1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche 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 film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche 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, examens publiés, etc.) ne sont pas microfilmés.

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

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

AUTOMATIC GENERATION OF MICROSEQUENCED DATA PATHS
TO REALIZE ADA CIRCUIT DESCRIPTIONS

by

Emil F. Girczyc

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

Doctor of Philosophy

Department of Electronics
Faculty of Engineering
Carleton University
Ottawa, Ontario
July, 1984

© copyright

1984, Emil Girczyc

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 0-315-29816-2

The undersigned recommend to the Faculty of Graduate Studies
and Research the acceptance of the Thesis:

"Automatic Generation of Microsequenced Data
Paths to Realize Ada Circuit Descriptions"

submitted by Emil F. Girczyc in partial fulfillment of the
the requirements for the degree of Doctor of Philosophy.

Prof. M.A. Copeland
Chairman,
Department of Electronics

Prof. J.P. Knight
Thesis Co-supervisor

Prof. R.J.A. Buhr
Thesis Co-supervisor

Dr. W.M. Snelgrove
External Examiner

Carleton University

July, 1984

ABSTRACT

This thesis describes a hardware compiler design capable of compiling an algorithmic circuit description written in a subset of Ada to a standard cell net-list. The heart of the compiler is an allocator based on graph grammars and urgency scheduling. The use of scheduling techniques allows the designer to specify timing latency at the functional level. The graph grammar formulation allows all technology dependent data to be kept in production libraries which are easily updated to incorporate new technologies. The allocator is cost based. The cost includes an estimate of interconnect and control overhead which allows the allocator to exchange interconnect area for active area to reduce the total area.

The requirements of an algorithmic level hardware description language are shown to be similar to software languages, with only limited feature enhancement. The feasibility of using a subset of Ada as a hardware description language is demonstrated. Methods are presented for realizing the extra features required for hardware description within the syntax of Ada. This allows the compiled Ada program to act as a functional simulator. Rules are presented for translating a circuit described in the Ada subset onto a control/data-flow graph, the intermediate level form of the allocator.

In memory of my father,
Fabian Girczyc

Acknowledgements

I would like to thank my supervisors, Drs. J.P. Knight and R.J.A. Buhr for their contribution to this work. I would also like to thank Dr. A.R. Boothroyd for helping me through the bureaucracy and for several philosophic discussions.

I wish to acknowledge the contribution to this work by the IC design and CAD staff at BNR, especially, Drs. P. Benedek, M. Nakhla, and L. Sasaki.

Finally, I wish to thank my mother, Emily, for her encouragement, and my wife Catherine for her encouragement, aid and support.

TABLE OF CONTENTS

1.0	Introduction	1
2.0	Related Work	5
2.1	CMU-DA	5
2.2	MacPitts	10
2.3	Mimola	11
2.4	Organick	12
3.0	Overview of Hardware Translator System	14
3.1	Overview of Proposed Complete Translator	15
3.2	Prototype Allocator	18
3.3	Introductory Example	24
4.0	Hardware Description in Ada	32
4.1	Algorithmic Hardware Description	32
4.2	Software Languages for Hardware Description	35
4.3	Methods for Describing Hardware in Ada	37
4.3.1	Signal Typing	38
4.3.2	Hierarchy	40
4.3.3	Concurrent Processes	41
4.3.4	User Defined Hardware Operators	44
4.3.5	Data Input/ Output	45
4.3.6	Timing Constraints	46
4.4	Hardware Interpretations of Ada Constructs	48
4.4.1	Program Modularity	49
4.4.2	Dynamic tasking	50
4.4.3	The Rendezvous	50
4.4.4	Shared Variables	53
4.4.5	Explicit Control Constructs	53
4.4.6	Exceptions	53
4.4.7	Generic Program Units	54
4.5	Functional Simulation in Ada	55
4.6	Example Ada Hardware Descriptions	56
5.0	Flow Graphs and Graph Grammars	72
5.1	Flow Graphs as an Intermediate Level Format	72
5.2	Graph Grammar Formulation of Hardware Design	76
5.3	Representation of Design Information as Productions	79
5.3.1	Data Flow Productions	80
5.3.2	Control Productions	83
5.3.3	Production Identification and Application	84
6.0	Cost Based Production Selection	87
6.1	Terminal Production Cost	88
6.1.1	Cell Cost	88
6.1.2	Control cost	90
6.1.3	Interconnect Cost	91
6.2	Nonterminal Production Cost	96

6.3	Production Selection	98
7.0	Scheduling Operations into Timeslots	102
7.1	Two Classes of Timing Specification	102
7.2	Timeslot Selection as a Scheduling Process.	103
7.3	Weighing the CDFG	107
7.4	Assignment of Operators to Timeslots	110
7.5	Realization of Control Flow Constructs	115
7.5.1	Conditional Branches	116
7.5.2	Loops	117
8.0	The Circuit Graph	120
8.1	Building the CG	120
8.2	Selection of Operator Cells	122
8.3	Manual Addition of Hardware	125
9.0	Implementation and Evaluation	127
9.1	Prototype implementation	127
9.2	FIR Filter Example	128
9.3	TDM Switch	140
9.4	Complexity Analysis	157
9.5	Testing	168
10.0	Conclusions	169
10.1	Summary	169
10.2	Suggestions for Further Work	172
	Bibliography	174
	APPENDIX 1: Description of Elf Data Structures	177

LIST OF FIGURES

Figure	Title	Page
2.1-1:	Block diagram of the CMU-DA system	6
3.1-1:	Block diagram of Elf compiler	16
3.2-1:	Overview of Elf allocation algorithm	19
3.2-2:	Sample graph grammar productions	22
3.3-1:	Ada Code for temperature controller	25
3.3-2:	Initial CDFG with node weights	26
3.3-3:	Final CDFG with timeslot and register assignments	29
3.3-4:	Circuit to realize temperature controller generated by Elf without initialization	30
3.3-5:	Circuit to realize temperature controller generated by Elf with increased clock frequency and user specified ALU.	31
7.4-1:	Timeslot boundary viewed as a cut across CDFG	111
9.2-1:	Ada code for voice frequency FIR filter	129
9.2-2:	CDFG for voice frequency FIR filter	130
9.2-3:	Circuit automatically generated to realize voice frequency FIR filter	131
9.2-4:	Ada code for high frequency FIR filter	134-135
9.2-5:	CDFG for FIR filter with inner loop unwound	136
9.2-6:	Circuit automatically generated to realize high frequency FIR filter	137
9.2-7:	CDFG of FIR filter which can be executed in one clock cycle	138
9.2-8:	FIR filter pipeline generated by Elf	139
9.3-1:	Block diagram of PBX system	142
9.3-2:	Frame format for A-bus and B-bus	143
9.3-3:	Decomposition of TDM switch into two tasks	145

LIST OF FIGURES (cont.)

9.3-4: Ada code for TDM switch	146-150
9.3-5: Circuit generated by Elf to realize task LINE_INT	153
9.3-6: Circuit from Fig 9.2-5 with unused interconnect removed	154
9.3-7: Circuit generated by Elf to realize task CONTROL	155
9.4-1: Pseudo-code for prototype allocator	158
9.4-2: Increase in execution time of algorithm with increasing description size	166
9.4-3: Increase in execution time of algorithm with increasing circuit size	167

1.0 INTRODUCTION

The cost of designing an integrated circuit is recovered by increasing the price of the final product. Recently, the percentage of the final cost attributed to the design phase has been increasing for several reasons. Advances in IC fabrication techniques have reduced the production cost per transistor, and thus, the cost of production relative to design. Further, these processing advances allow larger circuits to be designed. However, the cost per transistor of designing a circuit increases as the size of the circuit increases; i.e., the design cost increases more rapidly than the circuit size. Finally, as circuits become larger, they also become more specialized. Thus, the design costs must be amortized over a smaller production run. New design techniques must be developed to reduce the design cost before the potential of VLSI can be fully realized.

Two methodologies recently developed to reduce the design time are macro cells and array logic. Macro cells are large functional blocks such as four-bit controllers which can be used as building blocks in a larger design. The introduction of macro cells in VLSI designs is analogous to the move from SSI to MSI components that occurred in circuit design over a decade ago. It does not change the design process, but merely provides the designer with more powerful building blocks.

The second approach, array logic, actually alters the steps in the design process. Instead of reducing the logic equations to minimize the number of gates, the description is left in a sum-of-products form which is directly realized by a programmable logic array (PLA) or storage logic array (SLA). The regularity of the PLA structure reduces the complexity of the layout problem so that layout can easily be automated, further reducing the design time.

Both of the techniques mentioned above simplify the design of IC's, but the hardware realization of the desired function is still completely defined by the designer. For example, in both of these design methods, the designer explicitly specifies in which register the result of an operation is stored; it is the designer's responsibility to remember what variable is stored in each register at any time. Thus, these techniques are analogous to assembly level programming in that the designer specifies not only the algorithm, but also the realization of that algorithm.

Software programming has been further simplified by allowing the designer to specify the desired algorithm in a high level language (HLL) and leave the details of the implementation to an HLL compiler. In this functional design approach, the designer specifies the function to be realized, such as the assignment of the result of an operation to a variable. The compiler then automatically decides which register the variable should be stored in. A similar approach, the use of functional hardware description

languages and "silicon compilers", promise to simplify hardware design.

The use of HLL's for software programming has demonstrated several advantages of this approach. First, the programmer is allowed to concentrate on the algorithm to solve the problem; implementation details are left to an automatic compiler. This reduction in complexity leads to fewer programming errors, easier debugging, and thus, to reduced design time and cost. The absence of details in the HLL description also increases the understandability of the program. This is especially important when more than one person is working on the design, or when the design is left to another programmer for maintenance. Secondly, while the execution time of a program may vary from computer to computer, its function is machine independent. Thus, once written, a program can be compiled on several different machines. The purchase of a new computer simply requires recompilation of the source code, not rewriting of the program. Finally, this machine independence, coupled with the widespread usage of a few main languages, allows designers to learn computer languages at university and use them wherever they go.

The benefits of hardware description languages (HDL) for hardware design are analogous to those of HLL's in software programming. HDL's allow hardware designers to concentrate on the function of the circuit, leaving the logic design to a hardware translator. As with programming, this

leads to increased understandability, fewer errors, and decreased design time. An HDL description of a circuit is relatively technology independent. Thus, cost reductions can be achieved by recompilation of the HDL description in a new technology, rather than a complete redesign. Wide acceptance of a few HDL's would allow them to be taught as programming languages are now. Finally, HDL compilers open the possibility of designing hardware and software in a common language. This not only simplifies the verification of a system design, it also allows the designer to postpone the partitioning of the system into hardware and software modules to a later stage in the design cycle.

This thesis presents an overview of a hardware translator capable of designing standard cell based circuits from algorithmic descriptions written in a subset of the Ada programming language. Methods of describing hardware in Ada are discussed and hardware interpretations of special Ada constructs are presented. An algorithm is developed for automatically generating a circuit to realize the Ada functional description using graph grammar and scheduling techniques. A prototype implementation of this algorithm has been generated and initial results are presented.

2.0 RELATED WORK

The feasibility of hardware compilers has been demonstrated at several universities. As with HLL compilers for software, these prototype compilers produce less efficient designs than human designers. However, the reduction in design time and cost tends to offset the increase in unit cost of the final IC. Thus, HDL compilers are more cost effective than they appear. Several prototype compiler systems are described in the following sections.

2.1 CMU-DA

The Carnegie-Mellon University Design Automation (CMU-DA) project is the most advanced general hardware translator system reported to date [ThoD83, KowT84, NagA82, HafL82, LieG80, BarM78, McFM78, SnoE78]. The CMU-DA system generates a cell connectivity list (netlist) from a functional description of the circuit. The desired function of the circuit is described in ISP, a language developed for describing computers. ISP allows the designer to describe parallel processes, as well as sequential and parallel operations within a process (by explicitly specifying state transitions) [BarM78]. The bias towards computers introduced by choosing ISP as a description language has carried throughout the CMU-DA project. Most of the algorithms are slanted towards generating good computer architectures and

Fig. 2.1-1: Block diagram of the CMU-DA system

published examples are predominately CPU and microcomputer designs.

The basic data-flow of the CMU-DA system is shown in Fig. 2:1-1. The ISP description is first translated onto a "Value trace" (VT) [DirS81, SnoE78]. The VT is a data-flow graph of the description with control constructs added and is a common intermediate level form used for functional optimization, design style selection and allocation. The VT is functionally optimized using techniques similar to optimizing compilers. This includes the removal of redundant operations, moving code out of loops, and the elimination of code which can never be called. After functional optimization, a technology independent data path is designed using one of the data path allocators. A module binder then selects a technology dependent realization (ie cell) for each module in the design. Once the data path design is complete, a control allocator designs a microcode sequenced control path to switch the data path so as to realize the algorithm. Data path control consists of programming circuit elements to realize the desired function (eg. set ALU to perform addition), selecting the active input of a multiplexer, and specifying when a register loads the data on its input port. The control allocator also optimizes the control word format to minimize the size of the control memory [NalA82].

The first data path allocator in the CMU-DA system was written in 1978 [HafL78]. This allocator performed a

one-to-one mapping of the value-trace onto a circuit net-list. A cell was added to the net-list to realize each operation in the ISP description. This correspondence of structure and function effectively reduces the VT to a structural description. The CMU-DA system currently contains two allocators. One uses a deterministic algorithm, the other is a heuristic based expert system. The latest deterministic allocator designs the circuit in several passes [ThoD83]. The first pass assigns operations into timeslots. The algorithm used for this assignment attempts to minimize the number of operations of each type in a timeslot and the number of registers used to store intermediate results. The operations within each timeslot are then assigned to technology independent modules. This step includes interconnecting the modules. Finally, technology dependent cells are chosen to realize each module.

The second allocator, the design automation assistant (DAA); is based on expert system techniques [KowT84, ThoD83]. The system is written using a general knowledge-based expert system (KBES) tool. Rules are used to describe circuit design knowledge. This knowledge may be architectural on a global (use a bus) or local basis (connect outputs of registers storing input values of operation to be realized to ALU inputs). At present, all registers declared in the ISP description appear in the final design. Additional registers are automatically added to store intermediate results as required. The DAA adds computation

elements and interconnect to complete the design. The quality of circuits generated by the DAA program are slightly better than those generated by the deterministic system. However, these circuits were generated with manual assistance in that the results of the first pass were analyzed and new rules were added to generate an improved design. Also, the execution time of the DAA is larger than necessary as the program is written in a general KBES tool (OPS-5), rather than having a decision algorithm specialized for IC design. For example, on a VAX11/750, it took 4 cpu hours to design a 6502 microcomputer and 47 cpu hours to design an IBM 370 datapath.

The main advantage of the CMU-DA approach for functional to structural translation is the introduction of a graph intermediate form, the value trace. Translating the ISP description onto the VT extracts the low level parallelism of the description; i.e., it removes any unnecessary ordering of operations introduced by the syntax of ISP. The VT also provides a convenient representation for functional optimization and functional to structural translation. The control/data-flow graph (CDFG) used in this thesis as an intermediate level description is similar to the VT. However, the allocation process is significantly different as control step assignment and mapping of operations onto technology dependent cells are performed simultaneously.

2.2 MacPitts

The MacPitts system developed at MIT [SouJ83, SusJ82] is specialized for designing circuits with "microprogram sequenced data path operations"; i.e., the data path is controlled by a control word sequencer and the results of operations are either stored or output at the end of the clock cycle. The system includes automatic layout of the designed circuit, and thus, can be considered a full silicon compiler.

To design a circuit using MacPitts, the user writes a register transfer level description of the data path in a Lisp-like language, also called MacPitts. This description can include both sequential and parallel execution of operations. (At present, the operations are limited to simple logic, relational, and additive operators, but new operations can easily be added if cells are designed to realize them.) The first step in the compilation process is to check the MacPitts description for design errors such as trying to load a memory location simultaneously from two different sources. If this semantic check reveals no errors, the description is compiled to a technology independent intermediate level description. At this level, the data path is described as an interconnection of hardware units, while the control requirements are described by Boolean equations and state assignments. A functional simu-

lator which reads this intermediate description is used to validate the design.

The second stage in the compilation process realizes the control logic of the intermediate level description with a Wienberger gate array (a multi-level NOR realization). (PLA's have also been investigated as an alternative realization.) Automatic layout and routing is then performed. The output of MacPitts is an IC layout in CIF format for mask generation.

2.3 Mimola

Mimola [MarP79,ZimG79,ZimG80] is an interactive hardware design system for designing digital processors ("synchronous automata"). Rather than writing a functional description of the processor, the designer writes several algorithms which are representative of the type that will be run on the processor. An initial circuit is automatically generated by allocating enough hardware to perform each state transition in one clock cycle. Statistics indicating the fraction of time that each hardware unit is in use are generated. The designer examines these statistics and imposes hardware limits to reduce the amount of hardware generated (for example, he may limit the maximum number of registers). The allocation step is then repeated and serial processing is substituted for parallel processing to insure that the specified hardware limits are not violated. This

process is repeated until an acceptable design is achieved. The designer can override a portion of the allocation by specifying hardware in the hardware data structure.

The algorithms used to describe the system requirements are written in a dedicated language called Mimola. The language contains several macros for common programming constructs such as subprograms and loops. New constructs can be added with user defined macros. The Mimola description must explicitly define the state transitions of the processor. Each state may contain several parallel operations, but all storage operations are synchronized by clock transitions. This removes the possibility of race conditions or timing ambiguities.

2.4 Organick

The three preceding systems have all required the circuit description to be written in a specialized language. Elliot Organick heads a group at the University of Utah investigating the possibility of using the programming language Ada for hardware description [OrgE84, OrgE82, OrgE82b]. They state several advantages to this choice. Ada contains several constructs desirable for hardware description such as tasks to emulate parallelism. The modularity of the Ada description can be used to guide the allocator. Finally, as mentioned previously, it allows the

possibility of designing a hardware-software system in a common design language.

The present research at Utah is primarily a feasibility study which tries to span the full range from Ada to silicon layout. As such, the realization of the steps in the synthesis process are rather simplistic. For example, an Ada task which performs an accept in a rendezvous (see Section 4.4.3) is automatically realized as a two engine pair: one to manage the queue for the rendezvous and one to perform the algorithm of the task. All inter-task communication is via buses. The first results of this study employed only one rendezvous in which the rendezvous could be reduced to a ready flag because only one task called the entry. The general realizations can similarly be simplified in other cases to reduce the cost of the design (see Section 4.4.3).

Another oversimplification in the Utah approach is to realize all of the hardware as storage logic arrays (SLA's). SLA's are basically PLA's with memory elements, for outputs and feedback paths from these storage locations to the PLA inputs. This makes both hardware design and layout trivial problems, but leads to a larger, slower realization than is often necessary. The following chapters demonstrate how further analysis of the Ada code and a more sophisticated allocator lead to better hardware realizations of the Ada descriptions.

3.0 OVERVIEW OF HARDWARE TRANSLATOR SYSTEM

The translators described in Chapter 2 have demonstrated the feasibility of the HDL approach. This chapter presents an overview of a hardware compiler, Elf, capable of generating a circuit to realize an algorithm described in Ada. This compiler combines several features from the previous systems with a new allocation algorithm and a more complete cost function. The features borrowed from these predecessors include the use of Ada as the description language, the representation of the algorithm as a flow graph, and the separation of the circuit design into data path and control path design.

A new allocation algorithm was developed to better realize Ada circuit descriptions. By employing scheduling techniques, the algorithm can meet system level timing constraints specified by the user at the functional level. The ability to specify timing constraints in the functional description is necessary if the automatically generated circuit is to form part of a larger system. For example, a designer can limit the time allowed for a process to complete a calculation to be 5 usec so that a parallel process can access the result at that time.

The allocation algorithm designs the circuit by selecting rules from a rule library. This rule basis has the same advantages as an expert system approach with respect to adding new rules and changing process technologies. The use

of a specialized rule selection algorithm greatly reduces the execution time of the allocation process as compared to a general rule selection algorithm.

The rules to be applied are selected based on a cost calculated for each rule. This cost function includes the actual cost of the rule in the chosen technology, allowing the allocator to modify the circuit to reflect the different difficulty of performing operations in different technologies. The cost function includes an estimate of the interconnect and control overhead of applying a rule by increasing the cost of applying the rule proportional to the area of the interconnect and control memory which would have to be added. In addition to selecting rules which require less interconnect, this allows the algorithm to replicate active circuitry to save interconnect, an approach used to save area when manually designing integrated circuits.

3.1 Overview of Proposed Complete Translator

A block diagram of the complete Elf hardware compiler is given in Fig. 3.1-1. An algorithmic circuit description is first written in Ada [Ada82] (block A, see Chapter 4). This description can be compiled and executed on a general purpose computer to verify the correctness of the description (B). When the designer is satisfied with the Ada code, hardware compilation can proceed. The Ada code is first translated onto an intermediate level description, the

Fig. 3.1-1: Block diagram of Elf compiler.

control/data-flow graph or CDFG (C). The CDFG is a graph of the data-flow of the algorithm with explicit control constructs (see Chapter 5). The CDFG is analogous to the value trace (VT) in the CMU-DA system [DirS81] and presents the same advantages although its data format is completely different. Translation of the Ada code onto the CDFG is a relatively direct process as discussed in Chapter 5. The CDFG can be functionally optimized (D) either as the VT is in the CMU-DA system [SnoE78] or as in a graph grammar based optimizing software compiler [NagM81].

An initial design of the data path which meets all user specified timing constraints is then generated as described in the next section (E,F,G). The general data path architecture is a microsequenced register-mux-operator construct which can include both multicycle operations and user designed operator cells (K). By specifying extreme allocation parameters and postprocessing to remove registers, the allocator can also generate data-flow architectures and thus certain combinatorial logic designs.

The circuit design is optimized (H) and then completed by generating a control section similar to that in the CMU-DA system (I) [NagA82]. Finally, statistics on cell and memory utilization and operator realization cost are gathered (J) which allow the designer to guide the allocator to generate a better design (if required).

3.2 Prototype Allocator

The allocator translates the algorithmic circuit description represented by the CDFG (see Fig. 3.3-2, for example) to a standard cell net-list or circuit graph (CG) (Fig. 3.3-4). The allocator uses a modified urgency scheduling technique and design information represented as graph grammar productions to concurrently schedule operations into timeslots and select realizations for operations. The allocation algorithm is strictly additive in that cells and interconnect are automatically generated, but once added to the net-list, they can only be removed manually or by post-processing. The prototype allocator operates in automatic, manual or interactive modes and may be initialized with a designer specified starting circuit.

