FUNCTIONAL PROOF OF CORRECTNESS TECHNIQUES APPLIED TO RISC SIMULATOR

By

MARJORIE HYATT TURNER

Bachelor of Science

Indiana State University

Terre Haute, Indiana

1981

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fullfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1983

FUNCTIONAL PROOF OF CORRECTNESS TECHNIQUES APPLIED TO RISC SIMULATOR

Thesis Approved:

Donald D Fisher
Thesis Adviser

James R. Van Doren

[illegible]

Norman N Durham
Dean of the Graduate College

PREFACE

The performance of functional proof of correctness techniques is examined in this study by applying the techniques to a RISC simulator. The functional correctness theory is discussed, and simple examples illustrating the proof techniques are given. Then functional correctness is used to prove the correctness of some of the procedures in a RISC simulator. Finally the proof techniques are applied to a newly designed procedure of the RISC simulator.

I wish to acknowledge Arthurine Breckenridge, Dean Knight, and John Jagoe for their work on the RISC simulator. Also I wish to thank my committee members Dr. G. E. Hedrick and Dr. J. R. Van Doren for their contributions and advice, and Dr. Michael J. Folk for substituting during my oral examination. Dr. Hedrick, I hope that your foot in mouth problems begin to recede upon my departure from the university.

Finally to express my sincerest appreciation to my major advisor, Dr. D^2 Fisher, for his help on this thesis and for his continual assistance throughout my studies at Oklahoma State University, I raise my glass of water in thanks.

TABLE OF CONTENTS

TABLE

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Structured programming may be viewed as a relatively new approach to programming; however, the concept has been developing for at least fifteen years. As early as 1965 there was the suggestion of the elimination of the GOTO statement by Dijkstra at the IFIP Congress [16]. Later in 1968 Dijkstra [5] reiterated his opinion in a letter to the editor of the Communications of the ACM. Since Dijkstra made his suggestion in 1965, structured programming has developed well beyond just the elimination of GOTO statements, even though it is still commonly defined as gotoless programming. Structured programming is a philosophy of designing and writing a program in an organized pattern using a set of basic logic structures, function, sequence, ifthen, ifthenelse, whiledo, dountil, to form the program [7, 16]. A goal of structured programming is to improve readability and maintainability. Another goal is to have the program written in a manner such that systematic verification techniques which include proving the correctness of the program at various points can be applied. The objective of proving the correctness of the program in the design stage is to eliminate logic errors, inconsistencies, or even weaknesses prior to coding and testing the program [7, 16].

The goals of readability and maintainability are the two advantages most often stressed about structured programming; however, proof of correctness was one of the initial motivations behind developing structured programming. Dijkstra [4] states in his paper presented at the proceedings of the NATO conferences

> A number of people have shown that program correctness can be proved. . . . As is to be expected . . . the circulating examples are concerned with rather small programs and, unless measures are taken, the amount of labour involved in proving might well (will) explode with program size. Therefore, I have not focused my attention on the question 'how do we prove the correctness of a given program?' but on the questions 'for what program structures can we give correctness proofs without undue labour, even if the programs get large?' and, as a sequel, 'how do we make, for a given task, such a well-structured program?' (p. 223).

Thus, proof of correctness techniques preceded structured programming and have been developing along with it. At the present time a few different techniques exist such as an axiomatic approach which Hoare wrote about as early as 1969 [6], inductive assertion, loop invariant, and functional correctness [1, 2, 3, 8, 9, 10, 11]. Some of the different techniques are compared in Basili [1] and Basili [2].

Even though proof of correctness techniques have been in existence for several years, they do not appear to be widely presented or practiced. Recently, though, IBM has begun to move toward a well-defined structured programming approach including proof of correctness of programs where the correctness method that IBM has chosen is the functional correctness approach [8]. In this study the functional correctness techniques as described in the IBM lecture notes prepared for a 1983 software engineering workshop [8] and in Mills' [9] book are applied to a RISC, Reduced Instruction Set Computer, simulator program written by this author and three other colleagues during the 1983 Spring semester at Oklahoma State University.

The functional correctness approach consists of proving that the intended program function, the specification of what the program is suppose to do, is either equivalent to or a subset of the derived program function, the actual result of the program implementation. In mathematical terms if f is the defined program function and if g is the derived program function, then one of the following must be shown for program correctness:

1. $f = g$
2. $f \subset g$.

The first is called complete correctness and the second, sufficient correctness [1, 2, 3, 8, 9, 10, 11].

In proving either complete or sufficient correctness of a program P, the program is decomposed into prime programs. Basically, a section of a program is a prime program if it is a complete part and can not be broken down into smaller integral program parts. The prime programs are proved correct, and the proof of correctness of P becomes a bottom-up verification starting with verifying the inner most prime program and working upward and outward [9, 10]. Six control structures that are basic to structured programmming are prime programs. These six structures are function, sequence, ifthen, ifthenelse, whiledo, and dountil. A specific proving form exists for each structure, and these techniques for program verification are derived from the Correctness Theorem described in Mills [9] and Manna [10]. An important aspect of the functional correctness approach is that the program must be a structured program because the proofs depend on the structured programming control structures.

The RISC simulator mentioned previously is based on a description of RISC in Patterson [15] with some assumptions made by the authors where the RISC description is incomplete. The procedures used for proof of correctness are those procedures that simulate the RISC instuctions such as XOR (exclusive or), LDL (load long), STS (store short), and SLA (shift left arithmetic). Furthermore, a new arithmetic procedure was written from the beginning to include two new subtract statements described in a more recent article on RISC [13], and functional proof of correctness is applied to this new procedure which is then inserted into the RISC simulator and tested with a goal of zero logic errors. The purpose of applying the proof of correctness techniques to the procedures already programmed and tested is to provide an example of the techniques applied to a functional program and to determine whether the proofs point out any errors in the procedures that may not have been found during testing. The purpose of proving the new procedure correct is to see if the proof of correctness techniques eliminate logic errors of a procedure before coding and testing. Finally, the effectiveness of the functional correctness techniques as applied to the prewritten procedures of RISC and the newly written procedure of RISC is evaluated.

CHAPTER II

BRIEF RISC DESCRIPTION

General Concept

The incentive for developing the Reduced Instruction Set Computer, RISC, was to provide an alternative to the present day trend toward increasingly complex instruction sets which lead to complex architectural designs. The idea behind RISC is to provide an architecture that minimizes complexity and supports high level languages while reducing design time and design errors, making more effective use of the resources on a single chip, and forming a machine with high throughput [14, 15]. In order to achieve these goals, the designers restricted the instruction set and implemented special architectural features that support fast execution of the reduced instruction set. Some of these special features of RISC include single cycle execution, restricted memory access instructions, prefetched instructions, window registers, and uniform instruction size.

Special Architectural Features

Two of the architectural features are single cycle execution and restricted memory access. These two features improve performance of RISC, reduce chip size, and simplify the design. Single cycle execution implies that one instruction is executed per cpu cycle. The cycle time for RISC is determined by the time it takes to read a register, perform an ALU operation, and store the result back into a register [15]. All RISC instructions execute in one cycle except for load and store instructions. Load and store instructions are the only instructions to access memory, and because they access memory, they take 2 cpu cycles, adding the index register and the immediate offset in the first cycle and accessing memory in the second cycle [14].

Restricting memory access to load and store instructions in RISC differs from other computers which allow numerous instructions to access memory. This difference, though, simplifies the design of RISC. Single cycle execution improves performance and reduces the chip size because the speed of the single cycle instruction execution is equivalent to that of a micro instruction in other machines, and the RISC instructions are no more complicated than a micro instruction; consequently, RISC can eliminate one level of abstraction because microcode control is not necessary in RISC [15].

To increase performance, the designers implemented an instruction prefetch which fetches the next instruction in sequence while the current instruction is being executed, so the execution cycle of the current instruction is overlapped with the prefetch and decoding of the next instruction [14]. Prefetching an instruction improves performance, but on the other hand, it introduces a problem with branch instructions such as jumps or subroutine calls. The problem is a successful branch can make the prefetching useless because after the execution of a successful branch, the prefetched instruction is not the next instruction to be executed. So to solve this problem without using elaborate techniques which would add a complexity counter to the objectives of RISC, the designers of RISC set up a delayed jump. With a delayed jump the branch does not take effect until after the execution of the instruction following the branch instruction; consequently, the instruction prefetch is no longer useless during a successful branch. The reason that the prefetching is no longer useless is that the instruction prefetched during the execution of the branch instruction is now the next instruction executed, and during this instruction's execution, the instruction prefetched is the instruction where control was transferred by the branch instruction. However, the delayed jump can detract from the advantages of the instruction prefetch because sometimes the delayed jump necessitates the inclusion of a no operation (NOP) instruction following the jump such as an add instruction that would add zero to a register; thus, because of the NOP, it is possible that a jump could be equivalent to two instructions. Figure 1 gives an example of the use of a NOP instruction. Figure 1a shows a sequence of instructions that is executed in the order 1, 2, 3, 5, 6. Figure 1b shows the sequence of RISC instructions that have an equivalent execution. Because of the delayed jump, if the NOP at line 4 were not included, the load following the jump would be executed since the jump would not occur until after the execution of the instruction immediately succeeding it.

Normal Jump	Delayed Jump
1 ADD	1 ADD
2 SUB	2 SUB
3 JUMP 5	3 JUMP 6
4 LOAD	4 NOP
5 STORE	5 LOAD
6 XOR	6 STORE
	7 XOR

a.) Normal Jump b.) Delayed Jump

Figure 1. Normal and Delayed Jumps

The motivation behind the register setup of RISC is to speed up the subroutine calls which are more prevalent in RISC than in more complex instruction set computers because instructions in a complex instruction set computer are often implemented as subroutines in RISC. The two processes that cause subroutine calls to be time consuming are saving or restoring registers and passing parameters. RISC effectively eliminates the time consumed in saving and restoring registers by setting up a system of register banks so that registers do not need to be saved or restored for subroutine calls. Instead a pointer is changed, and a different set of registers is used for the called procedure; however, there is some overlapping of registers between called and calling procedures to support straightforward parameter passing. In RISC the registers currently being accessed are called window registers because changing the pointer can be visualized as moving a window over the registers to be used.

The window registers are set up such that 32 registers are always available. These 32 registers are divided into 3 sets: global, local, and parameter. Global registers are always registers 0-9 and are not included in the window. Registers 16-25 are local registers, registers 10-15 are the parameters to be passed to a called procedure, and registers 26-31 are the parameters passed by a calling procedure. Conceptually, registers 10-15 are called low registers and 26-31 are called high registers. The low registers of the calling procedure overlap with the high registers of the called procedure providing parameter passing between the two procedures. Figure 2 [13, 14, 15] gives a visual representation of the window registers.

```
                                    -------------------------
 -------------------------          |        High A         |
 |       26 High 31      |          -------------------------
 -------------------------          |                       |
 |                       |          |        Local A        |
 |       16 Local 25     |          |                       |
 |                       |          -------------------------
 -------------------------          |     Low A/High B      |
 |       10 Low 15       |          -------------------------
 -------------------------          |                       |
 -------------------------          |        Local B        |
 |                       |          |                       |
 |       0 Global 9      |          -------------------------
 |                       |          |         Low B         |
 -------------------------          -------------------------
a.) Register Partitioning           b.) Overlapped Registers
                                        Proc A Calls Proc B
```

Figure 2. RISC Window Registers

RISC has 138 registers, and, again conceptually, the window begins at the top of the registers and moves down for a subroutine call; thus, the window pointer is decremented for calls and incremented for returns. A register overflow stack exists in memory in the event that a series of nested subroutine calls exhausts the register banks.

