1

MicroD

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

HIGH-LEVEL SYNTHESIS OF DIGITAL CIRCUITS
USING GLOBAL SCHEDULING AND BINDING ALGORITHMS

by

Pierre G. Paulin, M.Sc.A.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of

Doctor of Philosophy

Department of Electronics
Faculty of Engineering
Carleton University
January 1988

© copyright

1988, Pierre Paulin

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-40636-4

The undersigned recommend to the faculty of Graduate Studies
and Research acceptance of the thesis:

"High-Level Synthesis of Digital Circuits
Using Global Scheduling and Binding Algorithms"

submitted by Pierre G. Paulin, in partial fulfillment of the
requirements for the degree of Doctor of Philosophy.

Prof. M.A. Copeland
Chairman, Dept. Electronics

Dr. J. P. Knight
Thesis Co-Supervisor

Dr. D. Agnew
Bell-Northern Research
Thesis Co-Supervisor

Dr. M. C. McFarland
Boston College
External Examiner

Dr. S. Curry
Bell-Northern Research
External Examiner

Carleton University

February, 1988.

To my mother,
for her support.

To my father,
for his example ...

ABSTRACT

The need to rapidly produce designs of digital integrated circuits has motivated the development of high-level synthesis tools that automatically generate a design of a digital system from an abstract specification of its behavior.

This dissertation describes a multi-paradigm implementation of a high-level synthesis tool that features:

- A new global *force-directed* scheduling algorithm that attempts to balance the distribution of operations that make use of the same hardware resources; therefore minimizing functional unit, register and interconnection costs.
- A rule-based expert allocator that performs a global analysis of the control and data flow graph and uses heuristic rules to preselect hardware modules based on a speed constraint.
- A stepwise refinement approach to scheduling and allocation where preliminary allocation information is used to guide and optimize the scheduling process.
- Data path synthesis algorithms that attempt to minimize interconnect at all stages of the synthesis process.

ACKNOWLEDGEMENTS

First and foremost, I would like to give special thanks to my supervisor John Knight, who despite his hectic schedule, always found time to help me with the difficult problems of synthesis — and knew when to trust me with the simple ones.

Thanks are also due to Emil Girczyc whose valuable early advice helped me get started on the right track. Much of the research presented here was strongly influenced by his landmark early work in the areas of synthesis and scheduling.

I also want to thank my co-supervisor from BNR, Dave Agnew, who was responsible for making this industrial-academic cooperative research project possible; devoted patient hours to proof-reading my early papers; helped send me to many fascinating conferences; and generally made my life at BNR easier.

Jenny Midwinter also deserves credit for her excellent suggestions with respect to interconnect allocation, not to mention our numerous insightful 'question periods'.

Finally, I would like to thank my fiancée Jane Carson who, through her example, demonstrated the advantages of a prompt resolution of the thesis writing phase; but also for a great many other things, most of which have nothing to do with this thesis.

TABLE OF CONTENTS

Chapter 1. Introduction	1
1.1 Evolution of VLSI CAD and Automation Tools	1
1.2 Silicon Compilers	2
1.2.1 Register-Transfer Level Synthesis	3
1.2.2 High-Level Synthesis	3
1.3 Motivations for High-Level Synthesis	6
1.4 Aim of dissertation	10
1.5 Thesis Outline	13
 Chapter 2. System Overview	 15
2.1 Synthesis Subtasks	15
2.2 Complete Proposed System	18
2.3 Prototype System Overview	20
2.3.1 Scheduler	20
2.3.2 Allocator	21
2.3.3 Scheduling and allocation by stepwise refinement	21
2.3.4 Data Path Synthesizer	22
 Chapter 3. Related Work	 24
3.1 Implementation Approaches	25
3.2 Synthesis Approaches	26
3.3 Comparison of synthesis approaches	32
3.3.1 Interactions Between Scheduling and Allocation	33
3.3.2 Operation Scheduling	34
3.3.3 Allocation and Binding of Functional Units	35
3.3.4 Register assignment	36
3.3.5 High-level exploration of the design space	36
 Chapter 4. Representations for Synthesis	 38
4.1 Behavioral Representations	38
4.1.1 Data Flow	39
4.1.2 Control Flow	40
4.1.3 Timing Constraints	46
4.2 Structural Representations	47
4.2.1 Functional Units	47
4.2.2 Storage Units	48
4.2.3 Interconnection Units	48
4.2.4 Input/Output Units	49
 Chapter 5. System Implementation	 50
5.1 Programming Paradigms	51
5.2 The Object Paradigm	52
5.2.1 Properties of Behavioral Objects	55
5.2.2 Properties of Structural Objects	57
 Chapter 6. InHAL: Force-Directed Scheduling	 60
6.1 Related Work in Scheduling	61
6.1.1 Independent scheduling/allocation schemes:	62
6.1.2 Interdependent scheduling/allocation schemes:	64
6.1.3 Summary of scheduling approaches	66
6.2 Comparison with Force-Directed Scheduling	66

6.2.1	Local vs global evaluations:	67
6.2.2	Independent vs interdependent scheduling/allocation	68
6.2.3	Topological vs middle-out scheduling:	69
6.2.4	Freedom vs Force:	70
6.3	Basic Force-Directed Scheduling Algorithm	71
6.3.1	Determination of time frames	72
6.3.2	Creation of distribution graphs (DGs)	76
6.3.3	Calculation of 'self' forces	80
6.3.4	Calculation of predecessor and successor forces	85
6.3.5	Lookahead	87
6.3.6	Summary	90
6.4	Complexity Considerations	91
6.4.1	Direct Method	91
6.4.2	Alternate Method	92
6.4.3	Algorithm description for the alternate method	95
6.4.4	Final Complexity Calculations	98
Chapter 7. Refined Scheduling Algorithm		101
7.1	Minimization of Bus Costs	102
7.2	Minimization of Register Costs	103
7.3	Local Timing Constraints	109
7.4	Scheduling with Loops	110
7.5	Incorporation of Structural Unit Area Costs	111
7.6	Multi-class DGs	112
7.7	Chained Operations	113
7.8	Multiple Control-Step Operations	114
7.9	Scheduling Under Fixed Hardware Constraints	115
7.9.1	General hardware constraints	116
7.9.2	Specific hardware constraints	116
7.10	Scheduling Under Mixed Time-Area Constraints	119
Chapter 8. Extensions for Pipeline Scheduling		121
8.1	Related Work	124
8.1.1	Algorithm pipelining	124
8.1.2	Structural pipelining	125
8.1.3	Summary of scheduling approaches	126
8.2	Extension for Algorithm Pipelining	126
8.3	Extension for Structural Pipelining	130
Chapter 9. MidHAL: Allocation of Functional Units		134
9.1	Overview	134
9.2	Rule Invocation	136
9.3	Sample rules and allocations	142
9.4	Summary of allocations	148
Chapter 10. Scheduling and Allocation by Stepwise Refinement		155
10.1	Scheduling/Allocation Phases	156
10.2	Use of Allocation Information for Scheduling	157
10.2.1	Types of functional units allocated:	157
10.2.2	Number of functional units allocated:	157
10.2.3	Functional unit speed:	160
10.2.4	Functional unit and interconnect area costs:	160

10.3	Sample Schedules and Allocations	161
Chapter 11. ExHAL: Data Path Synthesis		
11.1	Operation to FU binding	168
11.2	Creation and Insertion of Storage Operations	175
11.3	Storage to register binding	176
11.4	Data-transfer to interconnect binding	177
11.5	Alignment of FU operands	180
11.6	Register merging	184
11.6.1	Weight calculation	184
11.6.2	Weight-directed clique partitioning	186
11.6.3	Partitioned Register Merging	192
11.6.4	Incorporation of floorplanning information	194
11.7	Multiplexer merging	195
11.8	Control information	198
11.9	Use of Local Storage	198
Chapter 12. Design Examples		
12.1	Differential Equation Example from HAL	204
12.2	MAHA Code Sequence Example	211
12.3	Pipelined FIR Filter from Sehwa	219
12.4	Facet Code Sequence Example	221
12.5	Fifth-order Digital Elliptic Wave Filter	224
Chapter 13. System Limitations and Future Work		
13.1	Limitations of Scheduling and Allocation Approaches	230
13.2	Limitations of Data Path Synthesis Algorithms	233
Conclusion		235
Glossary		238
Bibliography		240
Appendix A. Cost Functions and Structural Unit Costs		A-1
Cost Scheme I		A-2
Cost Scheme II		A-5
Chosen Parameter Values and Structural Unit Costs		A-6

LIST OF ILLUSTRATIONS

Figure 1.	Transformations in High-Level Synthesis . . .	5
Figure 2.	Synthesis subtasks for a simple example . . .	18
Figure 3.	Overview of the behavioral synthesis process. . .	20
Figure 4.	Symbol description for data flow objects. . .	41
Figure 5.	Symbol description for conditional control constructs.	44
Figure 6.	Symbol description for loop constructs. . . .	47
Figure 7.	Symbol description for structural objects. . .	51
Figure 8.	High-level class hierarchy of the HAL system objects.	56
Figure 9.	Class hierarchy of behavioral objects. . . .	57
Figure 10.	Class hierarchy of structural objects. . . .	59
Figure 11.	Data flow graph (DFG) for first example . . .	75
Figure 12.	Simplified data flow graphs with loops not shown	76
Figure 13.	Time frames of operations (initial state). . .	78
Figure 14.	Distribution graphs (initial state).	79
Figure 15.	Distribution evaluation for CDFG with conditional statement.	80
Figure 16.	Time frame modifications for force calculations.	85
Figure 17.	Final time frames and DGs	93
Figure 18.	Storage lifetimes for all schedules	109
Figure 19.	Register requirements for ASAP and forced-directed schedules.	111
Figure 20.	Algorithm pipelining	124
Figure 21.	Structural pipelining	125
Figure 22.	Schedule obtained using regular scheduling algorithm.	129
Figure 23.	Distribution graph evaluation using algorithm pipelining.	130
Figure 24.	Schedule obtained using algorithm pipelining extension.	131
Figure 25.	Schedule and Distributions using non-pipelined multipliers	134
Figure 26.	Schedule and Distributions using pipelined multipliers	135
Figure 27.	Rule 0	139
Figure 28.	Rule 1	140
Figure 29.	Rule 2	140
Figure 30.	Allocation I	142
Figure 31.	Allocation II	142
Figure 32.	Rule 6	146
Figure 33.	Allocation III	146
Figure 34.	Rule 8	148
Figure 35.	Rule 9	149
Figure 36.	Allocation IV	150
Figure 37.	Allocation V	150
Figure 38.	Functional unit allocation for all time constraints	151
Figure 39.	Functional unit allocation for all time con-	

	constraints	152
Figure 40.	Functional unit allocation for all time constraints	153
Figure 41.	Cost curve for differential equation example	154
Figure 42.	Scheduling/Allocation by stepwise refinement	157
Figure 43.	Functional unit allocation for all time constraints	165
Figure 44.	Symbol description for graph objects.	169
Figure 45.	Control-Step Graph (CSG) for differential equation	171
Figure 46.	Binding table for differential equation	172
Figure 47.	Transfer table after c-step 1	174
Figure 48.	Transfer table after c-step 2	175
Figure 49.	Transfer table after c-step 3 (first tentative binding)	175
Figure 50.	Transfer table after c-step 3 (second tentative binding)	176
Figure 51.	Preliminary storage to register binding	180
Figure 52.	Partial Data Path	182
Figure 53.	Input vectors (initial state).	183
Figure 54.	Input vectors after exchanging (R4) & (R8).	184
Figure 55.	Partial Data Path	185
Figure 56.	Structural weights for different types of register merges	188
Figure 57.	Compatibility matrix of all registers	190
Figure 58.	Compatibility matrix (weight threshold = 4)	192
Figure 59.	Compatibility matrix (weight threshold = 3)	193
Figure 60.	Final register assignments	195
Figure 61.	Compatibility matrix of all muxes	198
Figure 62.	Final CSG with bound transfers.	200
Figure 63.	Comparison of Regular and Local Storage techniques	203
Figure 64.	Cost summary for HAL example.	208
Figure 65.	HAL Datapath (using regular FUs).	209
Figure 66.	CSG for differential equation example (using	211
Figure 67.	Splicer and HAL Datapaths (using pipelined multipliers).	213
Figure 68.	Schedule from MAHA system.	215
Figure 69.	MAHA example using force-directed scheduling.	217
Figure 70.	MAHA example using list scheduling.	219
Figure 71.	MAHA example using force-directed scheduling	221
Figure 72.	Summary of results for MAHA example	222
Figure 73.	FIR filter results for SEHWA Example	223
Figure 74.	Comparison of Facet and HAL datapaths	226
Figure 75.	Comparison of Facet and HAL area costs	227
Figure 76.	Schedule for wave filter example.	229
Figure 77.	Data path for wave filter example.	230
Figure 78.	Cost summary for Kung, Whitehouse and Kailath example.	232
Figure 79.	Interconnection Schemes	A-3

CHAPTER 1. INTRODUCTION

1.1 EVOLUTION OF VLSI CAD AND AUTOMATION TOOLS

The ability of IC foundries to produce increasingly complex chips has driven the development of sophisticated CAD tools to allow a chip designer to exploit this enhanced manufacturing capability. This new generation of CAD tools is leading the IC designer's involvement away from device-level considerations and towards architectural ones.

We have witnessed this shift through the increased utilization of automated tools for placement and routing; the definition of extensive standard-cell libraries; the availability of highly specialized module builders; and the addition of floor-planning capabilities to the spectrum of existing CAD tools.

These tools have progressively raised the level of abstraction of the input description to provide higher design leverage. They also have encouraged the integration of system designers into IC design teams.

One of the recent and most important tools that is contributing to this continuing trend is the so-called *silicon compiler* [1].

1.2 SILICON COMPILERS

In the most general sense, a silicon compiler can be described as a system which takes a high-level abstract description of a system, and automatically generates the final physical realization.

Input Description Language: The input description language of a silicon compiler can be divided into three major categories: structural, behavioral and algorithmic [2] as depicted in Figure 1 on page 5. A *structural* input language specifies a description of subcircuits and their interconnections. A well-known example of this type of description is the standard net-list. A more abstract structural input description is the *register-transfer level (RTL)* description which specifies the interconnections of functional units (such as adders and ALUs), storage units (e.g. registers, memory), and interconnection units (e.g. multiplexers, buses). A sample RTL architecture is given in the lower half of Figure 2 on page 17.

Conversely, an *algorithmic* input language describes a circuit by specifying its input/output mapping, without any explicit structure specification. A *behavioral* description is similar to an algorithmic one, except that some basic information about the architecture of the system is defined by the user. Algorithmic and Behavioral synthesis systems are

often referred using the generic term *high-level synthesis* systems.

1.2.1 Register-Transfer Level Synthesis

In the VLSI CAD industry, the term silicon compiler has generally been used to describe systems that use a *structural* input description language. In most cases, commercial silicon compilers are akin to specialized high-level module builders. When coupled with automated finite state machine (FSM) synthesis tools, the systems are able to perform synthesis from a register transfer level (RTL) description.

1.2.2 High-Level Synthesis

As RTL synthesis tools gain a stable foothold in industry, the automatic synthesis of a digital system from an *algorithmic* or *behavioral* description is the next step on the ladder of design automation hierarchy.

High-level synthesis systems take a functional description of a digital system and automatically generate a register-transfer level structural realization. This is illustrated using the tripartite Y-diagram [3] of Figure 1.

This RTL architecture is typically composed of a data path and a control path. The former performs data transfer,

data storage as well as arithmetic and logic operations, while the latter generates control signals to activate the data path units. These units can then be generated using technology specific module builders.

On the other hand, the control path is generally implemented with automatic finite state machine (FSM) generators. These FSMs are often microprogrammed or PLA based and usually involve some form of compaction or logic optimization step.

Automatic placement and routing systems are used to generate the final physical realization. These transformations are illustrated in the Y-diagram of Figure 1.

Y-Diagram

Figure 1. Transformations in High-Level Synthesis: The Y shows the three methods of describing a digital circuit. The paths show the necessary transformations to translate a high-level description into a manufacturable circuit.

In a more abstract sense, high-level synthesis can be viewed as a *many-to-fewer binding of functional operations* (arithmetic, logic, storage, transfer and I/O) to *structural units* (alus, logic gates, registers, buses and ports) while respecting time, area and power constraints.

A purely *algorithmic* input description contains no structural information. Only functionality and in some cases, timing constraints, are given. A *behavioral* de-

scription is similar to an algorithmic one except that it typically includes some specifications of the target architecture; in this case, registers, ALUs and buses may be pre-specified in the description. Furthermore, some or all of these may be bound to specific operations. This is often the case for general purpose IC descriptions such as micro-processors, where the ALUs and main registers are pre-specified.

In both algorithmic and behavioral synthesis, the automated design process consists of the following four steps:

1. **Translation:** translating input description to a control and data flow graph (CDFG) representation. This graphical representation defines the explicit data and control dependencies between the operations derived from the algorithmic (or behavioral) description.
2. **Scheduling:** assigning operations to specific control-steps (clock cycles).
3. **Allocation:** selecting a set of structural units (e.g. ALUs, registers, buses) to realize the operations.
4. **Binding:** assigning the operations to specific structural units which can then be interconnected.

A simple example illustrating these steps is given in Figure 2 on page 17 of "Chapter 2. System Overview".

1.3 MOTIVATIONS FOR HIGH-LEVEL SYNTHESIS

The ultimate promise of high-level synthesis is the possibility of bridging the software and hardware domains by

producing high quality chip designs from input descriptions akin to programming languages. A side-effect of this bridge is to allow the incorporation of many of the tools of software engineering into digital circuit design, and vice-versa. Furthermore, the decision of whether to implement a specific section of a system in hardware or software can be deferred since both may use the same description language. Simulation at the behavioral level should enhance system-level integration and ease the software/hardware tradeoff process.

In the interim, the specialized synthesis tools that are being developed now will give designers a fast turnaround automated design which will allow them to explore different architectures before settling on the final one. The potential area savings and/or speed improvements arising from architecture and algorithm exploration are often of an order of magnitude, while typically the savings resulting from optimizations at the logic and/or transistor level are much less important.

In conventional structural design, the designer has a particular architecture in mind and tries to fit an implicit behavior onto it. In doing so, he may neglect some of the inherent parallelism that can be extracted from that behavior. This is particularly true when complex pipeline schemes are attempted.

We will present here a few specific motivations and utilizations of high-level synthesis tools and algorithms.

RTL Optimization: As a logical extension of RTL-based design synthesis, high-level synthesis techniques can be used for RTL optimization. This will become particularly important as less experienced designers become involved in IC design, and are not aware of the implications of their RTL description on the resulting architecture.

This optimization can be achieved by extracting the inherent functionality of an RTL description, representing this function as a control and data flow graph (CDFG), and performing some of the transformations used in high-level synthesis. For example, the register assignments defined in the RTL description may be unbound and reassigned more efficiently using data flow information and register merging techniques.

Scheduling: A second area where synthesis may prove important is the scheduling of the events (operations) occurring in the datapath, to specific control-steps (clock cycles). Effective scheduling is crucial as it affects four essential aspects of the subsequent synthesized design:

- The number and type of functional units allocated (such as adders, ALUs, etc.)
- The timing constraints on these functional units
- The storage requirements (registers, memory)
- The data transfer requirements (buses, muxes, leads)

Ultimately, this affects:

- The area of the design given a speed constraint
- The speed of the design given an area cost constraint

In conventional design, this is perhaps the most difficult aspect to consider. This follows directly from the fact that the input is at the architectural level; where structural parallelism is explicit but temporal parallelism is usually implicit or worse, an afterthought. The problem is compounded by pipelining, loop winding/unwinding and other more subtle methods of achieving temporal parallelism.

Register and Bus allocation: Even if the user has generated a good schedule and has performed an effective selection of functional units, there still remain two non-trivial tasks to be realized:

- The creation and binding of registers to storage operations (variables)
- The creation of buses, muxes and leads to realize the data transfers between registers and functional units

The difficulty arises from the aforementioned latitude in the operation to hardware binding process that is inherent to high-level behavioral descriptions. Variables are not necessarily bound to specific registers or memory, and data transfers are not assigned to particular buses, muxes or direct connections. This results in a combinatorial explosion of possible bindings.

Most designers will create a set of dedicated buses and serialize operations onto these. Complex bus merging schemes are rarely considered as the bus utilization times are difficult to extract from purely structural architecture descriptions.

Typically, registers are assigned without much consideration of variable lifetimes, i.e. the set of control-steps for which a variable must be stored. This information can be used to reduce the total number of registers by assigning two or more variables to a single register when their lifetimes are non-overlapping.

These are areas where the extended search afforded by computers may be beneficial, as has already been proved in placement and routing — two problems that present many similarities with the register and bus allocation tasks.

1.4 AIM OF DISSERTATION

In this thesis, we will describe a novel approach to the high-level synthesis problem and present a prototype multi-paradigm software implementation; the *HAL*^A system. Early

^A *HAL* stands for "Heuristic and Algorithmic Logic", or for those who have seen the program code, "High-level Assembly Language"...

work on this system was presented in [4], [5], [6], [7], [8], [9], [10], and [11].

We will show how the use of multiple programming paradigms allows to partition the overall design task into separate subtasks better suited to different paradigms. These subtasks will be realized using an object-based paradigm, a knowledge-based paradigm, a functional paradigm, or combinations of any of the above.

The synthesis approach presented features global techniques for operation scheduling; a knowledge-based approach to functional unit allocation; and an interconnect-driven algorithm for register and bus allocation/binding.

However, the main goal of this work is to show the importance of operation scheduling with respect to the overall synthesis process. We will present a new scheduling algorithm that is part of an interdependent scheduling and allocation scheme. In this scheme, a hardware allocation estimate is used to guide and optimize the scheduling subtask. The allocation information includes the number, type, speed and cost of functional units as well as the associated bus, multiplexer and interconnect costs.

The global *force-directed* scheduling algorithm at the heart of this scheme has $O(cn^2)$ complexity (n is the number

of operations in the description and (c is the time constraint, expressed in control-steps) and represents a fundamental departure from existing scheduling techniques. It also supports what we believe is the most comprehensive set of constraint types and scheduling modes presented to date. These include:

- Multi-cycle and chained² operations
- Mutually exclusive operations
- Scheduling under fixed global time constraints with
 - minimization of functional unit costs
 - minimization of register costs
 - minimization of interconnect requirements
- Scheduling with local time constraints (on operation pairs)
- Scheduling under fixed hardware cost constraints
- Scheduling pipelined algorithmic descriptions
- Scheduling operations that make use of pipelined functional units

The second main goal is to show the importance of considering interconnect at all levels of the synthesis process. In particular, we will present new techniques for register and bus allocation that rely heavily on interconnect-based indicators.

² Chained operations are sequential combinatorial operations which are fast enough to allow two (or more) to be scheduled in the same control-step.

1.5 THESIS OUTLINE

In "Chapter 2. System Overview", we will describe the main synthesis subtasks and a proposed system to realize them. A brief overview of the prototype HAL system is also given.

This will be followed by a review of existing synthesis systems in "Chapter 3. Related Work". This short review will serve as a basis for the comparison of the HAL system with the latter systems.

"Chapter 4. Representations for Synthesis" deals with program and data structures for the representation of synthesis knowledge and design objects. Their implementation using artificial intelligence (AI) development tools is presented in "Chapter 5. System Implementation".

This lays the groundwork for the next three chapters which describe the *force-directed* scheduling algorithm that is the main emphasis of this thesis. "Chapter 6. InHAL: Force-Directed Scheduling" describes the scheduling task, presents existing scheduling algorithms and compares them with the force-directed algorithm. This is followed by a description of the basic algorithm, using an example that will be carried through the entire thesis.

"Chapter 7. Refined Scheduling Algorithm" presents the refinements necessary to make the scheduling algorithm usable for the wide range of constraints that a good synthesis tool should address. Algorithm extensions that allow the scheduling of pipelined datapaths are presented in "Chapter 8. Extensions for Pipeline Scheduling".

In "Chapter 9. MidHAL: Allocation of Functional Units", we present a knowledge-based approach to the functional unit (FU) allocation task. The following chapter presents a stepwise refinement approach to both of the scheduling and allocation tasks.

"Chapter 11. ExHAL: Data Path Synthesis" describes the methods used for binding the allocated datapath modules to specific operations. It also addresses the binding of storage to registers and data transfers to interconnect units.

In "Chapter 12. Design Examples", we present experimental results based on examples taken from current literature. When applicable, these results are compared with the ones obtained in the source reference. Finally, we present the system's limitations, propose future work, and conclude on the impact of the system presented.

CHAPTER 2. SYSTEM OVERVIEW

2.1 SYNTHESIS SUBTASKS

The task of automatically designing a digital system from a behavioral description is usually divided into many subtasks [12]. The first is the *definition of the circuit function* in a high-level hardware description language (HDL). Figure 2a) depicts a simple behavioral description that will be used to illustrate the synthesis process. This step is usually followed by a *translation to a graph-based representation* derived from the control and data flow (control-data flow graph or CDFG) as in Figure 2b).

The next step (Figure 2c) is the *operation scheduling* where operations are assigned a propagation delay value and partitioned into specific control steps (c-steps). This step can be preceded, followed or performed concurrently with the *allocation of hardware modules*. The latter consists of allocating functional units (FUs) to execute the operations (nodes) of the CDFG; registers to store the values (edges); and buses and muxes to realize the data transfers between the registers and functional units. Although scheduling and allocation can be performed separately, many systems attempt to link them in order to improve the overall process [4], [25], [27].

The next step is the *data path synthesis* where the allocated functional units, registers and the data transfer components are bound to specific operations. These structural units are then interconnected by using the data transfer information derived from the CDFG. The synthesis process usually includes an optimization phase that improves the register and bus bindings.

The final realization of the data path is done either by assigning the high-level modules to "pre-designed" templates that represent their gate-level structure or by synthesizing them directly with module generators.