An overview of the allocation algorithm is presented in Fig. 3.2-1. The first step is to read in and preprocess the cell and production libraries. Then, the CDFG of the algorithm is read and preprocessed. This preprocessing included assigning an urgency weight to each node in the CDFG. The weight of a node is defined to be the minimum number of clock cycles required to execute that portion of the CDFG from the node to the corresponding timing constraint (see Fig. 3.3-2). A node which is enclosed by more than one timing constraint is assigned a list of weights, one corresponding to each timing constraint. This preprocessing stage also identifies all nodes which have no inputs. Such

- Read in and process libraries
- Read in and preprocess CDFG
- Find initially feasible operations
- While feasible operations
 - . Choose and apply nonterminal productions
 - . Select terminal productions and assign cells
 - . Assign registers
 - . Realize control nodes
 - . Update feasible operations
- Print allocation results

Fig. 3.2-1: Overview of Elf allocation algorithm

nodes are termed "initially feasible" as they are feasible candidates for execution at the start of the algorithm.

The allocation then enters a loop which is performed for each timeslot. A subset of the feasible nodes is selected for execution in this clock cycle by comparing the node weight to the number of cycles left before the enclosing timing constraint is violated. The cost of realizing each selected node with all applicable rules is then calculated. A greedy algorithm with pairwise exchange is used to select which rule to use to realize each operator node. The outputs of the operations are assigned to available registers using a similar cost-based decision. After these assignments are complete, any feasible control nodes are processed and the list of operators feasible for execution in the next timeslot is determined. If there exist feasible nodes at the end of the cycle, the timeslot count is advanced and the process repeated. No feasible operations at the end of the cycle indicates that the allocation is complete.

The "knowledge" that the allocator uses to translate the CDFG to the CG is represented as graph grammar productions for operator nodes and as hard coded procedures for control constructs. This allows all technology dependent data to be contained in a rule library. Thus the allocator code does not have to be modified with each change in technology. Since the input and output of the allocator are

graphs, a graph grammar formulation for the rules was appropriate.

Graph grammars are an extension of string grammars [Clav79]. The CDFG represents the start graph. The words in the language correspond to all possible mappings of the CDFG onto circuits which can be generated by starting from the CDFG and applying productions in the libraries. Nonterminal productions replace single nodes or subgraphs in the CDFG which match the left hand side of a production with functionally equivalent nodes or subgraphs which form the right hand side of the production. Terminal productions are used to map functional operations in the CDFG onto cells in the CG (see Fig. 3.2-2).

When more than one production may be applied at a node in the CDFG, the selection of which production to apply is made based on a cost calculated for each production. This cost represents the increase in circuit cost if the production is applied. For a terminal production, the cell portion of the cost is computed by calling a user specified cost function with cost parameters contained in the cell description. At present, the cost parameters are cell area and number of control bits. (The latter is used to estimate the cell control overhead.) The cell cost of existing cells is 0 since using an existing cell adds no active cell area to the circuit. However, interconnect may have to be added to connect the existing cell to the registers containing the inputs to the operation. The allocator estimates the amount

left hand side

right hand side

subgraph $\rightarrow$ subgraph

subgraph $\rightarrow$ single node

single node $\rightarrow$ subgraph

single node $\rightarrow$ single node

single node $\rightarrow$ cell

Fig. 3.2-2: Sample graph grammar productions.

of additional interconnect required based on global and local considerations of the circuitry allocated to date. This estimate includes any increase in mux area and mux control and is added to the cell cost to obtain the total production cost. This cost formulation allows the allocator to add more than one instance of a cell to reduce the total amount of interconnect.

The cost of a nonterminal production is defined to be the minimum cost of realizing the right hand side minus the minimum cost of realizing the left hand side where right and left are as defined in Fig. 3.2-2. The cell cost of each side is the sum of the minimum cell area added for each node on that side of the production. The interconnect area required is estimated by multiplying the number of connections in the production by the average area of interconnect added per functional connection in the past 10 clock cycles. A negative cost for a general production indicates a lower cost design and the production is applied.

The last cost calculated by the allocator is the cost of storing the results of operations into different registers. After each operation in a cycle has been assigned to a cell, the storage cost is calculated for each free register. The storage cost is determined by comparing the operations realized by cells connected to the output of each register with the operations in the CDFG which access this result.

When the allocator encounters a control node such as a loop or conditional, it calls a corresponding procedure to realize the node. Since the allocator assumes a microsequenced architecture, the realization of most control constructs is reduced to microcontroller jump statements. The only data path consideration is to insure that corresponding outputs of loops and branches are stored in the same register. This is easily accomplished by constraining the storage of the node which generates the result or by performing a register transfer.

3.3 Introductory Example

Fig. 3.3-1 is an example of the Ada code for a simple oven controller (which forms part of a larger package). Temperatures are repeatedly read and processed. Flags are updated to turn the heater on or off and to signal that the oven temperature has stabilized. The code is written to pass implicit knowledge of the characteristics of the oven control to the allocator. For example, T1 and T4 are known to always be lower than T2 and T3. Thus, the calculation of the temperature difference reduces to $(T2+T3) - (T1+T4)$. The procedure display is specified as being realized directly by a hardware cell. Thus, the code in the procedure is ignored by the allocator and has been written to output data for simulation purposes.

```

type TEMP is INTEGER range -128 .. 127;
type STATUS_CODE is INTEGER range 0 .. 127;

HEAT_OFF_CODE : constant STATUS_CODE := 103;
HEAT_ON_CODE   : constant STATUS_CODE := 57;
HEAT_STATUS    : STATUS_CODE := HEAT_OFF_CODE;

task body OVEN_CONTROL is

    T1,T2,T3,T4,T_SET : TEMP;

    procedure DISPLAY ( T : in TEMP ) is
        pragma HARDWARE_CELL;
    begin
        put ( TRACE_FILE, T );
    end DISPLAY;

begin
    loop
        TIMING.REFERENCE (REF_LABEL => BEGIN_LOOP);
        GET ( TEMP_PORT, T1 );
        GET ( TEMP_PORT, T2 );
        GET ( TEMP_PORT, T3 );
        GET ( TEMP_PORT, T4 );
        GET ( TEMP_DIAL, T_SET );
        T_AVG := ( T1 + T2 + T3 + T4 ) / 4;
        T_DIFF := ( T2 + T3 ) - ( T1 + T4 );
        T_ERROR := ( T_SET - T_AVG );
        OVEN_READY := (T_ERROR<1) AND (T_DIFF<2);
        TIMING.CONSTRAINT (REF_POINT => BEGIN_LOOP,
                           MAX_TIME => 25E-6,
                           REF_LABEL => END_CALC );

        DISPLAY ( T_AVG );
        PUT ( STATUS_PORT, OVEN_READY );
        if T_SET > T_AVG then
            HEAT_STATUS := HEAT_OFF_CODE;
        else HEAT_STATUS := HEAT_ON_CODE;
        endif;
        TIMING.CONSTRAINT (REF_POINT => END_CALC,
                           MAX_TIME => 20E-6,
                           REF_LABEL => END_DISPLAY);
        TIMING.CONSTRAINT (REF_POINT => BEGIN_LOOP,
                           MIN_TIME => 10E-3,
                           MAX_TIME => 11E-3,
                           REF_LABEL => REPEAT_TIME);

    end loop;
end OVEN_CONTROL;

```

Fig. 3.3-1: Ada code for temperature controller

Fig. 3.3-2: Initial CDFG with node weights.

Fig. 3.3-2 represents the CDFG corresponding to the oven control. The nodes in Fig. 3.3-2 represent operations in the algorithm. Solid edges represent values which must be stored in registers. The dashed lines represent precedence relations; i.e., an ordering of operations not constrained by value edges but implied by the Ada code. The node weights corresponding to two of the timing constraints are given on the left side of the diagram.

An allocation with no initialization was performed using a cell library containing 4 adders, 2 subtractors, a shifter, a comparator, an ALU and a shared-variable arbiter. — The production library contained eight nonterminal productions. The resulting CDFG and CG are shown in Fig. 3.3-3 and 3.3-4 respectively. (Note that the variable HEAT_STATUS is declared outside the controller task and thus is treated as a shared variable.) The numbers on the left in Fig. 3.3-3 represents the timeslot that the operation is performed in.

Because of the timing constraints, it is impossible to perform the algorithm on a single processor with the minimum clock frequency. Actually, using the minimum clock frequency requires multiple adders comparators and/or subtractors because of the precedence relations. Thus, the results shown in Fig. 3.3-3 and 3.3-4 were generated by increasing the clock frequency 20%. If the clock frequency is further increased and the allocator "seeded" with a start circuit consisting of a single ALU (no registers, no connections),

the allocator generates the circuit shown in Fig. 3.3-5.
(The comparator is necessary as the ALU given could only realize addition, subtraction and shifting.)

Fig. 3.3-3: Final CDFG with timeslot and register assignments.

Fig. 3.3-4: Circuit to realize temperature controller generated by Elf
without initialization

Fig. 3.3-5: Circuit to realize temperature controller generated by EIT with increased clock frequency and user specified ALU.

4.0 HARDWARE DESCRIPTION IN ADA

The Elf system is aimed at creating specialized hardware to realize fixed algorithms. In this context, circuit descriptions closely resemble software programs. This led to an investigation of the possibility of using an existing programming language, Ada [Ada82], as a HDL rather than generating a new, specialized HDL.

This chapter discusses the requirements for a HDL used as a source language for hardware compilation and discusses how Ada satisfies these needs. Methods for specifying hardware functions and timing constraints within the syntax of Ada are presented. Constructs in Ada which are costly to realize in hardware are identified. A set of rules for translating an Ada circuit description onto the CDFG intermediate level form are discussed in Chapter 5.

4.1 Algorithmic Hardware Description

Viewed as a black box, a hardware circuit transforms a sequence of inputs into a sequence of outputs. The goal of algorithmic (or functional) hardware description is to unambiguously define an algorithm corresponding to this transformation. The description must define the input and output format of the circuit to insure that the circuit will interface correctly with the rest of the system. The accuracy of

internal calculations may be specified, but the internal structure (or architecture) of the circuit is not defined.

As an example, consider the description of a finite impulse response filter. The algorithm can be defined as a loop in which the delayed signal is multiplied by the filter coefficient corresponding to the delay, and the product added to a running sum. The description would also indicate that the input and output signal are analogue voltages, while the filter coefficients are represented by 8-bit two's-complement integers. However, the multiplication of the signal by the coefficient could be performed directly using a multiplying digital-to-analogue converter, by an analogue multiplier if the corresponding coefficient is first converted to a voltage, or totally digitally after converting the analogue signal. Further, all multiplications may be performed sequentially on one multiplier, or there may be a dedicated multiplier for each multiplication (eg. pipeline). It is only required that the performance criteria specified in the description are met.

This objective of capturing an algorithm without defining the hardware structure is related to the concept of using high level languages for software programming. However, most software programming languages (SPL's) implicitly assume a general purpose computer with a large, dynamically reassignable memory. The control constructs included in SPL's are data-processing oriented and the programmer has no control over how they are implemented. Hardware design

consists not only of defining the algorithm, but also the architecture that executes the algorithm. Further, hardware designers frequently design new cells to realize special functions. Thus, hardware design is much less restricted than software in both the data processing and the control flow aspects of the design. The removal of the underlying architecture in hardware design means that HDL's must be declarative enough to define the hardware requirements and flexible enough to accomodate new hardware cells and different circuit architectures.

Hardware naturally subdivides into a number of concurrent processes which communicate using a wide variety of methods. An algorithmic HDL must be able to describe and simulate this high level parallelism, including the explicit and implicit synchronization and communication between the processes. The description should be hierarchical to follow the design's development through a structured design strategy (eg. top-down).

Finally, hardware design is constrained not only by the algorithm performed, but also by the amount of time available to perform it. A HDL must contain a method of specifying and simulating both latency and throughput timing constraints.

4.2. Software Languages for Hardware Description

The basic capture of the algorithm realized by a hardware circuit resembles a generalized form of software programming. A common approach for developing a HDL is to modify the syntax of an existing SPL. This creates a superset of the SPL which contains constructs not recognized by the original SPL.

Utilizing the procedural nature of modern SPL's to extend the features of the language within the existing syntactic and semantic limits of the language has several advantages. First, using an existing programming language, or a subset thereof, rather than generating a specialized HDL (or superset SPL) allows the software language to act as a functional simulator. The hardware description is simply compiled and run on a general purpose computer to verify the circuit description. Second, the hardware and software sections of a design can be written in the same language. This not only simplifies the verification of a system design, it also allows the designer to postpone the partitioning of the system into hardware and software modules to a later stage in the design cycle. Third, designers experienced in writing software in the SPL will not have to learn a new language to use the system. However, different programming techniques may have to be learned in order to generate efficient designs. Finally, the designer can

utilize the SPL environment (interactive debugger, syntax checker, etc) to generate the description in less time.

While the use of a SPL with a structured programming style may increase the length of the source description, it also makes the code much more understandable. Further, this extra length is only in the source code; it does not necessarily reduce the efficiency of the final design. (If succinct descriptions are desired, a library of procedures can be generated which act as a shorthand notation for common hardware primitives.)

The main disadvantage of using SPL's for algorithmic hardware description is that some SPL constructs cannot be efficiently realized. Software is intended to run on a computer with large, fixed resources which can be dynamically allocated to different processes at different times during the execution of the program. Each process has access to all the resources of the computer (eg. hardware multiplier). Resource sharing also increases the effective size of available resources. For example, all stacks and queues may share a common stack space. This makes the maximum possible depth of an individual stack or queue very large; although not all stacks and queues can be large at the same time.

Efficient hardware realizations are generated by designing each hardware unit with the minimum amount of resources to achieve a specified algorithm. The operations which must be realized by the data path are explicitly

defined by the SPL description. However, the amount of memory required is not. For example, it is not always possible to determine the maximum length of any one stack, much less the combined maximum length of a number of stacks. The simplest solution to this problem is to exclude from the description language any constructs which imply dynamic allocation. This is the approach taken in most HDL's. A more flexible approach is simply to allow potentially inefficient realizations, but allow the designer to specify constraints. For example, the design may include structures like stacks provided that the designer specifies the maximum stack depth, and provided he is willing to accept the associated increase in hardware.

4.3 Methods for Describing Hardware in Ada

Several software languages were investigated as HDL's. Most were rejected because of their inability to describe and simulate concurrency. It was decided that a language which had a standard form was a better choice than a language with a standard core with implementation dependent additional features. This led to choosing a subset of Ada [Ada82] for our system.

The basic data flow of a single hardware process at the algorithmic level is captured directly by an Ada program body. Data operations in the description correspond to operations the hardware must perform. The results of oper-

ations represent values to be stored in internal registers and define an order in which the operations must be performed. This portion of algorithmic description is relatively simple and could be achieved in several SPL's. The advantage of Ada is in its capacity to meet the additional requirements mentioned in Section 4.1.

Parallelism is easily modelled with Ada tasks. While the rendezvous is the only way to synchronize tasks, all hardware synchronization mechanisms can be modelled via the rendezvous as discussed below. Tasks and packages combine to support a top-down design strategy. Data typing provides a mechanism for specifying bit requirements of variables. Finally, the Ada pragma provides a mechanism by which the designer can identify new operations (see Section 4.3.4).

The use of Ada as a HDL is also being investigated at the University of Utah [Orga84]. Their present research is primarily a feasibility study which attempts to span the full range from Ada to silicon. As a result, they realize most Ada constructs in a direct fashion with storage logic arrays (SLA's).

This section takes a closer look at how hardware can be described in Ada. Section 4.4 presents hardware interpretations for several special Ada constructs.

4.3.1 Signal Typing

Hardware description languages require signal typing

for several reasons. Typing serves to define the signal representation: analogue vs. digital, two's-complement vs. sign-and-magnitude, integer vs. real. The signal type limits the cells used to realize an operation and to store its result. For example, addition of two analogue signals could be performed using an op-amp while digital addition would be realized on a digital ALU. Similarly, an analogue signal would be stored on a sample-and-hold while a digital signal would be stored in a register of appropriate width. As with data typing in software programming, signal typing permits automatic checking of signal compatibility thus reducing the possibility of errors.

Ada contains strong typing and user definable data types to allow detailed description of hardware signals. For example, a type `ANALOGUE_VOLTAGE` can be defined as a subset of the real numbers with appropriate range limits. The automatic type checking will prevent connecting an signal specified as an `ANALOGUE_VOLTAGE` to a digital real number or to a signal of type `ANALOGUE_CURRENT`. Similarly, digital signal compatibility can be checked by defining different types for different digital formats (32-bit real, 8-bit two's-complement integer, one-bit Boolean). Finally enumerated types can be defined in Ada to deal with signals whose representation has not yet been defined. For example, the set of valid states for a traffic controller finite-state machine could be defined as (red, yellow,

green) without the bit representations for the states being specified.

4.3.2 Hierarchy

As hardware systems increase in size, there is an increased tendency towards structured hardware design. To be effective, an algorithmic HDL must be able to track the design from concept, through the stages of refinement, towards a finished design. Ada naturally tracks a structured design through a combination of program stubs and the compatibility of procedure and task entry calls.

Program stubs specify the externally visible elements of a package or task and can be defined before the code has been written. This allows the designer to estimate the interprocess communication in a top-down design decomposition, or to define the user interfaces in an edges-in design.

The use of a single calling syntax for procedures and task entries allows the designer to refine the description by replacing tasks with packages which may contain multiple internal tasks. The external calling syntax is maintained by creating an interface procedure in the package corresponding to each externally visible task entry. The designer can check that the refinement of the description has not introduced errors by running the old and new block descriptions in the same context. This hierarchical facili-

ty allows several designers to refine different blocks in parallel and helps save computer time during verification.

4.3.3 Concurrent Processes

Concurrency can be thought of at two levels: the high level concurrency of two relatively independent processes and the low level concurrent execution of two operations within a single process. Low level concurrency within an Ada task can be extracted from the Ada description as discussed in Section 5.1. High level concurrency is naturally represented by multi-tasking.

Each high-level concurrent process is identified as a separate Ada task. The code within the task represents the algorithm of that process. In hardware, as in software, the parallelism represented by multi-tasking may be realized as virtual concurrency via time sharing, or with physical concurrency via separate hardware. Hardware design also allows a hybrid realization in which two tasks are realized simultaneously on a shared data path as would typically be done for low-level concurrency. The designer must only insure that the execution time constraints are met.

Communication between parallel tasks is modelled using the Ada rendezvous. The rendezvous is the Ada mechanism for synchronizing two tasks. One task initiates the rendezvous by calling a named entry in the other task. The calling task is placed on a queue associated with the entry and its execution is suspended. When the called task executes an

accept of an entry, the task at the top of the queue passes the rendezvous "in" parameters and the code of the critical region is executed. At the end of the critical region, the "out" parameters are returned to the calling task and both tasks resume execution independently. If there are no tasks queued on an entry, the accepting task waits for one to arrive. Both entry calls and accepts may be conditional with alternate functions executed if the rendezvous cannot begin immediately.

Initially, the rendezvous' ability to model the wide variety of hardware communication mechanisms was questioned. The rendezvous represents a specific protocol which includes temporal queueing of tasks on entries. However, the basic call-accept structure of the rendezvous naturally fits the basic hardware mechanism of outputs driving inputs. More specific hardware protocols can be modelled with procedures containing (multiple) rendezvous. Although the rendezvous would appear to have a high overhead, it can be realized in hardware reasonable efficiently using counters (see Section 4.4.3). Further, simple realizations can be used in special cases, such as when only one task may call an entry.

The simplest hardware communications method is a direct connection between two combinatorial functions. When the output of the first block changes, the second determines its new output. The output change action is modelled in Ada by an entry call. The Ada code corresponding to the description of the second cell is embedded in a loop which

accepts the input change, determines the next output, notifies the connected cells and then waits on the accept for further changes. Parallel inputs are modelled with an accept statement for each input enclosed in a single select statement. (To prevent deadlock, one can decompose each process into two tasks, one to monitor the inputs and one to notify other tasks of a change in output.)

The rendezvous can be placed in loops and/or select statements to model hardware communication involving synchronization or communications protocols. For example, a microprocessor interrupt is acknowledged only at the start of a (multi-cycle) control step. If there is no interrupt, the next instruction is decoded. This can be modelled in Ada by the following code:

```
loop
  select
    accept interrupt
      do interrupt_service_routine;
    or when interrupt'count = 0 => -- no interrupt
      do decode_next_instruction;
    end select;
  end loop;
```

The use of conditional accepts within a select can be used to directly model fixed priority arbitration schemes. For example, a three stage daisy chain is modelled as:

```

select
  accept high do ... ;
or when high'count = 0 => -- high queue is empty
  accept medium do ... ;
or when high'count = 0 and medium'count = 0 =>
  accept low do ... ;
end select;

```

More complicated protocols can be described by writing call and receive procedures containing multiple rendezvous to perform the appropriate handshaking. This serves to simplify the description. This can also serve to simplify the realization if a hardware cell is specified as realizing the protocol procedure. Direct realizations are identified by "pragmas" as discussed in the next section.

4.3.4 User Defined Hardware Operators

Software is compiled and executed on predefined hardware. Thus, the programmer has a well defined set of operations which he can include in a program. Hardware designers regularly design new cells to realize special functions or to meet performance criteria. The decision to design a new circuit is often made at an early stage in the design process. In these cases, this decision should be identified in the algorithmic description.

The description of new functions is accomplished in Ada by defining a procedure for the new cell. The internal code of the procedure serves to describe the function for simulation purposes. The procedure is identified as having a direct realization by including the pragma "HARDWARE_CELL"

in the procedure body (see Fig. 3.3-1). The pragma is an Ada construct which allows the programmer to guide the compilation process from the source code. If a pragma is not recognized, the language specifies that it is to be ignored. Thus, we can have one set of pragmas for software compilation and another, possibly intersecting, set of pragmas for hardware compilation.

4.3.3 Data Input/ Output

As with signal typing, hardware circuits have a wide variety of input and output formats. Ada has no special I/O instructions. Rather, all I/O is performed by packages written within the language.

This philosophy also works in hardware description. All hardware I/O primitives are defined by writing corresponding I/O procedures and using a pragma to identify the correspondence. For example, input and output through latched ports is modelled by the basic get and put operations present in the package TEXT_IO. Similarly, to use an LED display driver cell, the user would create a procedure LED_DISPLAY. In the procedure code, the pragma HARDWARE_CELL would be used to indicate that there is a user supplied cell which realizes this operation. For simulation purposes, the procedure body is written to write the displayed value into a trace file (see Fig. 3.3-1).

4.3.6 Timing Constraints

In hardware, the time required to perform a task is often critical to the correctness of the result. For example, in communications networks, data must be transmitted and received at the same frequency. In process control, an error condition must be serviced within a fixed time delay. Such timing constraints must appear in the algorithmic description and must be properly accounted for during functional simulation.

Ada contains only a limited form of timing specification, the delay statement. Calling for a delay simply suspends execution at a particular point in the program for the specified time. While this can be used to model latency timing constraints, using the delay statement to specifying timing is ambiguous as there is no explicit indication of which operations are executed in the delay period. Further, the delay statement provides no mechanism for specifying throughput constraints.

