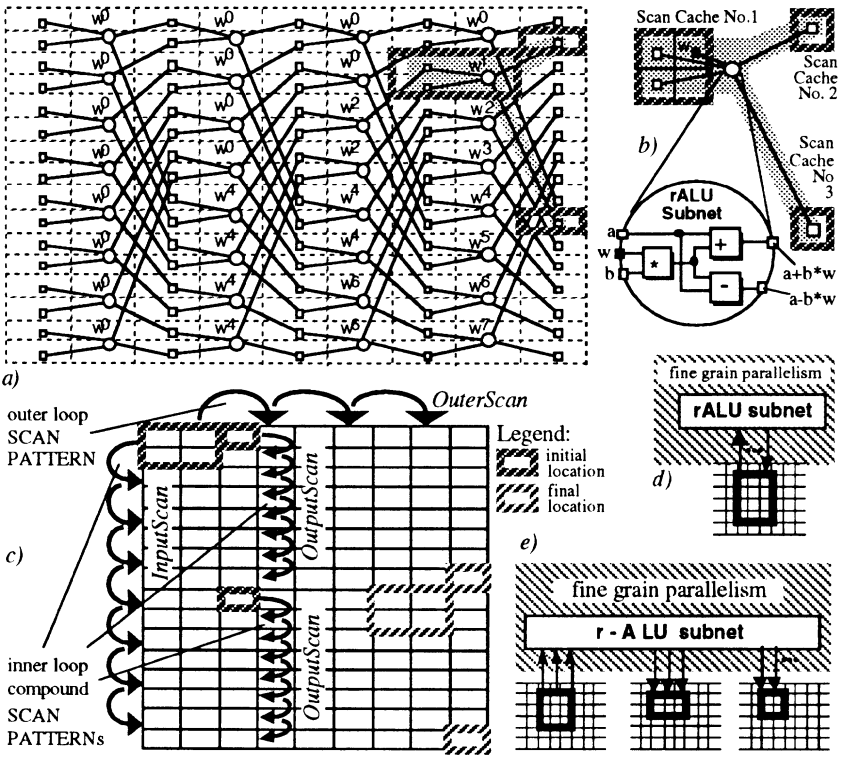


Fig. 10: Constant geometry FFT algorithm 16 point example using 3 scan caches synchronously in parallel: a) signal flow graph with data map grid and a scan cache location snapshot example, b) deriving rALU subnet, scan cache sizes and interconnect from compound operator, c) nested scan pattern illustration, d) illustration of fine grain parallelism: single cache use, e) multiple cache fine grain parallelism: field-programmable rALU as a communication mechanism.

Fig. 10 b shows the cache adjustments: the 2-by-2 cache no. 1 is the input cache reading the operands a and b, and the weight w. Caches no. 2 and 3 are single-word result caches. Fig. 10 b also shows the compound operator and its interconnect to the three caches. This is an example of fine granularity parallelism, as modeled by Fig. 10 e, where several caches communicate with each other through a common rALU. Fig. 10 c illustrates the nested compound scan patterns for this example. Note, that with respect to performance this parallelism of scan caches makes sense only, if interleaving memory access is used, which is supported by the regularity of the storage scheme and the scan patterns.

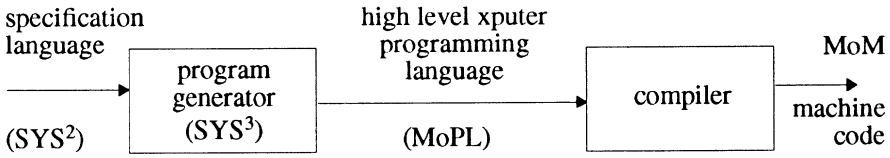


Fig. 11: MoM xputer language levels and translators

6. A PROGRAMMING LANGUAGES FOR XPUTERS

This section introduces two languages (also see Fig. 11): a specification language SYS^2 and a high level xputer programming language MoPL-3 (Map-oriented Programming Language) which is easy enough to learn, but which also is sufficiently powerful to explicitly exploit the hardware resources the xputer offers. For an earlier version of this language we have developed a compiler [41].

