A RETARGABLE MICROCODE GENERATOR FOR THE LAGER ENVIRONMENT

by

Monte F. Mar

Memorandum No. UCB/ERL M89/78

19 June 1989

A RETARGABLE MICROCODE GENERATOR FOR THE LAGER ENVIRONMENT

by

Monte F. Mar

Memorandum No. UCB/ERL M89/78

19 June 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A RETARGABLE MICROCODE GENERATOR FOR THE LAGER ENVIRONMENT

by

Monte F. Mar

Memorandum No. UCB/ERL M89/78

19 June 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

The ability to rapidly implement VLSI custom processor based systems is limited by the tools that can map high level language expressions into microcode for the new architectures. Past efforts have used dedicated code generators or simplified control structures not requiring code generators. A bottom up design approach has been used to develop a prototype self retargeting microcode generator for the Lager Silicon Assembly System. A graph tracing algorithm was developed and implemented as the basis of the code generator. By making use of structural information normally supplied for layout generation, the algorithm can map C language statements to a variety of architectures. The prototype code generator has been tested on parts of a multiprocessor fingerprint filtering system. Results show that the approach is feasible, but improvements are needed. The report discusses the design condsiderations and development of the prototype code generator. A tutorial example for the prototype microcode generator is presented in this report.

Acknowledgements

I thank my family for encouraging and supporting me throughout the past 2 years. Special thanks to my father for his advice on technical writing.

I would like to thank Professor Rajeev Jain of UCLA for providing the opportunity to work on this project. He is responsible for introducing me to the world of CAD tools and design automation. I would also like to thank Professors Robert Brodersen and Jan Rabaey for reading the report and suggesting changes and improvements. Special thanks to Professor Brodersen for being my advisor during the project. I also acknowledge Paul Yang, Paul Tahjadi, and Phil Duncan of UCLA for their help in the effort to implement to smoothing processor of the fingerprint filter.

This research was sponsored by DARPA under the contract N00039-87-C-0182 .

Contents

List of Figures

Chapter 1

Introduction

A design of a filter for a fingerprint recognition system has shown that a custom multiprocessor architecture can provide a compact high throughput system that will process an image in about 10 seconds [1]. Using multiple Kappa processors from the Lager system [2], an optimistic estimate is about 25 seconds. On a VAX 8650, it takes 5 minutes [3]. The amount of speedup attained in this case is directly related to the amount of customization. Custom designs can provide higher performance, but the increase in performance is gained at the cost of increased design time and increased processor complexity.

The Lager IV Silicon Assembly system provides support for the development of complex integrated circuits. It is oriented towards structural compilation from hardware descriptions and currently provides support for a single processor architecture that can be slightly altered. The user pays a cost for making changes: the hardware description and the architecture descriptions necessary for both the compiler and microcode generator must be modified and then verified. In order to explore an entirely new architecture, the user must create several new description files and debug them before they will be useful for processor design.

1.1 Goals and Objectives

The problem examined in this research project was the generation of microcode for custom micro-programmed processors. This research was motivated by the implementation of a multiprocessor fingerprint filter system. The design of the processors had already been completed [1], but since each processor used a new architecture, none of the existing

Lager tools could generate the microcode. Thus the objective of this research was to provide a means of mapping microcode descriptions to a class of architectures. Inherent in this objective was the desire to minimize the cost of retargeting the microcode generator when switching from architecture to architecture, allowing designers more freedom in their implementations without forcing them to use an existing target architecture or to rewrite existing tools.

The goal of this project was to create a prototype microcode generator that could retarget itself using information found in the design database. The designer would only have to specify the structure of the processor and provide a microprogram description. To aid the designer in microprogram verification, a simulator should be provided for microprogram descriptions. Finally, the prototype microcode generator should be able to generate the microcode for the 3 processors in the fingerprint filtering system.

1.2 Outline

This report consists of 6 chapters. Chapter 2 reviews concepts about retargetable microcode generators and their relation to silicon compilers. In Chapter 3, the basic approach to the problem is outlined. Some details of the implementation for the microcode generator developed in this project are discussed in chapter 4. Chapter 5 gives a tutorial example of using the microcode generator. The report concludes with a discussion of the results in chapter 6.

Chapter 2

Basic Concepts for Microcode Generation.

2.1 An Overview of Microcode Generation.

A basic code generator like the one shown in Fig. 2.1 translates an input description to binary machine code using information about the processor architecture. The architectural information includes the timing schemes used in the processor, the type of registers available, information about the controller, and the instruction set.

Software programmable processors use code generators that are optimized for a given target architecture. An example is the code generator in the Intel ASM86 assembler, which was written specifically to handle the peculiarities of the 8086 microprocessor [4].

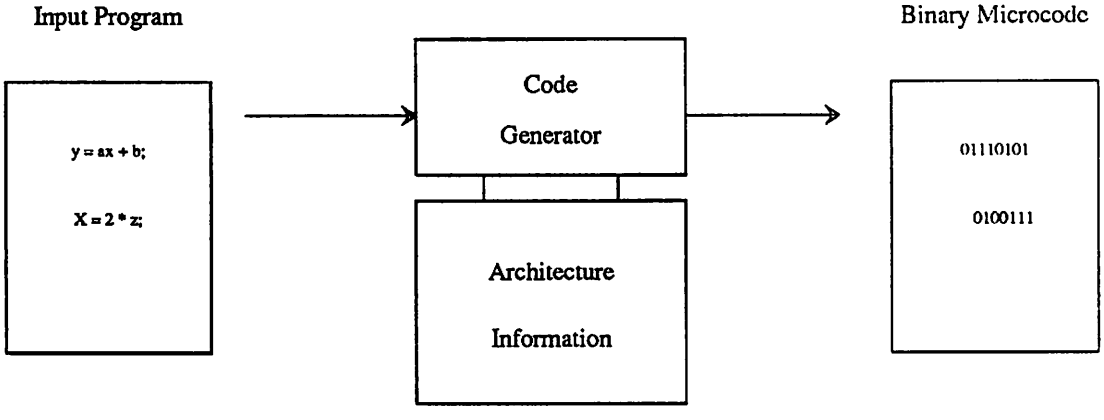


Figure 2.1: A Basic code generator.

The fixed architecture of a general purpose processor allows software developers to apply established techniques for code generators used in assemblers and compilers. Heuristic approaches must be used for code generation because the complexity for generating optimal code is not known [5]. The appropriate code generation techniques for a given architecture are determined by the target machine and the instruction set.

ASIC designs encourage the development of new architectures that handle computations for algorithms in a dedicated and efficient manner. Since a code generator is a function of the instruction set and the target machine, the best code generator for an ASIC processor is an application specific one. Development of specialized code generators for every new ASIC would be costly, so a more generalized code generator is desired.

A solution is to develop a retargetable code generator which can generate microcode for many different target processors. Retargeting is performed by providing information about the new target machine. Generalized code generation techniques are applied which can lead to suboptimal code. Although the code is suboptimal, the gain in development time is substantial. Thus retargetable code generators provide the more generalized framework for ASIC processor design.

2.2 Instruction Sets and Code Generation.

The information necessary for code generation consists of the processor instruction set and the collection of binary control words defined by a hardware implementation of the instruction set. These 2 sets of information can be summarized in a behavioral description. Mueller has pointed out 2 approaches for expressing instruction sets [6]. One method is based on grammar descriptions while the other is based on data flow graphs.

In the grammar method, a set of micro-operations is defined. These micro-operations are combined into lists to form instructions. The legal combinations of micro-operations are defined by a grammar, much like the way that English grammar provides rules for combining words into sentences. Micro-operations are usually defined as mnemonic strings. During code generation, control words are assembled by translating the mnemonic strings to their binary equivalent.

In the data flow method, instructions are expressed in terms of data flow graphs. Each node in the graph represents a micro-operation, while each edge represents data flowing from one node to the next. A collection of nodes and edges forms an equivalent data flow

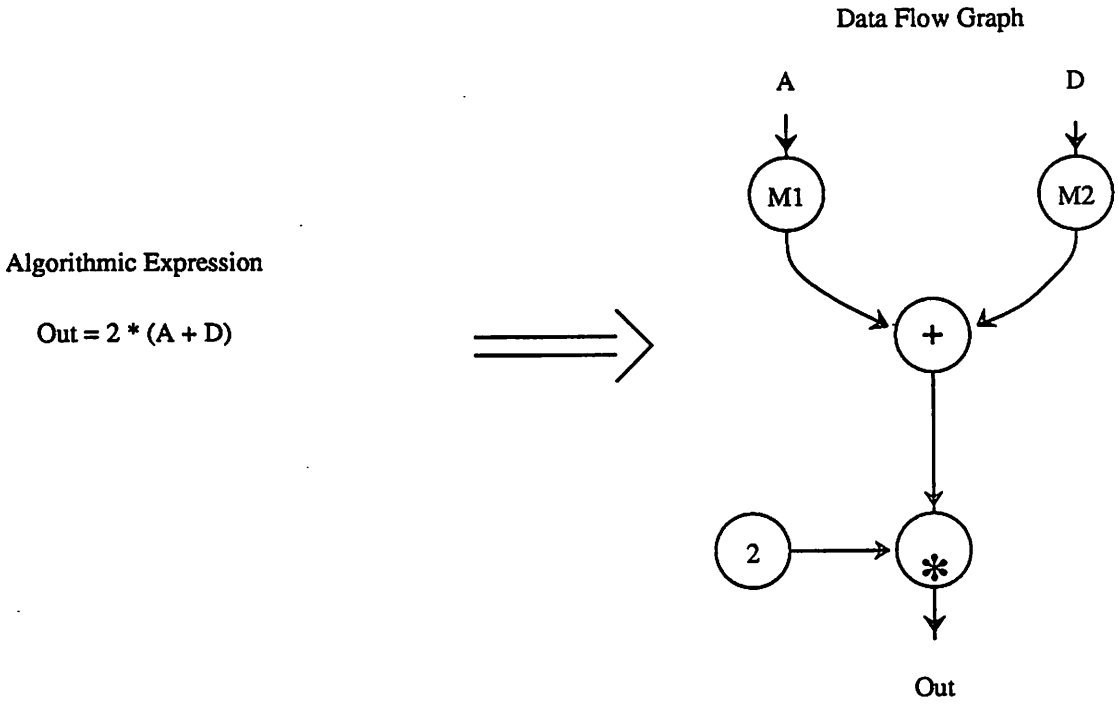


Figure 2.2: Using data flow graphs to describe an add instruction.

graph representing an instruction. Fig. 2.2 illustrates an example of assembling some nodes and edges to create an instruction representation. Since the instruction is represented in graphical form, an algorithm is needed to translate graphs to binary control words.

The data flow method provides a more generalized framework from which to build a retargetable code generator. With a data flow approach, an annotated graph of data flow in an architecture can be used to cleanly and uniquely describe instructions. The data flow approach has the advantage that more information can be stored in a graph then in a list of micro-operations. The arrangement of micro-operation nodes in the graph can reflect actual features in an architecture.

2.3 Silicon Compilation and Code Generation.

Retargetable code generators can be used in silicon compilers. For a given silicon compiler, various approaches give different results. No single approach seems to suit all the needs of the different code generation environments.

There are 2 types of silicon compilers. One type (structural compiler) uses struc-

tural descriptions to generate a layout. The Lager System is an example of this type [13]. The second type (behavioral compiler) uses behavioral descriptions to synthesize the necessary hardware descriptions from information found in leafcell libraries [7] [8] [9] [10]. There is some control over the decisions made at each level of synthesis, but not as much as with the structural compiler, which allows full specification of the design at all steps.

The behavioral silicon compiler usually includes powerful synthesis tools that can essentially define the instruction set and account for other processor characteristics. Thus the required code generator is a translator that matches mnemonics and substitutes binary machine language code. The simplicity of this type of code generator is gained at the cost of developing complex synthesis tools which define the instruction set for the processor.

Two types of code generators can be developed for structural compilers. The first type requires the human designer to provide a description of the instruction set. The code generator uses this description to map the algorithm description onto the architecture. The second type of code generator uses the structural description and supplementary behavior descriptions to deduce the instruction set and generate the microcode. In the first case, a human must provide extra information in the form of an instruction set to retarget the generator, but in the second case, the generator deduces the instruction set.

2.4 Microcode Generation and The Lager System.

2.4.1 Past Microcode Generation Work in the Lager System.

The Lager I system was targeted to a single processor architecture [11] [12] for which a dedicated assembler and assembly language simulator were developed. A predefined and limited instruction set limited the scope of the problem so that an effective solution was obtained.

The Lager III system [13] provided a structural description language that could be used to describe general hardware configurations. A datapath compiler provided the potential for a user to explore and rapidly prototype new datapath designs [14].

The Lager III system supported the Kappa architecture [2]. The assembler for this architecture, rassCG, translated a grammar description of an instruction into the corresponding machine language. A compiler was also developed that could take a high level program description in a C like language and compile it into the input for rassCG [15].

Simulation of processor designs was performed using a simulator called DSIM, which was based on LISP submodules and was slow.

Lager IV is an enhanced version of Lager III. It is implemented in the C programming language and uses the OCT database to store information. Lager IV uses a modified version of the Lager III microcode generator.

2.4.2 Background on Lager IV and the OCT database.

Before discussing the approach, it is necessary to review some of the basic concepts of the Lager IV system. The review presented here is limited. More information can be found in the Lager IV User's Manual and the OCT Tools Distribution Manual.

Lager IV.

Lager IV is a system used to perform the layout for ASIC designs. The user writes a structural description language (sdl) file describing the hardware configuration of the design. The sdl files can be written hierarchically. Directives in the sdl files dictate which layout generation tools will be used at each level of the hierarchy. The layout generation tools include a module generator, a place and route tool, and a datapath compiler. Designs can be parameterized to increase flexibility.

Fig. 2.3 shows the layout process which is directed by the Design Manager, the user interface for Lager IV. During the layout generation process, the Design Manager creates the structure master view (smv), the structure instance view (siv), and the physical view which are stored in the OCT database.

The OCT Database.