A better method of specifying timing is achieved by introducing a calendar package (or task), TIMING. The package has three entry procedures: REFERENCE, CONSTRAINT, and PERIOD. The approach described below is syntactically correct in Ada. Further, the calendar package can be written to properly model the implicit synchronization introduced by timing specifications (see Section 4.5).

The amount of time allotted for the execution of a code segment (i.e., latency) is defined by calls to the REFERENCE.

and CONSTRAINT entries. A timing reference point is first created at the start of the code segment with the call:

TIMING.REFERENCE (REF_LABEL)

where REF_LABEL is a unique identifier which is used to reference this statement. A timing constraint is then set at the end of the code segment with the call:

TIMING.CONSTRAINT (REF_POINT, MIN_TIME, MAX_TIME, REF_LABEL)

REF_POINT is the identifier of the statement at the start of the code segment to be constrained. MIN_TIME and MAX_TIME define the time interval in which the code segment is to be executed; i.e., time delay from passing REF_POINT to crossing constraint statement. Placing a REF_LABEL on a timing constraint allows it to serve as a reference point for a future timing constraint.

Throughput specifications are required for many data and signal processing applications. In these situations, an algorithm is repetitively executed at a specified rate. In an Ada description of such an algorithm, the repetitive nature is represented as a (continuous) loop. The rate of execution is specified by specifying a repetition time for the loop:

TIMING.PERIOD (MIN_PERIOD, MAX_PERIOD)

where MIN_PERIOD and MAX_PERIOD represent the minimum and maximum time delay between starting successive iterations of the loop. (It is clearer to work with periods than frequencies when combining throughput and latency constraints.) The hardware description may have both a throughput and a delay specification for a loop. A loop delay larger than the repetition period implies pipelining (see Section 9.2 for example).

It is the responsibility of the designer to correct inconsistencies in the timing constraints. For example, a minimum time specification enclosed by a maximum delay of shorter duration is considered to be a definition error and will be treated as such. Similarly, the description is considered to be in error if the loop latency is greater than N times the loop period and an operation in the loop uses a value generated in one of the preceding N iterations of the loop.

4.4 Hardware Interpretations of Ada Constructs

While hardware may be described in Ada, the differences between the target architecture of a software and hardware compiler allows (or requires) a different, although functionally equivalent, interpretation of certain language constructs. Our hardware interpretation of several Ada concepts is discussed in this section.

4.4.1 Program Modularity

Program modularity serves several purposes in software: it clarifies the description, it may support a particular design strategy, and it reduces the size of both the source and object code. At the source level, program modularity holds the same benefits for hardware description. However, the inclusion of timing constraints and the nature of hardware design disallow a direct mapping of all of the description's modularity into hardware.

It is desirable to follow the modularity of the description in the final hardware design as much as possible. This aids in testing and simplifies modifications for second iterations of circuits. However, the size of the final design is a larger consideration in hardware design than in software. For example, if two tasks can be performed by the same circuitry, it is desirable to combine them. Further, the modularity of the description must sometimes be ignored to guarantee that timing specifications are satisfied. For example, the standard software realization for procedures and functions is by microcode branch and returns to a common copy of the code. While the basic branch and return time can easily be accounted for, contention for the procedure code between concurrent tasks cannot. If a task has to wait for access to the code, its execution must be suspended until access is obtained. This delay may cause a timing constraint to be violated. (This contention problem also exists if each procedure is mapped

on to dedicated hardware [Orga84]. This effectively promotes the procedure to a task which is also undesirable.) To prevent contention, a separate (microcode or hardware) implementation of the procedure must be generated for each asynchronous task. Finally, it is sometimes advantageous to expand a procedure inline to extract the maximum low level parallelism of the description. The exact realization of the code is modified to accommodate different procedures which can operate in parallel to different calls on shared hardware.

4.4.2 Dynamic tasking

The term "dynamic tasking" refers to a process of creating and destroying copies of tasks at run-time. In software, dynamic tasking is achieved by temporarily duplicating the task's internal variables (state information). Run-time creation and destruction of memory is not feasible. Thus the only method of realizing dynamic tasking is to create and destroy copies of the variables within a fixed size memory. However, without a dedicated data path for each copy, it is impossible to guarantee that the timing specifications are satisfied. Thus, dynamic tasking is not allowed in our implementation.

4.4.3 The Rendezvous

The general rendezvous can be realized relatively effi-

ciently in hardware using a distributed entry queue. Each task entry has a dedicated counter to store the present length of the queue. Each task has a call address register to store the name of the entry it is queued on and a counter to store its position in the queue. Data is transferred between tasks on an entry bus.

An entry call is realized by the called entry incrementing the associated queue length counter. This value is returned to the caller as its position in the queue. When a task accepts an entry, it raises an accept flag and decrements its queue length counter. Each task queued on that entry decrements its position counter. The task whose counter has reached 0 is "accepted" and that task passes its rendezvous parameters to the accepting task. At the end of the critical region, the calling task lowers the accept flag and the 0-count task reads the "out" parameters off the entry bus.

The realization of the rendezvous can be simplified in several cases. These simplifications are based on assuming that the maximum entry queue length can be determined. This is possible if one assumes that the entire description is compiled at once. Alternatively, the designer could be allowed to specify a queue length via a pragma, but this introduces the possibility of careless errors.

When only one task can call an entry at any given time, the rendezvous can be accomplished by a test-and-set flag. When a task reaches the rendezvous, it checks the status of

the flag to see if the other task has arrived. If the flag is set, the waiting task is reactivated and the rendezvous begins. If the flag is not set, then the task simply sets the flag and waits for the second task to arrive. When at most two tasks can call an entry at once, a second test-and-set flag is used to perform entry queue arbitration. The calling task checks if the flag is set which indicates that the other task is already in the queue. If temporal precedence of calls is not important, the designer can prompt for a more efficient circuit by rewriting the entry as a select statement containing a dedicated entry for each calling task. Finally, simplifications may be achieved by the user embedding (multiple) rendezvous in a procedure which models the desired protocol and designing a hardware cell to directly realize the procedure.

The time two tasks spend waiting for a rendezvous to begin is difficult, sometimes impossible, to determine from the program description. Our current implementation simply ignores these indeterminate delays. All entry calls or accepts are assumed to have minimum delay; i.e., the rendezvous is assumed to begin immediately. Timing constraints which span an entry accept statement account for the execution time of the critical region, but constraints which span an entry call statement do not. It is recommended that the timing constraints are rewritten to separately define intervals for the code before and after the rendezvous if possible.

4.4.4 Shared Variables

Concurrent processes may communicate without explicit synchronization through shared variables. Because of the asynchronous nature of the concurrent processes, there is no method of knowing which process last updated the variable. Thus, shared variables must be realized as fixed memory locations. An arbiter is included in the circuit to allow only one task access to the variable at any time. The order of accesses to a shared variable within a task is maintained as part of the normal hardware compilation process.

4.4.5 Explicit Control Constructs

Ada contains several constructs which alter the program flow (eg. loops, case statements, if-then-else constructs). In our application, we are generating micro-sequenced hardware and thus realize these constructs using microcode branches.

4.4.6 Exceptions

Ada exceptions flag runtime errors and exceptional situations. Predefined exceptions include such errors as out of range array accesses and numeric operation failure (eg. divide by zero). The user may define new exceptions to flag program dependent error conditions such as singular matrices. When an exception is raised, the normal program execution is suspended and the exception handler associated

with the exception is executed. Ada also permits the user to suppress exception detection.

The concept of exceptions transfers directly to hardware description. Cells which realize operations which have associated, predefined exceptions can be designed to include error detection circuitry. For example, an ALU would contain an arithmetic overflow flag which is computed with each arithmetic operation. User defined exceptions can be used to detect data dependent errors. For example, if a framing bit error is detected, corruption of a data channel is flagged.

The handling of exceptions is often very important in hardware as the circuit may be operating in a remote location or in an unfriendly environment. A data repeater located in the middle of a 100 mile data link must have a resynchronization procedure to execute whenever a loss of synchronization is detected. Noncritical circuitry located in an office may simply light an error detected indicator and allow the operator to decide on the appropriate action. In either case, hardware is simply created to realize the specified exception detection and service routines.

4.4.7 Generic Program Units

Ada generic subprogram units provide a template from which executable code can automatically be created. The details required for the expansion are supplied at the point in the code where the generic is instantiated. Since gener-

ics are expanded at compile time, there is no theoretical problem with including them in hardware descriptions. In fact, generics are very convenient for describing parameterized cells (cells whose width is varied to match the data width). Parameterized width cells can easily be realized in hardware by designing a bit-slice library. However, the complexity of compiling other generic program units is not considered in this thesis.

4.5 Functional Simulation in Ada

One of the advantages quoted for using a SPL for hardware description was that the software language could be used for functional simulation. The description (with possibly some additional print statements) is simply compiled as software and run on a general purpose computer. This is ~~sufficient~~ for a single hardware process or descriptions which contain no timing constraints. Timing constraints add implicit synchronization between concurrent tasks. This synchronization must be properly modelled when the description is run as a simulation. This is accomplished by having the package TIMING delay tasks appropriately.

Simulation of timing delays could be accomplished by simply having TIMING call the Ada delay function. However, this requires scaling the time so that the execution time of the code is negligible. A more efficient method is to work

with virtual delays by having TIMING control an events calendar.

Each call by a task to TIMING.REFERENCE causes the current simulation clock time to be stored as the value of the reference label. When a call TIMING.CONSTRAINT is accepted, the statement is scheduled to execute at a time equal to the time value of the reference label plus the time specified in the call. The execution of code statements between the timing reference and the timing constraint can be performed at either end of the interval.

Loops which have throughput delays specified are modelled by setting a start loop event at the throughput period. If the loop has an iteration delay time longer than the loop period, N start loop events will have to be placed on the events calendar on the first iteration where:

$$N = \frac{\text{Iteration delay time}}{\text{Throughput period}}$$

A loop iteration cannot use results calculated in the preceding N iterations as these calculations may still be in progress.

4.6 Example Ada Hardware Descriptions

This section presents two Ada descriptions of a four-function calculator. Example 4.6-1 represents an initial design which describes only the basic function of

the calculator. This level of detail allows the designer to evaluate high level tradeoffs such as which functions can be included in a 4x5 keypad. A hardware circuit could be generated from this level of description, but a more efficient realization can be obtained by refining the description to guide the allocator. For example, the writing of X_REG to a display file implicitly includes decoding X_REG into a decimal representation, then to a 7-segment LED format. Since this is specified as a "write" primitive, a hardware cell would have to be designed for this function. The output operation can be greatly simplified by declaring X_REG as an array of BCD digits.

Example 4.6-2 is a more detailed description of the four-function calculator which would result in a simpler hardware circuit. Example 4.6-2 is a refinement of Example 4.6-1 characteristic of a top-down design step. The first step in the refinement is to convert the task into a package. Two internal tasks (KB_INTERFACE and DISPLAY_DRIVER) model the continuous action of monitoring the keyboard and refreshing the display. A third task, EXECUTION_UNIT, waits for notification of a valid key entry from KB_INTERFACE and then performs the appropriate function. If the execution of a key is not completed before another key is decoded, the protocol of the rendezvous causes KB_INTERFACE to wait until EXECUTION_UNIT is ready to accept another key.

The calculator communicates with the outside world through two interface procedures: INPUT and OUTPUT. Since

these cells will be manually designed, the Ada description does not need to accurately model their behavior. Thus, the procedures have been extended to translate the Boolean format of the calculator into ASCII code for the designer to read.

As mentioned earlier, the cost of the final design is influenced by the Ada description. This is because the allocator cannot interpret what the designer means, only what he says. If restrictions are not explicitly specified, the allocator must assume the most general interpretation of the Ada code. If Example 4.6-1 was automatically translated, X_REG, Y_REG, and M_REG could be realized as 32-bit binary registers and all functions might be realized as 32-bit, floating point operators. In Example 4.6-2, the designer has prompted the allocator to use 4-bit arithmetic operators by specifying the number storage as an array of BCD digits and by explicitly decomposing the operations into BCD addition and subtraction. (This decomposition could also have been done via nonterminal productions.)

Another advantage of refining the description is the inclusion of error routines. For example, an arithmetic overflow now causes an error symbol to be displayed; the entry of a ninth digit is ignored. Example 4.6-2 also includes timing constraints which specify the keyboard scan rate, the display refresh rate, the maximum time for execution of a key, and the power-up initialization times.

EXAMPLE 4.6-1

```
task CALCULATOR is
  -- four function calculator with memory register
  -- simulation reads from file KEYBOARD, one key per line.
  -- simulation writes to file DISPLAY, each line represents
  -- display.
end CALCULATOR;
```

```
task body CALCULATOR is
```

```
  type REGISTER is digits 8;
```

```
  ADD      : constant := 10;
  SUB      : constant := 11;
  MUL      : constant := 12;
  DIV      : constant := 13;
  EQUALS   : constant := 14;
  RECALL   : constant := 15;
  STORE    : constant := 16;
  CLEAR    : constant := 17;
  CHS      : constant := 18;
  DEC_PT   : constant := 19;
```

```
  X_REG, Y_REG, M_REG: REGISTER := 0.0;
```

```
  FRACTION: REGISTER := 1.0;
```

```
  KEY      : INTEGER;
```

```
  OP       : INTEGER range ADD..DIV;
```

```
  LAST_KEY_DIGIT, DECIMAL : BOOLEAN := FALSE;
```

```
  procedure COMPUTE
```

```
    -- performs calculation of entered operation on X_REG
    -- and Y_REG. Result is placed in X_REG, Y_REG
    -- retains initial value.
```

```
  begin
```

```
    case OP is
```

```
      when ADD => X_REG := Y_REG + X_REG;
```

```
      when SUB => X_REG := Y_REG - X_REG;
```

```
      when MUL => X_REG := Y_REG * X_REG;
```

```
      when DIV => X_REG := Y_REG / X_REG;
```

```
    end case;
```

```
  end COMPUTE;
```

```

begin
  loop
    READ ( KEYBOARD, KEY )
    case KEY is -- decode key
      when 0..9 => if LAST_KEY_DIGIT then-- update X_REG
                     if DECIMAL then -- add digit as fract.
                       FRACTION := FRACTION / 10.;
                       X_REG := X_REG + KEY * FRACTION;
                     else -- add digit to integer part
                       X_REG := X_REG * 10. + KEY;
                     end if;
                   else -- reset flags and X_REG
                     LAST_KEY_DIGIT := TRUE;
                     DECIMAL := FALSE;
                     FRACTION := 1.0;
                     X_REG := key;
                   end if;
      when ADD|SUB|MUL|DIV => OP := KEY; -- record operator
                           Y_REG := X_REG;
                           LAST_KEY_DIGIT := FALSE;
      when EQUALS => COMPUTE; -- perform calculation
                   LAST_KEY_DIGIT := FALSE;
      when CHS => X_REG := - X_REG;
                LAST_KEY_DIGIT := FALSE;
      when STORE => M_REG := X_REG; -- store X_REG
                  LAST_KEY_DIGIT := FALSE;
      when RECALL => X_REG := M_REG; -- recall M_REG
                   LAST_KEY_DIGIT := FALSE;
      when CLEAR => X_REG := 0.0; -- clear X_REG
                   if not LAST_KEY_DIGIT then
                     Y_REG := 0.0; -- clear Y_REG
                     OP := 0; --clear operator
                     LAST_KEY_DIGIT:= FALSE;
                   end if;
      when DEC_PT => DECIMAL := TRUE;
      when others => null; -- ignore invalid keys
    end case;
    WRITE ( DISPLAY, X_REG); -- write present value of display
  end loop;
end CALCULATOR;

```


EXAMPLE 4.6-2

```
package CALCULATOR is
  --four function calculator
  --reads data from file KEYBOARD
  --writes data to file DISPLAY
end CALCULATOR;
```

```
package body CALCULATOR is
```

```
type NINE_BITS is array 0..8 of BOOLEAN;
type BYTE      is array 0..7 of BOOLEAN;
type BCD       is INTEGER range 0..9;
```

```
-- declare registers and flags
X_REG,Y_REG,M_REG: array 0..7 of BCD;
X_DECIMAL,Y_DECIMAL,M_DECIMAL: INTEGER range 0..7;.
-- location of decimal point in each register
X_SIGN,Y_SIGN,M_SIGN: BOOLEAN;
-- sign of each register,.TRUE => negative
OVERFLOW: BOOLEAN := FALSE;
-- arithmetic overflow flag
```

```
task KB_INTERFACE is
  --keyboard interface for calculator
  --debounces keyboard via hardware interface
  --procedure and passes valid keyboard
  --depressions to execution unit.
end KB_INTERFACE;
```

```
task DISPLAY_DRIVER is
  --encodes X_REG into format for 7-segment display.
  --8 digits and sign are pulsed for 1 msec ever 10 msec.
  --hardware cell for output; code for OUTPUT translates
  --7-segment format back to decimal display for easy
  --interpretation. Writes to DISPLAY file every 10 msec.
end DISPLAY_DRIVER;
```

```
task EXECUTION_UNIT is
  --accepts valid keys from KB_INTERFACE
  --decodes key and performs appropriate function

  entry KEY_DEPRESSION ( LINE_CODE : in NINE_BITS );
end EXECUTION_UNIT;
```

task body KB_INTERFACE is

```
NO_KEYS_DOWN: constant NINE_BITS := (others=>FALSE);
NEW_LINES, OLD_LINES: NINE_BITS := NO_KEYS_DOWN;
KEY_DEPRESSED: BOOLEAN := FALSE;
```

procedure INPUT (LINES : out NINE_BITS) is

```
-- Models comparators which monitor input lines
-- to calculator from keyboard matrix.
-- Direct hardware realization to be provided.
-- For easy description in file, procedure
-- reads ASCII file and translates integer code
-- to bit vector.
-- note: keyboard file is read at 10 msec intervals
--       two successive entries which match signal
--       stable key depression.
```

pragma HARDWARE_CELL;

KEY: INTEGER;

-- codes of operators in file KEYBOARD

```
ADD      : constant := 10;
SUB      : constant := 11;
MUL      : constant := 12;
DIV      : constant := 13;
EQUALS   : constant := 14;
RECALL   : constant := 15;
STORE    : constant := 16;
CLEAR    : constant := 17;
CHS      : constant := 18;
DEC_PT   : constant := 19;
```

begin

loop

GET (KEYBOARD, KEY);

-- encode input into keyboard row-column format

LINES(0) := KEY=7 or KEY=8 or KEY=9 or
KEY=ADD or KEY=CLEAR;

LINES(1) := KEY=4 or KEY=5 or KEY=6 or
KEY=SUB or KEY=STORE;

LINES(2) := KEY=1 or KEY=2 or KEY=3 or
KEY=MUL or KEY=RECALL;

LINES(3) := KEY=3 or KEY=DEC_PT or KEY=CHS or
KEY=DIV or KEY=EQUALS;

LINES(4) := KEY=0 or KEY=1 or KEY=4 or KEY=7;

LINES(5) := KEY=DEC_PT or KEY=2 or KEY=5 or KEY=8;

LINES(6) := KEY=CHS or KEY=3 or KEY=6 or KEY=9;

LINES(7) := KEY=DIV or KEY=MUL or KEY=SUB or KEY=ADD;

LINES(8) := KEY=EQUALS or KEY=RECALL or KEY=STORE or KEY=CLEAR;

-- out of range key encoded as no keys pressed

TIMING.CONSTRAINT (, , 1E-6);

--delay is just time required to latch LINES

```

    end loop;
end INPUT;

-- main body of task KB_INTERFACE

begin
    TIMING.CONSTRAINT( , , 0.2); -- power on set up time
    loop
        TIMING.REFERENCE ("KB_SAMPLE");
        INPUT ( NEW_LINES );
        if OLD_LINES /= NEW_LINES then -- keyboard bounce
            OLD_LINES := NEW_LINES;
        elseif OLD_LINES = NO_KEYS_DOWN then -- keyboard idle
            KEY_DEPRESSED := FALSE;
        elseif not KEY_DEPRESSED then
            -- a key has just been validated
            KEY_DEPRESSED := TRUE;
            EXECUTION_UNIT.KEY_DEPRESSION ( NEW_LINES )
            --initiate action of key
        end if; -- ignores key being held and key roll-over
        TIMING.CONSTRAINT ("KB_SAMPLE", 9E-3, 11E-3);
    end loop;
end KB_INTERFACE;

```

task body DISPLAY_DRIVER is

```

    type SEVEN_BITS is array 0..6 of BOOLEAN;

    CODE_7S : constant array 0..9 of SEVEN_BITS :=
    -- look-up table for 7-segment codes.
    ((7 =>FALSE, others => TRUE), -- "0"
    (2|3 =>TRUE, others => FALSE), -- "1"
    (3|6 =>FALSE, others => TRUE), -- "2"
    (5|6 =>FALSE, others => TRUE), -- "3"
    (0|3|4=>FALSE, others => TRUE), -- "4"
    (1|4 =>FALSE, others => TRUE), -- "5"
    (1 =>FALSE, others => TRUE), -- "6"
    (0|1|2=>TRUE, others => FALSE), -- "7"
    ( others => TRUE), -- "8"
    (4 =>FALSE, others => TRUE)); -- "9"

```

```

DIGIT_CODE: array 0..8 of BOOLEAN;
-- bit = TRUE corresponds to digit being refreshed
SEGMENTS: BYTE;
-- bit = TRUE corresponds to LED segment lit

```

```

procedure OUTPUT (.SEGMENT : in WORD;
                  DIGIT : in NINE_BITS) is
    -- decodes display format to decimal file format
    -- hardware cell to be provided

```

```

pragma HARDWARE_CELL;

```

```

begin
  if DIGIT(8) then
    -- start new line and display sign
    NEW_LINE;
    if SEGMENT(6) then
      PUT ( DISPLAY, "-");
    else
      PUT ( DISPLAY, " ");
    end if;
  else
    -- display digit
    INDEX := 0;
    while SEGMENT(0..6) /= CODE_7S(INDEX) and
      INDEX < 10 loop
      INDEX := INDEX + 1;
    end loop;
    if INDEX < 10 then
      PUT ( DISPLAY, INDEX );
    else --display error message
      PUT ( DISPLAY, "ENCODING ERROR" );
    end if;
    if SEGMENT(7) then
      PUT (DISPLAY, ".");
    end if;
  end if;
end OUTPUT;

-- main body of task DISPLAY_DRIVER

begin
  TIMING.CONSTRAINT ( , ,0.2); -- power on init
  loop
    TIMING.REFERENCE ("ENCODE_SIGN")
    -- display sign of number
    OUTPUT ( SEGMENTS , (others=>FALSE));
    -- blank display during update
    if X_SIGN then
      SEGMENTS := (7=>TRUE, others=>FALSE);
    else
      SEGMENTS := (others=>FALSE);
    end if;
    DIGIT_CODE := (8=>TRUE, others => FALSE);
    -- refresh sign LED
    OUTPUT ( SEGMENTS, DIGIT_CODE );
    TIMING.CONSTRAINT ("ENCODE_SIGN",0.0,1E-5,"DISPLAY_SIGN");
    -- hold display for 1 msec
    TIMING.CONSTRAINT ("DISPLAY_SIGN",9E-4,11E-4);
    for INDEX in 0..7 loop
      --display digits of number
      OUTPUT ( SEGMENTS, (others=>FALSE));
      --blank display during update
      DIGIT_CODE := ( INDEX =>TRUE, others =>FALSE);
      if OVERFLOW then
        --display top half of "8" and dec.pt. in each digit
        SEGMENTS := (2|3|4=>FALSE, others=>TRUE);
      end if;
    end loop;
  end loop;
end;

```

```

else
--encode X_REG(INDEX)
SEGMENTS(0..6) := CODE_7S(X_REG(INDEX));
if INDEX = X_DECIMAL then
--display decimal point after this digit
SEGMENTS (7) := TRUE;
else
SEGMENTS (7) := FALSE;
end if;
end if;
OUTPUT ( SEGMENTS, DIGIT_CODE );
TIMING.CONSTRAINT ("ENCODE_DIGIT",0.0,1E-5,"DISPLAY_DIGIT");
TIMING.CONSTRAINT ("DISPLAY_DIGIT",9E-4,11E-4);
end loop;
end loop;
end DISPLAY_DRIVER;

```

task body EXECUTIONS_UNIT is

-- task which decodes and executes key

```

ADD : constant := 0
SUB : constant := 1
MUL : constant := 2
DIV : constant := 3

```

```

X_INDEX : INTEGER range 0..8;
-- address of the next digit to be entered
LAST_KEY_DIGIT : BOOLEAN := FALSE;
-- flag which indicates nature of previous key
FRACTION : BOOLEAN := FALSE;
-- flag which indicates if digit is in fractional part
OP : INTEGER range 0..3 := 0;
-- operator to be performed upn "EQUALS"

```

procedure ADD_X_Y is

```

-- adds data in Y_REG TO DATA IN X_REG
-- data stored in X_REG.

