1990

A prototype control and data path synthesis system

William R. Migatz
Lehigh University

A PROTOTYPE CONTROL

AND

DATA PATH SYNTHESIS SYSTEM

by

William R. Migatz

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

1990

This thesis is accepted in partial fulfillment of the requirements for the degree of Master of Science.

December 17, 1990

(date)

Frank W. Wiebelhaus

Professor in Charge

Lawrence J. Vanerin

Chairman of Department

ACKNOWLEDGEMENTS

The author would like to express his appreciation to the following people for their contributions:

Dr. Frank Hielscher, for his guidance and suggestions.

Dr. J. Bhasker, for his guidance and insight that was crucial during the early stages of the system's development.

My parents, Melvin and Jean Migatz, for their support and encouragement without which I could not have completed this thesis.

Table of Contents

List of Figures

ABSTRACT

This thesis describes a prototype high level synthesis system that attempts to take the delay and the area of both the data path and the controller into account. The system takes as its input an intermediate form (closely related to a data flow graph) describing an algorithm and technology files along with both area and time constraints. The output generated by the system is a description of both the data path and the controller along with estimates of the design's delay and area. Furthermore, this system rejects the idea that the delay of a design can be described by its clock cycle length and defines the delay as the total time of execution of the algorithm given certain statistical information regarding loops and conditional statements.

The system is iterative in nature, making incremental changes to the design on each pass. The design process itself starts with operation scheduling using a modified *as soon as possible* scheduling algorithm. This generates the schedule of operations. At this point, information used to determine variable lifetimes (the control steps during which variables must be stored in registers) is generated and information regarding the next state sequencing is generated. Next, variable lifetimes are determined and register allocation is performed. Once register allocation is performed, data path allocation and module binding are completed. At that point all of the information needed to generate a controller is available. The controller is generated and area and speed estimates are calculated. The algorithm iterates by adding functional units and, once again, generating a new design to determine the effect on the design.

Chapter 1
INTRODUCTION

Since the advent of the integrated circuit both the transistor density and the size of integrated circuits have been steadily increasing. At the present time chips with over a million transistors already exist. It is predicted that by the year 2000, logic devices with up to 50 million transistors [10] on a single die will be manufactured! Obviously, the increasing transistor count allows increasingly more complex designs to be placed on a single chip. At first glance it would be assumed that a direct result of this would be increasingly long design times for those chips. This, however, is not the case. If the design methodology remains at its current state of the art, within a few years it will be impossible for a logic device to be designed that takes full advantage of the maximum attainable transistor density and die size. The designs will have become so large that there will be no way for them to be verified by the existing methodologies. Thus, new design methodologies must and are being developed.

Before exploring these new design methodologies (which are being developed primarily for digital integrated circuits), it is necessary to understand what methods have been used in the past and what methods are being used today. When integrated circuits first appeared there were only a handful of transistors on each chip. As a result, it was possible to do the design, verify its functionality, and even lay out the masks completely by hand. Later, as the number of transistors increased, this became increasingly difficult. In the 1970's circuit analysis software such as SPICE [17] began to be used to verify the design. Also, software was developed that allowed the designer to layout the chips' masks on a computer. This type of "hand" layout has become known as full custom design and for some specialized applications is still used today.

Soon, however, these tools became inadequate for most designs. Logic analysis tools were developed to check the functionality and timing of digital circuits. Still, the design task itself was left completely up to the design team. Eventually what can be categorized as low level synthesis tools were developed. They allowed a designer to give a transistor description of certain circuits and the synthesis tools would use this design to produce the masks. What followed are the tools that are the main design tools used today. They are mostly logic circuit minimization software, silicon compilers, and logic synthesis software. Logic circuit minimization software usually tries to minimize the size of logic equations based on some cost function such as the number of literals or the number of product terms. An example of this would be ESPRESSO [6], a program developed at the University of California at Berkeley. A silicon compiler generates a mask for a regular logic structure such as a PLA or a ROM. This is a relatively simple task in that the format of a ROM or a PLA remains roughly the same from one ROM or PLA to the next and in that the structure is quite regular (many similar substructures). Several silicon compilers are commercially available today. Finally, logic synthesis takes a set of logic equations, minimizes them based on one or more cost criteria, and maps them into a particular technology. The cost criteria are usually area and delay. Two logic synthesis tools are MIS [14] and SOCRATES [16].

All of these design tools that are being used today are helpful. However, even some of today's largest designs have pushed these design tools past their useful limits. These tools still force the designer to come up with the architecture and a register transfer level (RTL) design. This in itself is no small task. Any given algorithm can be implemented in a large number of different RTL designs. Each one of these designs has its own unique characteristics in that each design requires a different amount of chip area and has a different

delay. Currently it is left up to the designer to come up with a design that meets the area and delay constraints. This design space, however, is too large for a computer to search through every possible design, let alone a human designer. An expert designer must use his experience to determine what type of design has a good chance of meeting the design specifications. The problem is further complicated by the fact that design decisions at high levels affect the lower levels and vice versa [1] [11]. For instance, what may appear to be a good RTL design may use a vast amount of area because of low level routing problems.

An entirely new type of design methodology is currently being developed by both universities and industry. The design methodology is called high level synthesis. The object of such a system is to take as its input an algorithmic description of the functions a chip is to perform (this is written is some high level language such as C or Pascal [8] or in a hardware description language such as ISPS or VHDL), along with area and timing constraints, and to produce an RTL description of a logic circuit that implements the algorithm's function and also meets the area and time constraints. From there, tools such as the ones being used today should be capable of generating the masks that implement the RTL design. This sounds like the impossible dream. However, enough research has already been done and enough experimental systems have been developed to prove that this is not only possible but practical. In fact, for the first time, corporations are seriously looking at these systems and in some cases are starting to very cautiously use them.

Why such a dramatic approach to the design problem? Well it has already been mentioned how, without such a new method, designs that take full advantage of the ever increasing transistor density and die size will not be able to be designed. There are also many other reasons. The world being what it is,

the most important of these reasons is financial. As the complexity of chips grows so does the design time. With design time now extending from months to years, it is becoming more and more difficult for a company to hit the market window for a design [18]. It is hoped that such systems will cut design time down to weeks or months, thus, making it easier to hit the market window. Also, along with the reduction in design time will come a reduction in the manpower [18] necessary to produce a design. The result of both of these factors will be a less costly product. A high level synthesis system that uses good heuristics will be able to search through a design space more efficiently than a human designer. Since a synthesis system might be able to produce a design in a number of hours it would be possible for a designer to chose from a number of machine generated designs from different areas of the design space [18]. Also, if a high level synthesis system can be shown to always produce correct designs, the likelyhood of an error being made during the design process dramatically decreases. Furthermore, these systems can be designed to keep track of and document the design through its various stages. Finally, such systems would ultimately allow people who are not experts in chip design to do just that - design a chip. More people will be able to take advantage of the technology.

Today, a large number of experimental high level synthesis systems exist. Most of these systems concentrate on the generation of the datapath. The consideration of the controller for the datapath (control is rarely incorporated into the data path but, rather, kept as a separate unit) is treated as an afterthought that is a consequence of the datapath. In this thesis a prototype synthesis system is presented that attempts to take into account the effect of the controller on the chip's overall area and speed.

In chapter 2 of this thesis a description of the overall approach to high level synthesis is given. Such questions as: What is high level synthesis? and

How does it work? will be answered in detail. Chapter 3 will explain what is different about the Control and Data path Synthesis System, and how the system works. Chapter 4 will present some of the results obtained from the synthesis system and will draw some conclusions from these results as well as from several problems encountered in its development. The advantages as well as the shortcomings of the system will be discussed.

Chapter 2
THE GENERAL APPROACH TO HIGH LEVEL SYNTHESIS

The start and end points of the digital integrated circuit design process have not been changed by high level synthesis. The starting point always has been some behavioral description of the functions that the chip is to perform along with area and timing constraints. The end point is the masks used to manufacture the chips. High level synthesis is generally concerned with going from the behavioral description to the register transfer level description. This is usually done in two major steps. The first step involves the translation of the behavioral description into a more useful form which is either a data flow graph (DFG) or a control and data flow graph (CDFG). The second step is to take this form and use it to generate a register transfer level description of a circuit that performs the specified functions.

A DFG is a representation of the operations that must be performed on the data. The DFG shows the ordering of the operations to be performed based on their data dependencies [18]. For example, given the two operations: C = A + B and E = C - D, it is clear that the subtraction operation must follow the addition operation because it relies on the result of the addition for one of its inputs. A CDFG also includes control constructs such as branch and join operations to represent the control flow of the behavioral description. The Design Automation Assistant (DAA) [3], [2] high level synthesis system, which was designed at AT&T Bell Laboratories, used a type of CDFG known as the "value trace". If a CDFG is not used, the control may be kept separate from the data flow in some other form. Figure 2-1 lists part of an algorithm used to compute the square root of X along with its DFG representation and a

```
Y = 0.22222 + 0.8889 * X;
I := 0;
DO UNTIL I > 3 LOOP
      Y := 0.5 * (Y + X/Y)
      I := I + 1;
ENDDO;
```

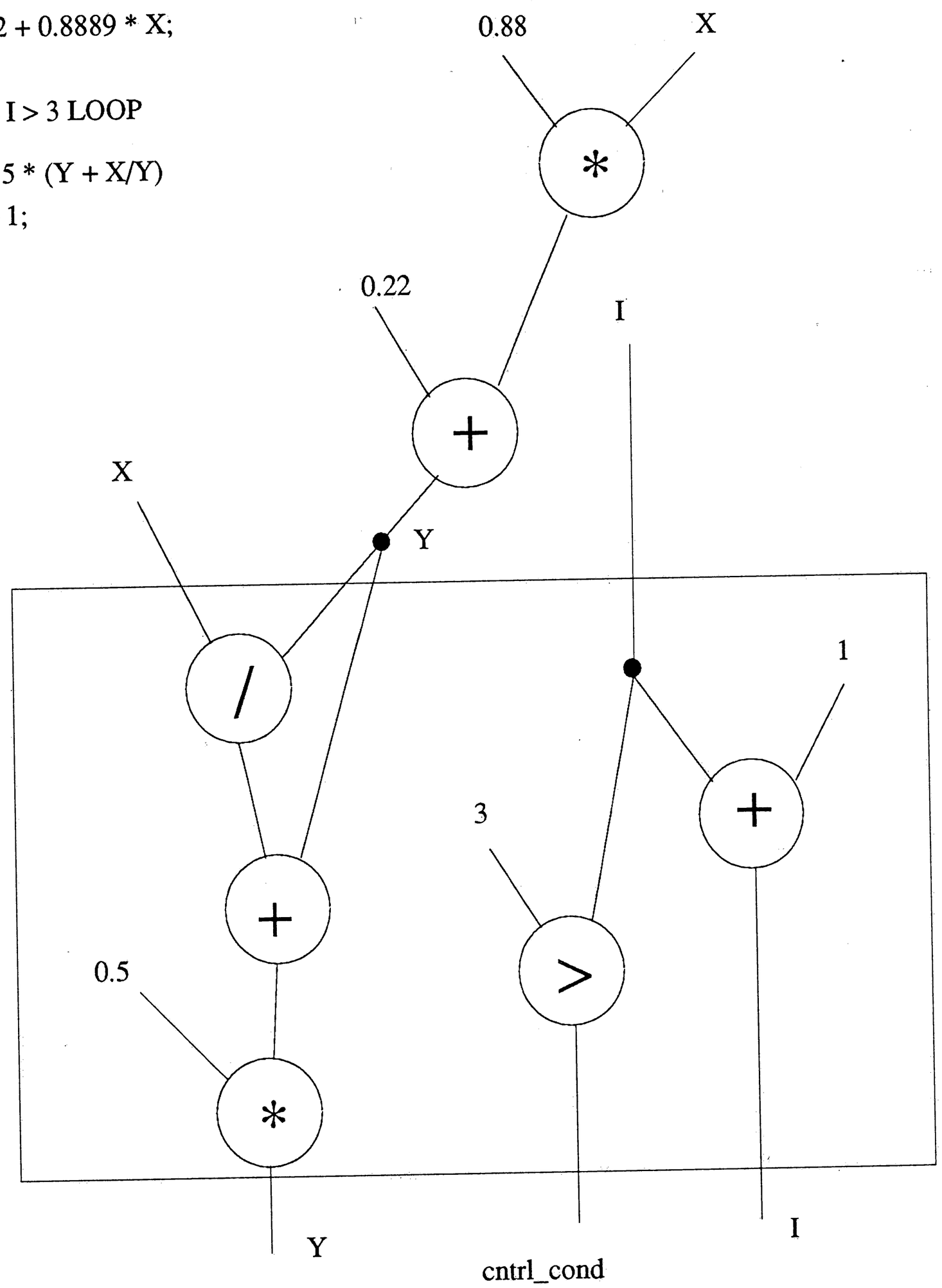


Figure 2-1: DFG for a Square Root Algorithm

representation of its control [18]. Once the behavioral description has been converted into a DFG or CDFG, numerous compiler-like optimizations can be performed. Some of the optimizations that can be performed are dead-code elimination, constant propagation, common subexpression elimination, in-line expansion of procedures, and loop unrolling. Because of its close relation to optimizing compilers much is known about this step.

The second major step is to go from the data flow graph to the register transfer level description. This is what is emphasized in this thesis. This step is further subdivided into two subtasks known as scheduling and allocation. Scheduling refers to the assignment of operations in the DFG to control steps. A control step is a period of time during which the data path is configured for a predetermined data flow. In simpler terms, a control step corresponds directly to one clock cycle. The goal here is to minimize the amount of time or the number of control steps needed in order to complete the execution of the program (program refers to the algorithm that the chip is to implement). Allocation involves the assignment of operations to functional units, the assignment of variables to registers, and the assignment of communications paths to buses and multiplexers. The goal here is to minimize the amount of hardware needed.

Most existing synthesis systems use scheduling as the first step. Here operations must be assigned to particular control steps based on certain constraints. One such constraint is the data dependencies. These dependencies are made explicit by the structure of the DFG. Other possible constraints (the existence of which depend upon the scheduling algorithm used) are the length of the clock cycle and the number of functional units available to perform each operation. Each operation must be implemented by a particular functional unit and each functional unit necessarily requires a finite amount of time to

complete its operation. The total amount of time needed to complete a sequence of scheduled operations cannot exceed the length of the clock cycle. Limits may be placed on the number of functional units to be used. For instance, if only three adders are to be used, then no more than three additions may be performed in any given control step. Figure 2-2 shows a DFG and lists the delays of the functional units that will be used to perform the various operations. The length of the clock cycle is also given. The table gives one possible schedule which meets the constraints. Time allows the fourth addition operation (operation 5) to be scheduled in the first clock cycle but the resource limits allow only three addition operations to be scheduled. In control step 2, only operations 5 and 6 can be scheduled due to time constraints and data dependencies. In control step 3, only addition operations 8 and 9 are scheduled despite the availability of three adders. This is because 3 additions in series plus the register delay would have a total delay of 16ns which is longer than the clock cycle length of 14ns. This is also why only one operation is scheduled in clock cycles 4 and 5. The dotted lines show the schedule listed in the table.

As can be seen from operations 4 and 7 in the example in figure 2-2 one operation may use the output of another operation as its input within the same control step. This is known as chaining. Chaining makes better use of the available time [4]. If one operation has a considerably longer delay than the other operations and chaining is not allowed, then much of the control step may be wasted executing only one operation when more of the shorter ones could be executed. Figure 2-3 demonstrates what is known as multicycling. That is the use of an operation that needs so much time to complete that it extends beyond the end of the clock cycle. In fact, it is possible for an operation to be scheduled that takes several clock cycles to complete. Multicycling is not always used. Its use depends upon the synthesis system and whether or not such a functional

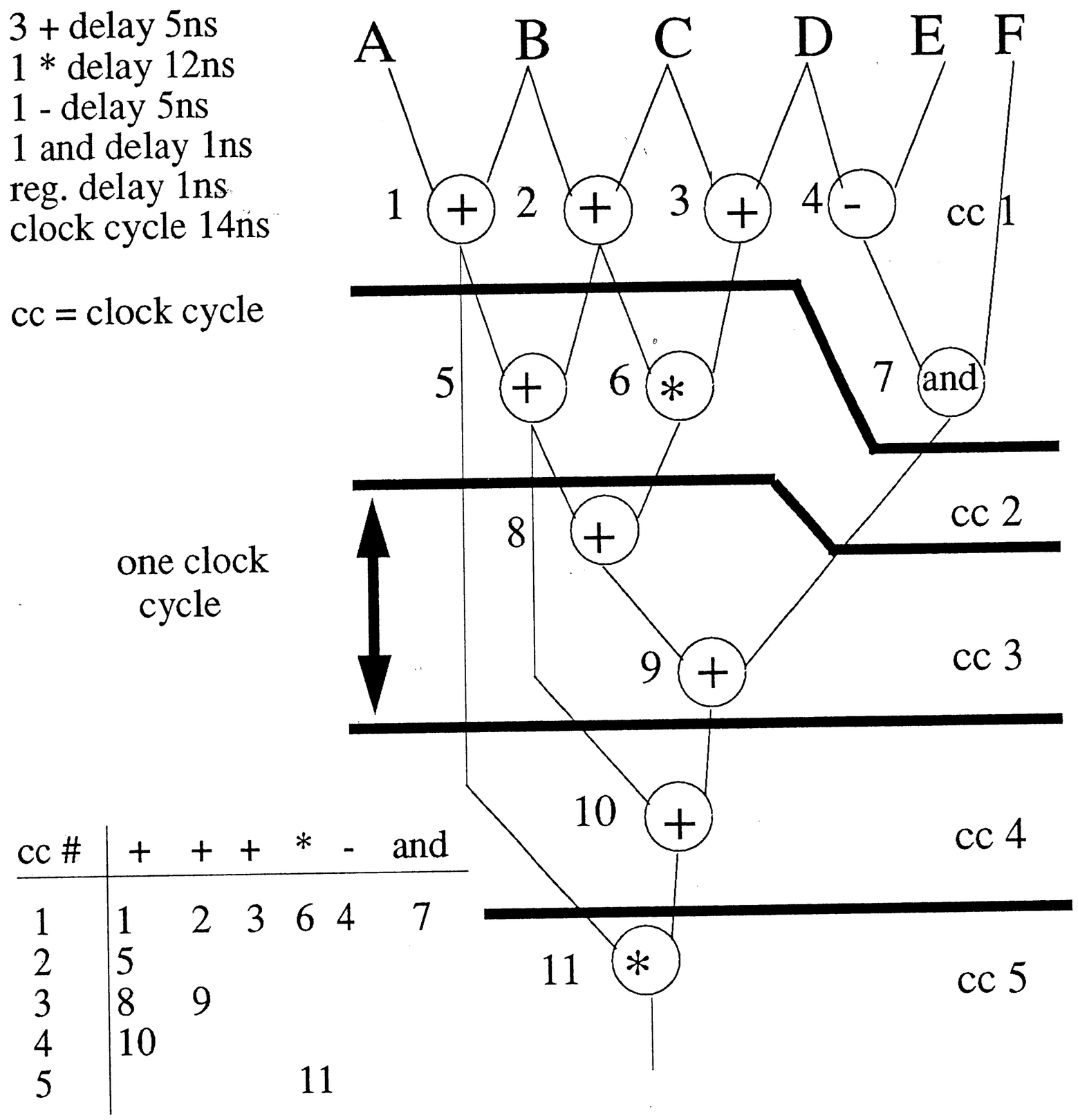


Figure 2-2: A Schedule For a DFG

unit is available. Multicycling allows the clock cycle length to be shorter than the longest functional unit delay. Also, if the multicycled unit allows new data to be input while it is still working on data obtained in the previous clock cycle, then that unit is pipelined [4].