A description of the OCT database can be found in the OCT Tools Distribution Manual [16], but a summary of OCT concepts is presented here to point out the more important issues. The OCT view is the place where information is stored. The view is accessed by opening a particular facet of the view. The siv and smv mentioned in the last subsection are examples of views. Within the view, different types of objects are used to construct a directed graph. OCT provides objects called NET, TERM (terminal), and INSTANCE. By attaching TERMs to INSTANCEs, NETs can then be used to connect other INSTANCEs through the terminals. Other OCT objects are BAGs and PROPs

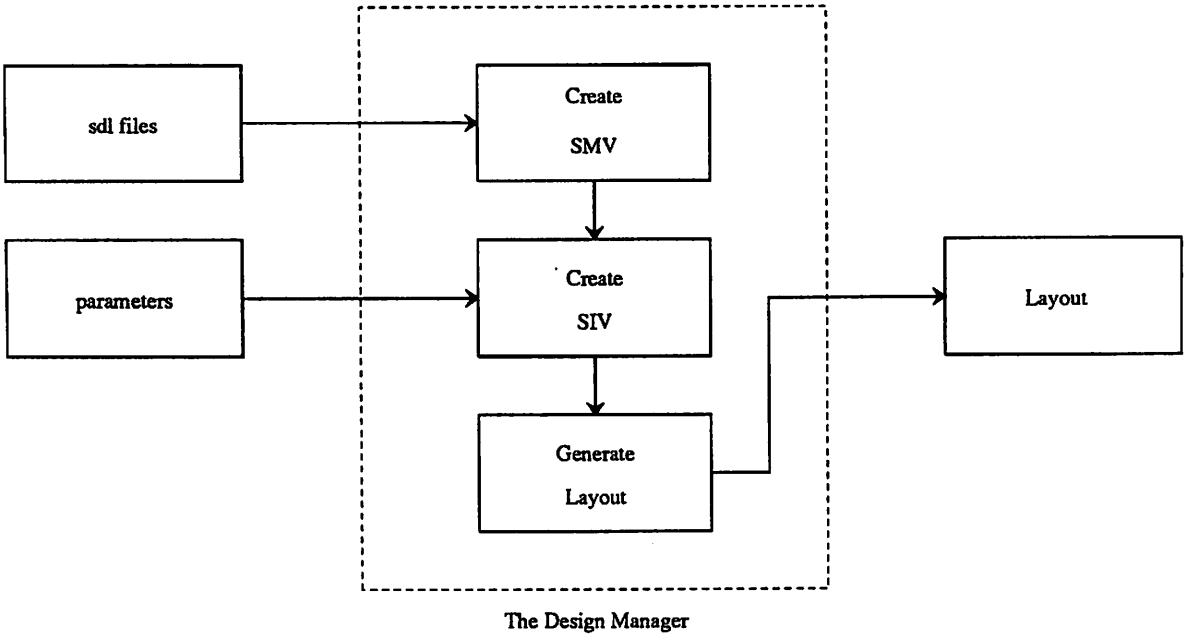


Figure 2.3: Design process using Lager IV.

(properties). BAGs can be used to hold collections of other OCT objects. A PROP can be attached to OCT objects to describe a property associated with that object.

2.5 Summary

To reduce design time for ASIC processors, flexible and retargetable code generators are needed. Within the Lager System, past efforts have focused on dedicated code generators. Lager IV provides the environment for the development of retargetable code generators. Since data is stored in graphical form using the OCT database, it is possible to create a code generator using the graphical approach to describe the instruction set.

Chapter 3

Developing a Microcode Generator for LagerIV.

3.1 Objectives for a Lager IV Assembler.

The objectives for this research were:

- To develop an assembler that would work for the fingerprint filter processors.
- To simplify the format of the architecture description file, or eliminate the need for one.
- To simplify and generalize the assembly language input.
- To provide a simulator for verification of the assembly language input.
- To provide a system that would help Lager IV users in developing new processor architectures for other applications.
- To place emphasis on microcode generation, leaving other issues like code compaction and optimization for future tools.

These objectives were developed from a study of the Lager III rassCG assembler. The assembler required a register transfer language program, the equivalences between register transfers and the actual microcode, and some additional information about timing. The input file formats for rassCG are discussed in [13].

3.2 Assumptions and Models for a Code Generator.

The goal of this project is not code optimization, but rather to focus on ways to minimize the effort needed to specify the necessary architectural information to a retargetable code generator.

In microcode and control hardware generation, previous researchers have focused on finite state machine solutions and code optimization techniques [17]. The approach has been to assume that the datapaths of a processor system were predefined. The task was then to optimize the definition of control word structure with respect to speed or area constraints, and then synthesize control hardware.

In this research, the datapaths were also assumed to be predefined. Additionally, a target control unit was assumed. The approach was to try and extract the architectural information from the smv design database and from some extra behavioral information and use this as input to a retargetable code generator.

3.2.1 An Overview of the Approach to Code Generation.

The data flow method for expressing a processor instruction was adopted for this microcode generator. The data flow method was introduced in section 2.2. Fig. 3.1 illustrates the approach to microcode generation. The input instruction must clearly indicate where the source operands and result are stored. The storage location of the result will be referred to as the destination. By using the sources and destination as reference points, a data flow graph of an instruction can be traced within the structure master view. This information along with supplementary behavioral information can be used to translate the instruction to microcode.

This scheme eliminates the need for the user to code the instruction set into a database. The code generator verifies which instructions can be implemented on the processor. The code generator can retarget itself, but the connectivity graph, in this case the structure master view, must be specified in such a way that a connectivity search will converge to the correct data flow graph that represents the current instruction. An architecture model provides the necessary guidelines.

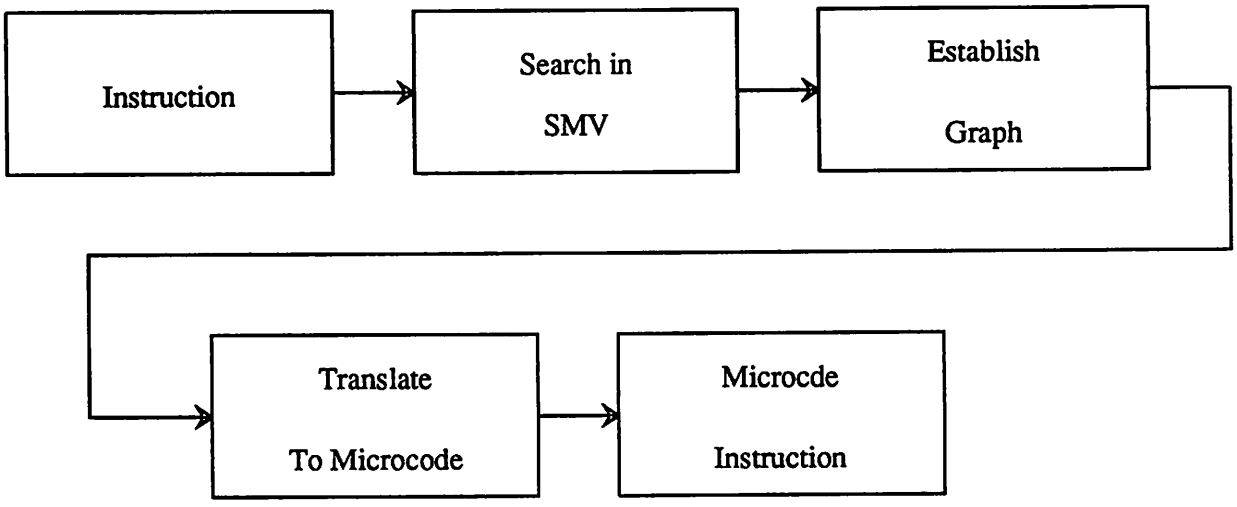


Figure 3.1: Approach to microcode generation.

3.2.2 The Architecture Model.

An architecture model provides limits that simplify the solution to the problem. The adopted model sufficiently describes the 3 fingerprint processors and most other processors that can be described in terms of blocks of combinational logic separated by latches or registers.

An example model architecture is illustrated in Fig. 3.2. There is a single controller and an arbitrary number of datapath blocks and memories. Each cell in the datapath must have a behavioral description that can uniquely identify the functions of the cell. Additionally, datapath cells must have only one connected output terminal for data. The last assumption is required by the algorithm used to map assembly language expressions to microcode. The number of data input terminals on datapath cells has been limited to 2 or less.

The architecture model assumes that the processor can be described in terms of datapaths implemented by the Lager IV program dpp, memory blocks like RAMs and ROMs, and the controller block. A system level description of the processor must contain only references to sdl files describing blocks that fit in this list.

The model assumes two phase non overlapping clocks since the controller model uses this clocking scheme. The controller model adopted for this project is based on the controller from the Lager Kappa processor, which is illustrated in fig. 3.3. The split structure allows control flow and data flow to proceed in parallel. Data operations are directed by the

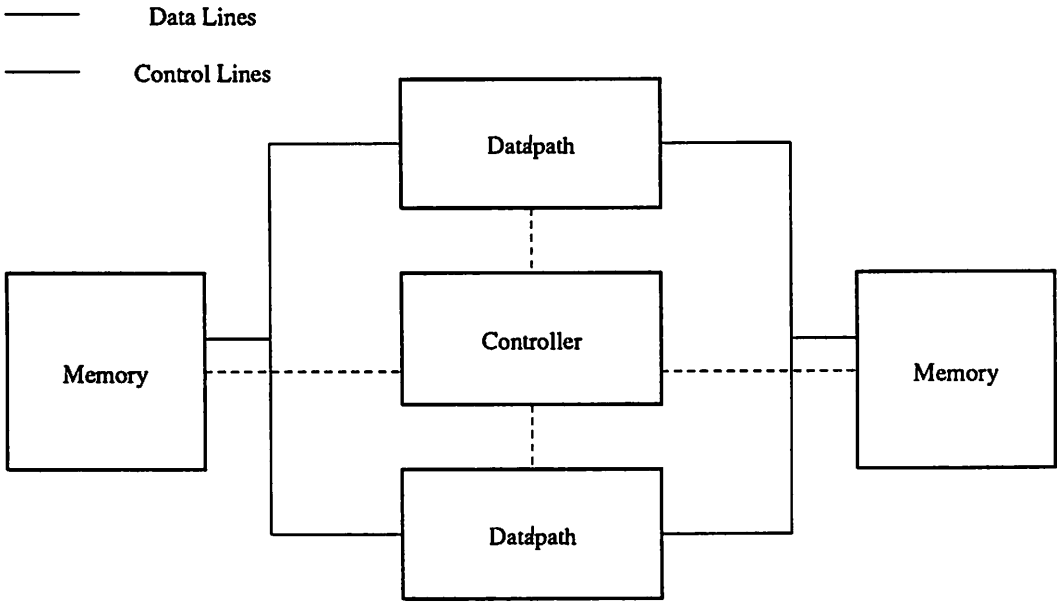


Figure 3.2: Architecture model example.

control store, while control flow is managed by the control finite state machine. The major changes made to the Kappa controller were the exclusion of the timer and loop counter. Loop counting is assumed to take place on the datapath itself. This allows nesting of loops, but also incurs a one cycle overhead that could be avoided by using the loop counter.

Previous use of the Lager Kappa architecture has shown the controller to be versatile. It handles looping and jumping instructions efficiently. Conditional inputs to the control finite state machine can be used for jumping or branching, but a one cycle delay occurs between the time the signal is presented to the controller and the time that the control signals from the control store reflect the branch.

Since the controller is partitioned into 2 parts, it is possible to create a single controller that oversees program execution spread across several datapaths. It is possible to broadcast the control finite state machine status to several partitioned control store finite state machines. In addition to increasing layout efficiency, the smaller partitioned control store finite state machines may be faster. This alternative has not been explored in this project.

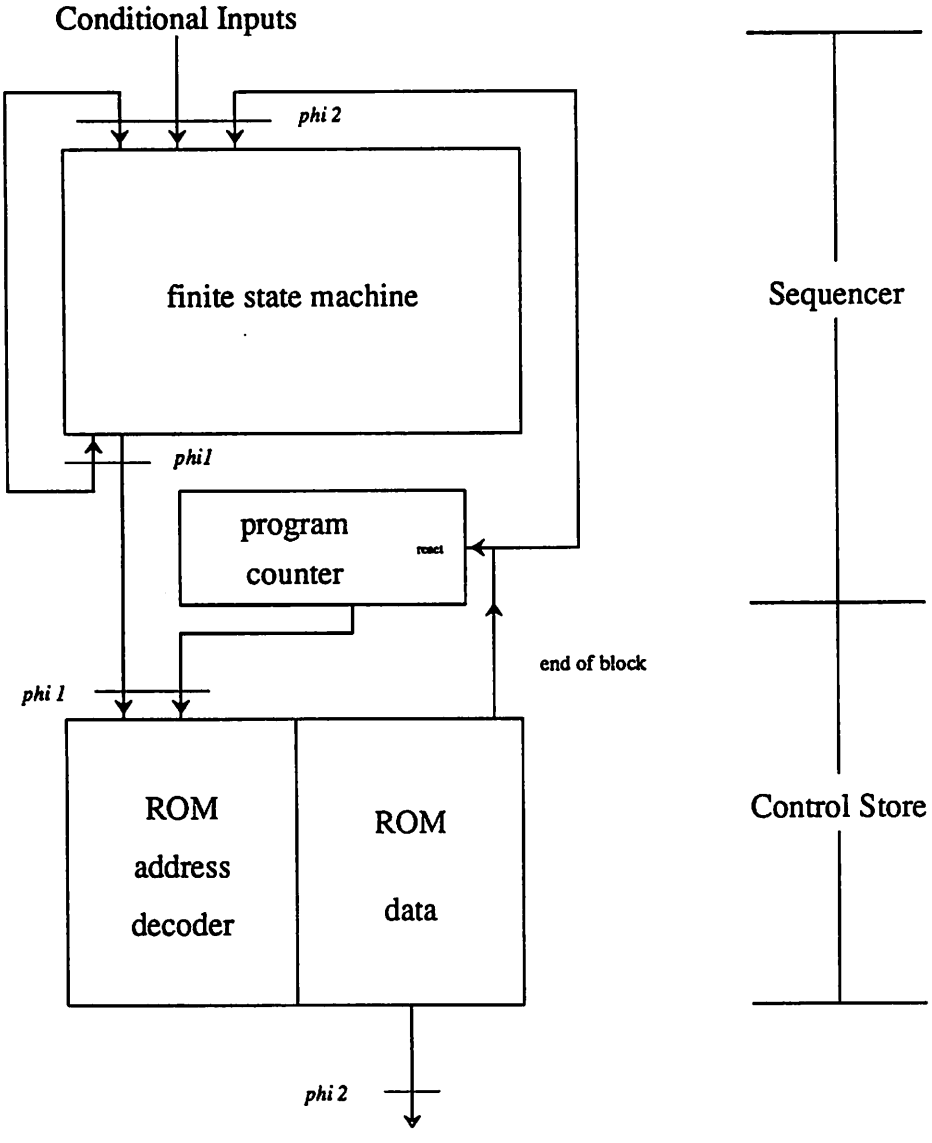


Figure 3.3: Block Diagram of the Lager Kappa Controller.

3.2.3 Effects of Assuming an Architecture Model.

The restrictions provide some general design rules to follow. The restrictions and assumptions were imposed to insure that all processor instructions can be expressed in terms of a data flow graph. Pipelining may seem difficult, but by following the restrictions, it is still possible to design pipelined architectures. Pipelined instructions can be expressed if the data transfers between pipeline latches are considered register transfers.

The choice of a single target controller limits the scope of the application. For processors that only require simple control, this controller would be too large and too complex. But for algorithms that contain numerous control flow operations, the controller provides good support.

The architecture model does not address the issue of multiprocessor systems that require communication between separate controllers. In order to design multiprocessor systems, it may be necessary to alter or enhance the architecture model to provide for interprocessor communication.

The architecture model also does not address input data streams. Since the fingerprint application stored data in memories, the algorithms were developed without considering streams. The introduction of some new control flow constructs in the input assembly language would allow the model to handle streams.

3.3 A Behavioral Description.

In addition to the connectivity information available from the structure master view, the binary control signals needed for the machine language instructions must be stored on a cell by cell basis. The information about control signals and operations that the cell can perform can be stored in the OCT database for each cell in a separate view. An experimental behavior view for the OCT database was developed for this application. It is assumed that designers will make use of library leafcells, so behavior views should be defined for each library cell. Thus an sdl description should be sufficient to describe a processor architecture because the behavior view can be obtained using the library.