```

```

TEMP: INTEGER range -10..19;
BORROW,CARRY: INTEGER range 0..1 := 0;

```

begin

-- equate the decimal location of the registers

if X_DECIMAL > Y_DECIMAL then

while X_DECIMAL > Y_DECIMAL loop

Y_REG(1..7) := Y_REG(0..6);

Y_REG(0) := 0;

Y_DECIMAL := Y_DECIMAL + 1;

end loop;

elseif Y_DECIMAL > X_DECIMAL then

while Y_DECIMAL > X_DECIMAL loop

X_REG(1..7) := X_REG(0..6);

```

    X_REG(0) := 0;
    X_DECIMAL := X_DECIMAL + 1;
end loop;
end if;
if not(X_SIGN xor Y_SIGN) then
-- both numbers have same sign, add magnitudes
  for INDEX in reverse 0..7 loop
    TEMP := X_REG(INDEX) + Y_REG(INDEX) + CARRY;
    if TEMP >= 10 then
      --generate carry
      CARRY := 1;
      X_REG(INDEX) := TEMP - 10;
    else
      CARRY := 0;
      X_REG(INDEX) := TEMP;
    end if;
  end loop;
  if CARRY = 1 then
    if X_DECIMAL = 7 then
      --arithmetic overflow
      OVERFLOW := TRUE;
    else -- shift display and set X_REG(0) to 1
      X_REG(1..7) := X_REG(0..6);
      X_REG(0) := 1;
      X_DECIMAL := X_DECIMAL + 1;
    end if;
  end if;
else
-- sign are different, subtract magnitudes
  for INDEX in reverse 0..7 loop
    TEMP := X_REG(INDEX) - Y_REG(INDEX);
    if TEMP < 0 then
      X_REG(INDEX) := TEMP + 10;
      BORROW := 1;
    else
      X_REG(INDEX) := TEMP;
      BORROW := 0;
    end if;
  end loop;
  if BORROW = 1 then
    -- result is negative, complement X_REG and change sign
    for INDEX in reverse 0..7 loop
      TEMP := 9 - X_REG(INDEX) + BORROW;
      if TEMP = 10 then
        X_REG(INDEX) := 0;
        BORROW := 1;
      else
        X_REG(INDEX) := TEMP;
        BORROW := 0;
      end if;
    end loop;
    X_SIGN := not X_SIGN;
    -- normalize result
    while X_REG(0) = 0 and X_DECIMAL > 0 loop

```

```

        X_REG(0..6) := X_REG(1..7);
        X_REG(7) := 0;
        X_DECIMAL := X_DECIMAL - 1;
    end loop;
end if;
end if;
end ADD_X_Y;

```

procedure SUBTRACT is

```

-- subtracts X_REG from Y_REG
-- data stored in X_REG

```

```

begin
    X_SIGN := not X_SIGN;
    ADD_X_Y;
end SUBTRACT;

```

procedure MULTIPLY is

```

-- multiplies X_REG by Y_REG
-- result stored in X_REG

```

```

TI      : INTEGER range 0..15;
-- index for P_REG access in loops
P_REG : array 0..15 of INTEGER range 0..31 := 0;
-- register for temporary product
P_DECIMAL : INTEGER range 0..16 := 0;
-- decimal location in P_REG
SHIFT, CARRY : INTEGER range 0..9;
TEMP : INTEGER range 0..127;
-- temporary storage

```

```

begin
    X_SIGN := X_SIGN xor Y_SIGN;
    P_DECIMAL := X_DECIMAL + Y_DECIMAL;
    -- multiply digit by digit
    for XI in reverse 0..7 loop
        for YI in reverse 0..7 loop
            TI := XI + YI;
            TEMP := X_REG(XI) * Y_REG(YI) + P_REG(TI+1);
            -- generate carry
            P_REG(TI) := P_REG(TI) + TEMP / 10;
            -- store remainder
            P_REG(TI+1) := TEMP mod 10;
        end loop;
    end loop;
    if P_REG(0) > 0 then
        -- shift result right 1 digit to normalize
        P_DECIMAL := P_DECIMAL + 1;
        SHIFT := 0;
    else
        SHIFT := 1;
        while P_REG(SHIFT) = 0 and P_DECIMAL > 0 loop

```

```

    SHIFT := SHIFT + 1;
    P_DECIMAL := P_DECIMAL - 1;
end loop;
if P_DECIMAL > 8 then
    --arithmetic overflow
    OVERFLOW := TRUE;
else
    --store result
    X_REG := P_REG(SHIFT..SHIFT + 7);
    X_DECIMAL := P_DECIMAL;
end if;
end MULTIPLY;

procedure DIVIDE is
begin
    -- appropriate BCD division algorithm
end DIVIDE;

procedure DIGIT_ENTERED ( DIGIT : in INTEGER range 0..9) is
    -- updates X_REG when a digit is entered
begin
    if not LAST_KEY_DIGIT then
        -- set X_REG := DIGIT and reset flags
        X_INDEX := 1;
        X_REG := (0 => DIGIT, other => 0);
        X_SIGN := FALSE;
        FRACTION := FALSE;
        LAST_KEY_DIGIT := TRUE;
    else
        -- update X_REG if not full
        if X_INDEX < 8 then
            X_REG(X_INDEX) := DIGIT;
            if not FRACTION then
                X_DECIMAL := X_INDEX;
            end if;
            X_INDEX := X_INDEX + 1;
        end if;
    end if;
end DIGIT_ENTERED;

procedure OPERATOR_ENTERED ( OPERATOR : in INTEGER range 0..3) is
    -- updates status when an operator is entered
    -- shifts X_REG to Y_REG when appropriate
begin
    -- store operator
    OP := OPERATOR;
    if LAST_KEY_DIGIT then

```



```

-- transfer X_REG to Y_REG
Y_REG := X_REG;
Y_SIGN := X_SIGN;
Y_DECIMAL := X_DECIMAL;
LAST_KEY_DIGIT := FALSE;
end if;
end OPERATOR_ENTERED;

procedure COMPUTE is
-- initiates computation of appropriate
-- arithmetic operation
begin
case OP is
when ADD => ADD_X_Y;
when SUB => SUBTRACT;
when MUL => MULTIPLY;
when DIV => DIVIDE;
end case;
end COMPUTE;

-- main body of task EXECUTION_UNIT
begin
TIMING.CONSTRAINT( , 0.2, 0.3, "EX_INIT");
--reset registers after KB_INTERFACE is stable
X_REG := (others => 0);
Y_REG := X_REG;
M_REG := X_REG;
X_SIGN := FALSE;
Y_SIGN := X_SIGN;
M_SIGN := X_SIGN;
X_DECIMAL := FALSE;
Y_DECIMAL := X_DECIMAL;
M_DECIMAL := X_DECIMAL;
TIMING.CONSTRAINT("EX_INIT", , 0.1);

loop -- forever
TIMING.REFERENCE("CALCULATION");
accept KEY_DEPRESSION( LINE_CODE : in WORD ) do
KEY_LINES := LINE_CODE;
end accept;
-- blank display during calculation
BLANK := TRUE;
-- if more than two lines are set, this
-- indicates a multiple key depression
ONES := 0;
for INDEX in 0..8 loop
if KEY_LINES(INDEX) then
ONES := ONES + 1;
end if;
end loop;
if ONES = 2 then

```

```

-- possibly a valid key depression
if OVERFLOW then
-- treat any key depression as a "CLEAR"
  LAST_KEY_DIGIT := FALSE;
  KEY_LINES := ( 0|8 => TRUE, others => FALSE);
  OVERFLOW := FALSE;
end if;
-- decode key
if KEY_LINES(0) then
  if KEY_LINES(4) then DIGIT_ENTERED(7);
  elseif KEY_LINES(5) then DIGIT_ENTERED(8);
  elseif KEY_LINES(6) then DIGIT_ENTERED(9);
  elseif KEY_LINES(7) then OPERATOR_ENTERED(ADD);
  elseif KEY_LINES(8) then -- "CLEAR" entered
    X_REG := (others => 0);
    X_SIGN := FALSE;
    X_DECIMAL := 0;
    if not LAST_KEY_DIGIT then
      -- also clear Y_REG and OP
      Y_REG := X_REG;
      Y_SIGN := X_SIGN;
      Y_DECIMAL := X_DECIMAL;
      OP := ADD; -- identity if Y_REG := 0
      LAST_KEY_DIGIT := FALSE;
    end if;
  end if;
elseif KEY_LINES(1) then
  if KEY_LINES(4) then DIGIT_ENTERED(4);
  elseif KEY_LINES(5) then DIGIT_ENTERED(5);
  elseif KEY_LINES(6) then DIGIT_ENTERED(6);
  elseif KEY_LINES(7) then OPERATOR_ENTERED(SUB);
  elseif KEY_LINES(8) then -- "STORE" entered
    M_REG := X_REG;
    M_SIGN := X_SIGN;
    M_DECIMAL := X_DECIMAL;
    LAST_KEY_DIGIT := FALSE;
  end if;
elseif KEY_LINES(2) then
  if KEY_LINES(4) then DIGIT_ENTERED(1);
  elseif KEY_LINES(5) then DIGIT_ENTERED(2);
  elseif KEY_LINES(6) then DIGIT_ENTERED(3);
  elseif KEY_LINES(7) then OPERATOR_ENTERED(MUL);
  elseif KEY_LINES(8) then -- "RECALL" entered
    X_REG := M_REG;
    X_SIGN := M_SIGN;
    X_DECIMAL := M_DECIMAL;
    LAST_KEY_DIGIT := FALSE;
  end if;
elseif KEY_LINES(3) then
  if KEY_LINES(4) then DIGIT_ENTERED(0);
  elseif KEY_LINES(5) then -- "DEC_PT" entered
    FRACTION := TRUE;
    if not LAST_KEY_DIGIT THEN
      -- number to be entered < 1

```

```

      X_REG := (others => 0);
      X_SIGN := FALSE;
      X_DECIMAL := 0;
      X_INDEX := 0;
      LAST_KEY_DIGIT := TRUE;
    end if;
    elseif KEY_LINES(6) then -- "CHS" entered
      X_SIGN := not X_SIGN;
    elseif KEY_LINES(7) then OPERATOR_ENTERED(DIV);
    elseif KEY_LINES(8) then -- "EQUALS" entered
      COMPUTE;
      LAST_KEY_DIGIT := FALSE;
    end if;
  end if;
end if;
-- display X_REG when calculation over
BLANK := FALSE;
-- set large time limit to minimize area
TIMING.CONSTRAINT("CALCULATION", ,0.3);
end loop;
end EXECUTION_UNIT;
end CALCULATOR;

```

5.0 FLOW GRAPHS AND GRAPH GRAMMARS

5.1 Flow Graphs as an Intermediate Level Format

Automatic generation of hardware directly from an Ada description is a difficult process because of the richness of the Ada language. Further, the in-line code syntax within an Ada task introduces unnecessary serialism which may result in slower execution and increased hardware size. Translation of the Ada description onto a flow graph intermediate form simplifies the translation process and extracts the maximum parallelism of the algorithm.

The use of an intermediate form simplifies the translation process by separating the syntax of Ada from the hardware allocation routines. All syntax checking, type checking, etc. is performed while translating the Ada code onto the intermediate form. The allocator can then assume that the intermediate form contains no errors. The allocator is also simplified by converting all Ada data types (eg. INTEGER.range -128..127) into types recognized by the allocator (eg. INTEGER:8 bits, two's complement).

Using a flow graph as an intermediate form removes any unnecessary serialism introduced by the Ada syntax. For example, the code segment:

```
A := B + C;
```

```
D := B * 5;
```

is equivalent to:

```

D := B * 5;
A := B + C;

```

or to executing both statements in parallel. A flow graph represents the above code as two parallel data operations. The order in which the operations are executed is determined by the available hardware during the allocation process.

The flow graph also separates each assignment to a variable. This allows different (available) memory locations to be used to store a variable at different times. This may reduce the size of the memory required. Further, the number of memory accesses may be reduced if an internal register may store the value for the required duration.

The allocator presented in this thesis uses a control/data-flow graph (CDFG) as an intermediate form. The CDFG is basically a data-flow graph with explicit control constructs. The nodes in the graph represent data operations (eg. addition) or control operations (eg. loop). The edges of the graph represent the results of operations. They connect the output of an operation generating a value to the input of an operation using that result. Thus, the edges serve to define the order in which the operations must be performed. Precedence edges are added to specify an order between two operations whose order is not defined by a value edge or to specify a start time for operations with no value inputs.

Translation of the Ada code onto a CDFG for hardware compilation is identical to compiling an Ada program onto a

graph for optimization. [NagM81]. The process includes syntax checking of the Ada description, type checking of variables, and determination of variable representations (i.e. bit requirements). All routines which use the CDFG as an input description may then assume that the description contains no syntactic errors. The CDFG is also a convenient representation for performing technology independent optimization of the circuit description. This is achieved by applying transformations on the CDFG which reduce the number of operations, or reduce the frequency with which operations are executed. These transformations are analogous to those performed by an optimizing software compiler [NagM81] or those performed on the value trace [SnoE78].

The general rules for translating an Ada description onto a CDFG are:

- a) All operations become nodes of the graph. Control constructs (branches, loops, etc) are represented by corresponding control nodes. Two node types, "RVCALL" and "ACCEPT" are added to the CDFG which directly correspond to rendezvous calls and accepts. (An accept also implies synchronization at each end of the critical region.)
- b) The edges of the graph represent the value of a variable. An edge is created from the output of the node which generates the result to the input of all nodes which use the value. Precedence edges are used to define an order of execution for operations which do

not have an order implied by value edges. For example, precedence arcs are required to ensure that the order of assignments and accesses of a shared variable within each calling task does not change. Nodes which have no data inputs are given an incident precedence edge from a preceding CDFG node to indicate when execution may begin. A precedence constraint is also used in the rendezvous to ensure that there is a task in an entry queue before the entry is accepted.

c) A subgraph is generated for each function or procedure. The subprogram call is translated as a node of the graph termed a "CALL" (a VT-body in the value trace). The inputs of the CALL are the input (and inout) parameters to the subprogram, plus any global variables accessed within the subprogram. The outputs of the CALL correspond to all output (and inout) parameters plus any global variables accessed by the subprogram.

d) Timing specifications are represented in the CDFG as nodes which simply pass data. The outputs of a timing reference are connected to the nodes within the constrained segment which access variables generated above the reference point. The inputs of a timing constraint are the outputs of nodes within the constrained segments which generate values accessed below the timing constraint.

e) Packages are used to obtain modular designs. They limit the scope of the optimization and allocation routines. Package boundaries are identified by subdividing the CDFG into subgraphs.

5.2 Graph Grammar Formulation of Hardware Design

A principal goal set for the Elf system was that all technology dependent design information was to be represented by rules contained in a library. This would allow new cells and new technologies to be introduced without having to change the allocator code. This goal could have been achieved by writing rules for an expert system tool as done in the CMU-DAA. This approach was not followed for two reasons. First, expert systems allow rules concerning the global nature of the algorithm and the circuit to be defined. Without a special data structure to contain this information, it is difficult to determine when such rules are applicable. Second, the rules searching algorithm of an expert system writing tool is not specialized for circuit design. This may lead to large execution times. For example, the CMU-DAA expert system [ThoD83, KowT84] required 47 cpu-hours on a VAX11/750 to generate a data path for an IBM 370. To maintain efficiency, it is necessary to have a well defined process for determining applicable rules and methods for sorting the rule base to limit the number of rules tested in each instance, and the number of times each rule is

tested. Formulating the rules as graph grammar productions helps achieve these goals (see Section 5.3).

The representation of hardware design information as graph grammar productions was complemented by choosing graphical input and output formats for the allocator. The CDFG was selected as the intermediate form as discussed in the previous section. Thus, the input to the allocator is a graph. The allocator was to generate a standard cell.net-list and a mapping to the algorithm onto this net-list. Representing the mapping by an extended CDFG made both outputs of the allocator graphs. The process of hardware design was then treated as deriving a word in a graph language.

Graph grammars are a two-dimensional extension of string grammars [Clav79]. A graph grammar is a tuple:

$$GG = (Vg, Tg, Pg, Sg)$$

where $Vg = (Ve, Vn)$ is a pair of sets of variable edge and node labels, $Tg = (Te, Tn)$ is a pair of sets of terminal edge and node labels, Pg is a finite set of graph productions, and Sg is the start graph.

In a hardware ~~con~~text, the CDFG representation of the algorithm corresponds to the start graph. The variable node and edge labels represent functional operations in the algorithm. The node labels represent operations to be performed (eg. add, loop). Edge variables are used to define the

signal type of a data edge (eg. INTEGER: 8 bits). Terminal node labels are used to indicate which cell is used to realize an operation. Terminal edge labels specify which register is used to store the value. The nonterminal productions in the grammar indicate functional equivalences (see Section 5.3). Terminal productions define which cells can be used to realize an operation.

As with string grammars, each graph grammar has an associated graph language. A graph language $L(GG)$ is defined to be the set of all terminal graphs derivable from the start graph:

$$L(GG) = \{H \mid H \in \text{GRAPHS} \text{ and } Sg \xRightarrow{*} H\}$$

A word in the language is generated by starting from Sg and applying productions in Pg to replace variable nodes, edges, or subgraphs until all nodes and edges have terminal labels. In hardware, the words in the language are the mappings of the algorithm onto the circuit for all timeslots. The language corresponding to the grammar is then the set of all mappings onto all circuits which can realize the algorithm. (The circuit itself cannot be a word in the language as the cells in the circuit are used to realize different operations in different timeslots.) These mappings can be represented as an extended CDFG. Each node (edge) is labelled with the cell (register) used to realize it. A particular mapping circuit and the corresponding mapping are

generated by assigning costs to the productions of the grammar and minimizing the sum of the cost of the productions applied.

5.3 Representation of Design Information as Productions

The representation of information in any rule-based system is based on the antecedent-consequent structure of the conditional in logic. As in logic, the system checks the antecedent. If it is true, then the consequent may be performed. However, the truth of the antecedent implies the ~~feasibility~~, rather than the absolute truth, of the consequent. Because of the possibility of choosing to perform one of several consequent actions, rule-based systems more closely resemble grammars than deductive logic. In a grammar context, the rules correspond to the productions of the grammar.

There are two types of rules used to generate the data path CG from the CDFG: data flow productions and control productions. Data flow productions are used to select realizations for operations in the CDFG. They are represented by "true" graph grammar productions. Control productions specify the effect of explicit control constructs on the structure of the data path. Control productions are usually realized by corresponding control procedures.

5.3.1 Data Flow Productions

A true graph grammar production consists of two sides, each of which is a single node, edge or subgraph. The left hand side (LHS) of the production corresponds to the antecedent of the conditional while the right hand side (RHS) corresponds to the consequent. A production is "applicable" if its LHS matches a portion of the CDFG (i.e. is true). The production is applied by replacing the portion of the CDFG matched to the LHS with the RHS. The RHS is connected into the CDFG by connecting each input (output) of the RHS to the node output (input) previously connected to the corresponding input of the subgraph matched to the LHS. When several productions are applicable at a given point in the CDFG, the decision of which production to apply is made by assigning each production a cost and attempting to minimize the sum of the costs of all productions applied. This selection process is discussed in Chapter 7.

Data flow productions are classified according to the characteristics of the LHS and RHS (see Fig. 3.2-2). There are two basic classes: terminal and nonterminal. Terminal productions have only terminal labels on their RHS. To simplify the implementation, terminal productions are further restricted to the form:

single nonterminal --> single terminal.

If the nonterminal is a node, then the terminal represents a cell capable of realizing the corresponding operation. If the nonterminal is an edge, then the terminal represents a storage cell type capable of storing the value. (Storage elements are assumed to be bit slice and are expanded to the required width automatically.) Sample terminal productions are realizing an addition on an ALU of appropriate width or storing an INTEGER value in a D-type register of appropriate width.

Nonterminal productions contain at least one nonterminal on the RHS. In Elf, the grammar is restricted by requiring that nonterminal productions contain only nonterminal labels on both the RHS and LHS. The LHS and RHS may be single nodes or subgraphs, as shown in Fig. 3.2-2. Thus, the productions may be context free or context sensitive.

Nonterminal productions specify functionally equivalent (sets of) operations. Their primary use is to describe arithmetic equivalences which may lead to more efficient realizations. For example, a multiplication by 4 is equivalent to two one-bit arithmetic shifts left (see Fig. 3.2-2). The ability to specify a subgraph as the RHS allows the allocator to decompose complex operations into a group of simpler operations. For example, multiplication can be decomposed into a sequence of shifts and adds. Realizing these equivalent forms, rather than realizing the LHS directly, may lead to a smaller circuit. A smaller circuit may also be realized by using a less efficient realization

for an operation if the realization is already in the circuit. If a circuit contains an adder, but no cells which realize shifts, a shift left can be realized by adding the number to itself.