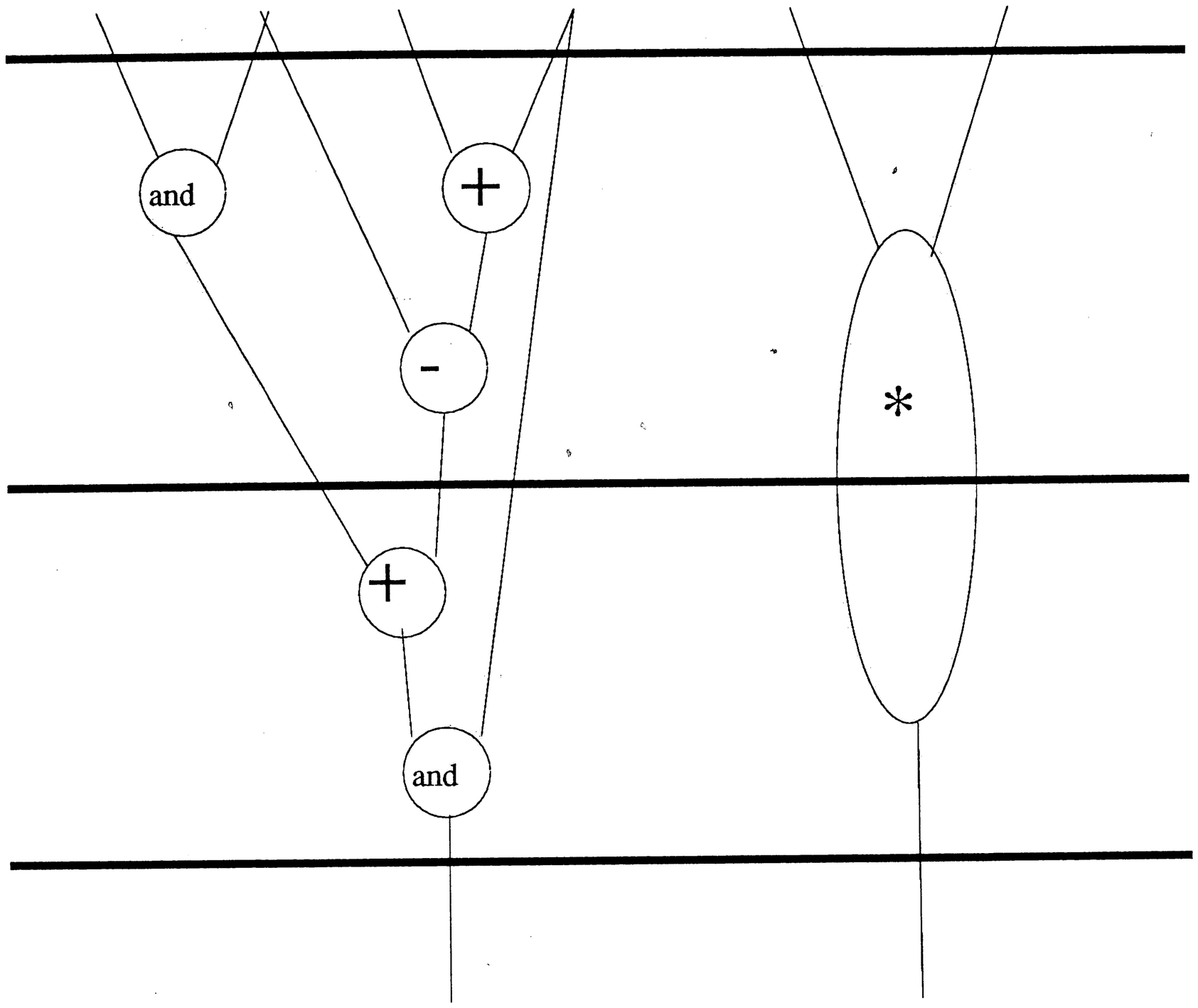


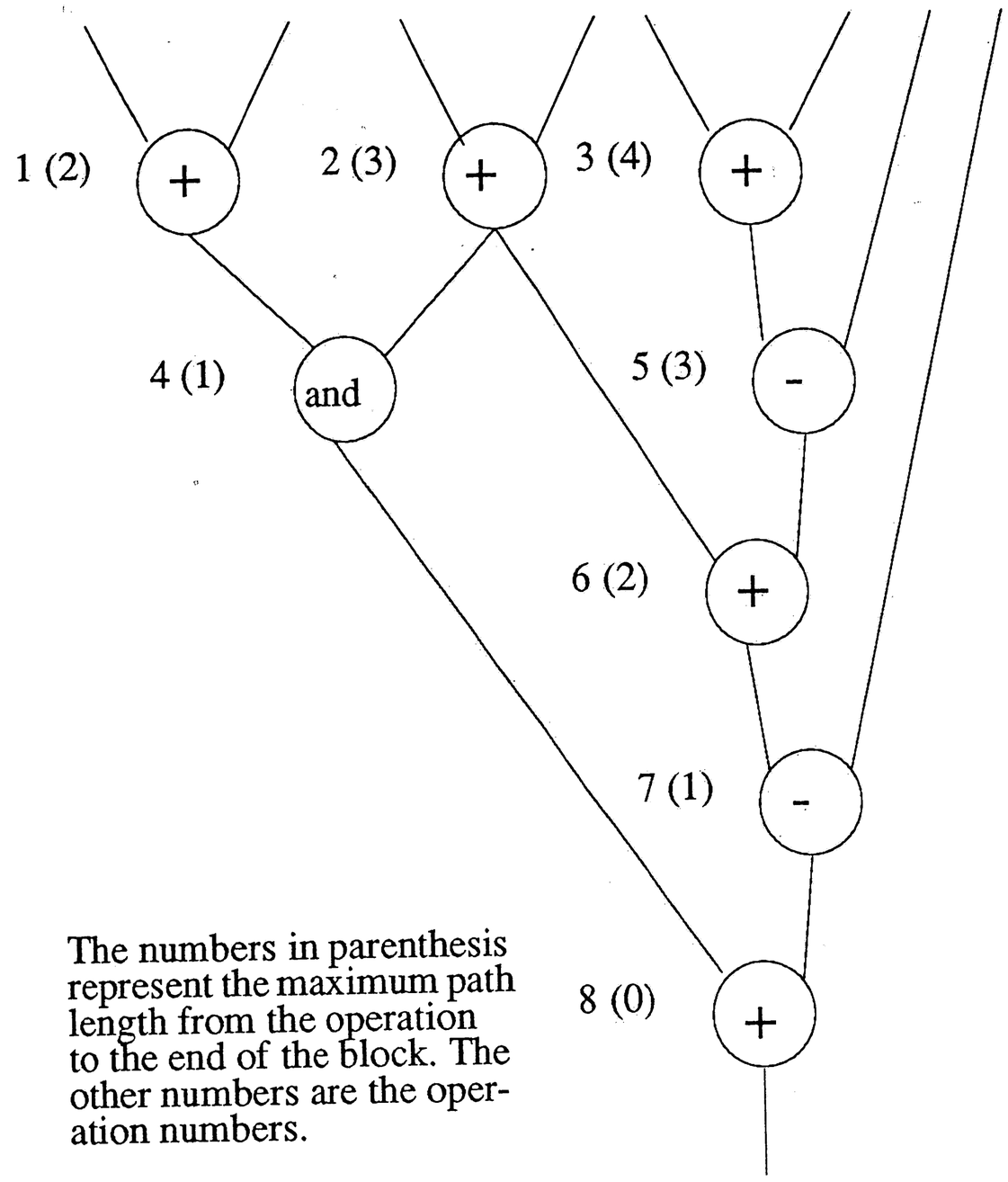
Figure 2-3: An Example of Multicycling

It would be nice to obtain the "optimal" schedule and use that. However, no such algorithm exists to obtain such a schedule and, for any practically sized problem, the time required to do an exhaustive search is unreasonably long. As a result, several algorithms have been developed to generate schedules. There are two classes of algorithms: transformational and iterative/constructive. Transformational algorithms start with a base schedule and perform various combinations of serial and parallel transformations that bring the schedule

closer to the imposed constraints. The Yorktown Silicon Compiler and the CAMAD design system both use such a technique. Iterative/constructive algorithms are somewhat more popular. They generate a schedule by examining each operation and deciding which control step it belongs to. There are several iterative/constructive scheduling algorithms including "as soon as possible", "as late as possible", list, and "force directed" scheduling [18].

The two simplest scheduling algorithms are "as soon as possible" scheduling (ASAP) and "as late as possible" scheduling (ALAP). In ASAP scheduling, the operations are sorted topologically. Operations that don't have any data dependencies are assigned to level 1. Operations that have data dependencies only on level 1 are assigned to level two. Operations that are dependent upon operations from level 2 are assigned to level 3. The operations are assigned to various levels in that manner until there are no more operations to be assigned to a level. For ASAP scheduling it is necessary to know how many of each type of functional unit is to be used. This determines what is known as the resource limits. Operations are selected from this topological list in order and assigned to the earliest control step that is allowed based upon the data dependencies, resource limits, and time constraints. Figure 2-4 shows a DFG and its ASAP schedule based on the given constraints. ASAP scheduling has been used in the CMUDA [2] (developed at Carnegie Mellon), MIMOLA [13] (developed at the University of Kiel in West Germany), and Flamel [8] (developed at Stanford University) high level synthesis systems. ALAP scheduling is the same as ASAP scheduling except that operations are scheduled in the last possible control step that they can be scheduled.

ASAP scheduling has one major problem. Operations on the critical path are not identified and thus all of one type of functional unit may be used up in one clock cycle by operations that are not on the critical path. The result of this



2 + delay 5ns
1 and delay 1ns
1 - delay 5ns
reg. delay 1ns
clock cycle 14ns

The numbers in parenthesis represent the maximum path length from the operation to the end of the block. The other numbers are the operation numbers.

ASAP

c.c. #	+	+	-	and
1	1	2		
2	3		5	4
3	6		7	
4	8			

List

c.c. #	+	+	-	and
1	3	2	5	
2	1	6	7	4
3	8			

Figure 2-4: Example ASAP and List Schedules of a DFG

is that the length of the schedule becomes longer than necessary. This can be

seen in figure 2-4 where operations 1 and 2 are scheduled first, forcing operation 3 to be postponed until the next control step. Since operation 3 is on the critical path, the schedule becomes one control step longer than necessary. List scheduling attempts to overcome this problem. In list scheduling, the operations that can be scheduled in each control step (this is determined by the data dependencies) are ordered by some priority function. The ones with the highest priority are scheduled first assuming all data dependencies, resource limits, and time constraints are met. When no more operations can be scheduled in the control step, the next control step is scheduled. The priority function can vary in this algorithm but is designed to locate operations on the critical path. The BUD [1] system used list scheduling. In figure 2-4 a list schedule for the given DFG is also given. The numbers in parenthesis on the DFG represent the maximum path length from the operation to the end of the block. This is the priority function used here. It can be seen that since operation 3 is at the top of the critical path, it has the highest priority and is thus scheduled first. The result is that fewer control steps are needed than in the ASAP schedule.

There is another scheduling algorithm known as force directed scheduling [5], [7]. Though this algorithm is not used in the synthesis system that is described in this thesis it is worth noting. The algorithm attempts to balance the concurrency of operations by placing similar operations in different control steps. This tends to reduce the number of functional units needed. It starts by determining the time frame into which each operation can be scheduled. This is done by determining both the ASAP and ALAP schedules. It also determines the probability of the operation being assigned to a control step. (If it can be assigned to 4 control steps then it has a probability of 0.25 that it will be assigned to any one of those steps.) Next a distribution graph is

generated. This graph uses the information about the time frames to add the probabilities of each type of operation for each control step. Finally operations are assigned to the control step that has the least force associated with it. The force of assigning an operation to a control step is the difference between the distribution value of that control step and the average of the distribution values for the control steps within the operation's time frame. This algorithm was first used in the HAL [9] system which was developed at Carleton University in Canada.

As previously mentioned, allocation involves the assignment of operations to specific functional units, the assignment of variables to registers, and the generation of communications paths (ie. the wiring of buses and multiplexers) [18]. The allocation of operations refers to the assignment of operations to functional units. This is straight forward if only one of each type of functional unit is used. However, if, for instance, two or more adders are used then a decision as to what addition operation should be bound to what adder must be made. The decision could be based on a number of criteria one of which may be which assignment would cause the least increase in wiring. Module binding is closely related to functional unit allocation. This involves selecting the proper functional unit to perform the operation. For instance, a library containing several adders each with different areas and delays might be used. Module binding would then involve selecting the one that will help most to meet the overall area and timing requirements. Sometimes special synthesis software can be used at this point to actually custom-generate a more ideal functional unit. This, however, requires considerably more time. Usually a number of multi-operation ALU's are used and the decision when to use an ALU and when to use a separate functional unit, as well as what functions should be combined into one ALU, go under the heading of module binding. Variables

must be allocated to registers. Each time the output of some operation extends beyond the control step in which it is first defined, that variable must be assigned to a register. Variables with disjoint lifetimes may be assigned to the same register. (The lifetime of a variable refers to the clock cycles during which it must be saved for use as the input to other operations.) For example, in figure 2-5 variables A, C, and D have disjoint lifetimes and may be assigned to the same register. One method of allocating variables to registers might be to allocate the variables in such a way that a minimum number of registers is used. Finally, the communications paths must be generated. This involves determining how the various functional units are wired to each other and to the registers. It also must be determined when to use buses and when to use multiplexers. Buses have the advantage of decreasing the amount of wiring necessary but they are often slower than multiplexers [18].

All of these allocation problems are interrelated. Solving them independently results in a solution that is less than optimal. For instance, if variables are assigned to registers in such a way that a minimum number of registers are used the savings in area might be offset by a corresponding increase in wiring cost because the resulting interconnections of the registers to the functional units may have caused an increase in the size or number of multiplexers or increased the length of the wires. Furthermore, an increase in the amount of wiring usually will result in a decrease in speed because of an increase in the capacitance associated with the wiring. Thus, ideally, all of these allocations problems must be solved together to get optimal results. This problem, however, becomes too complex for any reasonably large design and, as a result, the allocation problems are usually solved separately or possibly two problems might be solved in conjunction with each other and the rest solved separately.

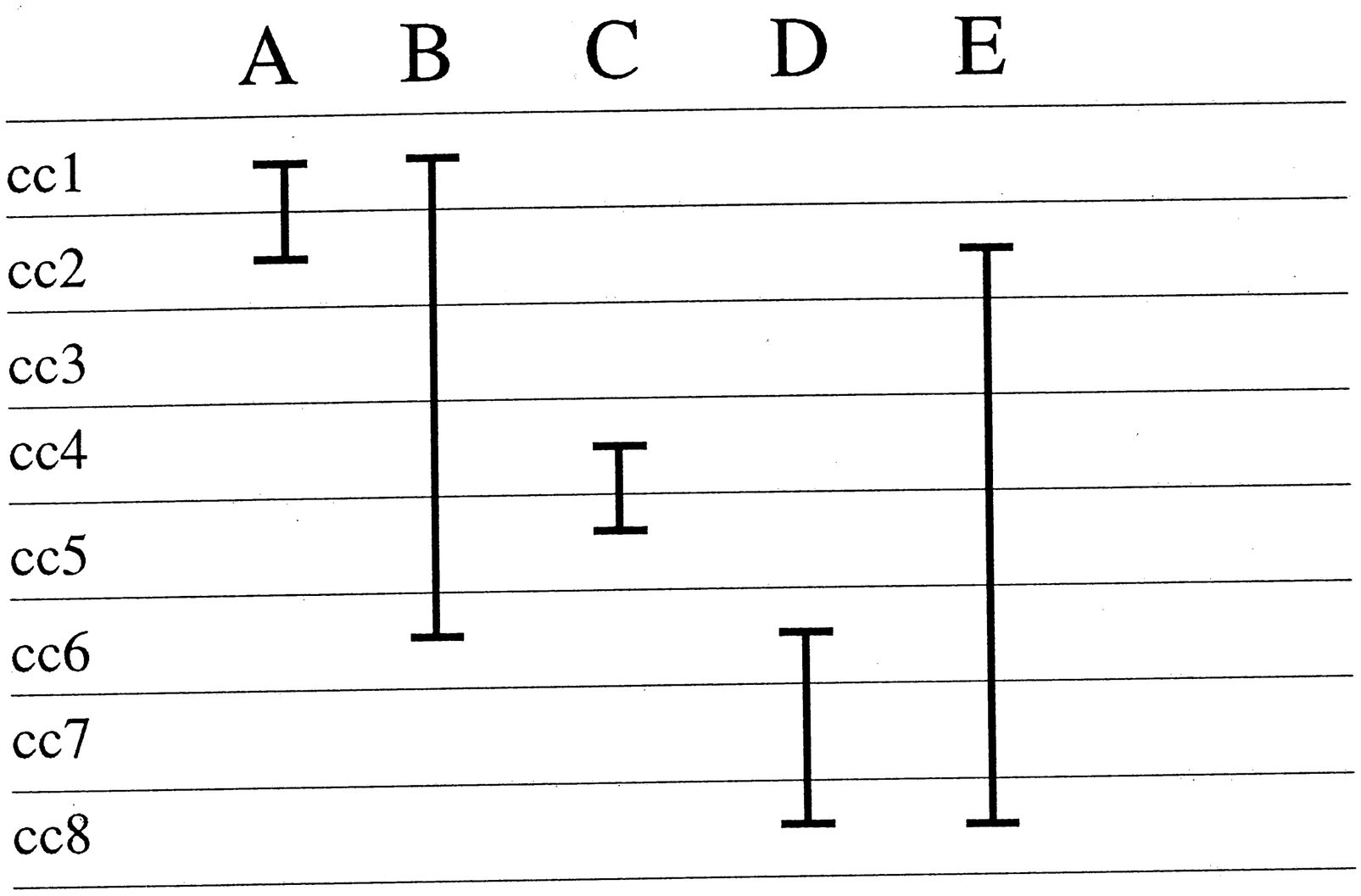


Figure 2-5: Variable Lifetimes

Allocation algorithms can also be broken up into two classes: iterative/constructive and global [18]. Iterative/constructive algorithms select an operation (or a variable or an interconnection) to be assigned, make the assignment, and then iterate the process until it is completed. How the selection is made varies from system to system. If a global criterion is used, all possible items are examined before the selection is made. What selection metric is used, of course, depends upon what is being allocated. Global seletion would tend to produce the best results but requires the most CPU time. When a local criterion is used the items are selected in some predetermined order that usually is closely related to the DFG. This requires less CPU time but the results are not as good as when global criteria are used. Algorithms of the

global class are based upon graph theory. Often the elements that are to be assigned to hardware are represented by nodes and those elements that can share the same piece of hardware have their corresponding nodes connected by arcs. The objective is to find sets of nodes that are all connected to one another. These would be elements that could share the same hardware. This is known in mathematical terms as the clique partitioning problem. The heuristics used to solve this problem, however, at best guarantee near optimal solutions.

Most of the problems described so far, including scheduling and the various allocation problems are known to be NP hard. This means that the number of steps required to find an optimal solution grows exponentially with the amount of input data. For instance, the time required to find an optimal schedule grows exponentially with the number of nodes in the DFG. This forces the use of heuristic algorithms such as the ones that have been described. All of these algorithms produce suboptimal solutions but produce them within a more acceptable time frame. To further complicate this problem, as was seen with allocation, many of the subproblems are interdependent. If this is not bad enough, the concepts of scheduling and data path allocation are themselves interdependent. The delays of the various functional units are needed in order to determine an accurate schedule. However, this cannot be known until allocation and module binding have been completed. The decision as to what operation to bind to which functional unit requires the knowledge of what operations are done in parallel or are disjoint. This can be found only from the schedule.

Up to now what has been described are the methods used to generate the data path. This is because most systems concentrate on the data path. A number of systems do generate the controller but that is the last stage of the high level synthesis process. No consideration is given during the design of the

data path as to what effect various decisions will have on the controller. As a result, it is quite possible for those systems to design a data path that meets all the area and timing specifications but that has a controller that is so large or slow that it causes the overall design to violate one or more of these specifications. The prototype synthesis system described in the remaining chapters of this thesis makes an attempt to take into account the controller's delay and area.

While not taking the controller's area and delay into account is a major oversight, it is also an intentional one. It is very difficult to estimate the delay and size of the controller until the controller has already been designed. Since the way the controller functions is entirely dependent upon the data path it is even more difficult to determine what effect a particular decision in the data path generation will have on the controller when the data path is not completely designed.

There a two different ways to build a controller. The controller can be microcoded or it can be designed as a finite state machine. A microcoded controller is relatively easy to design and a number of microcode compaction algorithms exist [12] and have been used for some time. This type of controller is, however, relatively large. A finite state machine (FSM) controller can be designed with the use of PLA's or random logic. A controller designed with PLA's will probably be smaller than a microcode controller but will also be somewhat slow. FSM controllers designed using random logic generally generate the smallest and fastest controllers. These controllers, however, take more time to generate and few systems (MIS and Socrates) exist that minimize multilevel random logic to meet both area and timing constraints.

At the present time the algorithms that exist to solve each of the individual tasks in isolation perform well. The problem is largely how to find a

good solution to the entire problem of high level synthesis while still being able to get results in a few hours. There are several other problems involving the concept as a whole. Somehow the design has to be verified. This may either entail proving that the synthesis system always produces an operable design or proving each individual design. It is also desirable to integrate all levels of design from behavioral to physical into some common data structure. Problems exist with designing systems that meet timing constraints designed to allow interfacing with other systems. Finally, integrating the design of the controller along with the design of the data path is a problem that needs to be researched.

One such solution to that problem is discussed in the the following chapters.

Chapter 3
CONTROL AND DATA PATH SYNTHESIS SYSTEM

3.1 Goals and Restrictions

The Control and Data Path Synthesis System (CaDSS) was designed as a prototype high level synthesis system that attempts to take the area and delay of the controller into account when generating the register transfer level description of its input algorithm. This is a prototype system designed for the purpose of including control in the design process. As a result, certain limitations were placed on the system in order to make it easier to develop. Most notable is that the input to the system is not a true behavioral description of the algorithm to be implemented, rather, it is an intermediate form that is more closely related to a form that can be directly translated into a DFG. The translation from a behavioral description into an optimized DFG or CDFG is not directly related to the goal of this system and was thus considered unnecessary to implement. Also, limits were placed on the design process itself. First of all, all functional units are assumed to have the same bit width which is unspecified. Secondly, only one type of each functional unit is allowed and no multi-operation arithmetic and logic units can be used. Also, all functional units must be binary in nature (they have two input strings). The system does not take advantage of the use of buses in its RTL description but, instead, relies entirely on multiplexers. Finally, in its calculations of area and delay, interconnect is not taken into account. All of these limitations were placed on the system so that its design could be expedited.

3.2 System Input

There are five input files to CaDSS. Two of the input files are technology files. They contain information about the components that will be used to generate the RTL design. The first file, functop.tech, contains the information about the various datapath operators. The information in this file is the operator's symbol, the functional unit's delay, whether or not the operation is commutative, and the area required by the functional unit. A sample technology file can be found in appendix B. It is worth noting that the functional units used for control purposes, such as various types of comparators, are treated just like any other functional unit and are thus included in this file. The file, cntrlop.tech, contains information about the various gates used to generate the controller. The format for this file along with a sample input file is also in appendix B. The primary input file is filename.dat. This is the file that contains the algorithm to be implemented. It consists of a series of binary operators and control constructs. Each assignment statement is of the form v1 = v2 op v3. Here v1, v2, and v3 are variables that can be any name up to 25 characters in length except for reserved words. Also, if the name starts with an i or an I the variable is considered to be an external input. The control constructs were limited so that the software design would be expedited. They are *if then*, *if then else*, *goto* label, label, and *if then goto* label. The names of the constructs are self explanatory. There are, however, certain restrictions on their use. No loops or conditional statements may be exited except by the normal terminating statements. Also, *if then else* statements are limited to one *else* clause. *Goto* statements should not be used except to define an infinite loop. A sample input file is found in appendix B. Another input file filename.prm contains the acceptable area and delay along with a number of other parameters. Finally, the last input file is exec.stats. It contains statistical information about loops

and conditional statements that are used in calculating the design's delay. A copy of the file along with its format can be found in appendix B. The purpose of this file will be described later on in this chapter. A diagram showing the relationships between the various files and the programs they are associated with can be found in appendix A.