A leafcell may be able to perform more than one operation. For instance, an adder subtractor cell can perform both addition and subtraction. The symbols "+" and "-" were used to describe the behavior. Within the behavior view for the cell, provisions were

made to store these types of descriptions and the corresponding binary signals necessary to execute the operation. Further details on the behavior view will be discussed in Chapter 4 as part of the details of the current implementation.

3.4 Summary

In this chapter, the main objectives and approach for a retargetable microcode generator were presented. An architecture model was adopted to allow assumptions which will simplify the implementation. While the restrictions seem cumbersome, they do allow the processor instructions to be defined in terms of data flow graphs. These data flow descriptions can be extracted from the information found in the structure master and the behavior views. A code generator based on this approach can retarget itself every time a new algorithm is mapped to a new processor architecture specified by a structure master view, provided that the corresponding behavioral views already exist in the library.

Chapter 4

Implementation Details of the Microcode Generator.

4.1 Overview.

The microcode generator was written and developed in the C programming language. The microcode generator is made up of a parsing unit and a mapping unit as shown in fig. 4.1. A C like language was chosen for the input program description. The sdl/bdl description shown in the figure can be accessed through the structure master view. The output of the assembler consists of various parameters needed by the Lager system to create the structure instance view from the structure master view. Among the output parameters are the logic tables for the finite state machines of the controller, which are written in a format accepted by the program *espresso* [16]. The Lager tools can use *espresso* to minimize these tables.

The code generation flow diagram is shown in fig. 4.2. The input program is first processed by the parsing unit, which creates a list of all the data manipulation statements and a list of all the control flow statements. The control flow statements are then mapped to the format required by the control finite state machine and are written out to a file. Next, the list of data manipulation statements is passed to the mapper unit, where each statement is processed. The mapper unit makes use of information from the behavioral view and the structure master view to map the statement to the machine language equivalent. When all the statements are correctly mapped, a file with the control store logic tables is written.

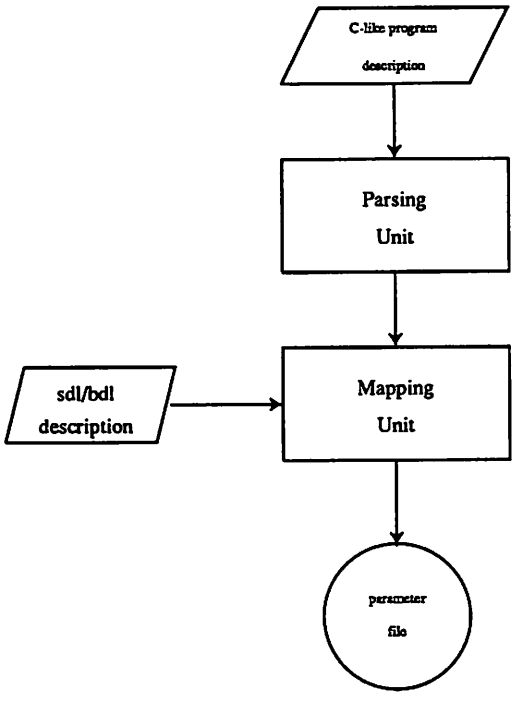


Figure 4.1: Basic diagram of the microcode generator.

The discussion of the microcode generator will be divided into 2 sections, one focusing on the parsing unit, and the other focusing on the mapping unit.

4.2 The Parsing Unit.

The parsing unit processes the input program, splitting it into 2 lists. The language is a subset of C with a few added constructs. C was chosen for its structured programming and the fact that it was originally developed as a portable language. In addition, C compatibility would allow the input code for the microcode generator to be compiled using the standard C compiler, making a microcode simulator unnecessary. Code could be written, developed, and verified in C.

4.2.1 The Input Program Model.

The input program must follow a general model which will guide the parser in recognizing program features in the program. The program model is illustrated in fig. 4.3 using a contrived program. In the declaration section, the variables are assigned types.

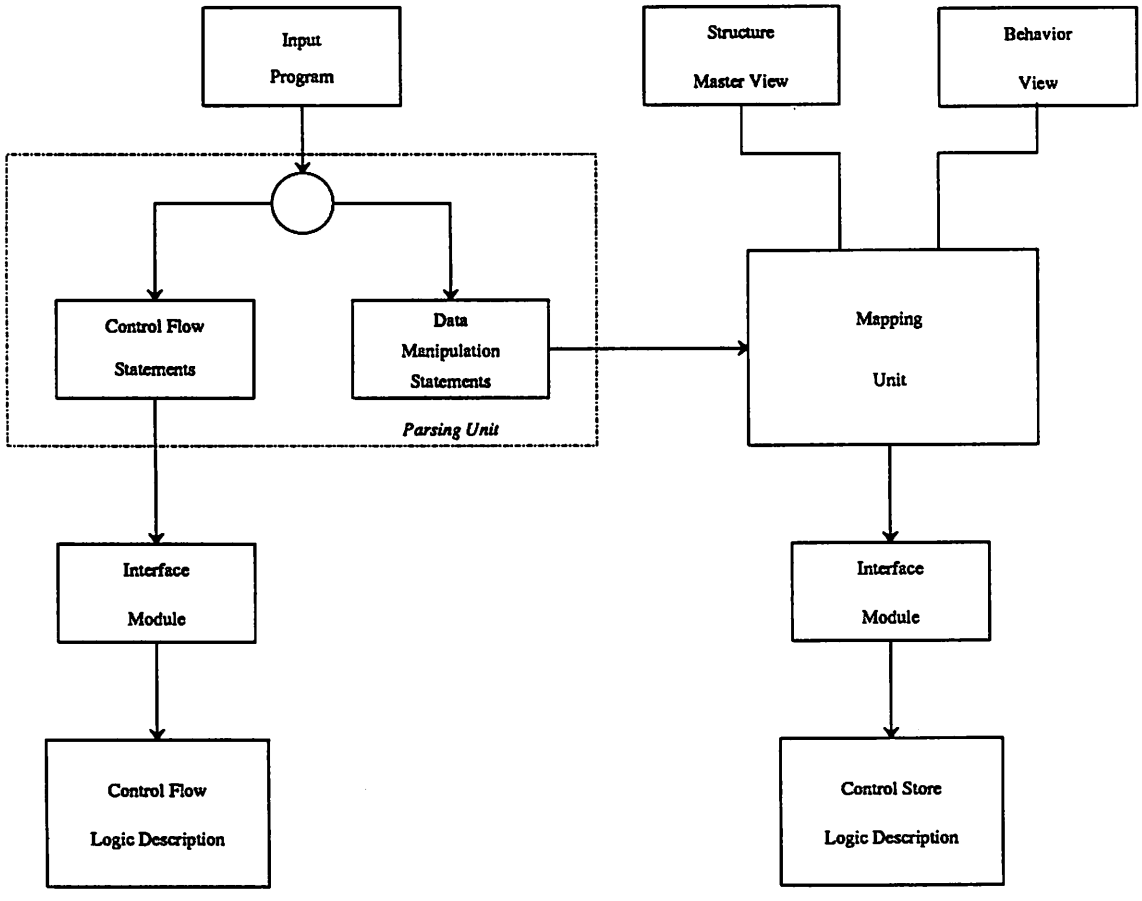


Figure 4.2: Block diagram of the microcode generator.

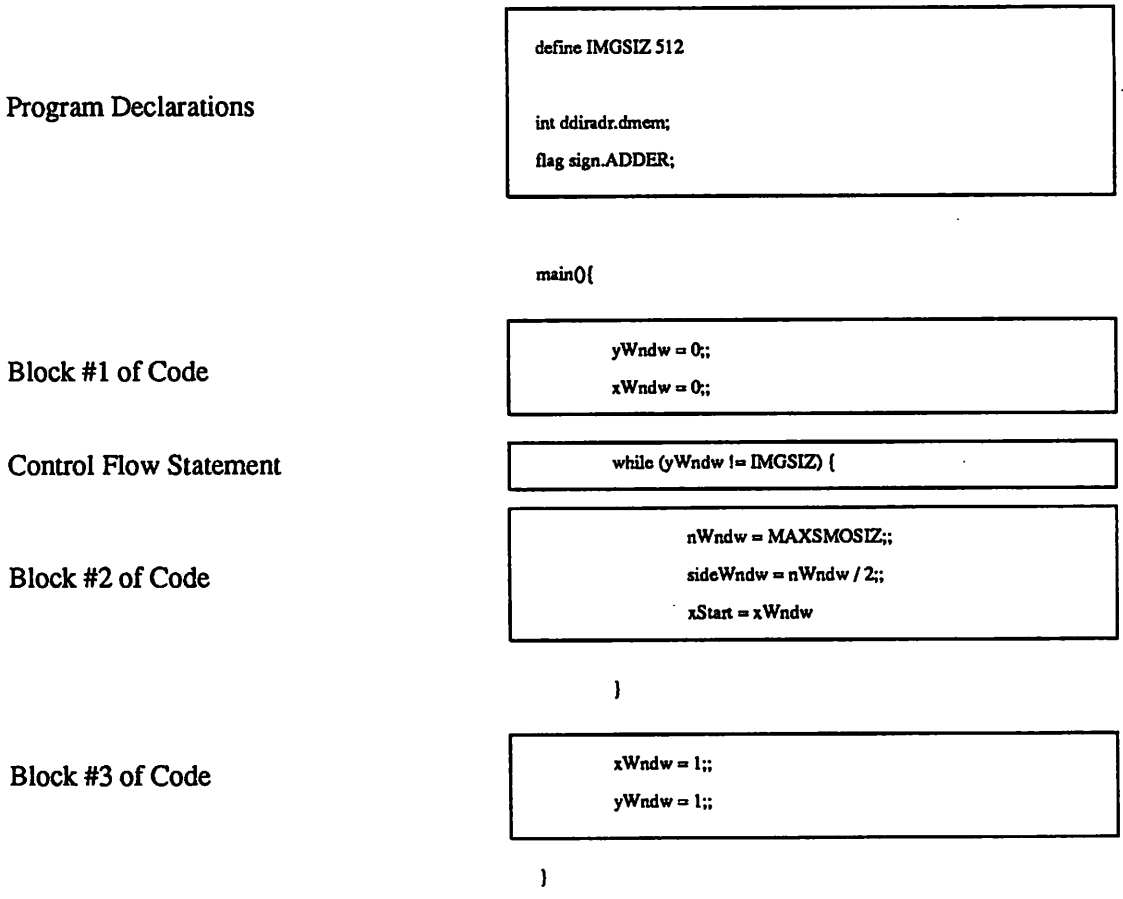


Figure 4.3: Example of the program model.

They are also paired with the names used for a memory or register in the structure master view where they will be stored. The exact syntax is given in Appendix A. The program model forces the user to bind the symbolic variables to the actual memory resources in the processor. The model also assumes that all variables are global. The program body following the declaration section is written in C. The body of the program is made up of blocks of sequential statements separated by control flow statements like *while*, *do*, or *if*.

4.2.2 Details About the Parsing Unit.

The parser was developed using the Unix facilities Lex and Yacc. Lex is used to specify a lexical analyzer and Yacc allows the user to specify rules by which tokens passed from the lexical analyzer can be recognized. Yacc accepts specifications that are based on LALR grammars with disambiguating rules. Together they can be used to generate a

procedure that will handle the input of a program from a text file.

The lex specification used in this project contained definitions for recognizing keywords. Some of the keywords were from C, while others were defined to aid recognition of special constructs. For instance, the keyword *nop* was added so that the microcode generator would recognize a statement for no operation.

The yacc description contained rules for controlling the parsing and recognizing the structure of single C statements. The parser was designed to parse a single statement at a time, then to process it, and then to go to the next statement.

The parser recognizes 2 consecutive semicolons as the end of a processor cycle. Consider the following statement:

$$z = x + y;$$

$$a = b;;$$

In standard C, this would be interpreted as 2 consecutive sequential statements: the double semicolon does not affect the operation. It merely implies that after the second statement, there is an empty statement. In contrast, the parser interprets the code to mean that 2 statements will be executed simultaneously.

To create the control flow graph, the parsing unit reads each line and determines whether it is a control flow statement or a data manipulation statement. Consecutive data manipulation statements form a block, which is assigned a number. When a control flow statement is encountered, the previous block of statements is considered closed. The next data manipulation statement starts a new block, which is assigned the next consecutive number. The parsing unit keeps track of which blocks to jump to and the conditions for jumping when the end of a block is encountered. This list of jumps and conditions for jumping constitutes the control flow graph. A simple interface module was written to translate this list to a finite state machine description.

When processing the data manipulation statements, the parser creates a parse tree of the statement. Fig. 4.4 shows how the parser assembles the parse trees into a larger structure. Parse trees belonging to the same machine cycle are linked together in a linked list. The linked list is then attached to a cycle node. Cycles are linked to form blocks. The final list is sent to the mapper unit.

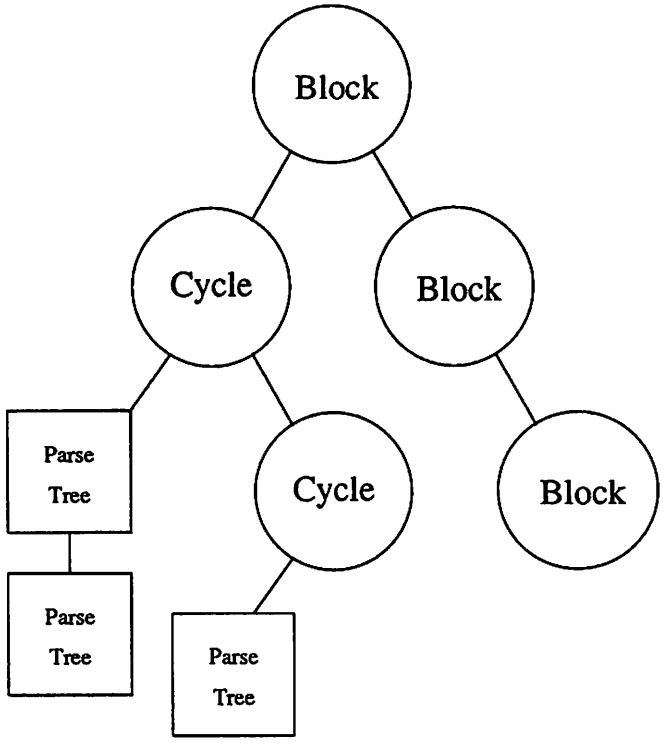


Figure 4.4: Data structure used in the parser.

4.3 The Mapping Unit.

The mapping unit deduces the control signals necessary for a register transfer given the structure master view of a processor. The mapping unit consists of the mapper module, the initializer module, and some interface modules. The initializer module determines the most general configuration for the processor control word. The interface modules translate the list of control words produced by the mapper to finite state machine tables.

4.3.1 Description and Theory of the Mapping Algorithm.

Three terms will be defined to avoid confusion in the ensuing discussion. Data flow will be used to refer to the classical data flow paradigm, which uses data flow graphs to represent algorithms. The term data path will refer to the actual physical path that data travels through when a register transfer occurs. Datapath, a single word, will refer to the hardware for evaluating arithmetic and logic operations in a processor.

The datapaths in a system are representative of data flow graphs. The legal data

flow can be configured by applying different combinations of control signals. During a given machine cycle, the datapath for a synchronous machine is analogous to a partitioned section excised from a larger data flow graph. Within the data flow graph, each hardware cell is represented by a node, while the edges entering and exiting the node represent the data or signals entering or leaving a cell. The firing of a node is accomplished by applying the correct control signals during a given machine cycle. With this interpretation, the data flow graph for one machine cycle can be described as a series of data transfers from edge to edge.