The schedule and the list of data path operations are used to create a state graph which specifies all required control signals in each control-step. This state graph can be used in turn to generate a finite state machine (FSM), which often is micro-programmed or PLA-based. A possible RTL architecture for our simple example is given in Figure 2d).

$$F = E * (A+B)$$

$$G = (A+B) * (C+D)$$

a) HDL Description

b) Data Flow Graph (DFG)

c) Scheduled DFG

d) RTL Architecture

Figure 2. Synthesis subtasks for a simple example

We can therefore summarize the synthesis task as follows:

1. Translation of HDL description to a CDFG
2. Operation scheduling
3. Allocation of datapath structural units
4. Data path synthesis and optimization
5. Generation of datapath structural units
6. Creation of state graph (control operations)
7. Control path synthesis

2.2 COMPLETE PROPOSED SYSTEM

According to [2], an ideal silicon compiler environment would combine both the structural and functional approaches to synthesis.

The system proposed here attempts to do so by allowing the user to partition the overall structure into high-level blocks and interfaces. An algorithmic or behavioral description, subjected to input/output and timing constraints, can then be specified for each block. The system uses the constraints and the algorithmic description of the internal function of each block to automatically generate a data path and the associated control information. This information can be used to generate a finite-state machine (typically microprogrammed or PLA-based) for the control functions.

Extended design-space exploration is made possible by varying the speed specification. The synthesized data path

may span the design space between a low cost full-serial realization and a high cost maximally parallel realization.

The proposed approach to data path synthesis is summarized in the flow chart of Figure 3.

Figure 3. Overview of the behavioral synthesis process.

2.3 PROTOTYPE SYSTEM OVERVIEW

The HAL system uses an AI multi-paradigm implementation [6] that is composed of three main modules. These correspond to the subtasks in the boxed area of Figure 3.

2.3.1 Scheduler

The InHAL module uses a novel force-directed scheduling algorithm that attempts to balance the distribution of operations that make use of the same hardware resources.

In this scheduling approach, the selection of the operation to be scheduled and of the control-step to which it will be assigned, is based on an evaluation of the move which causes the most balanced distribution of operations in each control-step. This algorithm is an iterative one in which one operation is scheduled at each iteration.

The intent is to reduce the number of functional units, registers and buses required by balancing the concurrency of the operations assigned to them, but without lengthening the total execution time. Concurrency balancing helps to achieve high utilization of structural units, which in turn minimizes the number of units required.

2.3.2 Allocator

The MidHAL module performs the allocation of functional units (FUs). This module accepts as input the scheduled CDFG, a global time constraint and a library of available processors and their associated speed and cost. It then uses heuristic rules to select a set of functional units (FUs) to execute the operations described in the CDFG, given the time constraints on these operations.

The selection process uses a rule-based expert system. The knowledge base consists of a set of heuristic FU selection rules. The rule set is invoked in an iterative fashion. Each invocation allocates a specific FU, based on the current state of the *operation trace*; a simplified representation of the CDFG that does not include precedence constraints, i.e. constraints defined by control and data dependencies. Once a component is allocated to a set of operations, the operation trace is reduced accordingly, and the rule set is re-invoked.

2.3.3 Scheduling and allocation by stepwise refinement

The scheduling algorithm also allows specific allocation information, such as the area and speed of a specific hardware module, to be fed back to optimize the scheduling process.

This is achieved by using a stepwise refinement approach (depicted in Figure 42 on page 155) that is divided into four main phases:

- Phase 1 Default Allocation: Allocation of default single-function FUs to perform each type of operation. The fastest FUs available in the technology dependent library are temporarily assumed.
- Phase 2 Preliminary Schedule: Balancing of the distribution of similar operations. This is done while taking into account the cost and speed of the FUs assigned to them in the first phase.
- Phase 3 Refined Allocation: Allocation of single and multi-function FUs. This allocation may actually substitute slower FUs or combine different operations into groups that can be performed by an ALU.
- Phase 4 Final Schedule: In this phase the force-directed scheduling algorithm is invoked once again. The goal now is to balance the distribution of operations using similar resources, since those resources have now been allocated more precisely.

2.3.4 Data Path Synthesizer

This module (named ExHAL) binds the operations to specific functional units and creates registers and buses to store and transfer data between FUs. It also performs data path optimization by combining registers, merging muxes into buses and aligning FU operands.

These tasks are realized by applying the following binding and transformation steps:

Operation to FU binding: All regular arithmetic and logic operations are bound to specific functional units. This is done using a top-to-bottom greedy algo-

rithm, where previous bindings are used to guide the ones in the current c-step.

Creation and insertion of storage operations: Each data transfer that crosses a c-step boundary implies that the value must be latched and stored. *Storage operations* are created to represent this type of operation.

Storage to register binding: This preliminary binding step uses the CDFG and operation-FU bindings to select a small subset of the disjoint storage operations. Each group formed from this subset is bound to a single register.

Data-transfer to bus binding: Here a straightforward data path synthesis is performed by creating temporary muxes and connecting them to the input of every register and FU. These muxes are used to form a path to each of their input source objects.

FU operand alignment: The operands of reversible operations (e.g. additions, multiplications) are selectively interchanged to minimize interconnect costs. This step is repeated after each of the register and bus merging steps described next.

Register merging: Here *structural locality* information derived from the initial data path is used to guide the merging of registers. The intent is to perform merging while adding minimum interconnect. A novel modification of a *clique partitioning* algorithm [46] is used to achieve this goal.

Mux merging: The initial data path is optimized by combining muxes into buses when they have disjoint usage times. Only mux pairs with one or more common inputs/outputs will be considered for merging.

Elimination of unnecessary storage: When the input of a FU is stable for more than one c-step, the output needn't be stored in the next c-step. For this reason, some storage operations are redundant. They are selectively removed whenever this results in a reduction of interconnect or register costs.

These are storage operations with non-overlapping usage times.

CHAPTER 3. RELATED WORK

Silicon compilation has been with us for nearly a decade. Johanssen [1] coined the term back in 1979 with his "Bristle Blocks" system. Space limitations will not allow us to review all of the synthesis systems that have appeared during this time. We have selected a subset of the most well-known ones that are also closely related to the system presented here.

- GTE Laboratory's Silc silicon compiler [13]
- The CMU Facet/Emerald system [14] from CMU (Carnegie-Mellon University)
- The EMUCS system, also from CMU [15], [16]
- University of Illinois' Chippe system [19], [20], [21].
- AT&T Bell Lab's BUD-DAA system [22], [23]
- The Elf system [25], developed at Carleton University (the main author is now with the University of Alberta)
- The MAHA/REAL systems [26], [27], [28] developed at the University of Southern California

We will briefly present the system implementation approaches used in these synthesis tools. This will be followed by a more detailed description of the synthesis approaches, and a comparison with the one used in the HAL system.

3.1 IMPLEMENTATION APPROACHES

As there are just about as many different implementation approaches as there are systems, we shall only attempt a loose classification into three large subclasses:

1. Algorithmic: MAHA, Elf, EMUCS, Facet/Emerald, Silc.
2. Mixed algorithmic and rule-based: Chippe, DAA.
3. Multi-paradigm: HAL [4].

Of the systems presented, HAL is the only one that makes use of a multi-paradigm approach that embodies rule-based, algorithmic ~~and~~ object-based programming paradigms. This decision stems from the fact that different tasks are best suited to different programming styles. In turn, each style has an associated computing efficiency that must be traded off against the ease of representation. High-level decisions such as the initial processor allocation require knowledge expressed in the form of heuristics. A rule-based production system is ideally suited for this type of knowledge representation.

On the other hand, tasks such as the operation scheduling or register binding are far better suited to purely algorithmic approaches, both for ease of representation and computing efficiency. Functional programming paradigms such as Lisp [34] could be used effectively for this task.

Finally, the overall representation and classification of the graphs, operations and hardware modules (e.g. ALU's, multiplexers, registers, etc.) would benefit from a rich data representation. An object-based programming paradigm seems ideally suited to this requirement.

3.2 SYNTHESIS APPROACHES

In this section we will review the data path synthesis approaches used by the systems listed above. We will focus nearly entirely on the allocation and binding tasks, as a more detailed review of scheduling techniques will be given in "Chapter 6. InHAL: Force-Directed Scheduling".

Silc: The input to the Silc silicon compiler describes functional blocks operating in parallel, each consisting of boolean and integer operations controlled by a finite state machine (FSM). Each functional block contains data path and control information which are handled separately. There is a one-to-one correspondance of storage elements in the functional specification and on the chip. There is also a direct correspondence between states implied by the specification and the state representation in the FSM sequencers. In other words, the operation scheduling and module allocation are implicit in the HDL description. These

limitations put this system at a level of abstraction just above RTL synthesis systems.

Facet: Tseng and Siewiorek's Facet system ~~starts with a~~ simple *as soon as possible* (ASAP) scheduling technique. It then uses clique partitioning [46] to group and bind operations to functional units, storage operations to registers, and data transfers to multiplexers. The data path is then directly generated from these bindings without subsequent optimization.

The main contribution of the Facet system is the development of a heuristic to simplify the NP-complete clique partitioning problem. A number of notions, such as *divide and conquer* and graph neighborhood properties are used to obtain a near-minimum number of cliques.

EMUCS: The EMUCS system [15], [16] is based on a greedy heuristic that was proposed by McFarland [17] at CMU. It makes use of a simplified list scheduling algorithm developed by Hitchcock [18]. The data path synthesis is performed using an incremental approach that builds iteratively on a partial design. Each iteration of EMUCS chooses a binding of an operation to a functional unit or a value to a register. When a binding is selected, it is added to the partial design along with the interconnection bindings associated with its input and outputs.

Bindings are selected by considering all possible bindings of operations to functional units and values to registers. A heuristic cost function, based on FU usage and interconnection costs, is used to evaluate bindings.

One important contribution of the EMUCS algorithm is the realization that each binding has consequences for the cost of the design that extend beyond the value of the cost function for the binding currently considered. Because a binding can block other bindings by making them infeasible, it can force other more expensive bindings to be selected later. So instead of choosing the minimum cost, EMUCS uses an approach that chooses the operation or value for binding that would suffer the most if its minimum cost binding were blocked. This is accomplished by ranking the binding costs for each operation and value and computing the maximum difference between the minimum and next-to-minimum cost. The operation or value with the largest difference is chosen for binding, avoiding this gain in cost.

Chippe: The Chippe system is one of the more recent high-level synthesis systems published. It uses a knowledge-based approach where a *design critic* (expert system) guides the design process by selecting strategies to improve the design. There are two main modules, *Slicer* and *Splicer*, that are responsible for generating the final RTL design, under the supervision of this critic.

The Slicer module is responsible for the scheduling task. It uses a list scheduling algorithm that will be described later in section "Related Work in Scheduling" of "Chapter 6. INHAL: Force-Directed Scheduling".

The Splicer module transforms the scheduled CDFG into a RTL circuit. It is similar to the EMUCS system in that it proceeds iteratively and builds on a partial datapath, starting from the first control-step downward. The assignments are made in four binding stages:

1. **Register to input bus:** At the beginning of each control-step, registers are placed in a sorted list and assigned to buses. If any of the registers have already been connected to buses during a previous control-step, then those register assignments will be tried first.
2. **Input bus to FU:** This is the most complex of the four steps, as it performs the actual operation to FU binding as well as a portion of the interconnect binding. A greedy strategy is used, where existing connections to FUs are reused whenever possible.
3. **FU to output bus:** This step is similar to the first and a similar algorithm is used.
4. **Output bus to register:** It is during this step that the register assignments are made. Variable lifetime information is stored in a table associated with that register, and is used in successive output bus to register assignments.

BUD-DAA: The AT&T Bell Labs DAA system is invoked after a preliminary planning stage is performed by the recent BUD system [22]. In this planning stage, the design is partitioned, operations are scheduled and an initial module allocation is performed. One of the major strengths of this planning stage is that it is one of the only systems to in-

corporate layout-based considerations. The partitioning is guided by a set of preliminary floor plan evaluations, one for each possible partition.

The DAA system then binds basic storage elements to hardware modules and ports. Synthesis then proceeds by assigning minimum delay information to develop a parallel design. It maps the unbound storage elements to register modules, and each data flow operation is bound to the pre-selected hardware modules. The system then optimizes the realization during the expert analysis phase that follows. This analysis is based on a speed-area cost function. Design expertise, stored as a set of *production rules*, is used to reduce the search space.

Elf: The Elf system uses a constructive iterative approach that differs from EMUCS in that it performs scheduling concurrently with allocation and binding.

Elf proceeds by assigning an *urgency* weight to each operation of the control and data flow graph (CDFG). Urgency is a measure of how much an operation can be delayed before it will cause a global timing constraint to be violated.

Like EMUCS and Chippe, it starts from the top of the CDFG and performs binding of FUs, registers and interconnect concurrently. However, since the schedule is not estab-

lished, it also has the option of delaying operations to minimize costs.

The 'knowledge' that the allocator uses to perform bindings is represented as graph grammar production rules. The allocator enters a loop which is performed for each time slot. A greedy algorithm with pairwise interchange is used to select which rule to apply for the realization of each operation.

MAHA-REAL: This system performs synthesis using three main packages whose functions are described as follows:

1. CSSP: Clocking scheme determination
2. MAHA: Scheduling and FU allocation
3. REAL: Register binding

The system first invokes the *Clocking Scheme Synthesis Package* (CSSP) written by Park [26]. Here the critical path is determined and divided optimally into n steps, one per clock cycle. The clock cycle is also automatically determined here.

The MAHA system then allocates functional units (FU) for the critical path in a first-come first-served fashion. The notion of freedom is used to guide the scheduling of the off-critical-path nodes. The node with the smallest freedom

is chosen for allocation. The FUs previously allocated are sequentially examined and the first control-step where FU sharing can occur is bound to the operation. If no resource sharing is possible, the earliest control-step is arbitrarily assigned to the operation.

If the hardware costs are exceeded at some point during execution, then the critical path is repartitioned into more steps. The allocation is restarted using the relaxed timing constraint.

The register binding task is performed by the REAL system. This system uses the well-known *left-edge* algorithm. It has been used successfully in channel routing to perform optimal track assignment.

Here, the variable lifetimes are substituted for the nets and registers for the tracks. Merging nets into the minimum number of tracks is the dual of merging variable lifetimes into the minimum number of registers.

3.3 COMPARISON OF SYNTHESIS APPROACHES

The system presented here is similar to these in some respects but also embodies many new concepts in the synthesis process as well as in the implementation approach. Further-

more, while many of these systems address the general purpose IC (GPIC) design task, with a strong emphasis on computer designs, the HAL system deals with application specific IC (ASIC) designs intended primarily for DSP and telecommunications applications, where explicit speed constraints are often specified. In the next five subsections, we outline the most important differences between the HAL system and the ones presented.

3.3.1 Interactions Between Scheduling and Allocation

In the Silc, Facet/Emerald, EMUCS and Chippe systems the scheduling is performed more or less independently of the functional unit (FU) allocation. The Elf and MAHA systems do both tasks concurrently.

The HAL and BUD systems, on the other hand, perform both tasks separately but not independently. In the HAL system, the operation scheduling and FU allocation are determined by stepwise refinement as depicted in Figure 42 on page 155 of "Chapter 10. Scheduling and Allocation by Stepwise Refinement".

The HAL system does a default allocation and uses that information to establish a preliminary schedule. The allocation is repeated using the schedule to perform a much more detailed analysis and an improved selection of FUs based on

operation concurrency. The scheduler is then re-invoked and the final schedule is established by optimizing the use of the preselected FUs.

3.3.2 Operation Scheduling

A second fundamental difference relates to operation scheduling (or control step partitioning). We will only briefly describe here two facets of the *force-directed* scheduling algorithm used in the HAL system. A complete analysis will be given in section "Comparison with Force-Directed Scheduling" of "Chapter 6. InHAL: Force-Directed Scheduling".

Local vs. global evaluations: This is, in our view, the most important novel contribution of the HAL system to the scheduling/allocation task. Virtually all the systems discussed rely on a local evaluation of the effect of a control-step assignment on the overall cost of the resulting datapath.

The HAL system considers the related effects of a control-step assignment on all the predecessor and successor operations. Moreover, the assignment's impact on the overall concurrency of operations is also evaluated.

Top-down vs middle-out scheduling: Another difference relates to the order of selection of operations during the

scheduling process. In all the systems mentioned above (with the exception of MAHA), the operations are scheduled iteratively by using their topological¹ ordering (depth-first order in BUD-DAA).² In the HAL system, the ordering is based on the lowest force. This value is independent of the precedence ordering and may be considered a 'middle-out' approach.

3.3.3 Allocation and Binding of Functional Units

The third difference relates to the allocation and binding of functional units. The iterative nature of systems such as EMUCS, Chippe and Elf allow little backtracking. If a misguided allocation was made early in the design process, then that allocation is nearly irreversible. This is due to the inherent short-sightedness of the systems.

The HAL system attempts to resolve this problem by doing the FU allocation in a global expert analysis phase. A temporary binding of operations to specific FUs is also performed to estimate multiplexer, bus and interconnect costs.

¹ Topological ordering is based on the predecessor-successor relations between operations, as dictated by control and data dependencies. Ordering starts with the operations without predecessors at the top of the CDFG and works its way down.

² Depth-first ordering is simply the topological ordering in reverse, i.e. starting from the bottom of the graph and working upwards.

The final binding is done later, when the data path synthesis module is invoked.

3.3.4 Register assignment

The EMUCS, Chippe, BUD-DAA and Elf systems perform register assignments in an iterative fashion, concurrently with the other data path synthesis subtasks.

The Facet, REAL and HAL systems perform it as a separate semi-independent task. The REAL system uses the left-edge algorithm to obtain the minimum number of registers. It ignores the associated effects on interconnect costs though.

Facet and HAL both use clique partitioning approaches. Facet makes use of heuristics based on the clique graph structure to prune the number of possible cliques. The indirect effect on interconnect costs are not considered. The HAL system makes use of interconnect information derived from the data path to reduce the clique graph. In this way, the interactions of register and interconnect are taken into account.

3.3.5 High-level exploration of the design space

The HAL system takes an explicit speed constraint and applies it in the beginning of the synthesis process. High-

level decisions such as operation scheduling and FU allocation are directly affected by this constraint. This allows the designer to explore a full spectrum of implementations, from maximally parallel ones to fully serial ones, simply by varying the constraint.

Furthermore, an extension of the force-directed scheduling algorithm allows to take into consideration a user-defined area constraint. Using this extension, the algorithm attempts to minimize execution time while respecting the area constraint.

The EMUCS and Facet systems from CMU place very strong emphasis on the synthesis of maximally parallel data paths. This is in keeping with the GPIC or computer design orientation their creators have given them.

On the other hand, the Chippe, BUD-DAA, Elf and MAHA systems also allow extended exploration of the design space. In the Chippe and MAHA systems though, this is accomplished primarily through the use of area constraints, not speed constraints. In the BUD-DAA system, the user can define area or speed constraints.

CHAPTER 4. REPRESENTATIONS FOR SYNTHESIS

The importance of effective data representations for synthesis can never be stressed too highly. In many cases, the choice of a proper representation can influence the determination of a good solution to a particular synthesis subtask.

In high-level synthesis, it is useful to distinguish two main classes of representations: behavioral and structural. These classes correspond loosely to the initial and final states of the input description. They can be subdivided into the following subclasses:

- Behavioral
 - data flow
 - control flow
 - timing
- Structural
 - functional units
 - storage units
 - interconnection units.
 - I/O units

We describe each of these subclasses in the next sections.

4.1 BEHAVIORAL REPRESENTATIONS

The first main class of representations corresponds to the functional axis of the Y-diagram of Figure 1 on page 5.

These are abstract operations derived from the algorithmic or behavioral description.

In this thesis, we use a control and data flow graph representation that is similar in many respects to CMU's *value trace* (VT) developed by McFarland [29] and refined by Nestor [30] to include local minimum/maximum timing constraints. However, we will see that a few fundamental differences exist.

4.1.1 Data Flow

The symbol used for data flow operations is given in Figure 4. The upper half identifies the operation type and index, while the lower half shows the bound functional unit (FU). Each operation and FU have an associated bit-width. The program ensures that these are compatible, i.e. that the bit-width of the FU is greater or equal to the largest bit-width of all the operations it performs.

The small circles at the c-step boundaries represent storage operations, the small squares represent a data transfer. Both are also assigned bit-widths. These operations will be discussed in further detail later.

4.1.2 Control Flow

There are two main types of control flow constructs implemented in the HAL system:

1. The case construct
2. The loop construct

Case Construct:

A two-way case construct (corresponding to an if-then-else statement) is shown in the simple example of Figure 5. The construct is actually formed of two main objects: a *fork* and a *join*.

The fork exerts a control dependency on every operation that is defined within the boundaries of the case (or if-then-else) statement in the user's behavioral description. Conversely, each operation has an internal field that defines which case statement (if any) and which branch of that case statement it is part of.

This type of dependency is implemented with *control edges* as indicated in Figure 5. Here, two sets of control edges are created, one for each side of the two-way case construct. As we shall see below, these edges can be selectively removed to allow certain operations to be removed from the case control-block.

The remaining control edges insure that an operation is not scheduled before the case statement has been evaluated. Case statements are assigned an execution delay that is dependent on the type of controller used.

On the input side of the fork construct there is a data dependency exerted by a data flow operation. During execution, the value of this data edge determines the branch of the statement that will be executed. A join construct is used to select the proper data edge and transfers that value to the data flow operations following the case statement. Typically, it is implemented with a multiplexer in the data path.

Control flow transformations:

As presented, the case construct resembles the conditional *basic blocks* used in CMU's value trace [29], and the *control blocks* used in trace scheduling [31].

However, in the representation used here, the control block boundaries are defined much less rigidly. This allows for the following control flow transformations:

- An operation that is not explicitly defined as part of the block may still be scheduled in the same control-step as an operation in the block. In this case, the operation becomes part of the block and is implicitly duplicated in all branches of the case statement. For example, in Figure 15 on page 78 of "Chapter 6. InHAL: Force-Directed Scheduling", the add and subtract operations on the far left are implicitly duplicated in both branches of the case construct.
- When an operation that is part of the case control-block does not transfer data to the *Join* construct that delimits the end of the block, then it may be scheduled after that construct. This could occur when the data generated by that operation is used unconditionally by every other operation it feeds. In this case, the operation is implicitly removed from the control block and will execute unconditionally. This may be done to shorten the critical path and improve performance.
- By allowing for the selective removal of control edges between the fork and an operation, the operation can be scheduled before the fork. As above, the operation will then execute unconditionally and this could shorten the critical path.
- Finally, multiple control-blocks may be scheduled in overlapping c-steps, i.e. the operations of these blocks may share some control-steps.

These are exactly the types of transformations used in Fisher's *trace scheduling* technique for microcode compaction although they use them for a different purpose; to minimize the number of control-steps, given a fixed data path architecture.

The advantage of our representation is that these transformations can be performed implicitly by the force-directed scheduling algorithm presented in Chapter 6. There, we will demonstrate the use of *distribution graphs* and show how the side-effects of each transformation are taken into account.

Loop Construct:

The loop construct is quite similar to a two-way case statement. Here, one branch of the fork leads to the operations contained in the loop while the second one leads to the operations succeeding the loop.

The join construct has a slightly different meaning here. On the first iteration, it selects the data edges corresponding to the initial loop value. For the next n iterations, it selects the data edge corresponding to values computed within the loop body. A simple behavioral description containing a loop is translated to our CDFG format in Figure 6 below.

Contrary to the case construct, operations that are not defined as part of the loop cannot be scheduled in the same control-steps as the loop operations. However, operations that have no data dependencies with the operations in the loop body may be scheduled either before or after the loop.

Note: The control constructs are depicted here as they are defined internally in the system. Despite its precision, the explicit graphical description used here does not lend itself easily to the depiction of large examples. We will use simpler graphical representations in the remainder of

the thesis; however, the same internal data representations are assumed.

4.1.3 Timing Constraints

There are two types of timing constraints on operations in a CDFG:

1. Global constraints
2. Local constraints

A global constraint is applied to a single CDFG body. It implies that all of the operations in the graph must be executed within the maximum timing constraint.

Local constraints imply a minimum or maximum time constraint between two operations. They can be used to enforce special timing behavior. Nestor has shown their usefulness for the synthesis of interfaces [30], [32], where I/O timing constraints are predominant.

A local constraint is implemented in the HAL system as a dummy *timing operation* that links two operations via control edges similar to the ones used in case and loop constructs. The timing operation contains a minimum and maximum delay value that is taken into account during scheduling.

4.2 STRUCTURAL REPRESENTATIONS

The second main class of representations corresponds to the structural axis of the Y-diagram of Figure 1 on page 5.

There are four main classes of structural units:

1. Functional units (FU)
2. Storage units (SU)
3. Interconnection units (IU)
4. I/O units

Each of these classes will be summarily described in the next three sections. The corresponding graphical symbols are given in Figure 7 on page 49.

4.2.1 Functional Units

Functional units (FU) are allocated to perform regular arithmetic and logic operations. They may be single function units (e.g. adder, comparator), or multi-function units (e.g. ALU). Each FU class is assigned technology specific propagation delay and cell area parameters that are subsequently used in scheduling and data path synthesis.

Operations are bound to FUs in a many to one fashion. These operations must either be mutually exclusive or scheduled in non-overlapping control-steps.

4.2.2 Storage Units

Storage units (SU) are created to store data variables when a data edge traverses a control-step boundary. Single element SUs (registers) and multiple element SUs (memories) can be used and may be assigned single or multiple bit-widths. The symbol for a register is given in Figure 7.

4.2.3 Interconnection Units

The third main class of structural units, interconnection units (IU), perform data transfers between FUs and storage units. A data transfer can be implemented using one or more of the four available IU types:

1. Direct single connection
2. Direct fanout connection
3. Mux
4. Bus

IUs are assigned a specific bit-width. When they are created, the program insures that their bit-width is equal to the largest bit-width of the data-transfer operations they perform. The data path fragment of Figure 7 includes all four IU types.

4.2.4 Input/Output Units

The last main class of structural units, input/output units or I/O ports, are used to describe the circuit's interface. They are often assigned local timing constraints to allow the implementation of complex I/O protocols. Their representation is given in Figure 7.

A representation for control I/O ports is also defined and is used to specify the interface of the data path with the control path.

CHAPTER 5. SYSTEM IMPLEMENTATION

The HAL software is written on a Xerox 1108 Lisp machine using the LOOPS multi-paradigm programming system [33] and consists of approximately twelve thousand lines of code. The decision to use this type of system stems from the fact that different tasks are best suited to different programming styles. In turn, each style has an associated computing efficiency that must be traded off against the ease of representation. For an exploratory project such as this one, where the need for rapid prototyping outweighs the requirements for system speed, it is essential that the implementation vehicle support a straightforward encoding of the algorithms, knowledge and design representations.