3.3 The Scheduler

The program used to do the scheduling is called asap. This program uses two input files: functop.tech and intermed.dat. The file functop.tech is the previously described technology file. The file intermed.dat is a file derived from the intermediate form input file. Essentially, it is a copy of that file with the numbers of each type of functional unit used explicitly listed in the beginning. Also, in this file (as well as in the algorithm file) is a number representing the length of the clock cycle. In reality, the clock cycle may be longer or shorter than this number. Rather than the clock cycle length, what it actually specifies is the maximum delay through the data path. Thus if operations are chained together the delay of the chain will never exceed that number. Naturally, this number must be greater than or equal to the delay of the slowest functional unit plus the register delay.

One of the problems associated with generating a schedule from the intermediate form of the input file is how to handle control constructs such as branches and loops. There are several ways that branches, in particular, can be designed into the hardware. Figure 3-1 shows a CDFG that illustrates this. The fork node represents an *if then else* construct where one of two possible blocks of operations are executed depending upon whether or not v3 > v2. Despite the fact that this fork node (and later on the join node) represent control constructs, it is possible to implement this structure in the data path without

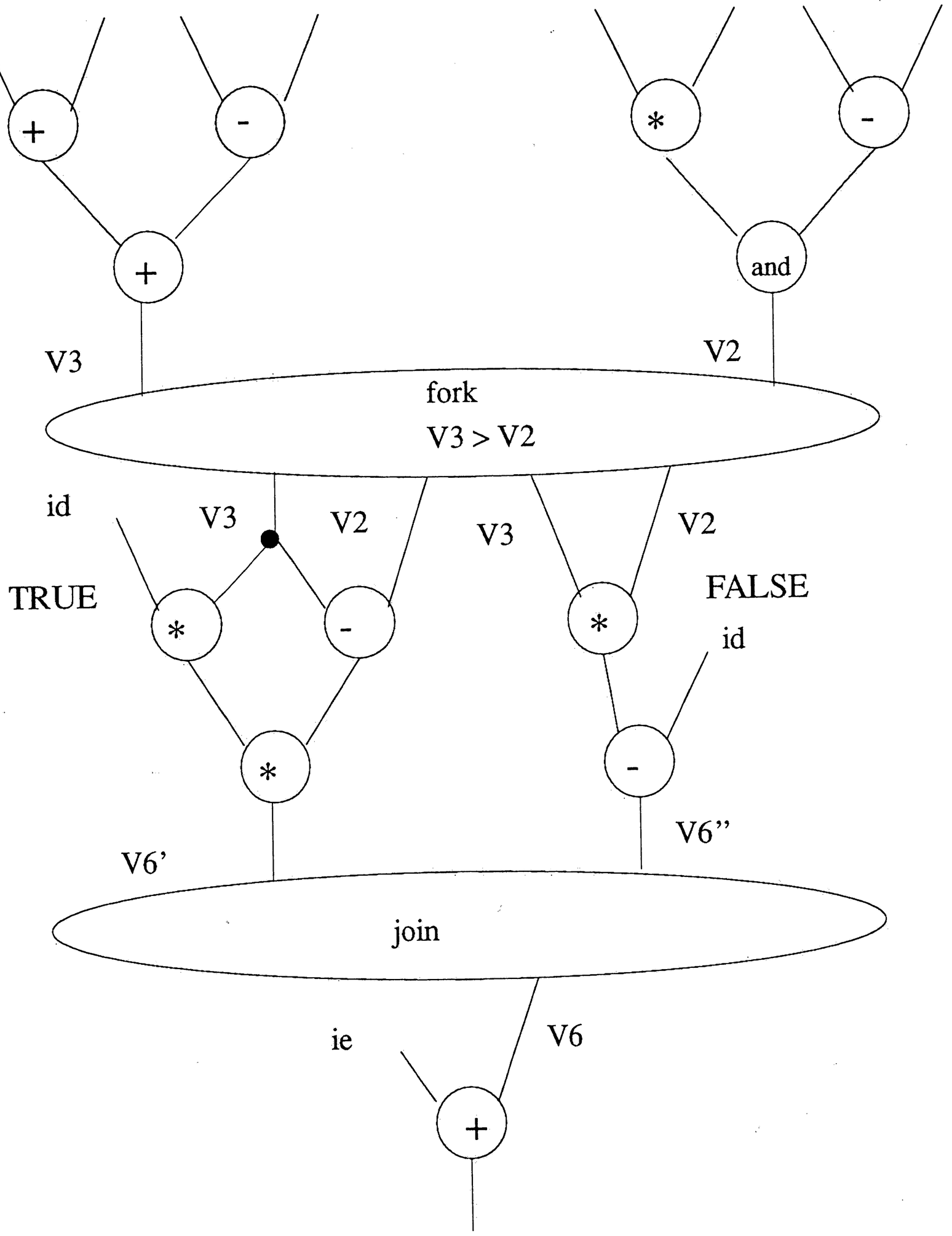


Figure 3-1: A CDFG with a Conditional

the need for the controller to be concerned about the result of the comparison between v2 and v3. Figure 3-2 shows the arrangement of a data path for such an implementation. Note that only the operations between the fork and join nodes are implemented in this data path. The operations for both possible paths of execution are carried out and the output from the proper path is selected by a multiplexer with its select line connected to a comparator. This implementation has the advantage of being very fast. It is not necessary for the result of the comparison between v2 and v3 to be known before the operations in the *if then else* structure are executed. In fact, when the variables that are used to select the branch are not used in either of the branches, the operations that generate those variables can be done in parallel with the branch. As a result, this is a very fast implementation. This implementation also has its drawbacks, however. The most obvious one is that it uses a large number of functional units thus increasing the area of the data path. The other drawback is that it can only be used on a practical basis for short branches that can be completed in one control step. For longer branches that span several control steps, resources are used to do the operations in both branches. However, here this causes a reduction in speed because rather than splitting the resources between the two branches the functional units could have been configured to handle the selected branch. All of the resources could then have been used for that branch, thus reducing the time needed to complete it. Also, if one branch is shorter than the other this method causes the delay to be at least as long as the delay of the longest branch and possibly much longer.

It was decided not to use the previously described method of integrating some of the control into the data path. Rather, the method of keeping all of the control in the controller was used. Using this method, the result of v3 > v2 must be known before any operations in the *if then else* construct can be

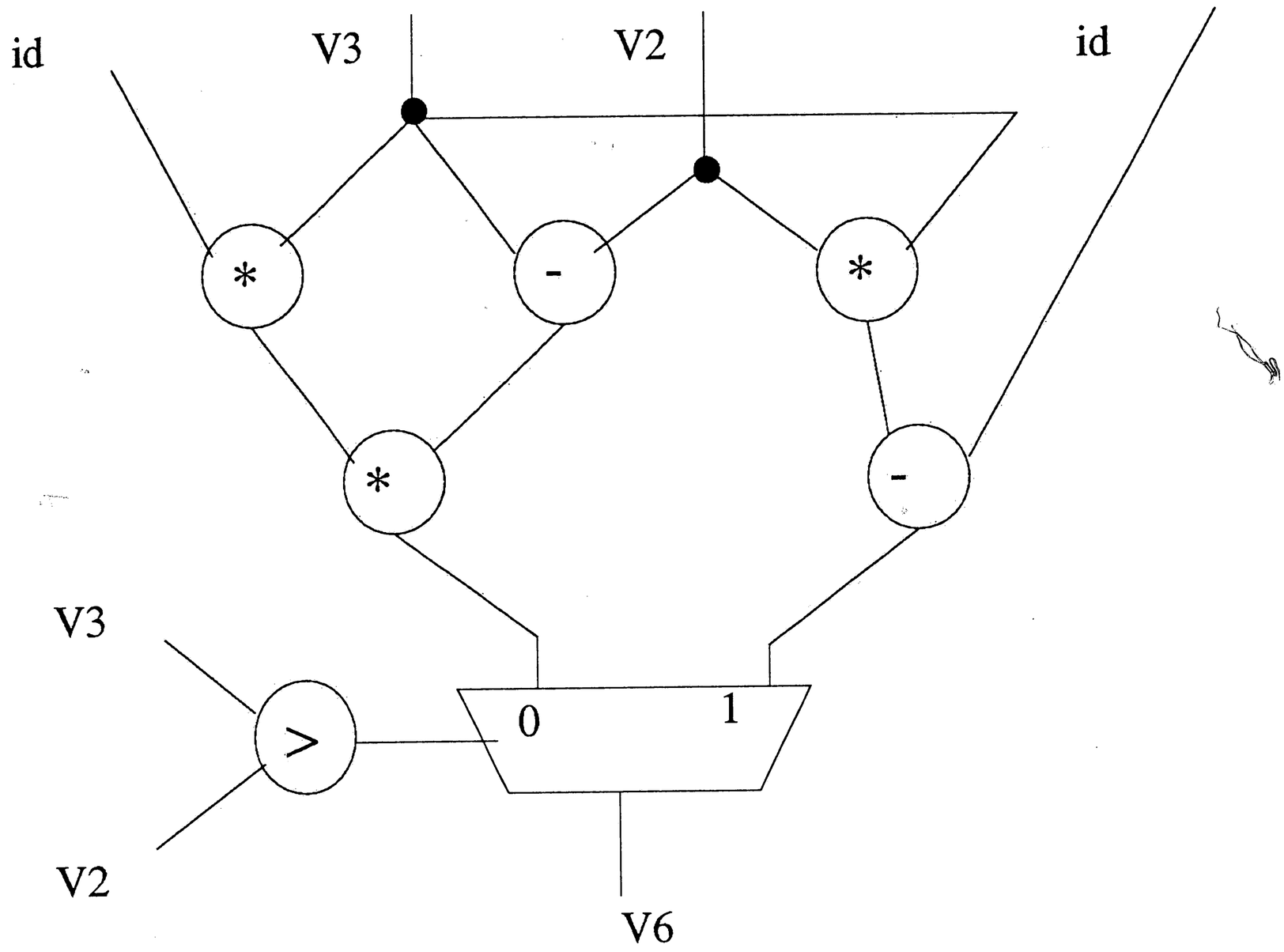


Figure 3-2: A Data Path that Includes Control

executed. Once this is known the controller will configure the datapath in the next control step to implement the correct branch. Figure 3-3 shows the data path for this type of an implementation. (Here the construct is implemented using only one clock cycle.) The controller configures the datapath by using the select lines of the multiplexers to change the inputs of the functional units. This method is somewhat slower for short branches, but for long branches it makes much more efficient use of the available resources. Also, the number of clock cycles needed to execute a given branch will be equal to the number that that branch can be scheduled into, not the number required by the longest branch.

This method also has the advantage of being easier to implement. It has its disadvantages, however. In particular, if an *if* construct has branches that are very short (can be implemented in under a clock cycle) then one entire clock cycle must be dedicated to the branch. Any operations before and after the *if* construct will be done in separate control steps. As a result, for short branches, the resources and time may be used inefficiently.

The name asap is somewhat of a misnomer. The scheduling algorithm used is actually a cross between ASAP and list scheduling. A DFG is generated and the nodes are sorted topologically. However, the nodes on each level are then sorted by some priority function. Next, nodes are scheduled level by level but, since the nodes on each level are sorted by a priority function, for each level the nodes with the highest priority are scheduled first. The priority function used here is the length of the longest path from the node to the end of the block. The advantage of this algorithm is that it has the simplicity of an ASAP algorithm yet it has some of the ability of a list scheduling algorithm to locate nodes on the critical path.

Besides scheduling, this program must also keep track of control. As the input file is read line by line a DFG is generated until a control construct (*if, else*, end of an *if* branch, *if then goto, goto*, or label) is encountered. The operation within the control condition of the *if* (if that is what was encountered) is then added to the DFG and the DFG is scheduled. A note is made that the control construct was encountered along with the clock cycle it was encountered in. If it was an *if, if then else*, or *if then goto*, it is kept track of it so that information about the various blocks of the statement can be added once they are known. The DFG is then cleared and a new one is generated from the next set of operations. At the end of the program all of the labels in the control statements are replaced by the corresponding control steps and a file known as

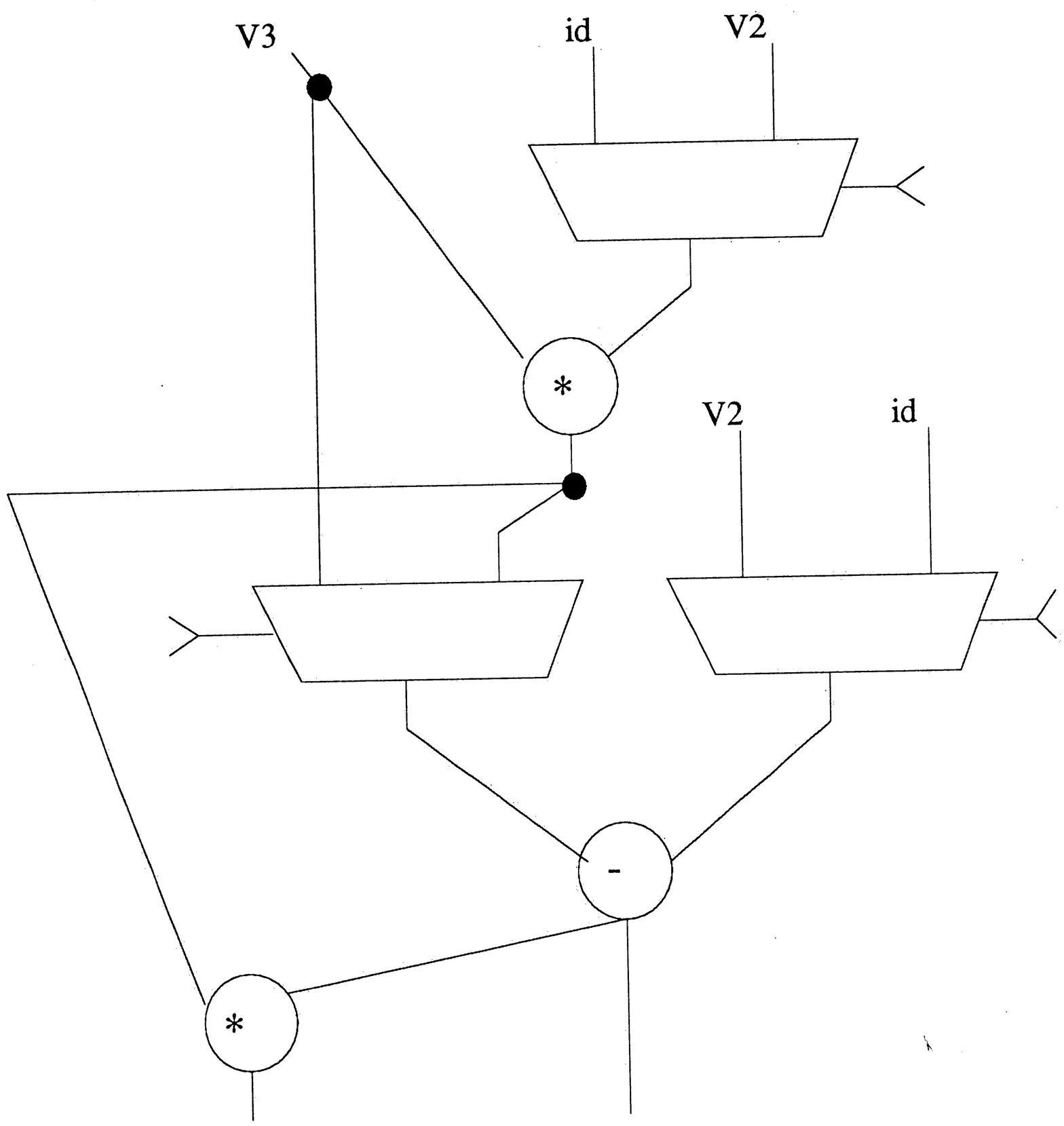


Figure 3-3: A Data Path Without Control

cntrl.int is generated from this control information. The format of the file along with a sample file is shown in appendix B. It is used to generate the circuitry that controls the next state sequencing.

It is also necessary to keep track of information about the variables that

are used. This is needed so that variable lifetimes can be determined. A variable can be used in two ways: it can be defined or it can be accessed. The two can best be described by an example. The variable v1 is defined in the following statement: v1 = v2 + ib. It is accessed in this statement: v4 = v1 - v3. Each time a node is scheduled a note is made if a variable was defined. If it was defined, the control step in which it was defined is added to a list. If the variable was accessed then the control step that it was used in is noted and is also placed on the list. Every time a variable is accessed the control step in which it is accessed replaces the last time it was accessed. When the program is completed a file called lifedat.int is generated. It consists of a list of the variable names along with a sequence of control steps during which each variable was defined or accessed. The format for this file only with a sample file can be found in appendix B.

To summarize the operation of this scheduling program, after the technology file is read, the algorithm file is read in one line at a time and graphed until a control construct (label, *if, else, if then goto*, or end of a structure ie. "}") is encountered. At that time the DFG is scheduled and the results output to a file listing the operations scheduled for each control step. As the various operations are being graphed information regarding the variables' usage is updated. Also, each time a control construct is encountered it is added to a list of the control constructs. This list contains information about the control steps in which it occurs and the control steps that it spans. Once the end of the input file is reached the last block of statements is scheduled. All of the labels used in the control statements are replaced with their corresponding control step. Finally, the control and variable information are output to their appropriate files. A flow chart describing the operation of this program can be found in appendix A.

3.4 Variable Lifetime Determination

The next step in the synthesis process is to determine exactly what the lifetimes of each variable are. The lifetime of a variable is the period of time that starts with the first time it is defined and continues until the last time it is accessed. It is necessary to know the lifetimes of all the variables because those variables with disjoint lifetimes can share a common register. This eliminates the need for the existence of one register per variable. For code without any control constructs the lifetime of a variable becomes obvious; however, with control constructs it is slightly more complex. The existence of loops and conditional statements has an effect upon the lifetimes of variables.

Loops present the biggest problem in determining the lifetime of a variable. If a variable is used but not defined within a loop the lifetime of the variable must extend at least to the end of the loop. This is because, since in a loop several control steps may be repeated, the variable must stay alive in those control steps so that it is available in the control steps that do access it. This can best be shown by example. In figure 3-4 variable v1 is first defined within loop number 1. It is not used inside loop 3 but it is used inside loop 4. Since loop 2 contains loop 4 that has a statement that accesses v1, loop 2 does in fact contain a statement that accesses v1. The numbers on the right hand side of figure 3-4 represent the control steps during which loops begin and end. Since loop 2 is the outermost loop that contains an access of v1, but not a definition, in order for v1 to still be available in successive iterations of loop 2 v1 must be alive for the entire span of loop 2. Thus v1's lifetime starts in control step 2 when it is defined and ends at control step 24 when loop 2 ends.

If and *if then else* constructs cause fewer problems. If a variable is used in both branches of an *if then else* statement then its lifetime must consist of at least part of both branches. If it is used in only one branch then it only has to

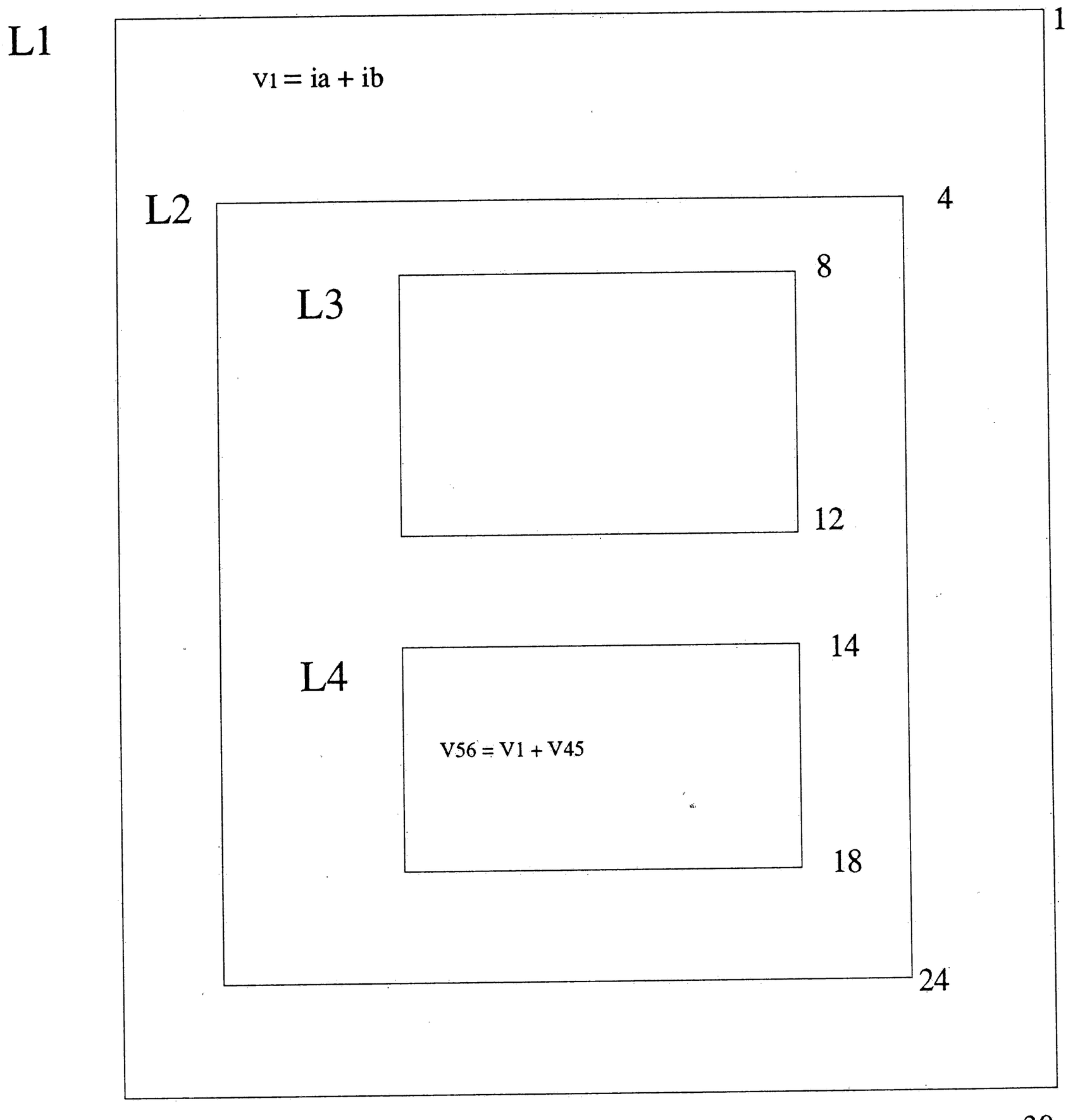


Figure 3-4: Illustration of Variable Lifetime Evalution for Loops
(L1 ... L4 represent loop bodies and 1 ... 30 represent control steps)

exist over the number of control steps spanned by that branch. (Note, however, that if the variable is used after the construct its lifetime will necessarily span

both branches.) In figure 3-5 variable v1 is defined in control step 1 and is used in both branches of **if2**. It is used in only one branch of **if1** and, as a result, its lifetime does not have to be in the *else* clause of **if1**. Here, minimally, it could have a lifetime from control steps 1 through 7, and 12 through 13. This is one lifetime, not two, since only one branch or the other of **if2** is executed. To make it easier to represent, however, the lifetime, in this program, is calculated as control steps 1 through 13 which is somewhat less efficient.