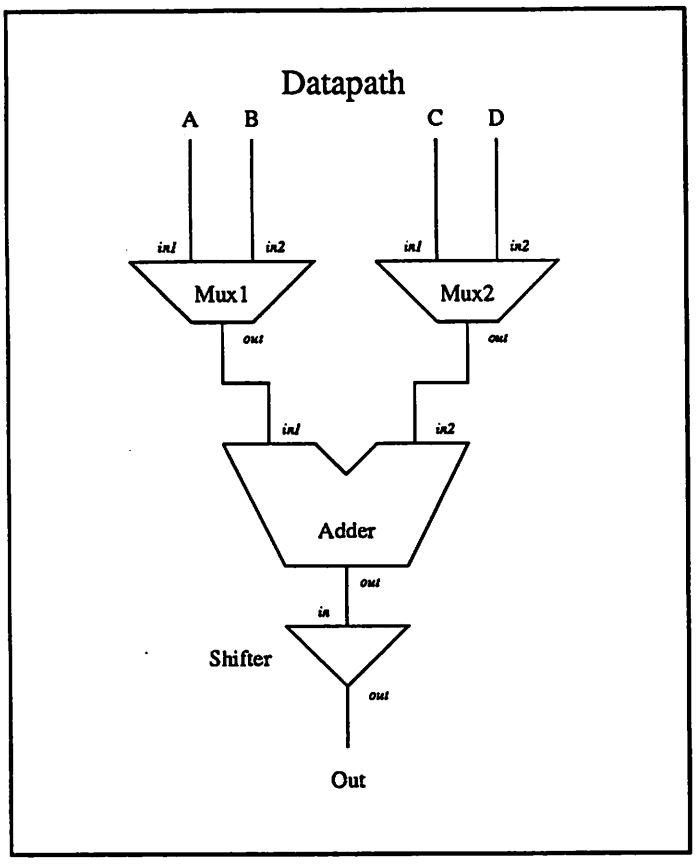
To illustrate this concept, a sample datapath is shown in Fig. 4.5. A data flow graph of a possible instruction is also shown in the figure. The 5 nodes in the graph are 2 multiplexers, an adder, a shifter, and a node that can supply the constant 2. A one to one mapping from the data flow graph to the datapath can be found. Using the table of control signals for the datapath blocks, we can determine the set of control signals that will achieve the operation described in the data flow graph.

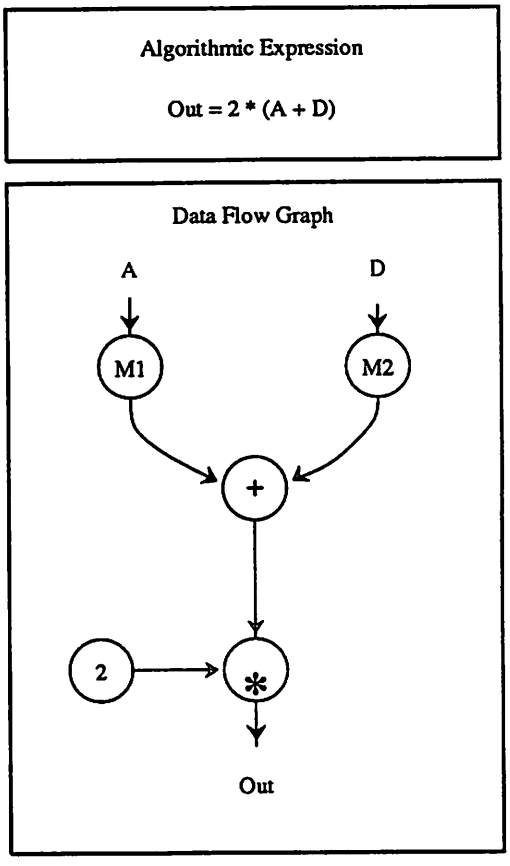
A mapping algorithm was developed to perform the mapping of register transfers as described in the last paragraph. In a given transfer, it is assumed that all data flows together to a single destination. Although several registers may serve as sources for the data, the final result must appear in a single register or memory location. With this assumption, a backward search starting at the destination will locate the correct path in which data must flow. This is why the architecture model must assume that all blocks have a single output terminal.

A depth first search was adopted for this algorithm. A depth first search returns the first solution it encounters, which may not be the optimal solution. Due to the nature of the problem, a register transfer usually only has a single mapping on a custom architecture so the the search should converge to the optimal solution.

This backward search will proceed through all the datapath cells, without regard to the function of the cell. The correct backward search termination condition occurs whenever a source operand register or memory is encountered. Since the search can only establish the path for data flow, a forward trace is needed to allow checking of the datapath cell functions. During the forward trace, the control signals for the register transfer are established by examining the behavior view.

The algorithm is organized in a sequential manner. The backward search proceeds by examining one source operand at a time. Once a source operand is matched, the forward





Control Signal Tables

element	Data Transfer	Control Signal
Mux	out = in1	CNT = Hi
	out = in2	CNT = Lo
Adder	out = in1 + in2	no CNT signal
Shifter	out = in	CNT = Hi
	out = 2 * in	CNT = Lo

Control Signals Needed to Implement Expression

M1:CNT = Hi

M2:CNT = Lo

Shifter:CNT = Lo

Figure 4.5: Mapping an algorithmic expression to control signals.

trace begins and continues until a cell with 2 inputs is encountered. At this point the backward search for a second operand ensues. Thus, the backward search and forward trace are interleaved. This interleaving requires that specific conditions to be defined for handling the failure of either forward or backward searches. These rules guide the algorithm so that it converges. Success of the algorithm occurs if the forward trace reaches the root node of the backward search and no source operands remain. At the conclusion of the algorithm, the necessary control word has been created.

4.3.2 The Implementation of the Mapper Module.

The actual implementation makes use of a stack, which is necessary to maintain the state of the search. Information about all cells previously visited in the current backward search must be in the stack, as well as indicators of the cells remaining to be searched. A stack allows a non recursive implementation that was easier to debug and required less memory.

The backward search of signal flow in the structure master view takes place by following the nets connecting terminals on cell blocks. For cells linked by nets within the same facet, this involves getting a terminal, then getting a net, and then locating a new terminal on the net which is connected to the next subcell. The search only proceeds along terminals that have been declared with the TERMTYPE property DATA_SIGNAL. The DIRECTION property of the terminals is examined to insure that the trace proceeds from the input terminal of the current cell to the output terminal of the new cell. However, if the trace involves cells on 2 different levels of the hierarchy, then several nets and the formal terminals of the instances are involved. Separate net tracing algorithms must be applied.

The backward search establishes the input terminal and the output terminal necessary for data transfer to occur through the node. This is part of the information stored in the stack during the search. The search makes use of the formal parameter MODULE_TYPE, which was added to the structure master view for this project. The MODULE_TYPE of the cell determines the general type of function the cell can perform. To terminate a backward search, the MODULE_TYPE is examined. If it corresponds to a memory, the name from the structure master view is compared to the name of the operand being searched for. If it matches, the backward search ends successfully.

The forward trace makes decisions at each cell to determine the correct control

In this case, once a register output enable is set, the control bits for the entire address field are set. The scheme to detect conflicts will then work correctly.

4.3.3 The initializer module.

In the LagerIII Kappa processor, the control word was optimized to minimize routing and area. With only the assumption of controller structure, the rest of the processor elements will define the control word. In addition, the control word is influenced by the type of looping and jumping that was implemented in the controller. Since the control word is not predefined, the initializer module must make a reasonable guess at an efficient word width and structure.

Several complications arise. The controller architecture is general and will support both horizontal and vertical control words. Horizontal control words allow faster communication, but take up more area since the control store is wide. Vertical control words take up less space, but can require complex instruction decoders and possibly pipeline delays.

The simplest solution is to have the initializer module find the most primitive horizontal control word and apply optimization later on. This was the approach adopted for this project because it allows the user to experiment with different control word sizes using post processors. Once the problem is more fully understood, optimization programs can be incorporated into the microcode generator to optimize the control word for the given application.

The initializer works as follows. The initializer examines every cell in the design hierarchy. For every cell, the initializer looks at each cell terminal to decide if it is associated with a control signal. Any terminal associated with a control signal must have TERMTYPE property CONTROL_SIGNAL. The initializer allocates a bit space in the control word for all terminals that have this property. A post processor is provided so a user can reorder this initial assignment, or delete signals that are not needed.

4.3.4 The interface modules.

As with the parser unit, the interface modules for the mapper convert the information from the mapper unit into programmable logic array (pla) descriptions. For both the parser and mapper units, the output files are in the *kiss* style format accepted by the optimization program *espresso*. In addition to the 2 files in *kiss* style format, a third file is

signals. The forward trace begins at the last correctly matched source operand. A parallel forward trace is performed on the parse tree of the operation being mapped. The control signals are determined by examining the current parse tree node and the behavioral view. The parse tree created by the parser shows the necessary operation to be implemented. Based on the current node in the parse tree, queries are made to determine the necessary control signals. Several C routines were written to process this information and determine which function a cell must perform. In essence, the routines form a mini expert system with hard coded rules for mapping operators to cells. To add new behavior primitives, new C routines will have to be added.

Once the desired function of the cell has been determined, a query to the behavior view will return the necessary control signals. Thus a behavior view must exist for every leafcell that requires a control signal. The behavior view can be accessed by examining the behavior view with the same leafcell name as the current cell being examined in the structure master view. Example behavior descriptions can be found in Appendix B. The information from this behavior description language (bdl) file is stored in the behavior view.

As the search proceeds, the control signals are collected in a control word. The configuration of the control word is determined by the initializer module, which will be discussed in the next section. A successful search and trace causes the control word to be passed to the interface modules for processing. If the search fails, the current operation cannot be mapped onto the datapath and the program terminates.

Since parallel register transfers are allowed, a scheme to detect resource allocation conflicts was implemented. During initialization, a control word format is created. Prior to generating the microcode for each cycle, the control word is initialized. Each time a control signal is added to the control word, the correct position for the control bit is established and a check is made to see what the value is at that position. If the value is unset, then the control signal is placed there. If the value is set, a check is made to see if the set value is the same as the value of the control signal to be added. If it is, no conflict has occurred and the mapper continues. Otherwise there is a resource allocation problem and the mapper notifies the user and aborts processing.

This simple scheme for resource allocation conflicts has a problem. If the control signals for the register files are assigned individually, then the scheme fails. Since the control bits are checked when they are set, it is possible to have 2 registers on the same bus with their outputs enabled. One way to avoid this is to use an address field for the register file.

created by an interface module. This third file contains the parameters necessary for the generation of the controller.

4.4 Summary.

The microcode generator implemented in this project consists of a parser unit and a mapper unit. The parser accepts a C like program description and creates the control finite state machine description from it. The mapper unit uses tracing algorithms to produce the microcode for the control store. The next chapter provides a detailed example for the detailed use of the generator.

Chapter 5

Use of the Prototype Microcode Generator

This chapter will provide a tutorial example on use of the microcode generator. A user's manual can be found in Appendix A. The example comes from the fingerprint filtering system [1].

5.1 The Design Process Using the Microcode Generator.

Fig. 5.1 shows a flowchart depicting the use of the microcode generator in the design process. In the first step, an algorithm is analyzed and a processor architecture is proposed. Concurrently, the preliminary sdl files describing the processor and the C like algorithm description are written. When the sdl files are complete, the structure master view is created using DMoct. After the algorithm is coded, it can be compiled with the C compiler and debugged. Next, the program and the smv can be submitted to the microcode generator, which creates the parameter files. If the results are satisfactory, the structure instance view and the layout can be generated using DMoct.

5.2 An Address Calculation Unit Example.

A segment of code from the orientation algorithm of the fingerprint filter will be mapped onto an address calculation unit. Extra detail about the design of the orientation processor can be found in reference [1].

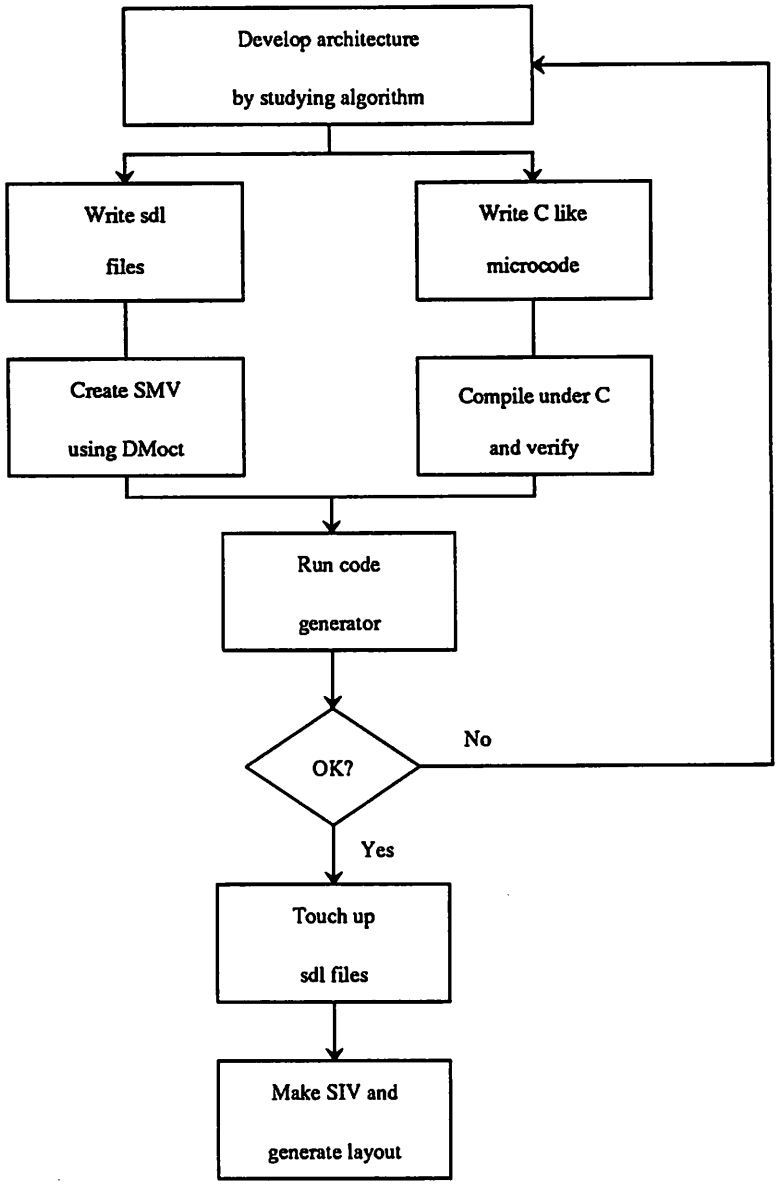


Figure 5.1: Flowchart for use of the microcode generator.