High-level global tasks such as the allocation of functional units are well suited to knowledge expressed in the form of heuristics. A rule-based production system is ideal for this type of knowledge representation. On the other hand, tasks such as scheduling and register binding are far better suited to purely algorithmic approaches, both for ease of representation and computing efficiency. Functional programming paradigms such as Lisp [34] could be used effectively for this task.

Finally, the overall representation and classification of the graphs, behavioral operations and structural units would benefit from a rich data representation scheme. Object-based programming environments fill this requirement [35].

5.1 PROGRAMMING PARADIGMS

The LOOPS programming system features:

1. InterLISP — list processing and functional programming
2. Rulesets — for creation of rule-based expert systems
3. Object-based programming with multiple superclasses
4. Active values — invocation of procedures when a variable is read or set (useful for simulation & testing)

The possibility of defining synthesis subtasks in a functional paradigm (InterLISP) or a rule-based paradigm (LOOPS rule sets) satisfies the requirements for easy encoding of algorithmic or heuristic knowledge by using the most appropriate paradigm.

However, neither of the functional or rule-based paradigms offer an effective high-level data structure. Fortunately, the LOOPS system embodies yet another useful programming convenience: the object-oriented paradigm.

5.2 THE OBJECT PARADIGM

The overall structure of the HAL system is object-based. For example, the CDFG, functional units, registers, etc. are all represented as objects. The rule sets and synthesis algorithms are described as *methods* of the objects.

The main advantage of object-oriented frameworks is that they make possible a direct and natural correspondence between the digital system and its model [35]. The origins of this approach can be found in artificial intelligence (AI) knowledge representation techniques, especially semantic networks, and in programming languages such as Smalltalk [36]. The object representation scheme in LOOPS is largely based on the latter.

Objects are organized into classes for the purpose of capturing common characteristics. These classes in turn are ordered hierarchically. A class placed at a lower hierarchical level inherits all the properties of its superclass (unless these properties are redefined in the class itself). Inheritance is useful here because it allows the programmer to limit his specification to the ways in which the subclass differs from its superclass(es).

A class can be viewed as a *template* that describes the common characteristics of a group of related objects. An

instance is a distinct copy of the template that can later have its contents altered, but within the boundaries set by its parent template. Typically, only the leaf classes of the inheritance tree are instantiated.

The class hierarchy of the HAL system is illustrated in the inheritance trees of Figure 8, Figure 9 and Figure 10. The highest level class is the *HalObject* as depicted in Figure 8. Two important subclasses are the *Element* and *Graph* classes.

The *Element* class is the ancestor of the two most important HAL object classes: the operation class, which corresponds to the behavioral operations forming the CDFG (its descendant classes are given in Figure 9); the operator class, which represents all the structural units of the synthesized datapath (its descendants are depicted in Figure 10).

The *Graph* class includes the four main graphs used in the system:

- **CSG (Control-Step Graph):** This graph contains all scheduling and control information required for the subsequent synthesis of a microprogrammed or PLA-based FSM. The global timing constraint and the clock cycle time are also stored here.
- **CDFG (Control and Data Flow Graph):** This is the main graph that holds all the control and data flow operations described previously.
- **DP (Data Path):** This graph contains all the structural units of the design synthesized from the CDFG.
- **DG (Distribution Graph):** These graphs represent operation distributions and are used for scheduling. They will be described further in "Chapter 6. InHAL: Force-Directed Scheduling".

5.2.1 Properties of Behavioral Objects

This class (labelled *Operation* in the inheritance tree) describes the behavioral objects that are derived from the operations of the CDFG. Operations are assigned to an object by creating an instance of the corresponding operation class (e.g. Add, And, Or, etc.).

2

Logic and Arithmetic Operations: A major subdivision of the operation class is the logic and arithmetic operation class (labelled *MathOpn*). Their graphical representation was given in Figure 4 on page 40 of "Chapter 4. Representations for Synthesis".

As illustrated by the inheritance tree, this class is further subdivided into one or two operand and invertible or non-invertible operations. The latter distinction will allow us to perform pairwise operand interchanges to minimize the number of multiplexer inputs, as we shall see later.

Control Operations: Another important subclass is the conditional operation class (labelled *CondOpn*) that represents control statements (case, loop). The detailed graphical representations of these operations was given in Figure 5 on page 42 and Figure 6 on page 45.

Storage Operations: The storage operation class is subdivided into regular storage and array storage classes (the local storage class will be described later). The regular storage operations represent non-subscripted data variables, while array storage is used for array variable structures.

Input/Output Operations: Finally, the I/O class is used to represent input/output operations. I/O behavior and timing constraints are implemented through these objects.

5.2.2. Properties of Structural Objects

This class (labelled *Operator* in the inheritance tree) describes all the objects that represent the hardware components of the final data path.

Structural Object Classes

Figure 10. Class hierarchy of structural objects.

The main subdivisions are the functional unit class (labelled *MathOpr*), the storage operator class, the data transfer operator class and the I/O port classes. Instances of these classes are bound to one or more instances of the operation class, with the exception of data transfer operators that are assigned to data dependency arcs between operations, and are not bound to any particular operation instance.

Registers are bound to one or more storage operations. They are used to store intermediate results available at the output of functional units. Arrays (RAM, ROM or register array) are generally bound to *ReadArray* or *WriteArray* operations. In turn, these operations correspond to indexed arrays in the behavioral description.

An interesting use of multiple inheritance (inheritance from more than one direct ancestor) is the definition of the ALU class that is defined as a subclass of two or more single function units. In other words, it inherits the combined properties of its parent classes. Following this reasoning, the ALU in Figure 10 inherits the combined properties of the adder, subtractor and comparator classes.

Furthermore, since classes can be created dynamically, new

ALU types can be created at will during the synthesis process:

¹ It is also a subclass of the *ALU* class (not depicted here) from which it inherits properties specific to ALUs, such as an additional control input port.

CHAPTER 6. INHAL: FORCE-DIRECTED SCHEDULING

The *InHAL* module is responsible for performing operation scheduling. This consists of assigning operations to specific control-steps so as to minimize functional unit, storage and interconnect costs. As shown earlier in Figure 3 on page 19 of "Chapter 2. System Overview", this is the first main step in the overall synthesis process.

Scheduling is a task whose importance has been recognized in many systems [25], [26], [27], [37]. According to Gajski [37], it is "perhaps the single most important step during the architecture synthesis". Ironically, it is also the one that has received the least attention in past literature. Notable exceptions are the papers presented by Emil Girczyc of the University of Alberta [25] and By Alice Parker's group from the University of Southern California [26], [27].

Operation scheduling determines the serial/parallel nature of the design and approximates cost-speed trade-offs [5]. If the design is subjected to a speed constraint, the scheduling algorithm will attempt to make sufficient operations run in parallel to meet the constraint. Conversely, if there is a limit on chip area, the scheduler can be asked to serialize operations to give the maximum speed consistent with the constraint.

Effective scheduling is crucial as it affects four essential aspects of the subsequent synthesized design:

- The number and type of functional units allocated (such as adders, ALUs, etc.)
- The timing constraints on these functional units
- The storage requirements (registers, memory)
- The data transfer requirements (buses, muxes, leads)

Ultimately, this translates to:

- The area of the design given a speed constraint
- The speed of the design given an area cost constraint

In this chapter, we will first describe related work in the area of scheduling, and then present the InHAL module which incorporates a new *force-directed* scheduling algorithm. The refined version of the algorithm will be presented in the following chapter.

6.1 RELATED WORK IN SCHEDULING

The simplest way to perform scheduling is to relegate the task to the user. This is the approach favored by the Silc system [13] under the assumption that the user should explicitly define the parallelism of the design.

6.1.1 Independent scheduling/allocation schemes:

The following systems have been classified as independent scheduling and allocation schemes; strictly speaking however, the scheduling is not totally independent of the allocation in most of these. For example, many of them derive estimates for the operation propagation delays based on the most likely functional unit allocation.

The first and simplest scheme is to schedule operations "as soon as possible" (ASAP) as is done in Carnegie Mellon University's Emerald/Facet system [14] and the recent CATREE system [38] from the University of Waterloo. An example of ASAP scheduling is given in Figure 11 on page 73. This technique has proved useful in the past for near-optimal microcode compaction [39].

A refinement of this concept is ASAP scheduling with conditional postponement of operations. In the MIMOLA system [40], postponement occurs whenever the operation concurrency is higher than the number of available functional units. The recently published Flamel system [41] developed at Stanford and AT&T uses the same idea. The scheme used in Kung, Whitehouse and Kailath's book on digital signal processing [42] is similar, except that an operation is postponed when it blocks a later one with a lower "as late as possible" (ALAP) level.

Continuing along the scale of increasing complexity, we have the algorithms that use list scheduling, as described in [39]. The EMUCS system developed by Hithcock and Thomas [15] at CMU; the behavioral synthesis of interfaces (BSI) system developed by Nestor and Thomas (also at CMU) [32]; and Pangrle & Gjaski's SLICER system developed at the University of Illinois [20] use this type of scheduling algorithm.

In list scheduling, operations are sorted in topological order (top to bottom) using the precedences dictated by data and control dependencies. The sorted operations are then iteratively scheduled into control-steps. When a resource conflict occurs (due to insufficient hardware), one or more operations are deferred by one control-step. The selection of the deferred operation(s) is determined by a local priority function that is applied to all operations that could be assigned to the current control-step.

In the BSI system, the priority function reflects whether placing the operation in the current step will violate a minimum time constraint and whether placing an operation in a later step will violate a maximum time constraint. In the SLICER system, the priority function is based on operation mobilities. The mobility of an operation is defined as the difference between its ASAP time and ALAP time. Operations on the critical path are therefore scheduled first.

6.1.2 Interdependent scheduling/allocation schemes:

The BUD-DAA system [22] from AT&T uses a stepwise refinement approach to the scheduling task. An initial schedule is produced and the design is partitioned using that schedule [24]. The partitioning information is then used to establish a better schedule. BUD uses a list scheduling algorithm with a priority function that is similar to the one used in the SLICER system. However, as opposed to most list scheduling approaches, BUD sorts the operations in depth-first order (bottom-to-top) instead of top-to-bottom.

In the next two systems, the operation scheduling is done concurrently with the functional unit allocation. Elf [25] uses a variation of the list scheduling algorithm where the priority function is based on operation weights and urgencies. An operation's weight is the minimum number of control-steps to execute the operation, added to the weight of its successor along the longest path. The operation urgency is determined by taking a ratio of the operation's weight and the number of control-steps left until its latest completion time. When an operation is delayed its urgency increases which raises its priority for assignment in the next control-step.

The scheme used in the MAHA system [27] from the University of Southern California relies on critical path determi-

nation and the concept of operation freedom to guide scheduling. The freedom of an operation is identical to the mobility calculated in the SLICER system described earlier. However, list scheduling is not used here.

The MAHA system first invokes the Clocking Scheme Synthesis Package (CSSP) written by Park [26]. Here the critical path is determined and divided optimally into n steps, one per clock cycle. MAHA then allocates functional units for the critical path in a first-come first-served fashion. The notion of freedom is used to guide the scheduling of the off-critical-path nodes. The node with the smallest freedom is chosen for allocation.

6.1.3 Summary of scheduling approaches

The scheduling approaches can be summarized as follows:

- Independent scheduling/allocation
 - ASAP scheduling approaches:
 - Direct (Facet-Emerald, CATREE)
 - Conditional deferment (MIMOLA, Flamel, [42])
 - List scheduling approaches:
 - Priority function: time constraints (BSI)
 - Priority function: mobility (Slicer)
- Interdependent scheduling/allocation
 - User-defined schedule (Silc)
 - List scheduling approaches:
 - Priority function: critical path, use of depth-first ordering (BUD-DAA)
 - Priority function: urgency (Elf)
 - Freedom and critical path scheduling (MAHA)
 - Force-directed scheduling (HAL)

6.2 COMPARISON WITH FORCE-DIRECTED SCHEDULING

The *force-directed* approach to scheduling is fundamentally different from the approaches discussed above. A few of the most important differences are described here.

6.2.1 Local vs global evaluations:

This is, in our view, the most important novel contribution of the HAL system to the scheduling/allocation task. Virtually all the systems discussed rely on a local evaluation of the effect of the control-step assignment of the current operation considered.

The HAL system considers the related effects of a control-step assignment on all the predecessor and successor operations. Every feasible assignment of each operation to any of its possible control-steps is considered at each iteration.

Moreover, the assignment's impact on the overall concurrency of operations is evaluated for three types of operations:

1. Arithmetic and logic operations (executed by functional units)
2. Storage operations (realized with registers)
3. Data transfer operations (realized with buses or muxes)

Unscheduled operations also contribute to the determination of the operation concurrency except that their contribution is probabilistic rather than deterministic.

Functional and structural pipeline scheduling are supported with extremely simple modifications of the algorithm. The ease with which they were implemented is directly related to the algorithm's global nature.

Finally, tradeoffs in functional unit, register and interconnect requirements are resolved using a built-in cost weighting mechanism. Using this mechanism, operation scheduling priorities are set according to their associated hardware area costs.

While this global approach would seem to imply large computation costs, the algorithm is $O(cn^2)$, where c is the time constraint (expressed in control-steps) and n is the number of operations. This relatively low order of complexity is due to an efficient method of propagating the side-effects of an operation's control-step assignment as described in section "Complexity Considerations" on page 91.

6.2.2 Independent vs interdependent scheduling/allocation

In the Facet, CATREE, MIMOLA, BSI, and Slicer systems, the scheduling is performed more or less independently of the functional unit allocation. The Elf and MAHA systems do both tasks concurrently.

The HAL system, on the other hand, does both tasks separately but not independently. The operation scheduling and functional unit allocation are determined by stepwise refinement as depicted in Figure 42 on page 155 of "Chapter 10. Scheduling and Allocation by Stepwise Refinement". The BUD-DAA system uses a similar approach.

6.2.3 Topological vs middle-out scheduling:

The final difference relates to the order of selection of operations during the scheduling process. In all the systems mentioned above (with the exception of MAHA), the operations are scheduled iteratively by using their topological order (depth-first order in BUD). The MAHA system first schedules the operations on the critical path and then schedules the remaining ones in order of increasing freedom.

In the HAL system, the ordering is based on the lowest force. This is a quantity which reflects the effect of an attempted control-step assignment on the overall operation concurrency. The force is positive if the assignment causes an increase of operation concurrency and negative for a decrease. The value is independent of the precedence ordering.

In all the systems, apart from BUD-DAA, MAHA and HAL, modifying speed or area constraints for a given example will not change the scheduling order significantly. In the MAHA and HAL systems, the ordering will be modified to adjust to the new constraint, as reflected by new freedom or force values. In the BUD-DAA system, modifying a constraint results in a new design partition which in turn may affect the scheduling order.

6.2.4 Freedom vs Force:

Although the notion of freedom is contained in the HAL system, it attempts to minimize the total force as opposed to favoring the lowest freedom. (No, this is not a subsection of a political science essay).

There are many cases when scheduling an operation with *high freedom* will have the most beneficial effect on the overall concurrency of operations in the DFG. In "Chapter 12. Design Examples", we will present an example where this occurs. This of course does not guarantee that it is the best possible assignment; only that it is the best assignment at that iteration.

6.3 BASIC FORCE-DIRECTED SCHEDULING ALGORITHM

In force-directed scheduling, the selection of the operation to be scheduled and of the control-step to which it will be assigned, is based on an evaluation of the move which causes the most balanced distribution of operations in each control-step. This algorithm is an iterative one in which one operation is scheduled at each iteration.

The intent is to reduce the number of functional units, registers and buses required by balancing the concurrency of the operations assigned to them, but without lengthening the total execution time. Concurrency balancing helps to achieve high utilization — or low idle time — of structural units, which in turn minimizes the number of units required.

In this section we describe a simplified version of the algorithm that attempts to minimize only the concurrency of arithmetic and logic operations. We will also temporarily assume that all operations execute in one control-step (we will use the term *c-step* from now on). The refined algorithm is presented in the next chapter.

6.3.1 Determination of time frames

The first step involves the determination of both an ASAP (as soon as possible) schedule and an ALAP (as late as possible) schedule. Combining results for both schedules will determine the possible time frames for each operation.

To illustrate this, we will use the example first presented in [4] and subsequently used in [19], [20], [21] and [38]. The algorithmic description and the corresponding data flow graph for this example are given in Figure 11.

Solution of Differential Equation:

$$y'' + 3xy' + 3y = 0$$

while (x < a) repeat:

 x1 = x + dx

 u1 = u - (3 * x * u * dx) - (3 * y * dx)

 y1 = y + (u * dx)

 x = x1; u = u1; y = y1

end

Data Flow Graph (with ASAP scheduling shown)

Figure 11. Data flow graph (DFG) for first example

The raw DFG would not have operations scheduled in time (control-steps). Here, ASAP scheduling is shown. For simplicity, it will be temporarily assumed that all operations execute in one c-step and that operation chaining (i.e. scheduling two sequential data-dependent operations in the same c-step) is not allowed.

The ASAP and ALAP operation scheduling are shown (in simplified form) in Figure 12.

By noting that an operation can be assigned to any c-step between its ASAP and ALAP c-steps, one can draw a time frame diagram (Figure 13). Here the width of the box containing a particular operation represents the probability that the operation will be eventually placed in a given time slot. A useful heuristic is to assume uniform probability of assigning an operation to any feasible c-step. The area of each operation is always one, but it is 'stretched' along its time frame.

This method of assigning probabilities is identical to the one used in Nagle's attraction algorithm [43]. The resemblance ends there however as the attraction algorithm is used for a very different application — namely the synthesis of a control path for a predefined data path. Furthermore, Nagle uses probabilities to select operation pairs that, if scheduled into the same c-step, will minimize the height of the control store without incurring large increases in the control word width. The pair of operations selected are the ones for which the product of the two individual probabilities is highest.

As we will see in the next sections, probabilities are used in the HAL system in a different way and for a different purpose — i.e. the minimization of data path costs.

6.3.2 Creation of distribution graphs (DGs)

The next step is to take the summation of the probabilities of each type of operation for each c-step (control-step) of the CDFG. The resulting *distribution graphs* (DGs) indicate the concurrency of similar operations. For each DG, the distribution in c-step i is given by:

$$DG(i) = \text{Sum} [\text{Opn_Prob}(i)] \quad (1)$$

where $\text{Sum}[]$ = Sum over all operations of a single type
and $\text{Opn_Prob}(i)$ = Operation Probability in c-step i

The distribution graphs derived from Figure 13 are given in Figure 14.

The first graph represents the distribution of the multiply operations, and the next one combines the distributions of the add, subtract and compare operations. The latter three operations are actually assigned to separate DGs but are grouped here for brevity. Each horizontal bar of the DGs corresponds to a distinct c-step. The boxed c-steps delineate the time frames where operations can occur. The others are excluded due to precedence constraints in the CDFG.

Conditional statements:

The presence of conditional statements (e.g. if-then-else and case statements) causes some operations to be mutually exclusive. When these operations can be executed on the

same type of FU (functional unit), they can be scheduled into the same c-step without increasing the number of required FUs. The same FU is simply shared by those operations as they will never be executed concurrently.

We use the following strategy to take advantage of this observation: for each c-step in which the time frames of the mutually exclusive operations intersect, the probability of only one of these is added to the corresponding DG. The operation selected is the one with the highest probability. This is illustrated for the CDFG fragment of Figure 15. The fork and join operations correspond to an if-then-else statement.

Without special treatment of the mutually exclusive additions, the total distribution would be 1.5 in both c-

steps. The unscheduled addition would then have an equal probability of being assigned to either c-step. It is obviously preferable to schedule it in the first c-step, as in this case only one adder will be required. This is exactly what will happen using the strategy just described due to the reduced distribution in the first c-step.

Trace Scheduling Transformations:

In Chapter 4, we described a series of transformations similar to those used in *trace scheduling* [31]. Here we present the two main classes of transformations and the methods used to take their side-effects into account.

1. An operation that is not explicitly defined as part of a case control-block is scheduled in the same control-step as an operation in the block. This is the case for the add and subtract operations on the far left of Figure 15. When this occurs, the operation is not assumed mutually exclusive with any others in the block. As described above, its probability is added unconditionally to the DG. This is equivalent to the duplication of the operation in each branch of the case control-block.
2. An operation that is part of a case block is allowed to be scheduled before (after) the fork (join) construct, i.e. its time frame extends before (after) that of the

fork construct. Therefore, for some c-steps, the operation ceases to be mutually exclusive with the ones in the other branches of the case block. In the c-steps where this occurs, the operation's probability is added unconditionally to the DG.

In summary, operations can be moved in and out of control-blocks by the scheduling algorithm to improve performance. The side effects of these moves are taken into account by selective addition of the operation probabilities to the DG.

6.3.3 Calculation of 'self' forces

Each operation of the CDFG will have a *self force* associated with each c-step i of its time frame. This is a quantity which reflects the effect of an attempted control-step assignment on the overall operation concurrency. It is positive if the assignment causes an increase of operation concurrency and negative for a decrease.

The force is much like that exerted by a 'spring' that obeys Hooke's law; $F = Kx$. K represents the spring's constant (rigidity), x the displacement and F the force caused by the displacement.

More precisely, each DG can be represented as a series of springs (one for each c-step) that will exert forces on all operations. The constant of each spring K is given by the value of $DG(i)$ where i is the c-step number for which the force is calculated. The displacement of the spring x is given by the increase (or decrease) of the probability of the operation in the c-step due to a rescheduling of the operation. For a given operation whose initial time frame spans c-steps t to b ($t \leq b$), the force in c-step i is given by:

$$\text{Force}(i) = DG(i) * x(i) \quad (2a)$$

where $DG(i)$ = the current distribution value
 (i.e. the spring's constant)
 and $x(i)$ = the change in the operation's probability
 (i.e. the spring's displacement)

The total self force associated with the assignment of an operation to c-step j is simply:

$$\text{self_Force}(j) = \text{Sum} [\text{Force}(i)] , \text{ where} \quad (2b)$$

$\text{Sum} []$ = Sum over all c-steps bounded by the time frame
 i.e. ($t \leq i \leq b$)

We will illustrate this equation by using the partial time frame diagram of Figure 16. The constant of each of the three springs corresponds to the value of the multiplication DG given in Figure 14 for the first three c-steps. We will attempt to schedule the circled multiply operation in c-step 1 as depicted in Figure 16 b). The probability of the operation will change from $1/2$ to 1 in c-step 1 and from

1/2 to 0 in c-step 2. These probability shifts correspond to the displacement x of each of the 'springs'. The resulting force associated with the assignment to c-step 1 is the sum of forces caused by the spring displacements in both c-steps of the time frame and is given by:

$$\begin{aligned} \text{self_Force}(1) &= \text{Force}(1) + \text{Force}(2) \\ &= (\text{DG}(1) * x(1)) + (\text{DG}(2) * x(2)) \end{aligned}$$

Using the values from Figure 14 and Figure 16 we obtain:

$$\begin{aligned} \text{self_Force}(1) &= (2.833 * +0.5) + (2.333 * -0.5) \\ &= + 0.25 \end{aligned}$$

The force is positive as expected because the concurrency in c-step 1 is higher than in c-step 2. Scheduling the multiplication in that c-step will have an adverse effect on the overall distribution.

Force Calculations

Added force = Kx

K : Current value of distribution graph

x : Added probability (for new scheduling)

Figure 16. Time frame modifications for force calculations.

Alternate Interpretation of Force

An alternate interpretation of the meaning of Force follows from the evaluation of (2b). The total *self force* associated with the assignment of an operation to c-step j is simply:

$$\text{self_Force}(j) = \text{DG}(j) * x(j) + \text{Sum} [\text{DG}(i) * x(i)]$$

Sum [] = Sum over all c-steps bounded by the time frame,
but excluding c-step j; i.e. ($t \leq i \leq b$, $i \neq j$)

Evaluating the displacements $x(j)$ and $x(i)$ yields:

$$\begin{aligned} x(j) &= (h-1)/h, \text{ and} \\ x(i) &= -1/h, \text{ where} \end{aligned}$$

$h = (b - t + 1)$: the time frame height

After rearrangements, we obtain:

$$\text{self_Force}(j) = \text{DG}(j) - \text{Sum} [\text{DG}(i)/h] \quad (2c)$$

Sum [] = Sum over all c-steps bounded by the time frame,
i.e. ($t \leq i \leq b$)

Simply put, the force associated with the tentative assignment of an operation to c-step j is equal to the difference between the distribution value in that c-step and the average of the distribution values for the c-steps bounded by the operation's initial time frame. In Figure 16 b), the use of equation (2c) yields:

$$\begin{aligned} \text{self_Force}(1) &= \text{DG}(1) - \text{Sum} [\text{DG}(i)/2], \quad (1 \leq i \leq 2) \\ &= 2.833 - (2.833 + 2.333)/2 \\ &= + 0.25 \end{aligned}$$

Which of course is identical to the result obtained earlier.

For the more general case, i.e. the force associated with the reduction of the initial time frame (bounded by c-steps t and b) to a new time frame (bounded by c-steps nt and nb) we may use the following equation:

$$\text{self_Force}(nt, nb) = \text{New_Sum}[DG(i)/nh] - \text{Sum}[DG(i)/h] \quad (2d)$$

$\text{New_Sum} []$ = Sum over all c-steps bounded by the new time frame ($nt \leq i \leq nb$)

$\text{Sum} []$ = Sum over all c-steps bounded by the initial time frame ($t \leq i \leq b$)

$nh = (nb - nt + 1)$: the new time frame height

In other words, the force is equal to the difference between the average distribution of the new time frame and the average distribution of the initial one.

This formulation of the force equation is more computationally efficient for two reasons. The first is that the initial average distribution need only be calculated once when evaluating the force associated with all c-steps of the time frame. The second is that it involves mostly subtractions. On the other hand, due to its more intuitive physical interpretation, we will use the former equations (2a, 2b) in the remainder of the text.

6.3.4 Calculation of predecessor and successor forces

Assigning an operation to a specific c-step will often affect the time frames of linked operations in the CDFG. In turn, this will create additional forces that can reduce or even counter the original intended improvement so it is imperative that they be accounted for.

This is achieved by calculating the extra forces due to the implicit modifications of the time frames of linked operations. They shall then be added to the 'self' force. There will be two extra force contributions: the predecessor forces and the successor forces.