6.1. SYS^2: Mapping systolic arrays onto xputers.

We have experimented with an approach using SYS^2 very high level specifications as a source input. With this approach we have examined some program generation techniques, which transform high level specifications into an equivalent high level xputer program (MoPL-1). Since an xputer scan cache provides neighborhood communication very efficiently [13], it is most promising to adapt techniques from the area of automatic synthesis of systolic arrays [26], briefly called *systolic synthesis* ([6], [25], [27] et al.); for an introduction to systolic arrays see [24], [30], [36], or others. Systolic Synthesis makes use of nearest neighbor communication within a VLSI processor array by projecting the data dependence graph of an algorithm into time and physical processor space. Systolic synthesis can handle only *systolic algorithms* or *systolizable algorithms* (i. e. algorithms which can be converted into systolic algorithms), which are algorithms with regular data dependencies. (For a survey on systolizable algorithms see [24].)

Projection techniques from systolic synthesis have been adapted for parallelizing compilers for parallel computer systems. In the scene of parallel computing such techniques are called *systolizing compilation*, where the usual processes are modeled by the processing elements known from systolic synthesis, so that a concurrent implementation is derived. An xputer, however, is a monoprocessor. The problem therefore is to map the spatially distributed parallelism onto a data sequencing scheme suitable for xputers. For illustration let us use a 3 by 3 matrix multiplication example:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \qquad (2)$$

Several systolic synthesis systems, which have been implemented recently, usually accept nested loop notations as high level specifications ([6], [25], [27]). Such specifications look procedural, but the semantics is quite different: the intension is to express data dependencies, but not an order of execution (compare Fig. 9 a and b). Expressed in SYS^2, the source language for the SYS^3 systolic synthesis system [25], the above matrix multiplication example is specified by the following source text:

```
for I := 1 to 3 do                                    (1)
   for J := 1 to 3 do                                 (2)
      for K := 1 to 3 do                              (3)
         C[I,J] := C[I,J] + A[I,K] * B[K,J];          (4)
```

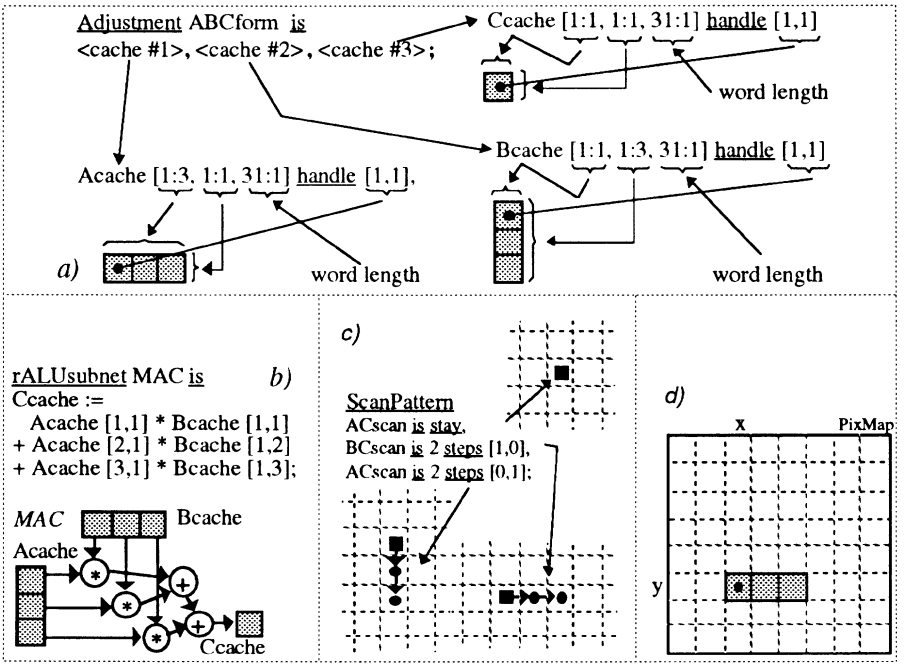


Fig. 12: Illustrating the declaration part of the MoPL program for multiple-cache 3-by-3 matrix multiplication: a) scan cache format adjustments ABCforms, b) compound operator MAC, three scan patterns Ascan, BCscan, and ACscan.

```
Array           A, B, C [1:3, 1:3, 31:0];                          (5)
                                                                   (6)
Adjustment  ABCform  is                                            (7)
               Acache [1:3, 1:1, 31:1] handle [1,1],               (8)
               Bcache [1:1, 1:3, 31:1] handle [1,1],               (9)
               Ccache [1:1, 1:1, 31:1] handle [1,1];              (10)
                                                                  (11)
rALUsubnet MAC is  Ccache = Acache [1,1] * Bcache [1,1]           (12)
                          + Acache [2,1] * Bcache [1,2]           (13)
                          + Acache [3,1] * Bcache [1,3];          (14)
                                                                  (15)
ScanPattern is  Ascan  stay,                                      (16)
                BCscan 2 steps [1,0],                             (17)
                ACscan 2 steps [0,1];                             (18)
```