Using nonterminal productions serves to reduce the size of the cell library. By decomposing operations into simpler operations using a nonterminal production, including a macro cell in the library to realize the complex operation is not necessary. Similarly, operations on large word widths can be decomposed into a sequence of operations on limited bit ranges allowing the library to contain cells of standard word widths or bit slice cells. Nonterminal productions can also be used to realize a limited form of functional optimization. For example, an operation which is duplicated can be replaced by a single operator. This is denoted by a production whose LHS is a subgraph containing two identical nodes with the same inputs. The RHS of the production is a subgraph consisting of one node generating both body outputs. However, this method of functional optimization is not necessarily efficient.

Finally, nonterminal productions can be used to group operations into an intermediate node. This allows one to bypass the restriction that the LHS of a terminal production must be a single node. To use a cell which multiplies two values and adds the result to the third input, a new operation "MADD" is created. A nonterminal production specifies that a multiply-add combination is equivalent to a MADD. A

terminal production then specifies that a MADD is realized by the given cell.

5.3.2 Control Productions

The algorithm described in this thesis designs only the data path of the circuit. The necessary control circuitry would be designed by a second pass, as in the CMU-DA system [NagA82]. However, explicit control flow operators in the algorithm such as loops and conditional branches (eg. if-then-else constructs and case statements) affect the allocation process and must be considered during the design of the data path. The structure of graph grammar productions restricts the types of rules that can be specified to substitutions. This limitation is caused by the way productions are applied; the LHS is simply replaced by the RHS. This forces the RHS to be equivalent to the LHS and disallows action rules of the form "if antecedent then perform consequent action". This type of rule is required to constrain the data path when an explicit control node is encountered. This format for control operators is desirable as it allows new control constructs to be easily added to the allocator.

Action rules may be represented by a modified production. The LHS is still a nonterminal node or subgraph. However, the RHS is a procedure which performs the necessary action to realize the node. When the LHS is

matched to a node in the CDFG, the production is realized by simply calling the RHS procedure.

As an example, consider the loop construct. This may be realized directly as a loop in the controller's micro code. In this case, the loop procedure simply constrains the storage of values such that values generated at the end of each loop iteration are stored in the same register as the corresponding value is stored in when the loop was entered. Alternately, if the number of loop iterations is fixed, the loop may be "unwound" by creating copies of the loop in sequence. This allows operations outside the loop to be performed in parallel with the loop to meet stringent timing constraints.

Control productions may also be nonterminal. In this case, a control node is replaced by alternate realizations of the control flow. For example, a case statement can be decomposed into an if-then-elseif-...-endif construct. This form of control productions resembles a nonterminal data flow production.

5.3.3 Production Identification and Application

As mentioned previously, a production is applicable if its LHS matches a portion of the CDFG. The production is applied by substituting the RHS for the portion of the CDFG matched to the LHS. This section describes the procedure used to identify if a production is applicable and the method for replacing the LHS with the RHS.

A production whose LHS is a single node is termed context-free (CF) as the decision of whether or not the production is applicable depends only on the characteristics of the node itself, not on its context. To determine if a CF production is applicable consists merely of checking the node type (i.e. operation realized) and the input/output data types. Two operators are matched if they have the same name. Data types are said to match if the CDFG node input/output type is a subset of the corresponding data type in the LHS of the production. Whether or not one data type is a subset of another is determined by calling a user definable function: SUBSET_DATA_TYPE. In the prototype, data types are represented by a type-name, bit-width pair (see Appendix 1). One type is a subset of another if it has the same type name and a smaller bit width. Productions may contain variables as bit widths to imply that they are applicable independent of word width (see Fig 3.2-2).

Context sensitive (CS) productions have a LHS which is a subgraph containing two or more nodes. A CS production is applicable if each node in the LHS of the production has a matching node in the CDFG (in the CF sense) and if the connectivity between the LHS nodes is identical to that between the corresponding nodes in the CDFG. Extra CDFG connections are allowed if the corresponding edge in the LHS is an input or output of the subgraph. Graph matching has been shown to be a NP-complete problem [AhoA74]. Thus an exhaustive (i.e. inefficient) check must be performed to

confirm if a CS production is applicable. For this reason, the number of nodes on the LHS and the number of CS productions should be kept small to have reasonable program execution times.

After selecting one of the applicable productions (see Chapter 6), the chosen production is applied by substituting the RHS for the subgraph matched to the LHS. If the LHS is a subgraph, the corresponding subgraph in the CDFG is first reduced to a single node whose inputs and outputs are the edges matched to the LHS subgraph inputs and outputs. The description of this node, or the node matched to the LHS of a CF production, is replaced by the description of the RHS of the production. If the RHS is a subgraph, it is expanded into the CDFG. This process implicitly matches the inputs and outputs of the LHS and RHS. The syntax of a side of a production has been made identical to the syntax of a CDFG SG node to simplify the substitution process (see Appendix 1).

A cost function was not developed for control nodes. Thus each control node type (eg. loop_begin) is realized by calling a corresponding control procedure which modifies the CDFG and CG appropriately.

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1908a
ANSI and ISO TEST CHART No. 2

6.0 COST BASED PRODUCTION SELECTION

In any rule based system, there is the possibility of having more than one rule applicable at any time. Sometimes the rules are such that all may be applied. Often however, the rules are mutually exclusive; i.e. the application of one rule makes another invalid. In a graph grammar system, rules are applicable because their LHS matches a subgraph of the CDFG. If two productions have their LHS's matched to subgraphs which do not intersect, then both may be applied independently. If the LHS's are matched to subgraphs which intersect (have at least one node in common) then the application of one production makes the other invalid. This exclusion arises because applying a production removes all nodes in the LHS. Rules may also conflict if their RHS's intersect. For example, ~~two~~ rules may specify that ~~two~~ operations may both be realized on the same hardware cell.

The frequency and type of rule conflicts determines the complexity and style of the rule selection algorithm. Simple conflicts may be resolved by simple heuristics. For example, in the CMU DAA, rule conflicts are settled by choosing the most specific rule; i.e., the rule with the most conditions in the antecedent [KowT84]. In a graph grammar approach, there are no global rules, only local productions. All productions specify realizations for subgraphs in the CDFG and thus may be considered equally "specific". (Arguably, a rule could be defined to be more

specific if its LHS contained more nodes, but a larger LHS does not necessarily imply a larger cost reduction.)

In this thesis, productions are selected on the basis of a cost assigned to each production. This cost is proportional to the cost of the additional circuitry required if the production is applied. This chapter discusses the cost formulation, and the selection of productions within a clock cycle. The effect of production cost on operator scheduling is discussed in the next chapter.

6.1 Terminal Production Cost

The cost of a terminal productions is the sum of three sub-costs: the cost of the cell specified on the RHS, the cost of controlling the cell, and the cost of the additional interconnect required to connect an operator cell to the appropriate registers (or vice versa). Since these costs depend on the existing circuit, a separate cost is calculated for each instance of the cell type given on the RHS of the production. In all cases, the cost is incremental in that it represents the cost of additional hardware required.

6.1.1 Cell Cost

The cell portion of a terminal production cost represents the added cost of using a given cell instance to realize an operation. There are three types of cell cost: existing, new and replacement.

The cell cost of using an existing operator cell is always 0 because the cost of the cell has been considered when the cell was first added to the circuit. However, the total cost of realizing an operation on an existing cell may not be zero as interconnect may have to be added (see Section 6.1.3).

The new cell cost represents the cost of adding a new instance of a cell to the circuit. This cost is calculated by calling a user specified cost function with the cost parameters specified in the library description of the cell. This approach allows different parameters to be used to describe important factors in different technologies. For example, power consumption is important in bipolar technologies, but not in CMOS. The user cost function may also depend on run specific data. This allows the designer to vary the relative weighting of the cost parameters (eg. area and power) and recompile to explore the design space.

In the prototype implementation, only the area of the circuit was considered in calculating the cost. The cost parameters listed in the cell library were the actual cell area and the number of control inputs to the cell. The new cell cost was the sum of the cell area and an estimate of the area of ROM required to control the cell.

The final type of cell cost is a replacement cost. It represents the cost of replacing an existing cell with a more powerful cell. For example, an adder could be upgraded to an ALU to realize a subtraction. Similarly, a 4-bit

register could be widened to store an 8-bit result. (The set of cells a given cell can replace is determined when the cell library is read in by comparing the operations each cell can realize.) The replacement cost is defined to be the difference in the cell cost of the cell added and the cell it replaces. Replacement costs are calculated by calculating the new cell cost of each cell and subtracting.

b.1.2 Control cost

The control cost represents the overhead of circuitry added to control the data path. This includes the control path and hardware added to the data path. The explicit control overhead in the data path is the switching circuitry (multiplexers, bus drivers, etc.) required to reconfigure the data path. This is easily accounted for during data path design by including the cost of these cells in the interconnect area estimate (see next section). The remainder of the control cost represents increases in the area of the control path.

The control path is generally realized as a finite state machine or as a microcode memory with associated microword decoding and next address calculation circuitry. The prototype allocator assumes a microword type controller. The size of this controller can be decomposed into the area of the ROM required to store the microprogram, and the area of the logic required to access and decode each microword. The area of the ROM address circuitry depends on the size of

the algorithm (address word width and memory address decoder) and on the type of branching required (complexity of next address calculation). The microword decoding circuit area depends on the degree to which the microword has been encoded. These factors are not considered at present.

The control cost calculated by the prototype allocator represents the increase in the size of the control memory. If a production lengthens the algorithm, then this increases the length of the control memory. The associated control cost is the number of additional cycles multiplied by the area required to store a control word in ROM. Alternately, the production may increase the width of the control word. This may be caused by adding a programmable cell, or by increasing the number of inputs to a multiplexer. The cost calculated for a width increase is the product of the bit width increase, the area of a ROM bit, and the total number of cycles in the algorithm (length of microcode store).

6.1.3 Interconnect Cost

The area occupied by interconnect in an LSI circuit is often comparable to the active cell area. This fact leads designers to employ architectures which reduce the amount of interconnection. For example, hardware is sometimes duplicated because the active area added is less than the interconnect area required to distribute the original signal. This philosophy is incorporated into Elf by including area estimates of the new interconnect required in the cost func-

tion. There is one cost calculated for connecting the inputs to the cell and another for connecting the outputs. In all cases, this calculation accounts for the width of the data word.

The area of an input connection can be more accurately estimated because the cell storing (or generating) the input is known. The estimate is calculated using a combination of global and local parameters of the circuitry allocated to data. The interconnect length assuming random cell placement (RIL) is first calculated. Assuming unblocked Manhattan routing, ie, direct vertical - horizontal wiring, the distance between two points, (x_1, y_1) and (x_2, y_2) , is $|x_1 - x_2| + |y_1 - y_2|$. For a square chip of side S:

$$\begin{aligned}
 \text{RIL} &= \frac{\int_0^S \int_0^S \int_0^S \int_0^S |x_1 - x_2| + |y_1 - y_2| \, dx_1 \, dx_2 \, dy_1 \, dy_2}{\int_0^S \int_0^S \int_0^S \int_0^S dx_1 \, dx_2 \, dy_1 \, dy_2} \\
 &= 2/3 \, S \\
 &= 2/3 [(\text{cell area}) + (\text{interconnect area})^{**0.5}] \\
 &= 2/3 [CA + NC * \text{RIL}]^{**0.5}
 \end{aligned}$$

where NC is the number of connections and CA is the total cell area. Rearranging and applying the quadratic formula:

$$\text{RIL} = 2/9 [NC + (NC^{**2} + 9 * CA)^{**0.5}]$$

The RIL models the increase in average interconnect length as the size of the circuit increases. However, it overestimates the interconnect length because cells are not placed at random and because connections on the same net are not routed independently.

The purpose of any placement algorithm is to arrange the cells so that the amount of interconnect used is small. Thus, the average interconnect length (AIL) should be less than the RIL. The final distance between two cells is determined by the connectivity of the circuit. Connections between the two cells cause the placement algorithm to place the cells close together. Connections to other cells tend to increase the cell separation. Finally, the cell separation is influenced by the circuit architecture. Large random logic circuits tend to have a large AIL because each cell is connected to many other cells. Structured architectures with regular data flow have fewer connections and the cells may be placed according to the data flow to further reduce the interconnect.

The estimation of the cell separation must be matched to the particular placement algorithm which will be used to lay out the circuit. In the absence of a specific placement algorithm, the following formulae were developed to estimate cell separation:

$$ACD = RIL * RD$$

$$RD = \frac{NCI + NCO}{(ACI + ACO) * NCB} * \frac{1}{RF}$$

where RD represents the cell separation relative to random placement, NCI and NCO are the number of connections to the input cell and output cell in the connection, ACI and ACO are the average number of connections to the input and output cell classes, NCB is the number of connections between the two cells, and RF is a regularity factor. RF is a user specified parameter which reflects the desired ratio of total interconnect area to total active area. Specifying a large value for RF causes the allocator to neglect the interconnect area and produce a design with minimum active area. A small value for RF biases the allocator towards adding active circuitry before increasing the amount of interconnect.

In addition to cell separation, interconnect area is influenced by net size. When circuit fanout or fanin is greater than one, the connections in the net will share a portion of the routing path with each other. To add a new connection to a net, the interconnect need not run from the cell output to the cell input, only from the new I/O point to the nearest point on the existing net. Elf assumes that the net area (NA) initially grows as the square root of the fanout or fanin (F). When the net is large, a small linear growth is assumed. This is modelled by:

$$NA = 0.9 * F^{**0.5} + 0.1 * F$$

Thus, the fraction of the net area associated with a new connection (NF) is:

$$NF = AIL * (0.1 + ([F+1]**0.5 - F**0.5))$$

The final parameter included in the input interconnect area is the increase in mux area. The actual mux size increases linearly with the number of mux inputs. The mux control is assumed to be binary encoded, and thus, increases as the log of the number of mux inputs. The control width increase is multiplied by the ROM length and the ROM bit area to represent the additional control ROM area.

The output interconnect cost cannot be estimated accurately as the destination of the connection has not been assigned to a cell. The output connect cost is estimated by multiplying the minimum number of connections which would have to be added by the average cost estimate of the input connections added in the last 10 cycles. In the case of an operator cell, one connection is tallied for each output not connected to a free register. When assigning an edge to a register, the minimum number of connections required is calculated by counting the number of operators accessing this value which cannot be realized on cells already connected to the output of the register.

6.2 Nonterminal Production Cost

The cost of a terminal production is clearly defined as the additional circuitry added to the CG if the production is applied. Nonterminal productions specify functionally equivalent operations and have no direct interaction with the CG. The exact influence, and thus the cost, of the production on the CG is determined by subsequent terminal productions which map the nodes of the LHS or RHS to the CG. The sum of the cost of these terminal productions is defined to be the cost of a side of the production. The cost of the production is defined to be the difference between the RHS cost and the LHS cost. If the cost is negative, then the RHS costs less than the LHS and the production should be applied.

Determining the cost of a nonterminal production with the same accuracy as was done for terminal productions requires excessive computer time. Each side of all applicable productions would have to be completely allocated to accurately determine the cost of subsequent terminal productions. Determining the cost of a side becomes a search of the derivation tree. If more nonterminal productions are included in these derivations, then the search may not end until the design is complete. Thus, this approach becomes an exhaustive search of the design space and can lead to an exponential execution time dependence.

To maintain acceptable execution times for cost estimation, the LHS and RHS are treated as isolated graphs. Further, below the first level, only context free nonterminal productions (i.e., productions whose LHS is a single node) are allowed. This reduces the computational complexity as each node is considered in isolation. There is no need to check all permutations and combinations of productions. Considering only context free productions also reduces the computation time since matching the single node LHS is much simpler than matching a subgraph LHS to determine if a production is applicable.

The cost of each side of the production is calculated as the sum of the individual node costs and an expected interconnect cost. The node cost is equal to the minimum cell cost (existing, new or replacement) available to realize that operation. (Each edge is assumed to have an available register and is assigned a 0 cost.) The number of new CG connections, NC, which will be added to the CG is given by:

$$NC = NN + NE * PN$$

where NN is the sum of the number of inputs and outputs to nodes realized on new cells, NE is the sum of the number of inputs and outputs to nodes realized on existing and replaced cells, and PE is the fraction of connections to existing cells requiring new interconnect in the last 10

clock cycles. The interconnect area increase expected is found by multiplying NC by the average area of interconnect added in the last 10 cycles.

6.3 Production Selection

The production costs discussed in the previous sections estimate the effect of applying a given production on the area, power consumption, etc. of the final circuit. Applying the cheapest production for each node does not necessarily lead to the minimum cost design. First, the minimum cost realizations may not all be applicable. Two nonterminal productions may be matched so that their LHS's intersect. Applying one makes the other invalid. Second, there may be resource contention. The minimum cost realization of two nodes may use the same cell at the same time. Finally, this approach neglects time multiplexing of cells between different operators. For example, the cheapest production may state that an adder be included in the circuit to realize an addition. However, choosing an ALU may reduce the total circuit cost as the ALU can later be used for other operations (eg. subtraction, shifting).

The goal of production selection is to minimize the sum of the cost of all productions applied. However, finding this minimum cost set of productions is computationally expensive. To maintain acceptable execution times, the problem is decomposed into scheduling the operations into

cycles (see Chapter 7) and selecting realizations for operations within each cycle. Within each cycle, the cost of determining the minimum cost realization is still an exponential problem. The following scheme is used to select a good, although not necessarily optimal, set of productions to apply in each cycle. The process consists of four distinct stages. The ordering of these stages is determined by the interaction of the production types, and by the architectural assumption that all results of operations are loaded into registers at the end of the clock cycle.

At the start of each cycle, the feasible nodes are checked to see if they can be realized by any nonterminal productions having a cost less than 0. If such productions exist, they are applied. Productions realizing the new feasible nodes are then determined. This process is repeated until no feasible nodes can be realized by negative cost nonterminal productions. These productions are performed before the terminal productions as they only alter the CDFG. There is no implicit time delay associated with the application of the production. Thus, nodes added to the CDFG from the RHS of a nonterminal production may be mapped to the CG by terminal productions during this clock cycle.

The next step is to choose terminal productions to realize the operations scheduled for this clock cycle (see Chapter 7). The method for this process is to select the minimum cost realization for each node which does not cause a conflict; i.e., uses a cell that has already been assigned

to realize another operation. Each lower cost realization for this node is then tested to see if changing a previous node assignment would reduce the sum of the costs. This exchange process is recursive and can be used to test for n-way exchanges where n is the user specified maximum recursion depth.

After the operator nodes have been assigned to cells, registers are selected to store each output of these operations. The results constrained to be stored in specific registers to match loop or branch conditions are assigned first. Register transfers required to empty specified registers are performed at this time. The remaining results are then assigned to registers using a "max-difference" approach described by Hitchcock [Hit83]. The cost difference between the two minimum cost registers available to store a value is first calculated. The value with the largest cost difference is assigned to its minimum cost alternative. The cost differences are recalculated and the process is repeated until all values are assigned to registers.

When register assignment is complete, the terminal productions are applied and the feasible operations for the next clock cycle are determined. Feasible control nodes are realized at this time as they may influence which operations are feasible in the next cycle. For example, when the branch at the start of a case block is executed, the top nodes of only the first branch are set feasible. When the join at the end of the case block is reached, the clock

cycle counter is reset to the branch time and the top nodes of the next branch are made feasible. If this is the last branch, then the output nodes of the join node become feasible. As mentioned previously, this action is performed by a user definable control procedure.

7.0 SCHEDULING OPERATIONS INTO TIMESLOTS

7.1 Two Classes of Timing Specification

There are two basic philosophies for considering execution time during the circuit design: "soft" timing parameters and "hard" timing constraints. Soft timing parameters influence the speed at which the circuit will operate, but do not specify the actual execution time. Typically, these parameters are introduced as a weight on the delay component of the cost function [HitC83]. A large weight biases the allocator towards selecting fast components. If the weight on the delay component is small, the allocator will choose components to minimize the circuit area and power consumption. This approach is well-suited towards stand alone systems, such as computers, where the operating speed must be considered, but the absolute delay does not influence the correctness of the algorithm.

When the circuit being designed must fit into a large system, the amount of time available to perform a portion of the algorithm may be determined by the system. For example, in process control, an error condition may have to be serviced within a specified time interval to avoid damaging the equipment. Hard timing constraints specify, in absolute units, the minimum and/or maximum execution time allotted for a portion of the algorithm, or the execution rate for a loop. Such constraints are necessary when the execution of

the algorithm must be coordinated with the rest of the system. Hard timing constraints are included in the circuit description and thus can be accounted for during functional simulation (see Section 4.5).

The techniques described in the preceding two chapters are capable of determining the minimum cost realization for operations. This is sufficient to design circuits where soft timing parameters are included in the cost function. Hard timing constraints could be met by performing several allocations with different soft timing parameters and choosing the cheapest one which satisfies the system requirement. A better approach is to extend the allocation technique to automatically meet hard timing constraints. This chapter discusses how urgency scheduling techniques can be utilized to satisfy hard timing constraints.

7.2 Timeslot Selection as a Scheduling Process.

The Elf allocator is restricted to designing synchronous circuits. Once the master clock frequency is set, any hard timing constraints can be converted into a number of clock cycles. There are several formulations and algorithms which can be employed to map the CDFG onto the CG in these timeslots. The formulation chosen is dependent on the method employed to build the circuit.

In a simple hardware translator, the operators present in the description are realized directly on cells. Thus,

all the operators in the algorithm are defined by the user and are known at the start of the allocation. This type of allocator can employ a two pass approach. First, a partitioning or allocation algorithm is used to divide the CDFG into timeslots. The partitioning criteria is typically to minimize the number of operators of each type in each timeslot. The second pass assigns the operations in each timeslot to cells.

The graph grammar approach developed in Chapters 5 and 6 allows the allocator to rewrite the operations contained in the algorithm using nonterminal productions. This rewriting process has several effects on the circuit requirements and on the timeslot assignment. First, rewriting an operation in terms of other operations may remove the necessity to include a cell which directly realizes the operation. Graph rewriting may also be employed to reduce the number of cells of a given type. For example, an algorithm containing 11 shifts which must be performed in 10 clock cycles may be mapped onto a circuit with one shifter if one shift left is converted into an addition of the number to itself. Finally, graph rewriting may change the length of the algorithm. This often occurs when an operation is decomposed as when an 8-bit addition is rewritten as two sequential 4-bit add-carries.

This dynamic nature of the CDFG limits the effectiveness of a two pass approach. A scheduling formulation allows the assignment of operators to timeslots and the

mapping of operators to cells to be performed concurrently. The scheduling parameters can be continually updated to account for the present status of the CDFG and CG. Further, a scheduling approach implicitly satisfies the precedence relationships in the algorithm and easily accommodates nested timing constraints.

Hardware compilation is formulated as a scheduling process in the following manner. Each operation in the algorithmic description is viewed as a process to be scheduled. Each operator cell in the circuit is viewed as a resource which may be used to realize specified operations. Each register is viewed as a storage site. The design process is viewed as scheduling each operation onto a hardware cell in a timeslot such that the precedence relations are maintained, the timing constraints are met, and the cost of the circuit is minimized.