For example, if the circled multiply operation in Figure 16 a) was tentatively assigned to c-step two, as in c), the succeeding multiply operation would implicitly be assigned to c-step three. The time frames of the other operations wouldn't be affected. The resulting force associated with that assignment would then be the sum of the two individual forces. The self force of the first multiply is given by:

$$\begin{aligned}\text{self_Force}(2) &= (\text{DG}(1) * x(1)) + (\text{DG}(2) * x(2)) \\ &= (2.833 * -0.5) + (2.333 * +0.5) \\ &= - 0.25\end{aligned}$$

The successor force due to the implicit assignment of the second multiplication to c-step 3 is given by:

$$\begin{aligned}\text{succ_Force}(3) &= (\text{DG}(2) * x(2)) + (\text{DG}(3) * x(3)) \\ &= (2.333 * -0.5) + (0.833 * +0.5) \\ &= - 0.75\end{aligned}$$

The resulting force due to the c-step assignment is:

$$\begin{aligned}\text{Total_Force}(2) &= \text{self_Force}(2) + \text{succ_Force}(3) \\ &= -1\end{aligned}$$

This c-step assignment is more effective than the one attempted earlier, and this is reflected clearly by the force calculations.

The determination of the *self*, *predecessor* and *successor* forces is performed for every feasible c-step of all the operations. The operation to c-step assignment selected is the one with the most negative force (or the lowest positive force). The operation's time frame is then reduced to the selected c-step, and any linked operation that is affected by this change will have its time frame modified accordingly.

The time frame determination, distribution calculation and force evaluation steps are repeated until each operation is assigned to a single c-step. The distribution graphs are updated at each iteration to reflect the current operation distributions. The forces on the remaining unscheduled operations will vary accordingly.

6.3.5 Lookahead

Lookahead is a useful but sometimes costly method of improving an algorithm's effectiveness. In the case of force-directed scheduling, the extra computing costs could not be justified by the small improvements obtained using even a simple one stage lookahead. However, the simple modification presented here can be applied to approximate a simple lookahead scheme, without increasing the algorithm's complexity.

The *self* force calculation (2a) given earlier can be modified to partially reflect the effect of an assignment on the future value of the DG, i.e. following the attempted assignment. The idea is to *temporarily* modify the spring constant $DG(i)$ so that it takes on a value somewhere between the current one and the value that would be obtained after the current iteration, i.e.

$$DG(i) \leq temp_DG(i) \leq next_DG(i)$$

The distribution value after the current iteration is simply given by:

$$next_DG(i) = DG(i) + x(i)$$

It is preferable to use a value for $temp_DG(i)$ that is closer to $DG(i)$ than $next_DG(i)$; after extensive experimentation, the best results were obtained using:

$$\begin{aligned} Force(i) &= temp_DG(i) * x(i) \\ &= (DG(i) + x(i)/3) * x(i) \end{aligned} \quad (2a')$$

which replaces (2a) presented earlier.

The force calculation for the example of Figure 16 b) is now given by:

```

Total_Force(1) = self_Force(1)
self_Force(1) = Force(1) + Force(2)
               = temp_DG(1) * x(1) + temp_DG(2) * x(2)
               = ( ( DG(1) + x(1)/3 ) * x(1)
                 + ( DG(2) + x(2)/3 ) * x(2) )
               = ( 2.833 + 0.5/3 ) * +0.5
                 + ( 2.333 - 0.5/3 ) * -0.5
               = + 0.41667
               /
               (the previous value was + 0.25)

```

This higher value will further reduce the likelihood that the circled multiply operation will be scheduled in c-step 1.

Similarly, the force calculation for the assignment of the circled multiply operation to c-step 2 (as in Figure 16c) is given by:

```

Total_Force(2) = self_Force(2) + succ_Force(3)
               = - 0.6667 (the previous value was -1)

```

This value is not as negative as the one previously calculated. This reflects the fact that the distribution will be more balanced after the c-step assignment, than it was before.

6.3.6 Summary

We can thus summarize the entire process:

1. Determination of time frames
2. Update of DGs (with conditionals taken into account)
3. Calculation of 'self' forces
4. Calculation of predecessor and/or successor forces
5. Scheduling of operation with lowest force

These steps are repeated until each operation is assigned to a single c-step. The distribution graphs are updated at each iteration to reflect the current operation distributions.

Despite the simplicity of our previous example, there are over five hundred possible schedules, of which only a few are optimal. Furthermore, if five c-steps were allowed instead of four, the number of possible schedules rises to over a million. Using force-directed scheduling however, processing time has a linear dependence with the maximum number of c-steps allowed.

Figure 17 depicts the final schedule and the distributions obtained for the example. It is obvious that the operation distributions are optimal. The result was obtained in just three iterations, although there were twice as many

operations to schedule. This is due to the global nature of the algorithm, where linked operations can be scheduled concurrently as all side-effects are taken into account.

6.4 COMPLEXITY CONSIDERATIONS

6.4.1 Direct Method

The algorithm described above is $O(cn^3)$ when implemented in a straightforward fashion (c is the time constraint expressed in c -steps, and n is the total number of operations). This complexity can be derived as follows:

1. Each iteration of the algorithm schedules at least one operation. This implies there can be at most n iterations. However, in most iterations many other operations are scheduled implicitly, so this can be considered a very conservative upper bound.
2. Within each iteration there are n operations for which forces must be calculated.
3. For each of these operations, the force must be calculated for h control-steps, where h is the height of the operation's time frame. In the worst case, where all operations are totally independent, the maximum possible time frame height is equal to c , the global time constraint. Typically though, operations are interdependent so that the average time frame height is a fraction of c .
4. Finally, for each tentative operation to c -step assignment, there may be at most $n-1$ predecessor/successor operations affected, and whose force must also be calculated.

The combined effect of these four considerations yields the $O(cn^3)$ complexity stated above.

6.4.2 Alternate Method

Fortunately, an alternate method of including the forces of predecessor/successor operations can be applied to reduce the order of complexity without affecting the final force values. In this method, the self, predecessor and successor forces are calculated in three separate passes.

First pass: In the first pass, the self_Force of each operation is calculated for all of its possible time frames. Let us consider an operation whose time frame is bounded by c -steps b and t , where $(b - t + 1)$ is the operation's frame

height h . Following an attempted control-step assignment, the operation's new time frame boundaries, nb and nt , can be obtained through three types of transformations:

1. Type 1: The operation is scheduled in c-step j , i.e. ($nb = nt = j$, $nh = 1$). The time frame is therefore reduced to a single c-step. Since ($t \leq j \leq b$), there are h possible force values.
2. Type 2: A predecessor of the operation is scheduled. The operation's time frame will therefore be reduced from the top, i.e. ($t+1 \leq nt \leq b$), ($nb = b$) and ($nh < h$). In this case, there are $h-1$ possible force values.
3. Type 3: A successor of the operation was scheduled. The operation's time frame will be reduced from the bottom, i.e. ($nt = t$), ($t \leq nb \leq b-1$), and ($nh < h$). Once again, there are $h-1$ possible force values.

To store the forces associated with each possible time frame reduction, we will create one vector $V1(j)$ of length h to store Type 1 self_Forces. We will also create two vectors $V2(nt)$ and $V3(nb)$ of length $h-1$ to store Type 2 and Type 3 self_forces. The values of j , nt and nb are subjected to the constraints given above.

The force calculations described above are performed in less than $3*c*n$ steps.

Second pass: In the second pass, the CDFG is traversed in depth-first order, i.e. from the bottom of the graph to the top. The total force for each operation is given by the sum of its self_Force and the stored total force of its direct successors only. Operations without successors (i.e. the

output operations at the bottom of the graph), have a stored total force of zero.

In other words, the value of $V1(j)$ of an operation increases by the sum of the $V2(j+1)$ of all its direct successors (assuming operations execute in one c-step). $V2(nt)$ of the operation increases by the sum of the $V2(nt+1)$ of the direct successors.

Third pass: In the third pass the process is repeated for the predecessor forces by taking the operations in topological (top-to-bottom) order.

Here, the stored value of $V1(j)$ of an operation is increased by the sum of the $V3(j-1)$ of all its direct predecessors, assuming again that operations execute in one c-step. $V3(nb)$ of the operation increases by the sum of the $V3(nb-1)$ of the direct predecessors.

In summary, the process consists of calculating the *self Force* of all operations. The graph is then traversed from bottom to top, and then from top to bottom. Each operation 'queries' its nearest neighbors for their stored force, which in turn is equal to the sum of their own *self Force* and that of their nearest neighbors. We are effectively performing a running sum of all the successor (predecessor) forces in linear time.

6.4.3 Algorithm description for the alternate method

This method can be described more succinctly using the following pseudo-code descriptions.

Assume a force evaluation function:

find_force(Opn, nt, nb) , where

Opn: operation identifier

nt: top of new (reduced) time frame

nb: bottom of new (reduced) time frame

Returns: self_Force of Opn; using equation (2d)
(or 0 if Opn is an I/O)

Also assume a force retrieval function:

ret_force(Opn, nt, nb) , where

Opn, nt, nb: defined as above

Returns: stored self_force
(retrieved from vectors V1, V2 or V3)

The main algorithm is:

Repeat until (all operations scheduled):

Step 1: Find time frames:

1.1 Find ASAP schedule

1.2 Find ALAP schedule

Step 2: Calculate DGs using equation (1)

Step 3: Calculate Forces

3.1 Calculate self_Forces for all possible time frames.

3.2 Take running sum of all successor forces.

3.3 Take running sum of all predecessor forces.

Step 4: Schedule operation with lowest force.

end Repeat;

The first two steps were described in detail earlier in the chapter. We describe the third and fourth steps below.

--- Step 3: Calculate Forces ---

--- Step 3.1: Calculate self Forces ---

Operation_List := list of all operations (excluding I/O)

for Opn in Operation_List do:

for c-step j from t to b do:
V1(j) := find_force(Opn, j, j)
end for;

for nt from t+1 to b do:
V2(nt) := find_force(Opn, nt, b)
end for;

for nb from t to b-1 do:
V3(nb) := find_force(Opn, t, nb)
end for;

end for;

--- Step 3.2: Calculate Successor Forces ---

Operation_List := operations sorted in depth-first order
(bottom-to-top order)

for Opn in Operation_List do:

t = top of time frame (for Opn)
b = bottom of time frame (for Opn)

for nt from t+1 to b do:

Succ_List = list of direct successors of Opn

for Succ in Succ_List do:
no = execution time of Opn
sb = bottom of original time frame of Succ
Succ_Force := ret_force(Succ, nt+no, sb)
V1(nt) := V1(nt) + Succ_Force
V2(nt) := V2(nt) + Succ_Force
end for;

end for;

end for;

--- Step 3.3: Calculate Predecessor Forces ---

Operation_List := operations sorted in topological order
(top-to-bottom order)

for Opn in Operation_List do:

t = top of time frame (for Opn)

b = bottom of time frame (for Opn)

for nb from t to b-1 do:

Pred_List = list of direct predecessors of Opn

for Pred in Pred_List do:

np = execution time of Pred

pt = top of original time frame of Pred

Pred_Force := ret_force(Pred, pt, nb-np)

V1(nb) := V1(nb) + Pred_Force

V3(nb) := V3(nb) + Pred_Force

end for;

end for;

end for;

--- Step 4: Schedule minimum force operation ---

Operation_List := list of operations

Min_Force := infinity (i.e. a large number)

for Opn in Operation_List do:

t = top of time frame (for Opn)

b = bottom of time frame (for Opn)

for j from t to b do:

Current_Force := ret_force(Opn, j, j)

if (Current_Force < Min_Force) then

Min_Force := Current_Force

Min_Opn := Opn

Min_cstep := j

endif;

end for;

end for;

t (for Min_Opn) := Min_cstep

b (for Min_Opn) := Min_cstep

6.4.4 Final Complexity Calculations

As before, the maximum number of iterations of the outer loop is equal to n , the total number of operations. However, we must recalculate the number of iterations for the four steps in the loop.

The first step requires $(2*n)$ iterations. The second requires at most $(c*n)$ iterations, where c is the time constraint. Step 3.1 and step 4 each require $(3*c*n)$ iterations at most.

Steps 3.2 and 3.3 are performed in $(k*c*n)$ iterations each, where k is the average number of operands of all operations. We can assume that each operation has a small finite number of operands. In our system all operations have three or less. This implies that the average number of direct predecessors of all operation is equal to a constant value k that is independent of n .

From symmetry considerations, it can be shown that the average number of direct successors is equal to the same constant value. With k equal to three at most, the maximum number of iterations for all four steps is $(13*c*n + 2*n)$ or $O(cn)$.

In other words, the order of complexity of the algorithm is now reduced to $O(cn^2)$, or simply $O(n^2)$ if c is considered a constant relative to n . This reduction in complexity is accomplished at the expense of a slight increase in computer memory utilization.

It can be argued that c (the time constraint given in c -steps) is equal to n in the very worst case; i.e., when all operations are on the critical path. This effectively raises the algorithm's complexity to $O(n^3)$. However, as mentioned earlier, the average time frame height is typically a small fraction of c . This is true for the case mentioned above, where the time frame height of a critical path operation is equal to one.

A particular case for which this is not true is when most of the operations in the CDFG are independent of one another. In this case, the time frames of all operations would be close or equal to the total number of c -steps. On the other hand, since most operations are independent, the global time constraint can be made very small and often will be. Therefore, the time frame heights will also be reduced.

• In summary then, although the theoretical complexity is $O(n^3)$, in reality nearly all practical examples will exhibit $O(n^2)$ behavior. However, to guarantee this level of com-

plexity in a more strict mathematical sense, the following solution may be applied:

- Perform a preliminary reduction of all time frames which exceed a constant maximum allowable height H (e.g. $H = 10$ c-steps).
- Forces are calculated in the usual fashion, and all the long time frames are reduced simply by removing from the time frame the c-steps with the highest forces.

This additional step is $O(cn)$ and the regular scheduling phase that follows is now reduced to $O(Hn^2)$, where H is a predefined constant. This solution therefore guarantees the desired $O(n^2)$ complexity.

CHAPTER 7. REFINED SCHEDULING ALGORITHM

The basic algorithm just presented is a powerful means of solving a relatively narrow class of problems. To make it usable for the wide range of constraints that a good synthesis tool should address, it was necessary to incorporate a few important refinements. However, we will see that the basic framework was extended in a straightforward and 'painless' fashion to include every refinement presented here. Two cases in point are the algorithm and structural pipeline extensions that required only a few additional program lines. The refinements can be divided into three broad classes:

1. Refinements that extend the algorithm's scope:

- Minimization of bus costs
- Minimization of register costs
- Local timing constraints
- Loops

2. Refinements that ease the inclusion of allocation information into the scheduling process:

- Incorporation of structural unit area costs
- Multi-class DGs (for ALUs)
- Chained and multi-cycle operations
- Scheduling under fixed area constraints

3. Refinements for pipeline scheduling:

- Extension for algorithm pipelining
- Extension for structural pipelining

This chapter will deal with the first two classes, while the pipeline extensions will be presented in the following one.

7.1 MINIMIZATION OF BUS COSTS

As we mentioned earlier, scheduling has an effect on the interconnect cost of the final circuit. For example, the minimum number of buses required is directly related to the number of concurrent data transfers. Scheduling an operation implies that at least two transfers will occur in the c-step to which it is assigned. The first transfer is from the output of the FU to the register storing the result of the operation it performed. The others are from the registers that supply the FU with its operands.

The minimum number of buses required for a given schedule can be found by taking the maximum over all c-steps of the number of simultaneous data transfers. To minimize the number of concurrent transfers and the associated bus costs, we create a special DG that contains the distributions of the data transfers. We will refer to this as the *transfer DG*. Since transfers are directly related to operations, the transfer DG is simply the sum of every operation distribution multiplied by the combined number of distinct inputs and outputs:

$$\text{Trans_DG}(i) = \text{Sum} [\text{Opn_Prob}(i) * \text{Opn_No_InOuts}] \quad (1)$$

where $\text{Sum}[]$ = Sum over all the operation types

and $\text{Opn_Prob}(i)$ = Operation probability in c-step i
 Opn_No_InOuts = Combined number of distinct inputs and outputs for the current operation.
 $\text{Trans_DG}(i)$ = Transfer distribution in c-step i

The additional forces due to these new DGs will be calculated in the same manner as for the regular operations.

7.2 MINIMIZATION OF REGISTER COSTS

The minimum number of registers required for the implementation of a scheduled CDFG is given by the largest number of data arcs traversing a control step boundary. For a CDFG in which many or all of the operations are unscheduled, it is more difficult to establish this lower bound.

In this subsection, we will present a technique that reduces register costs and also determines a lower bound on the register costs, at any stage of the scheduling process.

The word distinct is emphasized here because several operations may have common inputs. When this occurs, each common input should be counted only once. For example, in the first c-step of Figure 11 on page 73 there are only five distinct inputs (i.e. u , x , y , dx and 3) for the five operations. A common input should preferably be linked to the operation with the highest probability.

To achieve register minimization, we must create a new class of operations that we call *storage operations*. A storage operation is created at the output of every source operation that transfers a value to one or more destination operations in a later control-step. We will also need another special DG that will be referred to as a *storage DG*.

When both the source and destinations of a storage operation are scheduled, then the storage operation's distribution coincides with its *lifetime*. In other words, the distribution is bounded by the c-step of the source operation and the c-step of the last destination operation. The distribution value is equal to one, indicating it must exist with probability one for the entire lifetime.

If one or more of the source or destination operations aren't scheduled, then we have to determine probabilistic distributions. This can be done in much the same way as for regular operations. The only complication is that the length of the storage operation, i.e. the length of its lifetime, is now dependent on the final schedule.

For example, in the simple data flow graph of Figure 18 on page 107, there are three possible lifetimes for storage operation S, as depicted in parts a), b) and c) of the figure. This operation stores the value of the source operation s and transfers it to the destination operation d.

The following simple but effective heuristic can be used to quickly estimate the resulting storage distribution. The first step is to determine the lifetimes associated with the ASAP and ALAP schedules (*ASAP_life* and *ALAP_life*) as well as the longest possible lifetime (*max_life*). An estimate of the average length of all possible lifetimes is given by:

$$[\text{avg_life}] = \frac{[\text{ASAP_life}] + [\text{ALAP_life}] + [\text{max_life}]}{3} \quad (2)$$

where *[ASAP_life]* = length of lifetime for ASAP schedule
[ALAP_life] = " " " " ALAP " "
[max_life] = length of longest possible lifetime
= (*ASAP_life_begin* - *ALAP_life_end* + 1)

For our example (Figure 18), the ASAP, ALAP and maximum lifetimes of the storage operation S are:

ASAP_life = (2,3) - *[ASAP_life]* = 2
ALAP_life = (3,4) - *[ALAP_life]* = 2
max_life = (2,4) - *[max_life]* = 3

Therefore, equation (2) yields:

$$[\text{avg_life}] = (2 + 2 + 3) / 3 = 7/3$$

At this point, there are two possible situations. The first is when there is no overlap between the ASAP and ALAP lifetimes.¹ In this case the storage distribution in c-step *i* is given by:

$$\text{storage DG}(i) = \frac{[\text{avg_life}]}{[\text{max_life}]} \quad (3)$$

where ($\text{ASAP_life_begin} \leq i \leq \text{ALAP_life_end}$)

The second situation is when the ASAP and ALAP lifetimes overlap. This is the case in Figure 18 where:

$$\text{overlap} = (3,3) - [\text{overlap}] = 1$$

The overlap represents the c-steps where it is certain that a storage operation will be required. Therefore, for each c-step of the overlap, the distribution must be set equal to one. These overlaps can be determined at any intermediate stage of the scheduling process. This is extremely useful, as they can also be used to set a lower bound on the number of registers required to implement the CDFG, even if any or all of the operations have not been scheduled.

When taking overlaps into account, the storage distribution in c-step i is now given by equation (4).

$$\text{storage DG}(i) = \frac{[\text{avg_life}] - [\text{overlap}]}{[\text{max_life}] - [\text{overlap}]} \quad (4)$$

where ($i < \text{overlap_begin}$)
 ($i > \text{overlap_end}$)

storage DG(i) = 1 , for i inside overlap

i.e. ($\text{overlap_begin} \leq i \leq \text{overlap_end}$)

Storage Lifetimes

d) Storage Distribution (for storage S)

Figure 18. Storage lifetimes for all schedules

For our example, using the lifetimes and the overlap calculated earlier, equation (4) yields:

$$\text{storage DG}(2) = \frac{7/3 - 1}{3 - 1} = 2/3$$

$$\text{storage DG}(3) = 1 \quad (\text{due to overlap})$$

$$\text{storage DG}(4) = 2/3$$

The resulting *storage distribution* is depicted in Figure 18 d).

This process is repeated for all storage operations and the separate distributions are added to a single *storage DG*. Using the scheduling process described in "Chapter 6. InHAL: Force-Directed Scheduling", the additional force exerted by these new DGs can be calculated in much the same way as that presented for regular forces (equation 2a). This force is added to an operation's *self force* by applying a mechanism similar to the one used for predecessor and successor forces.

In Figure 19, we compare the register requirements for our differential equation example using a simple ASAP schedule and the force-directed schedule. For the ASAP scheduling at least seven registers are required as there are seven values to be stored at the end of c-step one. For the force-directed schedule this number is reduced to five.

7.3 LOCAL TIMING CONSTRAINTS

In certain classes of applications it is useful to be able to specify minimum/maximum timing constraints between operation pairs. This has been shown to be particularly useful for the behavioral synthesis of interfaces [32].

This is implemented in the HAL system as a dummy *timing operation* that links two operations via control edges similar to the ones used in case and loop constructs. These

timing operations can be inserted between any two regular operations and specify that one must be scheduled before the other. The timing operation contains a minimum and maximum delay value that is taken into account during scheduling. A pair of minimum and/or maximum timing constraints are associated with each timing operation. They are taken into account during the time frame determination phase of the scheduling algorithm.

7.4 SCHEDULING WITH LOOPS

Whenever a loop is specified in a behavioral description, the user must also specify a constraint on the loop iteration time. This constraint is implemented in the CDFG by asserting a local maximum timing constraint between the *Loop Fork* and *Loop Join* operations that delimit the loop. The operations bounded by the loop are scheduled within the constraint using the regular algorithm. For multiple embedded loops, the operations of the innermost loop are scheduled first, relative to the local time constraint. When this is done, the entire loop is treated as a single operation with an execution time that is equal to the loop's local time constraint. Operations external to the loop cannot be scheduled concurrently with loop operations. This process is repeated for all loops until the outermost loop is scheduled.

To force pipelining, the user can place a local time constraint on a loop which is shorter than the total execution time of the operations on the longest path through the loop. When this occurs, the scheduling is done using the extension that is described later in "Extension for Algorithm Pipelining" on page 126 of "Chapter 8. Extensions for Pipeline Scheduling". Here the latency is set equal to the time constraint.

In other words, any operation that is scheduled into a c-step that is greater than the latency, will actually be performed concurrently with some of the operations in the next iteration. This entails that operations from the current iteration will affect the time frames of those in the next one. This is taken into account in the time frame determination process.

7.5 INCORPORATION OF STRUCTURAL UNIT AREA COSTS

Different structural units have different realization costs. Operation types associated with high-cost units should be given a higher priority in the scheduling process. An easy way to do this is to multiply the constant of the 'spring' associated with the DG by a cost factor that reflects the unit's area. For example, the cost factor of the multiplication DG would be roughly equal to the allocated multi-

plier area, while that of the storage DG, is equal to the register area.

7.6 MULTI-CLASS DGS

In many cases we want to make effective use of multi-function units (e.g. ALUs). To deal with this, the concept of distribution graphs* (DG) was extended so that the distribution of one or more operation types can be stored in a single DG.

For example, if the allocator assigns additions and subtractions to an ALU, then a two class DG will be created in lieu of the two single class ones. Therefore, these two types of operations will tend to be scheduled in different c-steps to make best use of the ALU.

For each multi-class DG, the distribution in c-step i is given by:

* In "Chapter 10. Scheduling and Allocation by Stepwise Refinement", we will see that for FUs, the cost factor is also a function of the associated mux, bus and interconnect area..

$$\text{Multi_DG}(i) = \text{Sum} [\text{Opn_Prob}(i)] \quad (5)$$

where Sum[] = Sum over all operations of
 two or more specified types
 Opn_Prob(i) = Operation probability in c-step i

For example, if an ALU was allocated to perform the add, subtract and compare operations of our previous example, then we would obtain the multiple DG shown earlier in Figure 14 on page 77 of the previous chapter.

7.7 CHAINED OPERATIONS

The possibility of scheduling consecutive data-dependent operations in a single c-step (chaining) has also been incorporated into the system. This feature is implemented in a straightforward fashion by extending the time frames of fast combinatorial operations into the previous and/or next c-steps (when the total propagation delay in those c-steps is less than the clock cycle).

To determine if operations can be chained, a *propagation delay* value must be stored with each operation. When establishing the ASAP schedule, this value represents the running sum of propagation delay values of all the predecessor operations that may be scheduled in the same c-step. Conversely,

for the ALAP schedule it represents the sum of delays for all the successor operations in the same c-step.

7.8 MULTIPLE CONTROL-STEP OPERATIONS

Operations that require multiple c-steps for execution are also supported. They are implemented with a simple extension of the single c-step methodology. This extension takes into account the effect on the time frames evaluation, and includes a new method for calculating the distributions of multiple c-step operations.

Determination of time frames: The time frame determination is modified in a straightforward fashion by extending the approach used for chained operations. For example, let us assume a multiply operation (with a propagation delay of 120 nsec) that is linked to an addition (40 nsec) in a CDFG where the clock cycle is 100 ns and latch delays are 10 ns. If the multiply operation is scheduled in c-step one, then the addition cannot be chained in c-step one. Furthermore, it can only be scheduled in c-step two if the sum of the combined propagation delays plus one latch delay is less than two clock cycles (200 nsec). Here this sum is equal to 170 nsec, so the time frame of the addition will start in c-step two.

Evaluation of operation distributions: The distributions of multiple c-step operations must be calculated differently since each c-step (or stage) of the operation must be taken into account. For example, if a two c-step multiply operation is scheduled in c-step one, its distribution (or probability) is equal to one in both of c-steps one and two.