Fig. 13: MoPL declaration part for matrix multiplication example in Fig. 12

The program generator having been implemented at Kaiserslautern generates a MoPL-1 program (MoPL-1 is an earlier version of MoPL-3). Next section describes a MoPL-3 program solution of this algorithm example.

6.2. MoPL-3: A Data-procedural Programming Language

This section introduces the essential parts of the language MoPL-3 and illustrates its semantics by means of three program text examples: the above 3-by-3 matrix multiplication, the constant geometry FFT algorithm from Fig. 10, and the data sequencing part for the JPEG zigzag scan being part of a proposed picture data compression standard. MoPL-3 is an improved version of MoPL-2 having been implemented at Kaiserslautern as a syntax-directed editor [41].

The Language MoPL-3 is an extended dialect of the programming language C. The main extension issue is the *data location* or *data state* such, that we simultaneously have two different kinds of location or state. There is the familiar von-Neumann-type *control state (current location of control)*, which e. g. is handled by *goto* statements referencing control label locations within the program text, or, by other control statements. During execution of xputer programs such a *control state* is coexisting with one or more *data location states*, what will be illustrated subsequently. (The *control flow* notation does not model the underlying xputer hardware very well, since it has been adopted from C for compatibility reasons to minimize programmer training efforts.)

```
begin                                                              (19)
      adjust ABCforms;                                             (20)
      apply  MAC;                                                  (21)
      moveto A[1,1], B[1,1], C[1,1];                               (22)
      fork                                                         (23)
        ACscan(Ascan), Ascan(BCscan), ACscan(BCscan);              (24)
      join                                                         (25)
end                                                                (26)
```

Fig. 14: MoPL statement part for matrix multiplication example in Fig. 12.

6.2.1. Matrix multiplication example

In addition to current control locations MoPL-3 programs also have *current data locations*, which are manipulated by `moveto` statements and scan patterns. Such a current data location is the current location of the scan cache. The statement `moveto A[1,1]`, for instance, says: move the cache to the location, where the variable `A[1,1]` is stored. A current data location does not change unless a data flow statement is encountered. I. e. after completion of a scan pattern the cache does not change its location, until another scan pattern or a `moveto` statement is encountered. In case of multiple scan cache use the MoPL program has multiple current data locations. For example the statement `moveto A[1,1], B[1,1], C[1,1]` says: move physical cache no. 1 to `A[1,1]`, cache no. 2 to `B[1,1]`, and, cache no. 3 to `C[1,1]`. The following two MoPL program examples illustrate the issue of current data location.

Lines (5) thru (18) in Fig. 13 show the declaration part of the matrix multiplication example. In line (5) the operand matrixes (arrays) `A` and `B`, and the result matrix `C` are declared. In line (7) thru (10) the size adjustments are declared for the physical scan caches number 1 thru 3 (also see Fig. 12 a). The handle point preceded by the keyword `handle` indicates the particular word location within the cache, which defines *current cache location* for address generator and user. See example in Fig. 12 d, where the current data location is `PixMap[x,y]`. In line (12) thru (14) the compound operator named `MAC` is declared (see Fig. 12 b). In lines (16) thru (18) three scan patterns are declared which are named `Ascan`, `BCscan`, and `ACscan` (see Fig. 12 c). At declaration time scan patterns are not yet assigned to a physical scan cache, nor a starting point is defined.