There is one final problem of a variable having multiple lifetimes. If a variable is defined and then accessed within several control steps and then not used for several control steps and then redefined, it will have two disjoint lifetimes. This is illustrated in figure 3-6 where v1 has lifetimes 3 through 10 and 14 through 22. Note, however, that if its second definition was v1 = v2 + v1, then it would only have one lifetime because that statement requires a usage of v1.

The program genlife uses two input files and generates one output file. The input files are cntrl.int (which contains information about loops and conditional statements) and lifedat.int, both of which are output files of the scheduler. Its output file is lifetimes.int. The format for this file and a sample file can be found in appendix B. Notice that, since the external inputs are never assigned to registers, they have zero lifetimes. The lifetimes listed for cntrl_cond can also be ignored. Cntrl_cond is a variable that represents the outcome of a comparison in a conditional statement and is never stored in a register.

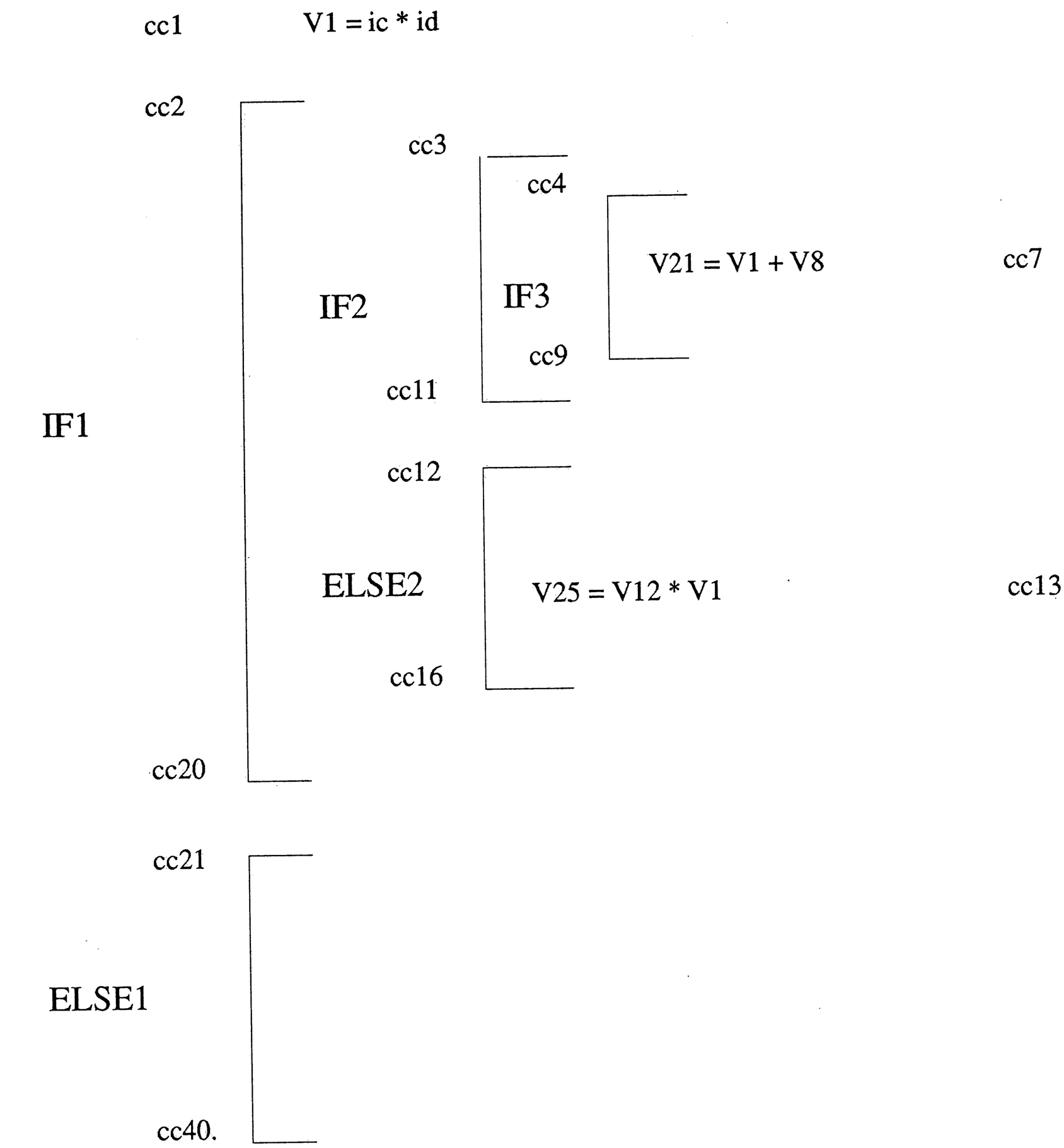


Figure 3-5: Illustration of Variable Lifetime Evalution for *If* Constructs

cc2 v1 = ie * ih

●
●
●

cc10 v20 = v1 - v19

●
●
●

cc14 v1 = v30 + v28

●
●
●

cc21 v45 = v43 - v1

Figure 3-6: Illustration of Variable Lifetime Evalution for a Variable with Multiple Lifetimes

3.5 Register Allocation

Register allocation is the assignment of variables to registers with the goal of minimizing some cost function. The program used in this system, regal, attempts to minimize the number of registers used to store the variables. The algorithm used to perform the allocation was adapted from REAL, [15] a register allocation program developed by Tadi Kurdahi and Alice Parker of the University of Southern California. The algorithm they developed is in turn an adaptation of the left edge algorithm that is used for channel routing. This is a simple algorithm that works as follows. The wire segments are sorted in increasing order of the coordinates of their left edges. The first wire segment is

assigned to the first track. The first wire that does not overlap the wire just assigned to the track is also assigned to that track. When no more wires can be assigned to the track a new track is used until it is also full. The algorithm is completed when all of the wires have been assigned to a track. The algorithm used by REAL works exactly the same way except that rather than wires and tracks, variables (whose starting and ending coordinates are their birth and death times) and registers are used. It has been shown that, for a DFG with no conditional branches, the algorithm is optimal and is probably optimal for the modification made for its use with conditional branches. Figure 3-7 gives an example of the use of this algorithm. The left hand side of figure 3-7 shows a sorted list of variable lifetimes. The right hand side shows the register assignments after the use of the algorithm.

Regal works in an almost identical manner. The input to regal is lifetimes.int. The variable lifetimes are read in and sorted. When a variable has multiple lifetimes each lifetime is essentially treated as a separate variable (as in effect they are) and may be assigned to different registers. The sort is done by increasing order of the birth times. The algorithm then proceeds as before with variables with disjoint lifetimes acting as nonoverlapping wires. Conditional branches are not a problem here since each branch is assigned to separate groups of control steps. Thus, as far as the algorithm is concerned, it does not see any conditional branches. The output of this program consists of two files: varassign.int and regassign.int. The file varassign.int consists of a list of variables. For each variable is given a list of the registers it is assigned to and the lifetime for which it is assigned to each register. The file regassign.int gives a list of registers. For each register is given a list of the variables assigned to it and the lifetimes for which each variable is assigned to that register. A sample of each of these files along with their formats is given in appendix B.

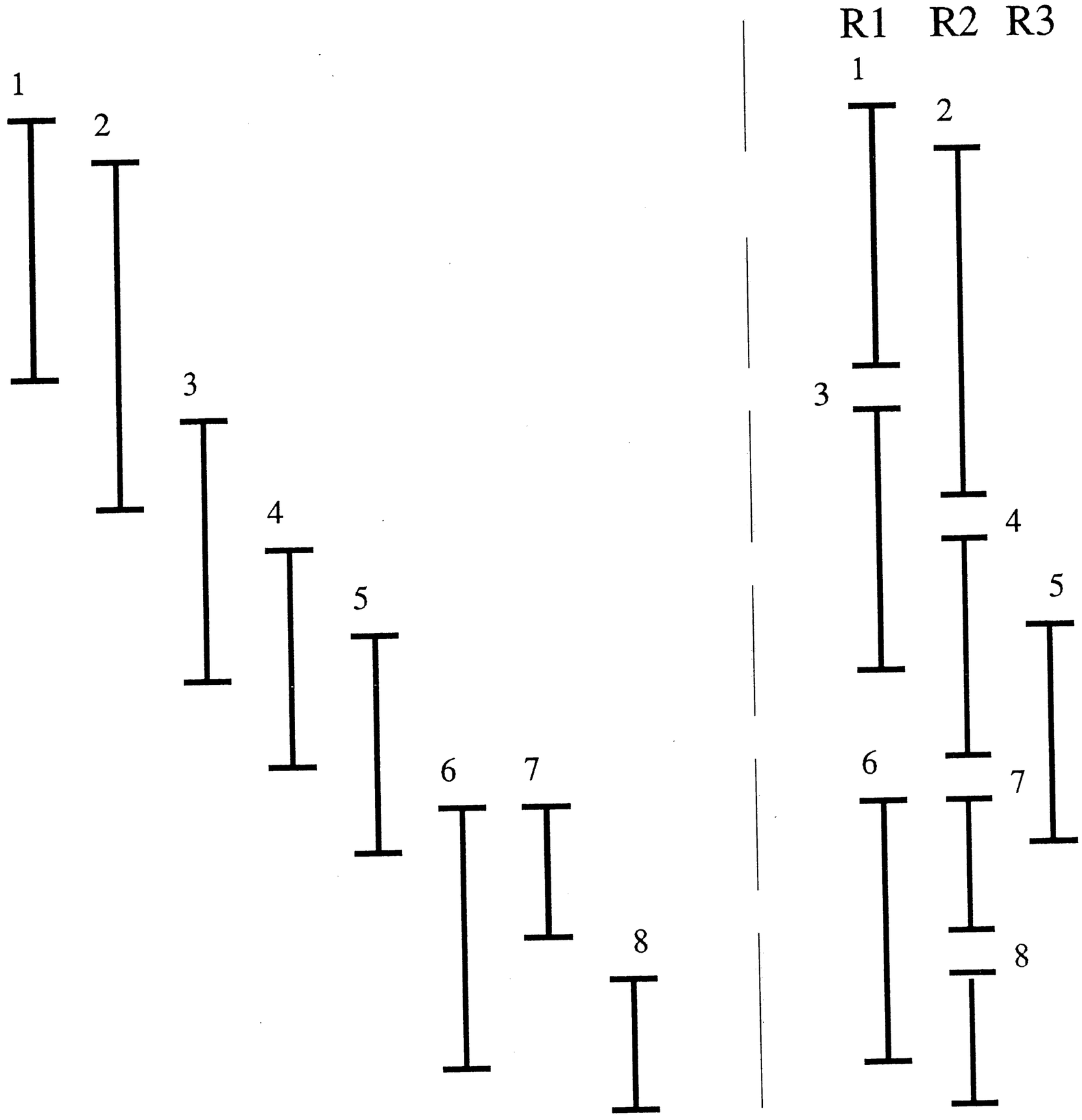


Figure 3-7: Example of Register Allocation Using the REAL Algorithm

3.6 Data Path Generation

The next step in the process is to use the schedule and the register allocations to generate a data path that is capable of implementing the schedule. Since this synthesis system is not attempting to generate a specialized architecture, the data path should idealy be made as general as possible. That is, as few restrictions as possible should be placed on the form of the data path. To limit the complexity of the system, however, it was necessary to place some limits on the structure of the data path. One restriction is that no buses are used, and as a result all wiring must go through multiplexers. Secondly, only one type of functional unit may be used to implement each binary operator. For example, a carry propagate adder or a carry look ahead adder may be used but both cannot be used in the same design. This simplifies the problem of operation binding.

These restrictions do, to some extent, force the data path into a particular format (although it is a general one). Each functional unit has a multiplexer at each of its inputs. The multiplexer select lines determine what the inputs to the functional unit are. Each register also must have a multiplexer at its input. This is unusual, but the original premise was not to use bus structures. The inputs to the multiplexers can be either external inputs, outputs of functional units, or outputs of registers. The controller is responsible for determining the settings of the select lines as well as the load lines of the registers. Figure 3-8 shows what the general format looks like. Also, in appendix B is an RTL diagram of the data path that was generated for the example used in appendix B up to this point. One part of the data path should be noted to be fairly specific. That part is the circuitry used to determine the value of cntrl_cond. As mentioned earlier, cntrl_cond represents the result of some comparison done in a conditional statement. This result must be fed back to the controller to

determine what the next state will be. Only a single bit is sent back to the controller. The multiplexer is used to determine which of the functional units (these functional units should be types of comparators) is being used to determine cntrl_cond for the present control step.

The datapath generator program (DPG) uses several input files. The most obvious and important of these is the schedule file, intermed.out. This file, which is the output of the scheduler, contains a list of control steps and the operations scheduled to be performed during each step. Also used as an input is intermed.dat (the input to the scheduler). It is used only to determine how many of each type of functional unit is available. The technology file, functop.tech, is needed. This, besides defining the type of operations available, will also be used to provide information on whether or not the operations are commutative. Finally, both varassign.int and regassign.int are used. Despite the fact that they contain identical information, both are necessary because the two different formats are convenient for accessing the information in different ways. For instance, varassign.int is useful for determining which register a variable is assigned to during a given control step; while regassign.int is useful for determining during what control steps a register contains no useful information (ie. its load line can be a don't care).

The actual flow of the program is fairly simple. For each control step the operations that are to be allocated to functional units are read in from the schedule file and graphed (placed into a DFG). When the end of the control step is reached, the operations are allocated to functional units, the interconnect of the functional units is determined, and the control information is determined. After all of the operations have been allocated the registers are checked to see if any are not holding a variable and their load lines are set to don't cares. Then the information about the control (multiplexer selects and register load lines) for

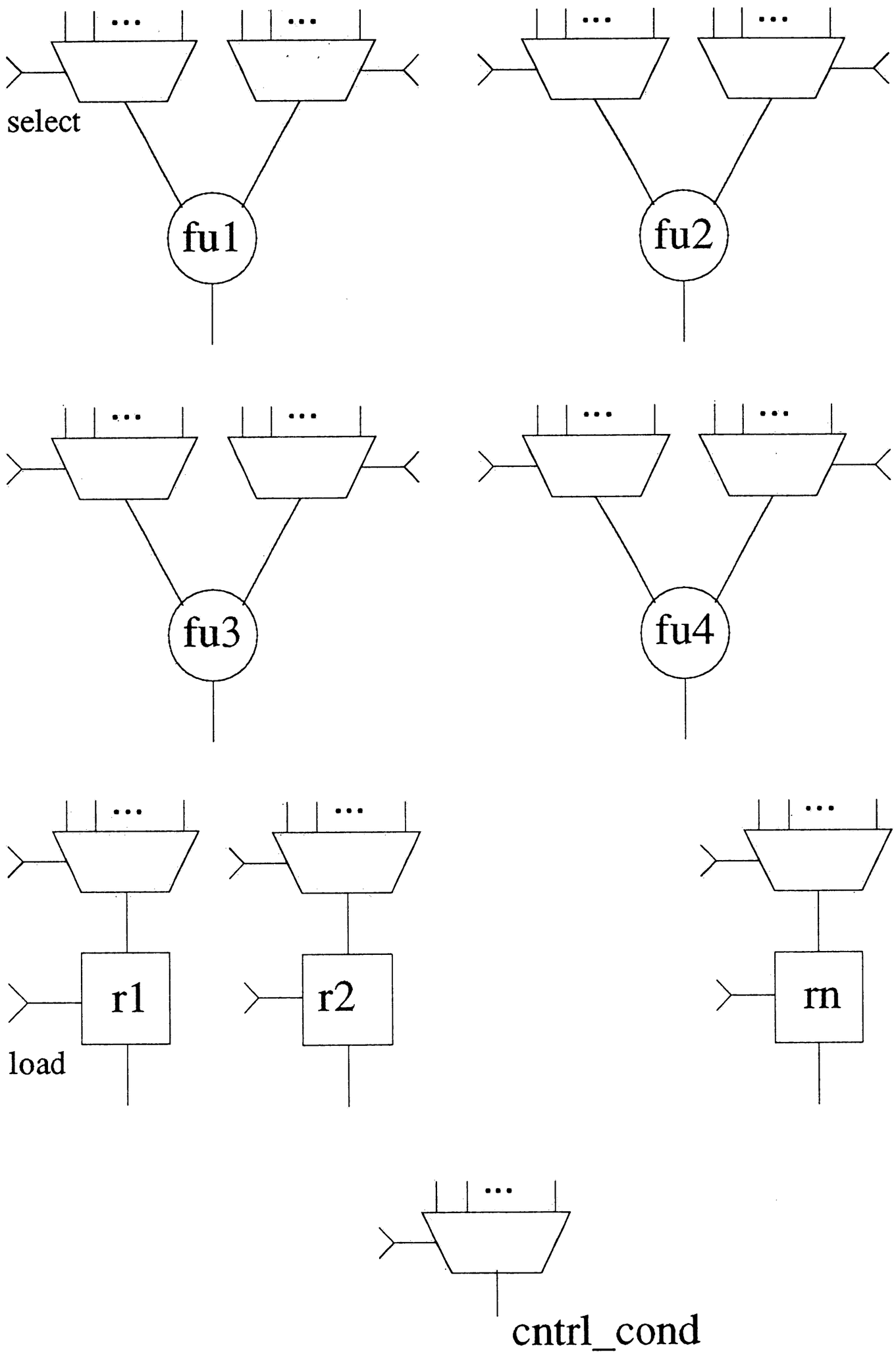


Figure 3-8: Data Path Generated by CaDSS

that control step is output to a file combo2.int. This process is repeated until there are no more control steps for which allocation is to be performed. Finally, the specification for the data path is output to the file datapath.spec. The formats and examples of both of these files can be found in appendix B. Also in appendix A is a flow chart showing the flow of this program.

The question that remains to be answered about DPG is how is the allocation actually performed? The first step is to use the information that is already known about the data path to set up a data structure that will be used to represent the data path. The information that is known at the start of execution of the DPG is how many of each type of functional unit will be used and how many registers will be used. The data structure that is set up to represent that data path is shown in figure 3-9. The structure consists of 3 main substructures: f_u_type, f_u_node_type, and f_u_edge_type. Initially, an array of n of the f_u_type substructure is generated. Here n is the number of different types of functional units plus one. This structure contains information about each type of functional unit such as its symbol, whether or not it is commutative, and how many of that type of functional unit will be used. Element zero of the array refers to the registers. Each of these substructures also points to an array of the substructure f_u_node_type. This array has the same number of elements as there are functional units of that type. There is one extra element in the register array. That extra element is used to represent cntrl_cond which has much the same characteristics as a register. The f_u_node_type substructure contains information that is used both in control generation and data path generation. It has three fields for the number of inputs to the multiplexers on its left and right inputs and for the number of outputs. Also, there are two fields to determine what the multiplexer select lines should be in the current control step. There is a field used only for

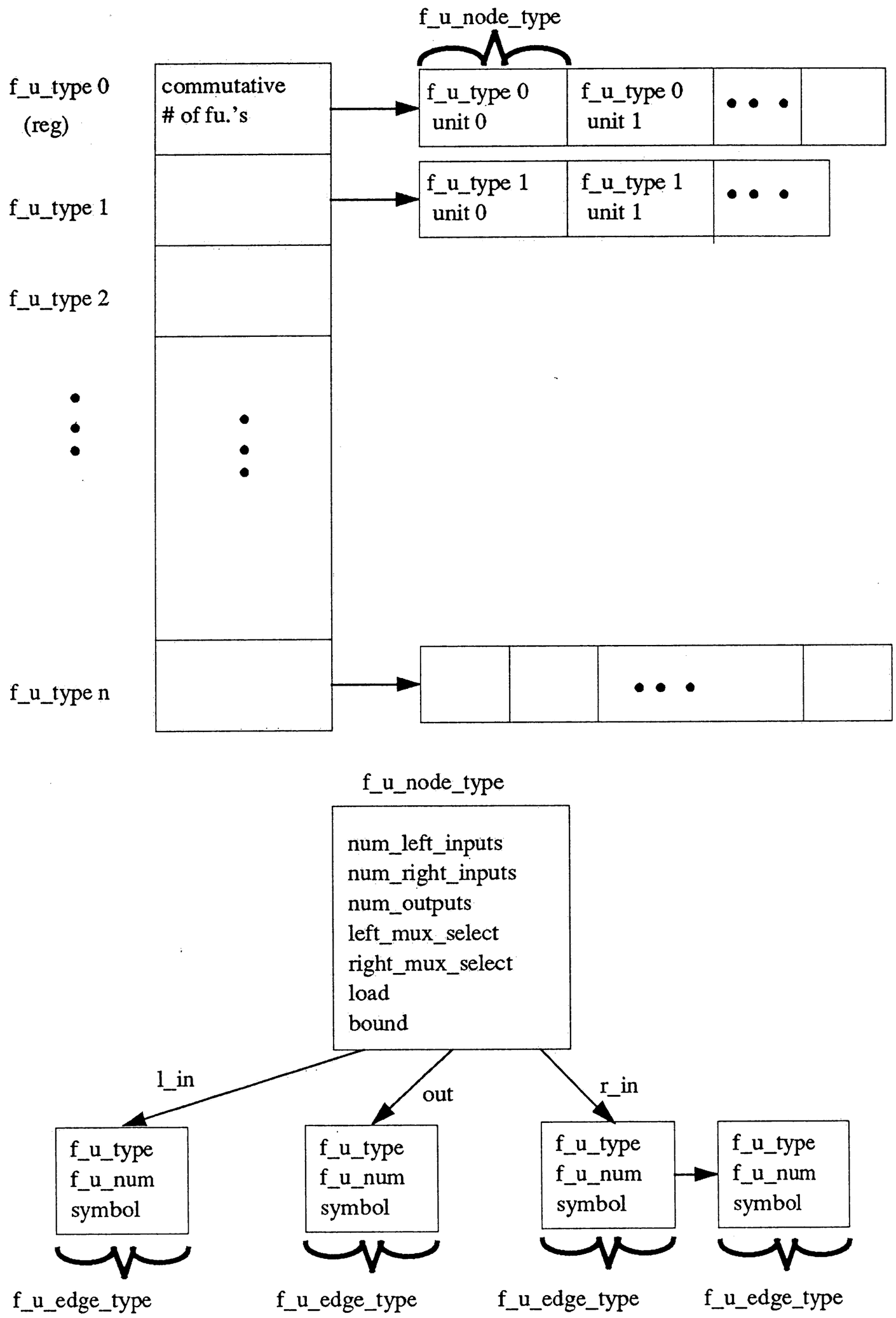


Figure 3-9: Data Structure Used by DPG to Represent the Generated Data Path

registers to determine if the register should be loaded. Finally, there is a binary field that is true if an operation has been bound to the functional unit in the current control step. The substructure also points to three linked lists of the substructure f_u_edge_type that determines what the various multiplexer inputs are. These linked lists are actually generated as the data path is determined. The substructure f_u_edge_type has two fields to represent the functional unit or register it is connected to. These two fields are the functional unit type and the functional unit number. If an input to the multiplexer is an external input then a field that points to its symbol in a symbol table is used.