The precedence relations are maintained by scheduling only feasible nodes at each point in the allocation. A CDFG node is defined to be feasible for execution in a timeslot if all of its inputs have been calculated. (At the start of the allocation, only nodes with no inputs are feasible.) The feasible nodes for the next timeslot are the set of nodes feasible in this timeslot which were not executed in this timeslot, plus all nodes which become feasible because they access the outputs of nodes executed in this timeslot. The latter is determined by checking the nodes accessing

values generated in this timeslot to see if all of their inputs have now been calculated.

The scheduling algorithm used in Elf is a modified urgency scheduling technique [TheA76]. Each node in the CDFG is assigned a weight equal to the length of the longest string of operations from the node to the end of the timing constraint interval (see Fig. 3.3-2). The node weight thus represents the minimum number of clock cycles required to satisfy both the precedence and timing constraints. During the allocation, each feasible node is assigned an urgency factor, U , equal to the ratio of the node weight and the number of clock cycles remaining until the end of the time interval. Thus, the urgency represents the relative importance of executing the operation in this cycle with respect to meeting the timing constraints. An urgency equal to one implies that the operation must be executed in this clock cycle or else the timing specifications will be violated. If the urgency is less than one, the node is scheduled for execution based on the cost of realizing the node on available hardware (see Section 7.4). An urgency greater than unity implies that the timing constraint cannot be met. The allocation is restarted with more hardware or a higher clock frequency.

7.3 Weighting the CDFG

The weight of a node is calculated by adding the minimum number of clock cycles required to realize the node to the maximum of the weights of the nodes accessing the output of this node. The minimum number of cycles required for an operation may be determined from the productions available to realize the operations, and the master clock frequency. However, for simplicity, the prototype allocator assumes each operator can be realized in one clock cycle.

The number of clock cycles required to realize a control node is dependent on the design of the circuit's control path. The prototype allocator reads the duration of each control node from a user specified lookup table. In addition, the weight of a loop node is increased by a factor equal to the number of loop iterations times the length of one loop iteration. In the case of "for" loops, the number of loop iterations is calculated from the loop parameters. When the number of loop iterations cannot be determined, as in a "while" loop, the designer is prompted to specify the number of loop iterations.

Other types of variable weight nodes include rendezvous and procedure call. Rendezvous are assigned a duration from a lookup table as are control constructs (i.e., the delay before the rendezvous begins is ignored). Procedure calls are assigned durations equal to the weight of the procedure body. If the procedure will be realized by a microcode

branch and return, then the branching delay is added to the body weight.

Each node is assigned a separate weight for each timing constraint which encloses the node. The weights are represented as a timing constraint name - corresponding weight pair. The weight pairs are stored in a list as a property of the node. The timing constraints enclosing a node are identified as part of the weighing processes discussed below. When a timing constraint is weighed, the constraint name is added to the list of enclosing constraints on the property list with a corresponding weight of 0. Passing the node weight consists of placing all timing constraints of the node being weighed on the constraints list of all input nodes. The numerical weight corresponding to each constraint on the input node is set to the maximum of the weight from the node being weighed, and the weight already passed to the input node from other, previously weighed, nodes. Thus all enclosing constraints are passed. When a timing reference is weighed, all timing constraints which specify this node as their reference are removed from the constraint list. Thus, nodes above the reference (which are not enclosed by the constraint) are not identified with the constraint.

Weights are assigned to CDFG nodes in two phases. All nodes are assigned an initial weight during the preprocessing of the CDFG. This initial weight assignment is accomplished by traversing the graph in a direction opposite to

that performed during allocation. The process begins by identifying all nodes with no outputs, assigning them a weight of 0, and placing them on a "ready to weigh" list. The node at the front of the list is "weighed" as discussed above. The weight of the node is passed to all nodes on the input list of this node. Each input node is checked to see if it is ready to weigh; i.e., if the rest of the nodes accessing the input node have been weighed. The weighing process terminates when the ready to weigh list is empty.

The second phase of node weighing occurs during the allocation process. Whenever a nonterminal production is applied, or a procedure is expanded in line, new nodes are added to the CDFG. Weights must be assigned to these new nodes and any affect of the new nodes on the weights of existing nodes must be accounted for.

The new nodes are also weighed using a graph traversal technique. The weight of nodes accessing the outputs of the RHS of the production are passed to the new CDFG nodes generating these results. New nodes which have all their outputs accessed by existing nodes are placed on the ready to weigh list. The weighing of the new nodes then proceeds as in the preprocessing phase until the ready to weigh list is empty. The weighing is limited to the nodes added from the RHS of the production by not placing any existing nodes on the ready to weigh list.

The change of the weight of nodes above the new nodes is of no consequence to this allocation attempt as these

nodes have already been assigned realizations. However, if the allocation is restarted (see Chapter 8), the weight change must be considered. This is accomplished by reweighing the CDFG as in the preprocessing stage, with each allocation restart.

7.4 Assignment of Operators to Timeslots

Formulating timeslot selection as a scheduling process allows a combined timeslot and production selection algorithm to trade the cost of realizing a node in this clock cycle against the probability of satisfying hard timing constraints. The cost function developed in Chapter 6 for selecting terminal productions within a timeslot is still the basis for the allocation. The production cost is combined with timing information by accepting higher cost realizations for urgent nodes.

The assignment of CDFG operations to timeslots creates a partitioning of the CDFG. Each timeslot boundary forms a cut across the graph (see Fig. 7.4-1). The edges cut by the timeslot are values currently stored in registers. Nodes above the cut have already been assigned realizations; nodes below the cut have yet to be allocated. The set of nodes which have all of their input edges crossing the cut are the feasible nodes for this timeslot. By scheduling nodes for execution in this timeslot, the cut for the next timeslot is implicitly generated.

Nodes realized in timeslot (N-1)

Nodes feasible for execution in timeslot N which are realized in timeslot N.

Nodes feasible for execution in timeslot N which are not realized in timeslot N.

Values stored in registers at start of timeslot N

Values stored in registers at end of timeslot N.

Fig. 7.4-1: Timeslot boundary viewed as a cut across the CDFG.

The selection of the subset of the feasible nodes which will be realized in a timeslot is a three pass process. The passes are organized to give the most urgent nodes priority in access to use the hardware resources in the CG. In the first pass, all nodes with $U = 1$ are scheduled for this timeslot. These operations must be performed now in order to meet the timing constraints. Realizations are assigned to these operations based on the cost of the productions available to realized the nodes (see Section 6.3). When a realization requiring new hardware is selected, a second decision algorithm is used to decide exactly which cell will be added (see Chapter 8). If no nodes have $U = 1$, then, the node with the largest urgency factor is scheduled for this timeslot. This insures that there is at least one operation realized in each timeslot.

The remaining two passes schedule nodes with $U < 1$ which have little or no cost penalty associated with early execution. This allows the allocator to more fully utilize the resources of the CG. As a result, addition of active cell area to realize a large number of operations which have all been postponed until the end of the time interval is avoided. In both of these passes, only existing operator cells may be used although new interconnect may be added. New registers may be added in the second pass, but not in the third.

The second pass considers nodes with an urgency $1 > U > U_c$ where U_c is a critical urgency specified by the designer. The node is scheduled for this timeslot if:

$$C_n < C_m * (1 - U_c) / (1 - U)$$

where C_m is the minimum cost available to realize this node on the CG and C_n is the cost of realizing the node in this timeslot considering the hardware used to realize other nodes already scheduled for this timeslot. C_n is given by:

$$C_n = C_a + C_r$$

where C_a represents the cost of realizing this node on an available cell; i.e., the cost of realizing the operation on hardware not assigned in the first pass but including an n-way exchange of hardware realizations. C_r is an adjustment to account for the difference between the number of registers freed and the number used by realizing this operation at this time. The cost of the registers is estimated as:

$$C_r = A_r / \max[1, FR]$$

where FR represents the number of free register and A_r represents the change in the active area of registers used if this operation is realized. If A_r is negative, the oper-

ation frees more registers than are required to store the values it generates. Ar is divided by FR to weight the area gain (penalty) by the need for (availability of) free registers. Thus, when there are few free registers, the use (creation) of free registers is weighted more heavily with respect to the addition of interconnect.

The third pass considers nodes with $U < U_c$. These nodes are scheduled for execution in this timeslot only if there is no associated cost penalty; i.e., if $C_n \leq C_m$ where C_n and C_m are as defined above. After the third pass, all the outputs of the scheduled nodes are assigned to storage locations and the productions are applied.

This three pass approach to scheduling introduces a hierarchy with respect to the utilization of silicon in the IC. Operator cells are added only when required to meet timing constraints, registers are only added if $U > U_c$, while interconnect may be added independent of the urgency factor. This bias is valid since the yield reduction in IC processing due to silicon defects is greater in the active cell area; i.e., areas where transistors are created in the substrate.

However, disallowing the addition of cells to realize an operation until the operation's urgency is unity causes unnecessary hardware to be added near the end of the time interval. Once an operation in a path reaches unity urgency, then all the timing margin in the path has been used. All subsequent operations in the path will also have an

urgency of one, and there is no longer any scheduling freedom. This may force a large number of operations to be realized in one timeslot to satisfy the precedence and timing constraints. This, in turn may require the addition of more hardware than necessary if cells were added earlier allowing some operations to be performed before their urgency reached unity.

The number of unnecessary operator cells added is limited by two processes. First, if no cell exists which can realize an operation, then selecting a production which requires the addition of hardware is permitted when $U_i = 1$. Second, the mapping process is restarted before adding a cell to the CG if another cell has already been added during this mapping attempt and that cell could have been utilized before it was added. The restart condition is determined by comparing a list of operations which were feasible but not executed in previous timeslots with the list of operation that the first cell added is capable of realizing. If the two lists intersect, then the mapping is restarted to see if using the first cell earlier will prevent adding the second. If the two lists do not intersect, the second cell is added and the mapping continues without restarting.

7.5 Realization of Control Flow Constructs

The scheduling approach described thus far is sufficient to allocate descriptions in which each timing

constraint does not enclose any constructs that alter the program sequence; i.e., loops and branches. The basic scheduling approach must be extended to correctly handle these constructs. As described in Chapter 5, the method employed to realize these constructs is to call a control procedure corresponding to the control operation when the control node is identified as feasible.

7.5.1 Conditional Branches

Conditional branches are used to represent mutually exclusive, possibly execution dependent, control flow alternatives. The alternatives are mutually exclusive as only one of the alternatives is chosen and executed. An execution dependence arises if the decision of which branch to select is based on run-time parameters such as the value of a variable. Ada program constructs which lead to conditional branches are the if-then-else, case, and select statements. Conditional branches are represented in the CDFG by a "fork" node at the alternative selection point and a "join" node at the point where the alternative paths reconverge.

The allocation of conditional branches assuming a microsequenced architecture is relatively straight forward. As far as the assignment of operations to cells and values to registers is concerned, the only restriction is that the corresponding outputs of different branches are stored in the same registers. This allows the nodes below the join

node to be controlled by the same microinstruction, or FSM state, independent of which branch was executed. The corresponding outputs are stored in the same register by placing a constraint on the node output which forces the allocation algorithm to store the output in the specified register. If the register is already storing a value, the value is moved to another register by a register transfer.

From a timing viewpoint, the weights assigned to each node in each branch are valid for each branch independently. Thus, the branches can be allocated sequentially using the regular timeslot selection algorithm. When a fork node is encountered, the top nodes of the first branch are set feasible and the branch is allocated in isolation. When the corresponding join node is reached, the registers storing the outputs of the branch are identified and the outputs of the other branches are constrained to be stored in the same registers. The timeslot counter is reset to the branch time, the list of feasible nodes is set equal to the top nodes of the next branch and the process is repeated. When the last branch has been allocated, the output nodes of the join node are set feasible and the allocation continues. The lengths of the branches are matched by padding the short branches with idle wait cycles.

7.5.2 Loops

Loops are a special type of conditional in which the

branch choice is to repeat the loop or to exit. The prototype allocator assumes that the loop will be realized by a microcode loop or a repetition of a sequence of FSM states. Thus, all iterations of the loop will use the same mapping of the loop onto the CG. As with conditional branches, the data flow between iterations is matched by storing corresponding inputs and outputs of loop entry and exit nodes in the same registers.

However, unlike conditional branches, each loop iteration has a different weight with respect to timing constraints which enclose the entire loop construct. This is caused by the different number of iterations that occur after this iteration, but before the timing constraint. Utilizing the weights from any one iteration for the scheduling process is not satisfactory because these weights are incorrectly biased by the weight of operations between the end of the last loop iteration and the timing constraint.

Proper scheduling is obtained by automatically creating a dummy timing constraint which encloses exactly one loop iteration. During the allocation, the maximum time allotted for the execution of one iteration, T/I , is calculated as:

$$T/I = \frac{T_m (W(LB) - W(LE))}{W(LB) * NI}$$

where T_m is the time remaining until the end of the tightest enclosing maximum execution time constraint, $W(LB)$ is the

weight of the loop begin node (first iteration), $C(LE)$ is the weight of the loop end node (last iteration), and NI is the number of iterations. This formula equally divides the remaining time period between loop iterations and the nodes following the loop end. The assignment of nodes within the loop iteration then proceeds using the usual scheduling process based on the weight of each node with respect to the dummy timing constraint.

8.0 THE CIRCUIT GRAPH

The circuit graph, CG, is a connectivity graph, or netlist, of the circuit generated. The nodes in the graph represent cells in the circuit while the edges represent connections between cells. The CG format is based on the LISP list structure, but can easily be translated to any other netlist format. The BNF of the CG is given in Appendix 1.

The construction of the CG as a background process during the mapping of the CDFG is discussed in the next section. Section 8.2 presents the algorithm which automatically selects which hardware cells to add to the CG, while Section 8.3 describes methods in which the designer may manually add circuitry to the CG.

8.1 Building the CG

The algorithms discussed in the last three chapters concentrated on mapping the CDFG to the CG. However, these algorithms specify the CG implicitly through the selection of productions which require additional hardware; i.e., productions which have a cost greater than 0. The CG is made explicit by building the netlist data structure during the application of these productions.

The mapping of the CDFG to the CG within a timeslot is defined by the list of terminal productions to be applied.

This mapping is identified by matching the LHS of each terminal production to the operation to be realized and the RHS to an instance of a cell or register in the CG. Additional hardware is identified by specifying that the RHS is realized on a new instance of a certain cell type.

When a production is applied, the cell (register) specified on the RHS is checked to see if it is a new instance. If so, an instance of the specified cell (register) type is added to the CG and the statistics of the cells (registers) in the CG are updated. The RHS of the production is then identified as being realized on the new cell (register) instance.

The second stage of terminal production application involves connecting the cells as required by the production. The registers storing the inputs to the CDFG operation are first identified. If the output of the register is not connected to the specified input of the cell realizing the operation, then this connection is added. Similarly, each cell output is connected to the register assigned to store that value if this connection does not already exist. In all cases, the addition of interconnect includes enlarging the input multiplexers and updating the interconnect statistics.

8.2 Selection of Operator Cells

When a terminal production is selected which requires new interconnect or registers be added to the CG, the specified hardware is simply added. However, if a selected production requires a new operator cell, an evaluation of the CG and CDFG is performed to decide if an alternate cell should be added and/or if the allocation process should be restarted.

Productions requiring new cells may be selected for three reasons. First, no cell currently in the CG may be able to realize the operation. Second, the timing constraints may force an operation to be realized in this timeslot but all cells which can realize the operation are already assigned to realize other operations. Finally, the cost of adding the new cell and connecting it may be less than the cost of connecting components existing in the CG to realize this operation.

In the first two cases, the computational power of the circuit must be increased to meet the functional and/or timing requirements of the circuit description. In the last case, the cost of duplicating active hardware is estimated as costing less than adding interconnect. In all cases, the production is selected based only on local data; i.e., the production which adds the smallest cost increase to realize this operation in this timeslot is selected.

However, once a cell is added to the CG, it is available for use in all timeslots. Thus, some consideration should be given to the effect of the hardware added on the rest of the algorithm. It may be desirable to select an alternate production which maps the operation onto a more expensive cell with greater functionality. This may lead to a less expensive circuit by preventing other hardware from being added at a later time. For example, adding an ALU to the circuit, instead of an adder, to realize an addition may reduce the circuit cost since the ALU can be used to realize subtraction, shifting and logic operations in the future.

The tradeoff between cost and functionality is made using a weighted production cost. This weighted cost is equal to the actual cost of the production divided by the expected cell usage. The expected cell utilization is calculated as the number of operations in the CDFG that the cell could realize, divided by the number of existing cells which may also realize those operations. This factor is then normalized by dividing by the number of timeslots in the algorithm.

This weighted cost is also used to decide if an unused, existing cell should be replaced with a cell having greater functionality, rather than adding a new cell to realize this operation. As with the actual replacement cell cost, the weighted replacement cost is calculated as the difference of the weighted costs of the new and existing cells.

The cell, replacement or new, which is added to the circuit is the cell with the smallest weighted cost. If two cells have equal weighted costs, the one with the smaller unweighted cost is selected.

The productions selected for one timeslot may require the addition of more than one operator cell. This is sometimes due to delaying too many operations before adding circuitry leading to an exaggerated bottleneck. In such cases, the allocation process is restarted to see if the addition of one cell at the start of the mapping would prevent adding the other cells at this point. Which cell to add is determined by calculating the weighted cost of realizing each of the operations and selecting the cell with the minimum weighted cost. The allocation is restarted by removing the realization information from the CDFG, reweighing the CDFG and resetting the timeslot counter to 0.

Alternately, these additional cells may be required because the CG simply does not contain enough functionality to realize the algorithm. If the operations requiring the additional circuitry cannot be realized by cells already in the CG and no two can be realized by any one cell in the library, then the addition of one cell cannot prevent the addition of the others. In such cases, a separate cell is added to realize each operation and the allocation continues.

8.3 Manual Addition of Hardware

The algorithms presented thus far are capable of generating the entire CG from the CDFG. However, there are several benefits for allowing the designer to manually add hardware to the CG and/or to map operations from the CDFG onto the CG.

The primary problem with the scheduling/ graph grammar approach is myopia. At the start of the allocation, the nature of the weighted production cost leads to selecting a small cell which realized the most common operation. The existence of this cell then causes more single function cells to be added. Since cells may be added or upgraded, but not removed, the resulting circuit may be larger than necessary. Allowing the designer to initialize the CG directs the allocator towards the desired, possibly more efficient, circuit architecture. Similarly, larger than necessary circuits may be generated because the allocation is restricted to adding cells only when $U = 1$. The cells added must realize the present operation, although the bottleneck may often be more effectively removed by adding circuitry to realize other nodes earlier in the algorithm.

In addition to efficiency, manual intervention adds continuity and reduces the CPU requirements of the algorithm. Allowing the designer to specify circuitry allows for intelligent exploration of the design space. A new allocation is performed by starting from a core of a previous

iteration. This iteration process may involve simply removing hardware with low utilization to try to achieve a more efficient design, or it may be used to evaluate the cost of adding new procedures to the algorithm. Manual circuit addition reduces the execution time of the program by reducing the number of restarts.

The prototype allocator contains three methods for manually adding circuitry. First, the allocator may read a start CG from a user specified file. Second, the user is prompted to add cells, registers, and interconnect online at the start of the allocation. Finally, the allocator allows the designer to manually select the productions to be applied in this timeslot. Circuitry is added by selecting terminal productions with positive costs.

The first two methods are used to initialize the CG. The interactive mode; i.e., manual production selection, allows the designer to map critical sections of the CDFG to the CG. The designer may actually perform the entire mapping manually using the interactive mode. In this context, the allocator simply provides the designer with the urgency and cost data of the feasible nodes and then applies the selected productions. By limiting the designer to selecting valid productions for feasible operations, the correctness of the design is insured.

9.0 IMPLEMENTATION AND EVALUATION

9.1 Prototype implementation

A prototype allocator was implemented in LISP to evaluate the algorithms presented in the preceding chapters. LISP was chosen primarily because the data structures for the CDFG, CG and productions can be easily represented using LISP lists (see Appendix 1), while temporary allocation information can be stored on the property list of the atom. Further, the standard list processing functions in LISP greatly simplified the implementation of graph processing routines. Finally, LISP easily accommodated the changes in the algorithms, program structure and data structures which occurred frequently since the prototype was developed in parallel with the algorithm. The entire allocator to translate the CDFG to the CG constituted only 6300 lines of LISP code.

To design a circuit using the prototype, the Ada code must first be manually translated to the CDFG format. The input CDFG contains only the node description and the list of inputs. The list of outputs is generated automatically to insure consistency with the list of inputs. The other input files are the cell library and nonterminal production library. Terminal productions are generated automatically from the cell description contained in the cell library.

At the start of the allocation process, the designer is prompted for the critical urgency, interconnect regularity factor, a start circuit, and the first timeslot to be manually allocated. The allocation then proceeds as discussed in the preceding chapters. During the allocation, the productions applied and the cell utilization in each timeslot are printed to a trace file (or on the terminal if so specified by the designer). When the allocation is complete, the CG and final CDFG containing the mapping information are stored on disk.

9.2 FIR Filter Example

The ability of the allocator to compile simple program constructs was presented in Example 3.1. The example presented in this section demonstrates how the circuit generated can be influenced from the functional description (CDFG). The next section presents a larger example involving multitasking.

Fig. 9.2-1 presents an Ada description of a four stage finite impulse response filter. The data signal is specified as being an analogue voltage while the filter coefficients are represented as 8-bit integers. The only timing constraint specified is the loop iteration period of 125 usec (corresponding to an 8 kHz sampling rate).

The direct translation of the Ada code for the FIR filter onto the CDFG format is given in Fig. 9.2-2.

task body FIR_FILTER is

type ANALOG_VOLTAGE is REAL range -12 .. 12;

type COEFFICIENT is INTEGER range -128 .. 127;

SIGNAL : ANALOG_VOLTAGE;

TEMP : array (0..3) of ANALOG_VOLTAGE;

COEF : array (0..3) of COEFFICIENT := (127, -109, 98, 23);

begin

loop

TIMING.PERIOD (124.9E-6, 125.1E-6);

get (ANALOG_INPUT, SIGNAL);

for N in 0..2 loop

TEMP(N) := TEMP(N+1) + SIGNAL * COEF(N);

end loop;

TEMP(3) := SIGNAL * COEF(3);

put (ANALOG_OUTPUT, TEMP(0));

end loop;

end FIR_FILTER;

Fig. 9.2-1: Ada Code for Voice Frequency FIR Filter

Fig 9.2-2: CDFG for voice frequency FIR filter.

Fig. 9.2-3: Circuit automatically generated to realize voice frequency FIR filter.