On the other hand, if the multiplication is not scheduled and may begin in either of c-steps one or two (i.e. its time frame is equal to two), then the distribution contribution of the operation's first stage is equal to one half in each of those two c-steps. Following the same reasoning, the contribution of the second stage is equal to one half in both of c-steps two and three. The combined distribution of both stages of the operation is therefore equal to '1/2', '1' and '1/2' in c-steps one, two and three respectively. The value '1' in the second c-step indicates that one of the two stages must be scheduled in that c-step.

7.9 SCHEDULING UNDER FIXED HARDWARE CONSTRAINTS

The scheduling approach presented above supports the synthesis of near-minimal cost datapaths under fixed timing constraints. Two extensions of this methodology also allow the determination of a schedule with a near-minimal number of c-steps, given fixed hardware constraints.

7.9.1 General hardware constraints

The first extension can be used when either a *global* constraint (e.g. a constraint on the total hardware cost) or a set of *specific* constraints (on the number of FUs of each type) are specified.

The basic idea consists of performing regular force-directed scheduling with a tight timing constraint. During scheduling, hardware resource costs are estimated at every iteration. Whenever a hardware constraint is exceeded (local or global), then an extra c-step is added to the time frames of all operations. Thus the time frames of unscheduled and scheduled operations are extended by one c-step.

This increases the time required to arrive at a solution, but not the order of complexity of the algorithm. This is because the partial scheduling is maintained, i.e. the time frames of operations are simply extended, not recalculated.

7.9.2 Specific hardware constraints

Another form of force-directed scheduling has also proven particularly effective for cases where *specific* hardware constraints are given. Here, the approach is to perform list scheduling as described in [39] except that *force* cal-

culations will be incorporated. We will refer to this technique as *force-directed list scheduling*.

Recall that in list scheduling, operations are sorted in top-to-bottom order by using control and data dependencies. The set of operations that may be placed in a control-step may then be evaluated; we call these the *ready* operations. If the number of *ready* operations of a single type exceeds the number of hardware modules available to perform them, then one or more operations must be deferred. In previous list scheduling algorithms, the selection of the deferred operations is determined by a local priority function such as the ones described in the previous chapter (e.g. freedom, urgency).

In *force-directed list scheduling*, the approach is similar except that *force* is used as the priority function. More precisely, whenever a hardware constraint is exceeded in the course of regular scheduling, force calculations are used to select the best operation(s) to defer. Here, a deferral does not necessarily mean that the operation will be scheduled in the next c-step — only that its time frame has been reduced so that it excludes the current c-step. The deferral that produces the lowest force is chosen. This is repeated until the hardware constraint is met.

Forces are calculated using the method described in the previous chapter. However, as these calculations depend on the existence of time frames, a global time constraint must be temporarily specified. Here it is simply set to the length of the current critical path. This length is increased when the only way of resolving a resource conflict is to defer a critical operation.

Using this approach, the advantages of both types of scheduling are maintained, i.e.

1. High utilization of allocated functional units, as this feature is intrinsic to list scheduling.
2. Global evaluation of all the side-effects of an attempted assignment of an operation to a c-step. This characteristic is the basis of force-directed scheduling.

Comparison of Scheduling Approaches

It is easy to demonstrate that the version of the force algorithm just presented is $O(n^2)$. It is therefore faster than the regular timing-constraint-based version of the algorithm presented earlier. Recent preliminary results have shown that this technique yields slightly better results as well.

On the other hand, this improvement is due to the fact that the user must supply more information about the design, i.e. the number and type of FUs that will be used. This also

implies that the capability of performing FU tradeoffs automatically by the algorithm is lost. Finally, there is no guarantee that the global timing constraint will be respected. The user must try different FU allocations until the constraint is met.

In conclusion, each version of the algorithm has advantages and disadvantages that are orthogonal. The choice of the 'best' one to use is mostly dictated by the type of constraints specified, and by the application itself.

7.10 SCHEDULING UNDER MIXED TIME-AREA CONSTRAINTS

Finally, there are cases where the user has not specified an explicit time or area constraint, but would like a compromise based on some speed-area tradeoff function. Using one of the three methods described, this can be achieved by modifying the time or area constraint iteratively until a satisfactory compromise is attained.

To deal directly with this type of mixed constraint, we propose (yet) another simple extension. This extension can be summarized as follows:

- The user specifies a minimum and maximum range of allowable execution times, t_{min} and t_{max} .

- The time frames and distributions of operations are determined in the usual fashion, using t_{max} as the time constraint.
- The total distribution in c-step i spanning t_{min} and t_{max} is given by:

$$\text{MixedDG}(i) = \text{DG}(i) * \text{Fn}(i - t_{min})$$

for ($t_{min} \leq i \leq t_{max}$)

and $\text{Fn}()$: a function that increases with $(i - t_{min})$

This has the effect of 'stiffening' the springs progressively as i increases. It follows that the chance of an operation being scheduled in higher c-steps decreases according to the time-area function $\text{Fn}(i - t_{min})$.

CHAPTER 8. EXTENSIONS FOR PIPELINE SCHEDULING

The optimized scheduling of pipelined behavioral descriptions is a recent development in the field of behavioral synthesis. Two types of pipelining can be distinguished: *algorithm pipelining* and *structural pipelining* [8].

In algorithm pipelining, the CDFG representation of an algorithmic description is subdivided into sequences of operation stages that will be performed concurrently. Successive stages are streamed into the pipe so that different algorithm instances are executed in an overlapping fashion on a single datapath. This is illustrated in Figure 20, where three algorithm instances are depicted.

Algorithm pipelining often yields better utilizations of functional units since they can be shared by two or more algorithm instances. In this way, the increase in throughput does not necessarily translate to a proportional increase in the number of FUs required.

Parallelism through Algorithm Pipelining

- Different instances of algorithm run concurrently

- Relative throughput increase = No. Cycles / Latency

Figure 20. Algorithm pipelining: An additional level of parallelism is introduced due to the operations of the second stage of *Algorithm Instance'* that run concurrently with the operations of the first stage of *Algorithm Instance'*. In most cases, the increased throughput is obtained at the expense of additional hardware in the datapath.

In *structural pipelining*, temporal parallelism is obtained through the use of pipelined functional units, e.g. a two stage pipelined multiplier. In this case, the operation instances are executed in an overlapping fashion. This is

illustrated for a pair of multiplications in Figure 21. With these multiplications scheduled in successive control-steps, only one pipelined multiplier is required to execute the two operations in three control-steps.

Parallelism through Structural Pipelining

- Use of pipelined functional units
- Different operation instances run concurrently

Figure 21. Structural pipelining

8.1 RELATED WORK

8.1.1 Algorithm pipelining

In Park and Parker's paper [44], the Sehwa system is described as the first synthesis program published in the open literature to perform algorithm pipelining. This system makes use of two polynomial-time pipeline scheduling algorithms:

1. feasible scheduling: schedule with constraints on the total implementation cost.
2. maximal scheduling: schedule for maximum performance, i.e. assuming there is no cost constraint.

Sehwa also incorporates an exhaustive algorithm for optimal scheduling. Here the search time is reduced by using the feasible or maximal schedule as an upper bound. The algorithms are invoked iteratively and each scheduling cycle is guided by the performance and cost estimation for the previous schedule.

Pipeline scheduling has also been recently demonstrated in the current version of the Elf system [45] developed at Audesyn Inc. Here loop winding is used to realize an alternate form of algorithm pipelining. Loop winding partitions each loop iteration according to path length and then winds

these in parallel to form a shorter loop executing at a higher frequency. User-supplied throughput and latency constraints are used to determine the length of the shorter loop. Scheduling is performed by partitioning the critical-path operations into equal length stages. The operation weights described earlier are used to assign non-critical operations to these stages. This is followed by a refinement phase where load balancing [4] is used to reduce the concurrency of similar operations.

8.1.2 Structural pipelining

The Slicer system mentioned in the previous subsection was one of the first systems to perform scheduling optimizations with pipelined functional units. As in regular scheduling, the Slicer system uses mobilities to guide the pipeline scheduling process.

8.1.3 Summary of scheduling approaches

The pipeline scheduling approaches can therefore be summarized as follows:

- Algorithm pipelining
 - Feasible, maximal or exhaustive scheduling (Sehwa)
 - Loop winding using operation weights (Elf)
 - Modified force-directed scheduling (HAL)
- Structural pipelining
 - List scheduling (Slicer)
 - Modified force-directed scheduling (HAL)

8.2 EXTENSION FOR ALGORITHM PIPELINING

The task of scheduling a pipelined algorithm is resolved here with a simple and straightforward modification of the regular force-directed scheduling algorithm. For a given latency L , the operations scheduled into c -steps $i + kL$ ($k = 0, 1, 2, \dots$) run concurrently. So now we must balance the distribution across all groups of concurrent c -steps, as opposed to the previous balancing across all individual c -steps. For our previous example, for a latency of two c -steps (i.e. $L=2$), the concurrent c -step groups are $(1', 3)$ and $(2', 4)$, as shown in Figure 22.

Regular Schedule

Figure 22. Schedule obtained using regular scheduling algorithm.

Our previous schedule now requires the allocation of at least four multipliers instead of two. The adder, subtractor and comparator allocations remain unchanged. This simple example illustrates the fact that the regular scheduling does not guarantee an even distribution of operations across groups of concurrent c-steps.

Fortunately, this can be improved by cutting the distribution graphs horizontally and superimposing the slices. The cut boundary is determined by the value of the latency.

In the previous example, the DG would be sliced in the middle and both halves superimposed so that the DG is reduced to two rows. The first corresponds to the combined distribution in c-steps 1 and 3 (as they are now concurrent) and the second to that of c-steps 2 and 4. From the regular DGs given in Figure 14 on page 77, we obtain the new DGs illustrated in Figure 23.

Figure 23. Distribution graph evaluation using algorithm pipelining.

By performing regular force-directed scheduling with these modified DGs, the operation distribution will be balanced while taking into account the additional level of parallelism due to algorithm pipelining.

Figure 24 illustrates the new schedule obtained by using this method. This solution requires three multipliers instead of four and all three are fully utilized. We therefore obtain a twofold increase in throughput while the number of multipliers is increased only by a factor of 1.5 and the adder, subtractor and comparator allocations are unchanged.

Figure 24. Schedule obtained using algorithm pipelining extension.

This technique can be used for straightline code as well as for loops. The example presented here is actually a loop, so we have in effect performed *loop winding* as described in [45].

8.3 EXTENSION FOR STRUCTURAL PIPELINING

The problem of scheduling operations assigned to pipelined functional units can be solved with another very simple extension. This extension is based on the fact that once the first stage of a pipelined functional unit is empty, it is considered available.

For example, let us make the realistic assumption that the multiplications in our previous example require two c-steps for execution. Furthermore, if a two stage pipelined multiplier is selected in the allocation phase, then independent multiplications such as $u * dx$ and $3 * x$ (refer to Figure 11 on page 73 of "Chapter 6. InHAL: Force-Directed Scheduling") could be scheduled in successive c-steps, and executed by a single multiplier. On the other hand, data-dependent multiplications (e.g. $3 * y$ and $dx * (3 * y)$) can only be scheduled in non-consecutive c-steps, the result of the multiplication being available only after two c-steps.

These considerations translate to the following simple extension of the scheduling algorithm: when calculating distributions for pipelined operations we need to sum only the probability of the c-step(s) corresponding to the first stage. The remaining ones are given an artificial probability of zero. The scheduler will then be effectively balancing the distribution of the first stage only, which accounts

for the fact that the functional unit is available after that stage is empty. The time frame calculations are unchanged however, as this ensures that data-dependent multi-cycle operations are not scheduled in successive c-steps..

To illustrate the algorithm extension, we will use our previous example with a new clock cycle of 50 ns (it was previously 100 ns). Multiplications, which have a propagation delay of 80 ns, now require two c-steps for execution. The remaining operations can still be assigned to a single c-step. Using the regular (non-pipelined) scheduling algorithm and a time constraint of 8 c-steps, we obtain the schedule illustrated in Figure 25. The scheduler arrives at a perfectly balanced multiplication distribution; however, since the first stages of each pair of concurrent operations are scheduled in the same c-step, there is no advantage in using pipelined multipliers.

Regular Schedule

Figure 25. Schedule and Distributions using non-pipelined multipliers

On the other hand, if we use the modified scheduling algorithm, we obtain the schedule shown in Figure 26. Here the scheduler obtained a perfect balancing of the first stage of the multiplications, so the use of a single pipelined multiplier will significantly reduce the overall costs. The adder, subtractor and comparator allocations are the same in both cases.

Pipeline Schedule

Figure 26. Schedule and Distributions using pipelined multipliers

CHAPTER 9. MIDHAL: ALLOCATION OF FUNCTIONAL UNITS

9.1 OVERVIEW

The *MidHAL* module is the second main module of the HAL system as depicted in Figure 3 on page 19 of "Chapter 2. System Overview". Its task consists of allocating a set of functional units to perform the operations of the CDFG, using the schedule generated by the *InHAL* module. *MidHAL* will ensure that a sufficient number of FUs is available to realize all operations, but does not bind them to specific ones. This final binding will be realized by the *ExHAL* module described later in "Chapter 11. *ExHAL*: Data Path Synthesis"

The *MidHAL* module accepts as input a scheduled CDFG, a global time constraint and a library of available functional units (FUs) and their associated speed and cost. It then uses heuristic rules to select a set of FUs to execute the operations described in the CDFG, given the time constraints on these operations.

When the specified global time constraint is longer than the one used to obtain the initial scheduled CDFG, the module will perform a simple form of scheduling to make use of the extra control-steps and reduce FU costs.

The function and intent of this module is closely related to that of the BUD module [22] in the recent BUD-DAA system. However, the implemented prototype's capabilities fall quite short of those of the latter. Consisting of approximately two thousand lines of code, it is the least complex of the three modules that make up the HAL system — the InHAL and ExHAL modules required over ten thousand lines of code.

The reasons for putting less emphasis on this task were fourfold:

1. The allocation task is well suited to user interaction and/or designer suggestion, since the error-prone portions of the process — such as binding and interconnection — are still performed by the program.
2. An experienced designer usually has a good insight on FU tradeoffs. This observation has also been reported [47] by the IMEC synthesis group in Leuven, who implemented the Cathedral-II DSP synthesis program [48].
3. Designer insight is more intuitive than algorithmic,¹ and is therefore difficult to represent efficiently with a program.

¹ A rule-based paradigm was chosen for the implementation of this module to help counter this difficulty.

4. A good allocation tool would benefit from some form of physical locality information. The implementation of a preliminary floorplanner, such as the one used in the BUD system, was clearly beyond the scope of this thesis.

To deal with these considerations, the MidHAL module was designed to allow the allocation task to be performed completely or partially by the user. The system will verify that the allocated hardware is sufficient and will add FUs as necessary. In this way the user can invoke the scheduling/allocation loop repeatedly with different FU selections and quickly explore the design-space in an interactive fashion. We will see how the simple form of scheduling implemented in this module can be used to speed this exploration process.

9.2 RULE INVOCATION

The selection process uses a rule-based expert system. The knowledge base consists of a set of heuristic FU selection rules that are invoked in an iterative fashion. Each invocation allocates a specific FU, based on the current state of the *operation trace*; a simplified representation of the CDFG that does not include precedence constraints. This representation specifies, for each c-step, the number of operations of each type that are active.

When a functional unit (FU) is allocated to execute operations of a certain type, the operation trace is reduced by removing one operation of that type in each c-step. The rule set is then re-invoked with the new operation trace. This is repeated until all operations have been removed.

The termination rule and two simple allocation rules are given in Figure 27, Figure 28 and Figure 29.

```
IF      All c-steps of Operation Trace are empty
THEN
    End of processing.
```

Figure 27. Rule 0

```
IF      All operations in the trace are similar.
AND     Enough time to do operations serially
        (no of operations  $\leq$  no of c-steps)
THEN
    Find single FU.
    Create an instance of FU.
    Add to list of allocated FUs.
    Remove from the operation trace the operations
    performed by the FU.
```

Figure 28. Rule 1

```

IF      All operations in the trace are similar.
AND     Not enough time to do operations serially
        (no of operations > no of c-steps).

THEN
    Find multiple FUs.
    Create instances of FUs.
    Add them to list of allocated FUs.
    Remove from the operation trace the operations
    performed by the FUs.

```

Figure 29. Rule 2

We can now illustrate the rule selection process by using the simple examples of Figure 30 and Figure 31. As depicted in Figure 30, there are five main elements to distinguish.

At the top of each figure is the time constraint (in c-steps). Below that, on the far left, is the CDFG's initial operation trace. The rule that is satisfied after examination of that trace is given by the indicator above the double arrows. These arrows point to the next state of the operation trace after the current allocation. The list of FUs currently allocated is given below the arrows. An ALU is represented using the '@' symbol.

The simple subtraction shown (under the timing constraint) calculates the minimum number of c-steps required to execute the operations using a single FU for each type of operation. As we will demonstrate later, this value is

equal to the total number of operations in the trace minus one for each pair of operations that may be performed concurrently. When there are no more operations in the operation trace then the allocation is terminated.

At the far right of the figure, the implicit schedule is shown to indicate that the time constraint can be met with the chosen functional units. Operations performed by an ALU are enclosed in brackets (as in Figure 33 on page 144).

Finally, the cost value (given on the bottom left-hand side of the figure) represents the total FU area using the costs given on the last page of "Appendix A. Cost Functions and Structural Unit Costs".

No_csteps = 5 -[user-defined time constraint]

(4-0)-[min no of c-steps needed with one FU
for each type of operation]

Original operation trace		Modified operation trace		Final scheduling (implicit)
(+++)	Rule1	()		C-step 1- +
(+)	--	()		C-step 2- +
				C-step 3- +
				C-step 4- +

[1+] -[current FUs allotted, i.e. one adder]

Cost = 15 -[final total FU area cost]

Figure 30. Allocation I

No_csteps = 3

(4-0)

(+++)
(+)

Rule2
--

()
()

1- ++
2- +
3- +

[2+]

Cost = 30

Figure 31. Allocation II

This rule invocation process is repeated till all operations are allocated to a set of functional units. Note that the final binding of specific FUs to operations is not done here. This binding is performed in the next module, while taking functional and structural locality considerations into account.

Upon completion, the MidHAL module returns the following information:

- The type of FUs allocated
- The number of FUs of each type
- Their propagation delay
- Their area cost (including mux and interconnect area estimate)

The mux and interconnect area cost is evaluated by partitioning the CDFG and doing a preliminary binding of operations to FUs, storage to registers, and data transfers to interconnection units. The first four subtasks of the ExHAL

module (described later in "Chapter 11. ExHAL: Data Path Synthesis") are used to perform these bindings.

From these we can calculate an upper bound on the number of interconnection units required. As preliminary floorplanning is not performed in the HAL system, the associated interconnect area is estimated using the cost function [49] that is reprinted in "Appendix A. Cost Functions and Structural Unit Costs".

If the time constraint given is longer than the number of c-steps of the simplified CDFG (the operation trace) then the MidHAL module will perform an implicit scheduling of operations. This is done by implicit insertion of c-steps in the CDFG.² This allows for more flexibility in the way the InHAL/MidHAL iteration loop can be utilized.

For example, the InHAL module might be invoked originally using a minimal time constraint (e.g. four c-steps for the differential equation example used in previous chapters). Then in the next phase, the MidHAL module could be invoked using a non-minimal time constraint (e.g. six c-steps). Finally, the InHAL module would be reinvoked using

² This occurs in Figure 30 on page 139 where the first c-step of the original CDFG (containing three add operations) is expanded into three c-steps so that only one adder is required.

the six c-step time constraint and the allocation estimate to improve' the schedule.

Alternatively, all phases could be invoked with the same time constraint. In this case, the MidHAL module will not perform any implicit scheduling. The advantage of this approach lies in its simplicity. The disadvantage is in the fact that applying a 'loose' time constraint for the first scheduling lengthens the time frames of all operations and increases the computing time.

9.3 SAMPLE RULES AND ALLOCATIONS

In this section, we have chosen a subset of more explicit rules to further demonstrate the allocation process. In this set of rules, we need to distinguish two main classes of operations:

- Operations that can be assigned to ALUs or *ALU operations* — additions and subtractions are good examples of these.
- Operations that cannot; *special operations* — multiply and divide operations fall into this class.

Whenever an ALU operation and a special operation are present in the same c-step of the trace, it is assumed that they

' Allocation information is incorporated into the scheduling process using the techniques described later in "Chapter 10. Scheduling and Allocation by Stepwise Refinement".

will be performed concurrently since they cannot be assigned to the same FU.

In Figure 33 on page 144, we present a more detailed example with a global time constraint of 15 c-steps. It is assumed here that the InHAL module (i.e. the scheduler) was invoked previously with a time constraint of six c-steps.

The minimum number of c-steps required for execution using a single FU for each type of operation is equal to eleven. This value is calculated as follows. There are sixteen operations initially scheduled into six c-steps. In five of these c-steps, there are at least two different operations, so that these may be performed concurrently.

IF	Operations are not similar.	[for Figure 33]
AND	Enough time with all dissimilar operations scheduled concurrently; (min no of c-steps $\leq$ No_csteps)	- YES
AND	No of dissimilar ALU operations is less or equal to (No_csteps - min. no of c-steps)	(11 $\leq$ 15) - YES
AND	Cost of ALU is less than total cost of single function cells.	$3 \leq (15-11)$ - YES
AND	Extra ALU interconnect costs do not offset cell area savings.	(30 < 40) - YES
THEN	Find the best ALU to perform operations. Create an instance of the ALU. Add to list of allocated FUs. Remove from the free operation list the operations performed by the ALU. Re-invoke rule set with modified operation trace.	- YES

Figure 32. Rule 6

No_csteps = 15

(x++)	(16-5)	(x)	(2-0)	()	1-	x (+)
(++)		()		()	2-	(+)
(<<-)	Rule6	()	Rule1	()	3-	(+)
(++-)	--	()	--	()	4-	(+)
(+--)		()		()	5-	(<)
(x+)		(x)		()	6-	(<)
	[1@]		[1@,1x]		7-	(-)
					8-	(+)
					9-	(+)
					10-	(-)
					11-	(+)
					12-	(-)
					13-	(-)
					14-	x (+)

Cost = 80

Figure 33. Allocation III

At this point, the first two clauses of both Rule 6 (Figure 32) and Rule 8 (Figure 34 on page 146) apply. The third condition of Rule 6 evaluates true if the number of c-steps with dissimilar *ALU operations* is less or equal to four (15-11). In this case, there are only three c-steps with dissimilar ALU operations — these are the third, fourth and fifth c-steps of the trace. Therefore, assuming the next two conditions, Rule 6 applies and an ALU is allocated to perform the add, subtract and compare operations. The ruleset is reinvoked and Rule 1 applies. A single multiplier is therefore added to the list of allocated FUs.

Rule 8 of Figure 34 is similar to Rule 6, except that it will allocate single function units instead of an ALU. This occurs when the operations performed by the ALU cannot be scheduled serially within the time constraint.

* Rule 8 also applies; however, Rule 6 is encountered first in the rule search and therefore is the one evaluated.

```

IF
    Operations are not similar.
AND
    Enough time with all dissimilar operations
    scheduled concurrently, i.e.
    (min no of c-steps  $\leq$  No_csteps)
AND
    No of dissimilar ALU operations is greater
    or equal to (No_csteps - min. no of c-steps)
THEN
    Find best single function FU for all operations.
    Create instances of FUs.
    Add to list of allocated FUs.
    Remove from the operation trace the operations
    performed by the FUs.

```

Figure 34. Rule 8

Note that rules 1, 2, and 8 are *terminal rules* — i.e. rules which cause all operations in the trace to be assigned to FUs.

Rule 9 of Figure 35 allocates a single FU and removes from the trace the operations performed by that FU only. Therefore, it is considered a *non-terminal rule*. This was also the case for Rule 6 which removed only the operations performed by the allocated ALU.


```

IF      Operations are not similar.
AND     Not enough time even with all dissimilar
        operations scheduled concurrently, i.e.
        (min no of c-steps > No_csteps)

THEN
    Find the operations with the highest concurrency
        within a c-step.
    If there is a tie, find the most used operations.
    If there is a tie, find the operation that is the
        cheapest to realize.
    Find a single function FU to realize operation.
    Create an instance of the FU.
    Remove from the operation trace
        the operations performed by the FU.
    Re-invoke rule set with modified operation trace.

```

Figure 35. Rule 9

Figure 36 and Figure 37 depict the allocation process for the previous example with tighter timing constraints (eight and seven c-steps respectively). These examples illustrate the use of rules 6, 8 and 9. We see that the non-terminal rule 9 is applied repeatedly until one of the terminal rules apply. When this happens, all operations are allocated to FUs and the process is terminated.

No_csteps = 8

(x++)	(16-5)	(x+)	(11-3)	()	1-	x + +
(++)		(+)		()	2-	+ +
(<<-)	Rule9	(<<-)	Rule8	()	3-	< -
(++-)	--	(+-)	--	()	4-	<
(+--)		(--)		()	5-	+ + -
(x+)		(x)		()	6-	+ -
	[1+]		[2+,1x,1<,1-]		7-	-
					8-	x +

Cost = 105

Figure 36. Allocation IV

No_csteps = 7

(x++)	(16-5)	(x+)	(11-3)	(x+)	(8-1)	1-	x + (+)
(++)		(+)		(+)		2-	+ (+)
(<<-)	Rule9	(<<-)	Rule9	(<<)	Rule6	3-	(<)-
(++-)	--	(+-)	--	(+)	--	4-	(<)
(+--)		(--)		(-)		5-	+ (+)-
(x+)		(x)		(x)		6-	+ (-)-
	[1+]		[1+,1-]		[1+,1-,1@,1x]	7-	x +

Cost = 110

Figure 37. Allocation V

9.4 SUMMARY OF ALLOCATIONS

The summary of all the allocations for the operation trace depicted in Figure 37 is given in the table of Figure 38 below.

COMPONENT	NUMBER OF C-STEPS									
	NUMBER OF COMPONENTS ALLOCATED									
	6	7	8	9	10	11	12	13	14	15
ALU(@)	(@)	(@)			(@)				(@)	(@)
comparator(<)	(<)		(<)	(<)		(<)	(<)	(<)		
subtractor(-)	(-)	(-)	(-)	(-)		(-)	(-)	(-)		
adder(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)		
multiplier(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)
FU Cost	120	110	105	105	95	90	90	90	80	80

Figure 38. Functional unit allocation for all time constraints

It is interesting to note the important variations of the set of allocated FUs even for small changes in the timing constraint. An iterative system that built upon a partial data path would not be likely to find all these intermediate transformations when attempting speed-area tradeoffs. This is because only local transformations are applied. Using a global analysis approach such as the one presented here, it is easier to determine the optimal allocations for each constraint.