The program starts with the DFG for the control step it is working on. The nodes are examined one node at a time starting with the nodes on the first topological level and working toward the bottom of the DFG. For each node it determines what type of functional unit is necessary to implement it (a fairly obvious decision) and then determines which of the various functional units of that type should be used. This is done by checking the linked lists of f_u_edge type for each functional unit and finding out how many connections already exist on the unit. The output is also checked to see if an appropriate register connection exists and how many of the output connections go to the appropriate type of functional unit. If the operation is commutative the left and right inputs of the operation are swapped and the functional units are rechecked. The operation is then bound to the functional unit with the most similar connections. Any connections that must be added to that functional unit are then added by adding an f_u_edge_type substructure to the end of the appropriate linked list (l_in, r_in, or out). The number of inputs and outputs are then updated and the multiplexer select settings are determined by searching through the linked lists. If the output of the functional unit goes to a register then its inputs and multiplexer settings are updated and its load field is set to 1. Once all of the

nodes in the DFG have been processed, the registers are checked to see if they are currently holding any variables. If not, their load lines are set to a don't care state. The control information is then output to the file combo2.int. The control information consists of multiplexer select line settings and load line values. Before the next control step is processed the DFG nodes are freed and all of the control information is set back to don't care values except for the load lines which are set to 0 (do not load).

The algorithm just described attempts to minimize interconnect by minimizing the size of the multiplexers. The algorithm is not ideal and, as a result, does not guarantee optimal results. When it checks output connections it can only check to see that the present functional unit has connections to a functional unit of the proper type since it has no way of knowing what functional unit that operation will actually be assigned to. Also, it does not check to see what effect binding an operation to a particular functional unit will have on the other functional units that have yet to have operations bound to them. Furthermore, since initially all of the functional units have no connections the majority of the operations are assigned to the first functional unit of the proper type. Thus the first functional unit of each type tends to have the largest input multiplexers.

3.7 Control Generation

The next step is the generation of a control unit. The purpose of the control unit is to determine the configuration of the data path from one control step to the next. For the data path being used this simply means determining the values of the multiplexer select lines as well as the register load lines. As mentioned earlier the control unit can be designed either using a microcode ROM or using a finite state machine. This system uses a finite state machine

for the controller. Again, it is best to restrict the architecture of the controller as little as possible.

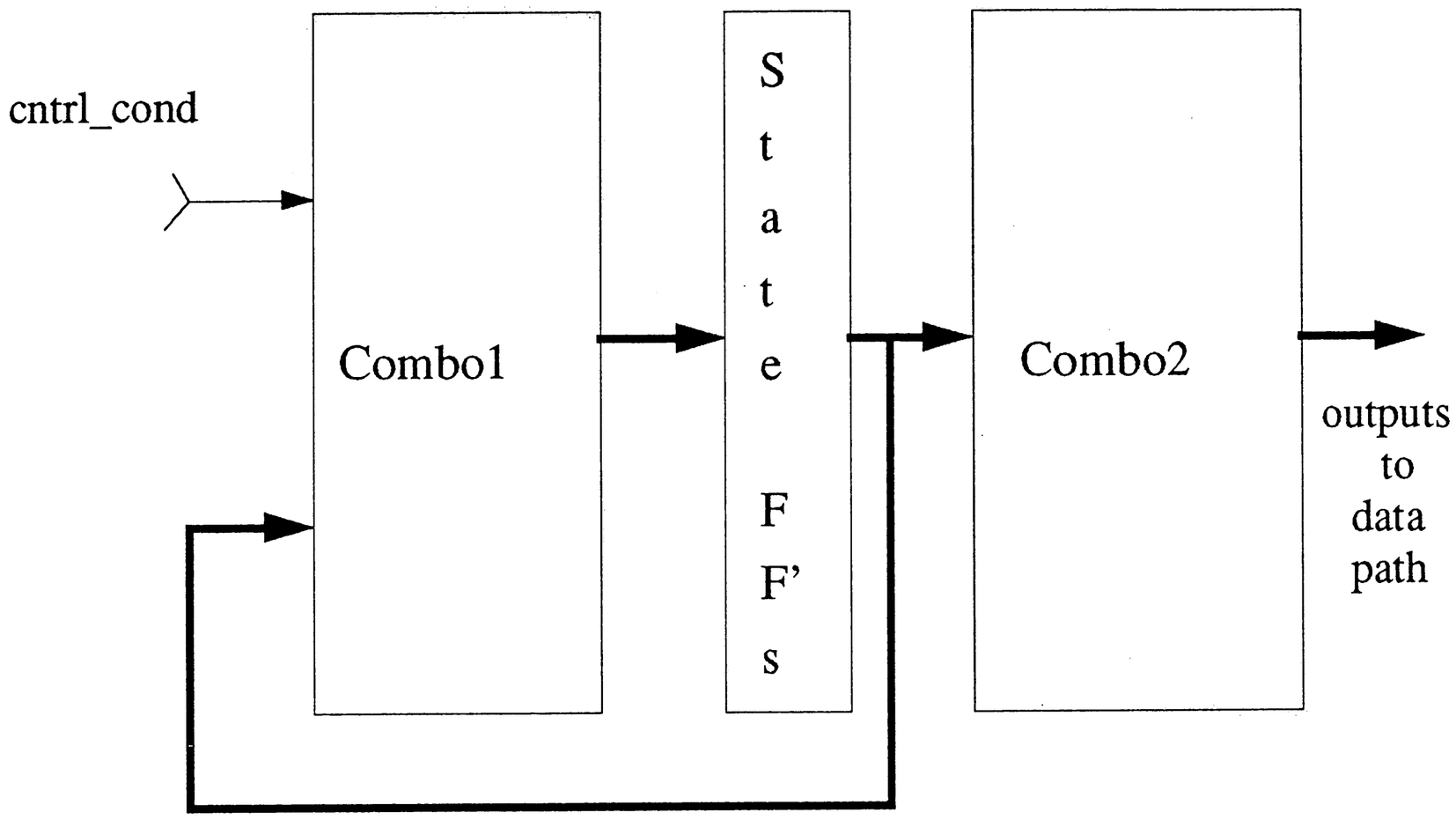


Figure 3-10: Representatiion of the Controller FSM Generated by CaDSS

Figure 3-10 shows the design of the controller used by CaDSS. Combo1 and Combo2 are blocks of combinational logic and the state flip flops hold the present control step number. Combo1 is used to determine the next control step. Its inputs are the present control step and cntrl_cond which is the result of the conditional statement that is presently being evaluated. This is usually a fairly small logic block in comparison to Combo2. Figure 3-11 shows a small segment of code broken up into control steps. If v3 > v2 then Combo1 sequences through the control steps as follows: cc1, cc2, cc3, cc5 and cc6. If v3 ≤ v2 then the sequence is as follows: cc1, cc2, cc4, cc5, and cc6. Combo2 determines the

configuration of the data path. Each control step will have its own data path configuration. Combo2 has one output for each register load line and a series of outputs for each multiplexer select in the data path.

cc1	v1 = ia * ib v2 = v1 + ic
cc2	v3 = v2 - v1 v4 = v3 + ia

cc3	if v3 > v2 then {
	v5 = v4 + v3
	v6 = v5 * v1
	}

cc4	else {
	v5 = v4 - v3
	v6 = v5 * v2
	}

cc5	v7 = v5 + v6 v8 = v7 * ic
cc6	v9 = v8 - v4 v10 = v7 + v8

Figure 3-11: Schedule of a Code Segment

The two control files, cntrl.int and combo2.int are all of the files that are necessary to develop a specification of the logic used to implement Combo1 and Combo2. The PLA minimization program ESPRESSO [6], which was developed at the University of California at Berkely, is used to generate the logic descriptions. Cntrl.int contains all of the pertinent information needed to generate combo1. The control steps are assumed to change sequentially unless some form of control statement implies otherwise. These statements are described in cntrl.int. Combo2.int contains the multiplexer select line settings as well as the load line settings for each control step. The programs cntrl2esp and c2toesp convert cntrl.int to combo1.esin and combo2.int to combo2.esin. The two .esin files are both in a format that is readable by ESPRESSO. When the conversions are made the various don't care conditions are converted to an ESPRESSO readable format also. Samples of the files along with their formats are in appendix B. ESPRESSO is then used to generate a minimized two level description of the logic circuits. At the conclusion of this step a design that implements the functionality of the input algorithm exists.

3.8 Area and Delay Calculations

Now that a design exists that meets the algorithm's functionality it is necessary to determine whether it meets the area and speed specifications. This involves several steps. First, the area and the delay of the controller must be calculated. Second, the area and delay for the data path must be calculated. At this point it is possible to determine whether or not the design meets the area requirements. The speed requirements on this system however are based on an overall expected delay for the execution of the entire algorithm to be implemented. This will be described in more detail shortly.

The controller area and delay is calculated first. These two costs are, of

course, dependent upon how the combinational logic blocks are implemented. ESPRESSO does a minimization for a PLA. This forces the use of either a PLA or a two level design with random logic gates. The latter was chosen because of the availability of information on the AT&T 1.25um standard cell library. It is assumed that the inputs to the controller have input buffers and the outputs to the data path use output buffers. Inside the combinational logic blocks only NAND gates, NOR gates, and inverters are used. The gates have either 2, 3, or 4 inputs. If any gate requires more than 4 inputs it is broken up into several gates using multiple levels that implement the same function. In general, the logic is implemented just as a PLA would be implemented with NAND gates used for both planes. Figure 3-12 shows a combinational logic circuit and how it would be implemented by CaDSS. When a gate must be broken up because it is too large the gates that replace it will alternate between NAND and NOR gates. If it ends up with a NOR gate at the final level then the NOR must be followed by an inverter.

The program that does the area and delay calculations is combocost. It uses as its input the ESPRESSO output files combo1.spec and combo2.spec as well as a technology file defining the controller gates, cntrlop.tech. The technology file contains information about the size of the various gates as well as their delays. The format of this file can be found in appendix B along with an example of the file. The delay specification consists of two numbers: a base delay and a delay factor. The base delay is the delay for a gate with a fanout of 1 and the delay factor is the increase in delay for each additional fanout. Thus the total delay is calculated as follows:

$$\text{delay} = \text{base\_delay} + (\text{fanout} - 1) * \text{delay\_factor}.$$

This formula was determined from the AT&T 1.25um standard cell catalogue

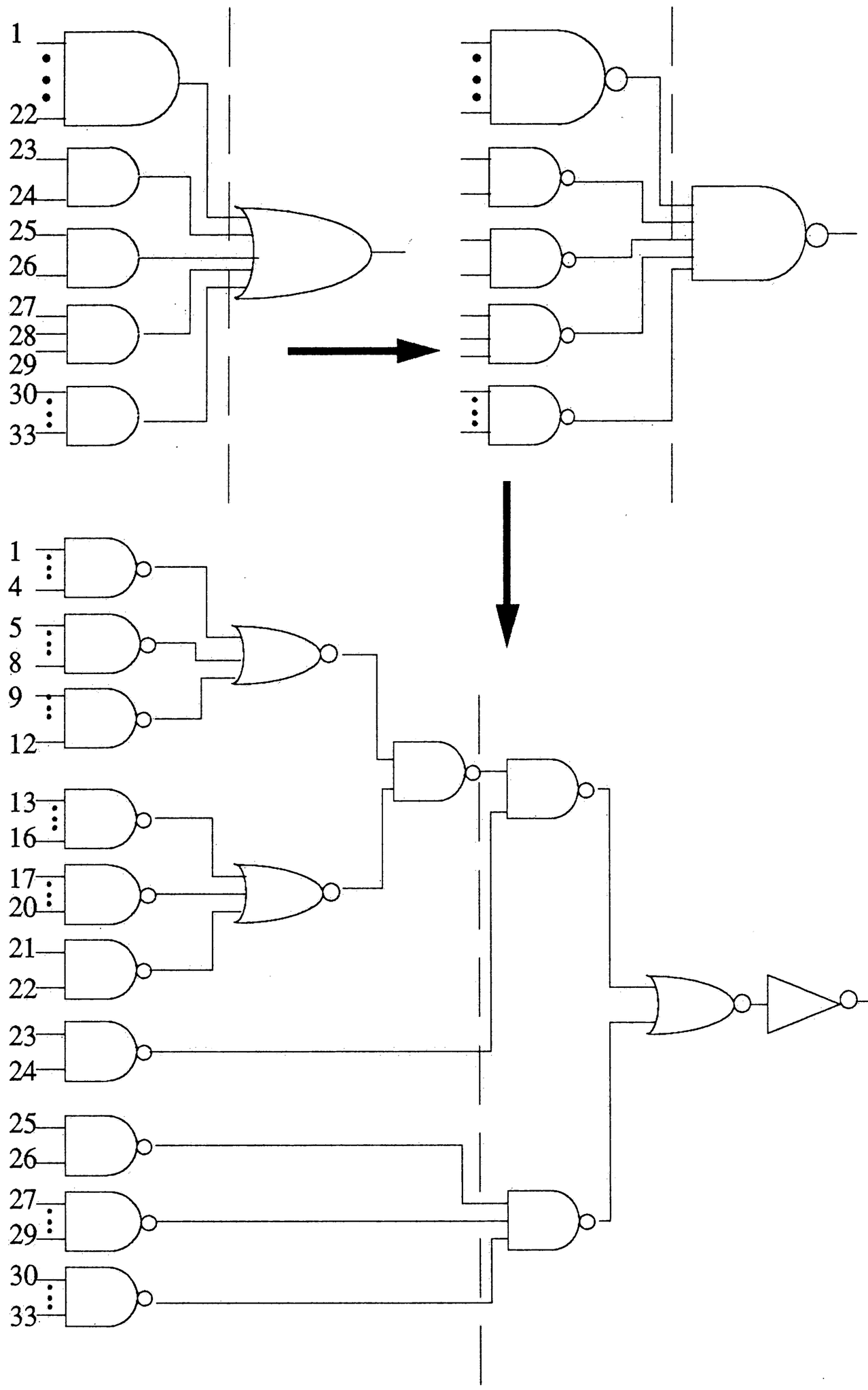


Figure 3-12: CaDSS' Implementation of a Combinational Logic Circuit

which uses standard cells whose published delays apparently increase linearly with fanout.

Combocost first checks the output of ESPRESSO to see which outputs have identical functions. It then eliminates the duplicate functions. Once that is done, it proceeds to break the gates up as described previously and calculates delays and areas taking fanout into account. The outputs of each combinational logic block are assumed to have a fanout of 1. The total area and delay of the combinational logic block is then output to the file stats.spec. See appendix B for this file.

The next step is to calculate the area of the datapath and add that to stats.spec. This is done by dparea. It does this by simply adding up the area of the functional units (functop.tech contains the area for each functional unit). Interconnect is not taken into account when calculating either the area or the delay of the data path. The delay of the data path was already calculated by asap and appended to stats.spec. This was done in asap because it was already necessary for it to keep track of the delays to make sure that the delay of the data path did not exceed its maximum allowable delay.

Next, the total area of the design and the minimum clock cycle length must be calculated. This is done by the program exectime. The total area is simply the area of the datapath plus the two combinational logic circuits plus the flip flops used to implement the state register in the controller. The clock cycle length is the delay of the controller (delay of combo 1, combo 2, and the state flip flops) plus the delay of the data path.

The program exectime also performs one other function, which is, in fact, its main function. The function is to determine the total execution delay of the algorithm. The delay that the synthesis system tries to meet is the delay of the execution of the total algorithm. In the file exec.stats is a list of the loops and

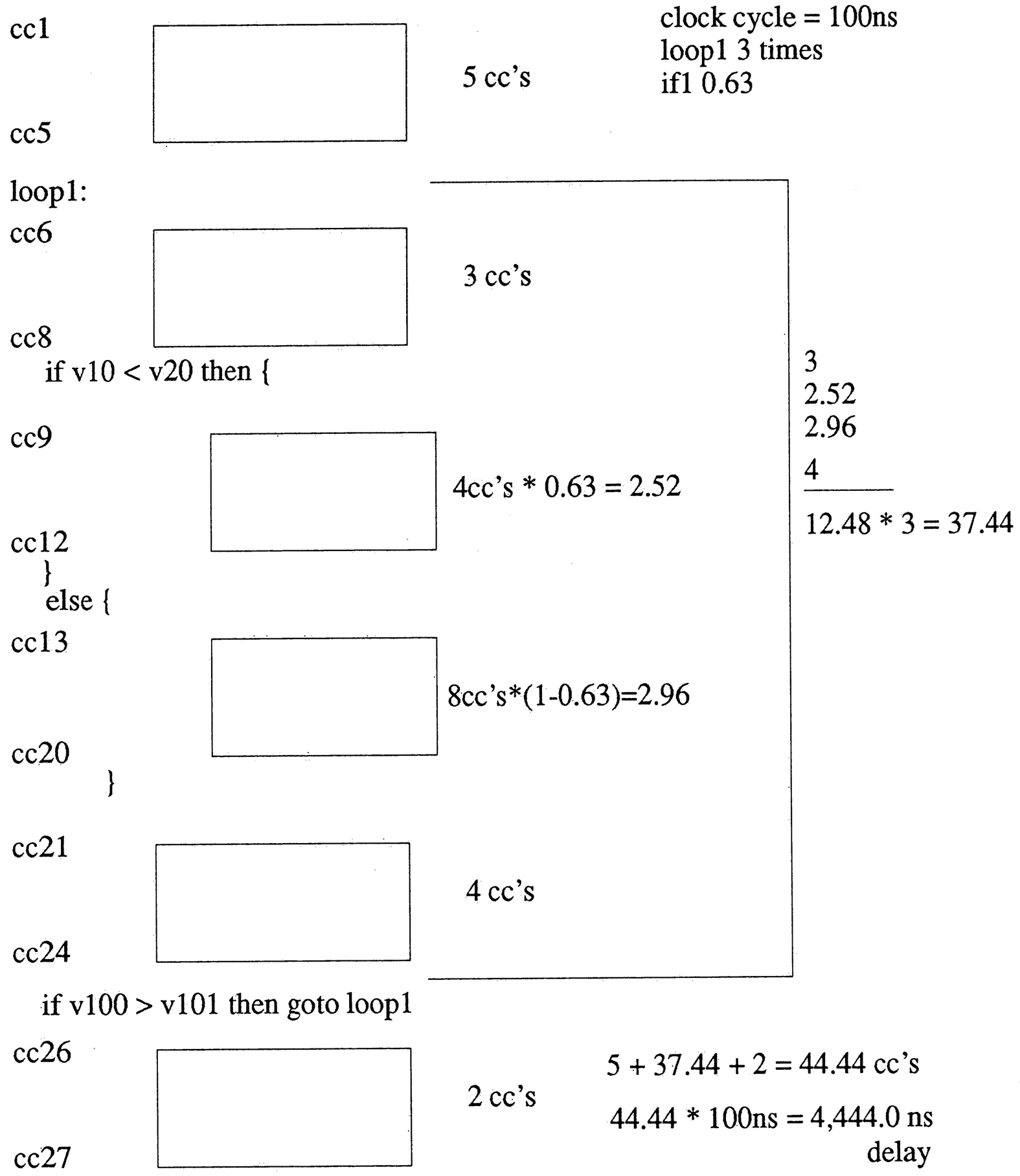


Figure 3-13: Sample Algorithm Block and its Delay as Calculated by CaDSS

the average number of times they are expected to be executed. The loops are listed in increasing order of their starting points. Also listed in increasing order are the conditional statements *if then* and *if then else*. For the conditional

statements the probability that the true branch will be executed is given. For loops, the delay of the loop is calculated by multiplying the number of control steps within the loop by the number of times that loop is executed followed by the length of the clock cycle. For an *if* statement, the delay is the number of control steps needed to execute the contents of the *if* multiplied by the probability of its execution multiplied by the clock cycle length. For an *if then else*, the delay is the sum of the of probability that the true branch will be executed multiplied by the number of control steps within it and the probability that the false branch will be executed multiplied by the number of control steps in the false branch. This is again multiplied by the clock cycle length. Figure 3-13 gives a sample algorithm block and its delay.