Lines (19) thru (26) in Fig. 14 show the statement part of the MoPL matrix multiplication program. The `adjust` statement in line (20) assigns a predeclared format or format list (here: `ABCforms`) to physical scan caches. This adjustment remains effective until another adjustment statement is encountered. The `apply`

```
array                CGFFT [1:9,1:16,31:0]                    (27)
ScanPattern          InputScan  is  7 steps  [0,2];          (28)
                     OutputScan is  7 steps  [0,1];          (29)
                     OuterScan  is  3 steps  [2,0];          (30)
                      .                                      (31)
                      .                                      (32)
     moveto CGFFT[1,1],[3,1],[3,9];                          (33)
     OuterScan (  fork                                       (34)
                    InputScan,OutputScan,OutputScan;         (35)
                  join   )                                   (36)
end                                                          (37)
```

Fig. 15: MoPL program of constant geometry FFT scan from Fig. 10.

statement in line (21) activates the predeclared compound operator named `MAC` and keeps it effective until another apply statement is activated. The `moveto` statement in line (22) is the kind of *data goto*, which makes the caches no. 1 thru 3 jump into a the particular locations indicated within this statement (also see first paragraph of this section).

Line (24) shows calls to predeclared scan patterns named `ACscan` etc. (compare line (16) - (18)). These calls activate scanning actions starting from the current data locations. Note, that a call to a scan patterns is a call to a loop. The expression `ACscan(Ascan)` in line (24) indicates a call to a nested scan pattern, where the scan pattern `Ascan` is called by the scan pattern `ACscan`. This means to call a loop by a loop, i. e. to call nested loops. The `fork` / `join` brackets (23) (25) around these three scan calls show, that the three scan actions run in parallel synchronously. Due to MoPL semantics the sequence of scan calls within the fork list refers to the order of physical caches no. 1, 2, and 3.

Fig. 16 illustrates the hierarchy of nested scan patterns of our example algorithm (rows 1 thru 3: outer loop, rows 4 - 6: inner loop) and shows the snapshots (rows 7 thru 9) of the sequence of triple cache locations created by these nested scan patterns. Whenever by SP1(SP2) a scan pattern SP1 calls a scan pattern SP2 this means, that SP1 determines nothing else than the sequence of start locations of SP2. The list of scan names within the fork/join clause refers to the (by lines (7) thru (10)) predeclared numbering scheme of physical caches: `ACscan(Ascan)` is applied to scan cache no. 1, `Ascan(BCscan)` to scan cache no. 2, etc.

6.2.2. Constant geometry FFT example

Next MoPL text sample shows in lines (34) thru (36) the nested scans of the FFT algorithm example in Fig. 10c (scan pattern declarations in line (28) thru (30).