Instruction Description

The design of the RISC instruction is another special feature of RISC. The RISC instruction was designed to promote simplicity of implementation and addressing. The instructions are all 32 bits long. The format of the 32 bit instruction, however, does provide a little flexibility in the operand specification. Figure 3 [14] shows the instruction format and the slight flexibility that it allows. There are basically six fields in the instruction, opcode, set condition code indicator (SCC), destination register, source one register, immediate value indicator (IMM), and source two register or immediate value. In the case of JMPR and CALLR instructions there are only four fields because the last three fields are combined to define one operand. Two of the six fields, SCC and IMM, are single bit fields. The SCC bit indicates whether the condition codes are to be set, and the IMM bit indicates whether the source two field is a register reference specified by the last 5 bits of the instruction (IMM=0) or whether source two is a 13 bit sign extended immediate value.

```
-----------------------------------------------------------------
| opcode | SCC |   Dest   | Sourcel | IMM | (unused) Source2  |
|  <7>   | <1> |   <5>    |   <5>   | <1> |   <8>      <5>    |
-----------------------------------------------------------------
| opcode | SCC |   Dest   | Sourcel | IMM | Immediate srce2   |
|  <7>   | <1> |   <5>    |   <5>   | <1> |      <13>         |
-----------------------------------------------------------------
| opcode | SCC |   Dest   |      Immediate operand            |
|  <7>   | <1> |   <5>    |            <19>                   |
-----------------------------------------------------------------
```

Figure 3. RISC Instruction Format

The final instruction set consists of 31 instructions [13, 14]. These instructions are divided into four groups, arithmetic-logical, memory access, branching, and miscellaneous. As previously mentioned, only load and store instructions access memory, and there are eight memory access variations allowing for 8 bit, 16 bit, and 32 bit sign-extended or zero-extended data. Table I shows the four groups of the instruction set and the definition of each instruction (the instructions shown in the table are the 28 instructions of the original RISC as given in Patterson [15] and two additional subtract instructions from later designs [13, 14]). Besides supporting data of 8 bits, 16 bits, and 32 bits, RISC also supports addresses of 32 bits. Furthermore, even though it initially looks as if only one addressing mode is offered, by using register zero which always contains a zero, addressing modes of indexed, absolute, and register indirect are possible.

Summary

Thus, the various features of RISC, a restricted instruction set and architectural support for fast execution, combine to reduce design time and design errors and make effective use of the resources on one chip. Furthermore, this type of architecture can be used to obtain a machine of high throughput.

The majority of this brief description of RISC was based on Patterson [15], one of the earlier RISC descriptions. More current, though similar, information on RISC may be found in Patterson [13] and Patterson [14].

TABLE I

ASSEMBLY LANGUAGE DEFINITION FOR RISC

MNE-MONIC	NAME	OPERANDS	ACTION
	ARITHMETIC-LOGICAL		
ADD	integer add	S1,S2,Rd	Rd<-S1+S2
ADDC	add with carry	S1,S2,Rd	Rd<-S1+S2+carry
SUB	integer subtract	S1,S2,Rd	Rd<-S1-S2
SUBC	subtract with carry	S1,S2,Rd	Rd<-S1-S2-carry
SUBR	subtract register	S1,S2,Rd	Rd<-S2-S1
SUBRC	subtract reg with carry	S1,S2,Rd	Rd<-S2-S1-carry
AND	logical and	S1,S2,Rd	Rd<-S1&S2
OR	logical or	S1,S2,Rd	Rd<-S1vS2
XOR	logical exclusive or	S1,S2,Rd	Rd<-S1 xor S2
SLA	shift left arithmetic	S1,S2,Rd	Rd<-S1 shift S2
SRA	shift right arithmetic	S1,S2,Rd	Rd<-S1 shift S2
SLL	shift left logical	S1,S2,Rd	Rd<-S1 shift S2
SRL	shift right logical	S1,S2,Rd	Rd<-S1 shift S2
	MEMORY ACCESS		
LDL	load long	(Rx),X,Rd	Rd<-M[Rx+X]
LDSU	load short unsigned	(Rx),X,Rd	Rd<-M[Rx+X]
LDSS	load short signed	(Rx),X,Rd	Rd<-M[Rx+X]
LDBU	load byte unsigned	(Rx),X,Rd	Rd<-M[Rx+X]
LDBS	load byte signed	(Rx),X,Rd	Rd<-M[Rx+X]
STL	store long	Rm,(Rx)X	M[Rx+X]<-Rm
STS	store short	Rm,(Rx)X	M[Rx+X]<-Rm
STB	store byte	Rm,(Rx)X	M[Rx+X]<-Rm
	BRANCHING		
JMP	conditional jump	COND,X(Rm)	pc<-X+Rm
JMPR	conditional relative	COND,Y	pc<-pc+Y
CALL	call	Rm,X(Rn)	Rm<-pc; pc<-X+Rn,CWP--
CALLR	call relative	Rm,Y	Rm<-pc; pc<-pc+Y,CWP--
RET	return	Rm,X	pc<-Rm+X,CWP++
	MISCELLANEOUS		
GTLPC	get last pc	Rm	Rm<-last pc
GTIN	get interrupt number	Rm	Rm<-INR

CHAPTER III

FUNCTIONAL PROOF OF CORRECTNESS THEORY

Different methods for proving the correctness of a program exist [1, 2, 3, 8, 9, 10, 11]. At least one of these methods, the axiomatic approach, preceded the introduction of structured programming; however, another approach, the functional correctness method, developed as an extension of structured programming. The functional proof of correctness technique requires that a program be structured because the basis of the proofs is dependent on the control structures of a structured program and the self-containment of program parts implied by the structuring. The technique is also based on the mathematical concept of functions as the name implies. The objective of the method is the comparison of the intended program function and the derived program function. Program structure and program functions form the foundation of the functional proof of correctness method.

Flowchart Symbols

Flowcharts are used to illustrate program structures and functions graphically. A flowchart consists of nodes and directed lines. Each node represents a program instruction, and the directed lines delineate the possible flow of control. The three node structures of a flowchart are shown in Figure 4. First is the function node characterized by having one in-line and one out-line. Next is the predicate node which has one in-line and two out-lines. In the flow of control one out-line is taken according to whether the decision represented by the predicate evaluates to true or to false. Conventionally, the upper line represents the true path; consequently, the out-lines of a predicate node are marked only in the case that there is an exception to this convention. The final structure is a collecting node characterized by 2 in-lines and one out-line [7, 8, 9, 11].

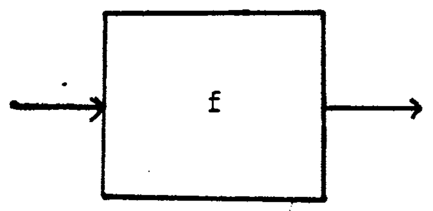
Prime Programs and Structured Programming

A proper program can be defined as a program with the following four properties:

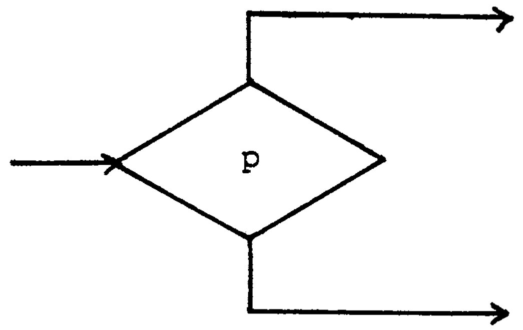
1. one entry
2. one exit
3. no unreachable code
4. no unleavable code [7].

Figures 5 and 6 [9] illustrate an example of a proper program and four programs that are not proper programs, each violating one of the properties of a proper program.

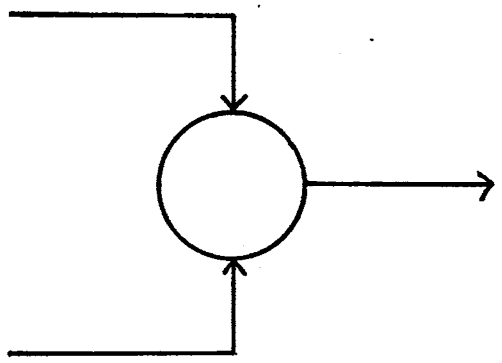
A proper program may contain parts that are themselves proper programs. These are called proper subprograms. A proper program that has no proper subprogram of more than