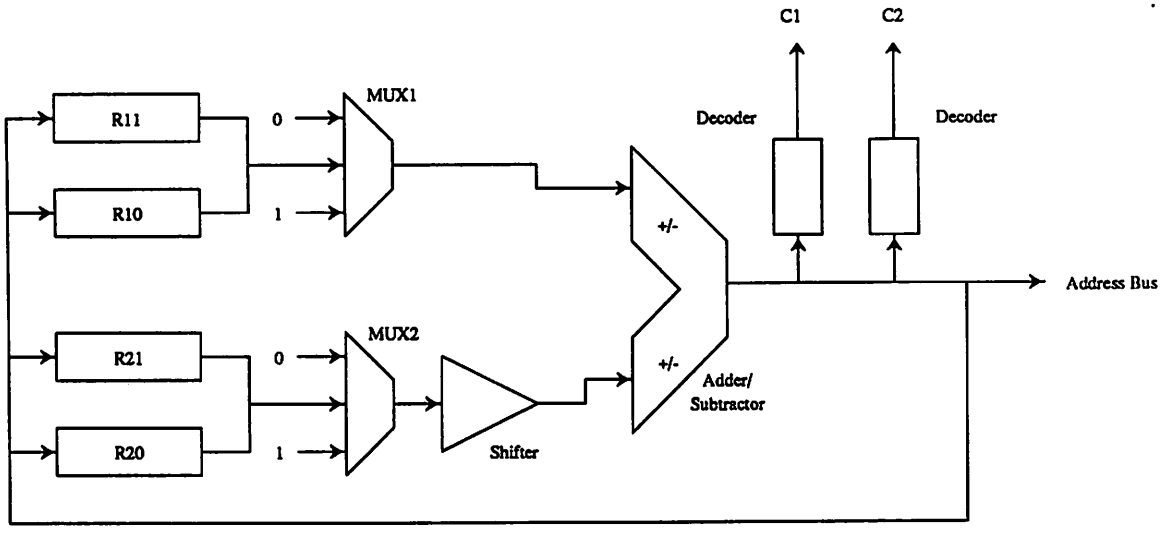


Figure 5.2: The address calculation unit datapath.

5.2.1 Examining the Algorithm.

The fingerprint image is stored in a 512 by 512 pixel array. The filtering operations only take place on the interior 504 by 504 pixel image. The algorithm causes the 4 pixel deep border to be assigned a background value of 0. The algorithm is illustrated below.

1. Write a zero to every pixel in the first 4 rows.
2. Write zeroes into the bottom 4 rows of the array.
3. Write zeroes into the first 4 columns of rows 5 to 507.
4. Write zeroes into the last 4 columns of rows 5 to 507.

The algorithm will require some jumping and nested looping. The controller model adopted by the microcode generator will provide support for this.

5.2.2 Defining the System Architecture

By analyzing the algorithm and consulting the Lager cell library, a datapath can be designed to handle the calculations. The datapath is shown in fig. 5.2. This datapath has more hardware than is necessary to implement the algorithm stated above because the complete orient algorithm requires it.

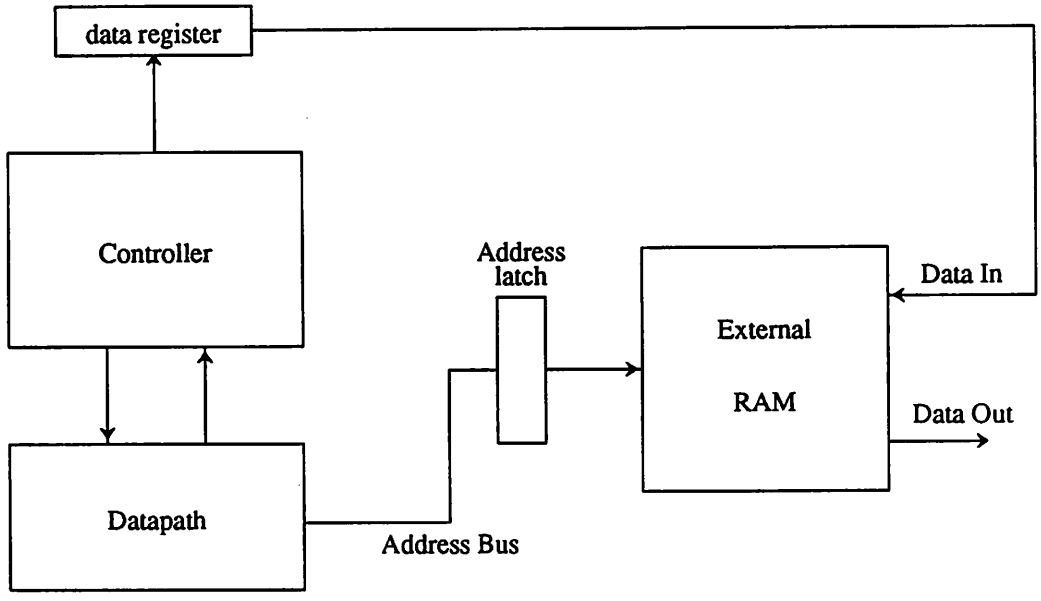


Figure 5.3: System block diagram.

In the initial design, the adder/subtractor is represented as a single cell. In reality, it is made up of an adder with 2 controlled inverters at each of the inputs. While it is possible to write tiling procedures to designate this cell combination as a leafcell, this description would be redundant and would clutter up the cell library. For microcode generation, it is simpler to consider it as a single cell since the adder/subtractor behavior is attributed to only 1 cell instead of being spread over 3. In the future, a structure processor might be used to substitute the 3 cells for the single cell designation during creation of the structure instance view.

The system block diagram for the processor is shown in fig. 5.3. Each block, even if it will eventually be external to the chip, must have an sdl description. So in this case, we will have an sdl file for the datapath and the ram. In the actual orient processor, the controller also controls a second datapath but for this example, the only necessary part from this datapath is a register to supply the zero to be written into the memory. It is named *data* in the figure. A separate sdl file was also written to account for this register.

Special attention needs to be given to sdl files written for the cells at the block level. Although these files may contain hierarchy, the mapper module was written to ignore some of it. Appendix B contains the sdl files for the RAM and datapath blocks shown in fig. 5.3. Note that the RAM sdl file contains TERMTYPE properties and the MOD-

ULE_TYPE is MEM for memory. The datapath sdl file has MODULE_TYPE DATAPATH, but TERMTYPE properties are only assigned to the terminals attached to the parent. The discrepancy here is due to the fact that for a datapath, the mapper needs to look at the leafcells with the behavior view descriptions. The MODULE_TYPE DATAPATH tells the mapper to go down in the hierarchy.

In addition to the sdl files at the block level, bdl files must also exist. For every cell used in the datapath, a corresponding bdl file should exist. Also, in this case, a bdl file should be written for the ram. Example bdl files are given in Appendix B.

Once the block level sdl files are written, a final sdl file describing the system architecture is written. The microcode generator is only concerned with the data flow in the system. There is no need to declare the controller at this level, so it will be left out. Only the data nets connecting data terminals need to be declared. The nets which eventually will connect the control terminals of subcells to the controller can be ignored for now.

The sdl files for this example can be found in Appendix B. The system architecture is defined, and the preliminary sdl files have been created. The user must now run DMoct with the -m option to create the structure master view needed by the mapper. In addition, the bdl files for all the necessary leaf cells should have been processed so that behavior views exist.

5.2.3 Algorithm coding.

Once the structure master view has been successfully created, the user can now code the algorithm into the pseudo C code accepted by the microcode generator. Appendix A contains a summary of the input syntax conventions. The coding must represent the algorithm in register transfers that will map onto the datapath previously created.

The coded program is shown in Appendix B. It consists of some declarations and the program body. The *int* declaration declares the variable name used in the program in the first field while the second field declares the name of the register or memory where the data will be stored. Note that the register name must be the same as the instantiated register name declared in the sdl files.

The code is nearly C compatible. By changing the integer declarations, the code can be compiled and executed. After the code is written, it should be compiled and verified.

5.2.4 Microcode Generation.

Once the user has tuned the microcode description and has successfully obtained a structure master view and the behavior view, the next step is to generate the parameters necessary for the layout generation of the controller. After placing the microcode description in the same directory as the structure master view, the microcode generator is invoked by typing *asm*. The program prompts the user for the program name. If parsing succeeds, the program prompts the user for the name of the structure master view. When the program succeeds, the successful ending prompt is returned.

If errors are encountered during parsing or mapping, the user needs to decide where the error occurred in order to correct it. The parsing errors can be deduced by using the -p flag and looking at the debug file called *parse.log*. The cause of the error can usually be determined from this file or from careful examination to make sure that all grammar rules were followed. If a mapping error occurs, attempt to trace the path, and see if the error can be detected. Errors in mapping can also occur if the behavior view is not correctly specified.

Upon successful execution of the microcode generator, several files are produced. The file *asm.tables* gives information about the symbols, state transitions, and flags that were translated from the program description. There are 2 files, *cfsm.esp* and *cstore.out*, that are written in *kiss* format. The *cstore.out* file can be used with the file *bit.data* as inputs to the post processor, which will be discussed in the next section. In addition, if debugging flags were specified, several log files will also be produced. The last file produced is called *pcu.parval*. This file contains the necessary parameters for the generation of the control unit. The actual files from this example are displayed in Appendix B.

5.2.5 Post processing.

At this stage, the user can decide how to order the control signals in the control store finite state machine. The file *bit.data* gives the configuration found by the microcode generator. By deleting or interchanging lines, the user can set up a specified configuration.

Once the *bit.data* file has been edited, the program *post* is run. *Post* specifically looks for the files *bit.data* and *cstore.out*. It reorders or deletes the fields in the espresso file. The resulting file is called *cstore.esp*. At this stage, the microcode generation is complete.

5.2.6 Layout

To complete the layout, the user must modify and complete the sdl files for the entire processor. The controller sdl files must be included. The file *pcu.parval* can be edited to add the extra parameters necessary for datapaths or other hardware. Once this is done, the Lager Design Manager can be invoked to complete the mask level layout.

5.3 The Fingerprint Smooth Processor, a Second Example.

The smooth processor performs smoothing of the orientation image produced by the orientation processor in the fingerprint filter. The design of the smooth processor is similar to the orientation processor [1]. Both processors make use of register files to avoid memory bandwidth problems associated with a single RAM. The main point of this example is to show how the microcode generator can be used for a full processor application.

The original microcode descriptions were written in C. After compiling the code and using it to filter some actual fingerprint images, many bugs were found. The microcode was written under a set of scheduling constraints which were imposed to eliminate simultaneous access of the shared memory was impossible. Because of the scheduling constraints and the bugs, the program had to be rewritten and rescheduled. During the rescheduling, a lifetime analysis for the program variables was performed. The number of registers required for the processor was reduced by one third. The final program was compiled and verified with actual fingerprint images.

Experiments with small segments of the code have shown that the size of the control store will be large. Register files require many control signals, so the size of the control store is linked to the size of the register files used in the processor. In the original design, the smooth processor required space for 37 variables in 2 register files. The lifetime analysis showed that 2 register files of 13 registers would be necessary. Work was performed to use address decoders to reduce the number of bits necessary to control the register files. In a future version of the smooth processor, the size of the register files could be significantly reduced by providing some dedicated hardware for nested looping.

Due to a problem in implementing a parser for the behavior description language, the work on the smooth processor was suspended.

Chapter 6

Evaluation and Conclusions.

6.1 Problems with the Current Implementation.

This project has shown that the Lager data structures in the OCT database can support retargetable microcode generation. A prototype retargetable microcode generated was presented. There are still many problems but it is clear that future work could provide viable retargetable code generators. Some of the problems will be discussed in the next few paragraphs.

The parser can be easily fooled. Because of the current parsing scheme, the parser can have trouble matching parentheses correctly. While it parses a subset of the C language, several additions to the syntax give clues to the parser on how to interpret control flow constructs. A description of the syntax is given in appendix A, and if it is followed, the results can be good. However, if something goes wrong, the error messages provided by the parser are vague. The parser was implemented quickly, so future work could focus on a better input language and a more versatile parser.

The mapper module works correctly, but it can run into problems if asked to map statements that are ambiguous. The mapper can usually deduce that single operand instructions imply that the value zero could be added to the operand as it passes through an adder. It is also possible for the mapper to get stuck somewhere in the depth first search. In most cases, the statement can be coded differently so the mapping succeeds.

6.2 Future Work.

The microcode generator is an unfinished product. While it can be used in its present state, the interfaces are primitive. It hasn't been integrated into the Lager system. Possibly, the microcode generator could be linked to the Lager III compiler to facilitate the mapping of high level descriptions to new architectures. The easiest way to link it would be to use the mapper unit to create the structural architecture description that currently must be written by hand.

Work still needs to be performed on refining the behavior description. Assigning operations to cells limits their flexibility. It is desirable to have higher level operations map to single cells, but it may be more efficient to combine several cells to perform a higher level operation. Perhaps a better approach would be to provide support for a behavior view that is based on hierarchy. Additionally, the behavior view should be examined and modified so that it can support the information needed by synthesis systems.

While this approach to retargetable microcode generation was shown to work, the large amount of searching makes it inefficient. Other approaches should be considered. Two separate approaches will be mentioned.

One alternative is to enhance the behavior view to support hierarchy. In this scheme, behavior views would be created at higher levels of hierarchy. At the datapath level, the behavior view would contain higher level operations matched to sets of control signals. At the highest level of hierarchy, the behavior view would contain the processor instruction set. This approach is a more elegant version of the sadl file implementation used in Lager III. With this approach, it may be possible to start the Design Manager from the behavior view level instead of the structure master view level. A structure processor might be developed that could create a structure master view from a behavior view.

The second approach would be to create a tool that facilitates the process of defining an instruction set. This tool would allow the instruction set to be declared mnemonically in a file. The mapper tool would then create a file with the correspondence between an instruction and the microcode. A table look up algorithm could then be applied to generate the microcode from an input program. This would greatly increase the efficiency of the code generator.

Future work will probably move away from the microcode generation problem. Many researchers are pursuing synthesis systems that can take a high level behavior descrip-

tion, and synthesize a system to perform a specified task. These compilers will undoubtedly provide processors that are well designed and optimized under certain specifications. But until the synthesis problem is solved, tools like a retargetable microcode generator will be invaluable in developing new ASIC systems.

6.3 Conclusion.

The microcode generator developed in this project has met the stated objectives. The C like input allows program development in C using the standard C compiler. The code generator was applied to parts of the fingerprint filter processor designs to show functionality. The code generator was fully functional and retargetable for these cases. The microcode generator developed in this project is inefficient since the searching algorithms are applied so many times. Future work could improve the efficiency and provide more rigorous testing. The project has shown that a retargetable microcode generator is a possibility for structural silicon compilers. Moreover, the code generators can significantly decrease the design time and increase the ability of designers or design tools to explore the design space.

Bibliography

[1] F. Catthoor, L. O'Gorman, and R. Jain, *ASIC Architecture for Contour Line Filtering*, Proceedings of ICASSP, (New York), Apr. 1988.

[2] S. K. Azim, *Application of Silicon Compilation Techniques to a Robot Controller Design*, University of California, Berkeley (May 1988). Ph. D. Thesis.

[3] L. O'Gorman, and J. Nickerson, *Matched Filter Design for Fingerprint Image Enhancement*, Proceedings of ICASSP, (New York), Apr. 1988.

[4] G. A. Gibson, and Y. C. Liu, *Microcomputer Systems: The 8086/8088 Family, 2nd ed.*, Prentice-Hall, Englewood Cliffs, 1986.

[5] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Menlo Park, 1987, pg. 513.

[6] R. A. Mueller, *Automated Microcode Synthesis*, UMI Research Press, Ann Arbor, 1984.