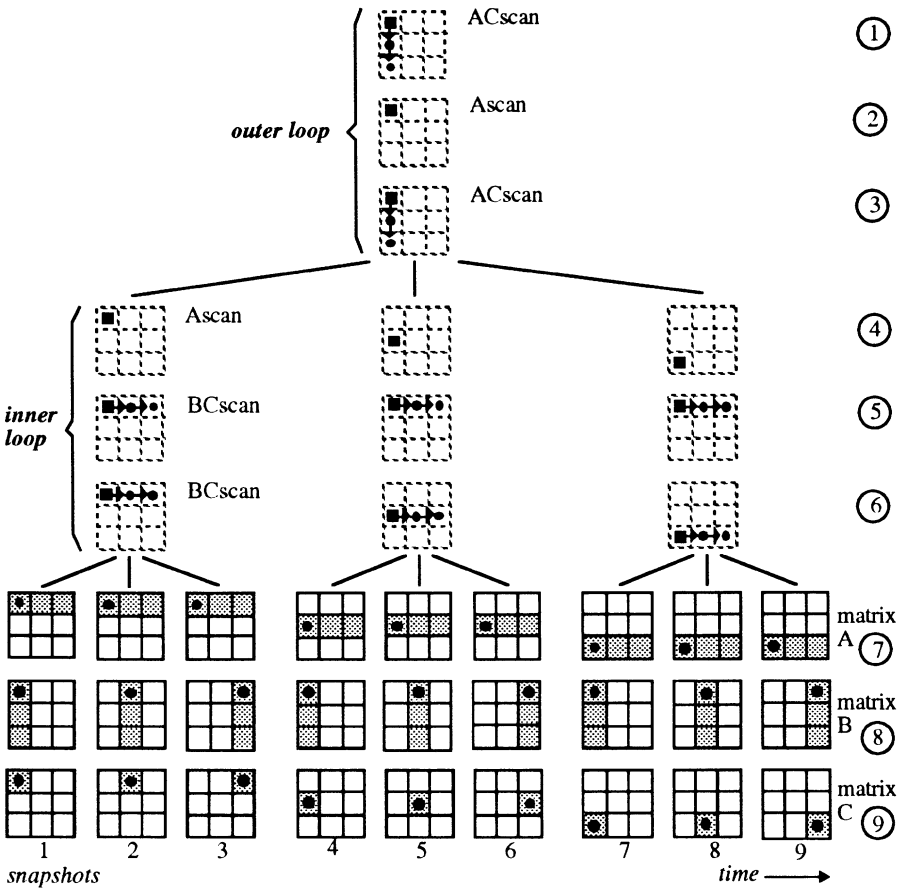


Fig. 16: Illustration of scan pattern execution for the 3 by 3 matrix multiplication example: the hierarchy of nested scan patterns (row 1 - 3: outer loop, row 4 - 6: inner loop, row 7 - 9: snapshot sequence of scan cache locations within matrixes).

6.2.3. JPEG zigzag scan example

The MoPL text from Fig. 18 illustrates programming the JPEG zigzag scan pattern (Fig. 17) named `JPEGzigzagScan` for scanning the array `PixMap` declared in line (38). This example uses a single 1-by-1 scan cache (adjusted as a single word buffer), which illustrates, that the performance benefit by the address

generator can be obtained also for accessing long sequences of single memory locations. Lines (39) thru (42) declare four scan patterns (also see Fig. 17), where the statements have the form:

<name_of_scan_pattern> <maximum_length_of_loop> STEPs <step_vector>.

The step vector specifies the next data location relative to the current data location (before executing a step of the scan sequence). By an escape a scan may also be terminated before <maximum_length_of_loop> is reached. E. g. see the `until` clause in line (48) indicating an escape on reaching a leftmost word within the `PixMap` array (see Fig. 17: the first execution of `SouthWestScan` at top left corner of the array reaches only a loop length of 1). The condition `@[≤1, ]` says: escape if within current array a data location with an x subscript ≤1 is reached. The empty position behind the comma says: ignore the y subscript).

Hardware-supported Escapes. To avoid overhead for efficiency the `until` clauses are directly supported by MoM hardware features of escape execution [13] (also see Fig. 8). To support the `until @` clauses by *off-limits escape* the address generator provides for each dimension (x, y) two comparators, an upper limit register and a lower limit register

The above program covers the following strategy. The first `while` loop at lines (46) thru (51) iterates the sequence of the 4 scan calls `EastScan` thru `NorthEastScan` fortheupperlefttriangle of theJPEGscan,from `PixMap[1,1]` to `PixMap[8,1]` (see Fig. 17). The second `while` loop at lines (56) - (61) covers the lower right triangle from `PixMap[8,1]` to `PixMap[8,8]`. The `SouthWestScan` between both `while` loops at line (67) from `PixMap[8,1]` to `PixMap[1,8]` connects both triangular scans to obtain the total JPEG pattern.

This section has introduced the essentials of the language MoPL-3, a C extension, by means of three algorithm implementation examples. The main objective of this section has been the illustration of the language elements for data sequencing programs and the illustration of its comprehensibility and the ease of its use.

7. APPLICATION AREAS FOR XPUTERS

Xputers like the MoM-3 and MoM-4 architectures are as universal as computers. A general competition between xputers and computers in all possible application areas would be unrealistic. This section briefly discusses suitable application areas for xputers from different points of view: for which algorithms and problem areas most benefits are to be expected - from which application environments least technology transfer problems will arise - in which application scenarios most cost/performance benefits can be expected.