a.) Function Node

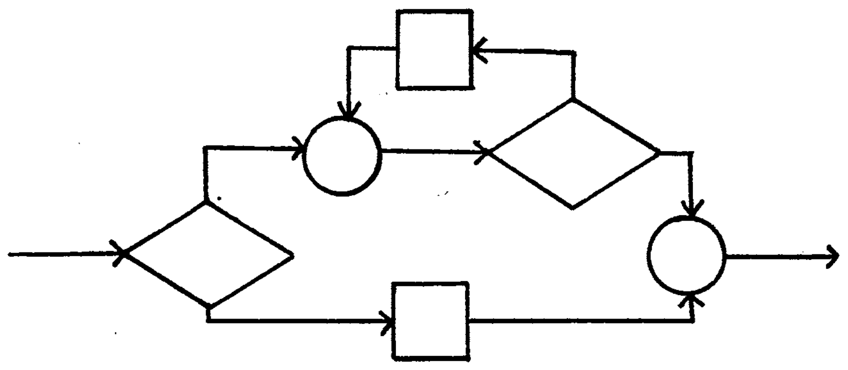


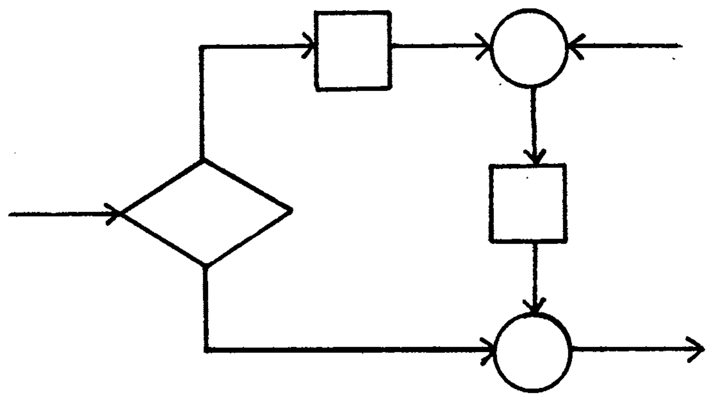
b.) Predicate Node



c.) Collecting Node

Figure 4. Flowchart Node Structures


a.) Proper


b.) Two Entries

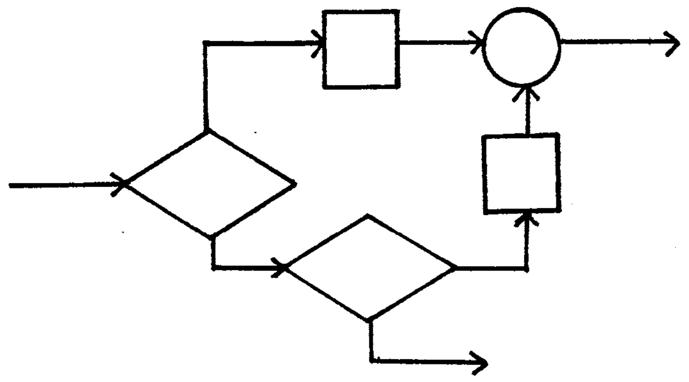
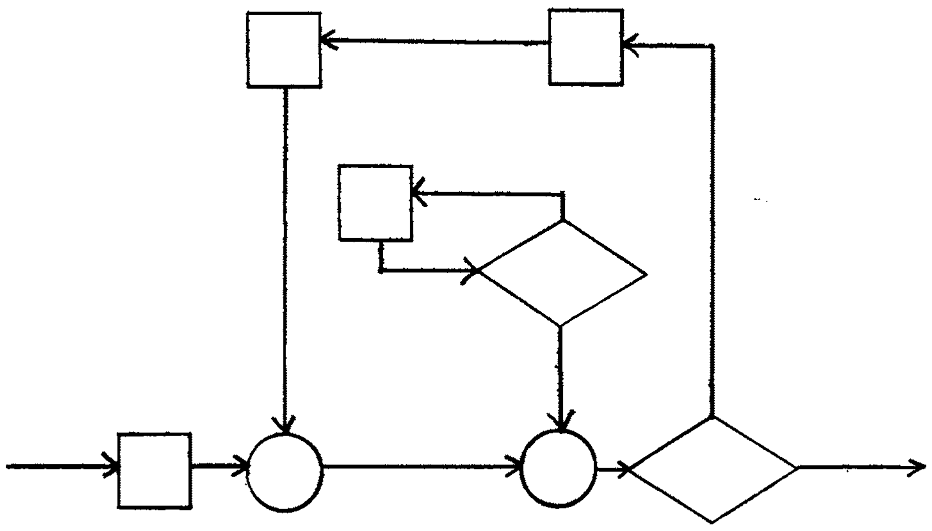

c.) Two Exits

Figure 5. Proper Program and Proper Program Violations


a.) Unreachable Code

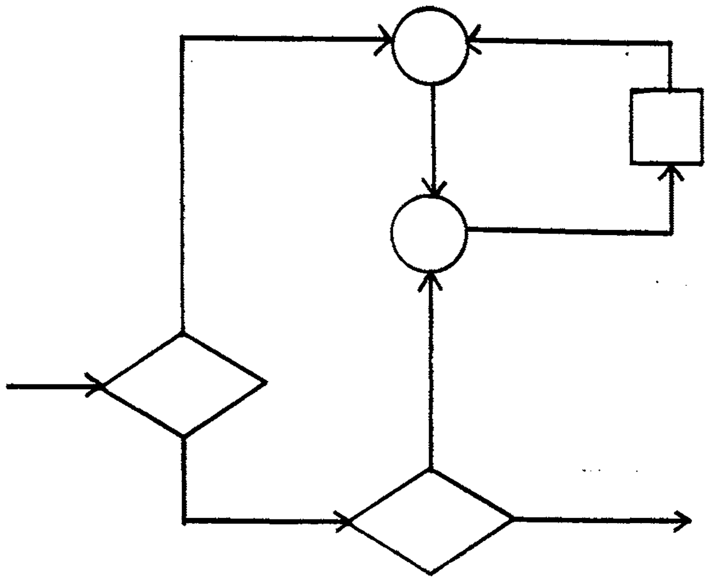

b.) Unleavable Code

Figure 6. Proper Program Violations Continued

one node is a prime program [8, 9]. Figure 7 [9] shows three proper programs containing proper subprograms of more than one node. These proper subprograms are shown in Figure 7. The proper subprograms of Figure 8 are also prime programs since these subprograms do not have any proper subprograms of more than one node. An analogy may be drawn between prime programs and prime numbers: the only factors of a prime number are itself and one, and the only proper subprograms of a prime program are itself and single-node proper subprograms.

A control structure is a representation of the ordering between function nodes, predicate nodes, and collecting nodes with no regard to the program text [8, 9]. A basis set is a fixed set of control structures [8, 9]. There are 6 control structures, function, sequence, ifthen, ifthenelse, dowhile, and dountil that form the basis set for a structured program. That is, a structured program can be constructed from these 6 control structures. If the prime programs of 1, 2, 3, and 4 nodes are enumerated, and the control structures that do not contain at least one function node are eliminated since they are not useful, the 6 control structures that make up the basis set for a structured program remain [8, 9]. Figure 9 illustrates these 6 structures. One other control structure, a dowhiledo, also remains. Figure 10a gives the structure of a dowhiledo. Mills [8] includes this structure in the basis set, but since the dowhiledo can be constructed from two subprograms

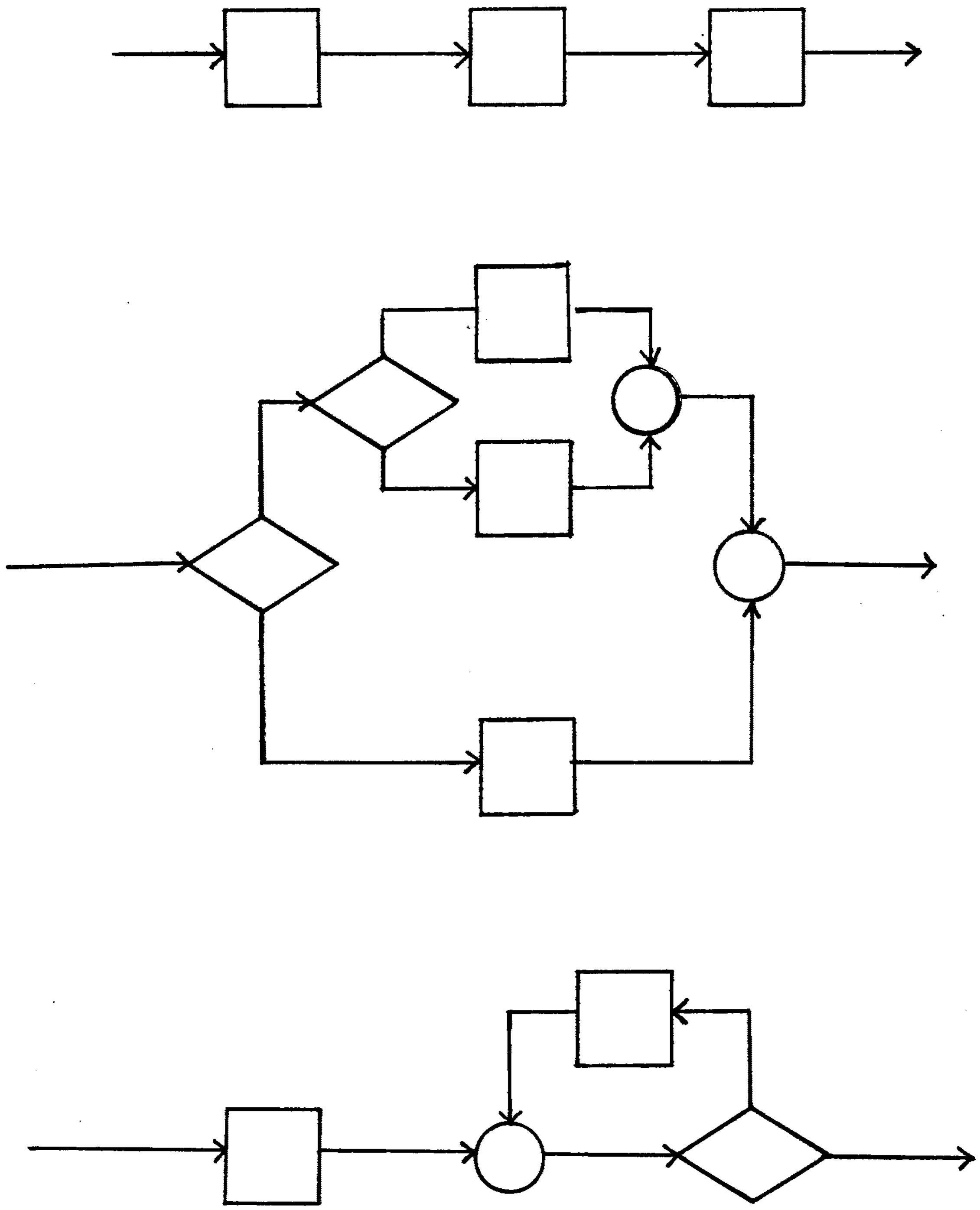


Figure 7. Proper Programs Containing Proper Subprograms of More Than One Node

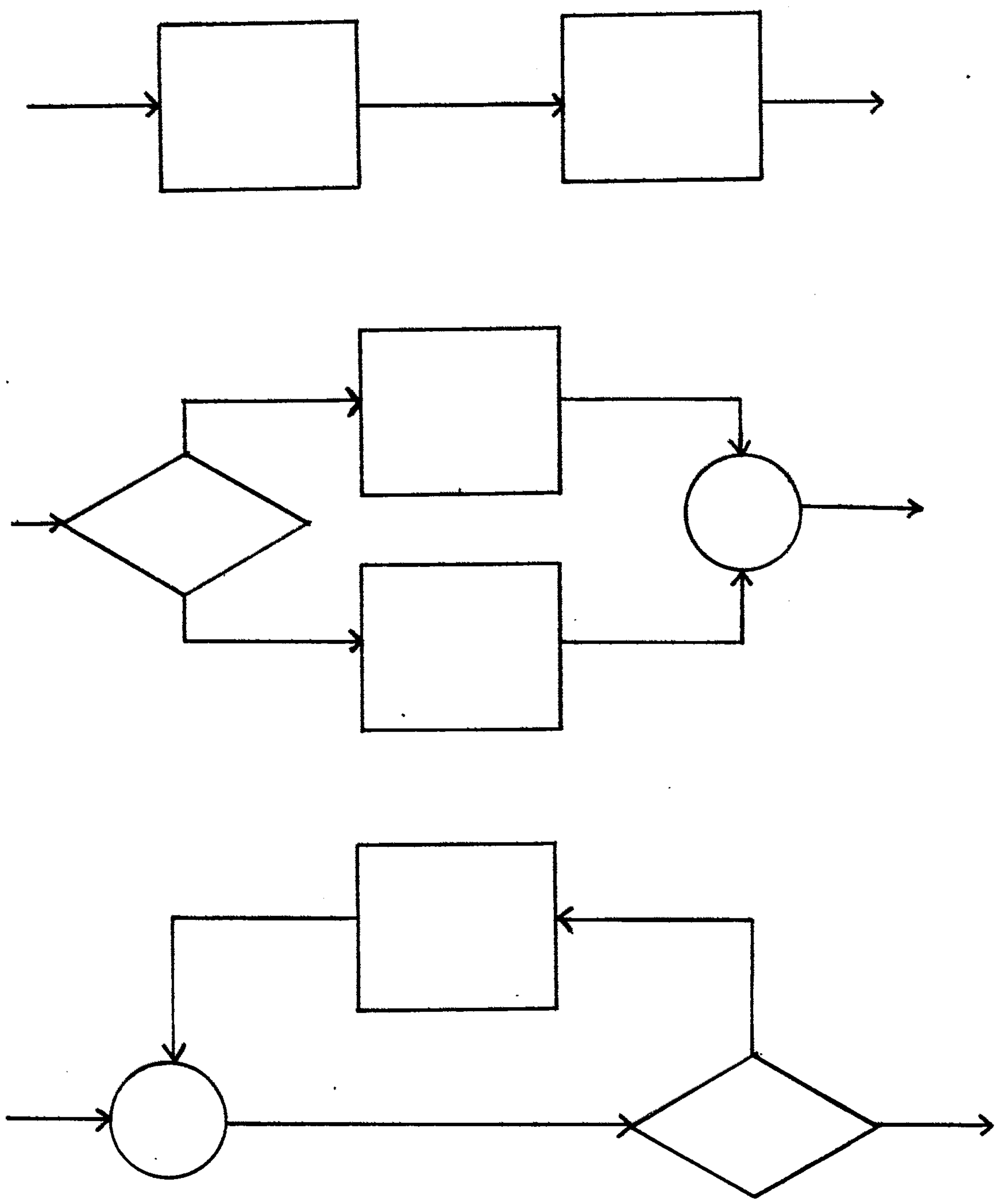
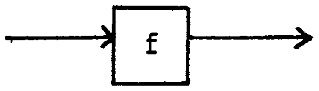
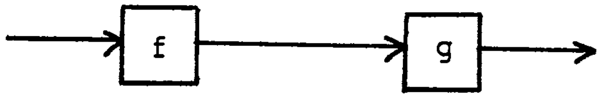