Fig. 9.2-3 represents the circuit which is automatically generated from that CDFG by the prototype allocator. The direct implementation of the coefficient loop within the sampling loop results in a completely sequential implementation requiring only one multiplier and one adder. Since the signal is an analogue voltage, the multiplications are realized by a multiplying D/A converter and the addition by an analogue adder. The results of these operations are stored in sample-and-holds while the digital coefficients and the loop counter are stored in digital registers. The sequential mapping plus loop overhead result in the inner loop requiring 6 clock cycles per iteration and a total of 28 time steps to calculate of each filter output. The master clock frequency was specified as 400 kHz, thus the controller delays until timeslot 50 before starting the next iteration.

If the filter is to operate above voice frequencies, the fact that the master clock frequency must be 28 times the sampling rate would be impractical. In high frequency signal processing, it is common to build pipeline structures which process the signal at the desired frequency, but which have a latency of several sampling periods. This can be specified in the Ada code by having a separate loop rate and loop latency specified as shown in Fig. 9.2-4(a). When the Ada code is translated onto the CDFG, the difference between the loop latency specification and the loop rate specification is realized by replicating the inner loop in parallel

for each iteration. The array accessed by the loop counter becomes four discrete values which are inputs and outputs of the outer loop. The loop delay corresponds to the sampling rate. The Ada code equivalent of the FIR filter with an expanded inner loop is given in Fig. 9.2-4(b). The CFG corresponding to this code is shown in Fig. 9.2-5.

If a clock frequency much higher than the sampling rate is specified, the allocator generates a circuit similar to that given in Fig. 9.2-3, but without the circuitry to realize the inner loop counter. If the clock frequency is specified as being twice the sampling frequency, the allocator is forced to create a very parallel realization as shown in Fig. 9.2-6. The circuit is almost a pipeline, except that each loop iteration is two clock cycles long and the first multiplier is used twice. To have the allocator generate the standard FIR pipeline implementation (Fig. 9.2-8), the loop length must be reduced to one clock cycle. This is achieved by splitting the product and sum from each iteration (see Fig. 9.2-7). The process of automatically expanding inner loops by replicating iterations in parallel to realize differing throughput and latency timing specifications should be relatively simple to automate. Further parallelization of loops would be more difficult.

This example demonstrates the inefficiency which is sometimes created by separating the design of the data path and the control path. The loop "forever" is realized by passing the value "TRUE" to the microprogram counter for a

task body FIR_FILTER is

type ANALOG_VOLTAGE is REAL range -12 .. 12;

type COEFFICIENT is INTEGER range -128 .. 127;

SIGNAL : ANALOG_VOLTAGE;

TEMP : array (0..3) of ANALOG_VOLTAGE;

COEF : array (0..3) of COEFFICIENT := (127, -109, 98, 23)

begin

loop

TIMING.PERIOD (9.99E-6, 10.01E-6);

TIMING.REFERENCE (START_CALC);

get (ANALOG_INPUT, SIGNAL);

for N in 0..2 loop

TEMP(N) := TEMP(N+1) + SIGNAL * COEF(N);

end loop;

TEMP(3) := SIGNAL * COEF(3);

put (ANALOG_OUTPUT, TEMP(0));

TIMING.CONSTRAINT (START_CALC, nil, 120E-6);

end loop;

end FIR_FILTER;

Fig. 9.2-4: Ada Code for High Frequency FIR Filter

(a) Throughput and latency specified

task body FIR_FILTER is

type ANALOG_VOLTAGE is REAL range -12 .. 12;

type COEFFICIENT is INTEGER range -128 .. 127;

SIGNAL : ANALOG_VOLTAGE;

T0,T1,T2,T3 : ANALOG_VOLTAGE := 0;

C0 : COEFFICIENT := 127; C1 : COEFFICIENT := -109;

C2 : COEFFICIENT := 98; C3 : COEFFICIENT := 27;

begin

loop

TIMING.PERIOD (9.99E-6, 10.01E-6);

get (ANALOG_INPUT, SIGNAL);

T0 := SIGNAL * C0 + T1;

T1 := SIGNAL * C1 + T2;

T2 := SIGNAL * C2 + T3;

T3 := SIGNAL * C3;

put (ANALOG_OUTPUT, T0);

end loop;

end FIR_FILTER;

Fig. 9.2-4: Ada Code for High Frequency FIR Filter

(b) Ada code for unwound inner loop

Fig. 9.2-5: CDFG for FIR filter with inner loop unwound.

Fig. 9.2-6: Circuit automatically generated to realize high frequency FIR filter.

Fig. 9.2-7: CDFG of FIR filter which can be executed in one clock cycle.

Fig. 9.2-8: FIR filter pipeline generated by Elf.

"branch on true" instruction. This can be corrected by having the control path allocator remove such circuitry from the data path, or by rewriting the procedure which realizes loops to recognize "forever" loops and treat them differently than regular loops. Another inefficiency is resetting the loop counter by loading 0 from the control path, rather than using a local reset function. After the reset, the address count is incremented on a separate cell, rather than upgrading the register to one which can increment. Both of these are caused by the allocator not recognizing that cells can both perform operations and store results. This is a deficiency of the prototype, not of the general methodology.

9.3 TDM Switch

This section presents a design of a time division multiplexed (TDM), 30 line private branch exchange (PBX) telephone switch. This example was chosen for several reasons. First, the system is not a copy of an existing circuit. Thus, it provided experience in designing new circuits at the Ada level. Second, the TDM PBX has predominantly data transfer and control processing operations with predefined operating frequencies. Thus, the applicability of the algorithms in these areas was tested. Finally, the example naturally divides into two processes. The ability to describe such processes with separate Ada tasks was one

of the reasons cited in Chapter 4 for selecting Ada as the description language.

The structure of the system that the switch is part of is shown in Fig. 9.3-1. The system consists of a TDM switch and up to 30 telephone lines. The lines with addresses between FIRST_CO_LINE and 31 are interfaces from the PBX to a telephone central office used for external calls. Line numbers below FIRST_CO_LINE represent internal lines (or may be unused). Each line receives data from the switch on a 2.048 MHz bus (A-bus) and sends digitized voice data to the switch on another 2.048 MHz bus (B-bus). Each CODEC reads and writes data at the position in the frame equal to the CODEC's hardwired address. Thus, the CODEC with address 3 reads and writes data position 3. Communication between lines is achieved by having the switch reorder the signals received on the B-bus. The signal read from each line is sent on the A-bus in the position equal to the address of the other line in the connection.

The first two positions in the frame are used to transfer control information between the switch and the individual lines. On channel 0, the switch sends 3 synchronization bits and a 5-bit address of the line which will be sent control information in channel 1 of this frame. The lines are polled sequentially to pass control information resulting in a control data rate of 2.13 kHz. The A-bus and B-bus are skewed by one channel to simplify data processing. The frame formats for the buses are given in Figure 9.3-2.

→ data flow

Fig. 9.3-1: Block diagram of PBX system.

Fig. 9.3-2: Frame format for A-bus and B-bus.

The PBX is decomposed into two tasks which communicate explicitly via a rendezvous and implicitly via the connection memory (see Fig. 9.3-3). The Ada code for the two tasks is given in Fig. 9.3-4. The task LINE_INT operates at a per channel rate. LINE_INT communicates with the CODEC's via two user designed hardware cells which perform the parallel to serial conversion. LINE_INT also passes control information between the buses and the task CONTROL in channel 1 and sends the synchronization and address information in channel 0.

The task CONTROL operates at a per frame rate. It processes the control information sent from each line to make and break connections. If the status of the line designates a call in progress, the address of the connected line is stored in the connection memory. If the call in progress is an external call, the line is connected to a CO-line interface which passes the information to the central office.

The task CONTROL maintains the status of each line at the corresponding address in the array STATUS. The line status is a record which specifies if the phone is on or off hook, if the phone is ringing, the tone returned to the ear-piece (eg. dial tone), and the dialing status (idle or the first digit dialed). When a CODEC or CO-line interface has no new control information to send to the switch, it sends its internal status. The task CONTROL compares the received status to the status stored locally. If the status

Fig. 9.3-3: Decomposition of TDM switch into two tasks.

Fig. 9.3-4: Ada Code for TDM Switch.

```
package body SWITCH is

  type DATA_WORD is integer range 0..255;
  type ADDRESS is INTEGER range 0..31;

  task LINE_INT;
    -- task which interfaces SWITCH to individual lines
    -- and performs reordering of data streams

  task CONTROL is
    -- task which performs control processing

    entry DATA (R_ADDR : in ADDRESS,
                 R_WORD : in DATA_WORD,
                 S_WORD : out DATA_WORD);
  end CONTROL;

  CONNECTION_MEM : array (2..31) of ADDRESS;

  task body LINE_INT is

    DATA_MEM : array (2..31) of DATA_WORD;
    WORD_NUM, ADDR : ADDRESS := 0;
    RCV_WORD, SEND_WORD : DATA_WORD;
    FRAME_BITS : constant INTEGER := 5;

    procedure SIPO (WORD : out DATA_WORD) is
      -- user designed serial in/ parallel out port
      pragma HARDWARE_CELL;
      begin
        get (DATA_INPUT_FILE, WORD);
      end SIPO;

    procedure PISO (WORD : in DATA_WORD) is
      -- user designed parallel in/ serial out port
      pragma HARDWARE_CELL;
      begin
        put (DATA_INPUT_FILE, WORD);
      end SIPO;

    procedure SWITCH_DATA (ADDR : in ADDRESS,
                           IN_WORD : in DATA_WORD,
                           OUT_WORD : out DATA_WORD) is
      -- reorders data samples using data memory
      pragma IN_LINE; -- expand-in-line for efficiency
```

```

begin
  DATA_MEM ( CONNECTION_MEM (ADDR)) := IN_WORD;
  OUT_WORD := DATA_MEM (ADDR);
end SWITCH_DATA;

begin
  loop
    TIMING.RATE ( 0.48828E-6, 0.48828E-6);
    SIPO (RCV_WORD);
    WORD_NUM := (WORD_NUM + 1) mod 32;
    case WORD_NUM is
      when 0 => CONTROL.DATA (ADDR, RCV_WORD, SEND_WORD);
      when 31 => ADDR := (ADDR + 1) mod 32;
                  S_WORD := FRAME_BITS & ADDR;
      when others => SWITCH_DATA (ADDR, RCV_WORD, SEND_WORD);
    end case;
    PISO (SEND_WORD);
  end loop;
end LINE_INT;

```

task body CONTROL is

```

type RETURN_TONE is (SILENT,DIAL,BUSY,RING_BACK);

```

```

for RETURN_TONE use -- type enumeration
  (SILENT=>0, DIAL=>1, BUSY=>2, RING_BACK=>3);

```

```

type LINE_STATUS is

```

```

  record
    ON_HOOK : BOOLEAN := FALSE;
    RINGING : BOOLEAN := FALSE;
    TONE     : RETURN_TONE := SILENT;
    DIAL     : INTEGER range 0..15 := 16#F#;
  end record;

```

```

for LINE_STATUS use -- record storage assignment

```

```

  record at mod 1
    ON_HOOK at 0*BYTE range 0..0;
    RINGING at 0*BYTE range 1..1;
    TONE    at 0*BYTE range 2..3;
    DIAL    at 0*BYTE range 4..7;
  end record;

```

```

STATUS : array 0..31 of LINE_STATUS;

```

```

MESSAGE : array 0..31 of DATA_WORD := (others=>16#0F#);

```

```

DATA_CORRUPTED : array 0..31 of
  INTEGER range 0..3 := (others=>0);

```

```

RCV_ADDR : ADDRESS;

```

```

RCV_DATA : DATA_WORD;

```

```

IDLE : constant INTEGER range 0..15 := 16#F#;

```

```

EXTERNAL_DIAL_STATUS : constant LINE_STATUS

```



```

:= (TRUE, FALSE, DIAL, 16#9#);
INTERNAL_DIAL_STATUS : constant LINE_STATUS
:= (TRUE, FALSE, DIAL, 16#E#);
CALL_IN_PROGRESS : constant LINE_STATUS
:= (TRUE, FALSE, SILENT, 16#E#);
RESET_STATUS : constant LINE_STATUS
:= (FALSE, FALSE, SILENT, IDLE);
BUSY_TONE : constant TONE := BUSY;
RING_BACK_TONE : constant TONE := RING_BACK;
ON_HOOK_CODE : constant DATA_WORD := 16#FF#;
OFF_HOOK_CODE : constant DATA_WORD := 16#FE#;

procedure PROCESS_DIGIT (ADDR : in ADDRESS,
WORD : in DATA_WORD) is
DIGIT : INTEGER range 0..16;
CO_LINE_NUM : ADDRESS;
NO_FREE_LINE : BOOLEAN;

begin
DIGIT := WORD mod 16; -- last 4 bits of word
-- case STATUS(ADDR).DIAL is
when IDLE => -- first digit dialed
if DIGIT = 9 then -- check for free CO line
get(addr_pins, CO_LINE_NUM); -- first co line
NO_FREE_LINE := TRUE;
while CO_LINE_NUM =< 31 loop
if CONNECTION_MEM(CO_LINE_NUM) = CO_LINE_NUM then
-- line is available for external call
NO_FREE_LINE := FALSE;
else
CO_LINE_NUM := CO_LINE_NUM + 1;
endif;
end loop;
if NO_FREE_LINE then
STATUS(ADDR).TONE := BUSY_TONE;
else
STATUS(ADDR) := EXTERNAL_DIAL_STATUS;
CONNECTION_MEM(ADDR) := CO_LINE_NUM;
CONNECTION_MEM(CO_LINE_NUM) := ADDR;
STATUS(CO_LINE_NUM) := CALL_IN_PROGRESS;
endif;
elseif STATUS(ADDR).DIGIT < 3 then
STATUS(ADDR).DIAL := DIGIT;
else -- nonexistent number
STATUS(ADDR).TONE := BUSY_TONE;
endif;
when 9 => -- external call, pass digit to CO_line_int
MESSAGE(CONNECTION_MEM(ADDR)) := DIGIT;
when 0..2 => -- this is second digit of an internal call
CONN_TO := ADDRESS(10*STATUS(ADDR).DIAL + DIGIT);
-- type conversion to type ADDRESS
if STATUS(CONN_TO).ON_HOOK and
CONNECTION_MEM(CONN_TO) = CONN_TO then
-- complete connection and set phones ringing

```

```

        CONNECTION_MEM(CONN_TO) := ADDR;
        CONNECTION_MEM(ADDR) := CONN_TO;
        STATUS(ADDR).TONE := RING_BACK_TONE;
        STATUS(CONN_TO).RINGING := TRUE;
    else
        STATUS(ADDR).TONE := BUSY_TONE;
    endif;
    when others => -- dialing or transmission error
        STATUS(ADDR).TONE := BUSY_TONE;
    end case;
end PROCESS_DIGIT;

```

```

procedure CHECK_LINE (ADDR: in ADDRESS,
                     DATA: in DATA_WORD) is
-- records transmission errors,
-- after three consecutive errors, lights error display
begin
    if STATUS(ADDR) /= DATA then
        if DATA_CORRUPTED(ADDR) = 3 then
            DISPLAY_LINE_FAULT (ADDR);
        else
            DATA_CORRUPTED(ADDR) := DATA_CORRUPTED(ADDR) + 1;
        endif;
    else -- data correct, reset error counter
        DATA_CORRUPTED(ADDR) = 0;
    endif;
end CHECK_LINE;

```

```

procedure ANSWER_INCOMING_CALL (ADDR : in ADDRESS) is
-- stops phone ringing and ring back tone.
begin
    STATUS(ADDR) := CALL_IN_PROGRESS;
    STATUS(CONNECTION_MEM(ADDR)) := CALL_IN_PROGRESS;
end ANSWER_INCOMING_CALL;

```

```

procedure RESET_LINE (LINE_NUM : in ADDRESS) is
-- resets line when party goes on-hook
begin
    CONN_TO := CONNECTION_MEM (LINE_NUM);
    CONNECTION_MEM(CONN_TO) := CONN_TO;
    CONNECTION_MEM(LINE_NUM) := LINE_NUM;
    STATUS(LINE_NUM) := RESET_STATUS;
    if STATUS(CONN_TO).ON_HOOK then
        STATUS(CONN_TO) := RESET_STATUS;
    else
        STATUS(CONN_TO) := INTERNAL_DIAL_STATUS;
    end if;
end RESET_LINE;

```

```

        endif;
    end RESET_LINE;

    procedure SPECIAL_FUNCTION (ADDR : in ADDRESS,
                                CODE : in DATA_WORD) is
        -- procedure to realize special functions such
        -- as call forward, autodial, etc.
    end SPECIAL_FUNCTION;

begin
    loop
        TIMING.REFERENCE (START_PROC);

        -- receive control data and return message
        accept DATA (R_ADDR : in ADDRESS,
                     R_WORD : in DATA_WORD,
                     S_WORD : out DATA_WORD) do
            RCV_ADDR := R_ADDR;
            RCV_DATA := R_WORD;
            S_WORD := MESSAGE(R_ADDR);
        end accept DATA;
        TIMING.CONSTRAINT (START_PROC, nil, 3E-6);

        -- process control data received
        MESSAGE(RCV_ADDR) := STATUS (RCV_ADDR);-- default

        case RCV_DATA is
            when ON_HOOK_CODE => RESET_LINE (RCV_ADDR);
            when OFF_HOOK_CODE => if STATUS(RCV_ADDR).RINGING then
                                    ANSWER_INCOMING_CALL(RCV_ADDR);
                                else
                                    STATUS(RCV_ADDR) :=
                                        INTERNAL_DIAL_STATUS;
                                    MESSAGE(RCV_ADDR) := DIAL_TONE_ON;
                                endif;
            when 16#F0#..16#F9# => PROCESS_DIGIT (RCV_ADDR, RCV_WORD);
            when 16#FA#..16#FD# => SPECIAL_FUNCTION(RCV_ADDR, RCV_WORD);
            when others          => CHECK_LINE (RCV_ADDR, RCV_DATA);
        end case;

        TIMING.CONSTRAINT (START_PROC, nil, 100E-6);
    end loop;

end CONTROL;

end TDM_SWITCH;

```

Fig. 9.3-4: Ada Code for TDM Switch.

does not match for three consecutive receptions, a line fault indicator is displayed.

Because the prototype allocator cannot allocate multi-task descriptions, the two tasks were allocated separately. The rendezvous was realized directly by adding a cell "RV_CALL" or "RV_ACCEPT" to each task. Since there are only two tasks, the counter/bus structure described in Chapter 4 to realize the rendezvous can be reduced to a test-and-set flag. The example suggested a simple method for performing rendezvous simplification. The allocator creates a generic "RV_CALL" cell in a task for each entry that the task calls. The number of "RV_CALL" cells for each entry that exist at the end of the design define the maximum number of tasks which can call that entry. The exact realization of the rendezvous is then chosen based on the number of "RV_CALL"'s in existence as discussed in Chapter 4.

The circuits generated to realize the tasks LINE_INT and CONTROL are shown in Fig. 9.3-5 and 9.3-7 respectively. Both tasks must operate at the same clock frequency during the rendezvous. Considering only the timing constraints, the task CONTROL could operate at a much slower clock frequency after the rendezvous if there was a method for changing the clock frequency after the "end accept" statement, or if an asynchronous realization for the rendezvous was used.

The quality of the circuit shown in Fig 9.3-5 was improved by adding specialized cells and productions. For

the task LINE_INT, cells were added to perform the parallel in/serial out and serial in/parallel out interfaces of the switch to the A-bus and B-bus respectively. A nonterminal production was added to indicate that incrementing modulo 32 was equal to a 5-bit count. However, the inability of the allocator to combine storage and operator processing in one cell meant that the output of the counter had to be stored in a register. A nonterminal production was also used to signify that checking if ADDR=0 and ADDR=31 could be realized by a 5-input NOR and 5-input AND gate respectively. The quality of the circuit can be further improved by post processing to remove interconnect not used in the final mapping, and to remove the passing of the value "TRUE" to the control path to realize the "forever" loop (see Fig. 9.3-6). This inefficiency arises as interconnect was added to connect operators to registers in a previous mapping iteration. In the next mapping, values were assigned to different registers, leaving certain connections unused. Automating the removal of unused circuit components would be a simple matter of setting a flag when a component or interconnection was used. At the end of the final mapping, all components whose flag was not set would be removed.

The only special accommodations made for the translation of the task CONTROL was specifying that the ALU could perform logical operations, and decomposing record processing operations into byte operations. The variable type

Fig. 9.3-5: Circuit generated by Elf to realize task LINE_INT.

Fig. 9.3-6: Circuit from Fig. 9.3-5 with unused interconnect removed.

Fig. 9.3-7: Circuit generated by Elf to realize task CONTROL

LINE_STATUS is specified as a four component record which is packed into an 8-bit word. The selection of components within each word was translated into bit masking operations. Operations on these components were achieved by appropriate masking and then using an 8-bit operator. For example, the Ada code to test if a line is ringing is "if STATUS(ADDR).RING = TRUE". This is translated to the CDFG as ANDing the status byte with hex 40 and then testing if the result is equal to hex 40. In both tasks, the case statements were decomposed into nested if-then-else constructs to remove the necessity to calculate the branch index in advance of the fork.

The design time for this example was only two weeks. Most of this time was spent in system definition and generation of the CDFG from the Ada code. The main problem experienced in designing these tasks at the Ada level was caused by thinking of the target architecture while writing the description. Having a hardware design background, a simple hardware realization for certain functions was obvious. It was very tempting to write the Ada code so as to prompt the allocator to generate that architecture. A second case where too much knowledge of the allocator was detrimental was knowing the allocator's inadequacies. There was a tendency to avoid using Ada constructs which would not be elegantly realized by the allocator. In the end, these inhibitions were overcome by using special function cells and adding productions to the library. These solutions were

intended to be used and were the reason that the allocator was designed to be rule-based. The creation of larger production libraries would reduce the necessity for user intervention at this level.

9.4 Complexity Analysis

The translation of the CDFG to the CG consists of several processes as shown in the overview of the allocation algorithm (Fig. 9.4-1). The first process is to read and preprocess the cell and production libraries. This process grows linearly with the size of the libraries as each cell or production is considered independently, and only once. Similarly, reading and preprocessing the CDFG grows linearly with the size of the CDFG as each node and edge is treated only once. The next process is to weigh the CDFG. Again, each node and edge is traversed only once, resulting in a linear dependence of the weighing process on the size of the CDFG. However, the CDFG is reweighed with each restart of the allocation process, which, in the worst case, is equal to the number of operator cells in the CG (N_c). Thus, the total complexity of the weighing process is $O(N_c \cdot (N+E))$ where R is the number of restarts, N is the number of nodes and E is the number of edges.

The process of mapping the CDFG to the CG consists of a loop which is executed for each timeslot in which a node is realized ("while" loop in Fig. 9.4-1). The first process is

Read in cells and create terminal productions.
Read in and preprocess nonterminal productions.
Read in and preprocess CDFG.

For each restart, loop:

 Weigh CDFG.
 Find initially feasible nodes.

While (feasible nodes and not restart), loop:

 For each feasible node, loop:

 Decide if node should be realized
 by a nonterminal production.

