1

MICROD

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

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.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

THÈSES CANADIENNES

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.

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

**AUTOMATED MICROCONTROLLER SYNTHESIS
FOR APPLICATION-SPECIFIC INTEGRATED CIRCUITS**

by

PETER KALAB, B. ENG.

This report is a thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of

Master of Engineering

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University

December 1986

© copyright

1986, Peter Kalab

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 0-315-35841-6

The undersigned hereby recommends to the Faculty of Graduate Studies and Research acceptance of the thesis,

"Automated Microcontroller Synthesis for Application-Specific Integrated Circuits"

submitted by PETER KALAB, B. ENG., in partial fulfilment of the requirements for the degree of Master of Engineering.

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University

January 1987

ABSTRACT

Hardware compilation is the process of automatically synthesizing a low-level structural description of a circuit from a high-level description of the circuit's behaviour. This thesis examines the issues involved in the automated design of microcoded controllers for circuits generated by hardware compilers. Issues examined include the separation of control and data paths, hardware implementation of subroutines, controller partitioning, and distributed control.

A prototype program for automated microcode generation is described. This program generates horizontal microcode given a description of the algorithm to be performed, the circuit to perform the algorithm, and a library of hardware cells used by the circuit. The program also performs horizontal microcode compaction to reduce the control memory area.

Acknowledgements

I would like to thank Dr. J. P. Knight, my supervisor, for his guidance during my studies, and for his contribution to my work. I would also like to thank Dr. M. A. Copeland for making the Department of Electronics' VAX available to a user from the 'other' Department. I also wish to thank Messrs. J. Byers, G. P. Bell, M. P. Gagnon, and N. Mikhail for their services, Dr. J. J. Salinas for assistance with the thesis presentation, and J. Midwinter for several philosophic discussions.

Finally, I wish to thank my parents for their continual encouragement and support.

Table of Contents

1. INTRODUCTION	1-1
1.1. Application-Specific Integrated Circuits	1-1
1.1.1. Gate Arrays	1-1
1.1.2. Standard Cells	1-2
1.1.3. Full-Custom Design	1-2
1.2. The Design Process	1-2
1.2.1. The Y-Diagram	1-4
1.2.2. Manual Design	1-6
1.3. The Promise of Hardware Compilation	1-9
1.4. Hardware Compilation - An Overview	1-10
1.5. Overview of this Thesis	1-12
2. RELATED WORK IN HARDWARE COMPILATION	2-1
2.1. MacPitts	2-2
2.2. CMU-DA	2-2
2.3. DAA	2-5
2.4. Facet	2-7
2.5. MIMOLA	2-8
2.6. Elf	2-9
2.7. HAL	2-11
2.8. Synapse	2-14
2.9. Common Features	2-16
3. HARDWARE DESCRIPTION LANGUAGES	3-1
3.1. Textual Descriptions of Hardware Structure	3-1
3.2. Languages for Hardware Compilation	3-3
3.3. VHDL	3-5
3.4. Other Hardware Description Languages	3-6
3.5. Software Programming Languages for Hardware Description	3-6
3.5.1. Extensible and Object-Oriented Languages	3-7
3.6. Effect of HDL on Control Synthesis	3-9
4. DATA PATH SYNTHESIS	4-1
4.1. Overview of Data Path Synthesis	4-1
4.2. Graph Productions	4-3
4.3. Control Step Partitioning	4-7
4.3.1. Urgency Scheduling in Elf	4-9
4.3.2. Multi-Cycle Operations	4-10
4.4. Load Balancing	4-11
4.5. Hardware Cell Selection	4-14
4.6. Effect of Data Path Synthesis on Controller Design	4-16
4.6.1. Effect of Control Step Partitioning	4-16
4.6.2. Effect of Hardware Cell Selection	4-17
5. CONTROL PATH SYNTHESIS	5-1
5.1. Self-Timed Circuits	5-2
5.2. Independent/Concurrent Synthesis	5-2
5.3. Architectural Considerations	5-4
5.3.1. Hardwired Controllers	5-4
5.3.2. Microcoded Controllers	5-5

5.4. Functional Division of Control and Data Paths	5-7
5.4.1. Interaction During the Synthesis Process	5-7
5.4.2. Interaction in Sophisticated Controllers	5-10
5.4.3. Interaction in Simple Controllers	5-11
5.4.4. Controller Complexity Tradeoffs	5-11
5.5. State Assignment and Minimization	5-12
5.6. Control Signal Encoding	5-13
5.7. Parallelism Issues	5-15
5.8. Meta-Control	5-15
5.9. Interconnect Cost and Distributed Control	5-16
6. MICROCONTROLLER DESIGN ISSUES	6-1
6.1. Microcomputer Generators	6-3
6.1.1. PLEX	6-3
6.1.2. Microcontroller Synthesizer	6-4
6.1.3. Microcomputer Generators for ASIC Design	6-5
6.2. Microcoded Control for ASIC Data Paths	6-7
6.2.1. Strengths of Microcoded Controllers	6-7
6.2.2. PLA-based Controllers	6-8
6.3. High-Level Constructs in Microcode	6-10
6.4. Microcode Implementations of Loops	6-12
6.4.1. Loop Unrolling	6-13
6.4.2. Loop Control Using Data Path Resources	6-14
6.4.3. Dedicated Hardware for Loop Control	6-17
6.4.4. Subscripts for Array Access	6-20
6.5. Subroutines	6-21
6.5.1. In-Line Expansion of Subroutines	6-22
6.5.2. Subroutine Linkage in Hardware	6-23
6.5.3. Promotion of Subroutines to Independent Tasks	6-24
6.6. Error Handling	6-25
6.7. Control Word Format	6-26
6.7.1. Horizontal Microcode	6-27
6.7.2. Vertical Microcode	6-28
6.7.3. Bit Steering: Two-Level Microcode Encoding	6-29
6.8. Microcode Compaction	6-30
6.9. Multi-Level Control	6-32
6.9.1. Indirect Addressing	6-32
6.9.2. Multi-Way Branching	6-34
6.10. Storage of Literals	6-35
6.11. Distributed Control	6-36
7. <u>MiSyn</u> - THE MICROCODE SYNTHESIZER	7-1
7.1. Major Assumptions of the <u>MiSyn</u> Program	7-1
7.1.1. Controller Design Style	7-1
7.1.2. Unimplemented Microcontroller Features	7-4
7.2. <u>MiSyn</u> Overview	7-4
7.2.1. Program Inputs	7-6
7.2.2. Program Outputs	7-7
7.3. Format of the Control/Data Flow Graph	7-7
7.3.1. OPERATION Nodes	7-9
7.3.2. CONSTANT Nodes	7-10
7.3.3. CONTROL Nodes	7-10
7.4. Phase 1 - Reduction of CDFG to Control Graph	7-13
7.4.1. The Traversal of the Control/Data Flow Graph	7-14

7.4.2. The Structure of the Control Graph	7-16
7.4.3. Construction of the Control Graph	7-18
7.5. Phase 2 - Generation of Symbolic Microcode	7-18
7.5.1. Transformation of FORK/JOIN Nodes	7-20
7.5.2. Transformation of Loops	7-20
7.6. Phase 3 - Creation of Control Memory Array	7-22
7.6.1. Processing of Unspecified Control Fields	7-25
7.7. Phase 4 - Microcode Compaction	7-28
7.7.1. The Compatibility Matrix	7-28
7.7.2. The Compaction Algorithm	7-29
7.7.3. Control Field Values After Compaction	7-31
7.8. Results	7-33
7.9. <u>MiSyn</u> Summary	7-33
8. FUTURE WORK AND CONCLUSIONS	8-1
8.1. Enhanced Controller Architectures	8-1
8.1.1. Extensions for Subroutines	8-2
8.1.2. Distributed Control	8-4
8.1.3. Multi-Way Branching	8-6
8.2. Alternate Control Word Formats	8-7
8.3. Microcode Partitioning	8-7
8.4. Integrated Simulation	8-9
8.5. Summary	8-10
8.6. Conclusions	8-12
REFERENCES	A-1
GLOSSARY	B-1
Sample Input and Output	C-1

CHAPTER 1

1. INTRODUCTION

Integrated circuit technology permeates every facet of life in industrialized countries. The chip's influence is easily felt in applications ranging from top-secret military applications to running shoes with built-in computers.

Integrated circuits may be classified as general-purpose or application-specific. General-purpose integrated circuits (GPICs) are devices such as microprocessors, memories, and other components which may be used by digital designers in a wide variety of products. The devices themselves are products of the IC manufacturers. Since their market is so large, they are manufactured in large quantities; consequently, their unit prices tend to be low.

1.1. Application-Specific Integrated Circuits

Application-specific integrated circuits (ASICs) are devices designed for one specific function. Since an ASIC may be used only in one product, that product must be manufactured in sufficiently large volume, or the ASIC must enhance its performance enough, to justify the high cost of its development. ASICs may be developed in several ways, depending on how much time, effort, and money is available.

1.1.1. Gate Arrays

The shortest development times, and lowest development costs, may be achieved using semi-custom gate-array technology. In this design methodology, the designer is given a chip with pre-defined gates, but without the final interconnections between the gates. The final chip may be developed quickly because only the interconnect layers

must be specially designed. The main disadvantage of this method is that the utilization of silicon area is low because the number and types of gates are fixed by the gate-array manufacturer.

1.1.2. Standard Cells

Standard cell design provides more flexibility by allowing the designer to choose the cells on the chip from a library of pre-designed and characterized cells. Only the cells which the designer wants will be included on the final chip, but the design time and cost is higher than for gate-array design since all mask layers must be custom-made.

1.1.3. Full-Custom Design

In cases where the highest performance goals must be met, the designer may opt for full-custom design. This means that each cell may have to be individually designed or modified for optimal performance for the application. This method provides the maximum speed or minimal area solutions, but it incurs the highest design costs.

1.2. The Design Process

The design of an integrated circuit is an iterative process. The V-diagram [GajD83], shown in Figure 1-1, represents the design process as an inward spiral, starting from high levels of abstraction and moving toward low-level descriptions. During this process, the design is converted between functional, structural, and geometric descriptions, as represented by the axes of the diagram.

Figure 1 - 1.

Y - DIAGRAM FOR VLSI DESIGN

The Y - Diagram shows the relationship between functional, structural, and geometric representations of a circuit. Moving further from the origin increases the level of abstraction. Arrows can be drawn to represent various design methodologies as mappings between levels of abstraction on the three axes. Design processes appear as an inward spiral, iteratively transforming a high - level description to lower - level ones, until the final transistor layout is produced.

1.2.1. The Y-Diagram

The functional axis represents different levels of design abstraction concerned with the circuit's behaviour and not its implementation. The behaviour can be represented as a design specification at the highest level, or by logic equations at the lower levels of abstraction. A register-transfer description, or an algorithm, are other examples of functional descriptions. In each case, the circuit's physical structure is not important; any functional description can be realized by many different circuits.

The geometric axis represents the implementation details of a circuit in silicon, with no concern for its function. This representation shows the actual mask layers at the lowest level of abstraction. Higher levels of abstraction group mask geometries into cells, and cells into chip floorplans.

The structural axis provides the means for converting a functional description to a geometric one. Structural descriptions represent the hardware at an abstract level. A functional description can be mapped onto a structural description; this in turn can be mapped onto a geometric description.

A functional description at the algorithmic level, for example, may include a logical NOT operation. This can be shown in the structural description by an inverter symbol in the circuit's logic diagram. In the geometric description, this inverter would be represented by a set of mask layers. Figure 1-2 shows functional, structural, and geometric representations of an inverter at various levels of abstraction. The functional description is written as a high-level Pascal-like program. A technology-independent structural representation is provided

Figure 1 - 2. FUNCTIONAL, STRUCTURAL, AND GEOMETRIC REPRESENTATIONS OF AN INVERTER

These diagrams show how an inverter can be described by an algorithm (Functional), a transistor schematic and logic diagram (Structural), and a cell layout (Geometric). The technology - dependent representations are for a CMOS inverter.

1) Algorithm

Output := NOT (Input)

2) Logic Diagram

3) Transistor Schematic

4) Cell Layout

This shows a virtual - grid layout of a CMOS inverter, which does not include all mask layers.

KEY

by a logic diagram; a transistor schematic shows that the inverter is implemented in a CMOS technology. A virtual-grid cell layout, which does not show all mask layers, provides a geometric description.

1.2.3. Manual Design

For conventional design, a functional specification, giving the performance, power, and other constraints, is used as the starting point for the design. This is converted to a structural design, such as a register-transfer description or a logic diagram.

The circuit can be checked by performing a pre-layout simulation; this checks if the circuit logic actually meets the functional specification. Timing and power dissipation estimates can also be made by estimating device characteristics. If the pre-layout simulation produces acceptable results, an actual layout is performed, transforming the structural description to a geometric one.

Once all components have been laid out and routed, the low-level geometric description can be converted by circuit extractor programs to a schematic diagram; to ensure that no errors were introduced during the layout phase, the extracted schematic can be compared with the original schematic.

Actual device parameters can also be extracted from the layout for simulation; this will produce more accurate results than the pre-layout simulation, particularly for signal delays due to routing. If the circuit performs in accordance with the functional specification, the layout can be used to create masks for each layer in the fabrication process. The chip can then be produced, tested, and packaged. If the

design fails to meet the design specification at any step, the process must be repeated. The design process is shown in Figure 1-3.

Although computer programs are capable of performing the operations described above, the human designer is still responsible for deciding what hardware components will be used and how they will be connected. The computer serves as a medium for expressing the designer's idea. Automatic placement and routing programs relieve the designer from much of the low-level design tasks. Use of standard cell libraries, which contain layouts with known performance and characteristics, simplifies the design process at the expense of increased constraints on the layout.

More sophisticated design aids, such as module builders and silicon compilers, go further in reducing the designer's responsibility for the low-level details of the design. Module builders are used to synthesize a cell, such as an AND, given the functions it must perform and other parameters, such as the number of bits in the inputs and outputs. The module builder then constructs the cell using bit-slices of more primitive modules to meet the specified parameters. A silicon compiler, as envisioned by [JohD79], takes the synthesis process one step higher. Such a silicon compiler takes the structural description of a circuit and converts it to a layout. The designer is still responsible for creating the structural description of the circuit.

Current computer-aided engineering simplifies the manual design process by providing interactive graphics editors and simulation tools. Module builders and silicon compilers can take over some of the low-level design tasks to further simplify the designer's task, but as fabrication technology allows millions of transistors on a single chip,

Figure 1 - 3. THE VLSI DESIGN AND FABRICATION PROCESS

This diagram shows the steps required to design and fabricate a VLSI chip. The steps within the shaded box represent the actual design process; these steps, and design evaluation, use a hierarchical description of the chip. Hardware compilation would replace the steps in the shaded box by a computer program which would generate a chip design from a high-level behavioural description. Its effect on the VLSI design and fabrication process would be greater than might appear, because the design process itself is a design / simulate / evaluate cycle. Hardware compilation ensures a design that is 'correct by construction'. The dashed lines show how failure during evaluation or testing leads to design and fabrication iterations.

even this may not be enough. This is where fully-automated hardware compilation enters the IC design process.

1.3. The Promise of Hardware Compilation

As technology progresses from Very-Large Scale Integration (VLSI) to Ultra-Large Scale Integration (ULSI), individual ICs may contain on the order of a million transistors. Clearly, a structured, top-down design methodology is required to deal with so many components. In fact, this problem is exactly analogous to what faced programmers as the complexity of their programs increased to the point where machine or assembly language programming became impractical. By using higher-level languages such as FORTRAN or Algol, programmers were no longer burdened with the responsibility of keeping track of CPU registers and other hardware-dependent features. By dealing with the problem at a higher level of abstraction, programmers could devote more time to their algorithms.

In the same way that software compilers increase programmer efficiency and speed up software development, hardware compilers promise increased efficiency and reduced hardware development times. In this thesis, hardware compilation refers to the synthesis of a low-level circuit description from a high-level behavioural description of the algorithm performed by the circuit.

Hardware compilation differs from 'silicon compilation' as defined in the previous section by placing most of the burden of design on the compiler. The different paths for hardware and silicon compilation on the Y-diagram illustrate this difference. The designer may direct the compilation by specifying timing or area constraints on the

circuit, but the hardware cell selection and interconnection is all determined by the compiler rather than the designer.

1.4. Hardware Compilation - An Overview

A hardware compiler is a program which converts, with minimal intervention by a human designer, a high-level description of a circuit into a lower-level description, which may then be used to fabricate the actual chip. The main things to note are:

- 1) by starting from a functional description, the designer does not need a detailed knowledge of hardware design. As long as the designer knows what the circuit is required to do, the compiler will worry about the implementation details. This is the long-term goal of hardware compilation.
- 2) the compiler itself can stay in the technology-independent part of the Y-diagram. The conversion of standard modules and net-list to masks can be performed by technology-dependent post-processors. In this way, the same top-level design can be implemented in any available technology.

The processing steps in a 'typical' hardware compiler are shown in Figure 1-4.

The Hardware Description Language (HDL) may be either a specially-developed language, or it may be a conventional software programming language such as Ada or Pascal.

The data-path synthesizer is responsible for analyzing any constraints imposed by the designer, selecting the hardware cells, and interconnecting them to meet the constraints. This stage of the compiler may also include an optimizer to reduce the number of registers,

Figure 1 - 4.

SEQUENCE OF OPERATIONS IN HARDWARE COMPILATION

This diagram shows the operations performed by a 'typical' hardware compiler. Inputs are shown as shadowed boxes; other boxes represent the operations.

or to reduce the number of functional cells by replacing several simple cells by a more sophisticated one (e.g. an ALU).

The job of the control-path synthesizer is to create a controller for the complete circuit. This involves examining the original functional description, as well as the synthesized data-path, to determine the states and transitions for either a finite state machine (FSM) or microcoded controller.

1.5. Overview of this Thesis

This thesis is concerned with the control synthesis part of hardware compilation. Background material in provides an overview of other aspects of hardware compilation.

The main contribution of this thesis is an original program for microcode synthesis. This program, which generates the microcode for a user-specified algorithm and data path, demonstrates one approach to the control synthesis part of hardware compilation. It uses a Control/Data Flow Graph to describe the algorithm being performed, a circuit net list to describe the circuit, and a cell library to describe the standard cells used in the circuit.

Chapter 2 gives a brief summary of some hardware compilation systems and their features.

Chapter 3 considers some of the issues in Hardware Description Languages (HDLs) and how they affect hardware compilation.

Chapter 4 provides a summary of the key concepts in data path synthesis.

Chapter 5 examines control path synthesis from a general point of view. It also discusses the relationship between data path and control synthesis.

Chapter 6 covers control synthesis as it applies to microcoded architectures.

Chapter 7 describes the microcode synthesis program. The appendix contains a sample Control/Data Flow Graph, circuit description, and the synthesized microcode.

Chapter 8 discusses possible enhancements to this program and summarizes the results of this thesis.

CHAPTER 2

2. RELATED WORK IN HARDWARE COMPILATION

Researchers have long known that conventional manual design techniques will be inadequate as chip complexity increases. Hardware compilation has been a research topic for many years. Recently, several commercial silicon compilers have appeared on the market. These products do not currently support hardware compilation as defined in this thesis, because they do not synthesize the circuit from a high-level behavioural description. Examples of such systems are Genesil from Silicon Compilers Inc., and Concorde from Seattle Silicon Technology [SchM86]. Genesil accepts structural representations of modules such as CPUs, RAMs, ROMs, and I/O ports, which are translated to the gate level. The user can examine and change the circuit at any level. A simulator is integrated into the Genesil package. Compilation of a 1000-transistor circuit requires less than one minute on a MicroVAX II. By comparison, Concorde's input consists of 'data sheets' filled in by the user. These are used to create parameterized cells which may be combined with standard cells.

The Macgen program from LSI Logic Corp. produces multiplier-accumulator layouts, and the SC-2 'silicon converter' used by AT&T is used to design digital signal processors. When the compiler's domain is reduced to highly-specialized types of circuits, compiled circuits have surpassed manual designs. Currently, however, these compilers have not been able to produce layouts as efficient as hand-crafted designs for general circuits.

The following sections briefly describe some hardware compilers which are intended to start from high-level behavioural descriptions.

Due to the large number of compilers, only a few have been selected for their historical significance or unique features.

2.1. MacPitts

MacPitts was developed at the Massachusetts Institute of Technology to design circuits with microprogrammed data paths [SouJ83]. The input description, written in a LISP-like language, was at the register-transfer level and included serial and parallel operations. The input was translated to a technology-independent intermediate form in which the data path was described as a net list of hardware cells, and the control path was described by state assignments and logic equations. The design was verified by a simulator which used this intermediate form. Control path synthesis was accomplished with Weinberger arrays. The control and data paths were then laid out and routed, and finally converted to CIF format for mask fabrication. The architecture of the chip was fixed by MacPitts. The entire process was automatic from the point where the user specified the MacPitts description of the circuit. As such, MacPitts was a hardware compiler as defined in the previous chapter. MetaSyn, a commercial version of MacPitts was available, but it did not sell because it was "too far removed from the real world" [SchM86].

2.2. CMU-DA

The Carnegie-Mellon University Design Automation project (CMU-DA) is a hardware compiler targeted mainly at computer circuits. This is shown by its input language (ISP), and by the examples used to demonstrate its capabilities: synthesis of the 6502, PDP-11/40,

and IBM 370 central processing units [NagA81a]. Its organization is shown in Figure 2-1.

The input description in ISP is converted to a control/data flow graph called the Value Trace (VT). This intermediate form is optimized before being passed to the Data Path Allocator. The Data Path Allocator actually contains two allocators: a deterministic algorithm, and an expert system.

Once the circuit design has been finalized by the module binder which selects the physical cells to perform the required operations, the Control Allocator designs a microcoded controller for the data path. The format of the microcode is also optimized to reduce the size of the control memory. This is done by two algorithms: the Autonomy Algorithm, and the Attraction Algorithm [NagA81b]. The microcode generator attempts to reduce the size of the control memory by encoding control fields.

The Autonomy Algorithm is used to identify which control fields should be made 'autonomous', meaning that the field will not be combined with any other fields in the control memory. Autonomous fields are never encoded. The remaining fields, however, may be encoded to reduce the overall width of the control memory.

The Attraction Algorithm is used to assign micro-operations to microcode words. This defines the actual control word format and the amount of parallelism in the control memory. This algorithm can be used to maximize the speed of the controller at the expense of increasing the control word width; in this case, the user specifies the maximum number of control words for the microprogram. Alternatively, the user may specify the maximum control word width; the control

Figure 2 - 1. ORGANIZATION OF THE CMU - DA SYSTEM

memory is encoded for maximal parallelism without violating this constraint. In the latter case, decreasing the control word width can be expected to increase the number of control words.

2.3. DAA

The VLSI Design Automation Assistant (DAA) is currently under development at AT&T Bell Laboratories [kowT85a]. Its organization is shown in Figure 2-2. This system is an extension of CMU-DA which adds register, operator, data path, and control signal detail to the Value Trace [kowT85b]. It has been used to synthesize the IBM 370 data paths. Implemented in the OPS5 expert system shell, it uses an expert systems approach to the VLSI design problem. The knowledge acquisition phase of the design included reviews of other synthesis systems and interviews with four digital designers ranging in experience from novice to expert levels.

Like the CMU-DA system, DAA accepts an algorithmic description of the chip written in ISPS and user-specified constraints. It uses a technology-sensitive database together with its built-in rules to direct its search of the design space. The system is currently being extended to handle pipelined designs.

Some parts of the DAA system, such as the module binder and control allocator, are algorithmic programs rather than expert systems. The module binder instantiates the modules requested by DAA in the selected technology. This is accomplished by selecting the module from a database or by using module builders to construct it from more primitive modules. The control allocator designs either a microprogrammed or PLA-based controller for the data path created

Figure 2 - 2.

VLSI DESIGN WITH DAA

This diagram shows the sequence of operations to create a VLSI design using the Design Automation Assistant (DAA). Cell generators are used when custom cells are required. Otherwise, DAA uses cells from a module database.

by DAA and the module binder. This program was originally developed at Carnegie-Mellon University, but it is being extended for pipelined architectures at AT&T Bell Laboratories.