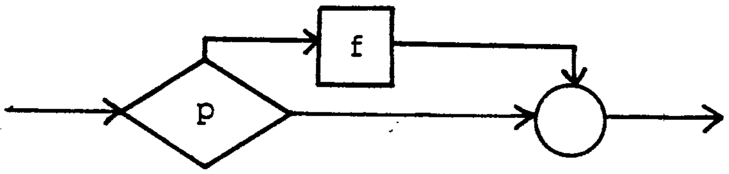
Figure 8. Proper Subprograms of the Proper Programs in Figure 7



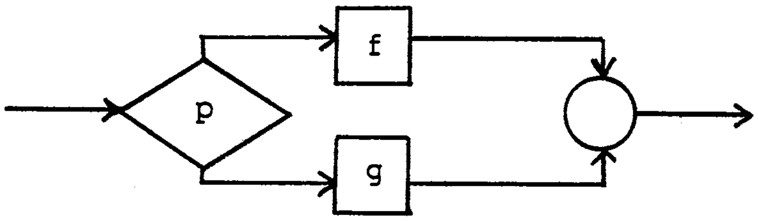
a.) Function f



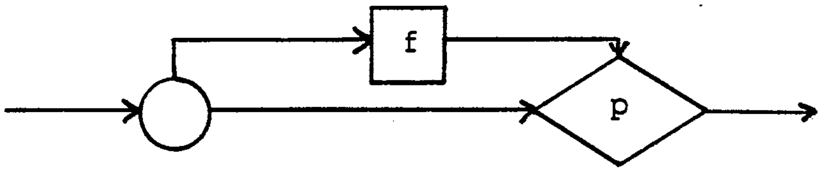
b.) Sequence f;g



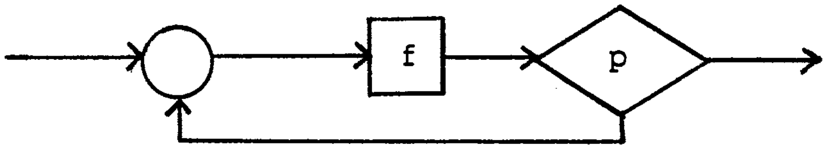
c.) Ifthen if p then f fi



d.) Ifthenelse if p then f else g fi

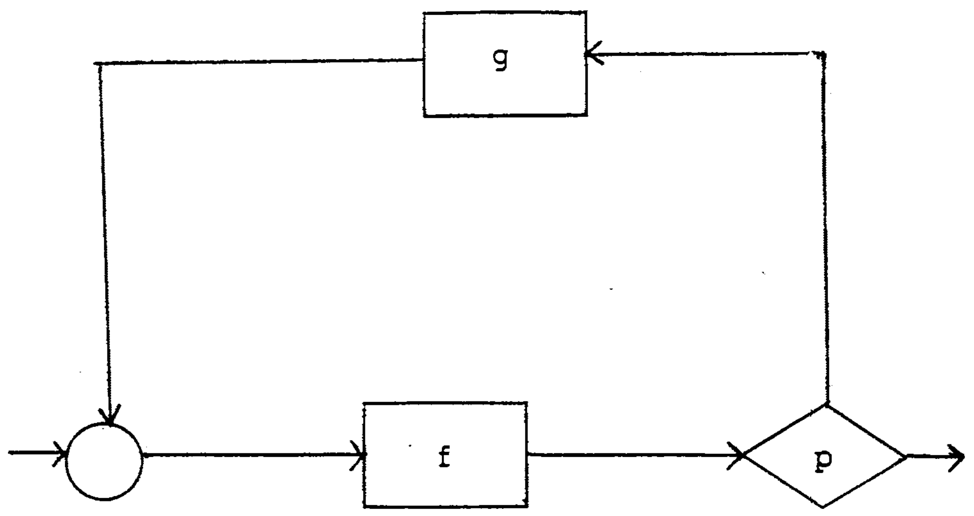


e.) Whiledo while p do f od

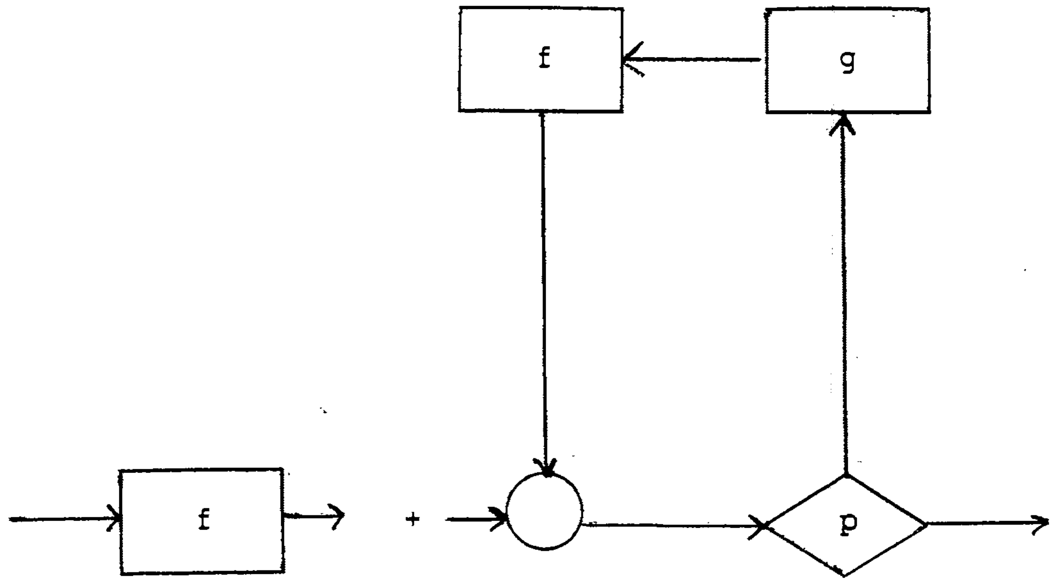


f.) Dountil do f until p od

Figure 9. Six Control Structures of Structured Programming



a.) do1 f while p do2 g od



b.) while p do g ; f od

Figure 10. Conversion of Dowhiledo Structure to an Equivalent Structure

as shown in Figure 10 [8], it is not included in the basis set in this report. This is consistent with some of the other sources such as Yourdan, IBM, and Hughes [7, 8, 16].

Program Functions

As stated earlier, the mathematical concept of functions is the basis of the functional correctness method. A function f is a set of ordered pairs with all first members unique. The notation $y = f(x)$ is used to indicate that the ordered pair (x,y) is an element of the function f. x is called the argument of f and y the value of f. The set of all arguments is the domain and the set of all values is the range [11]. A function may be expressed either by enumeration, listing the ordered pairs of the function, or by set notation, describing the function in words or mathematical notation within set brackets. An example of set notation is $\{(x,y) \mid x < y\}$ which is read "the set of all pairs x and y such that x is less than y."

A program determines a final data state given an initial data state. A program P contains variables $v_1, v_2, \ldots, v_n$. Each variable v_i may take on any value from a set of values d_i. The set DS of all possible combinations of variable values is the data space, $DS = d_1 \times d_2 \times \ldots \times d_n$. One element of DS, that is one combination of variable values, is a data state [1, 8]. Consequently, a program function is a mapping of a set of input data states into output data states. The function of a program is often represented by [P], read "bracket P" [9]. In set notation:

[P] = {(X,Y) | X is an initial data state and Y is the final data state after the program P is executed} [1, 9].

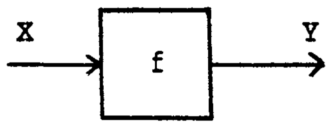
The domain of [P] is the set of all possible initial data states. Each element of the domain of [P] must be able to map to an output data state. The domain of [P] is either equal to or a subset of the data space [1, 8]. For example the following program

```
PROC addone(INOUT x: 1..3)
  x := x + 1;
CORP
```

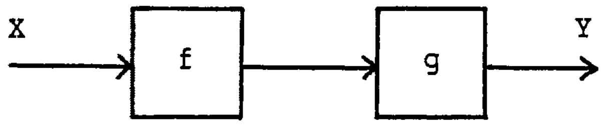
has the data space DS = {1,2,3}, which are the possible values for x. The domain D = {1,2} is a subset of the data space. The value 3 is not an element of the domain because 3 does not map into an output data state [8].

The functions of a program may be expressed in one of two ways, by use of set notation, or by use of conditional rules. Primarily conditional rules are used throughout this report, and set notation is used only briefly.