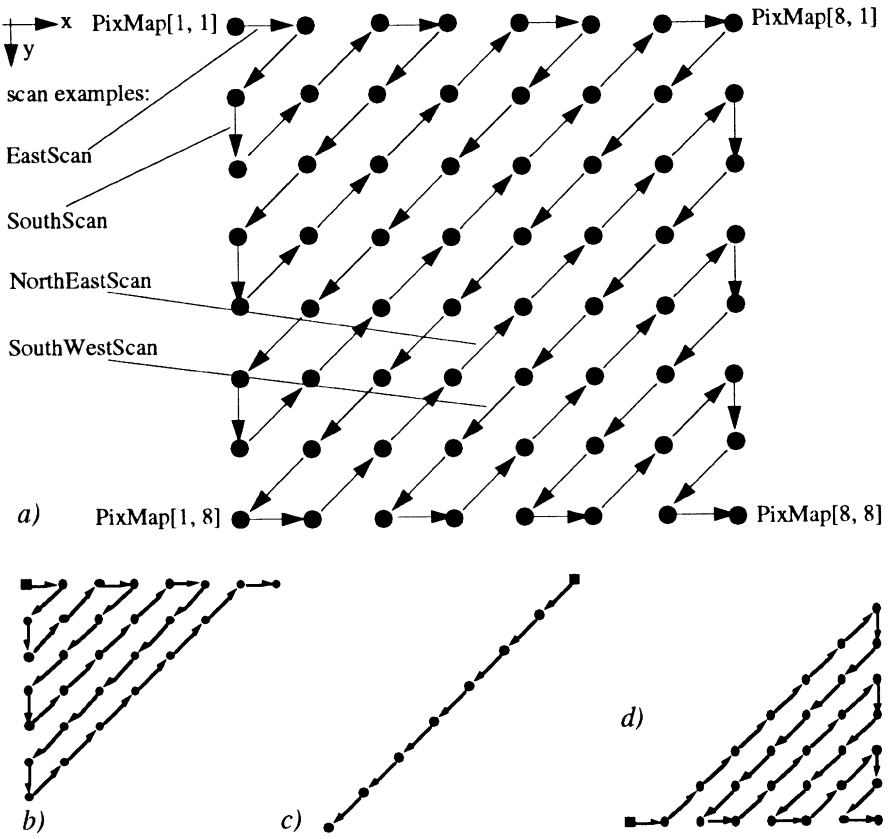


Fig. 17: JPEG zigzag scan pattern scanning an array PixMap [1:8,1:8] (a) and its subpatterns: b) upper left triangle UpLzigzagScan, d) lower right LoRzigzagScan, c) full SouthWestScan.

7.1. Application Environments

Xputers are not competitive to computers in general, since cross compilers, and application software environments are not available commercially. Competitiveness, however, is be expected for particular niches of application markets, such as image processing, digital signal processing, computer graphics, multi media applications, scientific computing, and others, where higher performance is needed at low hardware cost.

```
Array           PixMap [1:8,1:8,15:0]                          (38)
ScanPattern     EastScan       is  1 step  [ 1, 0];            (39)
                SouthScan      is  1 step  [ 0, 1];            (40)
                SouthWestScan  is  7 steps [-1, 1];            (41)
                NorthEastScan  is  7 steps [ 1,-1];            (42)
                                                               (43)
                UpLzigzagScan     is                           (44)
                begin                                          (45)
                     while (@[<8,])                            (46)
                     begin  Eastscan;                          (47)
                            SouthWestScan until @[≤1,];        (48)
                            SouthScan;                         (49)
                            NorthEastScan until @[,≤1];        (50)
                     end)                                      (51)
                end UpLzigzagScan ;                            (52)
                                                               (53)
                LoRzigzagScan     is                           (54)
                begin                                          (55)
                     while (! @[8,8])                          (56)
                     begin  EastScan;                          (57)
                            NorthEastScan until @[8,];         (58)
                            Eastscan;                          (59)
                            SouthWestScan until @[,8];         (60)
                     end                                       (61)
                end LoRzigzagScan ;                            (62)
                                                               (63)
                JPEGzigzagScan    is                           (64)
                begin                                          (65)
                            UpLzigzagScan                      (66)
                            SouthWestScan;                     (67)
                            LoRzigzagScan                      (68)
                end JPEGzigzagScan ;                           (69)
endScanPattern ;          (* end of declaration part*)        (70)
                    :                                          (71)
                    :                                          (72)
begin                     (* statement part*)                 (73)
     moveto PixMap [1,1];                                      (74)
     JPEGzigzagScan ;                                          (75)
end                                                            (76)
```

Fig. 18: MoPL program of the JPEG scan pattern shown in Fig. 17

Xputer use as co-processor. From a technology transfer point o view, and, for utilization of existing utilities, interfaces and application software an good symbiosis would be using the xputer as a universal accelerator co-processor,

hosted by a von Neumann computer, e. g. as an extension board within a workstation. Only those critical algorithms, which exceed the power of the host, are candidates for running on the co-processor, mostly only a few lines of source code.