The data path floor-planner is also an expert system which arranges the modules from the module binder and control allocator on the chip die. This process also considers user-specified speed and area constraints.

The DAA system incorporates a 'multi-paradigm' approach to hardware compilation. Rather than use expert systems techniques for the entire synthesis process, DAA uses a mixture of expert systems and conventional algorithmic programming to accomplish its tasks. Expert systems are notoriously slow, so this approach attempts to reduce processing time by restricting them to performing only some of its tasks. Future research with DAA will include experiments in optimization based on the chip's layout. This will allow the system to revise its design so that if a particular signal delay is too great, for example, DAA will redesign the architecture to allow the floor planner to reduce the delay to an acceptable value.

2.4. Facet

The Facet system [TseC86] was also developed at Carnegie-Mellon University. Unlike the rule-based CML-DA system, Facet uses an algorithmic approach to data path synthesis. It uses an ISPS of the circuit to be constructed as its input. The ISPS description containing variables is translated to a Value Trace in which the values are derived from the variables. This representation, which is essentially a control/data flow graph, is used as an intermediate form.

Lifetime analysis of variables and clique partitioning are used to allocate registers for values. The lifetime analysis is similar to what is done by optimizing software compilers to allocate variables to CPU registers. The resulting circuit may store several variables in the same register at different times, or it may put copies of the same variable in several registers. A clique partitioning algorithm assigns the ISPS variables to a minimal number of registers. The translator ensures that the algorithm is performed correctly. Facet also attempts to group registers into scratch pad memories to reduce the overall area. Lastly, data path operator cells and interconnections are allocated. This system has been used to synthesize data paths for the IBM 370 computer.

2.5. MIMOLA

The Mimola system is an interactive design aid for digital processors. The original Mimola Software System 1, MSS1 [Zim679], used a 'design by example' approach, in which the user wrote several algorithms of the type to be implemented by the hardware. The Mimola language contained macros for loops and subroutines, and new constructs could be added by user-defined macros. The algorithm explicitly specified each state transition of the processor and parallel operations in the same state. The Mimola system analyzed the input algorithms and allocated enough hardware to perform each state transition in one clock cycle, and generated utilization statistics for each hardware cell. The user examined these statistics to reduce the amount of hardware by eliminating seldom-used cells. Mimola then performed another iteration with the modified hardware.

The second version of the Mimola system, MSS2, is a considerably enhanced version of the original system [MarP84, MarP86]. The new system is composed of many different programs, called components, which communicate by intermediate files. The user may still describe the system at a register-transfer level, or at a higher level using a Pascal-like language. Mimola uses a multi-port memory for storage of variables; the number of ports is determined from the high-level description. MSS2 also supports modular architectures, whereas MSS1 did not. The control style used by Mimola uses 'direct encoding' of microcode, which appears to be a vertical format.

2.6. Elf

Elf is a prototype hardware compiler developed at Carleton University and Bell-Northern Research [GirE84]. Its input was a behavioural description of the circuit, written as an Ada program. Elf generated a net list of standard cells to realize the circuit; it did not, however, synthesize the controller. A block diagram of the Elf compiler is shown in Figure 2-3. Elf generated circuits using a microsequenced register-multiplexer-operator architecture.

A never-implemented compiler translated the Ada program into a Control/Data Flow Graph (CDFG). The Ada program contained timing constraints, which could be used to specify the maximum time to perform the entire program, or the maximum time which could elapse between any two points in the program, such as limiting the time spent in loops. This timing information was used to direct the allocator. If the timing constraint was very tight, the allocator was forced to add hardware cells to increase parallelism and chip area; if it was not

Figure 2 - 3.

ORGANIZATION OF THE ELF COMPILER

tight, the allocator used fewer hardware cells to reduce chip area, but this implies less parallelism and more serial processing.

Although Elf used rules to design the circuit, it was written in LISP rather than being constructed with an expert-system shell. This had the advantage of higher processing speed, although a more efficient compiled language would have been even faster. The rules used by Elf, called 'graph productions', were used to transform the CDFG as will be described in Chapter 4. In addition to allowing serial/parallel processing tradeoffs, they also considered the area occupied by cell interconnect and the control memory.

2.7. HAL

This hardware compiler, currently under development at Carleton University and Bell-Northern Research, is similar to Elf in many ways [PauP86]. Its input is also a conventional high-level software programming language, which is translated to a control/data flow graph. The graph is then optimized before the data path allocator selects hardware cells. The output generated by HAL is also a net list of standard cells. Unlike Elf, which was written entirely in LISP, HAL uses a 'multi-paradigm' approach implemented in the LOOPS environment. This allows a mixture of conventional procedural programming in LISP, object-oriented programming, and rule-based expert systems. HAL also assumes that the final circuit will contain a microcoded controller. The main effort, however, is aimed at producing the data-path synthesizer, not the control synthesizer.

HAL is composed of three main modules. The first, called Initial, accepts the data flow graph of the algorithm and partitions it into

control steps. Each control step corresponds to a micro-control word.

The partitioning method used by HAL consists of three phases:

- 1) The concurrency of similar operations is minimized. This is intended to reduce the amount of required hardware by reusing hardware cells in different control steps.
- 2) The MidHAL module is called to provide an estimate of the hardware resources required to realize the algorithm.
- 3) The concurrency of operations requiring similar resources is minimized.

The last phase is similar to the first phase, but since an estimate of the hardware allocation is performed by phase 2, this phase may affect operations ignored by phase 1. For example, the data flow graph may require concurrent subtraction and comparison operations. These are different operations, so they will not be considered by phase 1. If MidHAL decides to use the subtracter to perform the comparison, these operations will use the same resource (the subtracter). Consequently, phase 3 will attempt to move one of the operations to another clock cycle to reduce the concurrency of the common resource.

The second module, MidHAL, accepts the partitioned data flow graph, a time constraint, and a resource library. It selects a set of resources from the library to realize the operations specified by the data flow graph. The selection is performed with the aim of minimizing a cost function based on the area and interconnection requirements of each cell in the resource library.

The final module, ExHAL, performs the binding of hardware cells to operations in the data flow graph and the mapping of the data flow graph to a data path. This is performed in seven steps:

- 1) Hardware cells are assigned to operations in the data flow graph. This is done while attempting to minimize the amount of storage cells and interconnect by reusing hardware where possible.
- 2) Storage cells are created and inserted into the data flow graph.
- 3) Registers are assigned to storage operations. The previous step simply creates abstract storage cells wherever they are required. This step binds each storage operation to a register.
- 4) The data path is created by inserting multiplexers as required, and by interconnecting the hardware cells. HAL currently assumes a register-MUX architecture.
- 5) Registers are combined by clique partitioning techniques to reduce the number of physical registers.
- 6) Unnecessary registers are eliminated; this situation occurs when the input of an operator cell is stable so that a register is not required.
- 7) Multiplexer operands are interchanged to reduce the number of inputs to the multiplexer. This phase searches for hardware cells which perform commutative operations, so that inputs can be moved from left to right and vice versa. A breadth-first search of potentially 'good' interchanges is performed until no reductions in overall cost are achieved by interchanging inputs.

In selected examples, HAL has produced data paths with fewer hardware cells and less interconnect than the Facet system, but HAL is a more recent program. Planned enhancements to HAL include ex-

tensions to consider MUX/bus tradeoffs and object-oriented simulation. Unlike some other systems, HAL is intended to be fully automatic with little or no user interaction once the synthesis process has begun. This is not a desirable feature in the short term as it will be more difficult to win the acceptance of hardware designers.

2.8. Synapse

Synapse [SubP86] is an expert system for VLSI design under development at AT&T Bell Laboratories. The input consists of functional and performance specifications; the output is a symbolic layout of the synthesized cell. This symbolic layout may be processed by the Mulga virtual grid layout system to produce the final masks. Synapse is a distributed system implemented in a mixture of conventional programming languages (Pascal) and expert-system shells (KEE), running on VAX minicomputers, AI workstations, and VLSI workstations. Its organization is shown in Figure 2-4.

Synapse operates as a collection of cooperating experts; this arrangement was also used in Cadre [AckB85]. One of the aims of Synapse is to develop a system with learning capability. One of its most unusual features is the fact that the input is represented by algebraic expressions. The synthesis process in Synapse is:

- 1) Simplification of the input expressions using axioms and rewrite rules.
- 2) Interactive transformation of the input expression to an equivalent expression.

Figure 2 - 4.

THE SYNAPSE EXPERT SYSTEM

This diagram shows the general organization of the SYNAPSE expert system for VLSI design which is currently under development at AT&T Bell Laboratories

- 3) Automatic mapping of functional-level expressions to architectural-level expressions. This step includes general and technology-dependent optimizations:
- 4) Partitioning of the expression, using performance requirements, to obtain an architecture.
- 5) Conversion of the structural description into a physical layout.

A circuit synthesized by Synapse is controlled by a centralized state machine, which may be synchronous, asynchronous, or self-timed. In principle, the controller may be distributed.

2.9. Common Features

The hardware compilers described in the previous sections represent only a small portion of automated VLSI synthesis research carried out in the world. Nevertheless, there are some features which appear to be common in many hardware compilers.

Flow graphs are a popular intermediate representation of the algorithm to be realized in hardware. This is not surprising considering the ease of showing hardware concurrency in a Control/Data Flow Graph. Furthermore, serial/parallel tradeoffs may be explored by graph rewriting and control step partitioning. The Elf and HAL programs perform these operations automatically, guided by user-specified timing constraints.

Another common feature is the independence of data path synthesis and control path synthesis. Some researchers have expressed the belief that control and data path synthesis should be performed concurrently [ParA84]. All of the hardware compilers examined, how-

ever, perform control path synthesis after the data path has been designed. Many programs, such as MacPitts, Elf, HAL, and CMU-DA, assume that the controller will be implemented by microcode, with PLA-based controllers as an option.

The prototype system developed in this thesis demonstrates a method of microcode generation as a part of control path synthesis. It also assumes that control path synthesis follows data path synthesis. As such, it requires both the Control/Data Flow Graph of the algorithm, and the final net list of hardware cells created by the data path synthesizer. In addition, of course, it requires access to the cell library used by the data path synthesizer.

CHAPTER 3

3. HARDWARE DESCRIPTION LANGUAGES

Software for computer-aided design (CAD) of VLSI circuits is just beginning to include tools for design automation. Most systems provide designers with floorplanning, layout, routing, design rule checking, area compaction, and simulation.

3.1. Textual Descriptions of Hardware Structure

Current systems are heavily oriented towards graphical interfaces. The graphical representation of the final layout provides the best feedback in situations where the designer is responsible for the low-level design. Some VLSI CAD systems use various textual formats to represent the structure of the circuit. Some formats, such as Clf [MeaC80], are very primitive, being limited to describing only the actual mask layers. The text shown in Figure 3-1 is a Clf description of a CMOS inverter. It consists of alphabetic codes, such as 'L CF' for specifying the field oxide layer in a CMOS process. This is followed by a code for a geometric primitive, such as 'B' for box. The numbers give the X-Y coordinates of the box. This is a purely geometric description of the circuit. It is very difficult for a person to extract the function of a circuit from such a description.

Figure 3-1. CIF Description of a CMOS Inverter

```

DS 1 100 8;
9 inv;
L CF; B 28 28 112 184;
      B 28 74 68 155;
      B 28 78 68 31;
      B 28 28 112 0;
L CPW; B 104 120 90 26;
L CPP; B 44 90 68 155;
      B 44 44 112 0;
L CC; B 12 12 112 184;
      B 12 12 24 132;
      B 12 12 68 132;
      B 12 12 68 56;
      B 12 12 112 0;
L CP; B 12 140 24 92;
      B 8 44 34 140;
      B 56 12 66 156;
      B 10 28 13 132;
      B 64 12 62 28;
L CM; B 28 28 112 184;
      B 106 12 45 184;
      B 14 12 133 184;
      B 32 28 22 132;
      B 8 28 58 132;
      B 12 104 68 34;
      B 8 28 78 132;
      B 14 12 -1 132;
      B 58 12 111 132;
      B 8 28 58 56;
      B 8 28 78 56;
      B 28 28 112 0;
      B 106 12 45 0;
      B 14 12 133 0;
DF;
C 1;
End

```

Other formats, such as ABCD [VIVID], represent the circuit at a higher level than CIF. Figure 3-2, which shows the ABCD description of a CMOS inverter, is much more readable than the CIF format for the same layout. This is still a geometric representation, but it contains some structural information. Note, for example, that transistors are treated as separate entities (d1 and d2 are n_type and p_type devices) rather than as polysilicon overlapping a device well.

Figure 3-2. ABCD Description of a CMOS Inverter

```

#
# $Header$
#
# Created by ICE 1.2 on Tue Nov 18 10:42:25 1986
#
begin cinv bbox=(6,3,10,8)
  vss_se:      pin      metal (10,3)
  vss_sw:      pin      metal (6,3)
  vdd_ne:      pin      metal (10,8)
  vdd_nw:      pin      metal (6,8)
  out_e:       pin      metal (10,6)
  in_w:        pin      metal (6,6)
  c3:          contact   vdd (9,8) or=n
  c2:          contact   vss (9,3) or=n
  c1:          contact   autocontact (7,6) or=n
                wire     metal (6,6) (7,6)
                wire     metal (8,6) (10,6)
                wire     poly (8,7) (7,7) (7,4) (8,4)
                wire     metal (6,3) (8,3) (10,3)
                wire     metal (6,8) (8,8) (10,8)
  cd1D:        contact   autocontact (d1.d) or=n
  cd2S:        contact   autocontact (d2.s) or=n
                wire     metal (cd2S) (cd1D)
  d2:          device    p_type (8,7) or=n
  d1:          device    n_type (8,4) or=n
end cinv

```

Although this description is easier to understand than the CIF code for the same layout, it still gives no indication of the circuit's function. Languages for design synthesis must allow higher levels of abstraction than those offered by existing CAD systems.

3.2. Languages for Hardware Compilation

As VLSI CAD systems begin to incorporate design synthesis tools, hardware description languages (HDLs) must allow the designer to write functional or behavioural descriptions of the circuit to be designed. Initially, these HDLs will be quite similar to current programming languages. Ironically, hardware compilers currently use textual input similar to software programs. Software environments, how-

ever, are moving toward visual programming [RaeG85]. Some visual programming environments use state diagrams [JacR85] which are commonly used for hardware description. VLSI design environments will probably end up using graphical user interfaces, as they do today, but at a higher level of abstraction.

Since hardware compilation is analogous to software compilation, high-level HDLs are required for the same reasons as high-level software programming languages. The designer is allowed to focus on the high-level design, with the design automation system taking care of the actual implementation. The increase in productivity should be comparable to what can be achieved by using the C programming language instead of assembly language. It is likely, of course, that parts of the circuit will be more important to the overall performance than others. Allowing designers to override the synthesis process by including low-level structural descriptions should ensure that these critical sections can be hand-crafted for maximum performance.

To use the software analogy, high-level languages should still allow programmers to link assembly language subroutines. Although the hardware compiler is ultimately responsible for the efficiency of a synthesized circuit, good design of the HDL is crucial to the ease of use, and user acceptance, of the design automation system. It is not surprising that hardware compiler developers also invented their own HDLs in an attempt to create their version of the ideal design environment.

3.3. VHDL

One of the problems with current VLSI CAD systems is that each vendor has a proprietary description language. This lack of standardization means that translators are needed to convert files from one format to another. To further complicate things, some standard formats, such as CIF, are open to interpretation resulting in one vendor's CIF from being incompatible with another and requiring modifications in the technology definition files.¹

In an attempt to forestall similar chaos in future design automation systems, the U. S. Department of Defense is promoting the VHSIC Hardware Description Language (VHDL) as a standard design automation HDL [DewA86]. VHDL's main strengths are the fact that the procedural language uses the same constructs as the Ada programming language [ShaM85], and the fact that it can describe a circuit from an abstract behavioural level down to the gate level [AylJ86].

Since VHDL is based on Ada, systems designers will find it easier to use than HDLs which are not based on established programming languages. Ada also provides constructs for structured design. Circuits may be designed using functions to provide hierarchy, and packages for modularity. Unlike their software counterparts, VHDL functions cannot cause side effects because global variables are not allowed.

Data types supported by VHDL reflect the hardware orientation. Although modern high-level software programming languages support

¹ This is based on personal experience with the Electric and VIVIL programs.

floating-point, integer, and character data types. VHDL also defines a time data type. Timing information is essential for describing hardware systems, and as we shall see later, design automation tools also require timing information to direct their synthesis strategy.

3.4. Other Hardware Description Languages

VHDL is neither the first, nor the last, hardware description language. Many HDLs can trace their origins to software programming languages. Zeus [Ger85], for example, uses the syntax and structure of Modula-2, AHPL [Hil78] is based on APL, and MacPitts [Sou83] uses LISP-like syntax. In each case, the HDL is an extension of the basic programming language.

Other languages, such as Conlan [Pilk85], were developed specially for hardware description. Most conventional languages do not express concurrency, timing, or signals. Conlan defines a signal data type for describing periodic waveforms. Signals are contained in carriers, which are either of the terminal or variable data type. While optimized for a particular application, Conlan requires a designer to learn another completely new language.

3.5. Software Programming Languages for Hardware Description

Rather than developing new, special-purpose HDLs, some researchers have examined the feasibility of using conventional software programming languages for hardware description. The main disadvantage of this approach is that conventional programming languages are not suitable for low-level structural descriptions. They are very good for high-level functional description, and have the advantage that de-

signers do not need to learn a new language. Prolog has been examined as a potential VLSI design language by [Suz85], and the Elf compiler used Ada as its input language. It should be remembered, however, that Elf was intended purely for circuit synthesis from high-level descriptions.

One advantage of using an existing programming language, as the foundation for an HDL is that many language design issues should already have been addressed by the originators. Modern programming languages are designed for modularity and maintainability, with mechanisms such as version control for separate compilation of modules. Clearly, any HDL should support structured design. By using established languages, the developer of an HDL can be saved from 'reinventing the wheel'. Modern languages such as Ada or Modula-2 support structured design. By using packages, modules, and pragmas, hardware-dependent features can be added to the basic language without changing the language itself.

3.5.1. Extensible and Object-Oriented Languages

Extensible languages such as LISP show another way of using software programming languages for other tasks. In the case of LISP, the language may be extended by user-defined functions and macros. Unlike subroutine libraries in FORTRAN or Pascal, the functions and macros of LISP may be used as freely as LISP primitives. The user cannot distinguish between system-defined functions and user-written functions.

Object-oriented programming, which is another development in software programming, may also be used for hardware description.

The Smalltalk language was developed in the 1980's at the Xerox Palo Alto Research Centre as part of research into user interfaces. Object-oriented programming, which formed the basis of Smalltalk, has recently received much attention from the Artificial Intelligence community as one possible way to meet the goals of the fifth-generation computer projects. Object-oriented programming is based on the model of human interaction; people perform tasks by sending messages to each other, and their knowledge about solving problems is carried around with them. Using this model, object-oriented languages treat programs as collections of objects which send messages to each other. Each object has a message selector list which defines what messages it 'understands'. The code for processing messages, which probably involves sending messages to other objects, is contained in a list of 'methods'. Thus whenever an object receives a message, the message is compared against the message selectors to invoke the appropriate method for processing the message [KaeT86].

Another feature of object-oriented programming is the concept of classes and inheritance. Objects which share some properties may be grouped into a class. Other objects, which share some properties of a class, but also have their own differences, may be placed in subclasses. By using the class hierarchy, programmers need not define each object explicitly. Instead, objects may be defined by how they differ from their parent class. Any missing information is automatically obtained by the system from an object's parent.

The attractive feature of object-oriented programming is that it is a natural way of describing hardware. It is very easy to think of adders, registers, and other hardware cells as objects. The class hier-

archy can be used to define general objects, such as ALUs and adders, with specific instantiations defined as a subclass. All the generic information about the hardware cells can be stored in their parent class. This approach is also attractive for simulation, since the flow of data can be represented by message passing.

Object-oriented languages such as Smalltalk are difficult for novices because of their unusual syntax. The use of classes and inheritance is another potential stumbling block for programmers accustomed to conventional languages. Extensions of conventional languages, such as C++, attempt to bring object-oriented concepts to established languages. Another approach is to use environments such as LOOPS, which integrate LISP, object-oriented programming, and rule-based expert systems. The flexibility of such an environment, which allows parts of a system to be programmed in the most appropriate style, is why LOOPS was chosen for the HAL data path synthesizer.

3.6. Effect of HDL on Control Synthesis

The general structure of an HDL will affect the control synthesis part of hardware compilation. The control constructs in the HDL will affect the control/data flow graph (CDFG) produced by the HDL translator. A simple HDL with control constructs such as those found in FORTRAN IV, for example, would tend to use many conditional branches (IF condition, GOTO label), counted DO loops, and subroutine CALLs. Limited multi-way branch capability may be required by the arithmetic IF statement, which branches to one of three labels, as determined by the control variable which is either less than 0, equal to 0, or greater than 0.

A hardware description language based on Modula-2 would add many more constructs which would have to be represented in the CDFG. The CASE statement, for example, is a multi-way branch comparing one test variable against an arbitrary number of values. Furthermore, the test values need not represent all values which can be taken by the variable. Other control structures which would appear in the CDFG are WHILE and REPEAT loops. The HDL translator has two choices: either the control structures can be represented 'as is', or they may be decomposed into a sequence of more primitive control structures. The CASE statement, for example, could be represented as a series of simple IF statements; the WHILE and REPEAT statements can be represented by a combination of simple IFs with GOTOs to perform the looping. Expansion of control structures implies that simpler hardware will be used in the control section. If a CASE statement is expanded into IF statements, the controller needs to be capable only of simple conditional branches. If it is not expanded, the controller would need to be able to perform a multi-way branch directly. The former case also implies less hardware in the controller, saving area but increasing the processing time.

A more profound effect on control synthesis occurs when the HDL supports concurrency. Immediately, the control synthesizer must decide how concurrent tasks and their inter-process communication mechanisms will be implemented in hardware. This problem will be discussed further in Chapter 6.

CHAPTER 4

4. DATA PATH SYNTHESIS

Most hardware compilers treat data path synthesis separately from control path synthesis. The data path of a circuit represents the hardware cells and interconnections which are used to manipulate data. These are the hardware cells which implement the high-level circuit description. The control path, by comparison, represents the hardware which generates control signals for the data path, to ensure that operations occur at the correct time.

4.1. Overview of Data Path Synthesis

Data path synthesis is probably the most difficult phase of the hardware compilation process. The data path synthesizer has the task of constructing a circuit to perform the specified function while meeting all given constraints.

The synthesis of data paths from a high-level algorithmic description can be broken down into several subtasks. Although the actual steps will depend on the sophistication of the compiler, and the types of optimizations performed, the general sequence of processing steps is shown in the following list:

- 1) Translation of the high-level description to an intermediate form for subsequent processing. Many hardware compilers use a control/data flow graph representation, such as the Value Trace in CMU-DA, or the CD₁G in Elf.
- 2) Operations in the algorithm must be assigned to machine states. The algorithm contains implicit scheduling information in the form of precedence relationships. Some languages may require the user to provide explicit state information.

- 3) hardware cells must be allocated to the operations in the algorithm. This phase, called 'module binding' by some hardware compilers, involves searching cell libraries or invoking module builders to synthesize the required hardware cells.

The first phase, translation of the high-level description to an intermediate form, is equivalent to the parsing phase of software compilation. Conventional optimization techniques [AhoA86], such as moving loop-invariant calculations out of the body of a loop, can also be performed during the construction of the CLFG from the HLL. Unlike conventional compilers, which need not consider concurrency of operations, hardware compilers must take possible concurrency of the hardware into account. The translation strategy for associative operations can have considerable effect on the flow graph and the resulting hardware. Consider, for example, an algorithm which contains the following statement:

$$F := A + B + C + D$$

Conventional practice would be to treat the addition operations from left to right, resulting in the following equivalent statement:

$$F := ((A + B) + C) + D$$

This interpretation contains implicit scheduling information, because the parentheses indicate that the addition of 'A' and 'B' must be performed before the addition of 'C' and 'D'.