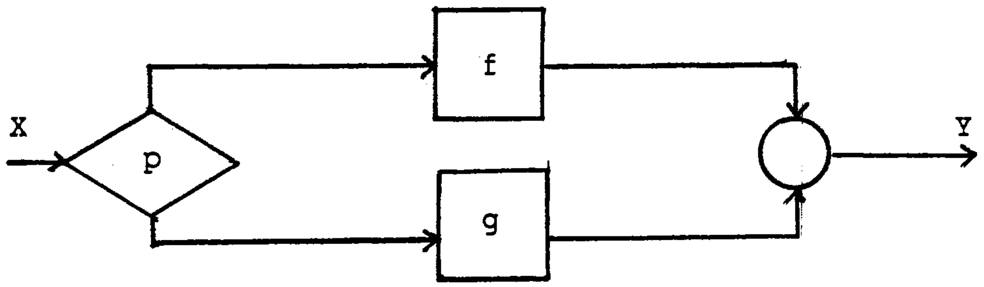
The program function of



is [P] = {(X,Y)| Y = f(X)}. The program function of the sequence



is [P] = {(X,Y)| Y = g(f(X))} where this function is the composition of the functions f and g. The program function of the ifthenelse



may be stated in set notation as

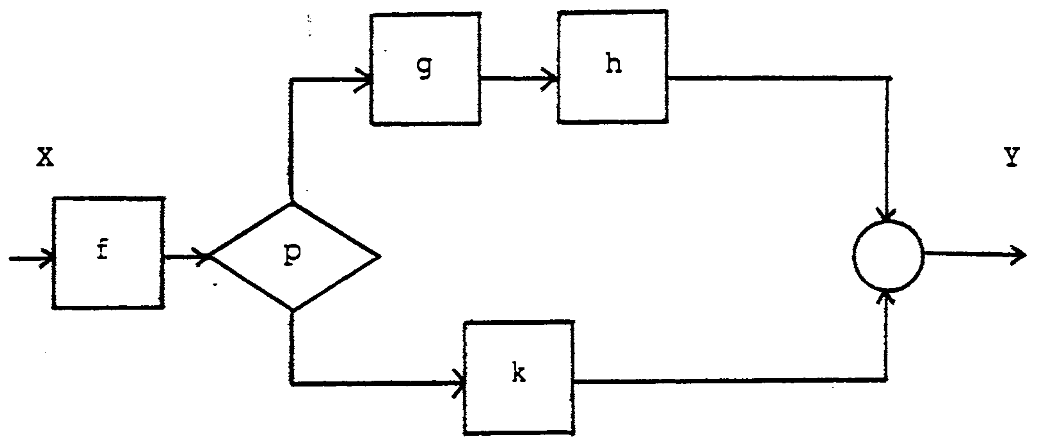
  [P] = {(X,Y)| p(X) & Y = f(X)} U

        {(X,Y)| ¬p(X) & Y = g(X)},

and as a conditional rule as

  [P] = (p(X) -> Y := f(X) | ¬p(X) -> Y := g(X)).

In the latter method, a condition, in this case the predicate p, implies (->) a data state transition. If the preceding three programs are combined to form the program

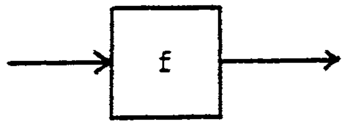


the program function developed for P is

$[P] = (p(f(X)) \rightarrow Y := h(g(f(X))) \mid$

$\neg p(f(X)) \rightarrow Y := k(f(X)))$ [9].

In a flowchart



represents a data state change. In a program an assignment statement represents a data state change. An assignment statement such as x := y+1 implies that the value of x is changed and the value of all other variables remains the same. The concurrent assignment x,y := y+1, y-1 means that the value of x and y have changed simultaneously, and again all other variables remain unchanged. It is important to note that the concurrent assignment implies that the value of y on the right hand side of the assignment x := y+1 is the value of y before the concurrent assignment y := y-1. A concurrent assignment may also be written x := y+1, y := y-1 with the comma indicating concurrency.

In illustration of a program function of a program that is less abstract than the previous program examples the program

```
IF
  a = 2
THEN
  x := b;
ELSE
  IF
    a is even
  THEN
    x := c;
  ELSE
    x := d;
  FI
FI
```

(the notational form of this program will be discussed in a later chapter) has the function

[P] = (a = 2 -> x := b| a is even -> x := c|
a is odd -> x := d) [8].

An alternate notation is

[P] = (x := (a = 2 -> b| a is even -> c| a is odd -> d)).

Another alternative is to use TRUE instead of the final condition a is odd. TRUE indicates a condition covering all other possibilities in the data space [1, 8, 9]. It is not as precise as actually stating the condition and must be used carefully. For example the following program functions are not the same:

1. [x > 0 -> x := x - 1| x < 0 -> x := 1]
2. [x > 0 -> x := x - 1| TRUE -> x := 1].

In 1 if x = 0, then there is no change in the value of x; however, in 2, x = 0 is part of the TRUE condition, so when x = 0, the value of x is changed to 1.

As previously mentioned a program maps an input data state into an output data state, and this mapping is the defined program function. There are actually two functions related to a program, the intended function and the derived function. The intended function is the stated functional intent of the program; whereas, the derived function is the actual mapping that occurs. Notationally, f represents the intended function and [P] represents the derived function, or alternately, if f represents the intended function, f' represents the derived function.

In the functional correctness technique the intended function and derived function are compared, and program correctness is proved if one of the following is true

1. $f = [P]$
2. $f \subset [P]$.

The first is called complete correctness, the same mapping and the same domain. The second is called sufficient correctness, same mapping for a common domain, but the derived function maps additional arguments that are not in the domain of the intended function [1, 2, 3, 8, 9, 10, 11]. Figure 11 [8] graphically illustrates the cases of incorrectness, complete correctness, and sufficient correctness. As an example of the levels of correctness the function $f = \{(1,M),(2,Tu),(3,W),(4,Th),(5,F)\}$ is an intended function. The following are three possible derived functions for f:

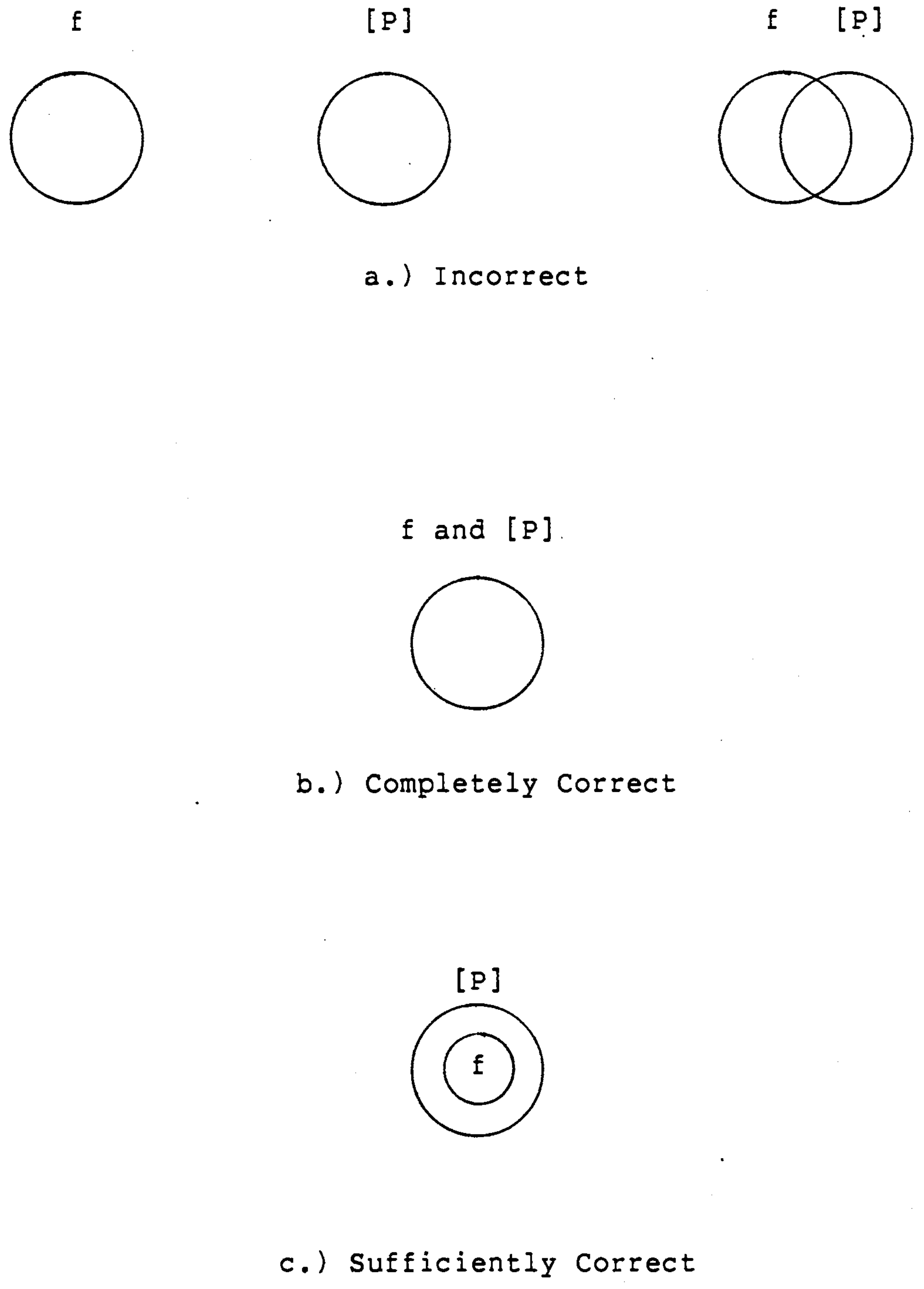


Figure 11. Graphical Illustrations of Functional Correctness

f' = {(1,Su),(2,M),(3,Tu),(4,W),(5,Th),(6,F),(7,Sa)}

f' = {(1,M),(2,Tu),(3,W),(4,Th),(5,F)}

f' = {(0,Su),(1,M),(2,Tu),(3,W),(4,Th),(5,F),(6,Sa)}.

The first demonstrates incorrectness, the second, complete correctness, and the third, sufficient correctness. In the sufficient correctness case the common domain of 1, 2, 3, 4, 5 are mapped to the same values, but two other arguments, 0 and 6 are also mapped to values by f' [8].

An important trait of functional proof of correctness that one should realize is that when the intended function is not equal to or not a subset of the derived program function, the proof does not resolve whether the function specification is incorrect or whether the function implementation is incorrect. Thus, both the program logic and the program specification should be considered when the reason for the failure of the proof is being determined.