[7] R. Jain, et al., *Custom Design of a VLSI PCM-FDM Transmultiplexer from System Specifications to Circuit Layout Using a Computer-Aided Design System*, IEEE Journal of Solid State Circuits, Vol. SC-21, no. 1, Feb. 1986.

[8] B. M. Pangrle, and D. D. Gajski, *'Design Tools for Intelligent Silicon Compilation'*, IEEE Trans. on CAD, Vol. CAD-6, No. 6, pp. 1098-1112, Nov. 1987.

[9] N. Park, and A. C. Parker, *Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications*, IEEE Trans. on CAD, Vol. CAD-7, No. 3, pp. 356-370, March, 1988.

[10] C. J. Tseng, R. S. Wei, S. G. Rothweiler, M. M. Tong, and A. K. Bose, *Bridge: A Behavioral Synthesis System for VLSI*, Proceedings of CICC, 1988.

[11] S. P. Pope, *Automatic Generation of Signal Processing Integrated Circuits*, University of California, Berkeley, (Feb. 1985), Ph. D. Thesis.

[12] J. M. Rabaey, S. P. Pope, and R. W. Brodersen, *An Integrated Automated Layout Generation System for DSP Circuits.*, IEEE Trans. on CAD, Vol. CAD-4, No. 3, July 1985.

[13] C. S. Shung, *An Integrated CAD System for Algorithm-Specific IC Design*, University of California, Berkeley (June 1988). Ph. D. Thesis.

[14] M. B. Srivastava, *Automatic Generation oc CMOS Datapaths in Lager Framework*, University of California, Berkeley, (June 1988), M. S. Report.

[15] K. Rimey, *Qualifying Examination Proposal: A Compiler for Horizontal Instruction Word Signal Processors*, University of California, Berkeley (April 1988).

[16] R. Spickelmier, ed., *Oct Tools Distribution 2.1*, Electronics Research Laboratory, University of California, Berkeley, 1988.

[17] A. W. Nagle, R. Cloutier, and A. C. Parker, *Synthesis of Hardware for the Control of Digital Systems*, IEEE Trans. on CAD, Vol. CAD-1, No. 4., Oct. 1982.

Appendix A

User's Manual

Note: This was intended to be a stand alone user's manual. Some of the information that appears repeats information already presented in the report.

A.1 Introduction

The assembler requires 2 input files: a C description of the algorithm and an sdl description of the data flow in the processor.

Previously in LagerIII, rassCG was used as the assembler tool. The rassCG program required the user to create a structural architecture description, which basically contained the same information found in the sdl file mixed with some behavioral description. It also required the user to write the program in the assembly language, which was defined in the structural architectural description file. There was no guarantee for the correctness of this algorithm.

The assembler described in this manual only requires the 2 files mentioned above. In addition, the parser was written in such a way that with only minor changes, the C description is still compatible with the standard C compiler.

A.1.1 A Discussion of the Assembler Applications and Limitations.

This assembler is meant to aid the rapid development of new architectures for custom processors. The assembler assumes that the controller is a modified version of the Lager Kappa controller. Target applications are algorithms with complex control flow, and lower sampling rates.

The prototype assembler has limited features. It is mainly intended to aid the design of a chip set for a fingerprint filter. However, the differences in architecture show that the tool may have potential for more general applications.

A.1.2 General Concepts of Implementation.

The assembler is implemented as a parser and a mapper. The parser serves to translate the C description to internal data structures and to extract the control flow graph. The mapper uses a tracing algorithm to determine the path of the register transfer implied

in the C code, and then retraces the path to determine the necessary control signals to implement the register transfer.

Parser Implementation.

The parser was written using Lex and Yacc. The main purpose of the project was to test out the retargetable assembler concept, so little attention was paid to writing a good error detecting parser. When errors occur, the parser usually doesn't provide helpful diagnostics. This is a general problem when using Yacc. In order to provide robust error handling, the Yacc description must contain matches for all anticipated errors and messages for handling these errors.

Most Yacc parsers are written by matching compound expressions. A variable name would form a simple expression, and this would be the lowest level of token matching in the Yacc file. An equation would become the second level of simple expressions. A group of expressions forms compound expressions and so on. The Yacc parser is built around this sequence of definitions.

In the attempt to extract control flow, the use of compound statements was abandoned. Instead, equations were matched, and control flow delimiters like while, if, and do were matched as terminators or beginning points for new blocks of sequential executed code. In this case, a terminating bracket should be associated with the last beginning point encountered. If brackets are misplace, the parser would have trouble recognizing this. This points out just one of the many problems with the parser.

If a future version of this tool is created, the parser should be developed with error handling in mind. It would be best to just parse the description in the classical manner, placing the results in a database or in internal memory. Not only would this allow better error handling and detection, but it would also help the detection of control flow, especially the cases where a block of code consists of a single instruction.

Mapper Implementation.

The real input to the assembler is not the sdl file, but the structural master view (smv) created by DMoct. The smv is stored in the OCT database format. If an sdl file is available, run DMoct with the -m flag to create the smv.

The basic mapping algorithm is to trace the actual path that will be taken by a signal. The tracing takes place in 2 operations. In the first operation, the destination and source memory locations are located in the smv. A depth first trace is used to locate the data flow path from destination to source. The assembler restricts all data transfers to the class where many source operands merge to one destination operand. The depth first search will locate the first viable solution. If the search does not converge, then the mapping is not possible.

The second operation is to trace the path from source to destination. Along the way, all blocks along the path are examined. The behavioral view is consulted to determine the correct operation to be performed by each block. The assembler accumulates the necessary control signals, and then prints them to a file. The output tables are in the espresso format.

Although the actual algorithm is more complex than this simple summary, the concepts may help you to understand why the assembler requires certain parameters like MODULE_TYPE and TERMTYPE.

A.2 SDL Descriptions.

The sdl description for data flow paths must have only 2 levels of hierarchy. By this I mean that there must be the top level description with macrocells like memories, datapaths, and controllers. The second level can contain description of datapaths using dpp, and other descriptions. Anything not on the data flow path can have more than 2 levels of hierarchy.

The datapath description must be written with register transfers in mind. The algorithm used to perform the mapping actually traces the data flow involved in a register transfer to determine the correct path. This means that a black box description of the external rams must be written and included in the processor sdl description.

All blocks must have a corresponding bdl description. This allows the assembler to determine which operations can be performed by each block. Currently, only the operations +, -, *, and / are recognized by the assembler. All other operations are considered to be data channeling operations, which are handled by multiplexers. Further, the * and / operators are associated with shifters. This limitation is caused by the current version of the assembler. A future version could eliminate this.

The assembler assumes that all blocks have a unique output, or only one output tied to an actual net. The blocks can have as many inputs as desired, but blocks must only have either one input, in which case it is considered a MUX, or 2 inputs producing one output, in which case it is of the class ADDER. Other block types supported are REG for registers, REGFILE for register files, and MEMORY for rams and other external memories.

The names are crucial in the sdl file. The names of registers and memories must correspond letter for letter with those used in the C description declarations. However, you must be sure to name the registers and memories at their lowest instantiation. Thus, if you want a reg2port, then instantiate that with the variable name.

It is also important that the properties TERMTYPE and DIRECTION are added to every file below the top level file. However, only do it for the data terminals on the parent! Doing it for the control signals will cause them to be added to the control word. Thus for control signals, their TERMTYPE (which should be CONTROL_SIGNAL) and DIRECTION should be declared at the lowest level of the hierarchy.

Be careful of feedback loops. Make sure that a closed circular path always has at least one memory, register, or regfile inside the loop. If not, the tracing algorithms will enter infinite loops.

The sdl description need not contain the controller. The purpose of the assembler is to generate the necessary parameters for the controller. The controller can be generated from a group of sdl files that aren't quite debugged yet. The top level file is called pcuH.sdl, which will probably change, so it won't be confused with the Kappa controller.

A.3 C Language Description File.

The assembler accepts a C language description for the algorithm to be implemented. However, the description must be written in a subset of the C language. The parser defines the acceptable syntax. It was written with the help of the Unix utilities Lex and Yacc, so if you are familiar with reading Yacc files, this is the best way to see what the actual accepted syntax is.

One feature of the assembler syntax is the use of double semicolons to delimit the end of a machine cycle. Statements that occur concurrently in one cycle are written like in the example below.

$$y = b + x;$$

$$a = c + d;;$$

This implies that 2 additions can take place in the same cycle. This may be possible, for instance, if a were an address being calculated in an address calculation unit, and y were held in a register in the main ALU datapath.

The double semicolon feature made parsing complex. Although the assembler will often catch misplaced semicolons, the error messages are not very informative. If the assembler crashes, this is one area to check.

A.4 Variable, Flag, and Constant Declarations.

A.4.1 Constant Declarations.

There are 3 types of variables: register, register file, and memory.

A.4.2 Flag Declarations.

All flags in the program need to be declared. In C, they would be declared as integers, but the assembler recognizes the keyword flag. There are 2 types of flags: those associated with sign bits from adders, and those that are other types. The other types can be a constant decoder, or some logic. The declaration is shown below.

```
flag sign.ADDER, notdone;
```

The flag sign should be taken from the block called ADDER defined in the sdl file. The flag notdone will be assigned a value if it is from a constant decoder. Otherwise, it is given no special type.

A.4.3 Variable Declarations.

Register Variable Declarations.

Register variables are those that reside in a single register. The block described in the sdl file will have MODULE_TYPE REG. They are declared as shown.

```
int variable.sdlname;
```

The field variable is the variable name used in the program. The field sdlname is the name given to the instance of the register in the sdl file.

Constants are declared much like in C. It would have been nice to use the c preprocessor, but that can come later. A constant is defined as follows.

```
#define KONSTANT 16
```

Constants are used as multiply or divide coefficients. They are also used to specify the value of the constants stored in the constant decoders.

Memory Variable Declarations.

Memory variable declarations assume that data is stored in memory like a one dimensional array. The declaration looks like this:

```
int ramname[size];
```

The string ramname must be the name of the instance of the memory in the sdl file. Size is not parsed. It is declared so the description maintains compatibility with standard C. The reference to the array has the same effect. It is the ramname that matters not the index. It is assumed that the address for the array was calculated in a previous cycle and latched into an address register for the ram. Immediate addressing is provided, but that will be found under register file declarations.

Register File Declarations.

Currently, a form of immediate addressing is supported, provided the module is given MODULE_TYPE REGFILE. It requires specialized declarations.

```
memsize regf[3];
memloc A.regf[0], B.regf[1], C.regf[2];
```

These 2 lines completely declare the immediate addressing mode. What happens is a field in the control word is set aside for the register file regf, and it will be 2 bits long, since log2(3) + 1 = 2. The program variable A will always have address 0 in the field and so on for B and C. The indexes must be manifest constants, not declared constants. If you desired to have a ram with mapped memory, then write a fake sdl file for it, and give it MODULE_TYPE REGFILE.

A.5 Assembler Syntax Conventions.

A.5.1 While and Do While loops.

The assembler syntax also supports looping. While and do while loops are supported. However, the controller architecture has a one cycle pipeline delay, so the test for exit condition must be hand scheduled so that it appears 2 cycles before the actual exit test in the code is written. An example follows.

```
while ( sign != TEN )
      {
      y = b + x;
      a = c + d;;
>>>>>  if (y - yMax < 0) sign = 1; else sign = 0;;  <<<<<
      y = y - 1;;
      }
```

The arrows point to the exit condition test. It is scheduled so the sign bit is presented to the controller at the correct time, 2 cycles before the might may be exited.

A.5.2 Flags and Timing Conventions.

Flags for conditions must be determined by the user. Any flag is ok as long as it is generated at the correct time. The adder sign bit can be used. Constant decoders can be used to set flags when a number on the datapath matches the number hard coded in the decoder. Decoders can be tricky to use, because the number to be decoded must be dumped onto the datapath 2 cycles before the flag is to used. The best way to achieve this is to calculate the number during the correct cycle, or transfer the number from its register to a temporary register.

If scheduling constraints won't allow the flag to appear on the correct cycle, delays can be used so that the delayed flag occurs on the correct cycle.

A.5.3 If Else Statements.

The if else, and if else if ect. constructs are supported. However, several syntax assumptions were made. We can divide the if statement into 3 classes. The first class is the simple test that sets a flag. Another is a 2 way branch that merges again. The last class is the multiway branch using the if else if construct.

Simple If Else.

The if else construct to set a flag looks like this:

```
if ( x - y < 0 ) flag = 1; else flag = 0;  or
if ( x - y < 0 ) flag = 1;                 or
if (!(x - y < 0)) flag = 1; else flag = 0; or
if (edge != NDIRN1) flag = 1; else flag = 0;
```

All these are examples of legal statements. Inequality is only supported as sign bits from adders. Thus, you may only compare to 0. The ! negation operator is supported to some extent, but it isn't as robust as in C. If the flag is associated with an undelayed sign bit, the flag must be declared with an adder notation (read the declaration section). The if statement does not have to be a single machine cycle. It can be part of a concurrent statement. If only a single flag is given, the else condition will assume it's complement.

If Else as a Branch.

A branch will assume that a choice can be made between blocks of executable code. The if else in this compiler assumes that the 2 paths merge. It may support an if with no else clause, but this hasn't been tested. It wasn't necessary for the present application. An example is shown below.

```
if ( sign )
     {
     x = y + z;;
     y = y + 1;;
     }
else
     {
     x = y;;
     y = y - a;;
     };   <<<<<
```

The arrows point to a very vital feature of the assembler. All if initiated branches must be terminated by };. If you forget this, the assembler does very weird things, and it may not give you an error!! Also note that the branch must be initiated off a flag value. The flag must be set up 2 cycles before the actual branch.

If Else If Multiway Branch.

The if else if construct is similar to the if else.

```
if ( sign && p != TEN)
    {
    x = y + z;;
    y = y + 1;;
    }
else if ( !sign && p == TEN)
    {
    x = y;;
    y = y - 1;;
    }
else if (!sign && p != TEN)
    {
    y = x;;
    y = y + 1;;
    }; <<<<<
```

The && (and) operator is supported. The —— (or) operator is parsed correctly, but the actual mapping for it has not been implemented. It requires some minor changes in the way espresso files are generated. A general problem in using multiway branches is

getting both flags set 2 cycles before the jump is to take place. One method is to delay one signal and evaluate the other at the correct time. Also, never terminate this type of if with a plain else clause. Always use the else if. The plain else clause can be difficult to implement because you must know which conditions weren't met. Again the arrows point to the only way to successfully terminate the else if structure.

A.5.4 Loops and jumping into and out of them.

Leaving loops. When you leave loops or branches, you must always jump to a block of code, never to another branch. This simplifies the state transitions, since states tend to multiply rapidly. Take the example below.

This code would be numbered like this:

```
                        leave block 0 (the reset state)
if (sign)
    {
    y = y + 1;;         block 1
    }
else
    {
    y = x;;             block 2
    };
if (p = NDIRN1)
    {
    x = z;;             block 3
    }
else
    {
    x = x + 1;;         block 4
    };
```

The required transitions are 0 to 1, 0 to 2, 1 to 3, 1 to 4, 2 to 3, 2 to 4. With a single pass assembler, it is difficult to determine this. Thus, it was not allowed for this prototype version.

A.6 Subroutines.

A single subroutine is allowed. It must be declared by the keyword subr(). It can be called as many times as you like. However, the assembler expects the main program, and then the subroutine after it. If it is the other way, it won't work, and you'll get that ambiguous message "syntax error". The jump before and the jump out of the subroutine must occur from a single block. Jumps from multiway branches will not be processed correctly.