If the translator was directed to extract maximal parallelism from the input, the equivalent statement would be:

$$F := (A + B) + (C + D)$$

This interpretation shows that the addition of 'A' and 'B' can be performed concurrently with the addition of 'C' and 'D'. The CDFG will depend on how the original statement is parsed. Control/data flow graphs for the above examples are shown in Figure 4-1.

Unless the data path allocator is capable of rearranging the control/data flow graph as part of its optimization strategies, the amount of hardware and the time required to execute the algorithm will be determined by the parsing method. 'Graph productions', used by the Elf compiler, are one method of transforming the control/data flow graph.

4.2. Graph Productions

A graph production is a rewrite rule for graphs. They are used for two purposes:

- 1) to transform the graph by replacing a subgraph with an equivalent subgraph while exploring serial/parallel tradeoffs.
- 2) to optimize the CDFG by replacing user-specified operations with more efficient, hardware-oriented operations.

In expert systems, rules consist of two parts: an antecedent (the set of conditions which must be true for the rule to apply) on the left-hand side, and the consequents (the set of actions to be performed when the rule is used) on the right-hand side.

Figure 4 -

SERIAL / PARALLEL TRADEOFFS IN CONTROL / DATA FLOW GRAPHS

This diagram shows how two different Control / Data Flow Graphs can be obtained from what appears to be the same high - level program. In this example, the high - level program is

$$A + B + C + D$$

which can be interpreted as: $((A + B) + C) + D$

$$\text{or } (A + B) + (C + D)$$

These different interpretations lead to serial and parallel realizations of the same Control / Data Flow Graph. Compare this with Figure 4 - 3, which shows serial / parallel tradeoffs without changing the form of the Control / Data Flow Graph.

1) Serial: $((A + B) + C) + D$

This CDFG uses the same adder serially. Registers can also be reused, so that only 2 registers are required. This CDFG needs 3 clock cycles to perform the operation.

2) Parallel: $(A + B) + (C + D)$

This CDFG performs as many operations as possible in parallel. It requires 2 adders, but it performs the operation in only 2 clock cycles.

In graph productions, both the antecedent and consequents take the form of graphs. Typical graph productions used by Elf are shown in Figure 4-2. The rule interpreter scans the CDFG and attempts to match the antecedents in its rule library with subgraphs of the CDFG. If the left-hand side of a rule in the library matches the CDFG, then that part of the CDFG may be replaced by the subgraph in the production's right-hand side.

The original CDFG is transformed as its nodes are replaced by the productions. Graph productions which simply substitute one set of CDFG nodes for another set of CDFG nodes are called non-terminal productions. If a graph production leads to the substitution of a CDFG node by a hardware cell, then it is called a terminal production.

Only one production may be applied if two or more graph productions match overlapping CDFG subgraphs. As soon as one production is applied, the subgraph will no longer match the other productions. In this case, a conflict resolution algorithm determines which production will be applied to the CDFG. In the Elf compiler, conflict resolution for terminal productions was based on a cost function which considered cell area, control overhead, and interconnect requirements. The production with the lowest cost was chosen whenever conflicts occurred.

The costs of non-terminal productions were calculated by estimating the amount of interconnect, and the minimum cost required to realize each node in the right-hand side of the production. Exact costs of nonterminal productions cannot be determined without creating a search tree which contains all possible transformations leading

Figure 4 - 2.

SAMPLE GRAPH PRODUCTIONS

This diagram shows only a few graph productions which can be used to optimize a Control / Data Flow Graph. Note that some productions may be applied in both directions (!), while others make sense in only one direction. Terminal productions are applied only in the forward direction

1) Non-terminal Productions

a) Serial / Parallel tradeoffs.

b) Slow operations replaced by faster equivalent operation

c) Replacement of subgraph by single node.

d) Single node replaced by subgraph to reuse existing operator cell.

b) Terminal Productions

Terminal productions were limited in ELF to the form of a single non-terminal node producing a single terminal node. They were used to select hardware cells for data path operations.

to the application of terminal productions. Only then can the costs of the terminal productions be known. In effect, Elf would have to perform an exhaustive search of the design space for each nonterminal graph production applied to the CDFG. In order to limit the depth of this search, Elf considered only the first level of the production as a subgraph; below this level, all nodes were considered in isolation. This improved execution times because graph matching did not have to be applied to the subgraphs. These approximations were required to maintain acceptable computational complexity for cost calculation.

4.3. Control Step Partitioning

In many hardware compilers, the control/data flow graph contains no explicit scheduling information other than the precedence relationships among nodes. An exception was the Maniula system, which required the user to specify each machine state in the input description. In other compilers, such as HAL, the flow graph specifies only that certain operations must occur before others. The graph nodes may be moved in time as long as these precedence relationships are maintained. One of the tasks of the data path synthesizer is to schedule CDFG operations to definite machine states.

Explicit scheduling information is added to the CDFG in Elf and HAL by partitioning the CDFG into control steps. The control step corresponds to a clock cycle, so all operations in the same control step are performed in parallel. Changing the control step partition allows the data-path allocator to change the realization of the circuit from full-serial to maximally parallel. Figure 4-3 shows the effect of different control step partitions for a very simple CDFG.

Figure 4 - 3.

CONTROL STEP PARTITIONING OF CONTROL / DATA FLOW GRAPH

These diagrams illustrate the effect of two different control step partitions on Control / Data flow graphs for the operation

$$A + B + C + D$$

a) Two control steps:

The first control step ($T-1$) requires 2 adders and 4 registers to store the inputs. One of the adders, and two registers, may be reused in the second control step ($T-2$). A circuit generated from this CDFG would use 4 registers and 2 adders; it would perform the operation $(A + B + C + D)$ in 2 clock cycles. This is good if speed is more important than circuit area.

b) Three control steps:

The first two control steps ($T-1$ and $T-2$) each require one adder and two registers to perform their operations. The output of the adder in $T-1$ must be saved during $T-2$, so that an extra register ($R3$) is used in $T-3$. Only one adder is required, because it may be reused in each control step. This circuit would use 3 registers. It would be better than the one at left if chip area is more important than speed, because it saves one adder and one register, at the cost of one extra clock cycle.

The control step partitioning is driven by user-specified timing constraints in the functional description. Since chip yield varies inversely with chip area, the compiler should always attempt to generate the smallest circuit which will meet the user's speed constraints. In order to minimize the amount of hardware, the data path allocator will tend to reduce parallelism of the circuit. If timing constraints allow, this would lead to a fully-serial implementation. If this is not possible, the allocator is forced to increase parallelism, and hence the number of hardware cells, so that the operations are completed in the specified time.

4.3.1. Urgency Scheduling in Elf

The Elf compiler used an urgency scheduling algorithm, which calculated an 'urgency factor' for scheduling a given node in a given control step. The urgency was calculated only on the basis of timing constraints. At the start of the control step partitioning process, no CDLG nodes were assigned to control steps. The urgency scheduling algorithm needed three passes over the set of nodes which had not been assigned to a control step. At the end of these passes, the CDLG would have some subset of its nodes assigned to a new control step. This process was repeated until all CDLG nodes had been assigned to a control step.

In its first pass, all nodes which had an urgency of 1 were assigned to the current control step. An urgency of 1 meant that the node must be scheduled in the current control step, or else the timing constraints would be violated and the circuit would not meet the specifications.

In the second pass, all nodes with an urgency less than 1 but greater than a user-specified urgency threshold were examined. If the cost of adding the node to partitioned CDFG was low, so that no new hardware cells other than registers or interconnect had to be added to the circuit, then the node would be placed in the current control step.

The final pass considered those nodes with an urgency less than the threshold. As in the second pass, these nodes were added to the current control step only if no new hardware was required to realize them.

Nodes which were not added to the current control step remained unassigned. The urgency of these nodes would reach 1 at the last interval where the operation could be scheduled. If it was not possible to add a node without requiring extra hardware, it would be unassigned until this last possible moment. Since its urgency would have reached 1, the allocator was forced to add it to the partitioned CDFG. This method tended to schedule operations in their latest possible control step because of its insistence that a node with an urgency less than 1 could not increase the number of hardware cells.

4.3.2. Multi-Cycle Operations

Control step partitioning must also take into account operations which cannot be executed in a single clock cycle. The multi-cycle nature of an operation may be implicit, such as in the fact that multiplication requires more than one cycle, or explicit when the CDFG contains loop constructs. Multi-cycle behaviour which requires explicit control over several operations, rather than a simple 'start' signal,

creates problems for the controller if it must be executed in parallel with sequential operations. If the loop requires a pre-determined number of iterations, its code can be unwound. The controller then steps through sequential states only. The loop cannot be unwound if the number of iterations is data-dependent. The controller must then find some way of looping a subset of its control signals. This situation is best handled by a separate control unit which receives a start signal from the main controller and sends a termination signal when it has completed its operations.

4.4. Load Balancing

In many cases, CDFG nodes may be moved within a range of control steps without violating any precedence relationships. In these cases, the data path allocator must have a strategy for scheduling the CDFG nodes in specific control steps.

Common scheduling techniques favour assigning operator cells to time slots as soon as data are available (ASAP scheduling), or wait as late as possible before assigning operators to time slots (ALAP scheduling). These scheduling methods run the risk that operator nodes may be bunched into one control step, while the next control step may be almost empty. Although there is nothing inherently wrong with ASAP scheduling, it may lead to a design with more hardware cells than what a human designer would use.

The HAL compiler introduced a load balancing algorithm to reduce the number of operator cells required by an algorithm. The load balancing algorithm moves operator nodes in the CDFG between their ASAP and ALAP control steps, in an attempt to find a compromise

which distributes operations as evenly as possible over all control steps. An example contrasting ASAP and load balanced scheduling of a simple CDFG is shown in Figure 4-4. Load balancing does not affect the number of control steps required by the algorithm; it simply moves CDFG nodes to reduce the number of operator cells required to realize the circuit.

The load balancing algorithm begins by assigning a weight to each operation. This weight is simply the reciprocal of the number of control steps between the operation's earliest possible control step and its latest possible control step. If an operation is constrained to only one possible control step, its weight is 1. If it can be assigned to one of 4 control steps, then its weight is $1/4$. The weights of each operator type are summed for each control step. The average of these sums, computed over all control steps, represents the ideal distribution of operations in the flow graph. Physical constraints prevent the circuit from achieving the ideal distribution in all cases, because the ideal value may not be an integer. Obviously, a circuit cannot contain fractions of hardware cells. Nevertheless, the ideal distribution is used to guide the load balancing algorithm by showing which operations are causing the greatest imbalance. Once each node in the control/data flow graph has been assigned a weight, the algorithm uses heuristics to shift the nodes.

The load balancing algorithm begins by examining the most costly imbalance. The nodes responsible for this imbalance are shifted in the flow graph so that the imbalance is reduced. As a consequence of moving these nodes, other operations may be forced to move as well to maintain the precedence relationships. The weights of each node

Figure 4 - 4

COMPARISON OF ASAP, ALAP, AND LOAD BALANCED SCHEDULING OF CDFG

These diagrams compare three possible operation scheduling strategies on a simple Control / Data Flow Graph. The CDFG is constrained to use 4 clock cycles for all cases

a) As Soon As Possible Scheduling

Each operation is placed in the first available clock cycle. Note that this places 4 multiplications in the first clock cycle, and none in the third or fourth.

A circuit generated from this CDFG requires:

- 4 Multipliers
- 1 Adder
- 1 Subtractor
- 1 Comparator

b) As Late As Possible Scheduling

In this case, each operation is placed in the last available clock cycle. Note that this spreads the multiplications apart, but places 2 additions in the third clock cycle.

A circuit made from this CDFG needs:

- 2 Multipliers
- 2 Adders
- 1 Subtractor
- 1 Comparator

c) Load Balanced Scheduling

The Load Balancing algorithm, used by HAL, spreads the multiplications so that at most 2 occur in the same clock cycle (as in ALAP case), but it also moves the additions so that only one adder is used during any clock cycle.

A circuit made from this CDFG requires:

- 2 Multipliers
- 1 Adder
- 1 Subtractor
- 1 Comparator

and the new distributions of operations are recomputed for the modified flow graph. The algorithm terminates when the weight of each node becomes 1, indicating that it cannot be moved to another control step without violating precedence constraints. Some nodes will end up at their ASAP control steps, others will be at their ALAP positions, while other nodes will be somewhere in between. The load balancing algorithm provides a more flexible scheduling than either ASAP or ALAP scheduling. It also seems to produce a better control step partitioning than urgency scheduling because it is driven to make the distribution of operations as even as possible. Urgency scheduling tended to add extra hardware as the urgency of a node became 1. In effect, urgency scheduling deferred assigning nodes to control steps to their ALAP positions unless they could be added without requiring extra hardware.

4.5. Hardware Cell Selection

Once the CDLG has been optimized and partitioned into control steps, the data path synthesizer may begin synthesis of the actual hardware. In many cases, the manufacturing technology becomes significant only now.

One of the problems faced by the data path synthesizer is the fact that more than one hardware cell may be used to perform an operation specified by a CDLG node. One purpose of graph productions, or other optimization techniques, is to replace inefficient hardware realizations with more efficient ones. For example, if a CDLG specifies division by an integer power of 2, this operation can

be carried out by a simple bit-shift. If a division operation is specified, but the dividend is unknown, then a shifter cannot be used.

Other cell substitutions cannot be made by graph productions even when all operands are known. If an addition operator is specified in the CDFG, it is reasonable to assume that an adder cell will be used to perform the operation. If an OR operator is in the CDFG, then OR-gates may be placed in the final circuit to perform the operation. If a CDFG contains many additions, subtractions, ANDs, and ORs, the data path synthesizer must consider the effect of using discrete cells for each operation, or the use of a larger cell, such as an ALU, to perform all of the operations. The decision to use an ALU cannot be made until all required operators are known.

In order for the data path synthesizer to make this decision, it must make more than one pass through the CDFG, or alternatively it must be able to look ahead in the CDFG. The Elf compiler was able to perform cell substitution if the new cell resulted in a lower overall cost. It could, for example, substitute an ALU for an adder.

In the case of compilers which generate net lists of standard cells, cell selection is a matter of searching the standard cell library. The data path synthesizer must select the hardware cell which can perform the specified operation within the given time constraints. If several cells meet these specifications, then the cell with the smallest cost (area, number of control lines) would be chosen. The actual cell placement can be performed by conventional placement and routing programs.

4.6. Effect of Data Path Synthesis on Controller Design

Since the controller must synchronize the activity of the data path, it is obvious that choices made by the data path synthesizer will have a direct effect on the control synthesizer. Transformations of the CLFG, such as the graph productions in Elf, can be used to convert complex control structures into simpler structures. The CASE statement, for example, can be transformed into a series of IF statements.

4.6.1. Effect of Control Step Partitioning

The control step partitioning phase of data path synthesis defines the states for the controller. If a microcoded controller is used, each control step corresponds to a micro-control word. The control synthesizer may need to add other micro-control words which are not explicitly specified in the CLFG. A multi-way branch, for example, may require extra jump instructions at the end of each conditional block. The number of control steps and concurrent operations defines the size of the control memory. The hardware cell selection phase determines the number of control lines required by the data path. If a centralized controller is used, the total number of control lines will affect the amount of interconnect required by the entire circuit. As feature size on chips decreases, the cost of interconnect becomes more important. The use of a good hardware cell selection strategy can decrease global interconnect requirements.

4.6.2. Effect of Hardware Cell Selection

The number of control lines also depends on the types of hardware cells chosen. An ALU will require several control lines to select its function, whereas a simple adder will require only a clock. Substitution of an ALU for single-function cells may reduce the area used by active components, but it will increase the number of control signals generated by the controller. In a microcoded controller, this implies an increase of one bit for each extra control line. In a long algorithm, this one bit can account for a significant increase in control memory area. On the other hand, if an ALU is substituted for several cells which need an ADD/SUBTRACT* control signal, the total number of control lines may be reduced. As far as control line cost is concerned, the decision to perform such substitutions cannot be made until all hardware cells have been allocated. Substitution of one ALU for one adder/subtractor may increase the control cost, but if additional adder/subtractors would be required, the ALU may reduce the control cost in the long run. Thus an increase in the control cost at one stage of the allocation may lead to a net decrease in overall cost at the end.

CHAPTER 5

5. CONTROL PATH SYNTHESIS

The focus of attention in hardware compilation research has been the synthesis of data paths. The preceding chapters attempted to provide a brief overview of the problems facing hardware compiler developers. As with data path synthesis, the real problem with control path synthesis is not how to do it, but how to do it well. Some systems, such as Elf or HAL, simply assume that some form of control synthesizer will be available as a post-processor.

Other systems, such as CMU-DA, do emphasize control path synthesis. CMU-DA is intended as a design tool for general-purpose computer circuits; it has been used to synthesize CPL and microprocessor circuits. The focus of this thesis, however, is on the problems of control path synthesis for application-specific ICs.

The general control-path synthesis problem is very difficult. The designer must consider the architecture of the controller, the functional division between the control and data paths, state assignment and minimization, control signal encoding, parallelism issues, and 'meta-control' issues (how to control the controller) [NagA81a].

Another issue is the effect of interconnect required by control signals on the overall wirability of the chip, which suggests that distributed control may be desirable as transistor density increases. This, of course, raises issues about the synchronization of the distributed system. Not surprisingly, today's research in distributed operating systems for multiprocessor computers may pave the way for tomorrow's integrated circuits.

This chapter will look at some of the issues mentioned above from a general viewpoint. The next chapter will look at how these

issues affect microcoded controllers. In particular, the problems of mapping high-level HDLs to silicon will be examined.

5.1. Self-Timed Circuits

All but the simplest circuits will require some form of controller to synchronize the data path cells. A notable exception to this are self-timed circuits, which directly realize a data-flow graph. Each operator cell in a self-timed circuit contains logic for detecting when its operation has completed, and the output is stable [DeaC80]. The cell does not begin processing data until all of its inputs are ready, as indicated by control signals from the self-timed cells connected to its input ports. Although a cell may use an internal clock to perform complex operations, the overall circuit is asynchronous.

In addition to saving the area required by a central controller, and sidestepping the problems of clock skew on large chips, the self-timed circuit also provides the fastest possible processing. Self-timed circuits are not used as the final architecture by hardware compilers because of the large chip area requirements. A self-timed circuit can directly implement a data-flow graph, with the penalty that each operator node must be unique. Most of the hardware compilers described earlier used various techniques to reduce the number of operator cells by performing operations serially where possible.

5.2. Independent/Concurrent Synthesis

The general consensus among hardware compiler developers is that the target architecture will be synchronous. This implies that a

control sub-system will be required in addition to the data path. Immediately, developers are faced with the following questions:

- 1) at what stage in the synthesis process should the control section be designed?
- 2) what architecture should be used for the controller?
- 3) unless the control synthesizer is designed for a fixed architecture, what criteria should be used to select the architecture of the controller?

Several authors have expressed the opinion that, ideally, controller design should proceed concurrently with data path design. This belief is supported by the argument that synthesizing the control and data paths simultaneously will allow better exploration of the design space. Effects of data path allocation can be evaluated immediately on the partially-synthesized controller, and vice versa. Additionally, simultaneous design is appealing if it could be performed on a multi-processor computer; the AI aspect of 'data path designer' and 'controller designer' expert systems working together in the manner of Synapse is much more interesting than algorithmic programs running sequentially from a batch queue.

It is not entirely clear, however, that the benefits of concurrent control path and data path synthesis are as great as some anticipate. A counter-argument in support of data path synthesis followed by control path synthesis is that by performing the synthesis tasks sequentially, the control path synthesizer can examine the complete data path for global optimizations. Furthermore, human designers tend to design the control circuitry after the data path has been defined, so sequential synthesis more closely models human behaviour.

Another advantage of performing the controller synthesis after data path synthesis is that the architecture of the controller can be independent of the data path. By running the control synthesizer with different specifications, the architecture of the controller could be changed without affecting the data path. This would not be possible if the control and data path synthesis was performed concurrently.

5.3. Architectural Considerations

The architecture of the controller is limited to three main choices: random logic, PLA, or microcode ROM. Each presents different tradeoffs in area, speed, and ease of synthesis. Random logic, the most efficient architecture in terms of area, is the most difficult to synthesize. Microcode, on the other hand, is easier to synthesize, but does not use area as efficiently, nor is it as fast as the hardwired approach.

The choice of architecture also depends on how the hardware compiler is to be used. If it is intended to synthesize a production chip, as the compilers should eventually be capable of doing, then obviously the smallest and fastest controller should be created. If the compiler's output is intended only for a prototype, perhaps one which will be hand-optimized, then the controller should be created so that it can be easily modified to reflect any changes in the data path.

5.3.1. Hardwired Controllers

A hardwired controller is usually constructed from flip-flops or counters and random logic. Design of such a controller requires the use of a state table, with transition and output functions.

Although a hardwired controller is the most space-efficient design, there are several problems before such a controller can be generated automatically. The main problem is that hardware compilers developed to date do not handle discrete logic gates well unless the logical operations are taken from a data flow graph. Before a HAL-type compiler could generate a hardwired controller out of flip-flops and arbitrary gates, it would need a data flow graph describing the controller's function.

An alternative to the flip-flop and logic gate approach to finite state machines is the use of a PLA. Automatic PLA generators can synthesize the necessary circuitry given the logic equations, making them an attractive addition to a circuit synthesis system [Bark85]. PLA folding can then be used to reduce the area of the controller.

Logic synthesizers, such as described by [Bark86], process logic equations to yield a circuit implemented as individual gates or PLAs. These programs require excessive computation time when the number of terms in the logic equations is large.

5.3.2. Microcoded Controllers

The other major implementation style for controllers is microcoding. The data path control signals are stored in a ROM, together with next-address information. A micro-program counter with next-address calculation hardware executes the microprogram in the correct sequence. Depending on the sophistication of the hardware implementation, the microcontroller may be capable of subroutines and multi-way branches in addition to the basic conditional and unconditional branches.

The main advantage of microcoded control in hand-crafted systems is that the behaviour of the circuit can be changed simply by changing the bits in the control memory. This is very useful while debugging a design, particularly since the control memory can be implemented as a RAM or EPROM to facilitate reprogramming. Changes to the data path can be accommodated by adding or deleting fields in the control memory. This also requires changes to the interconnect between the control memory and data path cells. Changes to the algorithm require only a new bit pattern in the control memory. This makes microcoded controllers much easier to work with than hardwired ones, where a small change in the data path or algorithm requires a complete redesign of the controller.

The flexibility of microcoded controllers is a great asset for manual design. Changes to the controller can be made relatively quickly and easily. These considerations are not as important in a design automation system, since any change to a chip will be made in the high level description. The hardware compiler then generates a new version of the chip with the design changes.

If the hardware compiler is to be used as a designer's assistant, rather than creating finished designs, then a microcoded control section still has merits. In this scenario, the hardware compiler is used to create a prototype of the design. The design space can be explored before selecting a particular implementation by modifying the design constraints. The design can then be optimized by hand before producing the final product. An automatically generated circuit with a microcoded controller would be much easier to modify than the same circuit with a hardwired controller.

Though microcoded controllers are easy to modify, they also tend to take up more area and are slower than hardwired controllers. The control memory is usually sparse, so that much of its area is wasted. Area compaction techniques exist, but their use means that it is no longer possible to make simple changes in the algorithm. These algorithms have the same results as PLA folding, although they reduce controller area, they destroy the regularity of the structure storing the control information. Nevertheless, microcode compaction is a quick way to reduce the area of a prototype chip.

5.4. Functional Division of Control and Data Paths

One deceptively simple problem facing compilers which perform control synthesis after data path synthesis concerns the functional division between the data path and control synthesizers. Since the data path and control synthesizers run independently, the overall synthesis system must take into account the interactions between these sub-processes. A related problem is the allocation of hardware for dedicated control tasks. This interaction can best be seen by considering how condition flags are brought from the data path into the control section.

5.4.1. Interaction During the Synthesis Process

Any realistic controller will require status inputs from the data path to determine its next state. For example, let us assume that a circuit is required to implement some algorithm described by the following pseudo-code:

```

initialize data path on power-up
LOOP
    perform some operations
    LOOP
        perform some more operations
    UNTIL x = 0
FOREVER

```

In this example, the controller is required to perform the inner loop until the value of some variable 'x' becomes 0.