 Apply nonterminal productions.

 For each feasible node, loop:

 Find minimum free cost realization for node.
 Attempt pairwise exchange with all nodes
 assigned to cells which would more cheaply
 realize this node.
 Decide if node will be realized in this timeslot

 If more than one cell added, restart mapping.
 Apply terminal node productions.

For all outputs generated in this timeslot, loop:

 Find cost of storing output in each free register

Until all outputs are assigned to registers, loop:

 Find 2 least expensive registers for each output,
 and calculate difference.
 Assign output with greatest cost difference to
 corresponding min cost register.

Apply terminal edge productions.
Realize feasible control nodes.

For all new feasible nodes
 (or all nodes if hardware added), loop:

 Calculate cost of all productions which realize node.

Print final CDFG and CG.

Fig 9.4-1: Pseudo-code for Prototype Allocator.

to choose and apply any nonterminal productions whose cost is less than 0. Let the maximum number of nonterminal productions applicable for any node be N_{np} and N be the number of feasible operations in this timeslot. Then, the maximum number of calculations required to check and select the nonterminal productions (N_{snp}) is:

$$N_{snp} = O(N_f * N_{np}) \quad 9.4(1)$$

(The complexity of this step can be reduced by considering each node when the cost is calculated, rather than in each timeslot. This would lead to a linear dependence on the number of nodes for each mapping.)

The next process is to select and apply the terminal productions. This process consists of finding the minimum cost production for each feasible node, checking a pairwise exchange of realizations with nodes already allocated, and then deciding if the node will be executed in this timeslot. The cost of using each applicable terminal production is calculated for each instance in the CG of the cell specified on the RHS of the production, for each cell instance in the CG which can be replaced by the cell specified on the RHS, and for adding a new cell of the type specified in the production. The maximum number of cells considered per terminal productions is thus $(1 + N_c)$ where N_c is the total number of operator cells in the CG. If N_{tp} denotes the maximum number of terminal productions applicable for any

node, the maximum number of cost entries per node is $N_{tp} * (1 + N_c)$. If the minimum free realization is not the absolute minimum cost realization, an exchange is tested for each realization cheaper than the minimum free one. There can be at most N_{na} such tests where N_{na} represents the number of nodes already assigned in this timeslot. Each exchange attempted costs as much as finding the minimum cost free realization of a node. Thus, the exchange process requires $N_{na} * N_{tp} * (1 + N_c)$ operations. Within each timeslot, all of the feasible nodes are tested. The total number of operations to select the terminal productions within a timeslot (N_{stp}) is thus:

$$N_{stp} = O(N_f * N_{tp} * (1 + N_{nr}) * (1 + N_c)) \quad 9.4(2)$$

where N_{nr} is the number of nodes realized in this timeslot.

After the nodes are assigned to cells, the outputs of these nodes are assigned to registers. The cost of storing each output in each free register requires at most $N_{out} * N_r$ calculations where N_r is the total number of registers and N_{out} is the number of outputs generated in this timeslot. Finding the minimum two storage costs for each output and the difference between them also requires $O(N_{out} * N_r)$ operations. However, this calculation is repeated after each output is assigned. Thus, the number of operations to assign the registers (N_{sr}) is:

$$N_{sr} = O((1+N_{out})(N_{out}*N_r))$$

9.4(3)

The realization of control nodes follows register assignment. Assuming that each control node is realized in constant time, this stage grows linearly with the number of control nodes.

The last process in each timeslot is to calculate the cost of realizing any new feasible nodes, or all feasible nodes if hardware has been added. From above, the number of cost calculations per node is $O(N_{tp} * (1 + N_c))$. For N_f feasible nodes, the total number of cost calculations in the timeslot (N_{cc}) is:

$$N_{ct} = O(N_f * (N_{tp} * (1 + N_c) + N_{np}))$$

9.4(4)

The total number of calculations per timeslot N_{pts} is given by:

$$N_{pts} = N_{snp} + N_{stp} + N_{sr} + N_{ct}$$

$$= O(N_f * N_{np})$$

$$+ O(N_f * N_{tp} * (1 + N_{nr}) * (1 + N_c))$$

$$+ O((1+N_{out})(N_{out}*N_r))$$

$$+ O(N_f * (N_{tp} * (1 + N_c) + N_{np}))$$

$$= O(N_f * [2 * N_{np} + N_{tp} * (1+N_c)*(1+N_{nr})]$$

$$+ N_r * N_{out} * (1 + N_{out}))$$

9.4(5)

Since each node and edge is realized only once per mapping, summing N_{nr} and N_{out} over all timeslots N_{ts} yields TN and TE , the total number of nodes and edges respectively. Summing $N_{out} * (1 + N_{out})$ over all timeslots is certainly less than $TE^{**}2$, but a better upper limit is $N_{OUT} * TE$ where N_{OUT} is the maximum number of outputs stored in any one timeslot (plus one). Summing Equation 9.4-5 over all timeslots and substituting TN and TE yields an expression for the complexity of each mapping iteration (N_i):

$$N_i = O(NF * [N_{ts} * 2 * (N_{np} + N_{tp} * (1 + N_c) + TN) + N_r * N_{OUT} * TE]) \quad 9.4(6)$$

where NF is the maximum number of feasible nodes per timeslot.

As mentioned earlier, the mapping may be restarted each time an operator cell is added, thus the total number of operations for the mapping process (N_m) is:

$$N_m = O(N_c * NF * (TN + N_{ts} * 2 * (N_{np} + N_{tp} * (1 + N_c))) + N_c * TE * N_r * N_{OUT}) \quad 9.4(7)$$

The maximum number of productions applicable at any node is determined by the cell and production libraries. It is independent of the CDFG and the CG size. Certainly, the maximum number of applicable productions is not greater than the total number of productions. Thus, the dependence of

the algorithms on the size of the libraries is at most linear. The remaining dependence is on the size of the CDFG, the size of the CG, the parallelism inherent in the CDFG and the parallelism implemented in the final mapping. The parallelism enters into the complexity analysis through the dependence on the number of nodes feasible for execution in each timeslot (NF), and the number of outputs which have to be stored in each timeslot (NOUT). The parallelism is clearly related to the CDFG and the CG size only if the algorithm is completely parallel (all unrealized nodes feasible in each timeslot) or completely serial (only one node feasible per timeslot). The expected behavior for the algorithm with general parallelism should be between these two extremes.

In a completely serial algorithm, there is only one node feasible (and thus realized) in each timeslot allocated. Thus, $NF = 1$, $N_{ts} = TN$, and N_{out} is equal to the maximum number of outputs of any one node (MO), yielding:

$$\begin{aligned} N_m &= O(N_c * (TN + 1 * TN * N_c) + T_E * N_r * MO) \\ &= O(TN * N_c^{**2} + T_E * N_r * N_c) \end{aligned} \quad 9.4(8)$$

i.e., the algorithm is linear with the size of the CDFG and quadratic with the size of the CG (number of cells and registers).

If the CDFG is completely parallel, it may be realized completely in parallel in one timeslot, completely in serial

with one node realized per timeslot, or somewhere in between. If the maximum parallelism of the CDFG is realized, then $NF = TN$, $Nts = 1$, $NOUT = TE$ and:

$$N = O(Nc^{**2} * TN + Nc * TN^{**2} + Nc * TE^{**2} * Nr) \quad 9.4(9)$$

However, for the CDFG to be realized in one timeslot, each operation must be realized directly on a cell and each edge must be stored in a register. Thus $Nc = TN$, $Nr = TE$, and

$$N = O(TN^{**3} + TN * TE^{**3}) \quad 9.4(10)$$

If a completely parallel CDFG is realized completely serially, $NF = TN$, $Nts = TN$, and $NOUT \leq MO = \text{constant}$ yielding:

$$N = O(Nc^{**2} * TN^{**2} + Nc * TE * Nr) \quad 9.4(11)$$

Again, each output must be stored in a register thus $Nr = TE$. In the worst case, a new cell may still be added to realize each CDFG node, resulting in $Nc = TN$ and:

$$N = O(TN^{**4} + TN * TE^{**2}) \quad 9.4(12)$$

The time complexity of the algorithm was tested by adding copies of the same description (temperature controller) in parallel and in series. Fig. 9.4-2 shows the increase in execution time as descriptions are stacked in

series. The solid line displays a linear growth in execution time with the size of the CDFG when the size of the CG generated and the parallelism are constant. The dashed line displays the execution time increase when the size of the circuit is constant, and the parallelism grows linearly with the size of the CDFG. The plot for parallel increase in description size does not exhibit the predicted quadratic dependence because as the timing constraints were relaxed to allow Elf to generate a serial realization, one less cell was required than was used for the single description. Fig. 9.4-3 displays the increase in execution time as the CDFG is constant and the timing constraints are adjusted to generate more parallel realizations. The expected behavior is quadratic, although this could not be verified since at most 3 circuits were instantiated in parallel.

The most direct change in the algorithm which would improve the execution dependence is to not restart the mapping each time a cell is added. An alternate scheme which involved restarting a fixed number of times would reduce the order of the dependence on N_c (T_N in the parallel algorithm) by one.

Fig. 9.4-2: Increase in execution time of algorithm with increasing circuit size.

Fig. 9.4-3: Increase in execution time of algorithm with increasing circuit size.

9.5 Testing

It is often mentioned that testing is rapidly becoming a serious problem in IC fabrication and that testability must be considered during the design of the circuit. Certainly, it appears undesirable to manually generate a test pattern for an automatically generated circuit.

The limited architecture that Elf can generate (register-mux-operator cell) makes automatic test pattern generation trivial if all registers are of the LSSD type [Wilt83]. Since all inputs to operators are from registers and all outputs are stored in registers, all operator cells become completely controllable and completely observable. Thus, if each cell in the library has a user specified test pattern, the entire circuit can be tested by simply combining the individual cell tests.

10/0 CONCLUSIONS

10.1 Summary

This thesis has presented a method of automatically generating circuits to realize algorithmic level descriptions. These methods are aimed at realizing fixed hardware algorithms rather than generating general purpose programmable hardware (i.e., computers).

An analysis of the language requirements for algorithmic hardware description was performed. This analysis demonstrated the similarity of describing hardware at the algorithmic level to software programming. This lead to selecting a software programming language, Ada, for describing hardware in this thesis. Choosing an existing software language allows the designer to simulate the functional description by compiling the description as software on a general purpose computer. Using a software language also allows the designer to benefit from the established software environment (eg. debugger) and allows a future avenue to combining the hardware and software design of large systems.

From the myriad of software languages, Ada was chosen for several reasons. Ada tasks allow the designer to specify high level parallelism between concurrent processes explicitly. Ada data types provide a mechanism for hardware signal typing. This serves to define the type of memory cell required to store the result of an operation. The

hierarchy produced by using Ada packages and tasks introduces modularity into the design. Finally, Ada pragmas allow the designer to specify new hardware cells, and to guide the allocator from the Ada description. Methods for realizing Ada language constructs in hardware were presented and a mechanism for specifying timing constraints in Ada was developed.

A hardware compiler, Elf, capable of realizing algorithmic descriptions written in Ada was designed. Several sections of this system were identified as comparable to work reported in the literature. However, the data path allocators previously reported were not well suited to realizing Ada circuit descriptions. A new data path allocator based on graph grammars and urgency scheduling techniques was developed and implemented. The prototype allocator starts from a control/data-flow graph representation of the algorithm. This intermediate level functional format was chosen as it extracts the low level parallelism inherent in the algorithm. The allocator generates a standard cell circuit to realize the algorithm, and a mapping of the algorithm onto the circuit.

The data path allocator is rule based. Since the input and output formats of the allocator are graphs, the rules were formulated as graph grammar productions. Nonterminal productions are used to specify functionally equivalent (sets of) operations. Terminal productions are used to map

operations in the functional description onto cells capable of realizing the operation.

When more than one rule can be used to realize an operation, an implementation is selected based on an estimate of the cost added to the circuit to realize the operation using each applicable rule. The cost of a rule consists of the area (power consumptions, etc.) of the cell specified in the rule, the control overhead of the cell, and an estimate of the additional interconnect area required to connect the cell to other cells in the circuit as required to realize this operation. If a new cell is selected to realize an operation, a second cost function based on global parameters of the algorithm and circuit allocated thus far is used to decide which cell to add. This allows cells to be selected which cost more, but which realize several operations and thus prevent the addition of more circuitry in the future.

The timing constraints specified in the Ada description are realized by urgency scheduling techniques. The acceptable cost of realizing a node is increased as the slack time remaining in which to realize the operation decreases.

A prototype implementation of the allocator was written in LISP. This prototype allows the designer to specify a start circuit and includes an interactive mode in which the designer can guide the allocation process. The results of several examples showed that the automatically generated circuits were comparable (within 50%) of the size of manual

ly designed circuits to realize the same description, using the same cell library.

10.2 Suggestions for Further Work

The graph grammar/ scheduling algorithm works reasonably well for mapping the CDFG to the CG. Some basic extensions are required to allow the system to generate more general architectures. Specifically, the allocator should recognize buses and combined operator/ storage cells (eg. counter, accumulators). The present allocator allows multi-cycle operations, but extensions are required to allow circuits to be designed which include two operator cells connected in series without an intermediate buffer register.

The main area for improvement in the algorithm is the process for adding cells to the circuit. First, a one order improvement in the execution time complexity of the algorithm could be achieved if the allocator was modified so that the mapping process was not restarted each time a cell is added. Second, the scheduling algorithm suffers from myopia. More analysis of the global parameters should be performed to arrive at a better method for selecting cells to add to the circuit. Finally, a method should be developed which would allow the allocator to remove cells and interconnect which are not well utilized.

At the description level, an area for research would be to investigate the possibility of combining the hardware and software design of a large system in a common language.

Bibliography

- [Ada82] Ada Reference Manual,
U.S. Department of Defence, July, 1982.
- [AhoA74] A. Aho, J. Hopcroft, and J. Ullman,
The Design and Analysis of Computer Algorithms,
Addison Wesley, Don Mills, Ontario, 1974.
- [BarM78] M.R. Barbacci,
Instruction set processor specifications for simulation, evaluation, and synthesis, IEEE Trans. on Computers, Vol C-30, No. 1, Jan. 81.
- [Clav79] Claus, V, Ehrig, H, and Rozenberg, G, Eds., Graph-Grammars and Their Application to Computer Science and Biology,
- [DirS81] S.E. Director, A.C. Parker, D.P. Siewiorek, and D.E. Thomas, A design methodology and computer aids for digital VLSI systems, IEEE Trans on Circuits and Systems, Vol CAS-28, July 81.
- [HafL78] L.J. Hafer and A.C. Parker, Register transfer level automatic digital design: the allocation process, Proc. 15th Design Automation Conf., 1978.
- [HafL82] L.J. Hafer and A.C. Parker, Automated synthesis of digital hardware, IEEE Trans. on Computers, Vol c-31, No. 2, Feb. 82.
- [HitC83] C.Y Hitchcock III, Automated synthesis of data paths, Master's Thesis, Dept of Electrical Engineering, Carnegie-Mellon Univ., Pittsburgh, PA, Jan. 83. Lecture Notes in Computer Science No. 73, Springer Verlag, Berlin, 1979.
- [KowT84] T.J. Kowalski and D.E. Thomas The VLSI design automation assistant: an IBM System/370 design, IEEE Desing and Test of Computers, Vol. 1, No. 1, Feb. 84.
- [LieG80] G.W. Lieve The design, implementation, and analysis of an automated logis synthesis and module selection system, Ph.D. dissertation, Dept. of Electrical Engineering, Carnegie-Mellon Univ., Pittsburgh, PA, 1980.
- [MarP79] P. Marwedel, The MIMOLA design system: detailed description of the software system, 16th Design Automation Conf., 1979.

- [McMF78] M.C. McFarland, The value trace: a data base for automated digital design, Master's Thesis, Dept. of Electrical Engineering, Carnegie-Mellon Univ., Pittsburgh, PA, 1978.
- [NagA82] A.W. Nagle, R. Cloutier, and A.C. Parker, Synthesis of hardware for the control of digital systems, IEEE Trans. on CAD, Vol. CAD-1, No. 4, Oct. 1982.
- [NagM81] M. Nagl, Application of graph rewriting to optimization and parallelization of programs, Comput. Suppl. 3, 1981.
- [OrgE82] E.I. Organick and G. Lindstrom, Mapping high-order language program units into VLSI structures, Compcon '82, 1982.
- [OrgE82b] E.I. Organick et al., Transformation of Ada programs into silicon, Contract report, ARPA order No. 4305.
- [OrgE84] E.I. Organick et al, Transforming an Ada program unit to silicon and verifying its behavior in an Ada environment: a first experiment, IEEE Software, Vol. 1, No. 1, Jan. 1984.
- [SnoE78] E.A. Snow, Automation of module set independent register-transfer level designs, Ph.D. dissertation, Dept. of Electrical Engineering, Carnegie-Mellon Univ., Pittsburgh, PA, 1978.
- [SouJ83] J.R. Southard, MacPitts: an approach to silicon compilation, IEEE Computer, Vol. 16, No. 12, Dec. 1983.
- [SusJ82] J.M. Suskind, J.R. Southard, and K.W. Crouch, Generating custom high performance VLSI designs from succinct algorithmic descriptions, MIT 1982 Conf. on Advanced Research in VLSI, Jan. 1982.
- [TheA76] A. Thesen Heuristic scheduling of activities under resource and precedence restrictions, Management Science, Vol. 23, No. 4, Dec. 1976.
- [ThoD83] D.E. Thomas et al., Automatic data path synthesis, IEEE Computer, Vol. 16, No. 12, Dec. 1983.
- [Wilt83] T.W. Williams and K.P. Porter, Design for testability - a survey, Proc. IEEE, Vol. 71, Jan. 1983.
- [ZimG79] G. Zimmermann, The MIMOLA design system: a computer aided digital process of design method, 16th Design Automation Conf., 1979.

[ZimG80] G. Zimmermann, Computer aided design of control structures for digital computers, International Conf. on Computers and Design, 1980.

APPENDIX 1

DESCRIPTION OF ELF DATA STRUCTURES

There are four basic data files associated with the Elf system. The control-data flow graph (CDFG) and circuit graph (CG) are representations of the input to and output of the allocator respectively. The CDFG is a data flow graph with explicit control constructs which describes the algorithm desired; the CG is a graphical representation of the netlist of the final circuit. The production and cell libraries contain information related to actually performing the allocation. The general (GP) and designer (DP) production libraries describe technology independent rules for replacing (groups of) operations in the CDFG with equivalent, potentially less expensive, (groups of) operations. The cell libraries (technology (TL) and designer (DL)) describe the cells available to realize a particular design.

The data formats for the Elf system are all based on standard LISP lists. Other implementation languages were discarded because they lacked a (dynamically) variable length data record which greatly simplifies the representation of the graphs and the graph rewriting operations. The use of LISP list format in the file format greatly simplifies file reading and writing operations.

The file format for all four data structures is a string of identifier-value pairs. The identifier is the name of the item and the value is a LISP data structure which describes the characteristics of the item. In the CDFG and CG, each item is a node in the graph and the value is the node description. This node description includes a node type and input list as described by the enclosed BNF. The list of outputs and realization information is generated by Elf and is not present in the input file. For productions, the identifier again names the item, but the value is now a pair of subgraphs which represent the left hand side (LHS) and right hand side (RHS) of the production. During allocation, if the LHS matches a subgraph in the CDFG, these nodes may be replaced by the RHS of the production. Choosing subgraphs to represent these productions greatly simplifies the application of the production. Finally, the identifier in a cell library file gives the cell name and the value describes the function(s) realized by the cell. A preprocessor in the allocator automatically translates this format into the production format used by the allocator. A complete BNF description of the four data files follows.

BNF FOR ELF DATA FILES

CONTROL/DATA-FLOW GRAPH

cdfg_file -> node_list (list_circuit_I/O_nodes)

node_list -> cdfg_node_id cdfg_node_description <node_list>

cdfg_node_description -> ((node_specs)(input_list)
<(output_list) <(realization)>)

node_specs -> gen_type specific_type characteristics

gentype	valid specific_type	valid characteristics
constant	value	(o/p_type, width)
operator	+, -, *, /, etc, user_def_fn memory_read/write RV_call, RV_accept word_join/split	(o/p_type, width), list
call	included expand named	cdfg_subgraph cdfg_node_name
control	fork join sync	corresponding join corresponding fork
passing	reg_transfer	
loop	begin_loop end_loop exit_loop	loop_par, loop_end_node <loop_exit_nodes> loop_begin_node
timing	reference constraint	ref_node, min, max
data_io	put get	(o/p_type, width)

NONTERMINAL PRODUCTIONS

```

production_library -> production_list
production_list -> production <production_list>
production -> production_id (left_hand_side right_hand_side)
left_hand_side -> mock_node_description
right_hand_side -> mock_node_description|cell_realization
mock_node_description -> ((mock_node_specs)(mock_input_list)
                        <(mock_output_list)>)
mock_input_list -> mock_input <mock_input_list>
mock_input -> (data_type bit_width)
                |((input_connection) [for internal call node])
mock_output_list -> mock_output <mock_output_list>
mock_output -> output_connection_list [for int. call nodes] )
                | call_output_list [for call desc node]
cell_realization -> (("CELL" cellname) delay cost-tuple))

```

!! mock_node may be a call

:.) calls must be "included"

CELL LIBRARY

```

cell_library      -> cell_list
cell_list         -> cell_definition <cell_list>
cell_definition   -> cell_name cell_description
cell_description  -> (operator_list cost_tuple)
operator_list     -> operator_details <operator_list>
operator_details  -> (operator_id time_delay (input_list)
                    <(output_list) <(control_list)>>)
input_list        -> cell_connection <input_list>
output_list       -> cell_connection <output_list>
cell_connection   -> (functional_input physical_input)
functional_input  -> (data_type bit_width <(equivalent_inputs)>)
physical_input    -> (list_of_pin_numbers_of_this_input)
cost_tuple        -> cell_area [extended at will]
control_list      -> control_input <control_list>
control_input     -> ((data_type bit_width) (pins) (value))

```

CIRCUIT GRAPH

```
cct_graph -> cell_list (list_circuit_i/o_cells)
cell_list -> cct_cell_id cct_cell_description <cell_list>
cct_cell_description -> ((cell_specs)(input_list) <(output_list)
cell_specs -> gen_type specific_type characteristics
```

gentype	valid specific_type	valid characteristics
cell	cell name of cell	nil
reg	reg_type	width
net	mux bus	width
port	input inout output	width

```
input_list -> input_connection <input_list>
input_connection -> (cell_id connection_id)
connection_id -> [integer cardinality of output connection]
output_list -> (output_connection_list) <(output_list)
output_connection_list -> (output_connection) <(output_connection_list)
output_connection -> cell_id connection_id
```

```
cct_cell_property_list -> NIL
```

3 OF/DE 3

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

END / FIN

30 03 87