The reason for calculating the algorithm delay in this manner rather than simply calculating the length of the clock cycle is that that does not give very much information about the design's performance. If very few functional units are used in the data path then the clock cycle may be very short but it will require many clock cycles to execute the entire algorithm. If many functional units are used, the length of the clock cycle may be very long but it will then take relatively few clock cycles to execute the entire algorithm. As a result, the total time needed to execute the entire algorithm (based on the statistical information in exec.stats) should give a much more accurate measure of the design's performance.

3.9 System Control

What has been described up to this point are the various programs that are sequenced through to generate and evaluate a design. What has yet to be described is the method used to change the design so that it meets the area and performance criteria. The program that handles this is CaDSS. It uses as its input files the algorithm file filename.dat and a parameter file filename.prm. Its job is to sequence through the set of previously described programs changing the number of functional units on each pass through the sequence.

The first function of this program is to generate the file intermed.dat from the schedule file. (Intermed.dat is just filename.dat in a form that is easier for the programs to read.) The various programs are then sequenced through in the described order. Then the program adds an additional unit and generates a new design. The functional unit added is the smallest unit. The idea here is to see if the performance and area can be improved by adding an operator. It makes sense to add the smallest functional unit first because it adds the least amount of area to the data path but still has the potential of increasing the speed because it tends to parallelize the data path. (Remember that it is possible for the addition of a functional unit to decrease the area of a design because the resulting controller may be smaller.) If this results in an improvement then another of this operator is added and the process iterates. This is repeated until the design passes through several iterations without improvement. The number of iterations that the design has to pass through without improvement is specified by the parameter failure_iterations in the .prm file. Once this number of failing iterations is reached the next smallest functional unit is added and the process is repeated until all of the functional units have been tried. Each time the best design is kept. Once all of the functional units have been used the process restarts from the smallest unit again to see if adding more functional

units to the current design can improve it. This continues until one pass through all of the functional units is made without improving the design or until a design is reached that meets all the design specifications. The file stats.spec contains the area and timing information for the current design. A flow chart can be found in appendix A describing the operation of the program.

The question remains of how it is determined whether or not the present design is an improvement over the last design. In the past attempts where made to achieve an optimal design by minimizing a cost function such as:

$$area \times time^{2\alpha} \quad 0 \le \alpha \le 1.$$

Here, however, specific area and time specifications are given and the goal is to meet those specifications. As a result, a somewhat different approach is taken. Each time a design is completed CaDSS computes the discrepancy of the area and delay from the goals. The formulas used are:

$$pda = (area - spec\_area) / spec\_area$$
$$pdt = (delay - spec\_delay) / spec\_delay$$

where pda is the percent deviation of the area and pdt is the percent deviation of the time (delay). Whichever discrepancy is larger determines what must be minimized next. Thus if pda is largest then the area must be minimized even if delay increases somewhat. The formula used to determine the acceptable discrepancies is:

```
if pdt > pda
  npda = pda + (beta * ( (alpha * pdt) - pda))
  npdt = pdt
else
  npdt = pdt + (beta * (pda - (alpha * pdt)))
  npda = pda
```

where npda is the new percent discrepancy in area that will be accepted and npdt is the new percent discrepancy in time that will be accepted. Alpha is a factor expressing the relative importance of time over area and beta is a factor determining how much of the difference between pda and the adjusted pdt will be added to the next acceptable percent discrepancy. These parameters are taken from the .prm file. This allows one cost to actually increase if the other one is further from the goal. How much of an increase is allowed is a function of the difference between the adjusted pdt and pda. The greater this difference the larger the allowed increase.

Chapter 4
RESULTS AND CONCLUSIONS

4.1 Results

One of the major problems with high level synthesis systems is the question of how to verify that they are producing functionally correct designs, let alone near optimal designs. The whole purpose of synthesis systems is to produce a design in a matter of hours that is comparable in quality to a design that a design team would produce in weeks or months. This makes it very difficult to check the designs produced by a synthesis system. The only practical way to test a system is to have it produce an extremely small design and verify it. Even doing this, the verification process is very time consuming and the problem of determining the optimality of the design is still difficult at best.

Keeping this in mind, CaDSS was tested in two different ways. First of all, during the development of the system, each individual program was tested on several customized input files. This demonstrated the individual functionality of each of the programs. Secondly, the system as a whole was tested on several small examples. Based on the results of these tests it was determined that CaDSS apparently produces functionally correct designs. To determine how optimal the designs are, however, is very difficult. It would require accurate information about functional unit sizes and delays that are not available at this time. The other questions as to how the data path affects the controller and if, in fact, the addition of a functional unit to the data path can actually decrease the size of a design by decreasing the complexity of the controller are harder to answer. It is certainly a fact that adding a functional unit may decrease the complexity of the controller but as to whether or not the resulting decrease in the size of the controller will offset the increase in the size

of the data path is harder to determine. To do this CaDSS would have to be run on a design that is relatively large (so that the controller itself is of a significant size). This would make it extremely difficult to verify the functionality of the design. Also, again, this would require accurate information on the size and delays of the various functional units and the logic gates used in the controller. Appendix B contains all of the input, output, and intermediate files for a sample run of CaDSS, and includes a diagram of the data path that CaDSS generated. The algorithm itself does not perform any useful function and should be viewed only as a demonstration. The algorithm used for this sample run was designed so as to produce small output and intermediate files so that they could be shown here. It was also designed to show how CaDSS handles looping and conditional branches.

4.2 Limitations

Certain limitations were known about the system before its design was completed because they were the result of the constraints placed on the design itself. This includes the use of an intermediate form to describe the algorithm and the assumption that this form has been pre-optimized and will, thus, generate an optimized DFG. Other restrictions involve the data path. These include the restriction to the use of binary operations, the restriction against using buses, and the use of the same generic bit width for all operations. Though these are significant limitations, they were and are considered to be irrelevant to the purpose of CaDSS, namely to provide a prototype high level synthesis system that takes both the data path and the controller into account.

There were, however, a number of problems that showed up after the design was completed and tested that are quite relevant to the purpose of CaDSS. Two of these problems involve the algorithm that sequences through

the various programs. It is subject to two significant problems, the first one involving the quality of the results and the second involving the time needed for execution. The first problem involves getting stuck in local minima. It is possible that during the process of adding functional units, a design is reached that is minimal (in either time or area) compared to all of the designs around it but is not an overall minimum (optimum) design. In other words, unless a very large number of functional units are added or an exceptionally large increase in either area or delay is accepted there may be no way to reach the optimal design. There are two ways that CaDSS can overcome this problem. They are increasing the failure iterations parameter or increasing the beta parameter. However, the result of increasing either of these parameters, especially the failure iterations, is an increase in the number of design iterations and consequently an increase in the CPU time needed to find a good design. Furthermore, the ideal value of those parameters will likely change from one input algorithm and from one technology file to another. There is no way to determine exactly what those parameters should be.

The second problem involves the speed of the algorithm. The algorithm that controls the sequencing of the program is iterative in nature. It will likely iterate through the design process dozens of times for even an extremely small design and possibly hundreds of times for a large design. This can be very time consuming, especially if one or more of the programs it must sequence through requires a large amount of CPU time. Unfortunately, one of the programs is slow even on small designs. ESPRESSO must be run twice; once to generate combo1 and once to generate combo2. Combo2 is generally much larger than combo1 and requires more CPU time. For the test algorithms used so far ESPRESSO accounts for well over 50 percent of the CPU time required for each pass. That means that, at least for small designs, ESPRESSO is extremely

costly in terms of CPU time. What is not known, however, is how the time required for ESPRESSO to minimize a two level design increases with the number of functions it must minimize. It is possible that after a point the addition of more functional units does not cause a significant increase in the run time of ESPRESSO.

Unfortunately, there are other problems associated with the use of ESPRESSO. ESPRESSO is a PLA minimization program and, as a result, it can only be used to minimize logic functions for a two level implementation. In general, however, a two level implementation is neither the smallest implementation of a combinational logic block nor, due largely to fanout induced delays, the fastest implementation. As a result, the controller that is designed by CaDSS is itself not optimal in terms of area or speed.

There are, of course, numerous other limitations on CaDSS' performance. These limitations are, however, more tradeoffs between the ease of the design of CaDSS and its ability to produce good designs than they are flaws in its methods of operation. They are decisions such as the use of a modified ASAP scheduling algorithm rather than a true list scheduling algorithm. Also, the algorithm used to perform data path allocation, though relatively simple, certainly does not obtain optimal results. Ideally, scheduling, allocation, and module binding should be performed simultaneously for the best results. Exploring all three simultaneously without using an absolutely unreasonable amount of CPU time is, however, difficult at best. As can be seen, these short comings do not reflect flaws in the general approach of CaDSS.

4.3 Future Improvements

There would appear to be two critical problems with the approach to high level synthesis taken by the CaDSS system. They are the use of ESPRESSO and the iterative nature of the algorithm that sequences through the various programs. ESPRESSO presents a problem with its ability to effectively minimize the combinational logic both in terms of area and speed as well as potentially presenting problems with the amount of CPU time that it requires. The iterative nature of the main controlling algorithm of CaDSS presents a problem due to the inherently large amount of CPU time it requires.

The inability of ESPRESSO to minimize delays through combinational logic and its limitation to two level logic can easily be corrected. There have been attempts to develop systems that minimize multilevel logic blocks to meet both timing and area constraints. These include Socrates [16] and more recently MIS. [14] Using these systems would, however, make the problems of the CaDSS' run time even worse. It would appear that ESPRESSO already causes problems with the amount of CPU time it requires. MIS and Socrates will almost certainly require even more CPU time because of the inherently more complex nature of the problem that they attempt to solve.

There are other methods that might be employed to reduce the CPU time the system requires to produce a design. One obvious solution is to modify the system so that it can produce a design in one pass through the design process. It is probably possible to produce a good data path design in one pass through the design process but to be able to optimize both the data path and the controller in that way is nearly impossible at this time. As explained earlier, the design of the controller relies completely on the data path design and, as a result, the controller cannot be completely specified until the data path design is completed. Further, it is not known exactly how the data path design affects the

controller. These two facts suggest that until more is known about how the data path affects the controller (if in fact that can be generalized at all) that an iterative design process is necessary if the controller is to be taken into account.

That suggests another method of improving the system. Since it would seem that the minimization of the combinational logic for the controller uses a significant percentage of the CPU time required by the system that some method could be used to minimize that time. There are two possible approaches to this problem. The obvious one would be to develop faster logic minimization algorithms. These are problems that have been worked on for some time and are currently being worked on by various groups. The second approach would be not to generate the combinational logic during each pass but rather to simply estimate its size. When a design is chosen the combinational logic could then be generated that one time. Again, the problem here is to develop the algorithms to do this.

4.4 Summary

This thesis presented a prototype high level synthesis system (CaDSS). The purpose of such a system is to take a behavioral (algorithmic) description of a chip and produce a register transfer level description of the chip. What sets this system apart from other high level synthesis systems that have been developed so far is that it attempts to take into account the effects that the data path has on the controller. That is, it attempts to find what effects an increase in the size of the data path has on the size and speed of the controller. It also defines the speed of the chip in terms of the total execution time of the algorithm rather than in terms of the clock cycle length.

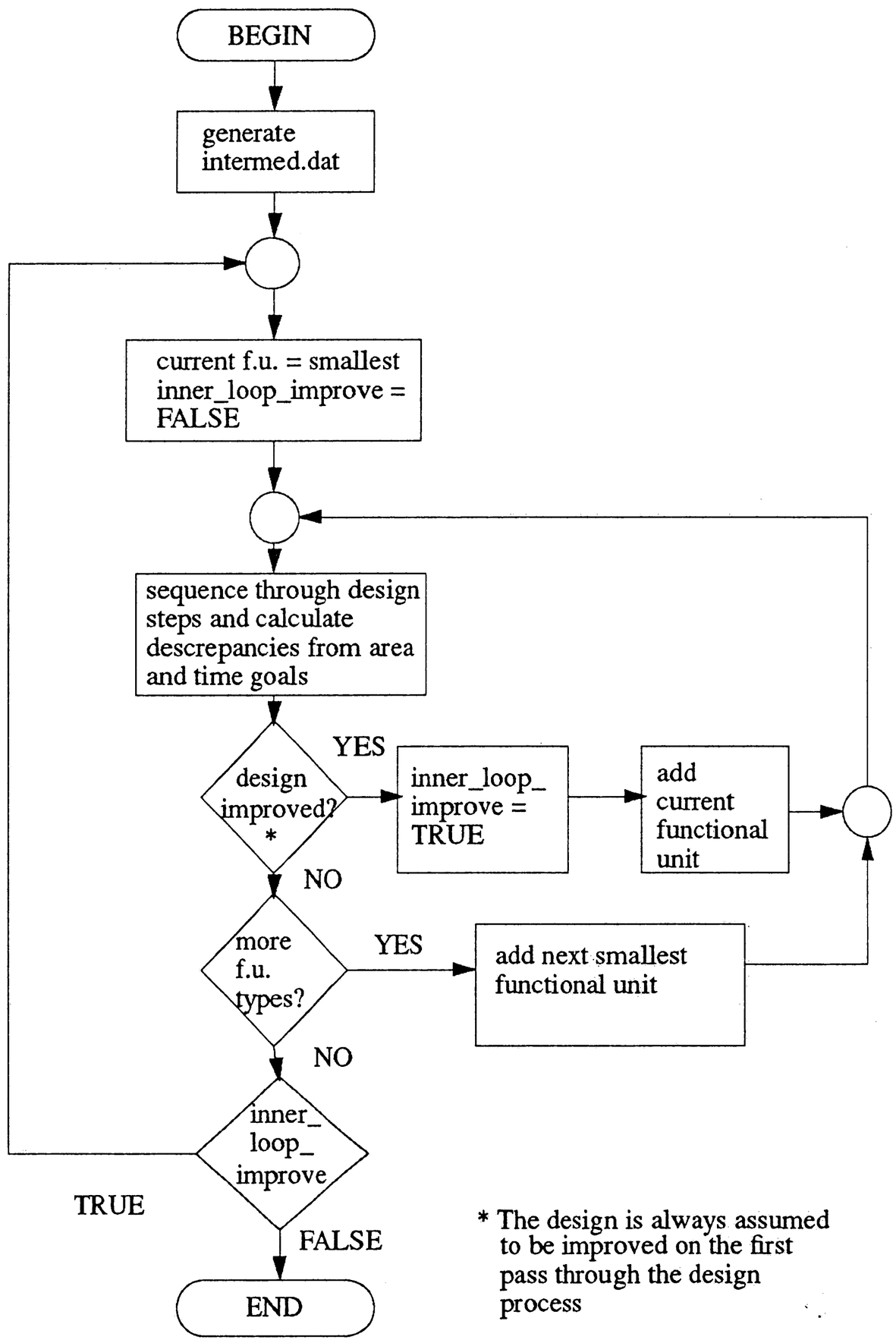
CaDSS consists of a series of programs written in the C programming language. (These programs are kept on file in Lehigh University's Computer

Science and Electrical Engineering Department.) The programs perform a series of tasks including scheduling of operations, determining the lifetimes of variables, register allocation, data path allocation and module binding, and control generation. Several iterations are made through the design process. Each iteration adds an additional functional unit to the data path. The process is completed when either the area and timing requirements are met or no improvements are made to the design by the addition of a given number of functional units.

The CaDSS system was found to produce functionally correct designs. The approach taken appears to be valid but has some inherent difficulties most of which involve the amount of time the system requires to run to completion. The goal of minimizing control along with the data path would benefit high level synthesis systems by producing designs that are both smaller and faster and, as a result, more research in this area would be beneficial.