As a software program, this situation is handled very easily. A memory location, which may be a CPU register, is associated with the symbol 'x'. The loop is performed by testing if this memory location is equal to the value 0; the result of this test is returned in a CPU flag. The program executes a conditional branch, which uses the value of the CPU flag to either branch or execute the next sequential instruction.

In hardware compilation, however, it is not quite as simple. The variable 'x' is a data path variable which is being used for a controller function. Since the data path synthesizer will process the data flow graph before the control synthesizer, it has the option of processing all references to 'x', or leaving all or some of them for the control synthesizer. If the data path synthesizer processes all references to 'x', it may be 'infringing' on the control allocator's job when it gets to the UNTIL statement. On the other hand, the data path synthesizer may decide that 'x' is being used as a control variable and so it should be left for the control synthesizer.

In this case, the control synthesizer may be forced to perform data path synthesis operations, because 'x' is probably used for operations other than just testing when to exit the inner loop. This is undesirable for several reasons. Firstly, it is very likely that such a

'control synthesizer' will end up doing a considerable amount of data path synthesis. Only rarely will a software program use a variable purely for control operations such as loop counting or conditional branching.

In most cases, variables will be used for both purposes. This would also make the control synthesizer more complex, because it would have to scan the data flow graph for variables which have not yet been instantiated. Otherwise the control synthesizer would be faced with the problem of generating the controller for an incomplete circuit. The control synthesizer would first need to synthesize the circuit associated with each 'control variable' before proceeding with its synthesis tasks.

This is one situation where concurrent synthesis of data and control paths may provide a clean solution. In situations where it is unclear which synthesizer should do most of the work, both synthesizers could work together. In the simple example shown above, the data path synthesizer could process all references to the variable 'x' until it became clear that 'x' was also being used as a control variable. The data path synthesizer could then inform the control synthesizer how the control variable was being used. In response to this message, the control synthesizer could request the data path synthesizer to perform some operation, such as including extra cells to generate a one-bit status signal from several ALU flags. Such an approach could be implemented using the 'blackboard' model of cooperating expert systems, as used by the Cadre system [Ack88b].

In more pedestrian systems, the developer must make a definite decision about what duties will be performed in the data path and

control synthesis phases, and how the system will handle such potentially ambiguous circumstances. In the interest of generating a 'good' data path, the best choice would be to have the data path allocator handle all ambiguities. The global optimizations performed by the data path synthesizer will have a greater impact if they are applied to a complete circuit. Otherwise, the number of iterations needed by both the data path and control synthesizers will increase if the control synthesizer ends up synthesizing the data path as well.

5.4.2. Interaction in Sophisticated Controllers

The distinction between data path and control functions is blurred in the final circuit as well. In this case, the question is how much dedicated hardware should be included in the controller, where it is unavailable for data path functions. Alternatively, the controller can be kept simple, with as little dedicated hardware as possible.

In the previous example, the circuit was required to execute two loops: a main loop which ran until the circuit was reset, and an inner loop which terminates when a data path variable takes on the value 0. A hardware-intensive controller design could use a special register for controlling loops. If the circuit required nested loops, a loop-control stack would be used to take care of loop nesting. In addition to the special registers for storing the loop control variables, such a controller would need a comparator to generate a signal to exit the current loop.

By adding dedicated registers for loop control, the controller would also require control signals to load, push, pop, increment, decrement, and test the contents of the registers. In effect, the control-

ler now requires its own instruction set. The controller approaches the sophistication of a simple microprocessor. It is equivalent to a bit-slice microcontroller such as the AMD-2909 [MyeG80].

5.4.3. Interaction in Simple Controllers

The alternative approach does not use dedicated loop control registers; consequently, the micro-opcodes to control the state machine are not required. Loop control is accomplished by manipulating data path registers, and using computational resources in the data path. Status signals are sent from the data path to the controller, where simple next-state generation circuits can determine if the controller should continue looping. The architecture of the controller is simplified, by eliminating dedicated registers, at the expense of more data-path operations.

5.4.4. Controller Complexity Tradeoffs

The dedicated registers within a sophisticated controller are unavailable for general data path operations. This leads to unnecessary duplication of operator cells, and an increase in chip area. If the dedicated controller resources are used during only part of the algorithm and are idle at other times, this additional hardware is used inefficiently.

In a simple controller, the data path synthesizer must know what control inputs will be required by the controller. In many cases, such as testing if a variable is equal to 0, this can be indicated by one status line from an ALU. If the hardware cell does not generate explicit status bits for the required condition, the data path synthe-

sizer may be required to add discrete components to derive the function. The ALU may have flags indicating if the result of the last operation was zero, negative, or overflow, but conditions such as 'less than or equal to' must be derived from these flags by combinatorial logic. This can be done either by the data path or controller synthesizers. The data path synthesizer can examine the data flow graph to see what status information is required by the controller.

Alternatively, the data path synthesizer can provide the 'raw' status lines to the controller, leaving it to the control synthesizer to add the combinatorial logic needed to obtain the required logical relation.

5.5. State Assignment and Minimization

Careful attention must be given to the assignment of machine states to their binary representation in controllers implemented as hardwired random logic or PLAs. This is a classical problem in synchronous circuit design. The control synthesizer must derive a minimal state transition table for the algorithm. This will ensure that the number of flip-flops will be minimal. The complexity of the logic used to determine the machine's next state depends on the binary representation of the states.

State assignment is not an issue in microcoded controllers because the binary representation of the state is simply an address in the control memory. States are assigned automatically as the micro-program is written. The next-state logic must determine only if the micro-program counter will be incremented or loaded directly from some external source. 'State reduction' techniques are program opti-

mizations such as those performed by software compilers, and lead to the removal of unnecessary words from the control memory.

5.6. Control Signal Encoding

State assignment and minimization is performed to reduce the size of the controller, particularly for hardwired architectures. The purpose of control signal encoding is to reduce the number of columns in the control memory of microcoded controllers. The amount of encoding is determined by the parallelism of the data path. Increasing the parallelism of the data path usually decreases the chances of encoding the control memory. Maximal parallelism is achieved by no encoding, where each control signal is stored by a unique bit in the control memory. This horizontal microcode format is shown in Figure 5-1(a). Most encoding schemes rely on grouping mutually exclusive micro-operations, so that the encoded control field specifies which micro-operation is to be executed by each control word. The control field must be passed through a decoder before leaving the controller to generate the actual data path control signals, as shown in Figure 5-1(b). This is a single-level encoding scheme, where the output of the control memory directly drives the decoders.

An alternate approach to single-level encoding is bit-steering. In this case, an extra field is required to control the interpretation of the encoded control fields, as shown in Figure 5-1(c). The steering bits select one of several decoders connected to the output of the control memory. Bit steering is a two-level encoding of the control memory. Encoding the control memory reduces the area of microcoded

Figure 5 - 1

CONTROL WORD FORMATS

a) Horizontal Microcode

Each control signal in a horizontal control word has its own unique bit position. Thus a circuit with 'n' control points requires a ROM at least 'n' bits wide. Sequencing information such as branch addresses and condition code select information require additional bits in the control word. The output of the control memory requires no decoding.

b) Vertical Microcode

A vertical control word consists of several control fields with decoders attached to their output. Each field is associated with a hardware resource, such as an ALU or register bank. Vertical encoding limits parallelism in the data path by encoding each control field into a single control signal. In many cases, combinations of control signals are invalid because they might, for example, lead to two talkers on a bus being active at the same time. Encoding reduces the width of the control memory, but generally leads to longer microprograms.

c) Bit - Steering

Bit steering requires the micro-operations to be partitioned into mutually exclusive sets. Each set is encoded, as for vertical microcode. The fields of the sets are shared in the control memory, so a steering field (S1 and S2 above) must be added. This field determines which decoder is used to decode the control field before it is sent to the data path.

controllers, but any changes to the algorithm may require a complete re-encoding of the entire control memory.

5.7. Parallelism Issues

The main reason for using an application specific IC to perform a function, rather than programming a microprocessor, is to take advantage of the parallelism available in hardware. One of the tasks of data path synthesizers is to extract the parallelism from an algorithm written in a high-level language. Traditional languages such as Pascal do not support concurrent programming, although there are extended versions such as Concurrent Pascal.

If a hardware synthesis system is based on a concurrent programming language such as Ada, the programmer can describe the algorithm with explicit concurrency. Ada allows concurrency at the task level where a task is, in effect, a procedure which runs concurrently with other tasks. The synthesis system must convert the designer's procedures and tasks into hardware. This poses additional problems for the control synthesizer. In conventional sequential programming, a single controller could be used for the complete algorithm. Once the algorithm is broken up into concurrent tasks, the control synthesizer must ensure that each data path will function correctly. Synchronization must also be available at the hardware level for communication between concurrent subcircuits which correspond to software tasks.

5.8. Meta-Control

This issue was alluded to when describing sophisticated controllers. In a simple finite-state machine, meta-control is not a problem,

because it does not really exist. If the finite-state machine is implemented as a counter, its sequencing information is derived only from its present state. In a hardwired controller, the sequencing associated with loops and subroutines in the high-level description is implicit in the state transition table.

Microcoded controllers, however, allow the designer to make the controller a complete microcomputer. The OM2 controller chip, described in [MeaC80], is a good example of a sophisticated controller. This chip contained stacks for nested subroutines and loops. The OM chip set was a general purpose computer, and because the controller and data path were on separate chips, the use of such dedicated resources did not degrade circuit performance or area.

For an application-specific data path, the advantage of these dedicated resources must be weighed against the increased area. Furthermore, the controller requires its own instruction set to control the dedicated hardware. Next-address calculation is not as simple as incrementing the micro-program counter or loading it with a branch address from the control memory.

5.9. Interconnect Cost and Distributed Control

In large integrated circuits, the area required by interconnect represents a large fraction of the entire chip area. Additionally, long wires increase signal delays; decreasing feature size also increases resistance and signal delays in wires. Consequently, considerable effort is devoted to minimizing interconnect length. Iterative cell placement algorithms can reduce the size of interconnection nets between data path cells and within the controller itself.

The controller is usually constructed as a single block occupying a distinct part of the chip's floorplan. Even if it is placed at the centre of the chip, the control signal lines can be long. Complex algorithms will require many control signals, making the global routing problem even more difficult. Routing area can be reduced by reducing the number of control signals required by the algorithm, and by distributing the controller across the chip. The cost function used by the Elf data path allocator included a component for the number of control lines required by a hardware cell. Selecting cells with relatively few control lines reduces the complexity of the controller (if hardwired), the width of the control memory (if microprogrammed), and the interconnect used by control signals. In this case, single-function cells, such as adders, are preferable to multi-function cells such as ALUs.

Distributed control is another way to reduce controller area and control line length. Rather than constructing the controller as one monolithic block, can be split into several sub-controllers to match the functional modules in the data path. This approach to microprogrammed control is suggested in [KriM84]. Using this model, each major functional block in the circuit would have its own controller. This approach is well suited to circuits such as microprocessors, which typically have several complex subcomponents such as bus interface units, execution units, on-board cache control, virtual memory support, and floating point accelerators.

The data path may not always be split so conveniently into functional units. The controller may be split according to the floorplan, so each subcontroller controls only the hardware in its immedi-

ate surroundings. PLAs can be partitioned using algorithms such as the one described by [LeMG83a]; this avoids control lines running the length and breadth of the chip. Of course, this architecture still requires a master controller to synchronize the subcontrollers. Distributed control, which will be explored further in the next chapter, also presents an elegant solution to concurrency issues.

CHAPTER 6

6. MICROCONTROLLER DESIGN ISSUES

Chapter 5 described some issues in control synthesis from a general point of view. This chapter examines control synthesis issues as they apply to microcoded architectures.

In some sense, microprogramming can be similar to assembly language programming. There are two main differences between these activities. Firstly, microprogramming can include explicit control over parallelism, whereas assembly language programs on conventional computers are strictly serial. This parallelism accounts for the increased performance of algorithms implemented in hardware, compared to the same algorithm running on a microprocessor.

The second difference is that resources in hardware are much scarcer than those available to software programmers. In most circumstances, programmers need not worry about squeezing their programs into the smallest number of bytes. The availability of large physical and virtual memories provides more memory than most applications need. Microprogrammers do not have such luxuries; they must deal with the hard limits imposed by the number of registers, hardware stack depth, and other available resources.

A minimal microcontroller, as shown in Figure 6-1, requires three main components:

- 1) a memory to store the control information for each controlled hardware cell.
- 2) a micro-program counter to store the current state of the controller.
- 3) next-state generation circuits which determine the next value of the micro-program counter.

Figure 6 - 1. A SIMPLE MICROCODED CONTROLLER

This diagram shows the architecture of a simple microcoded controller. This controller has 3 main parts: the control memory, the next - address generator, and the micro - program counter.

The control memory contains one bit for each data path control signal, stored in a horizontal control word format. The micro - program counter defines the state of the controller. It points to the control word which is sent out to the data path.

The next - address generator in this controller is very simple. The condition - code multiplexer is used to select one of the condition codes from the data path, as specified by the condition select field in the control word. The 1 - bit output from this multiplexer controls the multiplexer connected to the micro - program counter. If the condition is TRUE (= 1), then the micro - program counter is loaded with the branch address. If the condition is FALSE (= 0), then the micro - program counter is loaded with the output of the incrementer, which is just the next sequential address. This is how the controller performs conditional branches.

Depending on the sophistication of the controller, these basic components can be augmented by dedicated registers for loop control, subroutine return addresses, and other ancillary hardware. As the controller becomes more complex, it becomes more suitable for general purpose control tasks. It is possible to use such a controller on an application-specific integrated circuit, but this defeats the purpose behind custom design. Ideally, the controller for a custom IC should be customized to the same degree as the data path.

6.1. Microcomputer Generators

An extension of the sophisticated microcontroller concept is the use of a complete microcomputer for control applications. Currently, microprocessors provide control at the board level, but as feature size continues to decrease, there is no reason why a microcomputer could not be integrated with other data processing functions. Some researchers have developed programs which synthesize microcomputers from a user's specification. Although these programs do not meet the criteria for hardware compilation, they are interesting because they provide a means of generating customized controllers.

6.1.1. PLEX

The Plex system [BurM83] synthesized microcomputers from user specifications. The user chose the word width, number of registers, and other parameters such as the amount of on-board memory. Plex then created the hardware to meet these specifications. One other important feature of this system was that the user also customized the instruction set for the microcomputer. This was done simply by

writing assembly language programs which were to run on the system using instructions selected from a 'global instruction set'. The hardware was tailored to execute only the subset of instructions which were actually used. The synthesizer automatically created the instruction decoding circuits for the microcomputer. This allowed the synthesized microcomputer to be optimized for the user's applications. The resulting microcomputer consisted of three subsystems:

- 1) the program section, which contained the program ROM, program counter, hardware stack, and interrupt handler.
- 2) the data path, which contained both ROM and RAM for data, the ALU and scratchpad registers, and the I/O circuits.
- 3) the instruction decoder and status register.

Communication with the outside world was provided by interrupt request and acknowledge lines, and by an address bus with separate input and output data buses.

6.1.2. Microcontroller Synthesizer

Another group used a similar approach to microcontroller synthesis [JinB85]. The difference between a microcomputer (PLEX) and microcontroller, as described in this paper, is very fuzzy. The microcontroller synthesizer was intended to design controllers for use in telecommunications applications. The microcontroller consisted of a microprocessor, memory, communications protocol hardware, and both analog and digital I/O hardware. The synthesizer relied on several commercial packages for simulation (NETLIST from MIT) and layout (MAGIC from Berkeley).

The architecture of the microcontroller, shown in Figure 6-2, was microcoded using two PLAs. One PLA provided the data path control, while the other was used to control the controller itself. by partitioning the control memory into two PLAs, the total area was reduced. An additional PLA was used to determine the next address for the program counter. The microcontroller also contained a hardware stack for subroutine return addresses, an interrupt handler, on-chip RAM, an ALU, and a shifter.

The input to the synthesizer consisted of a parametrized description of the microcontroller. The user specified following parameters: the number of bits in the data path and program counter; the number of registers, I/O ports, and interrupts; the depth of the hardware stack. The user also defined the operations performed by the ALU, shifter, and branch PLA. Some of these parameters were obtained by analysis of a user-supplied program; as in Plex, the user wrote the program using a 'global instruction set'. Thus, if the user did not need arithmetic instructions, the ALU was constructed without arithmetic capabilities. Similarly, the shifter was constructed to execute only the shifts specified by the user's program.

6.1.3. Microcomputer Generators for ASIC Design

The use of a microcomputer to control a data path synthesized by a hardware compiler would be inefficient in terms of both space and time. Since the aim of an ASIC design is to maximize speed and minimize area, there is no need to burden the chip with an on-board microcomputer which requires time to decode its own instruction set before it can execute any control over the data path. Another con-

Figure 8 - 2. ARCHITECTURE OF JINKS' MICROCONTROLLER

This diagram shows the architecture of the microcontrollers synthesized by the program described by B. W. Jinks [Jin885]. The dashed blocks indicate enhancements for testability. The interconnections are simplified. The program counter can be loaded either from the RAM or from the jump PLA. The current state is fed back to the control and data PLAs either from the program counter or from an interrupt vector. The shifter, ALU, and RAM are actually connected by two busses.

cern is that the data processing capability of a microcomputer is not required when the rest of the chip is designed to perform a dedicated function. Furthermore, the general-purpose microcomputer does not have the same degree of parallelism as the application-specific IC. For these reasons, the Plex approach to controller synthesis does not appear to be suitable for integration with hardware compilers as described in previous chapters.

6.2. Microcoded Control for ASIC Data Paths

Given that the approach to microcontroller synthesis presented above does not appear suitable for use with hardware compilation, what other options are available? A simple microcoded controller is one workable solution. If the controller is too simple, however, other inefficiencies are introduced. Branch addresses and constants stored in the control memory are one source of inefficient memory usage. It is impossible to define an 'ideal' microcoded controller for ASICs synthesized by hardware compilers, because the needs will depend on the algorithm being executed.

6.2.1. Strengths of Microcoded Controllers

The real issue is why use microcoded, rather than hardwired control for these circuits. Although the data path will be synthesized by a hardware compiler, ultimately with no user intervention, a fully automated chip synthesis system is still far in the future. Until such compilers become a reality, users will need to interact with the synthesis process, and they may need to manually modify the compiler's output. To use the software compilation analogy, the user will occa-

sionally want to replace some of the high-level code with assembly language. Although human intervention may be considered undesirable from an intellectual viewpoint, it is doubtful if hardware designers would accept synthesis systems which exclude them from the detailed design process. Hardware compilers will first need to become accepted as designer's assistants for exploring the design space and rapidly prototyping a design.

Microcode allows the designer to easily modify the behaviour of the data path without requiring major changes to the hardware design. Changes to hardwired controllers would be extremely difficult without resynthesizing the entire chip. One of the credos of VLSI design is the use of highly regular structures to replace a hodgepodge of gates. The control memory provides this regular structure for storing control information. The ancillary circuitry required by microcoded controllers is generally much simpler than a hardwired controller, although this may not be true for very simple controllers.

Weinberger arrays [UllJ84] are an alternative to PLAs and discrete gates for logic synthesis. They provide a regular structure, but are not restricted to sum-of-product forms of logic as most PLAs. Weinberger arrays tend to grow slower than PLAs; unfortunately, they tend to grow much wider than they are high, and their optimal design is more complex.

6.2.3. PLA-based Controllers

A third approach, which provides the benefits of regular structures, together with the design concerns of hardwired controllers, is the use of PLAs. This requires the controller to be defined as logic

equations, usually in a sum-of-products form. Considerable effort has been devoted to automated PLA synthesis, as exemplified by [Ayr879] and [DeMG83]. The PLA synthesizer must perform logic minimization and state assignment as for hardwired controllers. The problem of logic minimization for functions with many inputs is another active research topic, but practical programs exist [bra84].

The relative merits of PLAs versus other design styles have been explored by [CbrM83]. In this study, the random-logic controller of the Motorola MC6800 was transformed to several variants of microprogrammed and PLA-based architectures, in addition to the timing generator style used by the Intel 8085. This study found a single-PLA architecture to be most wasteful in terms of both speed (2 times slower than the original MC6800) and area (40% increase). Other approaches, such as multiple PLAs, yielded more acceptable results with approximately 15% area increase and 15% increase in execution time. One possible reason for the large increase in execution time for the single-PLA implementation is the delay due to long polysilicon lines.

If the PLA is constructed with only a single layer of metal, then either the product or sum lines must be made in polysilicon. As the size of the PLA increases, so does the length and capacitive loading of these lines. This can be overcome by partitioning the PLA into several smaller PLAs, thereby reducing the length of the polysilicon lines. This has the additional benefit of reducing the overall area by taking advantage of sparsity in the PLA.

Considerable research has been done on PLA partitioning, for example by [DeMG83a] and [PapC85]. Alternatively, the PLA can be speeded up by using a double-metal process, since metal lines have

lower capacitive delays than comparable polysilicon lines. Although substitution of metal for polysilicon will increase speed, it will not reduce the PLA's area.

Folding is another approach to PLA area optimization. This method reduces the total number of columns in the PLA. Discussed in [LilJ84], folding rearranges the PLA's AND-plane inputs so that the same column is used for two variables; one input arrives from the top, while the other arrives from the bottom. Folding of the OR-plane is also possible for greater area reduction. The disadvantage is that changes to the logic equations are very difficult, if not impossible, in a folded PLA.

6.3. High-Level Constructs in Microcode

The ultimate goal of hardware compilation is to create a complete integrated circuit from a high-level behavioural description. This will allow the algorithm to be expressed as a software program in a language such as those described in Chapter 3. Consequently, the controller must be capable of performing the control constructs found in the high-level language. As a bare minimum, Pascal-like languages include the following flow control primitives:

- 1) Counted loops.

```
FOR var := initial_value TO final_value DO
  body;
```

- 2) Indefinite loops where the test is executed before the loop body, so that the body may never be executed.

```
WHILE condition_is_true DO
  body;
```

- 3) Indefinite loops where the test is executed after the loop body, so that the body is executed at least once.

```
REPEAT
  body
UNTIL condition_is_true;
```

- 4) Subroutine and function calls.
5) Conditional branches.

```
IF condition THEN
  block_1
ELSE
  block_2;
```

These can, of course, be nested to provide multi-way branching capability where the path taken can be determined by an arbitrarily complex set of conditions.

- 6) Multi-way branches where the path to be taken is determined by the value of a simple variable.

```
CASE variable OF
  value_1:      block_1;
  value_2:      block_2;
  ...
  value_n:      block_n;
OTHERWISE: default_case;
```

- 7) Unconditional branches, either in the form of the infamous GOTO, or disguised as the 'exit' and 'continue' statements in the C programming language.

In each program is the potential for two levels of concurrency. The control/data flow graph shows the low-level concurrency which can exist in any algorithm. This concurrency is at the level of mul-

visual operations. The time required for each operation is known exactly, so that scheduling is straightforward when the results of several concurrent operations are required as inputs by another operation.

Languages which allow explicit concurrency, such as Ada, introduce constructs for inter-task communication. This high-level concurrency is at the level of individual modules where duration of each task is indeterminate, so that scheduling becomes a problem. A protocol such as the Ada rendezvous is needed for concurrent tasks which interchange data. Tasking is a natural way to express the high-level concurrency available in hardware, so the control path synthesizer should be capable of using it to implement parallel hardware when required.

6.4. Microcode Implementations of Loops

Loops are easy to implement in microcode. The problems which arise concern the exact implementation style. As described in Chapter 5, there are many cases where the distinction between the data and control paths blurs. Loop control is one of these cases.

Two main styles for loop control are available. The first, which would be used by a simple controller, uses the hardware resources of the data path together with simple conditional branch instructions in the microcontroller. The second style, which is appropriate for use in more sophisticated controllers such as the OM2 chip described by [MeaC80], dedicates resources within the controller for loop control.

6.4.1. Loop Unrolling

In certain cases, loops can be avoided entirely. If the number of iterations is known, the loop can be unrolled into straight-line code. This technique is used by many vectorizing compilers for supercomputers to trade execution speed for code size.