Rapid turn-around ASIC synthesis. Because the rALU and the use of field-programmable logic the xputer has close relations to ASIC design methods. That's why with xputer parts held in cell libraries a new approach to ASIC design could be created. Debugging would be by orders of magnitude faster than in traditional ASIC design, since execution is used instead of simulation. By retargeting this programmable version could be converted into a hardwired gate array version for fabrication. For a few more details see paragraph on fast turn-around ASIC design in section 2.

New directions in supercomputing. Because of high acceleration factors in algorithms, which are subjects of supercomputing efforts, xputers, their compilers, and their applications are a source of ideas for new directions in supercomputing research - also in compilation techniques because of the paradigm's close relations to data dependency analysis. The xputer execution mechanism supported by scan caches and their address generators is a generalization of vectorization. With xputers the storage schemes for interleaved access memories are derived more easily and can be used for a wider variety of algorithms than with traditional supercomputers. (also see section 2.).

8. CONCLUSIONS

The paper has briefly summarized the new xputer machine paradigm, has demonstrated its basic execution mechanisms, and, has shown its very high efficiency and the reasons for it. The paper has introduced a new high level xputer programming language MoPL-3 being an extension of the language C and has illustrated its comprehensibility and the ease of its use in data-procedural programming for xputers. An earlier version of the language (MoPL-2) has been implemented at Kaiserslautern on VAX station under ULTRIX. For systolizable algorithms a program generator has been implemented as a front end, which generates MoPL programs by using modified versions of projection techniques known from systolic synthesis. It is an essential new aspect of this new computational methodology, that it is the consequence of the impact of field-programmable logic and features from DSP and image processing on basic computational paradigms. We have illustrated, that xputers, their languages and compilers open up several promising new directions in research and development - academic and industrial.

9. REFERENCES

[1] P. Bertin, D. Roncin, J. Vuillemin: Introduction to Programmable Active Memories; Int'l Conf. on Systolic Arrays, Kilarney, Ireland, 1989

[2] D. Causley, J. Z. Young: The Flying Spot Microscope - Use in Particle Analysis, Research 8, p.430-434, 1953.

[3] M. Christ: Texas Instruments TMS 320C25; Signalprozessoren 3; Oldenbourg-Verlag 1988

[4] M. D'Amour, et al.: ASIC Emulation cuts Design Risk; High Performance Systems, Oct. 1989

[5] G. P. Dinneen: Programming Pattern Recognition, Proceedings of the Western Joint Computer Conference, Los Angeles, 1955.

[6] J. A. B. Fortes, K. S. Fu, B.J. Wah, "Systematic Approaches for Algorithmically Specified Systolic Arrays", in Computer Architecture: Concepts and Systems, (ed.: V. Milutinovic), North Holland, 1988.

[7] R. Freeman: User-Programmable Gate Arrays; IEEE Spectrum, Dec.1988.

[8] D. Gajski, D. Padua, D. Kuck, R. H. Kuhn, "A Second Opinion on Data Flow Machines and Languages", Computer, pp. 58-69,Febr. 1982.

[9] F. A. Gerrtisen: Design and Implementation of the Delft Image Processor DIP-1; Ph.D Thesis, Dept. Electrical Engineering, Delft University, 1981.

[10] M. J. E. Golay: Apparatus for counting bi-nucleate lymphocytes in blood, U.S. Patent 3,214,574, 1965.

[11] S. B. Gray: Local Properties of Binary Images in Two Dimensions; IEEE Trans. on Computers, C-20 (5), 1971.

[12] M. D. Graham, P. E. Norgren: The diff3 Analyzer - A Parallel/Serial Golay Image Processor; in: Real-Time Medical Image Processing (Onoe, Preston, Rosenfeld eds.), Plenum Press, New York 1980.

[13] R. W. Hartenstein, A. G. Hirschbiel, K. Lemmert, K. Schmidt, M. Weber: A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware; Int'l Conf. on Information Technology, Tokyo, Japan, Oct. 1990.

[14] R. W. Hartenstein, A. G. Hirschbiel, K. Lemmert, K. Schmidt, M. Weber, "The Machine Paradigm of Xputers and its Application to Digital Signal Processing", Proc. of 1990 International Conference on Parallel Processing, St. Charles, Oct. 1990.

[15] R. Hartenstein, A. Hirschbiel, M. Weber: MoM - Map Oriented Machine; in: Ambler et al.: Hardware Accelerators, Adam Hilger, Bristol 1988.