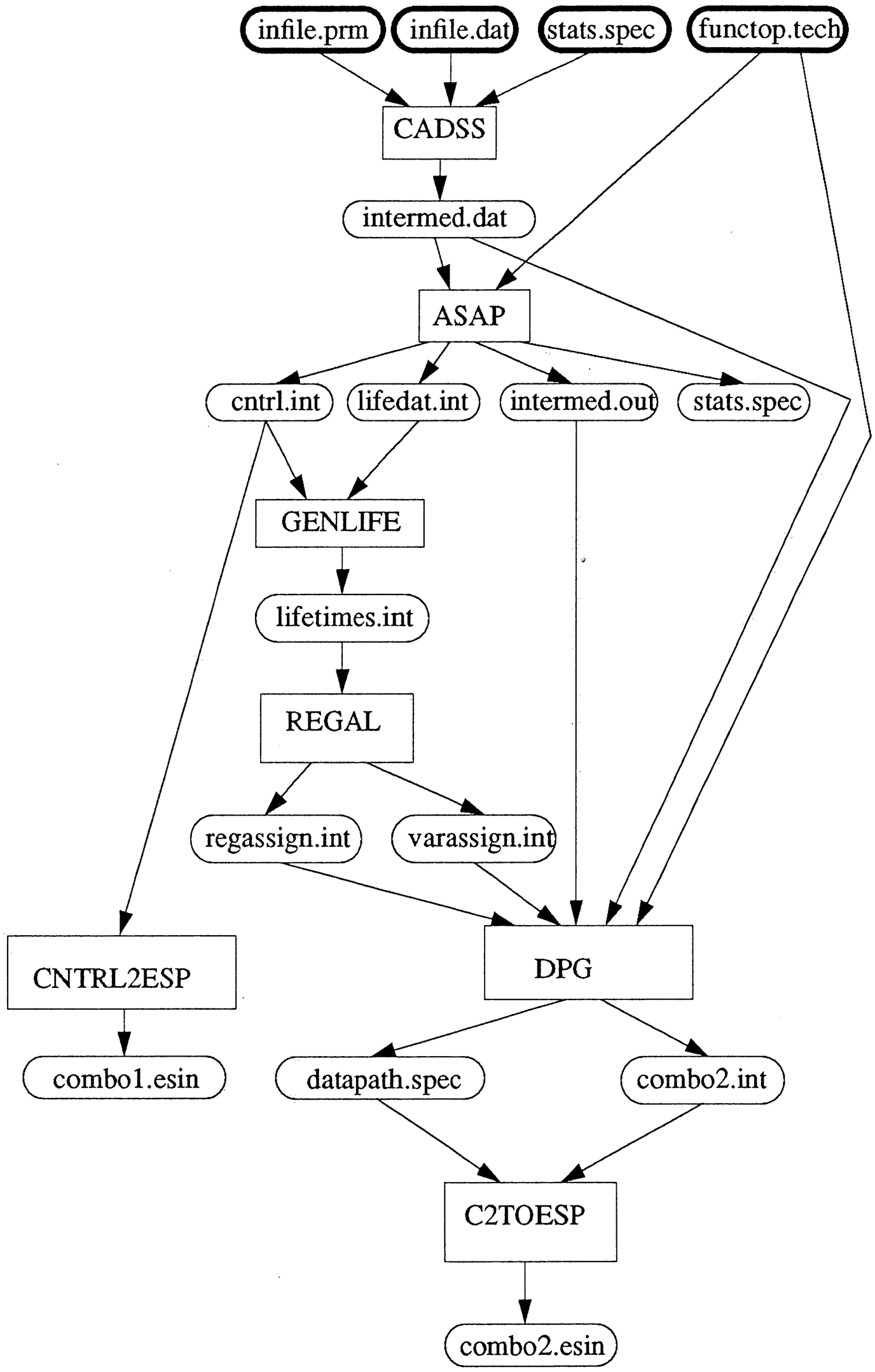
Appendix A
SYSTEM FLOW DIAGRAMS

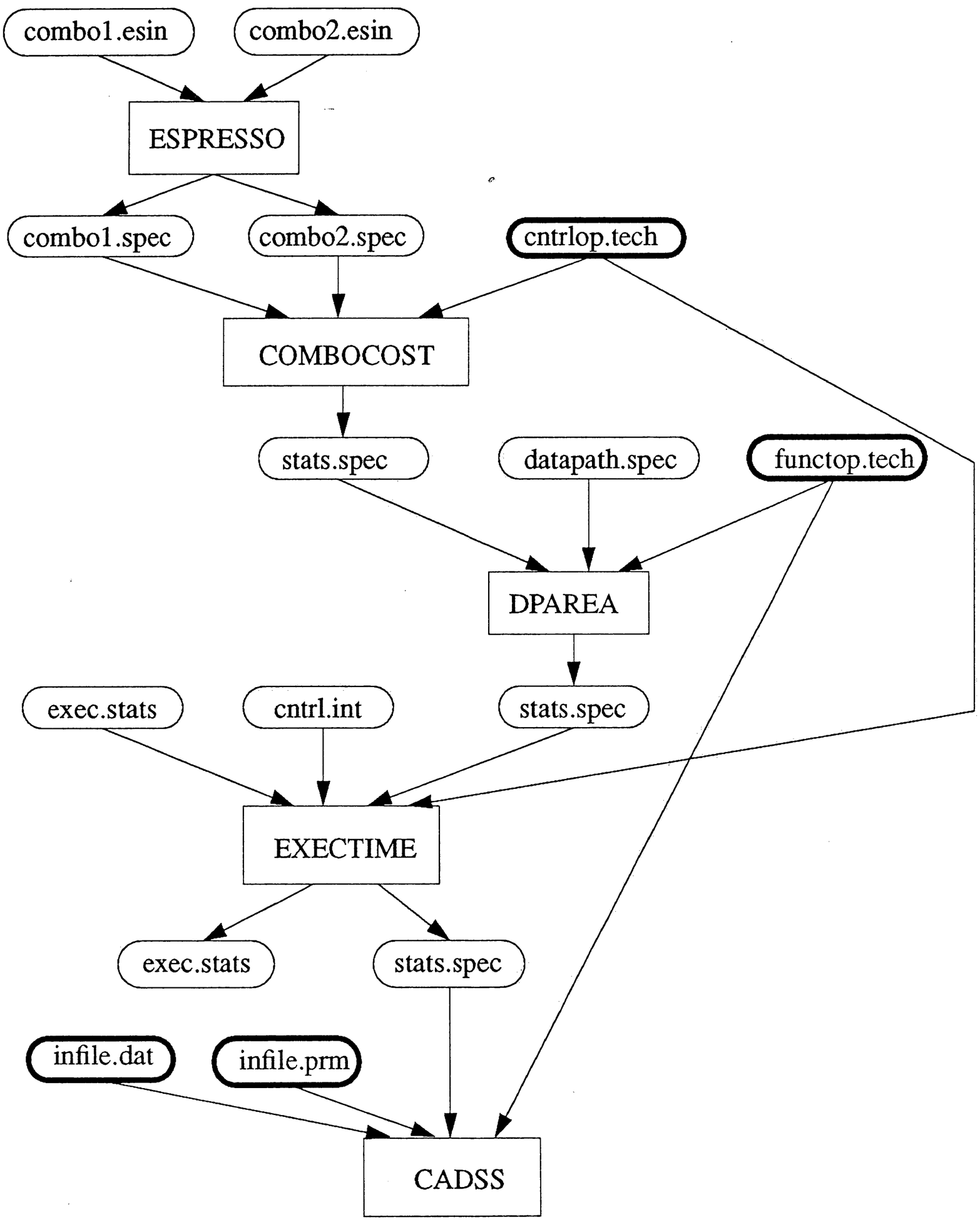
A.1 CaDSS SYSTEM FLOW CHART



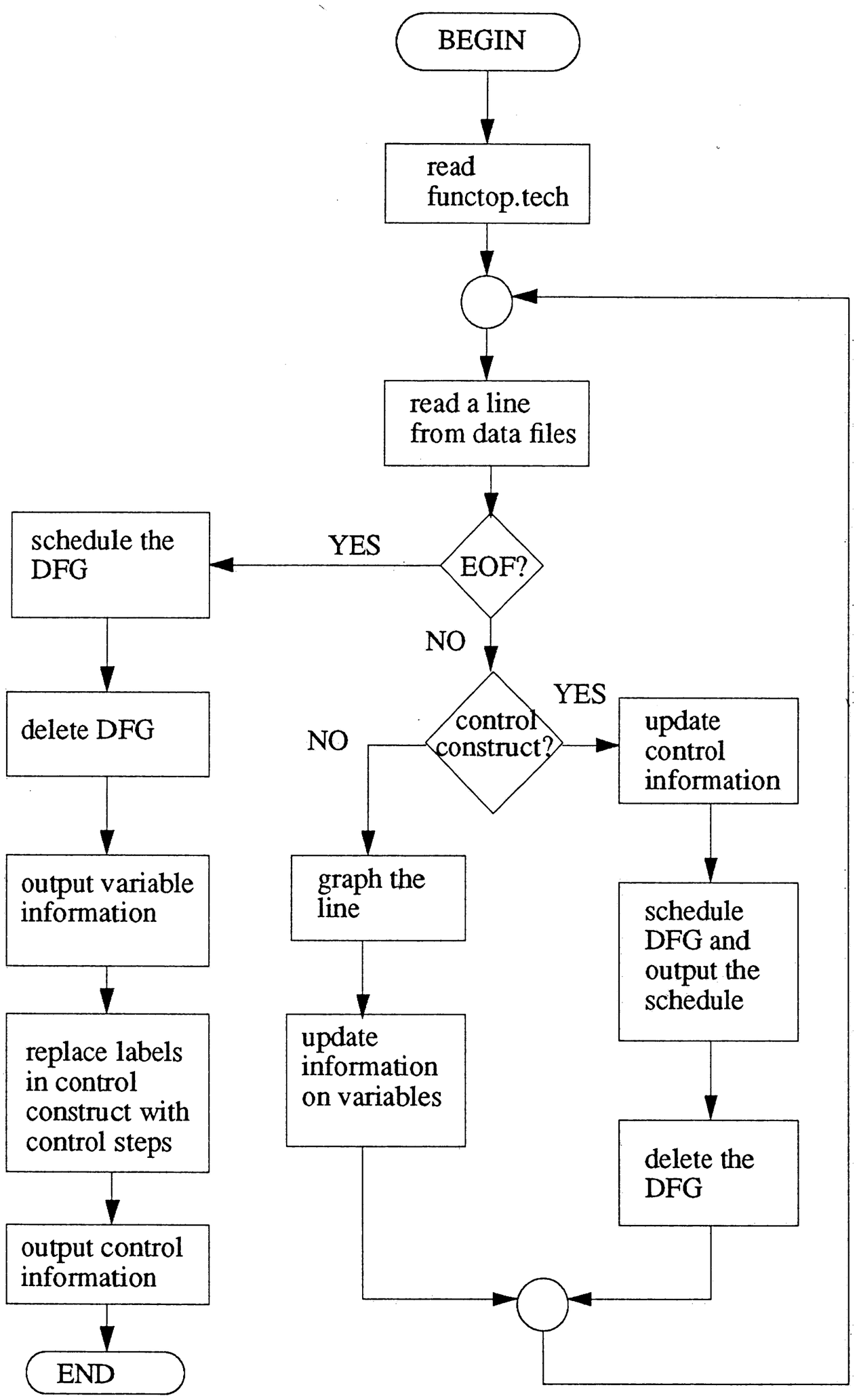
A.2 SYSTEM FILE STRUCTURE

The following two pages are a representation of the file structure used by CaDSS. Ellipses represent input and output files. The heavy lined ellipses are user generated input files. The lighter lined ellipses are generated by CaDSS. The rectangles represent the various programs within the system.

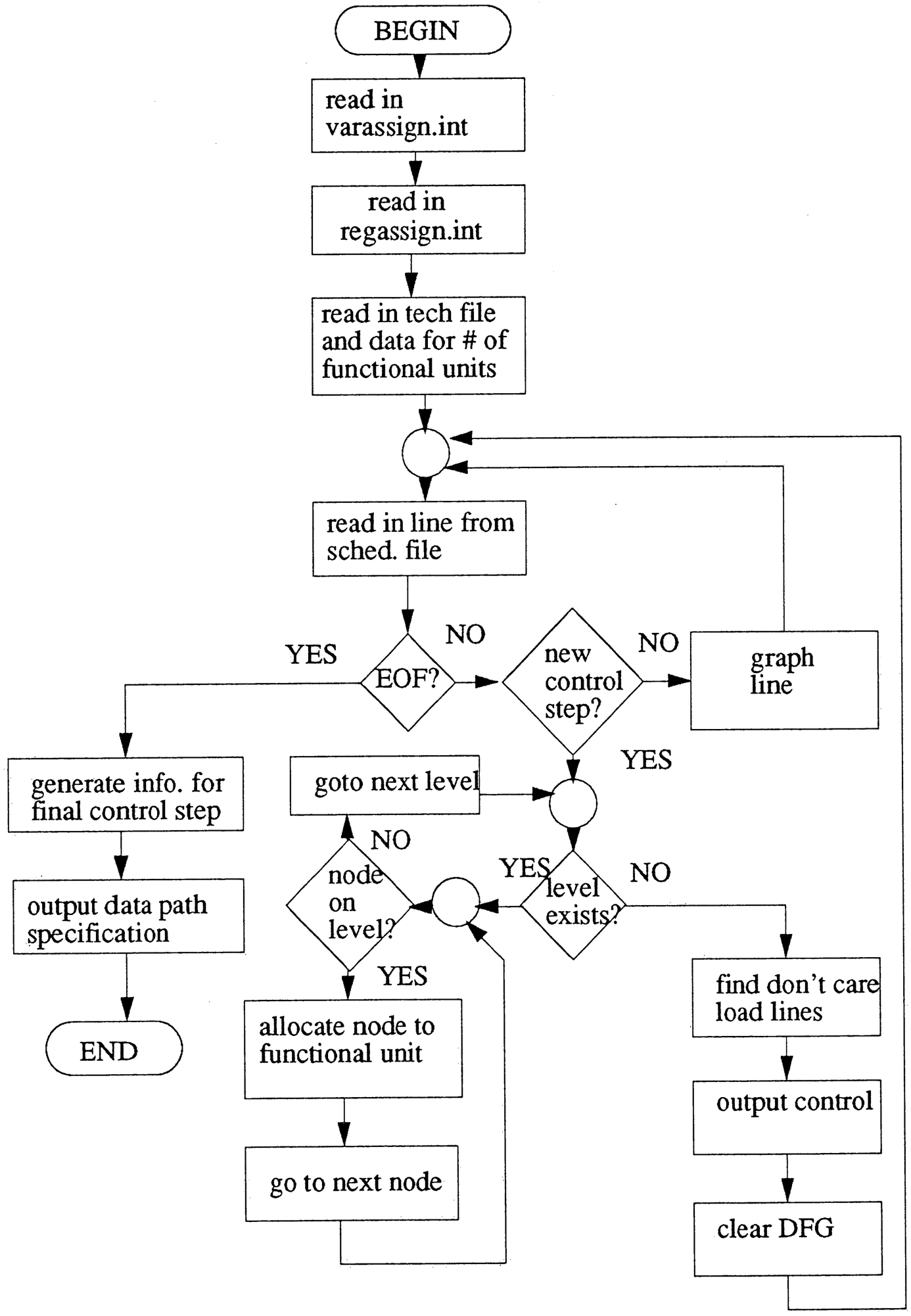

infile.prm
infile.dat
stats.spec
functop.tech
CADSS
intermed.dat
ASAP
cntrl.int
lifedat.int
intermed.out
stats.spec
GENLIFE
lifetimes.int
REGAL
regassign.int
varassign.int
CNTRL2ESP
DPG
combo1.esin
datapath.spec
combo2.int
C2TOESP
combo2.esin


combo1.esin
combo2.esin
ESPRESSO
combo1.spec
combo2.spec
cntrlop.tech
COMBOCOST
stats.spec
datapath.spec
functop.tech
DPAREA
exec.stats
cntrl.int
stats.spec
EXECTIME
exec.stats
stats.spec
infile.dat
infile.prm
CADSS

A.3 FLOW CHART FOR ASAP (SCHEDULER)



A.4 FLOW CHART FOR DPG (DATA PATH GENERATOR)



Appendix B
PROGRAM INPUT AND OUTPUT FOR A SAMPLE RUN

B.1 TESTSCHED11.DAT

This is the primary input file for a sample run of CaDSS. It was designed primarily for demonstration purposes and does not implement a useful algorithm. It was chosen for the small size of the resulting intermediate and output files. Furthermore, it demonstrates the way in which CaDSS implements looping structures and conditional branches.

format:

```
#clk_per num          ;this is a number representing
                      ;a pseudo clock cycle length
#op num               ;optional specification of the
                      ;initial number of each type of
                      ;functional unit
```

The rest of the file contains the algorithm. For this example, the algorithm is:

```
#clk_per 40.0
v1 = ia * ib
v2 = ic + id
v3 = id + ie
v1 = ih - v1
loop1:
  v4 = v1 * v3
  v5 = v4 and v2
  if v5 > v4 then {
    v6 = v5 - v4
    v7 = v6 * v4
    v8 = v7 - v2
  }
  else {
    v6 = v5 + v4
    v7 = v5 - v2
    v8 = v7 * v6
  }
```

```
    v9 = v8 - v6
    v10 = v9 + v7
    v11 = v10 * ib
    v12 = v10 - v2
if v12 > v10 goto loop1
v13 = v12 + v11
v14 = v13 + v10
v15 = v14 - v6
v16 = v13 * v14
```

B.2 TESTSCHED11.PRM

This is an input file that specifies the various control parameters used by CaDSS.

```
alpha  1.2
beta 0.8
max_area 800000.5
max_delay 2900.5
failure_iterations 2
```

B.3 FUNCTOP.TECH

This is the functional unit technology file. It lists all of the functional units that CaDSS has available to use in its design process. Each unit is described by a symbol, its delay, whether or not it is a commutative operator, and by its area.

format:

```
#funct_units num   ;number of types of f.u.'s
reg delay area     ;delay and area for a register
fu delay c area    ;functional unit, its delay, its
                   ;area, and c for commutative or n
                   ;if not
```

```
#funct_units 5
reg 5.0 42250.0
+     5.0 c 63375.0
-     5.0 n 71825.0
and   1.0 c 16900.0
*     20.0 c 226562.5
>     5.0 n 60153.2
```

B.4 CNTRLOP.TECH

This is the control unit technology file. The control unit is restricted to the use of a handful of gates. These are input and output buffers, registers, inverters, and nand and nor gates of various sizes. Each gate is described by its number of inputs, its area, its delay for a fanout of 1, and a factor used in calculating the delay for larger fanouts.

format:

```
op size base_delay delay_factor area
```

```
inbuff 1 2.98 0.6678 1024.3
inrb 1 2.01 0.5678 528.125
nd 2 2.29 0.7933 792.188
nd 3 2.75 1.1378 1056.25
nd 4 3.02 1.5267 1320.3125
nr 2 2.04 1.31 792.188
nr 3 2.53 2.0356 1056.25
nr 4 3.73 2.5889 1320.313
outbuff 1 4.00 3.01 2500.221
creg 5.0 5545.3125
```

B.5 EXEC.STATS

This file lists each loop or *if* statement. The letter l specifies a loop. The first number is the loop number counting from the top and the second number is the average number of times that loop is executed. The letter i specifies an *if* or an *if then else* statement. The first number is the number of the *if* statement counting from the top of the file and the second number is the percentage of the time that the true branch of the *if* is executed.

```
l 0 8
i 0 0.25
```

B.6 INTERMED.DAT

This file has the same general format as the input data file. The only difference is that the various parameters are a set number of spaces apart. This simply makes it easier for CaDSS to change their values.

```
#and      1
#>      1
#+      2
#-      1
#*      1
#clk_per 40.000000
v1 = ia * ib
v2 = ic + id
v3 = id + ie
v1 = ih - v1
loop1:
  v4 = v1 * v3
  v5 = v4 and v2
  if v5 > v4 then {
    v6 = v5 - v4
    v7 = v6 * v4
    v8 = v7 - v2
  }
  else {
    v6 = v5 + v4
    v7 = v5 - v2
    v8 = v7 * v6
  }
  v9 = v8 - v6
  v10 = v9 + v7
  v11 = v10 * ib
  v12 = v10 - v2
if v12 > v10 goto loop1
v13 = v12 + v11
v14 = v13 + v10
v15 = v14 - v6
v16 = v13 * v14
```

B.7 INTERMED.OUT

This is an intermediate file generated by asap that specifies which operations are to be performed in each control step.

Each line of this file has the format:

var3 = var2 op var1

and #cc n is specifies the control step that the operations that follow have been scheduled into.

```
#cc 1
v1 = ia * ib
v3 = id + ie
v2 = ic + id
v1 = ih - v1
#cc 2
v4 = v1 * v3
v5 = v4 and v2
cntrl_cond = v5 > v4
#cc 3
v6 = v5 - v4
v7 = v6 * v4
#cc 4
v8 = v7 - v2
#cc 5
v7 = v5 - v2
v6 = v5 + v4
v8 = v7 * v6
#cc 6
v9 = v8 - v6
v10 = v9 + v7
v11 = v10 * ib
#cc 7
v12 = v10 - v2
cntrl_cond = v12 > v10
#cc 8
v13 = v12 + v11
v14 = v13 + v10
v16 = v13 * v14
v15 = v14 - v6
```

B.8 CNTRL.INT

This file is generated by asap. It contains information about the control structures within the algorithm and the control steps in which they occur. The first line of the file gives that total number of control steps. The subsequent lines have one of the following 4 formats:

c ift var1 cond var2 fc

c ifte var1 cond var2 fc end

c iftg var1 cond var2 g

c goto g

These four formats correspond respectively to the following four statements: *if then, if then else, if then goto*, and *goto*. The letter c corresponds to the control step in which the control construct is first encountered. fc is the first control step executed if the test fails. end is the next state following an if then else construct. g is the state that control is passed to by an *if then goto* or a *goto* control construct.

```
#numstates 8
2 ifte v5 > v4 5 6
7 iftg v12 > v10 2
```

B.9 LIFEDAT.INT

This file is generated by asap and contains information used to find the lifetimes of all of the variables. The first line gives the number of states. Each subsequent line starts with a variable name followed by a sequence of number pairs. The first number is either a 0 or a 1 and the second number represents a control step. If the first number is a 1 then the variable was defined in the specified control step. If it is a 0 then the variable was simply accessed in the specified control step. Only the last access of a variable before a definition is listed. Also external inputs are always followed by the pair: 0 0.

```
#states 8
ia 0 0
ib 0 0
v1 1 1 0 1 1 1 0 2
ic 0 0
id 0 0
v2 1 1 0 7
ie 0 0
v3 1 1 0 2
ih 0 0
v4 1 2 0 5
v5 1 2 0 5
cntrl_cond 1 2 1 7
v6 1 3 0 3 1 5 0 8
v7 1 3 0 4 1 5 0 6
v8 1 4 1 5 0 6
v9 1 6 0 6
v10 1 6 0 8
v11 1 6 0 8
v12 1 7 0 8
v13 1 8 0 8
v14 1 8 0 8
v15 1 8
v16 1 8
```

B.10 LIFETIMES.INT

The file is generated by genlife. It contains a listing of all of the variables and their lifetimes. The first line specifies the number of control steps. The subsequent lines start with a variable name which is followed by a series of number pairs. Each pair specifies the control steps in which the variable is born and in which it dies.

```
#states 8
ia 0 0
ib 0 0
v1 1 7
ic 0 0
id 0 0
v2 1 7
ie 0 0
v3 1 7
ih 0 0
v4 2 5
v5 2 5
cntrl_cond 2 2 7 7
v6 3 8
v7 3 6
v8 4 6
v9 6 6
v10 6 8
v11 6 8
v12 7 8
v13 8 8
v14 8 8
v15 8 8
v16 8 8
```

B.11 REGASSIGN.INT

This file is generated by genlife. It lists each register. For each register information regarding the variables it holds and their lifetimes are given. The first two lines specify the number of registers and the number of control steps. The subsequent lines first specify a register. Then for that register each variable that is assigned to it along with the control steps for which it is assigned are listed.

```
#regs 8
#states 8
reg 0
v1 1 7
reg 1
v2 1 7
reg 2
v3 1 7
reg 3
v4 2 5
v10 6 8
reg 4
v5 2 5
v11 6 8
reg 5
v6 3 8
reg 6
v7 3 6
v12 7 8
reg 7
v8 4 6
```

B.12 VARASSIGN.INT

This file lists each variable the register it is assigned to and the control steps for which it is assigned to that register. The first two lines again specify the number of registers and the number of control steps. Each of the subsequent lines starts with a variable name and is followed by a trio of numbers. The first number specifies the register that the variable is assigned to and the remaining two numbers specify the lifetime for which it is assigned to that register. If the first number is a -1, it was not necessary to assign that variable to a register.

```
#regs 8
#states 8
ia -1 1 8
ib -1 1 8
v1 0 1 7
ic -1 1 8
id -1 1 8
v2 1 1 7
ie -1 1 8
v3 2 1 7
ih -1 1 8
v4 3 2 5
v5 4 2 5
cntrl_cond -1 1 8
v6 5 3 8
v7 6 3 6
v8 7 4 6
v9 -1 0 0
v10 3 6 8
v11 4 6 8
v12 6 7 8
v13 -1 0 0
v14 -1 0 0
v15 -1 0 0
v16 -1 0 0
```