When used in a hardware compiler, loop unrolling may reveal parallelism in the algorithm. Consider the simple loop below.

```
FOR i := 0 TO 15 DO
  a[i] := b[i] + c[i];
```

After unrolling, the code for this loop becomes:

```
a[0] := b[0] + c[0];
...
a[15] := b[15] + c[15];
```

In this simple loop, the iterations of the loop are independent of each other, so that each line can be executed in parallel if required. Naturally, this would require 16 adders in a fully-parallel implementation, but a load balancing phase could adjust the parallelism to satisfy any desired degree of parallelism.

In an optimizing hardware compiler, the only cost of unrolling loops when possible is the control memory size. Assuming that only one adder was available in the simple case above, the total time to execute the unrolled loop would be 16 clock cycles. If the loop is not unrolled, the execution time would be 16 clock cycles plus the loop setup time.

Unfortunately, only a limited class of loops can be unrolled. The compiler must be able to calculate the number of times the loop will be executed. The example above, where the limits of the FOR loop

2 of/de 2

MICRO

are explicitly stated, is the simplest case. A more complex situation arises when the limits of the loop are stored in variables, but their contents can be determined by data flow analysis. By using constant propagation [AhoA86], the compiler may be able to compute the number of iterations in the loop.

If the loop depends on run-time data obtained from the chip's I/O ports, then the compiler will not be able to calculate the number of iterations in the loop and hence the loop cannot be unrolled. Most WHILE and REPEAT loops probably will not be suitable for unrolling because they often derive the loop termination condition from a variable which is read or modified in the loop body. In these cases, the loops must be performed by the controller.

6.4.2. Loop Control Using Data Path Resources

The main reason for using data path resources for loop control is that any special hardware, such as increment/decrement logic and comparators, is available for the data path synthesizer. This ensures better utilization of these resources, since they can be used to perform operations not related to loop control when the resources are available. Furthermore, it is likely that no additional resources, other than interconnect, will be required if the data path is used to store loop counters and perform the loop termination tests. Any reasonably complex algorithm will require several registers and some form of ALU; these can be used to store and manipulate the loop counter.

Looping constructs must be expanded in the control/data flow graph if the resources used will be involved in load balancing or other optimizing operations. The expansion can be performed either

by the high-level language parser, or by a non-terminal graph production. Figure 6-3 shows how traditional FOR, WHILE, and REPEAT loops can be represented in a control/data flow graph. One prerequisite for using such expansions for loops, or for any other control construct, is that the data path allocator must know the number of cycles which will be consumed by the controller while performing these operations. Otherwise the data path allocator would be unable to guarantee that the scheduling of operations will meet any user-specified constraints.

A consequence of using the data path to perform loop control related operations is that the controller itself does not require any special opcodes for looping. This further simplifies the controller. The microcode to execute a FOR loop could be written as shown below:

```

1:   loop_counter := initial_value
2:   comparator_input_1 := loop_counter
     comparator_input_2 := final_value
3:   condition_code_select := comparator_equal_flag
     If comparator_equal_flag = 1 THEN GOTO address n+1
4:   first microcode word in body of the loop
     ...
n:   last microcode word in body of loop
n:   GOTO address 2
n+1: first microcode word after loop body

```

Note that the body of the loop also requires microcode to increment or decrement the loop counter as appropriate. This example requires only conditional and unconditional branches in the microcode. The controller requires a status bit from the comparator to indicate when the value of the loop counter is equal to its final value to terminate the loop. All other operations are accomplished by controlling hardware cells in the data path. The location of the loop counter would be determined by the data path allocator. Loop setup does re-

Figure 6 - 3.

REPRESENTATION OF LOOPS IN THE CONTROL / DATA FLOW GRAPH

These diagrams show how the three loop types (FOR, WHILE, REPEAT) can be represented in a Control / Data Flow Graph. It is up to the data / control synthesizers to interpret the special CDFG nodes; MiSyn implements LOOP - EXIT and LOOP - END with conditional branches.

a) FOR and WHILE Loop

Both FOR and WHILE loops test whether or not to continue the loop at the start of each iteration, so that the loop may not be executed at all depending on the conditions. The CDFG for these loops is identical. The difference is hidden in the loop body, which will increment or decrement the loop counter in a FOR loop. The LOOP BEGIN node simply marks the start of the loop. All operations between this node and the LOOP END are repeated in each loop iteration.

The calculation of the loop exit condition codes is performed by ordinary data path cells. If the condition is complex, this part of the loop could take many clock cycles.

If the loop exit condition is true, the program continues at the nodes after the LOOP END node. If the condition is false, the loop body is executed. When the algorithm reaches the LOOP END node, execution continues at the LOOP BEGIN node.

b) REPEAT Loop

The REPEAT loop differs from FOR and WHILE loops in the position of its LOOP EXIT node. In this loop, the loop termination test occurs after the loop body, so that the loop body is executed at least once.

As in the FOR and WHILE loops, the LOOP BEGIN and LOOP END nodes define the loop boundaries. The computation of the loop exit condition codes is performed by the data path. Although the LOOP EXIT and LOOP END nodes are separate nodes in the CDFG, they may be treated as a single EXIT&END node in the control graph.

quire loading the counter (at word #1), but this could be overlapped with other data path operations. The loop itself requires two cycles to compare the loop counter with its final value (at word #2) and to branch over the loop body (word #3) to end the loop. Another clock cycle is used by the unconditional jump (at word #n) to the top of the loop. The overhead in the above example is 1 clock cycle for set-up and 4 clock during each loop iteration.

WHILE loops can be microcoded in a similar manner. The only difference between a WHILE loop and the FOR loop in the example above is the microcode needed to compute the loop termination signal. If the high-level language program uses a complex logical expression in the loop test, this will have to be performed by the microcode. This adds clock cycles to the start of each loop iteration.

REPEAT loops require that the microcode for computing the loop termination signal be moved to the bottom of the loop body. One clock cycle is saved because only one conditional jump to the top of the loop is required. If the loop termination condition is true, the jump is not taken and the controller continues sequential execution of the microprogram.

These situations are equivalent to the compilation of loop control statements from the high-level to assembly level language. They achieve good utilization of available hardware resources at the cost of extra clock cycles during execution.

6.4.3. Dedicated Hardware for Loop Control

The OM2 controller chip described in [MeaC80] contained a dedicated hardware stack for nested loops. Figure 6-4 shows the block

Figure 6 - 4. DEDICATED HARDWARE FOR LOOP CONTROL

This diagram shows how a microcoded controller can be enhanced with extra hardware for loop control. The controller in this schematic contains a special loop counter register with increment / decrement and comparator cells. The increment / decrement cell implies that the loop counter can be used only for definite counted loops (FOR loops), but not indefinite loops such as WHILE and REPEAT loops. This limits the usefulness of the counter.

The counter can be loaded from three sources: the data path, such as the contents of a register, a constant from the control memory, or the value of the counter $+/- 1$. These inputs are selected by a field in the control memory. This field is also used to select the second (right) input to the comparator. The first (left) input is always the loop counter. This comparator outputs a single bit to the condition code multiplexer to signal whether or not the loop should end. The controller uses this status bit and the branch address field to perform the loop.

The counter must also be connected to the data path if it is to be used for array subscript calculations. This two-way communication increases the amount of interconnect on the chip and the complexity of the microcode. As explained in the text, it is better to use the data path resources for looping operations; the data path then needs to send only the loop termination signal to the controller.

diagram of another controller with hardware dedicated to the execution of loops. This approach to loop control can provide more parallelism because of the extra hardware on the chip, but this should not be an issue in hardware compilers which perform control step partitioning of the control/data flow graph. The main practical advantage in such systems would come from the fact that the dedicated loop control hardware would be placed near the control memory and hence should have shorter interconnections than a controller which relies on data path hardware.

The main disadvantage to dedicated hardware is the fact that its utilization may be low. If one part of an algorithm requires nested loops, the control synthesizer may create a loop counter with an associated stack which will be unused during most of the program's execution. Naturally, the controller's dedicated resources cannot be affected by any optimization of the data paths. Additionally, the controller's own instruction set must be expanded to include control over the loop counter and stack.

Another disadvantage with a dedicated loop counter is that it can be used only for counted loops (Pascal FOR loops). Indefinite loops, such as WHILE loops, do not benefit from a dedicated loop counter, particularly since the variables which are used to control indefinite loops are usually manipulated within the body of the loop. If a dedicated loop stack was used to store these variables, the controller would need to shuffle them back and forth between data path operator cells and their storage locations within the controller. This would increase interconnect cost, and time would be required to per-

7

form the data transfers. The entire operation becomes much more complex than necessary, without providing any benefits.

6.4.4. Subscripts for Array Access

Algorithms which use loops often use the loop counter as an index into an array. The data and control paths of a custom IC must allow easy access to loop counters for memory address calculations. The use of scratchpad memory in a custom IC depends on the algorithm and how it is described by the user.

Data path allocators such as HAL assume that registers will be used to store the outputs of hardware operator cells for the inputs at the next clock cycle. On the control/data flow graph, there is an assumed register on each edge at the control step partition. Analysis of register usage can be used to optimize the number of registers, since many variables will be used only temporarily. It is not necessary to have one physical register for each variable in the algorithm. Facet and HAL, described in Chapter 2, are two hardware compilers which perform such register optimizations.

These registers could be combined into a register bank of the type used in general-purpose microprocessors. This architecture has certain advantages when the number of registers becomes large. All registers in the bank share a data bus; this may save interconnect area. The penalty for using memory rather than discrete registers is that only one word of the memory may be accessed at a time. Dual-ported memory is more versatile, but it still does not have the ability to update all words in the same clock cycle. Scratchpad memory would not be useful for often-used variables, but it would be the

natural implementation style if an algorithm specifies arrays. Merging registers into banks is another optimization problem considered in some hardware compilers [TseC86].

Use of array subscripts in custom ICs has some similarities with array address calculations in assembly language. In the simplest case, the subscript can be used directly as the memory address. This requires only a connection between the register storing the index and the memory address register.

A more complex situation arises if one memory is used to store several small arrays. In this case, the base memory address of each array must be added to the array subscript. This is another situation when it is better to have the loop indices stored in the data path rather than in dedicated controller registers. If the data path contains several adders, they may be used to perform address calculations in parallel.

Dedicated hardware in the controller complicates address calculations, particularly for nested loops. Although the array indices could be stored in a stack, either each register would have to be transferred to the data path for the address calculation, or the controller would require even more special hardware to perform the calculation.

6.5. Subroutines

In many hardware compilers, the high-level language parser converts the input algorithm to a control/data flow graph. Normally, each software module would be translated to a separate graph. The subroutine calls would be represented by a 'graph call' node in the flow graph. The hardware compiler would employ some kind of con-

vention for passing arguments between routines; this is one application which would be handled well by a small memory. Note that so far the design of the controller is of no importance. Hardware compilation provides several alternatives for implementing subroutines, both during data path and controller synthesis.

6.5.1. In-Line Expansion of Subroutines

In the same way that loop unrolling can eliminate the overhead associated with microcoded loops, in-line expansion can eliminate overhead due to subroutines. Rather than preserving the modularity of the input algorithm, the parser could 'flatten' the module hierarchy by expanding each subroutine as in-line code. This approach is available in some software compilers for applications where execution speed, and not code size, is of utmost importance.

Flattening the control/data flow graph would have the effect of replacing each 'graph call' node by a subgraph corresponding to the subroutine. Argument passing would be explicit in the nodes and edges entering the subgraph. The resulting control/data flow graph would be much larger than a hierarchical implementation.

As with software compilation, in-line expansion represents a tradeoff between code size and execution speed. In hardware compilation, code size becomes chip size. If the algorithm makes heavy use of subroutines, the in-line expansion could lead to a chip which exceeds the user's area constraints. Furthermore, since the in-line expansion flattens the design hierarchy, the final layout could be unintelligible as the functional units in the algorithm would be merged together. User optimization or modification of the layout would be

more difficult unless the hardware compiler maintained a history record for each component.

The advantage of in-line expansion, other than the speed gained by elimination of subroutine calling and argument passing protocols, is the greater degree of global optimization possible in the control/data flow graph. With a hierarchical graph structure, graph optimizations of the type used by Elf or HAL would be restricted to the individual subgraphs. Optimization of the expanded control/data flow graph would allow all nodes and registers to be used in load balancing or register optimization.

Yet another advantage of in-line expansion is that no modifications are required to the controller. As far as the control synthesizer is concerned, the algorithm is performed by one large program, and all argument passing is invisible in the control/data flow graph.

6.5.2. Subroutine Linkage in Hardware

If algorithm is not flattened, each subroutine will appear as a subcircuit in the final chip. Some means are required for transferring control and passing data between these subcircuits. Subroutine linkage could be accomplished by using a small memory or assigned registers for argument passing between the hardware modules. This would serve the same function as the software stack used by many software programming languages.

The conventional method for handling subroutines in microcode is to use a hardware stack to store the micro-program counter. The depth of the stack depends on the depth of subroutine nesting, which can be determined from the algorithm. This is not possible for recur-

sive algorithms. Tail-recursive algorithms can be automatically transformed to iterative equivalents, but there is no method for automatically transforming any general recursive algorithm.

Introduction of a hardware stack implies that the controller's instruction set must be expanded to include subroutine, CALL and RETURN instructions. These would operate in the same way as their assembly-language counterparts, pushing the contents of the microprogram counter on the stack and loading the subroutine start address for a CALL, and popping the return address from the stack to perform a RETURN. This approach to subroutines is used in commercial microcontrollers [Myc80] and general-purpose controllers such as the CM2 [MeaC80].

6.5.3. Promotion of Subroutines to Independent Tasks

The approaches to subroutines described in the previous sections follow directly from software techniques. Another method, which takes advantage of the parallelism available in hardware engineering, is to treat each subroutine as an independent task. The control/data flow graph for each subroutine would be processed as an independent circuit with its own data path and controller. Arguments can be passed between modules via global memory or assigned registers. Each subcircuit could operate asynchronously with respect to the other subcircuits; the actual hardware would still be synchronous. Handshaking would provide synchronization between subcircuits.

This method has several potential benefits over the more conventional approaches to subroutines. Firstly, the degree of modularity is higher than the software-subroutine style described in the previous

section. In this style, both the data path and control for the subroutine are kept together as a unit in the floorplan of the chip. User modification of the final layout becomes easier because changes to modules are localized. The overall performance of the circuit need not be affected if the timing in some modules is changed because synchronization is accomplished by handshaking.

A second, and possibly greater advantage in very large systems, is the fact that the control section may be partitioned into many smaller controllers. As will be discussed later in this chapter, partitioning should reduce the total area of the control memory and the amount of interconnect between the controller and data path.

Since each subcircuit corresponds to a subroutine in the algorithm, the subroutine call becomes a simple 'start' signal to the controller for the appropriate subcircuit. The caller may then wait for the completion signal before continuing its own processing; this emulates a normal subroutine call. Alternatively, if the results from the subroutine are not required immediately, the caller can continue processing; in this case, the subroutine is treated as an independent task and transfer of data to its caller becomes a rendezvous (in Ada terminology).

6.6. Error Handling

One feature which many software programmers take for granted concerns error handling. In this section, error handling refers to situations such as division by zero or arithmetic overflow. Some languages, such as Ada, do allow programmers to write exception han-

dlers. Programmers not interested in writing error handlers can always rely on the language's run-time library for error handling.

Errors can occur in custom hardware as well, but unlike programmers, hardware designers do not have a run-time library to fall back upon. Unless logic is included on the chip to detect errors such as ALU overflow, the user will have no idea that an error occurred until the wrong data comes out.

In this case, the hardware compilation process differs from software compilation. The designer must specify each error condition and the action to be taken if it occurs. If no handler is specified for a possible error, such as ALU overflow, the only reasonable action the hardware compiler can take is to warn the user.

Error conditions are signalled by the data path to the controller in the same way as condition codes. The controller can either test these error signals explicitly as conditional branches, or use the approach described in section 6.9.2 to determine the branch address directly from the error signals. In the latter case, error conditions are sent to a separate next-address generator which forces a jump to the error handling microcode if an error occurs.

6.7. Control Word Format

The format of the control word determines how the bits are arranged in the control memory. Two basic styles are defined for microprogramming: horizontal and vertical microcode. Other styles which are variations on these basic styles also exist. The choice of microprogramming style affects the size of the control memory, the speed

of the controller, and the ease of modifying the algorithm stored in the control memory.

In this section, a control point is a hardware cell which is controlled by a field in the microcode. Registers, ALUs, and multiplexers are all examples of control points in the data path.

6.7.1. Horizontal Microcode

Horizontal microcode is not encoded. Each control line emanating from the controller is assigned a unique column in the control memory. This allows maximal parallelism in the data path at the cost of control memory area. In a pure horizontal format, all microcode fields, including those used by the controller for next-state determination, are contained in one long control word.

In complex systems, the sparsity of the control memory leads to inefficient area utilization. Nevertheless, controllers with a horizontal control word format are the easiest to debug and modify. Since each control line has its own column in the control memory, modification of one bit has no side-effects on other control fields. Hence it is possible to redefine the control sequence for part of the data path independently of the rest of the circuit.

For hardware compilation, the horizontal format also has the advantage of being the easiest to synthesize. The fields in the control memory are defined by the hardware cells in the data path. One field is allocated for each control point. The number of bits in each field is simply the number of control bits required by the hardware cell. Some cells, such as simple gates, may not require any control lines. A register will probably require only one control bit to latch

its input. An ALU, on the other hand, may require three or more control lines to select its function. This design style also simplifies the data path allocator's control cost estimates.

6.7.2. Vertical Microcode

Vertical microcode uses encoding of the micro-control word to reduce the width of the control memory. This also reduces the parallelism of the data path; the speed of the controller is also reduced slightly because the output of the control memory must be decoded before being sent to the data path. This additional delay increases the cycle time for the controller, but it can be overcome by pipelining. This, of course, adds to the complexity of the controller.

The encoding process greatly reduces the user's ability to modify the algorithm by simply rewriting a few bits. In the best case, the change to the algorithm can be made by modifying the encoded field which controls the desired data path cell. In the worst case, modification is impossible because the modification would violate the constraints under which the control memory was encoded. In this case, the entire control memory must be redefined.

Encoding the control memory also makes the data path synthesizer's control cost estimates more difficult since there is no one-to-one correspondence between the number of control lines and the width of the control memory. The control synthesizer's job is also a bit more difficult because it must encode the microcode fields.

6.7.3. Bit Steering: Two-Level Microcode Encoding

A two-level microcode encoding scheme called bit steering has received some attention in the literature, for example in [MatA81], [LisN85], and [KaoS85]. Vertical microcoding simply encodes the control memory on one level and uses one set of decoders to decode the control word fields. Bit steering uses additional control bits in the microcode to select one of several decoders. This is what makes it a two-level encoding scheme. The first level of encoding is in the microcode fields, which are encoded as for vertical microcode. The second level of encoding comes from the decoder select fields in the microcode. Note that the time required to select the correct decoder increases the cycle time of the controller even more than for vertical encoding.

Bit steering resembles vertical microcode with the addition of the steering field. The cost of this field must also be considered when encoding the control memory. The principle behind bit steering is that sets of mutually exclusive operations are encoded into one field, while the steering field controls the interpretation of the encoded field. The encoding algorithm is more complex than for vertical or diagonal microcode, and the data path allocator's control cost estimation is more difficult. A related control word format uses an opcode or type field to define the interpretation of the entire control word. As with any encoded microcode scheme, user modifications to the microprogram are virtually impossible without complete resynthesis of the control memory.

6.8. Microcode Compaction

Any microcode encoding scheme can be viewed as microcode compaction, because its aim is to reduce the total area of the control memory. Several other compaction schemes, which do not fall neatly into the classes described above, are also available. One approach, which may be called 'microcode fusion', takes advantage of unspecified fields in a sparse control memory. One problem with pure horizontal microcode is that it typically leads to a sparse, and hence poorly utilized, control memory. Conventional compaction techniques look for control fields which are mutually exclusive, so that they may be encoded or bit steered.

The simplest compaction algorithm involves scanning the control memory for columns which are logically related. In the simplest case, some columns may be identical so that elimination of duplicates reduces control word width. Each of the control points controlled by a deleted field would be reconnected to the field kept in the control memory. More complex relationships can be tested, depending on the time the user wishes to spend compacting the memory. This type of compaction requires an exhaustive comparison of the columns in the control memory for each logical relation being tested.

In many cases, however, the definition of equivalence of microcode fields may be modified to include unspecified or don't care fields. The equivalent fields may then be combined into a single control field. Two fields are considered equivalent if they contain the same bit (either 0 or 1) in each control word or if one of the fields is a don't care. The don't care fields are determined from the control/data flow graph.

If a control point is not used in a control step, and if the hardware cell does not perform a data storage operation, then its microcode field can be set to don't care. In [NagA81a], these are called select signals. Since the cell does not latch any values, its behaviour is relevant to the operation of the rest of the circuit only when it is actually used by the algorithm.

In the case of cells which latch their inputs, the controller cannot set unspecified control fields to don't care because this may lead to inadvertent activation of the control point after microcode compaction. These are called evoke signals by [NagA81a]. Closer inspection reveals that even evoke-type cells may have don't care fields. This is true in situations when the cell is used to latch a new value before it is used as an input to another cell. The evoke signal must be off only in the interval between latching a value and using it in another cell. The actions of the cell before latching a new value are not significant, because its contents will be replaced by the correct value before being used.

In the control/data flow graph, all register control fields are of the evoke type, while most other control fields fall are of the select type. Since most circuits will be somewhere between fully parallel and fully serial, it is expected that a reasonably large portion of the control memory will specify don't care fields. This should allow microcode compaction by 'fusing' compatible control fields, yielding a narrower control memory with fewer don't care fields.

It should be remembered, however, that any compaction of the control memory will destroy the main advantage of microprogramming: a compacted memory cannot be modified by the user. In ASIC design,

with microprogrammed data paths, conventional horizontal microcode has the advantage of maximal parallelism, fastest operation, and user modification. If the control memory is too large for the final product, microcode compaction can be used as a post-processing step to reduce the area of a fully debugged microprogram.

6.9. Multi-Level Control

The design styles discussed above all assume one main memory to store the control signals and sequencing information. Much of the information normally stored in the control memory can be moved out to auxiliary ROMs or PLAs, reducing the area of the main memory without affecting the ease of user modification.

6.9.1. Indirect Addressing

The cost of multi-level control is in the cycle time of the controller, which now has additional delays. One possible application of an auxiliary ROM is indirect branch addressing. A schematic diagram of this arrangement is shown in Figure 6-5(a). In a well-structured microprogram, the number of different branch addresses should be significantly less than the total number of control words.

For example, suppose that a microprogram contains 512 words and 64 different branch addresses distributed evenly throughout the address range. In a single-memory controller, 9 bits must be allocated for the branch address. The branch address field can be moved into a separate ROM with 64 9-bit wide words (576 bits total), and the branch address field of the control memory then specifies the address of the real branch address in this auxiliary memory. The indirect

Figure 6 - 5.

USE OF AUXILIARY HARDWARE IN THE CONTROLLER

These diagrams show how extra hardware can be added to the controller to decrease area or increase performance. Unlike the dedicated loop counter shown earlier, these enhancements do not require 2-way communication with the data path.

a) Auxiliary Memory for Indirect Addressing

In this controller, the main control memory uses an indirect branch address field to specify the actual branch address from an auxiliary memory. If the branch address ROM has few, but wide, words, it can considerably reduce size of the main control memory. If the control memory contains 1K words, a 10-bit branch address is required. If the microprogram uses only 64 different branch addresses, then a 6-bit indirect address can refer to the auxiliary ROM. This saves 4K bits in the main memory, at a cost of 640 bits in the auxiliary ROM.

b) Auxiliary PLA for Next Address Calculation

Rather than using jump tables or a series of conditional branches, the controller can use a PLA to generate branch addresses directly from condition codes or data path register values. The program counter must be input to the PLA to define the current state because the same condition codes can be used in different parts of the program to generate different branch addresses. This arrangement can also be used to generate error handler addresses when error signals are input into the PLA.

branch address field needs to be only 6 bits wide, for a saving of 1536 bits in the main memory. The total memory requirements, have been reduced by 960 bits.