The table in Figure 39 illustrates the allocation of components for the differential equation example using the 4

c-step ASAP schedule that was given earlier in Figure 11 on page 73 of "Chapter 6. InHAL: Force-Directed Scheduling".

COMPONENT	NUMBER OF C-STEPS									
	NUMBER OF COMPONENTS ALLOCATED									
	4	5	6	7	8	9	10	11	12	13
ALU(@)				(@)		(@)	(@)	(@)	(@)	(@)
comparator(<)	(<)	(<)	(<)		(<)					
subtractor(-)	(-)	(-)	(-)		(-)					
adder(+)	(+)	(+)	(+)		(+)					
multiplier (*)	(*)									
	(*)	(*)								
	(*)	(*)	(*)	(*)						
	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)
FU Cost	240	190	140	130	90	80	80	80	80	80

Figure 39. Functional unit allocation for all time constraints: Differential equation example from Figure 11 (using ASAP schedule).

Once again, the FU allocation is strongly affected by minor changes of the timing constraint — the allocations for the 7, 8 and 9 c-step constraints show this clearly.

The next table (Figure 40) illustrates the results of allocations for the schedule obtained using the force-directed algorithm, with a four c-step timing constraint (refer back to Figure 17 on page 91).

COMPONENT	NUMBER OF C-STEPS									
	NUMBER OF COMPONENTS ALLOCATED									
	4	5	6	7	8	9	10	11	12	13
ALU(@)		(@)	(@)	(@)	(@)	(@)	(@)	(@)	(@)	(@)
comparator(<)	(<)									
subtractor(-)	(-)									
adder(+)	(+)									
multiplier(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)
FU Cost	140	130	130	80	80	80	80	80	80	80

Figure 40. Functional unit allocation for all time constraints: Example of Figure 11 on page 73 (force-directed schedule).

The cost curves associated with the last two tables is shown in Figure 41 on page 152. The light solid line represents the cost curve that was obtained from the MidHAL module using the four c-step ASAP schedule. The heavy solid line indicates the cost curve obtained using the four c-step optimal schedule derived by the force-directed algorithm.

Functional Unit Area (with Interconnect estimate)

Figure 41. Cost curve for differential equation example: cost values are given for all possible timing constraints.

The small hatched section represents the allocation cost for a seven c-step constraint, after the second phase of the scheduling/allocation process. This particular result will be discussed in the next chapter which describes the four phase stepwise refinement approach in detail.

As we mentioned earlier, the MidHAL module also performs scheduling by inserting implicit c-steps to defer operations and minimize the number of FUs required. Using the initial four c-step optimal schedule, it derived optimal schedules

(and allocations) for the remaining constraints. However, using the sub-optimal ASAP schedule, the allocations are substantially more expensive. They are equal only when MidHAL manages to serialize most of the operations — i.e. for constraints of 9 c-steps and above.

From our experience, we have found that the quality of the allocation performed depends directly on the initial schedule used, as shown in the example above. The quality of the schedules obtained from the MidHAL module tend to degrade when the time constraint is much higher than the constraint used for the original schedule.

Quick design-space exploration: The implicit^o scheduling performed by the MidHAL module allows for much flexibility in the way that the scheduling/allocation loop is invoked. It also suggests a quick design-space exploration technique that can be summarized using the following example.

Let us assume we are interested in exploring the design-space zone bounded by time constraints of five and eight c-steps. We start by invoking the InHAL module to establish an initial schedule for a time constraint equal to the lower bound of the constraint zone (in this case, five c-steps). The reduced time frames of all operations ensures that this schedule will be obtained rapidly.

We then invoke the MidHAL module to perform allocations for all constraints from five to eight c-steps, using the original five c-step schedule. The implicit scheduling performed by the MidHAL module allows to quickly generate allocations for these longer time constraints. The allocation results will be the most precise for time constraints that are closest to that of the initial schedule.

CHAPTER 10. SCHEDULING AND ALLOCATION BY STEPWISE REFINEMENT

To be effective, a synthesis system must attempt to reconcile two conflicting goals:

- The optimal scheduling of operations without explicit foreknowledge of the FU allocation.
- The optimal allocation of FUs without exact information on the concurrency of operations or their propagation delays.

In the HAL system, this is accomplished by stepwise refinement of the operation scheduling and FU allocation as depicted in Figure 42.

10.1 SCHEDULING/ALLOCATION PHASES

The approach can be divided into four main phases:

Phase 1 Default Allocation: Allocation of default single-function FUS to perform each type of operation. The fastest FUS available in the technology dependent library are temporarily assumed. The default FU for each operation type is stored in the class describing that operation.

At this point there is no attempt to assign different operation types to multi-function FUS (e.g. ALUs). This should only be attempted when the concurrency of these operations has been estimated (in the next phase).

Phase 2 Preliminary Schedule: The *InHAL* module is invoked to balance the distribution of similar operations. The force-directed scheduling algorithm described earlier is used for this purpose. This is done while taking into account the cost and speed of the FUS assigned to them in the first phase.

Phase 3 Refined Allocation: Invocation of the rule-based *MidHAL* module to perform allocation of single and multi-function FUS. This allocation may actually substitute slower FUS or combine different operations into groups that can be performed by an ALU.

This can be done as it now has a better estimate of:

- The timing constraints on the individual operations (they may have been relaxed on some operations).
- The overall concurrency of operations of different classes.

As depicted in Figure 42, the mux and interconnect area costs are also evaluated here.

Phase 4 Final Schedule: In this phase the *InHAL* module is invoked a second time. The goal now is to balance the distribution of operations using similar resources, since those resources have been allocated more precisely by the *MidHAL* module in the previous phase.

10.2 USE OF ALLOCATION INFORMATION FOR SCHEDULING

The interdependence of scheduling and allocation make it necessary to transfer allocation information into the scheduling process. Four types of information can be taken into account:

1. Types of FUs allocated
2. Number of FUs allocated
3. FU speed
4. FU and interconnect area costs

10.2.1 Types of functional units allocated:

For each type of FU allocated, one single distribution graph (DG) is created. Multi-function FUs such as ALUs will be assigned to the multiple operation DGs described in "Chapter 7. Refined Scheduling Algorithm".

10.2.2 Number of functional units allocated:

In most cases, the MidHAL module allocates the minimum number of FUs for every type of operation. This number is equal to the maximum number of concurrent operations they perform. But there are also cases where a larger number of FUs are allocated.

This may happen when the cost saving due to FU sharing is offset by the cost of the additional interconnect required. For example, one bit operations are assigned to separate FUs, as the additional interconnect cost usually exceeds the cost of a duplicate FU. It may also happen when the designer manually enforces the allocation of a non-minimal number of FUs. In either case, we must include this information so that the forces applied by the assigned DG are reduced, as more than enough FUs are likely to be available.

More specifically, for each FU type (i.e. for each DG) and each c-step we calculate the unused FU capacity. This is given as the difference between the number of allocated FUs and the current number of operations scheduled in c-step i.

$$\text{Unused Capacity}(i) = \text{No. FUs} - \text{No. Opns}(i) \quad (1a)$$

In the absence of allocation information, the number of required FUs of any type may be estimated by taking the maximum (over all c-steps) of the number of concurrent operations of each type. This value is initially set to one, and will increase during the scheduling process as the concurrency of that type of operations increases. In other

words, if there is no lower bound specified on the number of FUs, then the No. FUs variable of equation (1a) is given by:

$$\text{No. FUs} = \max[\text{Scheduled_Opns}(i)] \quad (1b)$$

where $\max[]$ = maximum over all c-steps
(i.e. $1 \leq i \leq \text{highest c-step}$).

and $\text{Scheduled_Opns}(i)$: Number of operations currently
scheduled in c-step i .

This information can be used to establish a new *biased* DG value (i.e. a new spring constant), which is given by the old value less the unused FU capacity.

$$\text{Biased DG}(i) = \text{DG}(i) - \text{Unused Capacity}(i) \quad (2a)$$

When the number of scheduled operations in c-step i reaches the number of FUs allocated, $\text{DG}(i)$ will take on its original value.

To avoid large negative forces in c-steps with low distributions, it is preferable to set a lower bound on negative values of the Biased DGs. The best results were obtained using a lower bound of -1.

$$\text{Biased DG}(i) = \max[-1, \text{DG}(i) - \text{Unused Capacity}(i)] \quad (2b)$$

10.2.3 Functional unit speed:

This information is integrated in a straightforward fashion by assigning the propagation delay of each FU to the operations it performs. The determination of time frames and subsequent scheduling will thus be directly affected by these new values.

10.2.4 Functional unit and interconnect area costs:

As we mentioned in "Chapter 7. Refined Scheduling Algorithm", each DG element is multiplied by a cost factor. For the initial scheduling in phase 2, this cost factor is simply the FU area:

$$\text{Cost Factor} = \text{FU Area} \quad (3)$$

For the final scheduling in phase 4, we can use the estimate of the bus, mux and interconnect area derived in phase 3 to calculate a weighted cost factor:

$$\text{Cost Factor} = \text{FU Area} * \text{Interconnect Factor} \quad (4)$$

The interconnect factor represents the relative importance of the FU area with respect to the combined FU, bus, mux and interconnect area:

$$\text{Interconnect Factor} = \frac{\text{FU Area}}{\text{Total Area}}, \quad \text{where} \quad (5)$$

$$\text{Total Area} = (\text{FU} + \text{Bus} + \text{Mux} + \text{Interconnect}) \text{ Area} \quad (6)$$

For high-cost FUs such as multipliers, the interconnect factor tends towards one. Alternately, for simple FUs (e.g. logic gates) the associated interconnect area can be much larger than the FU area in which case the interconnect factor tends towards zero.

In general, this will result in an increased number of low cost FUs, but the associated area increase will be offset by reduced interconnect costs. On the other hand, sharing of high-cost FUs will still be encouraged.

10.3 SAMPLE SCHEDULES AND ALLOCATIONS

In Figure 41 on page 152 of "Chapter 9. MidHAL: Allocation of Functional Units", we presented a comparison of allocation costs for our differential equation example using two different schedules. A third result was also given in this figure, as represented by the small hatched section.

This curve segment represents the allocation cost for a seven c-step constraint, using the schedule obtained in the second phase of the scheduling/allocation process, as depicted in Figure 42 on page 155. Using this preliminary schedule, the MidHAL module allocated one multiplier, one adder and one ALU. The adder was required as an addition and subtraction were scheduled in the same c-step, and therefore could not share the ALU. This occurred simply because the scheduler had no way of knowing an ALU was likely to be allocated.

Following this allocation (i.e. the third phase), the INHAL module improved on the schedule in the fourth phase by assigning all ALU operations to different c-steps. This occurred because a multi-function DG was created for all the operations that could be performed by the ALU. This final schedule was then used by the MidHAL module which allocated an ALU and multiplier, thus reducing the total FU cost.

The next set of results presented in Figure 43 are also for the differential equation example. Here the multiplications are given a more realistic propagation delay of two c-steps. The critical path is now six c-steps long. The resulting FU allocations are tabulated for different time constraints. The refinements of the scheduling algorithm described above were applied here. The CPU execution times given are for a XEROX 1108 Lisp machine.

COMPONENT	(* cost > ALU cost)				(ALU cost > * cost)			
No Cycles	6	7	8-12	13	6'	7'	8'-12'	13'
multiplier (*)	(*) (*) (*)	(*) (*)	(*) (*)	(*)	(*) (*) (*) (*)	(*) (*) (*)	(*) (*)	(*)
ALU (@)	(@) (@)	(@) (@)	(@)	(@)	(@) (@)	(@) (@)	(@)	(@)
CPU (sec)	15	35	65-120	180	15	30	65-125	175

Figure 43. Functional unit allocation for all time constraints: Example of Figure 11.

The results in the first four columns illustrate the realistic case where the cost of the multiplier is higher than the cost of the ALU. The results in the next four columns represent an illustrative example where the cost factor of the ALU is set artificially higher than the cost of the multiplier. The change in forces resulted in the decrease of the concurrency of the ALU operations at the expense of the multiplication operations. This is exactly the trade-off that we are hoping for.

The results obtained in column 6' and 7' are identical to the ones that would be obtained if the MidHAL module (or the user) had allocated four and three multipliers respectively. This is due to the associated reduction of the mul-

tiplication forces dictated by equation (2) of section "Use of Allocation Information for Scheduling" on page 157, such that the concurrency of the ALU operations is reduced to one. Once again, this trade-off is the one that should follow logically.

CHAPTER 11. EXHAL: DATA PATH SYNTHESIS

In this third and final stage of the overall synthesis task, the final data path is generated. This task is realized by the *ExHAL* module which makes use of the schedule and allocated functional units obtained by the stepwise refinement process just described.

The *ExHAL* module binds the CDFG operations to specific functional units and creates registers and buses to store and transfer data between FUs. It also performs data path optimization by combining registers, merging buses and aligning FU operands. An overview of the module was given in "Chapter 2. System Overview".

When the module is invoked, it is assumed that all operations have been assigned to specific control-steps by the *InHAL* module. Moreover, it is assumed that a set of functional units has been allocated by the *MidHAL* module. Any existing legal bindings¹ are preserved and form the basis for the initial design. This allows experienced designers to specify a subset of bindings that defines a design starting point. When used in this way the system can be consid-

¹ Illegal bindings occur when module utilization conflicts arise. This will also occur when insufficient hardware has been allocated. In these cases an error will be reported.

ered as a behavioral synthesis tool as defined in "Chapter 1. Introduction".

The main steps of the data path synthesis task are as follows:

1. Operation to FU binding
2. Creation and Insertion of Storage Operations
3. Storage to Register Binding
4. Data-Transfer to Interconnect Binding
5. Alignment of FU Operands (repeated after step 6)
6. Register Merging
7. Multiplexer Merging

Before proceeding with the detailed description of these tasks, we give a complete description in Figure 44 of the graph symbols used throughout the chapter.

Graph Symbols

a) Behavioral Symbols

b) Structural Symbols

Figure 44. Symbol description for graph objects.

11.1 OPERATION TO FU BINDING

The first main synthesis task consists of binding arithmetic and logic operations to the specific functional units allocated by the MidHAL module. The system attempts to minimize interconnect by using functional locality information (i.e. information derived from the data flow graph) to guide the binding. The implicit underlying data path structure is also used.

The algorithm described below makes use of a *greedy* algorithm where the CSG (control-step graph) is traversed from top to bottom and functional units are assigned when available. Bindings from previous c-steps are examined when two (or more) FUs can be bound to an operation.

The CSG given in Figure 45 illustrates the end result of the assignment for the differential equation example using a time constraint of four c-steps. Each 'flattened' circle represents an arithmetic or logic operation. The upper half identifies the operation type and index, while the lower half shows the bound FU (the final binding is shown here). The small circles at the c-step boundaries represent storage operations, the small squares represent a data transfer. Both of these will be discussed shortly. The intermediate sized circles represent constant values.

Figure 45. Control-Step Graph (CSG) for differential equation example.

The binding of operations to functional units for this example is summarized in the *binding table* of Figure 46 which contains a list of the operation to FU assignments.

Functional Unit	Bound operation (c-step)
fu1 (multiplier)	*8 (1) , *11(2), *15(3)
fu2 (multiplier)	*7 (1) , *10(2), *14(3)
fu3 (adder)	+9 (1) , +17(4)
fu4 (subtractor)	-13(3) , -16(4)
fu5 (comparator)	>12(2)

Figure 46. Binding table for differential equation example.

This table was obtained using a two pass binding algorithm:

1. Binding of operations that can only be assigned to specific FUs.
2. Binding of remaining operations using a *greedy* algorithm.

In the first pass, fu3, fu4 and fu5 are directly bound to the add, subtract and compare operations respectively. This yields the last three lines of the table.

In the second pass, the CSG is traversed from top to bottom and the remaining FUs are assigned using existing binding information. The idea is to bind operations to FUs in such away as to minimize the number of different source and destination objects linked to the FU. By doing so,

interconnect costs tend to be reduced since transfers between identical source-destination pairs are usually assigned to the same path.

For example, the data transfer between operations *8 and *10 of Figure 45 implies the creation of a path¹ from fu1 to fu2. Similarly, the data transfer between *11 and *14 also implies a path from fu1 to fu2.

Using the register and bus merging schemes presented later, we will see that the same path will be assigned to this transfer to reduce the interconnect costs. A quick glance at the final bound CSG of Figure 62 on page 197 confirms this; the ds-R18-m29 path (direct connection to register R18 to mux m29) is used in both instances. We also see that this is the case for the fu2 to fu4 paths defined by data transfers *10 to -13, and *14 to -16. In this case the ds-R20-df path is used for both transfers.

We may now examine how this binding was obtained. We start by creating a *transfer table* which contains a list of the current source operands and output destinations for each FU. In the first c-step, the transfer table is empty so each multiplication can be bound to either multiplier. Using

¹ This path may be realized using single connections, fanout connections, muxes or buses. But since the data transfer crosses a c-step boundary, a register must also form part of the path.

the randomly selected binding of Figure 45, we obtain the following transfer table:

Functional Unit	Sources	Destinations
fu1	3, fu3	undefined
fu2	fu4, dx	undefined

Figure 47. Transfer table after c-step 1

The sources fu3 and fu4 are obtained by tracing variables x and y back to the end of the loop. As the operations *10 and *11 in c-step 2 are still unbound, the destinations are temporarily undefined.

This table can be used to determine the next operation-to-FU binding that will minimize the number of distinct sources and destinations. It is preferable to bind *11 to fu1 in c-step 2 as the sources '3' and fu3 are already linked to fu1. The transfer table is now:

We do not need to distinguish right and left operands since the operands of a multiplier are reversible. In "Alignment of FU operands" we will describe how operands are aligned.

Functional Unit	Sources	Destinations
fu1	3, fu3	fu2
fu2	fu4, dx, fu2, fu1	fu2, fu4

Figure 48. Transfer table after c-step 2

In c-step 3, the selection of one of two possible bindings is not as obvious. Let us explore both alternatives by displaying the tables that would result from each possible binding.

Functional Unit	Sources	Destinations
fu1	3, fu3, dx, fu1	fu2, fu1, fu4
fu2	fu4, dx, fu2, fu1	fu2, fu4, fu3

Figure 49. Transfer table after c-step 3 (first tentative binding) :

fu1 is bound to *14
fu2 is bound to *15

For this binding, the total number of sources and destinations is equal to fourteen (4+4+3+3).

Functional Unit	Sources	Destinations
fu1	3, fu3, fu4, dx	fu2, fu3
fu2	fu4, dx, fu2, fu1	fu2, fu4

Figure 50. Transfer table after c-step 3 (second tentative binding) :

fu2 is bound to *14
fu1 is bound to *15

Here, the total number of sources and destinations is reduced to twelve ($4+4+2+2$), and this is the binding chosen by the algorithm. This example demonstrates the importance of considering both sources and destinations since both bindings imply identical numbers of distinct sources.

Mutually exclusive operations: Conditional operations (e.g. if-then-else and case statements) specify sets of mutually exclusive operations. When these operations are of the same type and scheduled into the same c-step, they are generally assigned to the same FU. An exception to this rule occurs when another FU of the same type is free, and binding the operation to this one reduces the total number of sources and destinations.

11.2 CREATION AND INSERTION OF STORAGE OPERATIONS

The second step in the synthesis process is the determination and insertion of storage operations. Storage operations are inserted between data-dependent arithmetic/logic operations when the value transferred between them traverses a c-step boundary.

Storage operations are related to registers in the same way that arithmetic and logic operations are related to functional units; in both cases functional operations are bound to structural objects, but not necessarily on a one-to-one basis.

There are two types of storage operations: *regular storage* and *local storage*. Regular storage operations are based on variable lifetime analyses. They are similar to the ones used for scheduling purposes in "Chapter 7. Refined Scheduling Algorithm". Local storage operations are used only in particular circumstances and will be described in section "Use of Local Storage" on page 198 at the end of this chapter.

The storage operations for our example are depicted as small circles on the c-step boundary in Figure 45. A darkened circle indicates the beginning of the storage operation's lifetime — it also indicates the time at which data

is clocked into the bound register. The first three storage operations on the upper boundary of c-step 1 are actually the same as the three on the lower boundary of c-step 4, since the loop beginning and end must coincide.

The minimum number of registers required is given by the maximum number of circles on a given c-step boundary. In this case, at least five registers are required. We must however bear in mind that the use of a larger number of registers may sometimes be preferable if the interconnect area saving offsets the added register area.

11.3 STORAGE TO REGISTER BINDING

The next main step is the creation of registers and preliminary binding with the storage operations.* Once again, *functional locality* information — i.e. information derived from the data flow graph — is used to guide the binding. More specifically, the storage operations are first partitioned into groups. Each group is formed from the storage operations with identical source objects and destination objects. All disjoint members of this group are assigned a

* These operations can be either regular or local. For simplicity, we will use regular storage only for the demonstration of our differential equation example.

single register. This ensures that minimum interconnect will be required to connect the newly created register.

For example, in the CSG of Figure 51 on page 178 there are only two sets of storage operations that fulfill the conditions stated above. The members of the first set are s25 and s28 that share the common source fu1 and common destination fu2. They are both bound to register R18. Likewise, s27 and s30 share the common source fu2 and common destination fu4 and are bound to register R20.

The remaining storage operations are bound to single registers, as shown in the figure. At this stage, there are presently eight registers (R18 to R25). These may be merged later on in the synthesis process. This is only attempted when the preliminary circuit graph (CG) is established. In this way, the merging process can make use of structural information obtained from this circuit.

11.4 DATA-TRANSFER TO INTERCONNECT BINDING

In this fourth step the partially bound CSG is mapped to a distributed structure data path. Multiplexers are created as needed (to direct operands to processors) and intercon-

* These are storage operations that have non-overlapping usage times or that are mutually exclusive.

Figure 51. Preliminary storage to register binding

nection is performed. The result of this step is the fully bound CSG which constitutes a preliminary *data path* or DP.

The generation of a small portion of the data path is shown in Figure 52. All the operations assigned to the multiplier fu2 are examined. One multiplexer input is created for each operation. Each one of these is associated with a c-step number that indicates the data transfer times. The input is connected to the hardware element that is assigned to the operation that provides the operands. This may be a register, a constant or another FU. In this example the multiplier is assigned to three operations and therefore requires multiplexers with as many inputs.

The process is repeated for all functional units, registers and ports. In many cases, single input multiplexers are created. These could be compressed into a single direct connection, but since they could be merged with a bus in the bus merging step presented later, they are temporarily preserved.

Figure 52. Partial Data Path

11.5 ALIGNMENT OF FU OPERANDS

In this step, the operands of commutative operations (e.g. additions, multiplications) are selectively interchanged to minimize the number of multiplexer inputs and the associated interconnection costs. This task is often referred to as *operand alignment*.

The approach used here is a breadth-first search of the possible input interchanges. The lowest cost interchange is selected and the process is repeated until the cost becomes stable.

We can illustrate the method using the sample inputs given in the example of Figure 53. These could be the in-

puts for an ALU performing add, subtract and compare operations.

The first step is the derivation of the inputs of the mux for each c-step. Here, two 'n' input vectors are created, where 'n' represents the number of operations assigned to the FU. A third 'n' element vector is also created to indicate which operations have non-commutative operands (e.g. subtractions, compares, etc.). The vector representing the ALU inputs is shown in Figure 53. Numbers in square brackets represent input ports, those in round brackets, registers. Note that the second operation is a subtraction and cannot have its operands interchanged (this is shown using an 'X' in the third vector).

A breadth-first search of the possible input interchanges is performed. This is done by heuristically selecting all 'good' interchanges, evaluating their associated costs, and finally selecting the least expensive. This cost is given

by the number of mux inputs required after the interchange. In turn, the number of inputs is given by the number of different signals connected to the mux. This implies that inputs with similar operands will be combined. The cost associated with the example above (in Figure 53) is given by:

$$\text{Cost} = 4 + 4 = 8$$

A 'good' interchange is defined as one that will combine at least two similar inputs into the same mux. In this case only two need be considered; the interchange of (R4) and (R8) or that of (R8) and (R3). The cost associated with these possible interchanges is calculated. The one representing the lowest cost is selected. In this case, the interchange of (R8) and (R3) does not reduce costs. However, as we show below, the interchange of (R4) and (R8) reduces the cost by one. The inputs are now the following:

The cost associated with the example is now given by:

Figure 60. Final register assignments

$$\text{Cost} = 4 + 3 = 7$$

The process is repeated until the cost becomes stable. In this example the best solution was obtained immediately after the first iteration.

Following this, inputs with similar operands are combined and the control cycle numbers of the eliminated inputs are assigned to the combined input.

If we apply the algorithm described above to the partial datapath of Figure 52 on page 180, we obtain the one depicted in Figure 55.

Figure 55. Partial Data Path: after mux operand interchange and input combining.

11.6 REGISTER MERGING

In this important optimization step, compatible registers are selectively combined. This can be seen as a final binding of storage operations to registers, as the resulting registers will perform the union of the storage operations of the combined registers. The approach used here can be seen as a dual of the one used for preliminary storage binding above.

In this case, *structural weights* (i.e. information derived from the data path structure) are used to guide the binding. These weights can be determined from the preliminary FU, mux and interconnect bindings performed earlier.

The approach used here consists of applying regular clique partitioning to a subset of disjoint registers. The subset is determined by a structural weight threshold.

11.6.1 Weight calculation

The determination of *structural weights* is central to the register merging approach used here. These weights represent a form of interconnect affinity between design objects. They are based on a similar form of weights that is used in [49] for bus allocation.

Here, a structural weight is calculated for each pair of disjoint registers. There are six main weight values, four of which are depicted in Figure 56 on page 186.

<u>Weight</u>	<u>Meaning</u>
5	Both registers have same source and same destination. Combining the two will remove two interconnections.
4	One register feeds another. Combining the two will remove one interconnection.
3	Two registers have common sources.
2	Two registers have common destinations.
1	The source object of one register is identical to the destination object of the second one. Combining the two will require a mux and a fanout connection. However, the mux connection will be local.
0	There are no common objects in the sources and destinations of both registers. Combining the two will require a mux and a fanout connection.