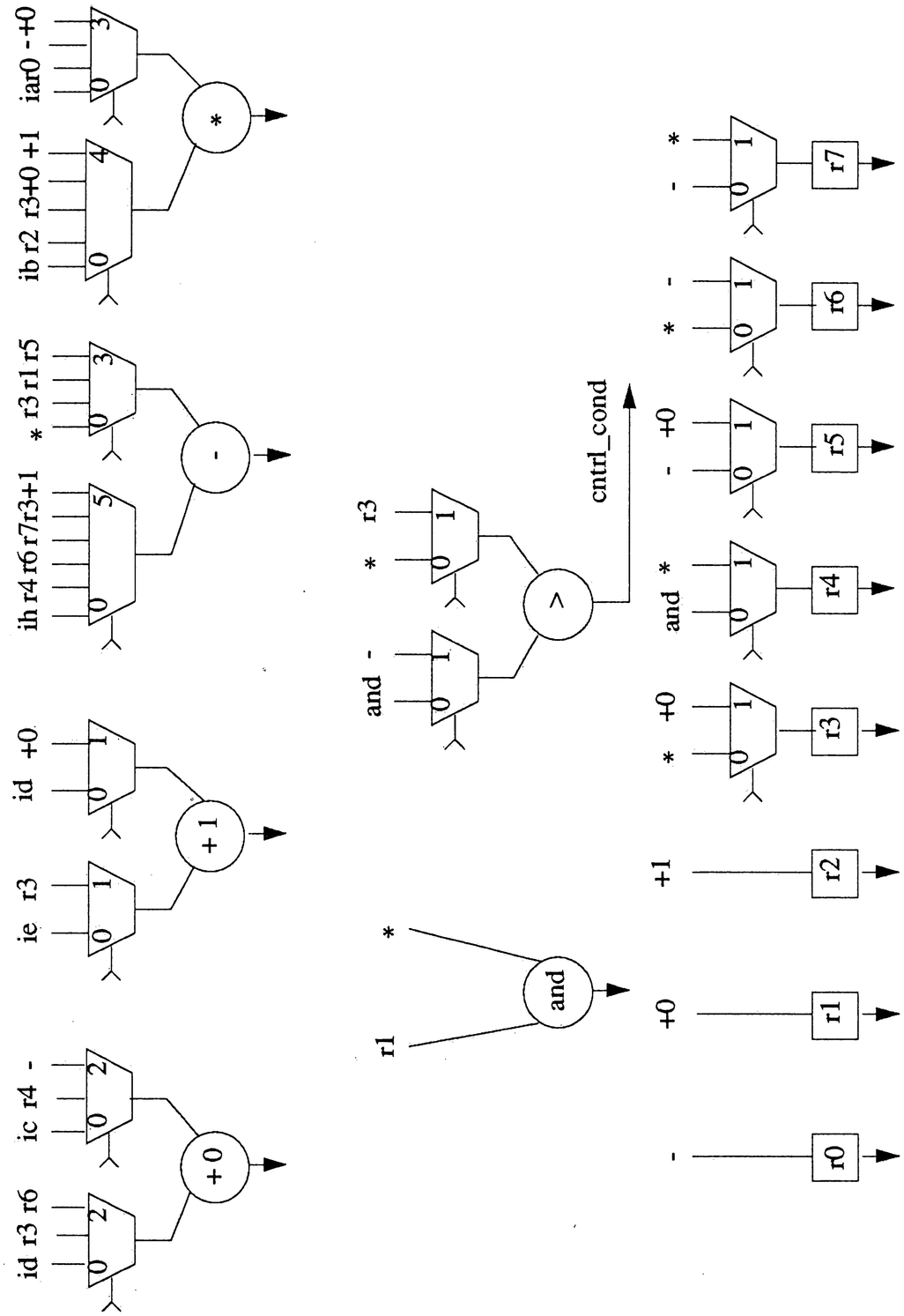
B.13 DATAPATH.SPEC

This file is generated by the program dpg. It describes the data path generated by CaDSS. The first line specifies the total number of functional units. For the subsequent lines, a '#' starts the description of a functional unit type by giving the number of functional units of that type. For registers that is followed by the number of each register along with the number of inputs that is has. Then the inputs to the mux that go into each register are specified starting with the input that is selected by a 0 at the mux select line. For other functional units, the number of left and right inputs are specified and then listed on the subsequent lines.

```
#num_func_units:  5
#reg 8
reg 0 num_inputs:  1
inputs:  - 0
reg 1 num_inputs:  1
inputs:  + 0
reg 2 num_inputs:  1
inputs:  + 1
reg 3 num_inputs:  2
inputs:  * 0 + 0
reg 4 num_inputs:  2
inputs:  and 0 * 0
reg 5 num_inputs:  2
inputs:  - 0 + 0
reg 6 num_inputs:  2
inputs:  * 0 - 0
reg 7 num_inputs:  2
inputs:  - 0 * 0
#cntrl_cond num_inputs: 1
inputs:  > 0
#+ 2
+ 0 num_l_inputs: 3 num_r_inputs: 3
left_inputs:  id reg 3 reg 6
right_inputs:  ic reg 4 - 0
+ 1 num_l_inputs: 2 num_r_inputs: 2
left_inputs:  ie reg 3
right_inputs:  id + 0
#- 1
- 0 num_l_inputs: 6 num_r_inputs: 4
left_inputs:  ih reg 4 reg 6 reg 7 reg 3 + 1
right_inputs:  * 0 reg 3 reg 1 reg 5
```

```
#and 1
and 0 num_l_inputs: 1 num_r_inputs: 1
left_inputs:  reg 1
right_inputs:  * 0
#* 1
* 0 num_l_inputs: 5 num_r_inputs: 4
left_inputs:  ib reg 2 reg 3 + 0 + 1
right_inputs:  ia reg 0 - 0 + 0
#> 1
> 0 num_l_inputs: 2 num_r_inputs: 2
left_inputs:  and 0 - 0
right_inputs:  * 0 reg 3
```

B.14 RTL DIAGRAM OF THE DATA PATH DESCRIBED IN DATAPATH.SPEC



B.15 COMBO2.INT

This file is generated by dpg. It specifies the control line settings for the multiplexors and registers in the data path. The first line specifies the number of control steps. Each #cc line specifies the control step that is to be described. Each line describes the configuration of one functional unit or register. For example, the line:

+ 0 l_mux_sel: 0 r_mux_sel: 1

says that adder number 0 should have a 0 at its left mux select line and a 1 at its right mux select line. Registers only have one mux but they use load to determine whether they should be loaded (1) or not (0) during the current control step. Note that a -1 specifies a don't care condition.

```
#num_states: 8
#cc 1
reg 0 load:  1 mux_sel:  0
reg 1 load:  1 mux_sel:  0
reg 2 load:  1 mux_sel:  0
reg 3 load:  -1 mux_sel:  -1
reg 4 load:  -1 mux_sel:  -1
reg 5 load:  -1 mux_sel:  -1
reg 6 load:  -1 mux_sel:  -1
reg 7 load:  -1 mux_sel:  -1
cntrl_cond mux_select -1
+ 0 l_mux_sel:  0  r_mux_sel:  0
+ 1 l_mux_sel:  0  r_mux_sel:  0
- 0 l_mux_sel:  0  r_mux_sel:  0
and 0 l_mux_sel:  -1  r_mux_sel:  -1
* 0 l_mux_sel:  0  r_mux_sel:  0
> 0 l_mux_sel:  -1  r_mux_sel:  -1
#cc 2
reg 0 load:  0 mux_sel:  -1
reg 1 load:  0 mux_sel:  -1
reg 2 load:  0 mux_sel:  -1
reg 3 load:  1 mux_sel:  0
reg 4 load:  1 mux_sel:  0
reg 5 load:  -1 mux_sel:  -1
reg 6 load:  -1 mux_sel:  -1
reg 7 load:  -1 mux_sel:  -1
```

```
cntrl_cond mux_select 0
+ 0 l_mux_sel:  -1  r_mux_sel:  -1
+ 1 l_mux_sel:  -1  r_mux_sel:  -1
- 0 l_mux_sel:  -1  r_mux_sel:  -1
and 0 l_mux_sel:  0  r_mux_sel:  0
* 0 l_mux_sel:  1  r_mux_sel:  1
> 0 l_mux_sel:  0  r_mux_sel:  0
#cc 3
reg 0 load:  0 mux_sel:  -1
reg 1 load:  0 mux_sel:  -1
reg 2 load:  0 mux_sel:  -1
reg 3 load:  0 mux_sel:  -1
reg 4 load:  0 mux_sel:  -1
reg 5 load:  1 mux_sel:  0
reg 6 load:  1 mux_sel:  0
reg 7 load:  -1 mux_sel:  -1
cntrl_cond mux_select -1
+ 0 l_mux_sel:  -1  r_mux_sel:  -1
+ 1 l_mux_sel:  -1  r_mux_sel:  -1
- 0 l_mux_sel:  1  r_mux_sel:  1
and 0 l_mux_sel:  -1  r_mux_sel:  -1
* 0 l_mux_sel:  2  r_mux_sel:  2
> 0 l_mux_sel:  -1  r_mux_sel:  -1
#cc 4
reg 0 load:  0 mux_sel:  -1
reg 1 load:  0 mux_sel:  -1
reg 2 load:  0 mux_sel:  -1
reg 3 load:  0 mux_sel:  -1
reg 4 load:  0 mux_sel:  -1
reg 5 load:  0 mux_sel:  -1
reg 6 load:  0 mux_sel:  -1
reg 7 load:  1 mux_sel:  0
cntrl_cond mux_select -1
+ 0 l_mux_sel:  -1  r_mux_sel:  -1
+ 1 l_mux_sel:  -1  r_mux_sel:  -1
- 0 l_mux_sel:  2  r_mux_sel:  2
and 0 l_mux_sel:  -1  r_mux_sel:  -1
* 0 l_mux_sel:  -1  r_mux_sel:  -1
> 0 l_mux_sel:  -1  r_mux_sel:  -1
#cc 5
reg 0 load:  0 mux_sel:  -1
reg 1 load:  0 mux_sel:  -1
reg 2 load:  0 mux_sel:  -1
reg 3 load:  0 mux_sel:  -1
reg 4 load:  0 mux_sel:  -1
reg 5 load:  1 mux_sel:  1
reg 6 load:  1 mux_sel:  1
reg 7 load:  1 mux_sel:  1
cntrl_cond mux_select -1
```

```
+ 0 l_mux_sel:  1  r_mux_sel:  1
+ 1 l_mux_sel:  -1  r_mux_sel:  -1
- 0 l_mux_sel:  1  r_mux_sel:  2
and 0 l_mux_sel:  -1  r_mux_sel:  -1
* 0 l_mux_sel:  3  r_mux_sel:  2
> 0 l_mux_sel:  -1  r_mux_sel:  -1
#cc 6
reg 0 load:  0 mux_sel:  -1
reg 1 load:  0 mux_sel:  -1
reg 2 load:  0 mux_sel:  -1
reg 3 load:  1 mux_sel:  1
reg 4 load:  1 mux_sel:  1
reg 5 load:  0 mux_sel:  -1
reg 6 load:  0 mux_sel:  -1
reg 7 load:  0 mux_sel:  -1
cntrl_cond mux_select -1
+ 0 l_mux_sel:  2  r_mux_sel:  2
+ 1 l_mux_sel:  -1  r_mux_sel:  -1
- 0 l_mux_sel:  3  r_mux_sel:  3
and 0 l_mux_sel:  -1  r_mux_sel:  -1
* 0 l_mux_sel:  0  r_mux_sel:  3
> 0 l_mux_sel:  -1  r_mux_sel:  -1
#cc 7
reg 0 load:  0 mux_sel:  -1
reg 1 load:  0 mux_sel:  -1
reg 2 load:  0 mux_sel:  -1
reg 3 load:  0 mux_sel:  -1
reg 4 load:  0 mux_sel:  -1
reg 5 load:  0 mux_sel:  -1
reg 6 load:  1 mux_sel:  1
reg 7 load:  -1 mux_sel:  -1
cntrl_cond mux_select 0
+ 0 l_mux_sel:  -1  r_mux_sel:  -1
+ 1 l_mux_sel:  -1  r_mux_sel:  -1
- 0 l_mux_sel:  4  r_mux_sel:  2
and 0 l_mux_sel:  -1  r_mux_sel:  -1
* 0 l_mux_sel:  -1  r_mux_sel:  -1
> 0 l_mux_sel:  1  r_mux_sel:  1
#cc 8
reg 0 load:  -1 mux_sel:  -1
reg 1 load:  -1 mux_sel:  -1
reg 2 load:  -1 mux_sel:  -1
reg 3 load:  0 mux_sel:  -1
reg 4 load:  0 mux_sel:  -1
reg 5 load:  0 mux_sel:  -1
reg 6 load:  0 mux_sel:  -1
reg 7 load:  -1 mux_sel:  -1
cntrl_cond mux_select -1
+ 0 l_mux_sel:  2  r_mux_sel:  1
```

```
+ 1 l_mux_sel:   1   r_mux_sel:   1
- 0 l_mux_sel:   5   r_mux_sel:   3
and 0 l_mux_sel:   -1   r_mux_sel:   -1
* 0 l_mux_sel:   4   r_mux_sel:   3
> 0 l_mux_sel:   -1   r_mux_sel:   -1
```

B.16 COMBO1.ESIN

This is a description of combinational unit number 1 (combo 1) that is in an ESPRESSO readable input format.

```
#combinational circuit number 1
.i 4
.o 3
.ilb ps_2 ps_1 ps_0 cntrl_cond
.ob ns_2 ns_1 ns_0
.type fr
0000 001
0001 001
0010 100
0011 010
0100 011
0101 011
0110 101
0111 101
1000 101
1001 101
1010 110
1011 110
1100 111
1101 001
1110 111
1111 111
.e
```

B.17 COMBO2.ESIN

This is a description of combinational circuit number 2 (combo 2) that is in an ESPRESSO readable input format.

```
#combination circuit 2
.i 3
.o 31
.type fr
.ilb s2 s1 s0
.ob load_reg0 load_reg1 load_reg2 reg3_0 load_reg3
  reg4_0 load_reg4 reg5_0 load_reg5 reg6_0 load_reg6
  reg7_0 load_reg7 +0_l_1 +0_l_0 +0_r_1 +0_r_0 +1_l_0
  +1_r_0 -0_l_2 -0_l_1 -0_l_0 -0_r_1 -0_r_0 *0_l_2
  *0_l_1 *0_l_0 *0_r_1 *0_r_0 >0_l_0 >0_r_0
.type fr
000 111----------000000000000000--
001 0000101-----------------0010100
010 000-0-00101--------0010101010--
011 000-0-0-0-001------01010-------
100 000-0-01111110101--0011001110--
101 0001111-0-0-01010--0111100011--
110 000-0-0-011--------10010-----11
111 ----0-0-0-0--100111101111001l--
.e
```

B.18 COMBO1.SPEC

This is the ESPRESSO output that describes the minimized combo 1.

```
#combinational circuit number 1
.i 4
.o 3
.ilb ps_2 ps_1 ps_0 cntrl_cond
.ob ns_2 ns_1 ns_0
.p 8
010- 010
-011 010
11-0 110
10-- 100
--10 100
1-1- 010
--0- 001
-11- 101
.e
```

B.19 COMBO2.SPEC

This is the ESPRESSO output the describes the minimized combo2.

```
#combination circuit 2
.i 3
.o 31
.ilb s2 s1 s0
.ob load_reg0 load_reg1 load_reg2 reg3_0 load_reg3
  reg4_0 load_reg4 reg5_0 load_reg5 reg6_0 load_reg6
  reg7_0 load_reg7 +0_1_1 +0_1_0 +0_r_1 +0_r_0 +1_1_0
  +1_r_0 -0_1_2 -0_1_1 -0_1_0 -0_r_1 -0_r_0 *0_1_2
  *0_1_1 *0_1_0 *0_r_1 *0_r_0 >0_1_0 >0_r_0
.p 8
000 11100000000000000000000000000000
100 00000000100000000000001000000000
11- 00000000000000001001000010000000
-01 00001010000000010000100000000100
0-1 00000000000010000000101000100000
010 00000000101000000000001010101000
1-1 00010100000001000110011100011100
1-0 00000001011110101000001001110111
.e
```

B.20 STATS.SPEC

This is the output file that describes the specifications met by the current data path. It gives the data path delay, the area and delay of the combinational logic units, the data path area, the design's total area, the minimum length of the clock cycle, and the total execution delay.

```
data_path_delay: 35.000000
combo1:  area: 21368.175781 delay: 13.887800
combo2:  area: 67213.046875 delay: 21.539600
data_path_area 840190.687500
total_area: 945407.875000
clock_cycle_time: 75.427399
total_execution_delay: 2715.386353
```

B.21 CADSS.OUT

This is the screen output of CADSS listing the the number of functional units used in each pass and the resulting delay and area estimates.

```
pass: 1
trying:  and 1 > 1 + 1 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 925612.687500 delay: 3381.589600 pda: 0.157015 pdt:
     0.165864
best operator: and 1
```

```
pass: 2
trying:  and 2 > 1 + 1 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 942512.687500 delay: 3381.589600 pda: 0.178140 pdt:
     0.165864
best operator: and 1
pass: 3
trying:  and 3 > 1 + 1 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 959412.687500 delay: 3381.589600 pda: 0.199265 pdt:
     0.165864
best operator: and 1
pass: 4
trying:  and 1 > 2 + 1 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 985765.875000 delay: 3381.589600 pda: 0.232207 pdt:
```

```
     0.165864
best operator: > 1
pass: 5
trying:  and 1 > 3 + 1 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1045919.125000 delay: 3381.589600 pda: 0.307398 pdt:
     0.165864
best operator: > 1
pass: 6
trying:  and 1 > 1 + 2 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 945407.875000 delay: 2715.386475 pda: 0.181759 pdt:
     -0.063821
best operator: + 2
pass: 7
trying:  and 1 > 1 + 3 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
```

```
execime
area: 1008782.875000 delay: 2715.386475 pda: 0.260978 pdt:
     -0.063821
best operator: + 2
pass: 8
trying:  and 1 > 1 + 4 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
execime
area: 1072157.875000 delay: 2715.386475 pda: 0.340197 pdt:
     -0.063821
best operator: + 2
pass: 9
trying:  and 1 > 1 + 2 - 2 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
execime
area: 1053549.750000 delay: 1981.446899 pda: 0.316936 pdt:
     -0.316860
best operator: - 1
pass: 10
trying:  and 1 > 1 + 2 - 3 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
```

```
combocost combo2
dparea
exectime
area: 1125374.750000 delay: 1981.446899 pda: 0.406718 pdt:
     -0.316860
best operator: - 1
pass: 11
trying:  and 1 > 1 + 2 - 1 * 2
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1171970.375000 delay: 2715.386475 pda: 0.464962 pdt:
     -0.063821
best operator: * 1
pass: 12
trying:  and 1 > 1 + 2 - 1 * 3
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1398532.875000 delay: 2715.386475 pda: 0.748165 pdt:
     -0.063821
best operator: * 1
pass: 13
trying:  and 1 > 1 + 2 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
```

```
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 945407.875000 delay: 2715.386475 pda: 0.181759 pdt:
    -0.063821
best operator: and 1
pass: 14
trying:  and 2 > 1 + 2 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 962307.875000 delay: 2715.386475 pda: 0.202884 pdt:
    -0.063821
best operator: and 1
pass: 15
trying:  and 1 > 2 + 2 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1005561.062500 delay: 2715.386475 pda: 0.256951 pdt:
    -0.063821
best operator: > 1
pass: 16
trying:  and 1 > 3 + 2 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
```

```
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1065714.250000 delay: 2715.386475 pda: 0.332142 pdt:
     -0.063821
best operator: > 1
pass: 17
trying:  and 1 > 1 + 3 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1008782.875000 delay: 2715.386475 pda: 0.260978 pdt:
     -0.063821
best operator: + 2
pass: 18
trying:  and 1 > 1 + 4 - 1 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1072157.875000 delay: 2715.386475 pda: 0.340197 pdt:
     -0.063821
best operator: + 2
pass: 19
trying:  and 1 > 1 + 2 - 2 * 1
asap
genlife
regal
```

```
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1053549.750000 delay: 1981.446899 pda: 0.316936 pdt:
     -0.316860
best operator: - 1
pass: 20
trying:  and 1 > 1 + 2 - 3 * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1125374.750000 delay: 1981.446899 pda: 0.406718 pdt:
     -0.316860
best operator: - 1
pass: 21
trying:  and 1 > 1 + 2 - 1 * 2
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1171970.375000 delay: 2715.386475 pda: 0.464962 pdt:
     -0.063821
best operator: * 1
pass: 22
trying:  and 1 > 1 + 2 - 1 * 3
asap
```

```
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
area: 1398532.875000 delay: 2715.386475 pda: 0.748165 pdt:
     -0.063821
best operator: * 1
asap
genlife
regal
dpg
cntrl2esp
espresso combo1
c2toesp
espresso combo2
combocost combo1
combocost combo2
dparea
exectime
exectime
unable to meat both area and time criteria.
This information can be found int stats.spec.
```

REFERENCES

1. M. C. McFarland, S.J. and T. J. Kowalski. Assisting the DAA: The Use of Global Analysis in an Expert System. AT&T Bell Laboratories, October 1986.

2. C. Y. Hitchcock III and D. E. Thomas. A Method of Automatic Data Path Synthesis. Proceedings of the 20th Design Automation Conference, Miami Beach, Florida, 1983, pp. 484-489.

3. T. J. Kowalski and D. E. Thomas. The VLSI Design Automation Assistant: What's in a Knowledge Base. Proceeding of the 22nd Design Automation Conference, Las Vegas, Nevada, 1985, pp. 252-258.

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

5. P. G. Paulin and J. P. Knight. "Algorithms for High-Level Synthesis". *IEEE Design & Test of Computers 6* (December 1989), 18-31.

6. R. Rudell et al. *ESPRESSO.* Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, 1985.

7. P. G. Paulin and J. P. Knight. Force-Directed Scheduling in Automatic Data Path Synthesis. Proceedings of the 24th Design Automation Conference, Miami Beach, Florida, 1987, pp. 195-202.

8. H. Trickey. "Flamel: A High-Level Hardware Compiler". *IEEE Transactions on Computer-Aided Design CAD-6* (March 1987), 259-269.

9. P.G. Paulin et al. HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis. Proceedings of the 23rd Design Automation Conference, Las Vegas, Nevada, 1986, pp. 263-270.

10. P. P. Gelsinger et al. Microprocessors Circa 2000. Intel Corporation, 1989. As presented at a Lehigh Univeristy IEEE student branch meeting.

11. M. C. McFarland, S.J. . Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions. Proceedings of the 23rd Design Automation Conference, Las Vegas, Nevada, 1986, pp. 474-480.

12. J. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". *IEEE Transactions on Computers C-30* (July 1981), 478-490.

13. P. Marwedel. A New Synthesis Algorithm for the MIMOLA Software System. Proceedings of the 23rd Design Automation Conference, Las Vegas, Nevada, 1986, pp. 271-277.

14. R. K. Brayton et al. "MIS: A Multiple-Level Logic Optimization System". *IEEE Transactions on Computer-Aided Design CAD-6* (November 1987), 1062-1081.

15. F. J. Kurdahi and A. C. Parker. REAL: A Program for REgister ALlocation. Proceedings of the 24th Design Automation Conference, Miami Beach, Florida, 1987, pp. 210-215.

16. D. Gregory et al. SOCRATES: A System For Automatically Synthesizing and Optimizing Combinational Logic. Proceedings of the 23rd Design Automation Conference, Las Vegas, Nevada, 1986, pp. 79-85.

17. L. W. Nagel. SPICE2: A Computer Program to Simulate Semiconductor Circuits. Tech. Rept. ERL-M520, Electronics Research Laboratory, University of California, Berkely, 1975.

18. M. C. McFarland, A. C. Parker, R. Camposano. Tutorial on High-Level Synthesis. Proceedings of the 25th Design Automation Conference, Anaheim, California, 1988, pp. 330-336.

VITA

William Richard Migatz was born in Port Jefferson Station, New York on April 6, 1966 to Melvin and Jean Migatz. He graduated from Comsewogue Senior High School in June, 1984. In May, 1988, he graduated summa cum laude from Manhattan College, receiving a Bachelor of Science Degree in Electrical Engineering with a minor in Computer Science.