Indirect branch addressing is most beneficial when the total number of words is large (implying a large branch address) and the number of different branch addresses is small (implying few words in the auxiliary ROM and a small indirect address field). The timing constraints must allow the controller to make the second memory access to obtain the branch address. This is not a severe problem, as the branches can be delayed by one clock cycle as in the Berkeley RISC microprocessors [Pat82].

6.9.2. Multi-Way Branching

Another application of multi-level control is next-address calculation for multi-way branches. Two-way branches can be implemented as simple conditional branches in microcode; only one branch address field since the micro-program counter is incremented if the condition fails. Multi-way branches, however, require more than one branch address; in horizontal microprogramming, this can lead to excessively wide control words. Multi-way branches can be decomposed into a sequence of conditional branches at the cost of processing time, but some controllers may need to perform them directly.

Another way to implement multi-way branches in hardware is to use a mapping ROM or PLA as shown in Figure 6-5(b). If the algorithm specifies a CASE statement, the register containing the case selector variable can be connected to a PLA. The output of the PLA is gated to the micro-program counter through a multiplexer. The

PLA is designed to accept the values from the CASE statement, and to output the branch address for the associated microcode. The size of the PLA is determined by the number of possible branches, but the translation time is independent of the number of branches. The only required modification to the micro-control word format is the addition of the next-address multiplexer, and a condition-select field value which connects the output of the mapping PLA to the micro-program counter.

6.10. Storage of Literals

Another common use for the control memory is for storage of literals. Again, in horizontal microprogramming, the need to devote a large number of bits for rarely-used literals can be prohibitive. An indirect storage scheme, similar to indirect addressing described in the previous section, can be used to reduce memory size.

An auxiliary memory still requires interconnect to all data path cells which use the literals. The ideal method of reducing this interconnect length is to place the literal at the cell where it is used. In a register-MUX architecture, this can be accomplished very easily by adding another input to the required multiplexers. This input can have its lines connected to the chip's power supply and ground buses to give the binary value of the literal. This obviates the need for a literal field in the main memory, or for an auxiliary memory and the associated access time.

Indeed, the only reason for storing literals in a centralized memory is so that they may be easily modified. A FLK filter chip, for example, could store its coefficients in a small RAM. In the case of

hardwired literals connected directly to the cells where their values are used, this is not such a great issue. Hardwired literals require only connections to the power and ground supplies, so that modification of their value does not involve major circuit redesign. On the other hand, the interconnect requirement for a centralized literal memory can be very high, particularly if the literals represent constants requiring many bits of precision.

6.11. Distributed Control

The basic concepts of distributed control have already been described when discussing the promotion of procedures to independent tasks (section 6.5.3). In this case, the distribution of control is obtained explicitly from the high-level algorithm. If an Ada-like language is used to write the algorithm, the use of tasks directs the control synthesizer to create a distributed controller, even if it uses conventional software representation for subroutines.

The use of tasks or subroutines as the basis for splitting provides the easiest conceptual model for distributed control. This is shown in Figure 6-6. Even if the algorithm was written as one monolithic program, the synthesized controller would be a candidate for partitioning.

If the hardware compiler generates an actual layout or floorplan of the finished chip, such as MacPitts, geographic criteria can be used to partition the microcontroller. Partitioning has two main objectives:

- 1) to reduce the area used by the control memory.

Figure 6 - 6.

CONTROLLER PARTITIONING FOR MICROCODED SUBROUTINES

1) High - level Program

This shows a main program which calls two subroutines.

PROGRAM main

...
CALL sub1

...
CALL sub2

END:

2) Conventional Subroutines in Microcode

This diagram shows how a single controller with a stack for subroutines could implement the algorithm. This approach is the same as would be used in assembly language programming of subroutine calls.

3) PARTITIONED CONTROLLER

This diagram shows how the program could be used to create a distributed controller. There is one main controller, and a subcontroller for each subroutine in the program. Each sub-controller has its own program counter and memory. The subroutine call is implemented as a START signal from the main controller. The subcontroller then runs; it indicates its completion by sending a DONE signal back to its caller.

- 2) to reduce the interconnect between the control memory and the controlled hardware.

Partitioning can reduce control memory size by reducing its sparsity. Consider an algorithm which uses some data path resource in only one block of control steps. The control memory must contain all control fields in all control words. Many bits in the memory are wasted. The width of the main control memory can be reduced by extracting the block and placing it in a separate sub-controller. The main control memory requires only one bit to activate the subcontroller, which runs concurrently with the main controller until it is deactivated again.

Interconnect length is reduced if the subcontroller controls data path cells in its immediate vicinity. This is achieved automatically if the partitioning is based on the high-level subroutine hierarchy. When the partition is derived from the bit patterns in the control memory, this locality of control points is not assured. Hardware compilers which produce net lists of standard cells can direct the final floorplanning phase to place these cells close to the subcontroller, as shown in Figure 6-7. Compilers which produce the layout directly can use this information to select candidate control fields for partitioning. Alternatively, they can invoke another floorplanning pass after partitioning the controller.

The arrangement described above is straightforward as long as the subcontroller executes straight-line code with no branches or loops. If branches are required, the main and subordinate controllers must be synchronized. One method, though not the most memory-efficient, is to duplicate the branch addresses and condition-code in-

Figure 6 - 7.

EFFECT OF CONTROLLER PARTITIONING ON CHIP LAYOUT AND INTERCONNECT LENGTH

These diagrams show how controller partitioning affects interconnect length in the final layout of a chip. If one central controller is used, its control lines must reach all data path cells. By partitioning the controller, each subcontroller can be placed close to the data path cells which it controls. This permits shorter control lines between the controller and data path, although some additional interconnect is needed for inter-controller communications.

1) Single Controller

2) Distributed Controllers

formation in both controllers. The branch addresses would, of course, be adjusted for the memory of the subcontroller, which would contain fewer words, and hence its micro-program counter and branch address field would require fewer bits than the main controller.

Another problem occurs if the block contains a branch out of the block, which would imply a branch out of the subcontroller's memory. In this case, the main controller simply deactivates the subcontroller if the branch is taken.

CHAPTER 7

7. MiSyn - THE MICROCODE SYNTHESIZER

This chapter describes the MiSyn (Microcode Synthesizer) program which generates horizontal microcode for a circuit, given the control/data flow graph of the algorithm, the net list of hardware cells and their interconnections, and a cell library. The program was written in Common LISP on the 'Department of Electronics' VAX-11/750 minicomputer running the VMS operating system.

The complete program source code, together with sample input and output listings, is available as a Department of Electronics Internal Report [KaIP86].

7.1. Major Assumptions of the MiSyn Program

The MiSyn was written as a demonstration program; it was not intended to interface to any hardware computer under development. Consequently, several assumptions were made during its development. Chapter 8 contains a description of possible extensions which could overcome some of the limits of the current program.

7.1.1. Controller Design Style

MiSyn assumes that the controller uses a horizontal control word format with no indirect control (see Chapter 6). Figure 7-1 shows a schematic diagram of the assumed controller.

The controller consists of three modules: the control memory, a micro-program counter, and a condition-code select multiplexer. The control memory is split into three main fields:

Data Path Control: This provides the horizontal control word for all data path cells which require control lines. This portion

Figure 7 - 1. CONTROLLER ARCHITECTURE USED BY MSyn

The architecture of the controller which is assumed by MSyn is the same as the simple micro coded controller shown in Figure 6 - 1. This controller can perform conditional branches; a series of conditional branches is needed to perform a multi-way branch or CASE statement. This controller cannot perform subroutine calls. Note that the condition-code multiplexer has two predefined inputs corresponding to TRUE and FALSE. These are used to perform unconditional branches, and to ensure that the micro-program counter advances to the next sequential address (the no-branch case). The control word is stored in horizontal format with three main fields: data path control, condition code select (1 = unconditional branch, 0 = no branch), and the branch address

Control word format

of the control memory may optionally be compacted to reduce its width.

Condition-code Select: This field selects the input for the condition-code multiplexer. The output of this multiplexer determines if the micro-program counter is incremented or loaded with a branch address from the control memory.

Branch Address: The last field in the control memory provides a branch address for the micro-program counter. If the selected condition code is TRUE, the branch address is loaded into the micro-program counter; otherwise the micro-program counter is incremented to the next address.

The condition-code multiplexer has one-bit inputs from the data path for each condition which must be tested by the controller. Thus if the algorithm contains the following statement

IF $x = 1$ AND $y = 3$ THEN ...

the data path synthesizer would be responsible for providing the hardware to perform the comparison and logical AND operation in such a way that a single bit indicates whether the result is TRUE or FALSE. This provides conditional branch capability. The micro-program counter is incremented if the selected condition is FALSE; if the condition is TRUE, the next address is loaded from the control memory. The condition-code multiplexer also has two pre-defined inputs for boolean TRUE and FALSE to provide the unconditional branch and no-branch cases.

7.1.2. Unimplemented Microcontroller Features

The controller supports only a very basic instruction set consisting of sequential program execution, unconditional branching, and conditional branching. The controller does not directly support multi-way branches or subroutines. More sophisticated control structures, such as loops and multi-way branches, are constructed from the available operations.

The synthesizer supports only a centralized controller architecture. Neither distributed control nor microcode partitioning by functional or geographical criteria are supported.

7.2. MiSyn Overview

The MiSyn program consists of three main parts which transform the input information to the horizontal microcode listing. The organization of MiSyn is shown in Figure 7-2. These parts perform the following functions:

- 1) reduction of the control/data flow graph to a control graph.
- 2) derivation of symbolic microcode from the control graph.
- 3) generation of the control memory array from the symbolic microcode.

A fourth part can be used to compact the width of the control memory. Splitting the microcode generation into several passes with intermediate representations for the control information allowed incremental program development.

This is an original program which is similar to the control synthesizer described in [LoeL85]; this system also processes a functional and structural description of a circuit to produce either horizontal

Figure 7 - 2.

MISyn ORGANIZATION

This diagram shows the sequence of operations performed by the Microcode Synthesizer (MISyn). Its inputs consist of the high-level algorithm's Control / Data Flow Graph, and the circuit net list which would be generated by the data path synthesizer. The cell library is also needed to specify the control signals required by the data path hardware cells. The final output consists of a Control Memory Array representing the microcode for the circuit.

microcode or optimized microcode, PLA, or random logic controllers. Coincidentally, MiSyn is also very similar to the system described in [PenZ86]. This program uses a timed Petri net instead of the control/data flow graph, but this is a trivial difference as 'places' in the Petri net serve the same function as nodes in the control/data flow graph. Otherwise, the 'reduced microprogram graph' appears to be equivalent to MiSyn's control graph, and the controller architecture is the same for both programs.

7.2.1. Program Inputs

The MiSyn program follows the Elf philosophy of using control/data flow graphs as an intermediate representation of the algorithm to be realized in hardware. Scheduling of operations in the CDFG is indicated by control step partitioning. As MiSyn is intended to demonstrate microcode synthesis only, and as no program is available on the VAX to generate control/data flow graphs with control step partitioning, the CDFG is written and partitioned manually.

The circuit description used by MiSyn is a net list, sometimes called a circuit graph, which specifies the hardware cells and their interconnections. As with the CDFG, the net list is created manually for input to MiSyn. This process includes selection of cells from the library and binding of CDFG operations to hardware cells. Additionally, all required registers and multiplexers, which are not shown in the CDFG, must be added to the circuit.

The last required input is a cell library which defines the hardware used in the net list. MiSyn currently uses the control-related

information from a library of hypothetical cells which was available on the VAX at the start of program development.

7.2.2. Program Outputs

The program produces microcode in two forms. Firstly, a 'symbolic microcode' listing shows the control inputs to all cells used in the CDFG together with a mnemonic for the function being performed. The branch address and condition-code multiplexer control fields are also shown. This listing does not define the control for all cells in each control word; fields for cells not referred to in a control step are not shown in the corresponding control word.

This listing is used to generate a numeric representation of the control memory in the form of a two-dimensional array. Control fields requiring more than one bit may be shown as a single multi-bit field, or they may be split into the required number of one-bit wide fields, as determined by an argument to the LISP program. This part of the program also processes don't care control fields. The control memory may be optionally compacted using the 'fusion' method described in Chapter 6.

7.3. Format of the Control/Data Flow Graph

The control/data flow graph is a list of CDFG structures defined by the Common LISP DEFSTRUCT constructor.¹ This is analogous to a record in Pascal, and the term 'record' will be used in this chapter to refer to this type of data structure. Each CDFG node has a set of

¹ see [SteG84] for a complete description of the Common LISP programming language.

associated input and output edges which are numbered from left to right. These structures defined the following fields:

SLOT: This defines the control step in which the CDFG node is located. This field is simply an integer between 0 and the maximum number of control steps in the algorithm.

CLASS: CDFG nodes were assigned to one of several classes to control how MiSyn would process them. The classes are: OPERATION, CONSTANT, CONTROL.

SPECIFICS: This field provides more specific information about the node. Its precise interpretation depends on the value of the CLASS field. For OPERATION nodes, this field defines the operation type, which must correspond to the operations defined in the cell library. For CONSTANT nodes, this field gives the value and type (integer, real, etc.) of the constant. CONTROL nodes use this field to specify the type of control node, which may be one of: FORK, FORK-BEGIN, FORK-END, JOIN, LOOP-BEGIN, LOOP-EXIT, LOOP-END.

CHARACTERISTICS: This field is used by CONTROL nodes of the FORK and LOOP-EXIT subtype to specify the condition used to select the path emanating from the FORK, or the condition which must be true for the algorithm to exit the current loop block.

CELL: This field is used in OPERATION nodes to give the name of the cell from the net list which is used to perform the operation. Note that this is not the same as the type of cell from the cell library. If an algorithm uses two 7481-type adders called ADDER-1 and ALDER-2 in the net list, an ADD

operation node would specify, for example, ADDER-1 or ADDER-2 in this field. The fact that these are 7481-type adders is stored in the net list.

INPUT: This list defines the backward edges from the current CDFG node. Each item in this list is itself a list of the CDFG node and its output edge number. The sublists define all the input edges to the CDFG node. If a node has no inputs, such as CONSTANT nodes, this is an empty list.

OUTPUT: This is a list in the same format as the INPUT list, but it defines the output edges. This list is empty if a node has no output edges.

In order to represent input/output operations in the CDFG, the INPUT and OUTPUT lists may specify an EXTERNAL pseudo-node. The program creates one instance of an EXTERNAL CDFG node to ensure correct handling of references to inputs or outputs. This node exists simply to allow the graph traversal routine to treat all CDFG nodes in an INPUT or OUTPUT list uniformly. A separate LISP function examines each CDFG node visited during graph traversal to determine if the traversal should proceed. This function returns a 'stop' value when it encounters the EXTERNAL node.

As there is no universal convention for data flow graphs, the following sections describe the types of CDFG nodes recognized by MiSyn and their uses.

7.3.1. OPERATION Nodes

These nodes are used to describe all computations performed by the circuit. Each operation node must have an associated hardware

cell in the net list and a list of inputs and outputs. As described in the previous section, the CELL field of an OPERATION node is filled with the net list name of the hardware cell, not the name from the cell library.

The graph edges defined by the input and output lists have implicit registers at control step boundaries. If a cell or register has more than one source for an input, then a multiplexer must be attached to that input.

7.3.2. CONSTANT Nodes

These nodes are used to define data path constants in the CDFG. The current microcode synthesizer, however, assumes that literals will be hardwired to the inputs of any cells requiring them. Consequently, it ignores CONSTANT nodes, but they are available for future expansion. If literals were stored in either the main or auxiliary memory, these nodes would be used by the program to compile usage and bit-width statistics. They would assist in the synthesis of a literal field in the main control memory, or in defining the requirements for an auxiliary memory.

7.3.3. CONTROL Nodes

This covers a large number of node sub-types. These nodes do not correspond to hardware cells, but are used to direct the microcode generator. These nodes provide a means for expressing control flow in the CDFG. Figure 7-3 shows how these nodes are mapped into the control graph. The basic types of CONTROL nodes are:

Figure 7 - 3.

REPRESENTATION OF CONTROL NODES IN THE CONTROL GRAPH

These diagrams show how CDFG CONTROL nodes are represented in the Control Graph by MISyn. Solid lines show the flow of the microprogram into and out of each node. The dotted lines show that a control input in the form of a condition code is required from the data path.

a) FORK / FORK - BEGIN

There is one path emanating from the FORK node for each FORK - BEGIN node in the CDFG. Only one path can be taken by the controller. This is selected according to the condition codes.

b) FORK - END / JOIN

There is one path entering the JOIN node for each FORK - END node in the CDFG. The JOIN node performs no control operation; it acts as the next node for each last node in each path between the FORK and JOIN nodes.

c) LOOP - BEGIN

The loop begin node defines the start of a loop. It performs no control operations. This node acts as the destination of the unconditional jump at the corresponding LOOP - END node.

d) LOOP - EXIT

The loop exit node performs a conditional exit from its loop. This node must be located between LOOP - BEGIN and LOOP - END nodes. If the loop termination condition is false, then execution continues with the next sequential node.

e) LOOP - END

The LOOP - END terminates a loop. It is an unconditional jump to the corresponding LOOP - BEGIN node.

f) LOOP - EXIT&END

This is a 'composite node' which combines the characteristics of LOOP - EXIT and LOOP - END nodes. It is used to terminate a REPEAT loop. If the loop termination condition is true, then the program executes the next sequential node; otherwise it jumps to the corresponding LOOP - BEGIN node.

g) JOIN&LOOP - END

This node combines the characteristics of JOIN and LOOP - END nodes, used when a FORK / JOIN is nested inside a loop. This node unites the branches of the FORK, and jumps to the start of the enclosing loop.

FORK: This node starts a branch in the CDFG. Although the FORK may start a multi-way branch, the microcode is implemented as a sequence of simple conditional branches.

FORK-BEGIN: This node type was introduced for MiSyn's internal bookkeeping. There is one of these nodes at the start of each branch. The arcs emanating from this node are used to identify all nodes on that branch which otherwise would not have a parent CDFG node.

FORK-END: As the FORK-BEGIN node, this is used for internal bookkeeping by MiSyn. There must be one FORK-END node for each FORK-BEGIN node. All FORK-END nodes lead directly to a JOIN node.

JOIN: The JOIN node terminates a set of mutually exclusive branches in the CDFG. There must be one JOIN node for each FORK node.

LOOP-BEGIN: This node indicates that all CDFG nodes between the current control step and the matching LOOP-END node are to be executed in a loop. The only way to exit from a loop is through a LOOP-EXIT node.

LOOP-EXIT: This node provides a conditional exit out of the body of a loop. Unlike FORK nodes, the branch taken out of a LOOP-EXIT node is implicit. If the condition is false, flow continues through the node; if it is true, control is transferred to the control step following the LOOP-END node. If no LOOP-EXIT node occurs in the body of a loop, the loop will execute indefinitely. More than one LOOP-EXIT node may be placed in the body of a loop.

LOOP-END: This node is used to indicate the end of a loop body. Control is transferred to the control step containing the corresponding LOOP-BEGIN node.

In addition to the basic CONTROL nodes described above, several 'composite' nodes are created during the first phase of MiSyn when certain combinations of CONTROL nodes occur in the same control step. This combination is done at run-time by the microcode generator; they are not visible in the CDFG. The composite nodes are:

LOOP-EXIT&END: This composite node implements a Pascal-style REPEAT .. UNTIL loop where the loop termination test (LOOP-EXIT) occurs as the last action before the end of the loop.

JOIN&LOOP-END: This node combines a JOIN with a LOOP-END. It is necessitated by the special treatment of its component nodes by the microcode generator.

7.4. Phase 1 - Reduction of CDFG to Control Graph

The first step performed by MiSyn is to reduce the control/data flow graph into a control graph. The first part of MiSyn traverses the CDFG, creating control graph nodes as required. It was found during program development that creation of the CG edges was easier if postponed until all nodes were created. Consequently, this part of the program is itself broken down into two main parts: the CDFG traversal routine, and the routine to link the resulting control graph nodes with the correct edges.

The control graph contains one node for each control step in the CDFG. If the CDFG contains mutually exclusive branches due to

FORK/JOIN nodes, then the control graph will have a corresponding fork and join. While the CDFG shows parallel operations as multiple nodes in the same control step, the control graph associates these operations with the same control node. The relationship between the algorithmic description, the CDFG, and the control graph, is shown in Figure 7-4.

7.4.1. The Traversal of the Control/Data Flow Graph

The MiSyn program uses a breadth-first traversal algorithm to process the CDFG. This algorithm visits each CDFG node once in a way that all branches from a FORK are traversed completely before the traversal continues with the node immediately after the corresponding JOIN node. The pseudo-code for the traversal algorithm is:

- 1: Mark all CDFG nodes as being 'unvisited' and clear the pending-node stack.
- 2: Find the next unvisited CDFG node; if none exist then end.
- 3: If the current node has been visited or if it is an EXTERNAL node, then stop the traversal at this level of recursion; if it is a JOIN node, then push its successor nodes from its OUTPUT list onto the pending-node stack and stop the traversal at this level of recursion.
- 4: Process the current CDFG node.
- 5: Recursively apply the traversal algorithm from step 3 for each successor node in the OUTPUT list.

When processing the CDFG node, the traversal routine maintains path information to identify newly-created control graph nodes. This information is defined by the name of the FORK node and the branch number where the CDFG nodes for the control graph node are found.

Although the traversal is breadth-first in a global sense, each branch is traversed in a depth-first manner. The algorithm backtracks

Figure 7 - 4.

RELATIONSHIP BETWEEN ALGORITHM, CDFG, AND CONTROL GRAPH

1) THE ALGORITHM

```

IF d < 0 THEN
  d := d + inc1
ELSE
  y := y + 1
  d := d + inc2
END IF

```

2) THE CONTROL / DATA FLOW GRAPH

3) THE CONTROL GRAPH

The control graph reduces the information from the CDFG. There is only 1 node for each control step in the CDFG. The FORK / JOIN in the CDFG is shown as a branch in the control graph. As in the CDFG, branches in the control graph are mutually exclusive. The branch node (2) requires control INPUT from the data path. All other nodes provide control OUTPUT to the data path. The join node (6) does not perform control operations.

only when it encounters a node which meets one of the following criteria: if it has already been visited, if it is the EXTERNAL pseudo-node, or if it is a JOIN node. This implies that the traversal will proceed through several control steps before backtracking. Consequently, MiSyn must know whether to create a new control graph node or to update an existing one.

To accomplish this, each new control graph node is stored in a LISP vector (one-dimensional array) in which each element is treated as a push-down stack. The control step acts as an index into this vector; the stack-like usage of each vector element ensures that the correct control graph node will always be at the top. The use of a vector instead of a list frees the program from having to process CDG nodes in strictly sequential order of control steps. If the path information in the top control graph node matches the path being traversed, then the program simply updates the existing control graph node.

In addition to generation of control graph nodes, the CDG processing routine maintains data structures to keep track of loop start, exit, and entry points, and usage information for each hardware cell. A list of required microcode fields is generated at this time. Each hardware cell which requires control inputs has an associated data structure which stores the number of bits in the microcode field and the cell's usage information.

7.4.3. The Structure of the Control Graph

The control graph is implemented as a simple LISP list of node names. This list is used when simple traversal of the control graph is

being performed. The vector described in the previous section is a second data structure used for direct access to the control graph nodes. Each control graph node is a structure, defined by the LISP DEFSTRUCT, with the following fields:

TAG: The TAG field is used by the node linking program. As the CDFG is traversed, the path information for each new control graph node is placed in this field..

SLOT: This field gives the control step for the control graph node.

CLASS: This field classifies the control graph node using a subset of the CDFG node classes.

CONTROL-IN: The control input list specifies all data path status information which is used by the controller for conditional branching. This list is used to determine the number of inputs in the condition-code select multiplexer, and hence the number of bits in the control memory's condition-code select field. Each condition is assigned a binary code starting from 2; values 0 and 1 are predefined for the no-branch and unconditional branch conditions respectively.

CONTROL-OUT: The list of control outputs simply defines the active microcode fields in each control word.

Other fields are used to identify the CDFG nodes which created the control graph node, to associate a micro-control word with the CG node, and to represent the edges linking the CG nodes.