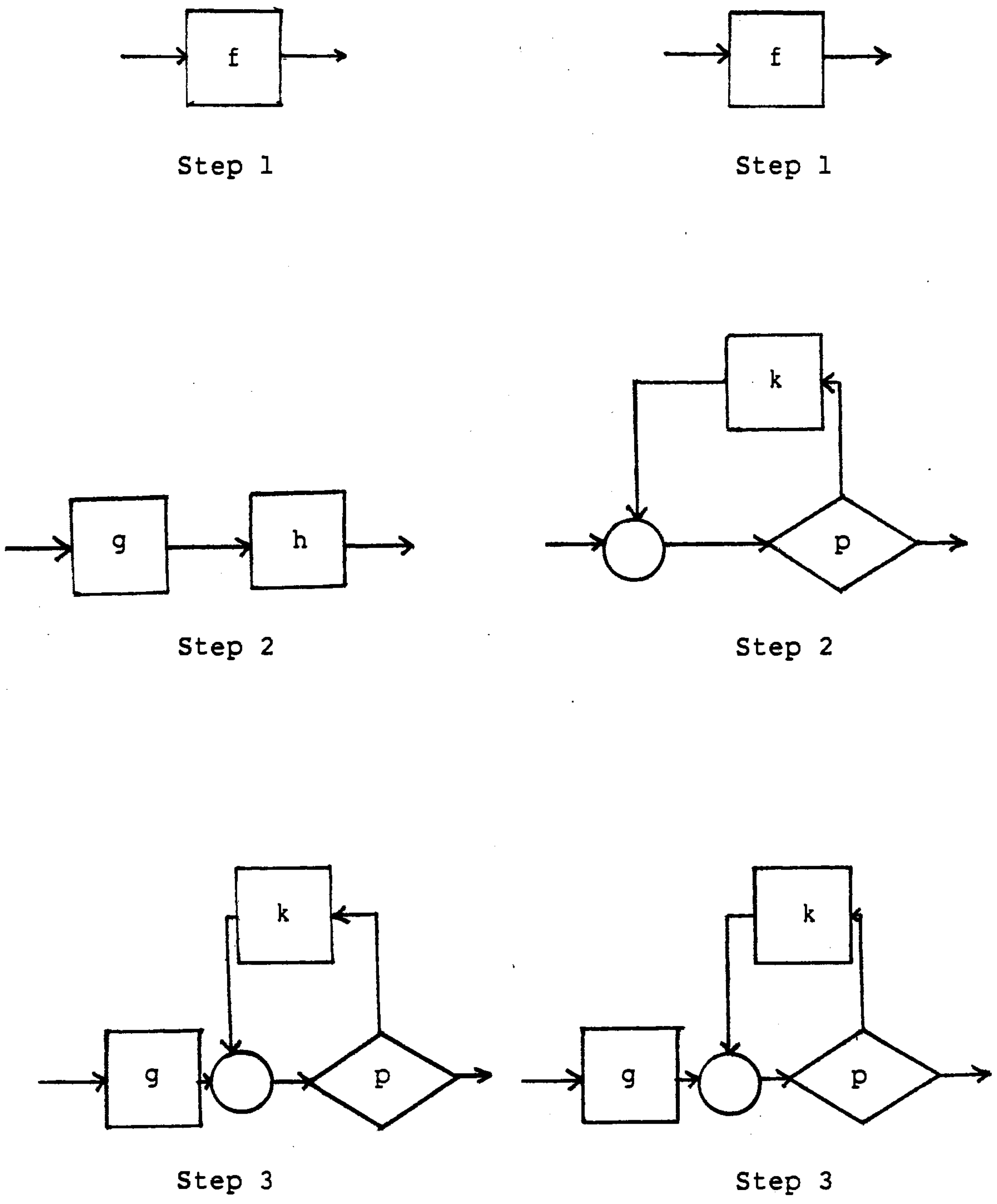
Stepwise Abstraction

A compound or composite program is a program which contains at least one proper subprogram of more than one node [8]. A structured program is a compound program constructed from a fixed basis set of prime programs (the 6 prime programs previously mentioned) [9]. A structured program begins as a single function and is developed into a compound program by a method called stepwise refinement. This is an iterative process consisting of replacing function nodes of a program by the prime programs in the basis set [8, 9, 11]. Figure 12 [11] illustrates the principle of stepwise refinement. Two replacement sequences are shown in Figure 11. In the first sequence the function f is replaced with the sequence structure g;h, and then h is replaced with the ifthen structure if p then k fi. This sequence shows stepwise refinement. The second sequence has the same final program; however, it does not follow stepwise refinement because of the discontinuity between steps 2 and 3. The function g is introduced in step 3 with no indication of its derivation. It did not come about by being replaced by a previous function [11].

When the functional correctness method is applied to prove the correctness of a program, stepwise abstraction which is the reverse of stepwise refinement is applied. Stepwise abstraction is an iterative process of replacing a prime program by a new function node until no prime programs remain to be replaced [9]. The final result is a program expressed as a single function node. Figure 13 illustrates an example of stepwise abstraction. The basis of both stepwise refinement and stepwise abstraction is the Axiom of Replacement:

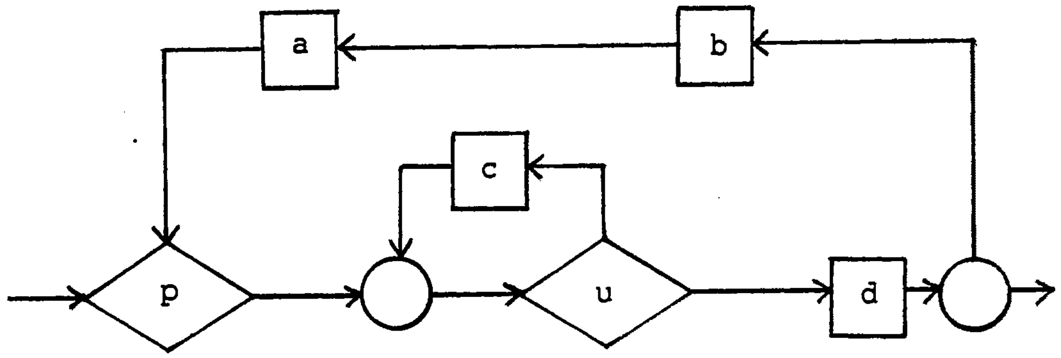
> Let P be a proper subprogram of Q and let the replacement of P by P' within Q result in Q'. Then [P] = [P'] -> [Q] = [Q'] (IBM [8] p. FN 7-07, Mills [9] p. 148)

Thus, the proof of correctness of a program P becomes a proof of correctness of each abstraction of P that results from the stepwise abstraction process. It consists of

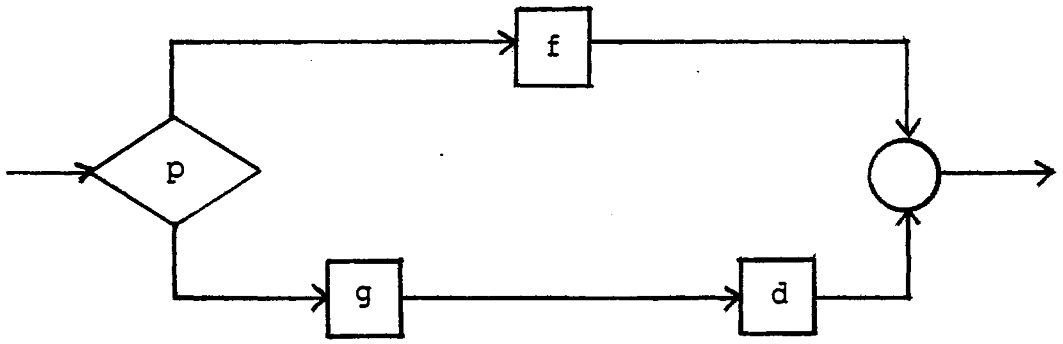


a.) Sequence 1 b.) Sequence 2

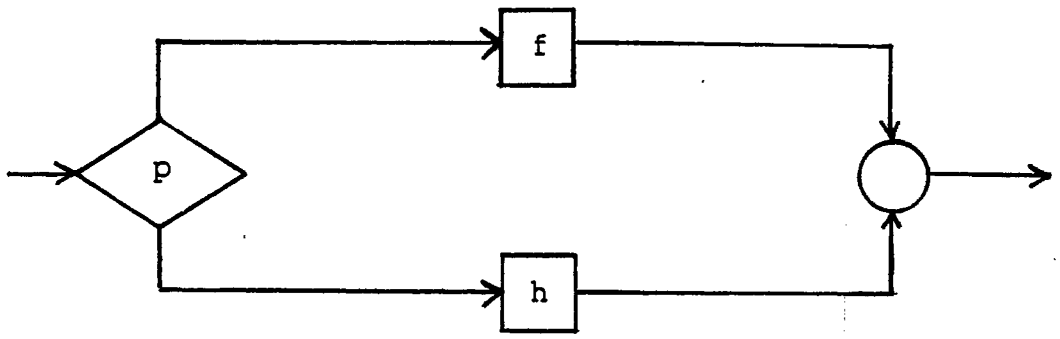
Figure 12. Illustration of Stepwise Refinement



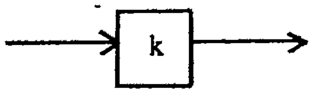
a.) Greatest Detail



b.) Step 1



c.) Step 2



d.) Step 3

Figure 13. Illustration of Stepwise Abstraction

proving the correctness of a proper subprogram of P which is usually a prime program, and replacing the subprogram with its intended function. P is now at a higher level of abstraction, and the process is repeated. For example, if the following program is given: F = if p then G else H fi, where G and H represent proper subprograms, then the approach taken might be to prove g = [G] and h = [H] where g and h represent the intended functions of subprograms G and H respectively. Then F becomes if p then g else h fi, and f = [F] may be proved [9]. Complete correctness results only if all the subprograms satisfy complete correctness; otherwise, if any of the subprograms satisfies only sufficient correctness, the whole program satisfies only sufficient correctness [9].

Program Correctness vs Program Verification

Proving the correctness of a program is not the same as program verification. There are many aspects to program verification that are not included in a correctness proof. The functional correctness method verifies a program's defined function, or in other words, this correctness method verifies that a program maps a specified set of input data states into the desired output data states. The emphasis is on the mapping and the domain. Two aspects not verified by proof of correctness are argument-parameter agreement and variable correctness.

Also, in some proof of correctness techniques such as the axiomatic approach the differentiation between local and global variables is important; however, this is not true in the functional correctness method [2]. The process of verification by stepwise abstraction relies on the self-containment of each proper subprogram verified, and the intended function is specified such that it encompasses only its designated subprogram. For example the intended function of

```
x,i := 0,1
WHILE
  i <= 2
DO
  x,i := x+1, i+1
OD
```

is x,i := 2,3. But the intended function of the whiledo subprogram without the preceeding initializations

```
WHILE
  i <= 2
DO
  x,i := x+1, i+1
OD
```

is x,i := x+3-i, 3. The function definition for this subprogram does not rely on the initialization of x or i. The concept illustrated in this example can be expanded to procedures and global variables. The specified intended function of a procedure does not rely on the possibility that a variable is a global variable and that its value may be effected by another procedure. This implies that functional correctness does not verify the interface among procedures with respect to global variables; however, there is partial verification. The proof does not establish that the correct global variable is used in a procedure, but it does guarantee the variable's functional value at the completion of a procedure.

Summary

The functional correctness method evolved from the premise that a program has a function that maps an input data state to an output data state. The intended program function is compared to the derived program function, and the result is the program is proved to be incorrect, completely correct, or sufficiently correct. In comparing the intended and derived functions of a program, one uses a process called stepwise abstraction. The 6 control structures of a structured program are important in stepwise abstraction because they are prime programs and in stepwise abstraction the objective is to replace each prime program with a single function until a program is represented by one function. Finally, program correctness and program verification are not equivalent. Program correctness is just one part of program verification and does not involve verifying all aspects of a program.

CHAPTER IV

PROOF SYNTAX AND METHODS

The main objective of proof of correctness is to eliminate program logic errors by applying a systematic mathematical approach of program validation. An important aspect of this objective is that the approach is methodical. For a program to be validated methodically and not to be confirmed randomly, guidelines for the proof process are established. Guidelines outlining the form in which the program and general proof are written provide the framework for the functional proof of correctness method. Also included in the framework is the form and objective of the proofs for each control structure. These guidelines promote thoroughness and correctness in the proof process and also document the program and proof. Once the framework is established, then various techniques can be employed to shape the body of the proof. These techniques include mental verification, table tracing, array and anonymous data handling, and conditional rule manipulation.

Program Design Form

A program design language or PDL is often used for the design stage writing of a program. There is no standardized PDL format; however, most PDL's have similar conventions. In this report the PDL format used is based on the format in Mills [9] augmented by some conventions from IBM [8].

An assignment statement is the basic statement in programming, and in PDL, a colon followed by an equal sign, :=, is the assignment symbol. Another general convention is enclosing self-contained sections of a program such as subprograms and procedures between a beginning keyword and an ending keyword. The ending keyword is the beginning keyword written backwards. For example the beginning keyword for a procedure is PROC, and the ending keyword is CORP. Keywords are written in capitals, and all other words are in small letters. In addition to writing nonkeywords in small letters, the text delineated by keywords is further delineated by being indented.