Register Merging Weights

Figure 56. Structural weights for different types of register merges: A higher weight indicates a more favourable merge.

11.6.2 Weight-directed clique partitioning

Clique partitioning [46] can be described as an algorithm for the generation of the maximal independent sets of linked elements in a graph.

Let G be a graph consisting of a finite number of nodes and a set of undirected edges connecting pairs of nodes. A

non-empty collection C of nodes of G forms a complete graph if each node in C is connected to every other node of C . A complete graph C is said to be a clique with respect to G if C is not contained in any other complete graph of G .

The *clique-partitioning* problem is to partition the nodes in G into a minimum number of disjoint cliques. This means that each node appears in one and only one clique, and that some may be alone in their clique.

The relevance of the clique-partitioning problem to the register binding and merging process can be seen easily; each register is a node, undirected edges represent register disjointness. Two registers are disjoint in either of the following situations:

- Their usage times are not overlapping
- They perform mutually exclusive storage operations

Partitioning the graph into the minimum number of disjoint cliques immediately yields the minimum cost register merging. Unfortunately, the problem of clique partitioning has been proved to be *NP-complete*. Heuristics must therefore be used to reduce the problem complexity.

To illustrate the need for such heuristics, let us examine the clique partitioning problem for our differential equation example. The first step is to determine, for each

register defined in Figure 51 on page 178, the list of disjoint registers. This yields the register compatibility matrix shown in Figure 57.

	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25
R16	-	-	-	-	-	-	-	X	-	X
R17	-	-	-	X	-	-	-	X	X	X
R18	-	-	-	X	-	-	-	X	X	X
R19	-	X	X	-	-	X	-	X	-	X
R20	-	-	-	-	-	X	-	X	-	X
R21	-	-	-	X	X	-	-	X	X	X
R22	-	-	-	-	-	-	-	-	-	-
R23	X	X	X	X	X	X	-	-	X	-
R24	-	X	X	-	-	X	-	X	-	X
R25	X	X	X	X	X	X	-	-	X	-

Figure 57. Compatibility matrix of all registers: an X indicates that two registers are disjoint, and therefore potential candidates for merging.

An exhaustive evaluation of all cliques for even this simple example will generate thousands of possible cliques. Each one of these must then be evaluated for the lowest cost. While this guarantees the minimum cost register merge, the computing time is unacceptable. We must therefore reduce the problem complexity or use another method.

Left-edge algorithm: One possible algorithm that may be used is the left-edge algorithm. It is used in the USC REAL register allocation program [28] that is invoked after the MAHA [27] or Sehwa programs [44]. The left-edge algorithm has been proven optimal for track assignment in channel routing [50] and therefore guarantees the minimum number of

registers when the storage operations have continuous lifetimes. For this same reason, it cannot be used for successive register merging, as the register lifetimes become discontinuous.

Clique Partitioning: The Facet system [14] uses a clique partitioning approach to determine a near-minimum number of registers. It makes use of heuristics based on the clique graph structure to prune the clique graph, therefore reducing the number of possible cliques.

The main limitation of these two approaches however, is that the repercussions of a specific register merging on interconnect costs are ignored. In most cases, there are several register merging configurations that will yield the same final number of registers. The associated interconnect cost of each of these possible merges may vary considerably. Furthermore, the configuration with the lowest global cost is not necessarily the one with the minimum number of registers.

We have therefore opted for a register merging strategy where we can generate a reasonably sized subset of the best possible solutions. We may then evaluate interconnect costs for each solution and select the one with the lowest overall cost. This overall cost is the sum of register and inter-

connect costs. In this way, the interactions of register and interconnect are taken into account.

This strategy uses a stepwise refinement approach to perform regular clique partitioning on a reduced problem. Interconnect information derived from the data path is used to reduce the clique graph. This is done by considering only the compatible registers that have a structural affinity — as determined by the structural weights described earlier — that is above a certain threshold. This threshold is progressively lowered as the number of compatible register pairs decreases.

For our example (Figure 51 on page 178), the algorithm starts with an initial weight threshold equal to four and establishes the feasible combinations. This yields:

	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25
R16	-	-	-	-	-	-	-	X	-	-
R17	-	-	-	-	-	-	-	-	-	X
R18	-	-	-	-	-	-	-	-	-	-
R19	-	-	-	-	-	-	-	-	-	-
R20	-	-	-	-	-	X	-	-	-	-
R21	-	-	-	-	X	-	-	-	-	-
R22	-	-	-	-	-	-	-	-	-	-
R23	X	-	-	-	-	-	-	-	-	-
R24	-	-	-	-	-	-	-	-	-	X
R25	-	X	-	-	-	-	-	-	X	-

Figure 58. Compatibility matrix (weight threshold = 4)

The number of possible cliques is drastically reduced. All possible cliques are generated from the reduced matrix, and the one with the lowest interconnect cost is chosen. This interconnect cost is estimated by determining the number of distinct sources and destinations for each of the combined registers. As in the functional unit binding subtask, the algorithm favors cliques which minimize the total number of sources/destinations. For our example, the lowest cost clique is: [(R16-R23), (R17-R25), R18, R19, (R20-R21), R22, R24].

In the next iteration of the stepwise refinement process, the weight threshold is reduced to three. This yields the following matrix.

	R16 R23	R17 R25	R18	R19	R20 R21	R22	R24
R16,R23	-	-	-	-	-	-	-
R17,R25	-	-	-	-	-	-	X
R18	-	-	-	X	-	-	-
R19	-	-	X	-	-	-	-
R20,R21	-	-	-	-	-	-	-
R22	-	-	-	-	-	-	-
R24	-	X	-	-	-	-	-

Figure 59. Compatibility matrix (weight threshold = 3)

In some cases, this number may still be beyond reasonable computing limits. This can easily be predicted by inspection of the compatibility matrix. Whenever this occurs, we use simple graph-based techniques, similar to the ones presented in the Facet system, to further reduce the clique graph.

This matrix allows only one possible clique: [(R16-R23), (R17-R24-R25), (R18-R19), (R20-R21), R22]. Reapplying lower weights yields empty matrices. This is simply because the minimum number of registers — five in this case — has been attained. The final register binding is shown in Figure 60.

As the operands of FUs have changed as a result of the register merges, the FU operand alignment step described in the previous section is reinvoked here. An operand interchange is indicated by a double arrow under each operation affected, as depicted in the final bound CSG of our example in Figure 62 on page 197. Here the operands of operations *7 and +9 were interchanged.

11.6.3 Partitioned Register Merging

For very large examples, the number of possible cliques may still be too high, even when structural weights and graph reduction techniques are used to prune the search space. In these cases, we partition the registers into local groups.

Registers are partitioned into groups associated with the inputs or outputs of each FU, i.e. the registers of a group are structurally related to a specific FU.

This is done for two reasons:

- Members of a group tend to be naturally disjoint. This is because all of these registers provide data to the

Figure 60. Final register assignments

same FU. In turn, this FU can only consume one or two data items at a time, so the registers are often needed at different (disjoint) times.

- Members of a group will tend to be physically close. This is a natural consequence of any placement program that will attempt to minimize interconnect by placing related modules close to each other. Combining these registers will not incur too much extra interconnect.

Weight-directed clique partitioning is then used to find different possibilities of register combinations within each group. If the number of possible combinations is reduced substantially, a lower initial weight threshold (e.g. two) will be selected.

11.6.4 Incorporation of floorplanning information

Although the HAL system does not perform any floorplanning or any other form of physical partitioning, this type of information can be easily included in the register merging process.

One possible approach is to partition the initial data path into a two-dimensional tree similar to the one used in [38]. In this representation, each branch divides into four sub-branches which correspond to a two-dimensional physical partitioning of the data path elements. Using this representation, a secondary *physical locality weight* could be determined and added to the structural weight described

earlier. Two registers assigned to close partitions would be given a high weight and vice-versa.

These physical weights could also be used to prune the clique graph for large examples, and would therefore eliminate the need for the "Partitioned Register Merging" method just described.

11.7 MULTIPLEXER MERGING

The problem of merging muxes (multiple inputs, single output) into buses (with multiple inputs and outputs) is relatively close to the register merging problem. The main difference is that mux usage times are not continuous; a mux created in the method described earlier is assigned a series of disjoint active c-steps. Algorithms like the left-edge cannot be used for the same reason they couldn't be used to merge registers that had disjoint lifetimes.

A clique partitioning method can be used however. We have elected to use the same approach as for register merging. To limit complexity in this case, we use a threshold on the number of common inputs instead of the structural weights defined for registers. The use of physical locality weights described in the previous section could also be used here, if a physical partitioning was previously performed.

In Figure 61, we depict the *mux compatibility matrix* for the differential equation example. The minimum number of common inputs threshold is set to one here.

	M26	M27	M28	M29	M30	M31	M32	M33	M34	M35	M42	M46
M26	-	-	-	-	-	-	X	-	-	-	-	-
M27	-	-	-	-	X	-	-	-	-	-	-	-
M28	-	-	-	-	-	X	-	-	-	-	-	-
M29	-	-	-	-	-	-	-	-	-	-	-	-
M30	-	X	-	-	-	-	-	-	-	X	-	-
M31	-	-	X	-	-	-	-	-	-	-	-	-
M32	X	-	-	-	-	-	-	-	-	-	-	-
M33	-	-	-	-	-	-	-	-	-	-	-	-
M34	-	-	-	-	-	-	-	-	-	-	-	-
M35	-	-	-	-	X	-	-	-	-	-	-	-
M42	-	-	-	-	-	-	-	-	-	-	-	X
M46	-	-	-	-	-	-	-	-	-	-	X	-
M48	-	-	-	-	-	-	-	-	-	-	-	-
M49	-	-	-	-	-	-	-	-	-	-	-	-
M50	-	-	-	-	-	-	-	-	-	-	-	-

Figure 61. Compatibility matrix of all muxes: an X indicates that two muxes have disjoint utilization times, and are therefore potential candidates for merging.

Once again the number of possible merges is reasonable. Each one is attempted and the interconnect cost is evaluated. The best set of merged muxes for this example was: [(M28, M31) , (M26, M32) , (M27, M30)] and were assigned to buses [B53, B52, B54] respectively. All single input muxes (i.e. M33, M34, M35, M42, M46, M48, M49, M50) were transformed into direct connections. The remaining mux (M29) was unaffected.

The final bound CSG is depicted in Figure 62.

The datapath corresponding to this CSG is depicted in Figure 65 on page 206 of "Chapter 12. Design Examples". In that chapter, we will compare this result with the ones obtained from other systems.

11.8 CONTROL INFORMATION

During the entire synthesis process, all control information relevant to each design object is stored directly with the CSG. This characteristic will render the implementation of an FSM type controller extremely straightforward.

11.9 USE OF LOCAL STORAGE

In this last section, we present a new form of storage allocation that was conceived and implemented as part of the HAL system. We have avoided describing it earlier in the chapter for fear of hiding the fundamental concepts presented there.

Local storage is defined as a storage operation that lasts only one c-step and is assigned to store the values of all operations bound to a particular FU. For each operation assigned to this FU, a single local storage operation is created in the c-step where the value is generated. When the

value is also required later, a regular storage operation is created for the remaining c-steps.

The easiest way to demonstrate the use and benefit of this type of storage operation is through a simple example. In Figure 63, we depict a simple CSG where a comparison is made between the regular lifetime technique and our technique which uses local storage operations. In the first case, the regular storage operation *rs1* spans c-step 1 and 2 and is bound to *R1*. The output of *fu1* must therefore be stored in a different register (e.g. *R2*). This results in the slightly complex datapath shown in Figure 63 a), where *fu1* must transfer values to *R1* and *R2*.

In Figure 63 b), both local and regular storage operations are used. The small circles with a cross represent local storage and are labelled *lsn*. The label directly below indicates the bound register. In this example, a group of local storage operations (*ls1*, *ls2* and *ls3*) are created to store the result of all the operations performed by *fu1*. Likewise, local storage *ls4* is required for the operation performed by *fu2*.

a) Regular Storage

b) Local Storage

Figure 63. Comparison of Regular and Local Storage techniques

By definition, a local storage operation can only store a value for one c-step. The regular storage operation `rs1` is therefore created to store the value previously held by `ls1`. The important observation here is that `ls1`, `ls2` and `ls3` can be assigned to the same register, which results in the simpler datapath depicted in the lower half of Figure 63 b). Here, `fu1` need only transfer values to `R1`.

Binding of Local Storage: In the previous example we demonstrated the potential benefit of using local storage operations. They must be used judiciously however. Since they increase the total number of storage operations, the register binding task is made more difficult.

This potential difficulty is resolved with the following strategies:

1. Creating local storage operations in groups
2. Performing preliminary register binding
3. Restricting their creation to particular cases where they are likely to be useful

Local storage operations are created in groups where each group is associated with a particular FU. For each operation assigned to this FU, a single local storage operation is created in the c-step where the value is generated.

When the value is also required later, a regular storage operation is created for the remaining c-steps.

Local storage operations from the same group are guaranteed to be disjoint since they exist for only one c-step. Therefore we can immediately bind a single register to all the elements of a group.⁷ It can be considered as an implicit latch connected to the output of the FU.

Finally, only certain functional units are considered when creating groups. These are the ones for which the output values are used by other FUs predominantly in the next c-step. If this is not the case, regular storage operations are used to minimize the number of register-to-register transfers.

In the next chapter, we will present more difficult examples where the concept of local storage was used successfully.

⁷ This register can in turn be merged with others in a later step.

CHAPTER 12. DESIGN EXAMPLES

The examples presented in this chapter were used in some of the systems described in the literature survey. They were chosen to illustrate the flexibility of the HAL system to different constraints and to allow comparison with the results obtained from these systems.

Experimental Procedure:

For each of the examples presented, synthesis was performed using the same assumptions¹ as the original reference. They are listed at the beginning of each subsection. They varied considerably, which put the flexibility of the HAL system to test. Except where explicitly stated, the synthesis was always performed using the same parameters. The results were obtained without any fine tuning of the algorithm to the examples.

The interconnect costs presented are obtained using the simple cost function described in [49]. "Appendix A. Cost

¹ Disclaimer: The results presented in this chapter are given for illustrative purposes only. Whenever a comparison with another system is made, the assumptions and constraints of that system were duplicated as accurately as possible. However, as it is always possible that some constraints were left unspecified in the original references, these comparisons may or may not be absolute.

Functions and Structural Unit Costs" contains a reprint of this cost function as well as the selected cost values for different structural units. The costs are used here to indicate cost trends, and not absolute cost values as these are layout and technology dependent.

The CPU execution times given are for the scheduling phase only, using a XEROX 1108 Lisp machine. This is a single-user workstation in the medium-low performance range by today's standards, yet the system allows nearly interactive synthesis. For moderate size examples, the scheduling phase usually required less than two minutes CPU time, while the allocation and binding tasks ran typically in a minute or less (ten minutes for the largest example presented).

12.1 DIFFERENTIAL EQUATION EXAMPLE FROM HAL

This example corresponds to the CDFG of Figure 11 on page 73 of "Chapter 6. InHAL: Force-Directed Scheduling". It was first presented in [4] and subsequently used in [38], [20], [19] and [21]. The summary of costs for these and the HAL system is given in Figure 64. In the first four columns, the following conditions are assumed:

- The clock cycle is 100 ns
- The maximum time constraint is 400 ns (4 cycles)
- Additions are performed by adders in 40 ns
- Multiplications are performed by multipliers in 80 ns
- The latch delays are 10 ns
- Multi-cycle and chained operations are allowed

- Registers are preset with their original values (u, x and y)
- Constants (e.g. $\dot{d}x$, a, 3) are implemented as direct connections to power and ground lines.

Summary of Area Costs

- Differential equation example

Area Cost (time = 400ns)

System	HAL '86	Splicer '87	Catree '87	HAL '87	Splicer '87	HAL '87
CPU	40 s	n/a	n/a	40 s	n/a	120 s
Interconnect %	100	93	93	79	107	84
Registers %	100	100	83	83	100	83
FUs %	100	100	100	100	64	57

Figure 64. Cost summary for HAL example.

The first column in Figure 64 corresponds to an early version of the HAL system [4]. This result was improved on by the CATREE [38] and Splicer [19] systems as indicated in

the second and third columns. The fourth column corresponds to the datapath obtained using the current version of the HAL system. Interconnect, register and FU cost percentages (given below the graph) are given relative to the initial result of the early HAL system. In the current version, the register and bus allocation have been significantly improved to minimize storage and interconnect costs. The generated datapath is depicted in Figure 65.

As both the CATREE and Splicer systems made use of the HAL schedule, the minimum FU and register requirements will be the same. Each solution requires a minimum of two multipliers, one adder, one subtractor, one comparator and five

registers. The only possible cost reduction is obtained through the register and interconnect assignments.

One way of obtaining a more significant reduction is through *structural pipelining*. This was achieved in the Splicer system [21] by using a timing constraint of eight clock cycles, dividing the cycle time by two (i.e. 50 ns instead of 100 ns) and making use of a two-stage pipelined multiplier. In this way, the original time constraint is still respected. The fifth column in Figure 64 corresponds to the Splicer datapath, shown in the top portion of Figure 67.

The last result (sixth column) is for the HAL system using the schedule given in the fully bound CSG of Figure 66. The assumptions are the same as those given earlier, except for the following:

- The clock cycle is 50 ns (instead of 100 ns)
- The time constraint is still 400 ns (but now 8 c-steps)
- Multiplications are performed by pipelined multipliers in 2 c-steps
- The cost of a pipelined multiplier is roughly equal to that of a regular one (the cost of the extra stage latch being negligible with respect to the total FU area)
- Multi-cycle and chained operations are allowed

Figure 66. CSG for differential equation example (using pipelined multipliers).

The datapath obtained from this CSG is depicted in the lower half of Figure 67. The use of force-directed scheduling and a different approach to register and bus allocation are the two main reasons for the efficiency of the design obtained from the HAL system.

Splicer Datapath (pipelined multiplier)

HAL Datapath
(Pipelined Multiplier)

Figure 67. Splicer and HAL Datapaths (using pipelined multipliers).

12.2 MAHA CODE SEQUENCE EXAMPLE

This example was presented in [27] and makes use of the CSSP program [26] for the initial stage partitioning. Two results are given in the paper, the fastest allocation and the cheapest one. For the fastest allocation, the graph is partitioned into four stages (clock cycles). For the cheapest allocation, the graph is partitioned into eight stages.

Assumptions:

- All similar operations have equal propagation delays.
- For the fastest allocation: the clock cycle time is such that a maximum of three successive operations can be combined in a single c-step.
- For the cheapest allocation: only one operation can be performed per c-step.
- There is no sharing of FUs executing mutually exclusive operations (i.e. operations that are on either side of an if-then-else statement).
- MAHA execution times are for a VAX 11/750.

For illustrative purposes, let us make the following additional assumptions (which are consistent with the ones given above):

- Add and subtract operations are performed by adders or subtractors in 40 ns
- The latch delays are 20 ns
- Adder and subtractor cell areas are nearly equal

For the cheapest allocation (constraint: 8 c-steps), the HAL system generates a schedule requiring the same number of

FUs as the MAHA system (one adder and one subtractor). This was expected (hoped for) as all FUs have 100% utilization.

MAHA schedule

For the fastest allocation (time constraint: 4 c-steps), the MAHA system arrived at the schedule shown in Figure 68. Functional unit utilizations are given on the right-hand side of the figure.

From this figure, two important observations can be made:

- The clock cycle time must be at least equal to the sum of three adder/subtractor propagation delays plus the latch delays. This is due to the longest delay path in the first c-step (two additions and one subtraction). Here, this would yield a minimum clock cycle of 140 ns.
- A minimum of two adders and three subtractors are required, as there are two additions and three subtractions scheduled in c-step one.

Although not optimal, this schedule is far better than the one obtained using ASAP scheduling which would require four subtractors and three adders.

HAL schedule

Using the same time constraint as specified above, the HAL system generated the schedule depicted in Figure 69.

This schedule improves on the previous one in two ways:

- The longest path in any c-step is equal to the combined propagation delays of an adder and a subtractor, plus the latch delay. Using the values above, this gives 100 ns.
- A minimum of two adders and two subtractors are required. FU utilizations are 100%.

For this optimal schedule, the reduction in FU area (with respect to MAHA) is equal to 20%. Furthermore, this was achieved while reducing the clock cycle time by 25-30%

(the actual value will depend on the latch propagation delays).

List schedule

It is instructive to generate schedules using a list scheduling method with operation *urgencies* or *mobilities* chosen as the priority function (refer back to "Chapter 3. Related Work" for definitions). List scheduling requires a specified clock cycle and a limit on the maximum number of functional units of each type. Since we have a lower bound established already from the previous HAL schedule (two adders, two subtractors), we will simply use this.

We will perform scheduling using two different clock cycle times:

1. Clock cycle = 100 ns (to compare with HAL result)
2. Clock cycle = 140 ns (" " " MAHA ")

The schedule obtained using the 100 ns cycle time is given in Figure 70. The numerical value alongside each operation is equal to its *urgency*.

From this figure we can see that to respect the constraints on the number of FUs, the scheduler required an additional c-step. By comparing this schedule with the HAL schedule of Figure 69 on page 214, we can see why this occurred. The HAL system scheduled the *add* and *subtract* operations on the far right in two completely different c-steps, to obtain the best distribution balancing. In list scheduling, both of these operations have low urgencies

(equal to 2) and high mobilities. Using either urgency, or mobility as a priority function these operations will always be deferred so that more urgent (or less mobile) ones can be scheduled. As the list scheduler has no global indication of operation concurrencies, it cannot predict that these operations will get caught in the last c-step and force the last addition to be deferred in c-step five to respect FU constraints.

Running this example again with the same clock cycle as MAHA (140 ns), the algorithm will generate a schedule which now requires four c-steps (and still uses two adders and two subtractors). This result is therefore cheaper than MAHA's but still slower than HAL's.

HAL schedule (with mutual exclusion)

As we mentioned earlier, MAHA does not allow sharing of mutually exclusive operations by a single FU. If we lift this restriction, and take mutual exclusion into account by using the method illustrated in Figure 15 on page 78 of "Chapter 6. InHAL: Force-Directed Scheduling", we obtain the schedule shown in Figure 71. Fork and join nodes are represented as small black circles. The four additions scheduled in the third c-step are therefore mutually exclusive.

The table of Figure 72 summarizes the allocations presented above. The total time constraint (number of c-steps multiplied by clock cycle) is given for each allocation.

COMPONENT	MAHA	List Sched.		HAL	HAL mutual excl
		4	5		
C-steps	4	4	5	4	4
Tot. Time	560	560	500	400	400
subtractor (-)	(-) (-) (-)	(-) (-) (-)	(-) (-) (-)	(-) (-) (-)	(-) (-) (-)
adder (+)	(+) (+)	(+) (+)	(+) (+)	(+) (+)	(+) (+)
CPU (sec)	160	n/a	n/a	25	45

Figure 72. Summary of results for MAHA example [27].

12.3 PIPELINED FIR FILTER FROM SEHWA

The results presented here are for the pipelined 16-point digital FIR filter example borrowed from [44]. In this example, *algorithm pipelining* is used to increase throughput.

The following assumptions are stated:

- The maximum stage time limit is 100 ns.
- The latency is equal to 300 ns.
- Additions are performed by adders in 40 ns.
- Multiplications are performed by multipliers in 80 nsec.
- The latch delays are 20 ns.
- Chained operations are allowed.

In Figure 73 we present the results for three scheduling methods:

1. SEHWA: Backward feasible scheduling.
2. SEHWA: Exhaustive feasible scheduling.
3. HAL: Force-directed pipeline scheduling.

These results include the algorithm complexity, the number of cycles, the FU allocation, and CPU time. The feasible schedule is used to establish an upper cost bound for the exhaustive scheduling phase that follows.

The force-directed algorithm generated a schedule unlike the ones obtained by the feasible or exhaustive algorithms. However, the allocation cost is identical to that obtained by the exhaustive algorithm (i.e. an optimal cost). The datapath realization was not given in [44].

Pipelined 16 point FIR filter (DAC '86 Sehwa article)

System	SEHWA		HAL
Algorithm	Feasible	Exhaustive	Force-directed
Complexity	$O(n^2 \log n)$	$O(n^{n+2})$	$O(n^2)$
No. of cycles	7	6	6
No. of adders	6	5	5
No. of multipliers	3	3	3
CPU time	n/a	n/a	30 sec

Figure 73. FIR filter results for SEHWA Example

We also ran the second pipeline example from the same article [44]. This example contained mutually exclusive operations that require special consideration in pipelined data-flow graphs. In this case, our system generated a schedule identical to the one obtained from Sehwa's exhaustive algorithm.

12.4 FACET CODE SEQUENCE EXAMPLE

To highlight the difference in the clique partitioning approaches used in the Facet and HAL systems, we will use the example presented in [14]. In this case, we will start with the schedule and functional unit allocation generated by the Facet system. Therefore, the register and interconnect binding phases can be compared.

Assumptions:

- Each operation requires one c-step for execution.
- All five input variables (V1, V2, V4, V6, V10) are available from five registers, so no input ports are needed. This also implies that a minimum of five registers are required.
- The code sequence generates two output variables that must also be stored. However, the original variables may be overwritten. For example, in the Facet data path, the registers V1 and V2 are overwritten by these output variables. In the HAL data path, the output variables overwrite registers R1 and R2.

The Facet system used a four c-step timing constraint and allocated three functional units:

- FU1: performs ADD, OR and MULTIPLY² operations
- FU2: " AND, ADD and SUBTRACT "
- FU3: " DIVIDE operations

The data paths obtained from both systems are given in Figure 74. The correspondance between the Facet variables and the HAL registers is indicated beside the latter. The cost graph shown in Figure 75 was obtained with the cost function (Appendix A) used earlier. We can see that the reduction in register cost was obtained without incurring extra interconnect costs. On the contrary, these costs have also been reduced significantly. There are several reasons for the reduced cost obtained by the HAL system:

- The *local storage* technique described in the previous chapter was used extensively here. Both registers R1 and R2 are bound to local storage operations. For example, the first part of the lifetime of variable V3 is bound to the local register R2, while the second part is bound to R3.
- Further reduction in interconnect was achieved with the operand alignment technique also described earlier.
- Due to the relatively small size of the example, the initial weight threshold was set quite low (to a value of 2). This allowed for a fairly extensive exploration of the possible register merges.
- Finally, several muxes were merged into a single bus, further reducing interconnect costs.