7.4.3. Construction of the Control Graph

The traversal of the control/data flow graph does not directly produce the control graph. Instead of attempting to generate all edges at the same time as the CG nodes, the traversal routine simply creates all required CG nodes, which are marked by the path information in their TAG fields, and stored in the CG vector ordered by their control steps. Once all CDFG nodes have been processed, the control graph nodes are linked by edges to form the actual control graph. This is done by filling in the input and output lists of each CG node.

The TAG field provides the main information for linking control graph nodes. Most nodes on the same path will have the same TAG value. The only exceptions occur at FORK and JOIN nodes, which are identified partially by their CLASS field, and by a different format of their TAGs which indicates what CDFG paths are starting (in the case of FORK nodes) or ending (in the case of JOIN nodes). Loops are linked with the aid of the auxiliary data structures generated during traversal of the CDFG. All edges are created in both directions, so that all nodes are in successor and predecessor relationships.

7.5. Phase 2 - Generation of Symbolic Microcode

The control graph shows the sequence of operations which must be performed by the controller. Since it is still a graph, it does not explicitly show the branching within the microcode. The second part of MiSyn transforms the control graph into an equivalent sequential microcode listing with conditional and unconditional branch instruc-

tions. This graph is related to the microcode by the fact that the operation of the controller can be viewed as a traversal of the control graph. The transformation is accomplished by traversing the control graph using the same algorithm used to traverse the CDFG.

This is the part of MiSyn which assigns control graph nodes to physical microcode addresses. Ordinary OPERATION nodes are assigned to consecutive addresses. branches in the control graph simply refer to CG nodes which are not in the next consecutive control step. To aid branch address generation, as each control graph node is assigned to a microcode address, this address is stored in the node's data structure. Backward branches can be resolved easily because the physical address is available in the target CG node's record.

Forward branch addresses, which are required by FORK and LOOP-EXIT nodes, cannot be determined as easily because the target CG node is not yet assigned to a microcode address. One method of handling forward references is to perform two passes through the control graph. The first pass would usually assign CG nodes to microcode addresses, while the second pass would actually process the CG nodes. MiSyn uses a 'backpatching' technique to handle forward references. When a forward reference is encountered during control graph processing, the name of the CG node where the reference occurs, and the name of the target CG node for which an address is required, are placed on a list of unresolved addresses. After all control graph nodes have been processed, MiSyn examines this list and fills in the branch address field of the CG node with the microcode address of the target. Unlike a conventional two-pass algorithm,

the second 'backpatching' pass considers only those CC nodes which contained unresolved references.

7.5.1. Transformation of FORK/JOIN Nodes

The control graph makes no distinction between conditional or multi-way branches. Any number of paths may emanate from a FORK node. The architecture of the controller, however, has been assumed to support only simple conditional branches. One of the jobs performed by this part of miSyn is the mapping of a multi-way branch to a series of conditional branches in the microcode.

This mapping cannot be performed directly because all branches in a FORK/JOIN pair are in the forward direction. The program must treat them as references which are resolved in the backpatching pass.

For each path emanating from a FORK node, the microcode will contain a block of control words which is reached by a conditional branch and which ends with an unconditional branch to the control word corresponding to the JOIN node. The exception is that the last block does not require an unconditional branch at its end to reach the JOIN control word because that is at the next sequential address in the control memory. The mapping of the FORK/JOIN construct from the control graph into microcode is shown in Figure 7-5.

7.5.2. Transformation of Loops

Loops do not present the same problems as multi-way branches. In many cases the entire algorithm is performed in an infinite loop. This is the easiest loop to implement in microcode because it requires only an unconditional branch to the start of the loop. Since the start

Figure 7 - 5.

MAPPING OF FORK / JOIN NODES INTO MICROCODE

This diagram shows how a multi-way branch, represented by the FORK / JOIN nodes in the Control graph is implemented in microcode. In MiSyn, this is done in the same way as in assembly language; the multi-way branch is decomposed into a series of simple conditional branches. The last branch is taken by default, although an explicit test followed by an unconditional 'jump to error routine' could be added.

1) The Control Graph

2) The Microcode

node is processed before the loop's end node, the destination branch address is known and can be placed in the microcode.

Other types of loops, which use data path conditions to exit the body of the loop, are characterized by forward branches due to LOOP-EXIT nodes and backward branches due to LOOP-END nodes. The backward branches can be processed directly, but as described above, forward branches must be processed during the backpatching pass. The relationship between loop constructs in the control graph and their realization in microcode is shown in Figure 7-6.

One situation which may occur in algorithms which use nested loops is that the conditional branch exiting from one loop may jump to the end of another loop. This situation is shown in Figure 7-7. If the branch target is an unconditional branch, then the net effect of these jumps is a conditional branch to the address specified by the second (unconditional) branch but executed in two clock cycles. The microcode generator recognizes these situations and modifies the branch address of the conditional jump to bypass the second unconditional jump. This preserves the algorithm, but it reduces its execution time by one clock cycle.

7.6. Phase 3 - Creation of Control Memory Array

The last part of MiSyn processes the symbolic microcode generated by Phase 2 to create a two-dimensional array representing the binary contents of the actual control memory. Most of this process is straightforward, requiring only that the data path control, condition code select, and branch address fields be copied from the symbolic microcode into the array.

Figure 7 - 6. IMPLEMENTATION OF LOOPS IN MICROCODE

a) FOR and WHILE Loops

These loops test for their termination before executing the loop body. The LOOP BEGIN defines the first micro - control word of the loop; it is followed by code to calculate the termination condition. If the loop terminates, the LOOP EXIT performs a conditional branch to the word after the LOOP END. If the loop does not terminate, the loop body is executed, followed by an unconditional branch to the start of the loop.

b) REPEAT Loops

The REPEAT loop differs from FOR and WHILE loops by testing the loop termination condition after the loop body. This loop requires only one conditional branch at the end of the loop. If the loop is not to be repeated, then execution continues with the operations after the branch.

Figure 7-7. ELIMINATION OF TRANSFER NODES IN LOOPS

Algorithms with nested loops can lead to Control Graphs with a redundant LOOP END node, as shown in the diagram at left. If the outer LOOP END does not perform any data path control, it acts only as a transfer node between the inner LOOP EXIT and the outer LOOP BEGIN nodes. It can then be removed by changing the branch address field in the inner LOOP EXIT node, as shown in the Control Graph at right.

Nodes 1 and 5 define the outer loop;
Nodes 2 and 4 define the inner loop

The LOOP EXIT at node 3 leads to node 5 because that is the node after the last node in the inner loop.

Node 5 is just an unconditional jump back to node 1 (the start of the outer loop). If this node does not perform any other operation, then the jump from node 3 can be changed to jump directly to node 1. This saves one clock cycle and microcode word without changing the algorithm.

The size of the control memory is easily determined from the number of words in the symbolic microcode, and from the number of bits required by each micro-control field. The number of bits required by the condition code select and branch address fields can be calculated from the number of inputs to the condition code multiplexer and from the number of micro-control words. The initial contents of the control memory are set to don't care. In the actual control memory, of course, all bits must be either 0 or 1, as a don't care value as such does not exist; it is, however, used by the optional microcode compaction phase. If compaction is not performed, don't care fields are set to 0.

This phase of MiSyn processes each control word from the symbolic microcode list. Each data path control field has its value copied into the control memory array. The control fields which require special consideration are those not explicitly specified in the control word. If possible, MiSyn keeps the field as a don't care to improve chances of merging it with other fields during microcode compaction.

7.6.1. Processing of Unspecified Control Fields

If the unspecified field controls a cell which does not latch its input (a select signal, see Chapter 6), its behaviour during this control word will not affect the algorithm so its field in the control memory is set to don't care. This is the action performed for all multiplexer control fields, with the exception of the condition code select field.

When the field controls a register or other cell which stores its input (evoke signal), the controller must ensure that the latched

value does not change in such a way that it would affect the algorithm. This is why the first phase of MiSyn build a list recording when the cell's input is latched, and when its value is used by another cell. These lists of usage information, which will be referred to as the node trace, are used by Phase 3 to fill in the control fields for unspecified evoke signals such as registers. Another example is a random number generator, which should not be modified unless the circuit is explicitly initialized.

Loops introduce a complication to the node trace. Figure 7-8 shows a situation when the node trace does not provide complete information about a cell's usage. In this example, a register latches its input before the start of a loop at time (A), and again near its end at time (C). The contents of the register are used shortly after the start of the loop at time (B). The node trace correctly shows that the register's contents must be preserved in the interval from (A) to (B). The node trace indicates that the register is not used as an input to another cell after time (B). This is not correct because of the loop; the register's value must be preserved during the loop.

MiSyn's strategy for handling unspecified evoke signals in loops is to propagate the field's value from the start of the loop body until the node trace can be used reliably. For the example given above, a value of 1 in the register's control field will latch the current input, and a value of 0 will hold its contents unchanged.

By using the register's node trace for the interval from (A) to (B), MiSyn determines that its control field must be 0 to hold its contents until they are used at (B). By checking the loop information generated by Phase 1, MiSyn knows that it is now in a loop body.

Figure 7 - 8.

EFFECT OF LOOPS ON NODE TRACE

The 'Node Trace' is maintained by MTSyn during traversal of the Control / Data Flow Graph. The Node Trace records when each operation node is used as an input to another operation, and when it receives a control signal from the controller. The Node Trace is currently used to determine when control inputs to registers can be set to Don't Care without affecting the algorithm.

In sequential microcode, a register's control field can be Don't Care if the register is not used as an input to another cell before it next latches a new value. As long as the register must hold its value, its control field cannot be Don't Care.

This reasoning fails if the Node Trace does not account for loops in the microcode. In this case, the register's contents may need to be preserved from the end of a loop back to the start of the loop. An example of this situation is shown in the diagram below.

In this example, register R1 latches its input at time 'A' in the Control / Data Flow Graph. This is before the start of a loop.

The same register is used as an input to an addition operation at 'B'. If the register does not latch a new value immediately prior to B, then the Node Trace correctly determines that R1's control field must not be Don't Care in the interval from A to B.

R1 latches its input again at 'C', before the end of the loop. If R1 is not used again, the Node Trace will incorrectly assume that R1's control field may be Don't Care after C. In this case, however, the value from C must be preserved for the next loop iteration.

MTSyn handles this problem by checking if the control field can be Don't Care at the start of the loop. If not, the control field is not set to Don't Care until after the end of the loop. Loop nesting is also taken into consideration.

The register latches a new value at (C), which is within the loop body, so its control field can be set to don't care in the interval from (B) to (C). After (C), however, the node trace indicates that the register's contents are not used. The end of the loop occurs in this last time interval, so MiSyn examines the register's control field at the start of the loop (0) and propagates this value from (C) to the end of the loop. The control field from the end of the loop to the end of the microprogram is set to don't care.

7.7. Phase 4 - Microcode Compaction

The last, and optional, part of MiSyn is a microcode compaction routine which uses the 'fusion' technique described in Chapter 6. Briefly, this technique merges columns in the control memory if the value of the columns is equal for all control words. A don't care is considered to be equal to any other value, so a large proportion of don't care fields increases the probability of combining columns.

7.7.1. The Compatibility Matrix

The 'fusion' algorithm begins, as most other microcode compaction methods [AndM80], with a matrix which defines the pairwise compatibilities of all microcode fields. This matrix must be constructed by an exhaustive search of the control memory. To construct the compatibility matrix, the multi-bit fields are reduced to several fields of one bit each. MiSyn does not compact the condition code select and branch address fields. All fields to be examined by the compaction algorithm are stored in a LISP list.

This algorithm assumes that all microcode fields are compatible. It then performs an exhaustive search of the control memory in search of conflicts between the pairs. If incompatible fields are found, the appropriate matrix elements are updated and the algorithm moves on to the next pair of fields. The pseudocode for this algorithm is:

n = the total number of microcode fields being considered for compaction.

- 1: Create an empty compatibility matrix; this is a square two-dimensional array with rank equal to the length of the field-names list; initialize the matrix to the value TRUE.
- 2: For each microcode field f_i ($i = 1, 2, \dots, n$) in the field-names list do:
- 3: For each microcode field f_j ($j = i + 1, \dots, n$) in the field-names list do:
- 4: For each micro-control word in the control memory do:
- 5: If $f_i \neq f_j$ and one of f_i or f_j is not a don't care, then the (i, j) position in the compatibility matrix is set FALSE; exit the inner loop; resume algorithm at step 3.

This algorithm must compare all pairs of microcode fields, so its complexity is $O(n^2)$ where 'n' is the number of fields. It must scan each word in the control memory, but only if the fields are compatible; if they are incompatible the search ends and the next pair of fields is tested. Once the compatibility matrix has been constructed, MSyn uses it to group compatible microcode fields.

7.7.2. The Compaction Algorithm

The aim of the compaction algorithm is to find groups of microcode fields which are fully pairwise compatible. Each group of fields, which is called a compatibility class, may be represented by a single

column in the control memory. The output of each column will be connected to several hardware cells. There are several microcode compaction algorithms, such as bit steering [kao85] or maximal compatibility classes [And80]. The definition of compatibility used by these algorithms requires the microcode fields to control mutually exclusive operations; this is not required by MiSyn.

The compaction algorithm in MiSyn uses a simple method to create the compatibility classes. While creating the compatibility matrix, MiSyn counts the number of conflicts for each field. The list of microcode fields is then sorted by decreasing number of conflicts. Each microcode field in this list is tested for pairwise compatibility with all fields in each existing compatibility class. A new class is created if the field cannot be included in one of these classes. A pseudo-code description of the algorithm follows:

m = the total number of compatibility classes currently existing (this value increases as the algorithm runs).
 n = the total number of microcode fields.
 p = the total number of microcode fields grouped in a given compatibility class.

- 1: Initialize the list of compatibility classes to an empty list.
- 2: Sort the list of microcode fields by decreasing number of conflicts.
- 3: For each field f_i ($i = 1 \dots n$) in the sorted list do:
- 4: For each compatibility class c_j ($j = 1 \dots m$) do:
- 5: For each microcode field f_k ($k = 1 \dots p$) in c_j do:
- 6: If the compatibility matrix at (f_i, f_k) is FALSE, field f_i cannot be added to c_j ; continue algorithm with the next value of j at step 4.
- 7: If f_i is compatible with all fields in c_j , add f_i to c_j ; continue algorithm with next value of i at step 3.
- 8: If f_i is not compatible with any c_j , create a new compatibility class; place f_i in the new class; Continue algorithm with next value of i at step 3.

by starting with the 'most incompatible' microcode field, the algorithm attempts to reduce the total number of compatibility classes. Typically, the fields with the greatest number of conflicts become the seeds of several compatibility classes. As the algorithm moves to fields with fewer conflicts, these fields are placed in one of the existing classes. Fields are placed in compatibility classes on a 'first fit' basis. MiSyn also records the compatibility class to which each field has been added. Since the algorithm does not evaluate all possible groupings, it does not guarantee an minimal solution. Pairwise interchange of fields might change the distribution of fields among the classes, but note that the number of fields combined into one class is not important. The classes are not encoded, so only the total number of compatibility classes affects the reduction of the control memory's bit² dimension. The current algorithm is straightforward and serves to illustrate how compaction can reduce the size of the control memory.

7.7.3. Control Field Values After Compaction

The compaction algorithm creates a list of compatibility classes. Each class is itself a list of microcode fields belonging to the class. The last step in this part of MiSyn uses this information to make a new control memory array.

The control bits for the compatibility class are taken from a three-valued OR of the bits of each component field. The three-valued OR simply takes into account the fact that some fields will be don't care; the truth table is shown with don't care denoted by 'X':

	0	1	X
0	0	-	0
1	-	1	1
X	0	1	X

Note that it is impossible for two fields in the same compatibility class to have values of 0 and 1, as indicated by '-'.

Once the compatibility classes have been created, MiSyn makes a new symbolic microcode listing in the same format as in Phase 2. Each field in the uncompact control word is replaced by its compatibility class, and its value is replaced by the value computed by the three-valued Ok function. This is repeated for each micro-control word. A pseudo-code description of this algorithm follows:

- m = the number of control fields in a given micro-control word.
- n = the total number of micro-control words in the control memory.
- 1: For each micro-control word w_i ($i = 1 \dots n$) do:
- 2: Create a new micro-control word w_i ; copy the condition code select and branch address fields from w_i to w_i .
- 3: For each control field f_j ($j = 1 \dots m$) in w_i do:
- 4: If the compatibility class c_j for f_j has more than one element create a new control field for c_j in w_i , otherwise copy f_j into w_i and continue at step 3.
- 5: Find each control field from c_j which is also in w_i ; if one of these fields is not don't care, copy that value into w_i ; Remove each of these fields from w_i so that they are not processed again.

The compacted symbolic microcode is used to make a compacted control memory array. The routine from Phase 3 performs this job exactly as before. Since don't care fields are of no significance in the compacted memory, they are replaced by zeroes.

7.8. Results

The MiSyn program has been tested with several input files. A control/data flow graph for implementing Bresenham's circle drawing algorithm [FolJ84], containing 46 CDFG nodes, created a control memory containing 16 words of 25 bits. The control graph was generated in 58 seconds of CPU time; the control memory was generated and compacted to a width of 17 bits in 78 seconds of CPU time.

A second example, shown in the Internal Report, used a CDFG with 67 nodes for implementing the Shell sort algorithm. The control graph was generated in 84 seconds of CPU time. The 23 word control memory was generated and compacted from 41 bits to 23 bits in 240 seconds of CPU time. The CDFG and control memories for this example are included in Appendix A of this thesis.

7.9. MiSyn Summary

The MiSyn program generates microcode for a simple controller with a subset of the features described in Chapter 6. The main bottleneck in the program is the manual generation of the inputs; currently there is no program on the VAX to convert a high-level program with control structures such as IF-THEN-ELSE to a control/data flow graph.

The program generates horizontal microcode which may be compacted to reduce control memory area. It can handle conditional and unconditional branches; these are used to perform loops and multi-way branches. The next chapter describes how MiSyn could be enhanced for more sophisticated microcontroller architectures.

CHAPTER 8

8. FUTURE WORK AND CONCLUSIONS

This chapter examines some possible future extensions to MiSyn which would enhance its functionality. As indicated in Chapter 7, MiSyn is a demonstration program which does not attempt to include all features of an ideal control synthesizer.

The choice of LISP as the implementation language leads to slower execution and high run-time overhead, but the interactive environment and debugging features are worth the penalty for experimental programs.

Re-writing MiSyn in a more efficient compiled language such as C would improve its performance, particularly if the number of intermediate steps was reduced. This would be practical only if MiSyn was to be used as part of a real hardware compiler.

8.1. Enhanced Controller Architectures

In its present form, MiSyn generates microcode for a simple and fixed architecture. Support of different architectures, to suit the algorithm, would be a desirable feature.

It is unclear whether architectural decisions should be performed by the data or control synthesizers, or whether a special style selector should run before the synthesis programs. Style selection seems to be a task best suited to rule-based expert systems techniques. Many hardware compilers use conventional algorithmic programming although Synapse and HAL do use artificial intelligence techniques. An expert system seems to be overkill for microcode synthesis because the problem can be solved by a graph-traversal algorithm.

8.1.1. Extensions for Subroutines

One essential feature which should be added to ViSyn is the capability to perform subroutines. In the case of conventional subroutines, this requires the addition of a stack for return addresses. The controller also requires control over this stack, so that new control fields must be added to its control word format. Figure 8-1 shows one possible architecture for conventional microcoded subroutines.

As this controller contains a stack for return addresses, the control synthesizer must determine the stack depth. Commercial microcontrollers, such as the AM-2900 series, contain fixed hardware stacks only 4 or 5 levels deep [Mye80]. It is unreasonable to limit the user to an arbitrary nesting limit in the high-level algorithm; this is not natural in conventional software programming languages.

The control/data flow graph can be analyzed to find the maximum subroutine depth. This value would determine the depth of the controller's stack to ensure that the user need not worry about stack overflow during subroutine calls. Any changes to the structure of the high-level program could, of course, change the depth of the subroutine stack. This does not allow recursive programming as the stack depth could not be determined by analyzing the CDFG. Only the special case of tail recursion, which can be transformed to iteration, could be performed in hardware.

The control memory would require additional fields for next address selection. In the controller used currently, the only possible sources for the next address are the branch address from the control memory, and the incremented value of the micro-program counter. Subroutine returns require the next address to be taken from the

Figure 8 - 1.

MICROCONTROLLER WITH STACK FOR SUBROUTINES

This diagram shows one possible arrangement of a microcoded controller with a stack for subroutines. The stack is implemented as a shift register capable of three operations: PUSH, POP, and NO - OPERATION, as specified by the STACK CONTROL field in the control memory. The ADDRESS SELECT field determines the source of the next address. It can be either the branch address, the top of the stack, or the next sequential address. The CONDITION SELECT and ADDRESS SELECT fields work together to determine the next address. This allows conditional POPs, if required. The interpretation of these control fields is shown in the truth table below.

ADDRESS SELECT	CONDITION CODE	NEXT ADDRESS
PC + 1	0	PC + 1
	1	
BRANCH	0	PC + 1
	1	BRANCH ADDRESS
STACK	0	PC + 1
	1	TOP OF STACK

stack. The control memory must also include fields to control the operation of the stack. The microprogram must be able to push the microprogram counter on the stack, and to pop the top element from the stack. Note that a 2-bit control field is required, as a no-operation state must be included to preserve the stack when it is not being used.

The stack itself can be implemented in the conventional software way as a small memory with a stack pointer register. An alternative approach is a shift register where the direction of the shift determines whether a push and pop is performed.

8.1.2. Distributed Control

Distributed control is the main alternative to conventional microcoded subroutines. This has some advantages for control synthesis because the subcontrollers do not need stacks for subroutines. Synchronization between controllers requires two control lines: a START signal from the caller to the subcontroller, and a DONE signal from the subcontroller back to the caller. Figure 8-2 shows the interconnections and how they would be used by the controller.

The subroutine call would consist of raising the START line to the subcontroller. If a controller calls several subroutines, then a decoder must be used to select the correct START line. Similarly, a multiplexer is needed to select the corresponding DONE line.

Raising the START line enables the subcontroller's microprogram counter and resets its DONE flip-flop. The subcontroller performs the operations in its own control memory. During this time, the caller can continue executing the main program until it needs data

Figure 8 - 2.

DISTRIBUTED CONTROLLER

This diagram shows some of the internal details of a distributed controller which uses a START / DONE communication protocol. In this example, each controller corresponds to a software subroutine.

As each subroutine may have many callers, all the START signals must be connected to an arbitration / queuing circuit. This circuit must resolve which caller is serviced if more than one START request arrives at the same time, and to queue the callers which are waiting for service.

The DONE flip-flop is set when the controller has completed the subroutine; this signals the caller to continue execution if it has been waiting for data. It also signals the arbitrator that another call may be serviced.

Note that steps must be taken to ensure that the data for one caller is not overwritten by the execution of the subroutine for another caller.

from the subroutine. It then checks if the DONE flip-flop has been set by the subcontroller to indicate that the subroutine has completed. If the DONE flip-flop has not been set, the caller must wait until the subroutine finishes. The DONE signal must be latched because the subroutine may finish before the caller needs to synchronize itself.

As with any concurrent system, precautions must be taken to avoid deadlock. A simple START/DONE protocol would run into trouble if the same subroutine was called by a second caller before it finished its first call. Callers must be queued, with the effect that implementing the subroutine call becomes a question of implementing an Ada rendezvous in hardware.

8.1.3. Multi-Way Branching

Another useful enhancement to the controller would be the capability to directly execute multi-way branches. MiSyn decomposes multi-way branches to a series of conditional branches. Although this approach is used by many assembly languages, it does not take full advantage of the capabilities offered by custom hardware.

A schematic diagram of a controller with a PLA for branch address generation was shown in figure 6-5(b). The PLA requires condition codes from the data path and the micro-program counter as inputs. The micro-program counter is needed as an input to this PLA to define the controller's current state.

This enhancement does not require additional control fields in the control memory. In fact, the branch address field could be removed from the control memory since all branch addresses, not just multi-way branches, could be obtained from the auxiliary PLA. MiSyn

would have to use the conditional branching information and the branch address to define the next-address PLA. This would be input to a normal PLA generator during the layout generation.