Since the intended function plays a major part in a functional correctness proof, the proper specification of the intended function for each part of the program is important. Brackets, [], are used to delineate the intended function, and its placement precedes the section of PDL which is to perform that function. The intended function of a whole procedure is placed after the variable section.

The control structures that make up a structured program are fundamental in the writing of a program. Each structure has a specified format and intended function placement. The structures whose formats are defined are the sequence, ifthen, ifthenelse, whiledo, and dountil which are the basic structures. Also two other structures, the fordo and the case which are special cases of the sequence and ifthenelse respectively, are described. These two structures are not necessary but are practical extensions to the basic six control structures because they enhance program design without detracting from program structure.

The sequence structure is composed of component operations written one below the other. Sometimes a semicolon is used to delimit the parts of the sequence when it is needed for clarity; otherwise, the semicolon is omitted. Usually there is no beginning or ending keyword delimiters for a sequence; however, DO-OD may be used if a sequence performs a specific function. The intended function is placed to the side of the DO in this case.

```
Ex.  DO [x,y,z := y,z,x]
       x := y
       y := z
       z := x
     OD
```

The formats of the other control structures are as follows:

```
ifthen
          [f]
          IF
            p
          THEN [intended function for thenpart]
            g
          FI


ifthenelse
          [f]
          IF
            p
          THEN [intended function for thenpart]
            g
          ELSE [intended function for elsepart]
            h
          FI


whiledo
          [f]
          WHILE
            p
          DO [intended function for dopart]
            g
          OD


dountil
          [f]
          DO [intended function for dopart]
            g
          UNTIL
            p
          OD


fordo
          [f]
          FOR
            i := L1 to Ln
          DO [intended function for dopart]
            g
          OD
```

case
```
          [f]
          CASE
            p
          PART CL1 [intended function for part 1]
            g1
            .
            .
            .
          PART CLn [intended function for part n]
            gn
          ELSE
            h
          ESAC
```

The functional correctness method relies on strongly typed variables. The type specification is not verified by the proof, but it is applied in the proof. For example a type specification of x: INTEGER >= 0 and a condition of x <= 0 implies x = 0, and this fact may affect the result of the proof. So it is important that the type of each variable be specified. The type specification for a parameter should occur after the parameter in the parameter list, and the type specifications for the local variables should occur at the beginning of a procedure before the function specification. A practical means of including the local variable type specifications in a procedure in the PDL is to place them in a "data procedure" and then indicate their use in a procedure by the keyword USE followed by the data procedure name. An example of a procedure with variable type specifications included is

```
PROC dotdotdot(x,y: INTEGER >= 0,
                 c: A..Z )

  USE othervars

  [f]

   .
   .
   .

CORP

 DATA othervars

   t,u: INTEGER

     b: ARRAY[1..3] OF 0..9

 ATAD
```

Proof Form and Function

The general proof form consists of four parts written in a tabular form [8, 9]. These four parts are

FUNCTION

  statement of or reference to the intended function

PROGRAM

  statement of or reference to the subprogram

PROOF

  proof body

RESULT

  PASS or FAIL

Under result a pass or fail is used to specify the proof outcome. If the data type of a variable influences the outcome of a program, then the type of this variable should be stated under the FUNCTION or PROGRAM section of the proof so that the type is readily perceivable when the proof refers to it.

The proof body under the PROOF section has a specialized form according to the proof objective for each control structure. The proof of each control structure is derived from the Correctness Theorem:

> The Correctness of an Alternation Expression. To prove f = IF p THEN g ELSE h FI it is necessary and sufficient to show, for every $(x,y) \in f$, that either $p(x) = T$ and $y = g(x)$ or $p(x) = F$ and $y = h(x)$.
> The Correctness of a Composition Expression. To prove f = g;h it is necessary and sufficient to show, for every $(x,y) \in f$, that $y = h(g(x))$.
> The Correctness of Iteration Expression. To prove f = WHILE p DO g OD it is necessary and sufficient to show, for every $(x,y) \in f$, that the iteration terminates and that either $p(x) = T$ and $y = f(g(x))$ or $p(x)= F$ and $y = x$ (Mills [11] p. 47).

This is a condensed version of the Correctness Theorem. A more extensive version which includes the fordo, case, and dountil structures can be found in Mills [9].

The derived function of a sequence structure, g;h, is the composition of the functions in the sequence h o g (o represents composition). It is derived through trace tables, a proving technique discussed later in this chapter. The proof body has no specific form beyond the derivation of the program function. The rest of the structures, however, do have a proof form.

In the following descriptions, f represents the intended function. The ifthenelse structure, if p then g else h fi, has the form

```
IFTEST TRUE  (p -> g)

  show f = g
  PASS or FAIL

IFTEST FALSE  (¬p -> h)

  show f = h
  PASS or FAIL
```

An ifthen structure, if p then g fi, is similar to the ifthenelse. The difference is in the IFTEST FALSE section the program function is the identity or I, so f = I is shown. Examples of ifthenelse and ifthen proofs follow [9].

```
FUNCTION

  x := min(a,b)

PROGRAM

  IF
    a < b
  THEN
    x := a
  ELSE
    x := b
  FI

PROOF

  IFTEST TRUE  (a < b)

    f: x := min(a,b)
         := a

    g: x := a

    f = g
    PASS

  IFTEST FALSE  (a >= b)

    f: x := min(a,b)
         := b

    h: x := b

    f = h
    PASS

RESULT

  PASS
```

```
FUNCTION

  y := abs(y)

PROGRAM

  IF
    y < 0
  THEN
    y := -y
  FI

PROOF

  IFTEST TRUE  (y < 0)

    f: y := abs(y)
         := -y

    g: y := -y

    f = g
    PASS

  IFTEST FALSE  (y >= 0)

    f: y := abs(y)
         := y

    f = I
    PASS

RESULT

  PASS
```

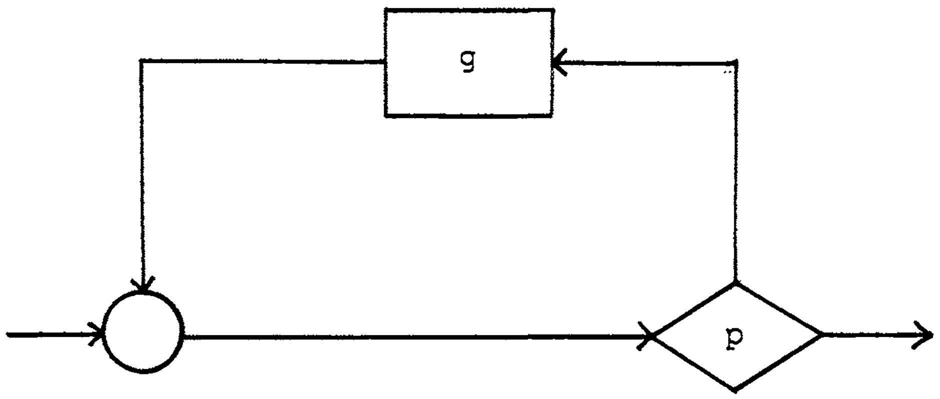
The case structure, case p part (CL1) g1 part (CLn) gn else h esac, has the form

```
PART n   (p ∈ CLn -> gn)

  show f = gn
  PASS or FAIL

ELSEPART  (p ∉ (CL1,...,CLn) -> h)

  show f = h
  PASS or FAIL.
```

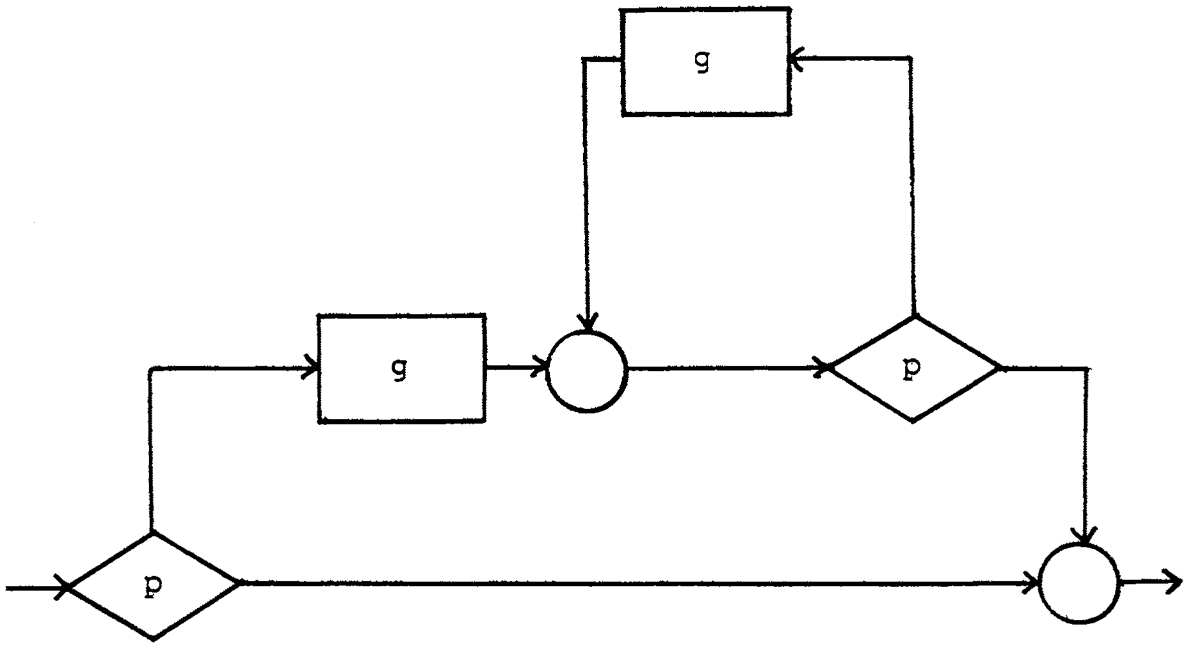
The proof of a whiledo structure, while p do g od, is dependent on the iteration recursion theorem [1, 8, 9] which states f = [while p do h od] if and only if the loop terminates and f = [if p then g;f fi]. Figure 14 illustrates the equivalence of while p do g od and if p then g; while p do g od fi. The iteration recursion theorem is a recursive application of this equivalence. A proof of the theorem is in Mills[9] and a discussion of the theorem is in IBM [8]. Basically there are two steps, first the loop is shown to terminate, then the iterative whiledo loop is converted to an equivalent recursive ifthen structure

```
[f]                                      [f]
WHILE                                    IF
  p          converted to                  p
DO           ------------->              THEN
  g                                        g;f
OD                                       FI

iterative                               recursive
```