[16] R. Hartenstein, G. Koch: The universal bus considered harmful; in: (eds.) R. Hartenstein, R. Zaks: Microarchitecture of Computer Systems, North Holland, 1975

[17] R.W. Hartenstein, R. Hauck, A.G. Hirschbiel, W. Nebel, M. Weber: PISA - A CAD package and special hardware for pixel oriented layout analysis, Proc. ICCAD 1984,

[18] J. M. Herron, J. Farley, K. Preston Jr., H. Sellner: A General-Purpose High-Speed Logical Transform Image Processor; IEEE Transactions on Computers, C-31(8), 1982.

[19] A. G. Hirschbiel: A Novel Processor Architecture based on Auto Data Sequencing and Low Level Parallelism; Ph. D. dissertation, Universität Kaiserslautern, 1991

[20] P. A. Kaufmann: Wanted: Tools for Validation, Iteration; Computer Design, Dec.1989

[21] R. A. Kirsch: Experiments in Processing Information with a Digital Computer, Proc. Eastern Joint Computer Conference, Washington, 1957.

[22] B. Kruse, P. E. Danielsson, Gudmundsson: From Picap I to Picap II; in Special Computer Architectures for Pattern Processing (K.S. Fu, T. Ichikawa, eds.), CRC Press, Boca Raton, 1982.

[23] B. Kruse: A Parallel Picture Processing Machine, IEEE Computer C-22(12), 1973.

[24] S.-Y. Kung, VLSI Array Processors; Prentice-Hall, 1988.

[25] K. Lemmert, SYS3 - a Systolic Synthesis System around KARL, Ph. D. dissertation, Kaiserslautern University, 1989.

[26] C. Lengauer, On the Projection Problem in Systolic Design; report, Carnegie-Mellon University, CMU-CS-88-102, Pittsburgh, 1988

[27] D. I. Moldovan, "ADVIS: A Software Package for the Design of Systolic Arrays", IEEE Transactions on Computer Aided Design, pp. 33-40, Jan., 1987.

[28] N. N. (Motorola): DSP 56000 / 56001 Digital Signal Processor User's Manual; Motorola Corp., 1989.

[29] R. Nawrath, J. Serra: Quantitative Image Analysis: Theory and Instrumentation; Micros. Acta 82 (2) p.101-111, 1979.

[30] N. Petkov: Systolische Algorithmen und Arrays, Akademischer Verlag 1989.

[31] N. N. (Plessey): Quickgate (Product Overview); Plessey Semiconductors, Swindon, U.K., May, 1990.

[32] K. Preston Jr.: The CELLSCAN system - A leucocyte pattern analyzer, Proc. Western Joint Comput. Conf., 1961

[33] K. Preston Jr.: Use of the Golay Logic Processor in pattern-recognition studies using hexagonal neighborhood logic; Proc. Symp. Comput. Automata, Polytechnic Press, New York 1971.

[34] K. Preston Jr.: Cellular Architectures for Image Processing; Int'l Conference on Computer Design: VLSI in Comput., New York, IEEE Publ. no. CH1935-6/83, 1983.

[35] K. Preston Jr., M. J.B. Duff: Modern Cellular Automata, Theory and Applications; Plenum Press, New York 1984.

[36] P. Quinton, Y. Robert: Systolic Algorithms & Architectures, Prentice Hall 1989.

[37] H. J. Siegel: Interconnection networks for large-scale parallel processing; McGraw-Hill, New York, 1990

[38] Thomas W. Starnes: MC68000: Philosophie und praktische Realisierung einer 16/32-Bit_Mikroprozessorfamilie; Das 68000-Sonderheft, Franzis-Verlag, München 1985.

[39] S. R. Sternberg: Cytocomputer real-time pattern recognition; Proceedings of the 8th Automatic Imagary Pattern Recognition Symposium, 1978

[40] S. H. Unger: A Computer Oriented Toward Spatial Problems; Proc. IRE 46, 1958.

[41] M. Weber: An Application Development Method for Xputers; Ph. D. dissertation, Kaiserslautern University, 1990.

[42] S. S. Wilson: One Dimensional SIMD Architectures - The AIS-5000; in Multicomputer Vision (Levialdi eds.), Academic Press, 1988

Index