² The HAL system would normally not merge MULTIPLY operations with simpler combinatorial operations such as ADD and OR.

Facet Data Path

HAL Data Path

Figure 74. Comparison of Facet and HAL datapaths

12.5 FIFTH-ORDER DIGITAL ELLIPTIC WAVE FILTER

The CDFG used in this example is borrowed from Kung, Whitehouse and Kailath's book on signal processing [42] and implements a fifth-order wave digital elliptic filter. This is a much more substantial behavioral description that contains 43 operations (additions and multiplications) submitted to over 60 precedence constraints. This example was chosen as a benchmark for the "1988 Workshop on High-Level Synthesis" [51].

It is stated in [42] that multiplications require an execution time that is twice as long as that for additions. Here they were assigned two and one c-steps respectively. The critical-path is therefore 17 c-steps long.

The CSG given in Figure 76 was obtained using a time constraint of 19 c-steps, under the assumption that two-stage pipelined multipliers were available from the cell library. Although they are not shown in Figure 76, all multiplications have second operands which are the filter coefficients. We will assume that these operands are fixed and can be stored in a small coefficient ROM.

The data path obtained using the schedule shown is given in Figure 77 on page 227. Once again, the use of local storage here was the cause of a non-negligible reduction in total interconnect costs; approximately 15% in this specific case. Multiplexer merging and operand alignment were also important sources of interconnect reduction.

Figure 76. Schedule for wave filter example.

Data path of Fifth-Order Wave Filter

Figure 77. Data path for wave filter example.

There are four other observations that can be made from the data path of Figure 77:

1. As there are eleven distinct data edges crossing the boundary of c-steps 14 and 15, a minimum of eleven registers are required. This design used twelve to minimize interconnect costs.
2. A relatively small number of buses were used. These buses have mostly local connections to/from them.
3. The system generated the equivalent of a register file (top half of the figure) that is associated with the most used bus (i.e. bus 2 in the center of the figure).
4. Most of the interconnections are local to the area defined by a single functional unit. This is particularly true for the two adders on the lower half of the figure.

The graph shown in Figure 78 summarizes the FU, register and allocation costs for the example. The figure also gives the number of adders, multipliers, registers and mux inputs³ that were used in each design. The 17 c-step result (requiring 3 multipliers and 3 adders) is to be compared with that given in [42] (4 multipliers and 4 adders). Moreover, the flexibility of the system allows to explore alternate speed-cost trade-offs.

The 19 c-step result (1 pipelined multiplier and 2 adders) is a good example of a substantial area cost reduction (approximately 50%) against a small loss in speed (close to 10%) as compared to the 17 c-step result given in [42].

This example is eloquent proof that the system can solve relatively difficult synthesis problems in a reasonable time. A close examination shows that the FU utilizations are optimal for all four results.

Our experience has shown that it is extremely difficult to solve this problem manually and obtain results of the same quality. This is mostly due to the large number of interactions between operations (i.e. 43 operations submit-

³ This value is actually the combined number of inputs to muxes and buses; where a bus is equivalent to a mux with multiple outputs.

ted to over 60 precedence constraints). This problem is compounded by the fact that the number of possible schedules grows exponentially with the number of c-steps allowed.

13.1 LIMITATIONS OF SCHEDULING AND ALLOCATION APPROACHES

Although most types of scheduling problems can be solved using the refined force-directed scheduling algorithm, there remain a few difficult problems that may need to be addressed.

Researchers at CMU have implemented the basic algorithm presented in [7] and are presently investigating its use for pipeline synthesis [52]. Some limitations of the force-directed scheduling were discovered.

Determination of operation execution time: A first difficulty lies in the determination of an operation's execution time when two FUs of different speeds can be assigned to it. A simple heuristic is to assign the fast FU to operations on the most critical paths, and the slower FU to the others. The latter operations may then be assigned multi-cycle execution times. Substitutions can be performed after the preliminary schedule is established and more accurate timing constraints are available.

Allocation of multiple ALUs: Another problem is when an operation can be assigned to many different ALUs. In this

case it is difficult to decide which DG should be used for the force calculations. A relatively straightforward solution is to perform an early assignment of operations to each ALU. One multi-class DG will be created for each ALU and the force will be evaluated using the DG of the ALU assigned to the operation.

Combined algorithm and structural pipelining: The CMU researchers also pointed out that some applications may benefit from the possibility of scheduling pipelined operations in an algorithm that is itself pipelined. Although this possibility has not been tested yet, the existing algorithm should support this directly.

Combined area-cost tradeoff function: Finally, they added that there are cases where the user has not specified an explicit time or area constraint, but would like a compromise based on some speed-area tradeoff guideline. In the original version published in [7], this can only be determined by modifying the time or area constraint iteratively until a satisfactory compromise is achieved. To deal with this type of mixed constraint, the simple extension proposed in section "Scheduling Under Mixed Time-Area Constraints" on page 119 of "Chapter 7. Refined Scheduling Algorithm" may be used.

Short-sightedness: For very loose timing constraints, the force algorithm sometimes exhibits short-sightedness in the first few iterations. It may schedule one operation such that, although it is the best assignment given the current distribution, it may unnecessarily constrain many other operations. These operations are then 'caught' into reduced time frames.

A promising solution is to multiply each operation's force with the width of its time frame. This will favor the scheduling of operations with low force and short time frames so that the ones with larger time frames will tend to be scheduled later.

Another solution is to use one of the two algorithm extensions presented in section "Scheduling Under Fixed Hardware Constraints" on page 115 of "Chapter 7. Refined Scheduling Algorithm", provided that a global or specific hardware constraint has been specified.

Clock cycle determination: In the system's present state, the user must define the system clock cycle time. This decision was taken on the assumption that the system is often part of a larger one, and the clock cycle is predefined. To automate the clock cycle determination, the addition of a clocking scheme synthesis phase similar to [26] would be useful for stand-alone digital systems.

13.2 LIMITATIONS OF DATA PATH SYNTHESIS ALGORITHMS

Physical proximities: The interconnect-directed clique partitioning algorithm used for register and multiplexer merging would benefit from the availability of two-dimensional physical partitioning information. In the algorithm's present form, this information can be integrated simply by dividing the structural weight by a physical locality factor.

For example, this factor would be equal to 1 for two registers that are part of the same physical partition. For registers assigned to different partitions, the factor would increase with the distance between them.

Floorplan information would also be useful in making the interconnect area estimation used by the MidHAL module more precise. For these reasons, the system would benefit from a preliminary floor planning phase such as the one used in the BUD-DAA system [22].

Hierarchical descriptions: The system generates only purely synchronous data paths (with the exception of chained operations) from a flat CDFG description.

These data paths are synthesized from separate data flow graphs that are treated independently. The data flow graphs

cannot be nested, although inline expansion could be used to achieve the same effect.

Another method of nesting can be achieved by using I/O operations. The boundary of a nested CDFG is determined by I/O operations that can be later substituted for links with operations of another surrounding CDFG.

Optimization of data paths with multiple bit-widths: Although the bit-widths of all structural units (SU) are stored and maintained in the HAL system, only simple methods for combining SUs of different bit widths are implemented.

In register and bus merging for example, more advanced methods will be required for data paths where many different bit-widths are specified.

CONCLUSION

We have shown the innovations the HAL system brings to automatic data path synthesis by comparing it with well-known systems. These innovations are present in the implementation approach as well as in the synthesis process itself.

The overall structure of the programming environment is consistently object-based, while the methods associated with each object can be expressed in one or another programming paradigm. This allows for a conceptually uniform representation of the digital system throughout the whole synthesis process, while enabling partitioning of the overall task into several complementary subtasks suited to different paradigms.

The scheduling methodology presented features a novel *force-directed* scheduling algorithm that is invoked in a four phase scheduling/allocation scheme. We have shown how this approach makes it feasible to incorporate explicit allocation information to optimize the scheduling process. The information transfer is made possible by stepwise refinement of the scheduling and allocation tasks.

The force-directed scheduling algorithm at the heart of this process was shown to be $O(n^2)$. In spite of this rela-

tively low complexity, the algorithm explores the search space in a global fashion and produced optimal results for nearly all of the examples attempted to date.

Results obtained were shown to be as good or better as the ones previously published. Furthermore, the flexibility of the system was highlighted by the variety of constraints and requirements it had to deal with. These include:

- Chained and multi-cycle operations
- Mutually exclusive operations
- Algorithm pipelining
- Optimized use of single and multi-function FUs (e.g. Alus)
- Minimization of required hardware modules for a specified time constraint
- Optimization of system speed with limited hardware resources

The synthesis methodology presented features knowledge-based global allocation of functional units to fulfill explicit speed/area constraints; a functional unit binding approach that minimizes the number of data transfer paths; a powerful storage to register binding mechanism that exploits the notion of *local storage*; optimized alignment of functional unit operands; and finally, an effective interconnect-directed clique partitioning algorithm for reg-

ister and multiplexer merging, where interconnect affinities are used to prune the search space.

Comparisons with other systems have demonstrated the effectiveness of this approach. The data paths generated where characterized by low register and interconnect area costs.

Finally, the extended design-space exploration enabled by this methodology is of major importance in relation to the desired use of the system (i.e. application specific IC's).

GLOSSARY

ALAP: As late as possible; scheduling technique where operations are scheduled as late as possible without lengthening the time required to execute a data flow graph.

ASAP: As soon as possible; scheduling technique where operations are scheduled as soon as their operands are available.

ASIC: Application Specific Integrated Circuit; circuit intended for a specialized application (as opposed to GPICs — see below). ASICs are usually produced in relatively small quantities.

Allocation: Creation of a set of functional units to perform sets of operations.

Binding: Actual mapping of a specific functional unit to a distinct set of operations.

CDFG: Control and Data Flow Graph; data flow graph augmented with control constructs (e.g. loop, case) and control dependencies (see definition below).

Control Dependency: Indicates that one or more operations must be performed before/after one or more others. This could be due to local timing constraints between operations or due to conditional operations (e.g. a fork must precede the corresponding join).

Control Path: Part of a circuit where the sequencing of data path operations is stored. It generates all control signals to the data path based on the stored event sequence.

CSG: Control step graph; refined representation of the initial data flow graph. Each operation in the DFG is assigned to an explicit control-step, while maintaining any precedence constraints.

Data Dependency: Occurs when one operation transfers data to a second one. In this case the latter can only be initiated after the former has terminated execution.

Data Path: Part of a circuit where data transfer and processing occurs.

Data Transfer: Transfer of a data value from one structural unit to another.

DFG: Data flow graph; graphical representation of an algorithmic description which specifies data dependencies explicitly. Each operation is represented as a node, each value (result of operation) is represented as an edge.

DG: Distribution Graph; used to store the sum of operation probabilities in every control-step. It can be assigned to single or multiple operations types.

FU: Functional Unit; hardware component that will be assigned to execute a set of combinatorial (arithmetic or logic) operations.

GPIC: General Purpose Integrated Circuit; circuit intended for a general class of applications (as opposed to ASICs — see above). GPICs are usually produced in large quantities.

IU: Interconnection Units; hardware components that will be assigned to perform data transfers between storage units (SU) and functional units (FU). They include multiplexers, buses, direct single connections and direct fanout connections.

Operation: Arithmetic, logic or storage operation usually present in a data flow graph.

Operation trace: Simplified representation of the control step graph that does not include precedence constraints.

Precedence Constraints: Constraints on operations in a data flow graph due to data or control dependencies.

RTL: Register-Transfer Level; structural description of an architecture which specifies the functional units, registers and interconnections.

Storage (local): Storage operation that is tagged on to the output of specific arithmetic and logic operation. It is used for only one control-step.

Storage (regular): Storage operation that is inserted between dependent arithmetic and logical operations. It may be used for one or more control-steps.

Structural Units: class of hardware components that make up the data path. These components include functional units (FU), storage units (SU) and interconnection units (IU).

SU: Storage Units; hardware components that will be assigned to perform storage operations. This class of components includes hardwired constants, registers and memory (ROM, RAM).

Time frame: Range of control-steps in which an operation could possibly be scheduled.

BIBLIOGRAPHY

- 1 D. Johannsen, "Bristle Blocks: A Silicon Compiler", Proceedings of the 16th Design Automation Conference, June 1979, pp. 310-313.
- 2 A.V. Goldberg et al, "Approaches Toward Silicon Compilation", IEEE Circuits and Devices Magazine, May 1985, pp. 29-39.
- 3 D.D. Gajski, "Silicon Compilation", IEEE Computer Magazine, December 1983.
- 4 P.G. Paulin, J.P. Knight, E.F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", Proceedings of the 23rd Design Automation Conference (DAC), July 1986, pp.263-270.
- 5 P.G. Paulin, J.P. Knight, "Extended Design-Space Exploration in Automatic Data Path Synthesis", Proceedings of the 1986 Canadian Conference on VLSI, Montreal, October 1986, pp. 221-226.
- 6 P.G. Paulin, "Use of a Multi-Paradigm AI Software Tool for Automated VLSI Design", Proceedings of COMPEURO 1987, Hamburg FRG, May 1987, p. 510.
- 7 P.G. Paulin, J.P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis", Proceedings of the 24th Design Automation Conference (DAC), Miami Beach, July 1987, pp.195-202.
- 8 P.G. Paulin, J.P. Knight, "Scheduling and Allocation for Pipelined ASICs", Proceedings of the 1987 Canadian Conference on VLSI, Winnipeg, October 1987.
- 9 P.G. Paulin, J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs", submitted to the IEEE Transactions on Computer-Aided Design, December 1987 (awaiting acceptance).
- 10 P.G. Paulin, J.P. Knight, "High-level Synthesis Benchmark Results using Global Scheduling and Register Allocation Techniques", Proceedings of the International Workshop on Logic and Architecture Synthesis for Silicon Compilers, Grenoble, France, May 1988 (to be published).
- 11 P.G. Paulin, J.P. Knight, "HAL: Behavioral Synthesis Using Global Scheduling and Binding Algorithms", invited paper to "Integration: the VLSI Journal", North-Holland (to be published).

- 12 A.C. Parker, "Automated Synthesis of Digital Systems", IEEE Design & Test, November 1984, pp. 75-81.
- 13 T. Blackman et al, "The Silc Silicon Compiler: Language and Features", Proceedings of the 22nd Design Automation Conference (DAC), June 1985, pp. 232-237.
- 14 C. Tseng, D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems", IEEE Transactions on CAD, July 1986, pp. 379-395.
- 15 C.Y. Hitchcock, D.E. Thomas, "A Method of Automatic Data Path Synthesis", Proceedings of the 20th Design Automation Conference, July 1983, pp. 484-489.
- 16 D.E. Thomas et al, "Automatic Data Path Synthesis", IEEE Computer, December 1983, pp. 59-70.
- 17 M.C. McFarland, "Allocating Registers, Processors and Connections", Internal Report, Dept. of Electrical Engineering, Carnegie-Mellon University, August 1981.
- 18 C.Y. Hitchcock III, "Automated Synthesis of Data Paths", Master's thesis (available as report no. CMUCAD-83-4), Dept. of Electrical Engineering, Carnegie-Mellon University, January 1983.
- 19 B.M. Pangrle, D.D. Gajski, "Design Tools for Intelligent Silicon Compilation", IEEE Transactions on Computer-Aided Design, Vol. CAD-6, November 1987, pp. 1098-1112.
- 20 B.M. Pangrle, D.D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation", Proceedings of the IEEE International Conference on Computer Design (ICCD), October 1987.
- 21 F.D. Brewer, D.D. Gajski, "Knowledge-Based Control in Micro-Architecture Design", Proceedings of the 24th Design Automation Conference (DAC), Miami Beach, July 1987, pp. 203-209.
- 22 M.C. McFarland, "BUD: Bottom-Up Design of Digital Systems", Proceedings of the 23rd DAC, Las Vegas, July 1986, pp. 474-479.
- 23 T.J. Kowalski et al, "The VLSI Design Automation Assistant: From Algorithms to Silicon", IEEE Design & Test, August 1985, pp. 33-43.
- 24 M.C. McFarland, Private Communication, February 1988.
- 25 E.F. Girczyc and J.P. Knight, "An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Schedul-

- ing", Proc. of the IEEE International Conference on Computer Design (ICCD), October 1984, pp. 726-731.
- 26 N. Park and A.C. Parker, "Synthesis of Optimal Clocking Schemes", Proceedings of the 22nd Design Automation Conference (DAC), July 1985, pp. 489-495.
 - 27 A.C. Parker et al, "MAHA: A Program for Datapath Synthesis", Proceedings of the 23rd Design Automation Conference (DAC), Las Vegas, July 1986, pp. 461-466.
 - 28 F.J. Kurdahi and A.C. Parker, "REAL: A Program for Register Allocation", Proceedings of the 24th Design Automation Conference (DAC), Miami Beach, July 1987, pp. 210-215.
 - 29 M.C. McFarland, "The Value Trace: A Data Base for Automated Digital Design", Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, December 1978.
 - 30 J. Nestor, "Specification and Synthesis of Digital Systems with Interfaces", Ph.D. thesis (available as report no. CMUCAD-87-10), Dept. of Electrical Engineering, Carnegie-Mellon University, April 1987.
 - 31 J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", IEEE Transactions on Computers C-30(7), July 1981, pp. 478-490.
 - 32 J. Nestor, D.E. Thomas, "Behavioral Synthesis with Interfaces", Proceedings of the IEEE International Conference on CAD (ICCAD), November 1986, pp. 112-115.
 - 33 M. Stefik et al, "Knowledge Programming in LOOPS: Report on an Experimental Course", The AI Magazine, Fall 1984, pp.3-13.
 - 34 P. Winston and B. Horn, "LISP", Addison-Wesley, 1981.
 - 35 A. Borgida, S. Greenspan, J. Mylopoulos, "Knowledge Representation as the Basis for Requirements Specifications", IEEE Computer Magazine, April 1985, pp.82-91.
 - 36 A. Goldberg, D. Robson, "Smalltalk-80 The Language and its Implementation", Addison-Wesley, Don Mills, Ontario, 1983.
 - 37 D.D. Gajski, N.D. Dutt and B.M. Pangrle, "Silicon Compilation (Tutorial)", Proceedings of the IEEE 1986 Custom Integrated Circuits Conference, Rochester NY, May 1986, pp. 102-110.

- 38 C.H. Gebotys, M.I. Elmasry, "A VLSI Methodology with Testability Constraints", Proceedings of the 1987 Canadian Conference on VLSI, Winnipeg, October 1987.
- 39 S. Davidson et al, "Some Experiments in Local Microcode Compaction for Horizontal Machines", IEEE Transactions on Computers, July 1981, pp. 460-477.
- 40 P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System", Proceedings of the 23rd Design Automation Conference (DAC), Las Vegas, July 1986, pp. 271-277.
- 41 H. Trickey, "Flamel: A High-Level Hardware Compiler", IEEE Transactions on CAD, March 1987, pp. 259-269.
- 42 S.Y. Kung, H.J. Whitehouse, T. Kailath, "VLSI and Modern Signal Processing", Prentice Hall, 1985, pp. 258-264.
- 43 A.W. Nagle, A.C. Parker, "Algorithms for Multiple-Criterion Design of Microprogrammed Control Hardware", Proceedings of the 18th Design Automation Conference, June 1981, pp. 486-493.
- 44 N. Park, A.C. Parker, "SEHWA: A Program for Synthesis of Pipelines", Proceedings of the 23rd Design Automation Conference (DAC), Las Vegas, July 1986, pp. 454-460.
- 45 E.F. Girczyc, "Loop Winding -- a Data Flow Approach to Functional Pipelining", Proceedings of the International Symposium on Circuits and Systems (ISCAS), Philadelphia, May 1987, pp. 382-385.
- 46 E.M. Reingold, J. Nievergelt and N. Deo, "Combinatorial Algorithms: Theory and Practice", Prentice-Hall, 1977.
- 47 Jan Vanhoof, Jan Rabaey and Hugo De Man, "A Knowledge-Based CAD System for Synthesis of Multi-Processor Digital Signal Processing Chips" Proceedings of the IFIP International Conference on VLSI, Vancouver, August 1987.
- 48 H. De Man et al, "Cathedral-II : A Silicon Compiler for Digital Signal Processing", IEEE Design & Test Magazine, December 1986, pp. 13-25.
- 49 J. Midwinter, "Improving Interconnect for the Behavioral Synthesis of ASICs", M.Sc. Thesis, Carleton University, February 1988 (expected graduation date).
- 50 A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignment Within Large Apertures", Proceedings of the 8th IEEE Design Automation Workshop, 1971, pp. 155-169.

- 51 E. Detjiens (Workshop Chair), "Workshop on High-Level Synthesis", Orcas Island, Washington, January 1988.
- 52 R. Cloutier, Private Communication, November 1987.

APPENDIX A. COST FUNCTIONS AND STRUCTURAL UNIT COSTS

This appendix contains a reprint of the cost-function description written by Jenny Midwinter as part of her M.Sc. thesis [49].

Improving Interconnect for the Behavioral Synthesis of ASICs

Jenny Midwinter, February 1988

This thesis presents a register-transfer level interconnection cost evaluation algorithm. The need for such an algorithm is driven by the questions:

"How can the interconnection that has been automatically generated be evaluated without actually performing a full placement, layout and routing of the circuit? What is the area cost of this interconnection?"

Two cost evaluations schemes were developed and are presented here. Following this, we present the parameter values used in this thesis for interconnect cost evaluation, as well as the cost values used for constants, registers and different types of functional units.

COST SCHEME I

Figure 79 shows three types of possible connections: busses, multiplexers and leads. In each case, one bit wide transfers are assumed.

Let: A = the cost (area) of one lead
G = the cost (area) of one pass transistor
B = the cost (area) of one control bit
n = the number of inputs to a mux or bus
m = the number of outputs from a lead, mux or bus

Consider the cost of a lead connection illustrated in Figure 79 a). The cost of connecting registers R1 and R3 is proportional to the lead area between the two registers, i.e. area A1. The cost of connecting R1 and R4, once the connection between R1 and R3 has been established, is proportional to the lead area A2. Note that some of the area A1 is used by the R1 to R4 connection. From this, let it be assumed that the cost of connections between R1 and each additional destination, will be half of that of the first one. This can be more formally stated as the fanout cost:

$$\text{Lead Cost} = A(m+1)/2 \quad (1)$$

Interconnection Schemes

Figure 79. Interconnection Schemes

Consider the cost of the bus configuration depicted in Figure 79 b). This cost consists of the decode area, pass transistor area, and bus lead area. Let:

$$\begin{aligned} \text{Bus Control Cost} &= \text{Area of decode logic} \\ &\quad + \text{Area of control leads from} \\ &\quad \text{decode logic to pass transistor} \\ &= B \log(n) + A_n \end{aligned} \quad (2)$$

$$\begin{aligned} \text{Pass Trans. Cost} &= \text{Area of pass transistor} \\ &= G_n \end{aligned} \quad (3)$$

$$\begin{aligned} \text{Bus Lead Cost} &= \text{Area of fanout leads} \\ &= A(m+1)/2 \quad [\text{from (1)}] \end{aligned} \quad (4)$$

Note that there is no lead area cost associated with the inputs to the bus. This cost has been assumed to be absorbed by the pass transistor cost and the bus lead cost. Therefore, the bus cost is:

$$\text{Bus Cost} = B \log(n) + G_n + A_n + A(m+1)/2 \quad (5)$$

The cost of the mux type interconnection (Figure 79b) will consist of the decoder and pass transistor areas — which are integral parts of the mux — the *mux-to-destination* lead area, and the *source-to-mux* lead area. Note that there is no lead cost associated with the control, as the decoder and pass transistors are usually part of the mux. Let:

$$\begin{aligned} \text{Mux Control Cost} &= \text{Area of decode logic} \\ &= B \log(n) \end{aligned} \quad (6)$$

$$\begin{aligned} \text{Pass Trans. Cost} &= \text{Area of pass transistor} \\ &= G_n \end{aligned} \quad (7)$$

$$\begin{aligned} \text{Mux lead Cost} &= \text{Area of source-to-mux leads} \\ &\quad + \text{Area of mux-to-destination leads} \\ &= A_n + A(m+1)/2 \end{aligned} \quad (8)$$

Therefore,

$$\text{Mux Cost} = \text{Blog}(n) + G_n + A_n + A(m+1)/2 \quad (9)$$

On comparing the mux cost, equation (9), with the bus cost, equation (5), it can be seen that both are identical. This is an artifact of the assumption of bit-wide data paths.

COST SCHEME II

Cost scheme II is similar to the one presented in the previous section, except that b -bit wide transfers are considered. Recalculating equations (1) to (4) yields:

$$\text{Bus Cost} = \text{Blog}(n) + G_{nb} + A_n + A_b(m+1)/2 \quad (10)$$

From equations (6) to (8), we obtain:

$$\text{Mux Cost} = \text{Blog}(n) + G_{nb} + A_{nb} + A_b(m+1)/2 \quad (11)$$

Therefore, it can be seen that for b -bit wide data paths, the mux is more expensive than the bus. This difference can be explained by the fact that the ' A_n ' term in the bus cost equation represents the cost of the leads from the

decode unit to each set of pass transistors — only one lead per source is required. Whereas the 'Abn' term in the mux cost equation represents the cost of the leads from each source to the mux — there are b leads for each source.

CHOSEN PARAMETER VALUES AND STRUCTURAL UNIT COSTS

In this thesis we have decided to use the formula (11) for both mux and bus costs. This ensures that we can establish an upper cost bound that is independent of the final choice of a mux or a bus for the final implementation. Using this cost function, the values chosen for A, B and C are:

A (cost of one lead)	= 1
G (cost of one pass transistor)	= 1
B (cost of one control bit)	= 1/2

Furthermore, the values given in the thesis correspond to a single bit datapath. i.e.:

b (bit-width of data paths)	= 1
-----------------------------	-----

The costs (per bit) for the registers and functional units used were chosen as:

Register (per bit)	= 4
Constant "	= 1 = A (i.e. cost of one lead)
Multiplier	= 50
Multiplier (pipelined)	= 50
Adder	= 15
Subtractor	= 15
Comparator	= 10
Alu (performing add, subtract and compare)	= 30

END

08-08-88

FIN