8.2. Alternate Control Word Formats

Adding more fields to a horizontal control word unnecessarily increases control memory width, particularly if the exotic functions such as subroutines are used only rarely in a microprogram. An opcode field controlling the interpretation of the rest of the control word would allow special-purpose fields to be stored in the same bit positions as data path control fields. This control word format is shown in Figure 8-3.

As long as the microprogram spent most of its time performing useful data path operations, the penalty of the additional control words would be offset by the reduction in word width. There would also be a slight penalty in execution speed, because all control operations would require an additional clock cycle.

8.3. Microcode Partitioning

The benefits of distributed control for large, modular systems were discussed in Chapters 5 and 6. Partitioning the control memory without high-level guidance is a more difficult problem. The microcode generator would need to analyze the bit patterns in the control memory to find a partition of microcode fields for the greatest area reduction.

rather than duplicating branch addresses in each subcontroller, as suggested in section 6.11, the chip may use a centralized next-

Figure 8 - 3.

VARIABLE CONTROL WORD FORMATS

This diagram shows a control word format which uses an opcode field to determine the interpretation of the remaining bits. The OPCODE field controls decoders attached to the output of the remainder of the control word. This selects the destinations of the control bits, thereby determining if they are used as data path control, condition select, or branch address literals. This adds considerable decoding logic to the output of the control memory, but it has the advantage of reducing the width of the control memory. A fully - horizontal control word would store the data path control, condition code select, branch address, stack control, and any other control fields, all in one very wide word. If some fields, such as stack control, are used rarely, then much of the memory would be wasted. The variable control word format reduces width without affecting data parallelism. The number of words in the memory is increased because data and control path operations cannot occur in parallel.

OPCODE

REMAINDER OF CONTROL WORD

DATA PATH CONTROL	HORIZONTAL CONTROL BITS
----------------------	-------------------------

CONDITIONAL JUMP	CONDITION SELECT	BRANCH ADDRESS	NOT USED
---------------------	---------------------	----------------	----------

SUBROUTINE CALL	NOT USED	BRANCH ADDRESS	NOT USED
--------------------	----------	----------------	----------

SUBROUTINE RETURN	NOT USED
----------------------	----------

address generator to determine a branch address from the data path flags and a micro-program counter. This circuit would be connected to the micro-program counters of all subcontrollers; whenever a branch is to be taken, the micro-program counters would be loaded from the next-address generator.

The advantage of this approach is that the branch address information would not need to be duplicated in each subcontroller. The cost is the size of the next-address generator, but this should not be much larger than the next-address circuitry of a subcontroller; a more significant penalty would come from the interconnect needed between the next-address generator and the subcontrollers. A 1K-word control memory would need a 10-bit branch address, and a 10-bit wide bus between the subcontrollers.

8.4. Integrated Simulation

As an aid to design verification, the control synthesizer could include a behavioural simulator. This would be an ideal application of an object-oriented extension to LISP. Each hardware cell in the net list could be represented by an object. The controller would send messages corresponding to control signals to each data path object. The state of the circuit would be represented by the state of each object. This type of simulation would quickly verify the operation of the circuit and controller, and changes to the microcode could be checked easily.

A complete hardware compiler should, of course, include facilities for detailed electrical simulation. Nevertheless, a fast behavioural simulator would be useful for systems-level simulation during the cir-

cuit design phase when the compiler is being used as a designer's assistant. Although hardware compilers provide circuits which are 'correct by construction', there is always the danger that the high-level input does not correctly describe the user's intentions.

8.5. Summary

This thesis has shown how microcode can be generated automatically from a high-level description of an algorithm and from a net list of the circuit to perform the data processing. The microcode could be used by a controller to form a complete data and control path for computer synthesized integrated circuits.

An overview of hardware description languages was provided to show how hardware can be described at the behavioural level, as opposed to the geometric or structural level used by common VLSI design tools. A good hardware description language allows the designer to express both the high-level behaviour of a circuit and its low-level details in a consistent manner. The high-level description is a program to be performed by the circuit; this hides implementation details from the user so that a circuit, although not necessarily an optimal one, can be created by designers with little hardware experience.

A review of hardware compilation showed how high-level algorithms can be used to synthesize the data path of special-purpose circuits. This introduced control/data flow graphs as an intermediate representation used by hardware compilers. Flow graph optimization techniques, such as graph productions and load balancing were described because the form of the control/data flow graph determines the final microcode.

This lead to a discussion of the issues in controller design for application-specific integrated circuits. The question of how the design task should be split between data and control synthesizers was resolved in favour of the data path. Whenever the input to the hardware compiler can be handled in either the control or data paths, such as loop counters, the data path synthesizer has a better chance of making full use of the hardware needed to realize the operation.

Focusing on microcoded controllers, expression of high-level constructs such as subroutines, loops, and multi-way branches in hardware were explored. The choice of control word formats and use of compaction to reduce memory area was examined, leading to the selection of horizontal microcode for maximal data path parallelism.

Concerns about memory size and control line length led to the consideration of controller partitioning. This was found to follow naturally from a modular program, so that each subroutine could be implemented as an independent subcircuit with its own controller. Argument passing between controllers requires synchronization and communication protocols similar to the Ada tasking model.

A demonstration program, MiSyn, was written to show how microcode can be generated from a control/data flow graph and circuit net list. This program generates horizontal microcode for a simple controller with conditional and unconditional branching capability. The control/data flow graph is reduced to a control graph which contains all control information for each control step, but not assigned to definite addresses in the control memory.

During construction of the control graph, MiSyn maintains a history record for each cell in the net list, recording when each cell

receives a control input, and when it is used as an input to another cell. This record is important for determining when control fields may be treated as don't care.

The control graph is then mapped into microcode by assigning each graph node to an address; this also defines the branch addresses. Misyn then creates a control memory array from the microcode, using the history records to set don't care fields where possible. This horizontal control memory may be compacted by searching for sets of columns which are identical in each control word. These columns are then combined into a single column in the control memory. The presence of don't cares increases the chance of merging columns.

This program was written in Common LISP. It was developed to where it could generate microcode for simple high-level programs containing nested loop and branch control structures. Multi-way branches are decomposed into a series of conditional branches.

8.6. Conclusions

Increasing circuit complexity and demand for custom integrated circuits shows a need for VLSI design tools more sophisticated than the current layout programs.

Hardware compilation aims to perform the detailed circuit design from a high-level behavioural description. This would relieve the designer from low-level details, allowing them to concentrate on the high-level design. In the short term, hardware compilers could be used as designer's assistants, allowing users to quickly compare different realizations for a circuit before selecting one for optimization and final production.

A microcoded controller with horizontal control words has advantages of relatively easy synthesis as well as easy user modification of the control memory. Large control memories can be compacted, but a better long term solution is to partition the controller into distributed controllers which can be placed closer to the data path cells they control. In addition to reducing total control memory area, partitioning will reduce area used by the control lines.

Hardware compilation is not a panacea, nor is it a pipe dream. Extensive research in this field shows that it is a viable concept in need of refinement and user acceptance. The state of the art, as of 1986, appears to be at the same stage as software compilation was in the 1950's. The road to hardware compilers capable of matching experienced human designers is still long. It seems likely, however, that sometime in the future hardware compilation will be as common as high-level programming languages are today.

APPENDIX A

REFERENCES

- [AckB85] B. D. Ackland et al., "CADRE - A System of Cooperating VLSI Design Experts", Proc. IEEE International Conference on Computer Design: VLSI in Computers, Oct. 1985, pp. 99-104.
- [AhoA86] A. V. Aho et al., "Compilers, principles, techniques, and tools.", Addison-Wesley, Reading, Mass., 1986.
- [AndM80] M. Andrews, "Principles of Firmware Engineering in Microprogram Control", Computer Science Press, Rockville, Md., 1980.
- [Ay1J86] J. H. Aylor et al., "VHDL - Feature Description and Analysis", IEEE Design & Test, April 1986, pp. 17-27.
- [AyrR79] R. Ayres, "Silicon Compilation - A Hierarchical Use of BLAs", Proc. 16th Design Automation Conference, 1979, pp. 315-315.
- [Bark86] K. Bartlett et al., "Synthesis and Optimization of Multilevel Logic under Timing Constraints", IEEE Transactions on Computer Design, Vol. CAD-5, No. 4, October 1986, pp. 582-596.
- [BarM85] M. Bartholomeus et al., "PLASCO: A Procedural Silicon Compiler for PLA Based Systems", Proc. IEEE 1985 Custom Integrated Circuits Conference, pp. 226-229.
- [BisN85] N. N. Biswas, "On Bit Steering in the Minimization of the Control Memory of Microprogrammed Processors", IEEE Trans. Comp., Vol. C-34, No. 11, Nov. 1985, pp. 1057-1061.
- [BraR84] R. K. Brayton and C. McMullen, "Minimization of Multistage Logic", IEEE 1984 International Conference on Computer Design, Port Chester N. Y.
- [BurM83] M. R. Buric et al., "The Plex Project: VLSI Layouts of Microcomputers Generated by a Computer Program", Proc. IEEE 1983 International Conference on Computer-Aided Design, Sept. 1983, pp. 49-50.
- [DeMG83a] G. De Micheli and M. Santomauro, "Topological Partitioning of Programmable Logic Arrays", Proc. IEEE 1983 International Conference on Computer-Aided Design, pp. 182-189.
- [DeMG83] G. De Micheli et al., "Computer-Aided Synthesis of PLA-Based Finite State Machines", Proc. IEEE 1983 International Conference on Computer-Aided Design, pp. 154-156.
- [DewA86] A. Dewey and A. Gadiant, "VHDL Motivation", IEEE Design & Test, April 1986, pp. 12-16.
- [FolJ84] J. D. Foley and A. van Dam, "Fundamentals of Interactive Computer Graphics", The Systems Programming Series, Addison-Wesley, Reading, Mass., 1984.

- [GajD83] D. D. Gajski, "New VLSI Tools", COMPUTER, Vol. 16, No. 12, December 1983, pp. 11-14.
- [GerC86] C. M. Gervehsi, "Comparisons of CMOS PLA and Polycell Representations of Control Logic", Proc. 23rd Design Automation Conference, 1986, pp. 638-642.
- [GerS85] S. M. German and K. J. Leiberherr, "Zeus: A Language for Expressing Algorithms in Hardware", COMPUTER, Vol. 18, No. 2, February 1985, pp. 55-65.
- [GirE84] E. F. Girczyc, "Automatic Generation of Microsequenced Data Paths to Realize Ada Circuit Descriptions", PhD. Thesis, Department of Electronics, Carleton University, 1984.
- [HilF78] F. J. Hill and G. R. Peterson, "Digital Systems: Hardware Organization and Design, 2nd Edition", John Wiley & Sons, 1978.
- [JacR85] R. J. A. Jacob, "A State Transition Diagram Language for Visual Programming", COMPUTER, Vol. 18, No. 8, August 1985, pp. 51-59.
- [JinB85] B. W. Jinks et al., "Automated Generation of Microcontrollers", Proc. IEEE 1985 International Conference on Computer-Aided Design, Sept. 1985, pp. 335-337.
- [JohD79] D. Johannsen, "Bristle Blocks: A Silicon Compiler", Proc. 16th Design Automation Conference, June 1979, pp. 310-313.
- [KaeT86] T. Kaehlen and D. Patterson, "A Small Taste of Smalltalk", BYTE, Vol. 11, No. 8, August 1986, pp. 145-159.
- [KalP86] P. Kalab, "HiSyn - The Microcode Synthesizer", Internal Report, Department of Electronics, Faculty of Engineering, Carleton University, December 1986.
- [KowT85a] T. J. Kowalski et al., "The VLSI Design Automation Assistant: From Algorithms to Silicon", IEEE Design & Test, August 1985, pp. 33-42.
- [KowT85b] T. J. Kowalski, "An Artificial Intelligence Approach to VLSI Design", Kulwer Academic Publishers, 1985.
- [Kri84] M. Krieger and P. Piche, "Segmented Microprogramming: A Technique for Designing Special Purpose VLSI Processors", IEEE 1984 International Conference on Computer Design: VLSI in Computers, Port Chester N. Y., pp. 393-396.
- [LoeL85] L. Loeffler et al., "Processor Control Part Synthesis Tool Yields Short Turnaround Times by High Level Description", IEEE 1985 International Conference on Computer Design, pp. 266-268.

- [MarP84] P. Marwedel, "The MIMOLA Design System: Tools for the Design of Digital Processors", Proc. 21st Design Automation Conference, 1984, pp. 587-593.
- [MarP86] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System", Proc. 23rd Design Automation Conference, 1986, pp. 271-277.
- [MatA81] A. Mathialagan and N. N. Biswas, "Bit Steering in the Minimization of Control Memory in Microprogrammed Digital Computers", IEEE Transactions on Computers, Vol. C-30, No. 2, Feb. 1981, pp. 144-147.
- [MeadC80] C. Mead and L. Conway, "Introduction to VLSI Systems", Addison-Wesley, Reading, Mass., 1980.
- [MyeC80] G. J. Myers, "Digital System Design With LSI Bit-Slice Logic", John Wiley & Sons, 1980.
- [NagA81a] A. W. Nagle, "Automated Design of Digital System Control Sequencers from Register-Transfer Specifications", PhD. Thesis, EE Dept., Carnegie-Mellon University, 1981.
- [NagA81b] A. W. Nagle, "Algorithms for Multiple-Criterion Design of Microprogrammed Control Hardware", Proc. 18th Design Automation Conference, 1981, pp. 486-493.
- [ObreM83] M. Obrebska, "Algorithm Transformations Improving Control Part Implementation", Proc. IEEE 1983 International Conference on Computer-Aided Design, pp. 157-158.
- [PapC85] C. Papachristou, "PLA Compaction by Partition and Fusion", Proc. IEEE 1985 International Conference on Computer-Aided Design, pp. 166-168.
- [ParA84] A. C. Parker, "Automated Synthesis of Digital Systems", IEEE Design & Test, Vol. 1, No. 4, November 1984, pp. 75-81.
- [PatD82] D. A. Patterson and C. H. Sequin, "A VLSI RISC", COMPUTER, Vol. 15, No. 9, September 1982, pp. 8-21.
- [PauP86] P. G. Paulin et al., "HAL - A Multi-Paradigm Approach to Automatic Data Path Synthesis", Proc. 23rd Design Automation Conference, 1986, pp. 263-270.
- [PenZ86] Z. Peng and K. Kuchinski, "Synthesis of Control Structures from Petri Net Descriptions", Microprocessing and Microprogramming 18 (1986), pp. 335-340.
- [PilR85] R. Piloty and D. Borriane, "The Conlan Project: Concepts, Implementations, and Applications", COMPUTER, Vol. 18, No. 2, February 1985, pp. 81-92.

- [RaeG85] G. Raeder, "A Survey of Current Graphical Programming Techniques", COMPUTER, Vol. 18, No. 8, August 1985, pp. 11-25.
- [KaoS85] S. S. S. P. Rao and J. R. Isaac, "Micro-Store Width Optimization Through Bit Steering", Microprocessors and Microprogramming, Vol. 16, 1985, pp. 17-30.
- [SchM86] M. Schindler, "Silicon Compilers Traverse Rough Road to Acceptance", Electronic Design, May 1, 1986, pp. 156-166.
- [Sham85] M. Shahdad et al., "VHSIC Hardware Description Language", COMPUTER, Vol. 18, No. 2, February 1985, pp. 94-103.
- [SouJ83] J. R. Southard, "MacPitts: An approach to silicon compilation", COMPUTER, Vol. 16, No. 12, Dec. 1983.
- [SteG84] G. L. Steele Jr., "Common LISP: The Language", Digital Press, 1984.
- [SubP86] P. A. Subrahmanyam, "Synapse: An Expert System for VLSI Design", COMPUTER, Vol. 19, No. 7, July 1986, pp. 78-89.
- [SuzN85] N. Suzuki, "Concurrent Prolog as an Efficient VLSI Design Language", COMPUTER, Vol. 18, No. 2, February 1985, pp. 33-40.
- [TseC86] C-J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems", IEEE Transactions on Computer-Aided Design, Vol. CAD-5, No. 3, July 1986, pp. 379-395.
- [UllJ84] J. D. Ullman, "Computational Aspects of VLSI", Computer Science Press, Rockville, Md., 1984.
- [VIVID] VIVID Designer's Documentation, Microelectronics Center of North Carolina.
- [ZimG79] G. Zimmermann, "The MIMOLA Design System: A Computer Aided Digital Processor Design Method", Proc. 16th Design Automation Conference, June 1979, pp. 53-58.

APPENDIX B

GLOSSARY

ASIC (Application Specific Integrated Circuit): An integrated circuit designed for one special application. It may be implemented as a gate array, standard cell design, or handcrafted full-custom design, depending on the design turnaround, circuit performance, and cost constraints.

Behavioural Description: A representation of a circuit in terms of what the circuit does. This representation is independent of the technology and implementation details. A program written in a high-level programming language is an example of a behavioural description if the program represents the operations to be performed by the circuit.

Caore: An expert system for VLSI design developed at AT&T Bell Laboratories.

CDFG (Control/Data Flow Graph): A graphical representation of a high-level algorithm. Nodes of the CDFG represent operations on data. Directed arcs connecting the nodes represent the flow of data between operator cells in the circuit.

CMC-DA: Hardware compiler developed at Carnegie Mellon University.

Control Graph: A graphical representation of the control behaviour of a circuit. This graph is similar to a state diagram. Each node represents the control operations for one clock cycle. Directed arcs between the nodes represent the sequencing of operations.

Control Path: The part of a circuit which generates control signals for each clock cycle. This part of the circuit may also perform loops, conditional branches, and subroutine calls.

Control Point: A cell in a circuit which is controlled by a field in the control memory.

Control Step: Corresponds to a clock cycle in a synchronous circuit.

Control steps are used to assign operations in the control/data flow graph to clock cycles in the actual circuit. Operations in the same control step are performed in the same clock cycle.

DAA: A hardware compiler under development at AT&T Bell Laboratories. It is an extension of CML-DA.

Data Path: The part of a circuit which performs the actual data processing operations. The data path and control paths together form a complete circuit.

Elf: A hardware compiler developed at Carleton University and Bell-Northern Research.

Evoke Signal: A control field controlling a hardware cell which stores data. A register's control field is an example of an evoke signal.

Facet: A hardware compiler developed at Carnegie-Mellon University.

Functional Description: A representation of a circuit in terms of its high-level behaviour; a behavioural description.

Geometric Description: A representation of a circuit in terms of its physical implementation as a chip. At its lowest level, a geometric description consists of the masks for the chip.

GPIC (General Purpose Integrated Circuit): An integrated circuit designed to perform in a broad range of applications. Examples of GPICs are devices such as microprocessors and memory chips.

HAL: A hardware compiler under development at Carleton University and Bell-Northern Research.

Hardware Cell: A VLSI component for performing a specified function.

Cells form the building blocks of the standard-cell design methodology. Simple cells, such as inverters or I/O pads, may contain a few components; other cells, such as adders and ALUs, may be very complex. Parameterized cells are constructed so that some parameter, such as the number of bits in a register, is needed to realize the cell. Cells are typically stored in a database called a cell library.

Hardware Compiler: A program which produces a low-level structural description of a circuit from a high-level behavioural description of the algorithm to be performed by the circuit. hardware compilation involves the synthesis of a circuit to perform a user-specified function, meeting user-specified constraints.

HDL (hardware Description Language): A textual language for describing the structure or behaviour of a circuit.

ISP: A language for describing central processing units of computer systems.

MacPitts: A hardware compiler developed at MIT. A commercial version, MetaSyn, was not successful.

Minola: A hardware compiler developed at the University of "el and subsequently ported to Honeywell.

Net List: A circuit description which specifies the hardware cells used in the circuit and their interconnections.

Select Signal: A control field controlling a hardware cell which does not store data. A multiplexer's control field is an example of a select signal.

Silicon Compiler: A program which converts a structural description of a circuit to a layout. The difference between silicon and hardware compilation, as used in this thesis, is that the input to a hardware compiler is at a higher level of abstraction than for silicon compilers.

Structural Description: A representation of a circuit in terms of its architecture. This description is abstract because the circuit is represented by schematic diagrams even at its lower levels.

Synapse: An expert system for VLSI design under development at AT&T Bell Laboratories.

VHDL: The hardware description language of the Very High-Speed Integrated Circuit program.

APPENDIX C

Sample Input and Output

This appendix contains the Control/Data Flow Graph for the Shell sort algorithm, and the microcode produced by HiSyn. Both the uncompact and compacted control memories are shown. The pseudo-Pascal program below represents the algorithm being performed by the circuit.

```
LOOP
  Read (n)
  m := n
  WHILE (m > 1) DO
    m := m/2
    k := n - m
    j := 1
    REPEAT
      i := j
      break := 0
      REPEAT
        h := i + m
        IF (mem[i - 1] > mem[h - 1]) THEN
          t := mem[i - 1]
          mem[i - 1] := mem[h - 1]
          mem[h - 1] := t
          i := i + m
        ELSE
          break := 1
        END IF
      UNTIL (i < 1) OR (break = 1)
      j := j + 1
    UNTIL (j > k)
  END WHILE
END LOOP
```

The following pages show the Control/Data Flow Graph and a register-IIUA circuit for this program.

(continued on next page)

t 14

45

41

DH

51 WR (memory write)

t 15

42

FE

FE

52

JOIN

53

55

t 16

56

T

BR

OR

57

J

K

t 17

58

LOOP - EXIT

59

LOOP - END

t 18

60

1

61

t 19

62

J

63

t 20

64

LOOP - EXIT

65

LOOP - END

t 21

66

LOOP - END

t 22

67

LOOP - END

This diagram shows the circuit for the Shell sort example.
Different line styles are used for clarity only.

This page shows the control-memory generated by MSyn. Note the large number of don't cares, indicated as 'X'.

[illegible][illegible]

The microcode compaction algorithm groups compatible control fields into compatibility classes. The list below shows how the original microcode fields are grouped together.

Class CLASS-1: (REG-M-OP)
Class CLASS-2: (REG-N-OP)
Class CLASS-3: (REG-J-OP)
Class CLASS-4: (REG-K-OP)
Class CLASS-5: (REG-MK-OP)
Class CLASS-6: (MUX-06/1-OP REG-I-OP)
Class CLASS-7: (MEMORY/1-OP)
Class CLASS-8: (ADD-01-OP)
Class CLASS-9: (MUX-08/1-OP MUX-07/1-OP)
Class CLASS-10: (MUX-07/2-OP)
Class CLASS-11: (MUX-09/0-OP MUX-08/0-OP MUX-06/0-OP)

Class CLASS-12: (MUX-09/1-OP MEMORY/0-OP MUX-06/2-OP)
 Class CLASS-13: (REG-T-OP MUX-07/0-OP REG-01-OP)
 Class CLASS-14: (MUX-05-OP MUX-04-OP REG-DH-OP MUX-10-OP ADD-02-OP)
 Class CLASS-15: (SHF-01-OP MUX-03-OP MUX-01-OP MUX-02-OP REG-DI-OP
 REG-H-OP REG-0H-OP)

This shows the control memory after compaction. The 33 original control fields have been reduced to only 15 fields.

```

A C C C C C C C M M R R R R R C B
D L L L L L L L E U E E E E E C R
J A A A A A A A M X G G G G G - -
- S S S S S S S O - - - - - S A
U S S S S S S S R U B J K M N E D
1 - - - - - Y 7 R - - - - L D
- 1 1 1 1 1 6 9 / / - 0 0 0 0 K
0 1 2 3 4 5 1 2 0 P P P P P
P - - P

```

```

-----
0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 X
1: 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 X
2: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0
3: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 X
4: 0 1 1 0 0 1 0 1 0 0 0 0 0 0 1 0 0 X
5: 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 X
6: 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 X
7: 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 X
8: 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 X
9: 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 X
10: 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 X
11: 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 X
12: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 15
13: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 14
14: 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 17
15: 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 X
16: 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 0 1 17
17: 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 X
18: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 7
19: 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 X
20: 0 0 0 1 0 1 1 0 0 0 0 0 1 0 0 0 0 X
21: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 6
22: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1

```

END

0|4.01.88

FIN