A.7 Assembler Directive STACK

The assembler directive STACK causes a stack to be used in the processor. A stack may be advantageous if there are quite a few blocks of code in the subroutine, and/or the subroutine is called many times. The reason for this is that if no stack is used, the state machine will just use new states for the subroutine, guaranteeing a return to the proper state. A stack will only require 2 extra states per subroutine call. The directive STACK must appear with the rest of the program declarations.

If a stack is used, then be aware that an extra nop cycle is encountered in the return jump. Refer to Khalid Azim's thesis for the details. An extra state called Dummy is inserted.

A.7.1 Multiplies and Divides.

All multiplies and adds are assumed to be implemented via shifts. This means one of the operands must be a constant, and should be a power of 2. The constant coefficient must be declared using the #define declaration. An expression like a*b involving 2 variables will cause the assembler to crash. Also, an expression like a*16 will also crash.

A.8 Other Topics.

A.8.1 NOP - the no operation.

NOP is supported. To make the C compiler skip a NOP, just define NOP as a semicolon. The actual implementation for a NOP causes all blocks to default to the bdl description for NOP.

A.8.2 Output Files

Currently, the assembler outputs all kinds of files. "pcu.parval" contains the parameters needed by pcuH.sdl. "cfsm.esp" contains the espresso input file for the control finite state machine. "cstore.out" contains the espresso file that must be run through the post processor. "asm.tables" contains information about the assembly process like symbol tables, the state transition table, and the flags used in the program.

The file "bit.info" tells what each field of the control word corresponds to. By rearranging the lines the user can reorder or delete redundant or unnecessary control fields. This file and "cstore.out" become inputs to the post processor, which reorders the espresso file as the user specified when altering "bit.info".

The espresso files are set up for plagen. The right most bit of the input and output planes gets mapped to bit 0 of the pla. In "cstore.out", the EOB signal should always be in bit 0, the right most bit. Bit info numbers the fields with respect to this convention.

Debugging files can be obtained by using the asm invocation with flags specified. The -p flag gives the "parser.log" file, with all the debugging information from the parser. The -n flag gives all information from the initialization after reading the smv. The -t flag gives all information according to the terminal tracing algorithm.

getting both flags set 2 cycles before the jump is to take place. One method is to delay one signal and evaluate the other at the correct time. Also, never terminate this type of if with a plain else clause. Always use the else if. The plain else clause can be difficult to implement because you must know which conditions weren't met. Again the arrows point to the only way to successfully terminate the else if structure.

A.5.4 Loops and jumping into and out of them.

Leaving loops. When you leave loops or branches, you must always jump to a block of code, never to another branch. This simplifies the state transitions, since states tend to multiply rapidly. Take the example below.

This code would be numbered like this:

```
                        leave block 0 (the reset state)
if (sign)
    {
    y = y + 1;;         block 1
    }
else
    {
    y = x;;             block 2
    };
if (p = NDIRN1)
    {
    x = z;;             block 3
    }
else
    {
    x = x + 1;;         block 4
    };
```

The required transitions are 0 to 1, 0 to 2, 1 to 3, 1 to 4, 2 to 3, 2 to 4. With a single pass assembler, it is difficult to determine this. Thus, it was not allowed for this prototype version.

A.6 Subroutines.

A single subroutine is allowed. It must be declared by the keyword subr(). It can be called as many times as you like. However, the assembler expects the main program, and then the subroutine after it. If it is the other way, it won't work, and you'll get that ambiguous message "syntax error". The jump before and the jump out of the subroutine must occur from a single block. Jumps from multiway branches will not be processed correctly.

A.7 Assembler Directive STACK

The assembler directive STACK causes a stack to be used in the processor. A stack may be advantageous if there are quite a few blocks of code in the subroutine, and/or the subroutine is called many times. The reason for this is that if no stack is used, the state machine will just use new states for the subroutine, guaranteeing a return to the proper state. A stack will only require 2 extra states per subroutine call. The directive STACK must appear with the rest of the program declarations.

If a stack is used, then be aware that an extra nop cycle is encountered in the return jump. Refer to Khalid Azim's thesis for the details. An extra state called Dummy is inserted.

A.7.1 Multiplies and Divides.

All multiplies and adds are assumed to be implemented via shifts. This means one of the operands must be a constant, and should be a power of 2. The constant coefficient must be declared using the #define declaration. An expression like a*b involving 2 variables will cause the assembler to crash. Also, an expression like a*16 will also crash.

A.8 Other Topics.

A.8.1 NOP - the no operation.

NOP is supported. To make the C compiler skip a NOP, just define NOP as a semicolon. The actual implementation for a NOP causes all blocks to default to the bdl description for NOP.

A.8.2 Output Files

Currently, the assembler outputs all kinds of files. "pcu.parval" contains the parameters needed by pcuH.sdl. "cfsm.esp" contains the espresso input file for the control finite state machine. "cstore.out" contains the espresso file that must be run through the post processor. "asm.tables" contains information about the assembly process like symbol tables, the state transition table, and the flags used in the program.

The file "bit.info" tells what each field of the control word corresponds to. By rearranging the lines the user can reorder or delete redundant or unnecessary control fields. This file and "cstore.out" become inputs to the post processor, which reorders the espresso file as the user specified when altering "bit.info".

The espresso files are set up for plagen. The right most bit of the input and output planes gets mapped to bit 0 of the pla. In "cstore.out", the EOB signal should always be in bit 0, the right most bit. Bit info numbers the fields with respect to this convention.

Debugging files can be obtained by using the asm invocation with flags specified. The -p flag gives the "parser.log" file, with all the debugging information from the parser. The -n flag gives all information from the initialization after reading the smv. The -t flag gives all information according to the terminal tracing algorithm.

A.9 Post Processor.

The post processor is called post. It requires "cstore.out" and "bit.info" as inputs. It will delete or rearrange the control signal fields given the edited "bit.info" file from the assembler.

Appendix B

Example Files for the Address Calculation Unit Example.

The contents of this appendix document the example presented in 5 for the orient address calculation unit.

B.1 Input Microcode Description.

```
#define RIDGE 4
#define IMGSIZ 512
#define IMR2 508
#define SIZE 262144
int yImg.R20, xImg.R10, dirnImg[SIZE], RIDGESIZ.C13, IMR.C12;
int address.latch;
int zero.imreg;

main() {
yImg = 0;
xImg = 0;;
while (yImg != RIDGE)  {
address = xImg + yImg*IMGSIZ;;
while (xImg != IMGSIZ)  {
dirnImg[address] = zero;
xImg= xImg+1;;
address = xImg + yImg*IMGSIZ;;
}
yImg= yImg+1;;
xImg = 0;;
}

/* Note: xImg is still 0 here. */
yImg = IMR;;
```

```
while (yImg != IMGSIZ)  {
address = xImg + yImg*IMGSIZ;;
while (xImg != IMGSIZ)  {
dirnImg[address] = zero;
xImg= xImg+1;;
address = xImg + yImg*IMGSIZ;;
}
yImg= yImg+1;;
xImg = 0;;
}

yImg = RIDGESIZ + 1;;
/* Note: xImg is still 0 here. */
while (yImg != IMR2)  {
address = xImg + yImg*IMGSIZ;;
while (xImg != RIDGE)  {
dirnImg[address] = zero;
xImg= xImg+1;;
address = xImg + yImg*IMGSIZ;;
}
yImg= yImg+1;;
xImg = 0;;
}

yImg = RIDGESIZ + 1;;
xImg = IMR;;
while (yImg != IMR2) {
address = xImg + yImg*IMGSIZ;;
while (xImg != RIDGE) {
dirnImg[address] = zero;
xImg = xImg + 1;;
address = xImg + yImg*IMGSIZ;;
}
yImg = yImg + 1;;
xImg = xImg + 1;;
}
NOP;;
/* Test subtraction */
xImg = yImg - 1;;

}
```

B.2 SDL Description of the Address Calculation Unit.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  This is the system level sdl description of the Orient ACU example.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(parent-cell system)
(parameters N N2 NDIRN1 HALFLT IMR R RM1 IMRM1 IMGEDG)
(layout-generator Flint)
(subcells
  (oacudp DP ((N N) (NDIRN1 NDIRN1) (HALFLT HALFLT) (IMR IMR) (R R)
    (RM1 RM1) (IMRM1 IMRM1) (IMGEDG IMGEDG)))
  (genericmem dirnImg ((ADDWIDTH N) (DATWIDTH N2)))
  (data   DAT ((N N2))))

;Data Nets
(net connect (NETWIDTH N) ((DP ADRBUS) (dirnImg ADDRESS)))
(net tozero (NETWIDTH N2) ((dirnImg DATAIN) (DAT OUT)))

(end-sdl)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  This is the sdl file for the datapath used in the ACU example.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(parent-cell oacudp)
(parameters N NDIRN1 HALFLT IMR R RM1 IMRM1 IMGEDG
  (MODULE_TYPE "DATAPATH"))
(layout-generator Flint a)
(structure-processor dpp)
;It seems that N is a keyword in DPC
(subcells  (bufferbig     BIN1 ((N N)))
           (buffersmall   BIN2 ((N N)))
           (reg2port      (R10 R11 R12 R13 R14
 R15 R20 R21 R22 latch) ((N N)))
 (regconstant   C10 ((N N) (constant NDIRN1)))
 (regconstant   C11 ((N N) (constant HALFLT)))
 (regconstant   C12 ((N N) (constant IMR)))
 (regconstant   C13 ((N N) (constant R)))
           (acumux        MUX1 ((N N)))
           (acumux        MUX2 ((N N)))
           (up1           SHA  ((N N)))
           (up2           SHB  ((N N)))
           (up2           SHC  ((N N)))
           (up4           SHD  ((N N)))
 ;(xor1         XOR1 ((N N)))
 ;(xor1         XOR2 ((N N)))
```

```
 ;(adder          AS   ((N N)))
          (addersub       AS   ((N N)))
 (decoder        DEC1 ((N N) (constant RM1)))
 (decoder        DEC2 ((N N) (constant IMRM1)))
 (decoder        DEC3 ((N N) (constant IMGEDG))))
;DATA NETS
;register files
(net r1  ((BIN1 OUT) (R10 IN) (R11 IN) (R12 IN) (R13 IN)
(R14 IN) (R15 IN)))
(net r1out ((R10 OUT) (R11 OUT) (R12 OUT) (R13 OUT) (R14 OUT)
  (R15 OUT) (C10 OUT) (C11 OUT) (C12 OUT) (C13 OUT)
  (MUX1 IN)))
(net r2 ((BIN2 OUT) (R20 IN) (R21 IN) (R22 IN)))
(net r2out ((R20 OUT) (R21 OUT) (R22 OUT) (MUX2 IN)))
(net bus2a ((MUX2 OUT) (SHA IN)))
(net sh12 ((SHA OUT) (SHB IN)))
(net sh23 ((SHB OUT) (SHC IN)))
(net sh34 ((SHC OUT) (SHD IN)))
;(net bus2b ((SHD OUT) (XOR2 IN)))
(net bus2b ((SHD OUT) (AS IN1)))
;(net bus2c ((MUX1 OUT) (XOR1 IN)))
(net bus2c ((MUX1 OUT) (AS IN2)))
;(net add1 ((XOR1 OUT) (AS IN1)))
;(net add2 ((XOR2 OUT) (AS IN2)))
;Net connection to the external bus
(net toadrreg ((latch IN) (AS OUT) (BIN1 IN) (BIN2 IN) (DEC1 IN)
(DEC2 IN) (DEC3 IN)))
(net toout ((latch OUT) (parent ADRBUS)))

;CONTROL NETS
;register load signals
(net CNTLr1 ((parent A1LOAD0) (R10 LOAD)))
(net CNTLr2 ((parent A1LOAD0INV) (R10 LOADINV)))
(net CNTLr3 ((parent A1LOAD1) (R11 LOAD)))
(net CNTLr4 ((parent A1LOAD1INV) (R11 LOADINV)))
(net CNTLr5 ((parent A1LOAD2) (R12 LOAD)))
(net CNTLr6 ((parent A1LOAD2INV) (R12 LOADINV)))
(net CNTLr7 ((parent A1LOAD3) (R13 LOAD)))
(net CNTLr8 ((parent A1LOAD3INV) (R13 LOADINV)))
(net CNTLr9 ((parent A1LOAD4) (R14 LOAD)))
(net CNTLr10 ((parent A1LOAD4INV) (R14 LOADINV)))
(net CNTLr11 ((parent A1LOAD5) (R15 LOAD)))
(net CNTLr12 ((parent A1LOAD5INV) (R15 LOADINV)))
(net CNTLr19 ((parent A2LOAD0) (R20 LOAD)))
```

```
(net CNTLr20 ((parent A2LOADOINV) (R20 LOADINV)))
(net CNTLr21 ((parent A2LOAD1) (R21 LOAD)))
(net CNTLr22 ((parent A2LOAD1INV) (R21 LOADINV)))
(net CNTLr23 ((parent A2LOAD2) (R22 LOAD)))
(net CNTLr24 ((parent A2LOAD2INV) (R22 LOADINV)))
;register oe signals
(net CNTLo1 ((parent A1OEN0) (R10 OEN)))
(net CNTLo2 ((parent A1OEN0INV) (R10 OENINV)))
(net CNTLo3 ((parent A1OEN1) (R11 OEN)))
(net CNTLo4 ((parent A1OEN1INV) (R11 OENINV)))
(net CNTLo5 ((parent A1OEN2) (R12 OEN)))
(net CNTLo6 ((parent A1OEN2INV) (R12 OENINV)))
(net CNTLo7 ((parent A1OEN3) (R13 OEN)))
(net CNTLo8 ((parent A1OEN3INV) (R13 OENINV)))
(net CNTLo9 ((parent A1OEN4) (R14 OEN)))
(net CNTLo10 ((parent A1OEN4INV) (R14 OENINV)))
(net CNTLo11 ((parent A1OEN5) (R15 OEN)))
(net CNTLo12 ((parent A1OEN5INV) (R15 OENINV)))
(net CNTLo13 ((parent A1OEN6) (C10 OEN)))
(net CNTLo14 ((parent A1OEN6INV) (C10 OENINV)))
(net CNTLo15 ((parent A1OEN7) (C11 OEN)))
(net CNTLo16 ((parent A1OEN7INV) (C11 OENINV)))
(net CNTLo17 ((parent A1OEN8) (C12 OEN)))
(net CNTLo18 ((parent A1OEN8INV) (C12 OENINV)))
(net CNTLo19 ((parent A1OEN8) (C13 OEN)))
(net CNTLo20 ((parent A1OEN8INV) (C13 OENINV)))
(net CNTLo21 ((parent A2OEN0) (R20 OEN)))
(net CNTLo22 ((parent A2OEN0INV) (R20 OENINV)))
(net CNTLo23 ((parent A2OEN1) (R21 OEN)))
(net CNTLo24 ((parent A2OEN1INV) (R21 OENINV)))
(net CNTLo25 ((parent A2OEN2) (R22 OEN)))
(net CNTLo26 ((parent A2OEN2INV) (R22 OENINV)))
;mux nets (2 muxes)
(net CNTLm1 ((parent AMUX1SEL) (MUX1 CNTL1)))
(net CNTLm2 ((parent AMUX1SELINV) (MUX1 CNTL2)))
(net CNTLm3 ((parent AMUX1OTHER) (MUX1 CNTL3)))
(net CNTLm4 ((parent AMUX1LSB) (MUX1 CNTL4)))
(net CNTLm5 ((parent AMUX2SEL) (MUX2 CNTL1)))
(net CNTLm6 ((parent AMUX2SELINV) (MUX2 CNTL2)))
(net CNTLm7 ((parent AMUX2OTHER) (MUX2 CNTL3)))
(net CNTLm8 ((parent AMUX2LSB) (MUX2 CNTL4)))
;shifter controls
(net CNTLs1 ((parent ASHA) (SHA SHIFT1)))
(net CNTLs2 ((parent ASHB) (SHB SHIFT2)))
```

```
(net CNTLs3 ((parent ASHC) (SHC SHIFT2)))
(net CNTLs4 ((parent ASHD) (SHD SHIFT4)))
(net CNTLs5 ((parent ASHINVA) (SHA SHIFT1BAR)))
(net CNTLs6 ((parent ASHINVB) (SHB SHIFT2BAR)))
(net CNTLs7 ((parent ASHINVC) (SHC SHIFT2BAR)))
(net CNTLs8 ((parent ASHINVD) (SHD SHIFT4BAR)))
;xor controls
;(net CNTLx1 ((parent ASSUB1) (XOR1 CNTL1)))
;(net CNTLx2 ((parent ASSUB2) (XOR2 CNTL1)))
(net CNTLx2 ((parent ASSUB2) (AS CNT)))
;adder controls
(net CNTLa1 ((parent AASCIN) (AS CIN)))
(net CNTLa2 ((parent AASCININV) (AS CININV)))

(terminal ADRBUS (TERMTYPE DATA_SIGNAL)(DIRECTION OUTPUT))

(net Vdd ((BIN1 Vdd)(BIN2 Vdd)(R10 Vdd)(R11 Vdd)(R12 Vdd)(R13 Vdd)
  (R14 Vdd)(R15 Vdd)(C10 Vdd)(C11 Vdd)(C12 Vdd)(C13 Vdd)(MUX1 Vdd)
  (MUX2 Vdd)(SHA Vdd)(SHB Vdd)(SHC Vdd)(SHD Vdd)(AS Vdd)(DEC1 Vdd)
  (DEC2 Vdd)(DEC3 Vdd)(parent Vdd)))
(net GND ((BIN1 GND)(BIN2 GND)(R10 GND)(R11 GND)(R12 GND)(R13 GND)
  (R14 GND)(R15 GND)(C10 GND)(C11 GND)(C12 GND)(C13 GND)(MUX1 GND)
  (MUX2 GND)(SHA GND)(SHB GND)(SHC GND)(SHD GND)(AS GND)(DEC1 GND)
  (DEC2 GND)(DEC3 GND)(parent GND)))

(end-sdl)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Sdl file for the cell data used in the system.sdl file.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(parent-cell data)
(parameters N (MODULE_TYPE "DATAPATH"))
(layout-generator Flint a)
(structure-processor dpp)
(subcells (reg2port imreg ((N N))))

(net out ((parent OUT) (imreg OUT)))
(net load ((parent ILOAD) (imreg LOAD)))
(net loadi ((parent ILOADI) (imreg LOADINV)))
(net oen ((parent IOEN) (imreg OEN)))
(net oeni ((parent IOENI) (imreg OENINV)))

(net Vdd ((parent Vdd)(imreg Vdd)))
(net GND ((parent GND)(imreg GND)))
```

```
(terminal OUT (TERMTYPE DATA_SIGNAL)(DIRECTION OUTPUT))

(end-sdl)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Although not part of the ASIC, a model was needed for a RAM.  This
;;; sdl file describes a generic RAM cell.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(parent-cell genericmem)
(parameters (MODULE_TYPE "MEM"))
(subcells
(memory_load LOAD)
(memory_out  OEN))
; nets for the parent cell.
(net in1 ((parent IN) (LOAD IN)))
(terminal READ)
(terminal WRITE)
(terminal OUT (DIRECTION OUTPUT))
(terminal IN (DIRECTION INPUT))
(end-sdl)
```

B.3 Behavior View BDL files used by the Assembler.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;  bdl description for acumux.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(behavior-cell acumux)
(terminal CNTL1)
(terminal CNTL2)
(terminal CNTL3)
(terminal CNTL4)
(terminal IN (DIRECTION INPUT))
(terminal OUT (DIRECTION OUTPUT))
;symbolic truth table
(symbolic_tt
(case (output (VALUE "IN"))
(condition (CNTL1 0)(CNTL2 1)(CNTL3 0)(CNTL4 0))
)
(case  (output (VALUE "1"))
(condition (CNTL1 1)(CNTL2 0)(CNTL3 0)(CNTL4 1))
)
(case  (output (VALUE "0"))
(condition (CNTL1 1)(CNTL2 0)(CNTL3 0)(CNTL4 0))
```

```
)
(case  (output (VALUE "-1"))
(condition (CNTL1 1)(CNTL2 0)(CNTL3 1)(CNTL4 1))
)
(case  (output (VALUE "nop"))
(condition (CNTL1 1)(CNTL2 0)(CNTL3 0)(CNTL4 0))
)
)
(end-bdl)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bdl description for addersub
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(behavior-cell addersub)
(primitives
(addition ADD)
(subtraction SUB)
)
; nets for the parent cell.
(net in1a ((parent IN1) (ADD IN1) (SUB IN1)))
(net in2a ((parent IN2) (ADD IN2) (SUB IN2)))
; Control and outputs
(terminal CIN)
(terminal CININV)
(terminal COUT)
(terminal COUTINV)
(terminal COUTN_1INV)
(terminal OUT)
(termianl OUTINV)
(terminal CNT)
; Symbolic truth table
(symbolic_tt
(case  (output (TERM (ADD OUT) (parent OUT)))
  (condition (CIN 0) (CININV 1) (CNT 0))
)
(case (output (VALUE "nop"))
(condition (CIN 0) (CININV 1) (CNT 0))
)
(case (output (TERM (SUB OUT) (parent OUT)))
(condition (CIN 1) (CININV 0) (CNT 1))
)
(end-bdl)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; bdl description for genericmem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(behavior-cell genericmem)
(primitives
(memory_load LOAD)
(memory_out  OEN))
; nets for the parent cell.
(net in1 ((parent IN) (LOAD IN)))
(terminal READ)
(terminal WRITE)
(terminal OUT)
; Symbolic truth table
(symbolic_tt
(case  (output (VALUE "nop"))
  (condition (WRITE "0") (READ "0"))
)
(case (output (TERM (parent OUT) (OEN OUT)))
(condition (READ "1") (WRITE "0"))
)
(case  (output (OP "LOAD"))
(condition (READ "0") (WRITE "1"))
)
)
(end-bdl)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bdl description for reg2port
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(behavior-cell reg2port)
(primitives
(memory_load LOAD)
(memory_out  OEN))
; nets for the parent cell.
(net in1 ((parent IN) (LOAD IN)))
; Control
(terminal LOAD)
(terminal LOADINV)
(termianl OEN)
(terminal OENINV)
(terminal OUT)
; Symbolic truth table
(symbolic_tt
(case  (output (VALUE "nop"))
  (condition (LOAD 0)(LOADINV 1)(OEN 0)(OENINV 1))
```

```
)
(case (output (TERM (OEN OUT) (parent OUT)))
(condition (LOAD 0)(LOADINV 1)(OEN 1)(OENINV 0))
)
(case  (output (OP "LOAD"))
(condition (LOAD 1)(LOADINV 0)(OEN 0)(OENINV 1))
)
)
(end-bdl)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bdl descriptions for shifters up1, up2, and up4.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(behavior-cell up1)
(primitives
upshift1 D1)
(net in1 ((parent IN) (D1 IN)))
(terminal OUT)
(terminal SHIFT1)
(terminal SHIFT1BAR)
(symbolic_tt
(case (output (VALUE "nop"))
(condition (SHIFT1 0)(SHIFT1BAR 1))
)
(case  (output (VALUE "IN"))
(condition (SHIFT1 0)(SHIFT1BAR 1))
)
(case (output (TERM (parent OUT) (D1 OUT)))
(condition (SHIFT1 1)(SHIFT1BAR 0))
)
)
(end-bdl)

(behavior-cell up2)
(primitives
upshift2 D2)
(net in1 ((parent IN) (D2 IN)))
(terminal OUT)
(terminal SHIFT2)
(terminal SHIFT2BAR)
(symbolic_tt
(case (output (VALUE "nop"))
(condition (SHIFT2 0)(SHIFT2BAR 1))
)
```

```
(case  (output (VALUE "IN1"))
(condition (SHIFT2 0)(SHIFT2BAR 1))
)
(case (output (TERM (parent OUT) (D2 OUT)))
(condition (SHIFT2 1)(SHIFT2BAR 0))
)
)
(end-bdl)

(behavior-cell up4)
(primitives
upshift4 D4)
(net in1 ((parent IN) (D4 IN)))
(terminal OUT)
(terminal SHIFT4)
(terminal SHIFT4BAR)
(symbolic_tt
(case (output (VALUE "nop"))
(condition (SHIFT4 0)(SHIFT4BAR 1))
)
(case  (output (VALUE "IN1"))
(condition (SHIFT4 0)(SHIFT4BAR 1))
)
(case (output (TERM (parent OUT) (D2 OUT)))
(condition (SHIFT4 1)(SHIFT4BAR 0))
)
)
(end-bdl)
```

B.4 Output Files from the Assembler.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is the "asm.tables" file created by assembler from the orient.c
;;; program and the structure master view of the sdl file system.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
********Symbol Table********
NAME= dirnImg          TYPE= Variable   MEM= dirnImg
NAME= xImg             TYPE= Variable   MEM= R10
NAME= yImg             TYPE= Variable   MEM= R20
NAME= zero             TYPE= Variable   MEM= imreg
NAME= IMR              TYPE= Variable   MEM= C12
NAME= IMR2             TYPE= Constant   VALUE= 508
NAME= IMGSIZ           TYPE= Constant   VALUE= 512
NAME= RIDGESIZ         TYPE= Variable   MEM= C13
```

```
NAME= RIDGE           TYPE= Constant   VALUE= 4
NAME= SIZE            TYPE= Constant   VALUE= 262144
NAME= address         TYPE= Variable   MEM= latch


********StateTable********
Block=  1 NextBlock=  2  Cond: EOB
Block=  2 NextBlock=  3  Cond: EOB
Block=  3 NextBlock=  3  Cond: (not (IMGSIZ))
Block=  3 NextBlock=  4  Cond: (IMGSIZ)
Block=  4 NextBlock=  2  Cond: (not (RIDGE))
Block=  4 NextBlock=  5  Cond: (RIDGE)
Block=  5 NextBlock=  6  Cond: EOB
Block=  6 NextBlock=  7  Cond: EOB
Block=  7 NextBlock=  7  Cond: (not (IMGSIZ))
Block=  7 NextBlock=  8  Cond: (IMGSIZ)
Block=  8 NextBlock=  6  Cond: (not (IMGSIZ))
Block=  8 NextBlock=  9  Cond: (IMGSIZ)
Block=  9 NextBlock= 10  Cond: EOB
Block= 10 NextBlock= 11  Cond: EOB
Block= 11 NextBlock= 11  Cond: (not (RIDGE))
Block= 11 NextBlock= 12  Cond: (RIDGE)
Block= 12 NextBlock= 10  Cond: (not (IMR2))
Block= 12 NextBlock= 13  Cond: (IMR2)
Block= 13 NextBlock= 14  Cond: EOB
Block= 14 NextBlock= 15  Cond: EOB
Block= 15 NextBlock= 15  Cond: (not (RIDGE))
Block= 15 NextBlock= 16  Cond: (RIDGE)
Block= 16 NextBlock= 14  Cond: (not (IMR2))
Block= 16 NextBlock= 17  Cond: (IMR2)
Block= 17 NextBlock= 18  Cond: EOB


*************Symbols and their notation**************
Name: IMR2              Type: Decoder     Decode value: 508
Name: RIDGE             Type: Decoder     Decode value: 4
Name: IMGSIZ            Type: Decoder     Decode value: 512


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is the file "cfsm.esp" created by the assembler from the program
;;; "orient.c".
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.i 9
```

```
.o 10
.type f
00000---- 1000000001
10000---0 1000000001
01000---0 0100000010
11000---0 1100000011
00100---0 0010000100
10100---0 1010000101
01100---0 0110000110
11100---0 1110000111
00010---0 0001001000
10010---0 1001001001
01010---0 0101001010
11010---0 1101001011
00110---0 0011001100
10110---0 1011001101
01110---0 0111001110
11110---0 1111001111
00001---0 0000110000
10001---0 1000110001
01001---1 0100110010
10000---1 0100000010
01000---1 1100000011
11000--01 1100000011
11000--11 0010000100
00100-0-1 0100000010
00100-1-1 1010000101
10100---1 0110000110
01100---1 1110000111
11100--01 1110000111
11100--11 0001001000
00010--01 0110000110
00010--11 1001001001
10010---1 0101001010
01010---1 1101001011
11010-0-1 1101001011
11010-1-1 0011001100
001100--1 0101001010
001101--1 1011001101
10110---1 0111001110
01110---1 1111001111
11110-0-1 1111001111
11110-1-1 0000110000
000010--1 0111001110
```

```
000011--1 1000110001
10001---1 0100110010
.e


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is the file "cstore.esp".  It was created by editing the file
;;; "bit.info" which gives information on how the bits will be stored in
;;; the finite state machine.  The program "post" was used to generate
;;; this file.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.i 7
.o 41
.type f
1000001 10000000000010000000000010010000000000001
0100001 01000000000001000010000000000011110000001
1100001 11000000000000000000000000010100000001010
1100010 01000000000001000010000000000011110000001
0010001 00000000000011000000000010100000000000000
0010010 10000000000000000000000010010000000000001
1010001 00000000000010000000001000010000000000001
0110001 01000000000001000010000000000011110000001
1110001 11000000000000000000000000010100000001010
1110010 01000000000001000010000000000011110000001
0001001 00000000000011000000000010100000000000000
0001010 10000000000000000000000010010000000000001
1001001 00000000000010000000000100010100000000001
0101001 01000000000001000010000000000011110000001
1101001 11000000000000000000000000010100000001010
1101010 01000000000001000010000000000011110000001
0011001 00000000000011000000000010100000000000000
0011010 10000000000000000000000010010000000000001
1011001 00000000000010000000000100010100000000000
1011010 10000000000000000000001000010000000000001
0111001 01000000000001000010000000000011110000001
1111001 11000000000000000000000000010100000001010
1111010 01000000000001000010000000000011110000001
0000101 00000000000011000000000010100000000000000
0000110 11000000000000000000000000010100000000001
1000101 00000000000000000000000010010000000000000
1000110 10000000000001000000000010100000011000001
0100101 00000000000000000000000010010000000000001
.e
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is the file "pcu.parval".  It was generated by the assembler.
;;; It provides the parameters needed by the controller in order for the
;;; Design Manager to generate the final layout.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(num_of_blks 18)
(num_of_states 19)
(stack_depth 0)
(num_of_cond 3)
(num_of_loopslice 0)
(cfsmtab "cfsm.esp")
(max_blk_size 2)
(cstoretab "cstore.esp")
```