Thus, a whiledo proof is actually an ifthen proof. The proof in the IFTEST TRUE is a proof of a sequence structure



while p do g od



if p then g fi ; while p od g od

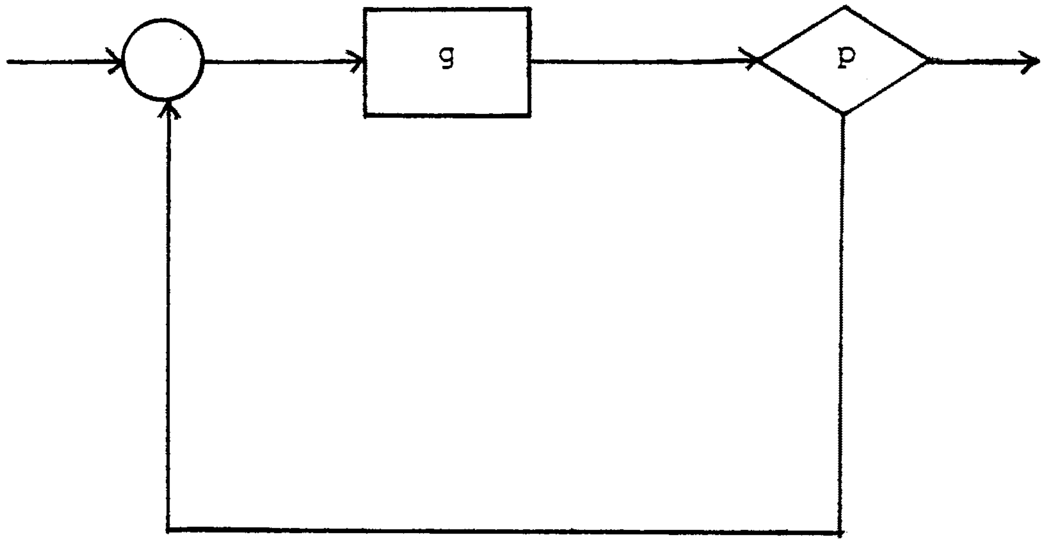
Figure 14. Flowchart Equivalence of a Whiledo and Ifthen ; Whiledo Sequence

with the dopart of the whiledo the first function of the sequence and the specified function the second sequence function. The proof structure is

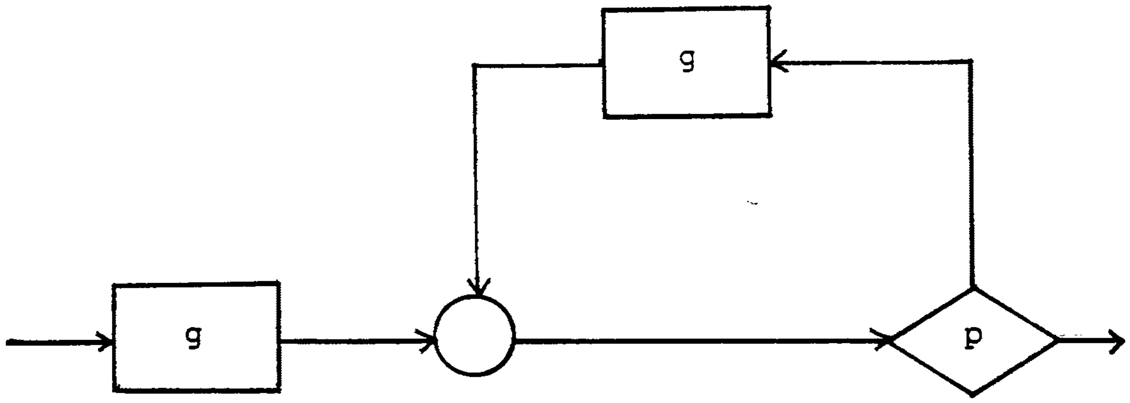
```
TERM

  show the loop terminates
  PASS or FAIL

WHILETEST TRUE  (p -> g;f)

  show f = f o g
  PASS or FAIL

WHILETEST FALSE  (¬p -> I)

  show f = I
  PASS or FAIL
```

where WHILETEST TRUE and WHILETEST FALSE are used instead of IFTEST TRUE and IFTEST FALSE, respectively. An example of a whiledo proof is given in the next section after trace tables are introduced.

A dountil structure, do g until p od, can be verified in two ways [8, 9]. One way is based on the iteration recursion theorem. A description and an example of this way can be found in Mills [9]. The other way is to convert the dountil structure to an equivalent initialized whiledo structure as shown in Figure 15. After the conversion, a combination of a whiledo proof followed by a sequence proof is used to prove the dountil. If one becomes accustomed to the whiledo proof, then this second method of proving a dountil is usually the easier of the two methods because the iterative recursive method of the dountil can be more difficult than the whiledo proof since it deals with the



a.) do g until p od



b.) g ; while ¬p do g od

Figure 15. Flowchart Equivalence of Dountil and Initialized Whiledo

composition of the predicate p and the function g.

The final structure to be considered is the fordo structure, for i := L1 to Ln do g od. There are three possible approaches to prove a fordo, expand the loop into an extended sequence, convert the fordo to an equivalent initialized whiledo, or apply mathematical induction [8, 9]. The first approach is used only if the structure is a simple fordo with a small index list. For example [8]

```
FOR
  i := 1 to 3
DO
  sum := sum + i
OD
```

can easily be expanded to

```
sum := sum + 1
sum := sum + 2
sum := sum + 3.
```

The second approach, converting to an initialized whiledo, is probably the most viable method of the three. This approach is similar to the second method described for proving a dountil. Figure 16 shows a fordo converted to an equivalent initialized whiledo structure.

In the induction method, the induction variable can be either a variable in the fordo list description or the size of the fordo list [9]. This method is the most difficult of the three approaches, and the rigor required for this approach is not usually needed. The mathematical induction method is not used in this report.

Mental Verification and Trace Tables

The program syntax and proof syntax set up the documentation and program function derivation and verification. Mental verification and trace tables are the means of function derivation and verification. Mental verification is used when the program function can be derived and verified by inspection. Usually when mental verification is used, the program segment is very simple or the intended function and program implementation correspond straightforwardly so that deriving the program function is superfluous. The goal of the correctness proof is a conviction that the program is correct, and oftentimes, a mental check is enough to convict one of a segment's correctness. For example

```
DO [x := a * a]
  x := a
  x := x * x
OD.
```

This program is simple and can be easily verified by inspection rather than by the more lengthy method of trace tables.

Trace tables provide a general trace of the data state from the initial data state of a subprogram to the final data state. Then backward substitution is applied to specify the final data state in terms of the initial data state. The subscript of 0 is used to indicate the initial state, and sequentially increasing subscripts are used for subsequent states. For example, the program function of

```
DO
  x := x + y
  y := x - y
  x := x - y
OD
```

is derived through a trace table

state change	x	y
1st	$x_1 = x_0 + y_0$	$y_1 = y_0$
2nd	$x_2 = x_1$	$y_2 = x_1 - y_1$
3rd	$x_3 = x_2 - y_2$	$y_3 = y_2$

Each assignment statement corresponds to a data state change, and if the data state change is the nth change, then the n-1 values affect this change because they are the most recent values of the variables. One other aspect of the trace table is that a data state encompasses all variables of a program, so at each step the value of each variable is indicated even though the values of some of the variables remain the same. For example, at the second data state change the value of x was not altered, and this is indicated by $x_2 = x_1$.

After the table is set up, the program function is derived by backward substitution:

$$\begin{aligned} x_3 &= x_2 - y_2 \\ &= x_1 - (x_1 - y_1) \\ &= (x_0 + y_0) - ((x_0 + y_0) - y_0) \\ &= y_0 \end{aligned}$$

$$\begin{aligned} y_3 &= y_2 \\ &= x_1 - y_1 \\ &= (x_0 + y_0) - y_0 \\ &= x_0 \end{aligned}$$

Now the final value of each variable is expressed in terms of the initial value of each variable which defines the function of the program. In this example the derived function is x,y := y,x [8].

The trace table can be used for any size sequence with any number of variables as in the following example [9].

```
DO
  w,x := x+y, y+z
  y := z+w
  z,w := w+x, y-z
OD
```

w	x	y	z
$w_1 = x_0 + y_0$	$x_1 = y_0 + z_0$	$y_1 = y_0$	$z_1 = z_0$
$w_2 = w_1$	$x_2 = x_1$	$y_2 = z_1 + w_1$	$z_2 = z_1$
$w_3 = y_2 - z_2$	$x_3 = x_2$	$y_3 = y_2$	$z_3 = w_2 + x_2$

derivations:

$w_3 = y_2 - z_2$

$= (z_1 + w_1) - z_1$

$= (z_0 + (x_0 + y_0)) - z_0$

$= x_0 + y_0$

$x_3 = x_2$

$= x_1$

$= y_0 + z_0$

$y_3 = y_2$

$= z_1 + w_1$

$= z_0 + (x_0 + y_0)$

$z_3 = w_2 + x_2$

$= w_1 + x_1$

$= (x_0 + y_0) + (y_0 + z_0)$

$= x_0 + 2y_0 + z_0$

derived function: w,x := x+y, y+z,

y,z := z+x+y, x+2y+z.

Five Examples

The following five examples illustrate various aspects discussed about functional correctness proofs. An example of a whiledo and a fordo proof is given, and also illustrated are proof syntax, trace tables, proof failure, and sufficient correctness.

The first proof is an example of a whiledo proof [9].

```
FUNCTION

  x,y,a := 0, ax + y, a
  VAR x: INTEGER >= 0

PROGRAM

  WHILE
    x > 0
  DO
    x,y := x-1, y+a
  OD

PROOF

  TERM

    x is decremented regularly by 1, so it eventually will
    be less than zero.
    PASS

  WHILETEST TRUE  (x > 0)
```

	x	y	a
dopart	$x_1 = x_0 - 1$	$y_1 = y_0 + a_0$	$a_1 = a_0$
f	$x_2 = 0$	$y_2 = a_1 x_1 + y_1$	$a_2 = a_1$

derivations:

$$y_2 = a_1 x_1 + y_1$$
$$= a_0(x_0 - 1) + (y_0 + a_0)$$
$$= a_0 x_0 - a_0 + y_0 + a_0$$
$$= a_0 x_0 + y_0$$

```
    derived function: x,y,a := 0, ax + y, a
    PASS
```

```
  WHILETEST FALSE  (x <= 0)
    x <= 0 combined with data type x >= 0 implies x = 0
    y = ax + y
      = 0 + y
      = y
    x = 0
      = x
    a = a
    f = I
    PASS
RESULT
  PASS
```

In the trace table in the WHILETEST TRUE section the rows of the trace table are labeled as dopart and f to reinforce the origin of the data state changes represented in these rows.

The next example demonstrates sufficient correctness.

```
FUNCTION

  n < 30 -> n := n + i

PROGRAM

  IF
    n < 30
  THEN
    n := n + i
  FI

PROOF

  IFTEST TRUE  (n < 30)

    f: n := n + i

    g: n := n + i

    f = g
    PASS

  IFTEST FALSE  (n >= 30)

    f is undefined for n >= 30 and the program function maps
    n and i into the identity, thus, it maps additional
    arguments.

    PASS (sufficient)

RESULT

  PASS (sufficient)
```

The next two proofs are examples of a failure in the proof.

```
FUNCTION

  x := min(a,b)

PROGRAM

  IF
    a < b
  THEN
    x := b
  ELSE
    x := a
  FI

PROOF

  IFTEST TRUE  (a < b)

    f: x = min(a,b)
         = a

    g: x = b

    f ¬= g
    FAIL

  IFTEST FALSE  (a >= b)

    f: x = min(a,b)
         = b

    g: x = a

    f ¬= g
    FAIL

RESULT

  FAIL
```

```
FUNCTION

  i <= 4 -> x,i := x+4, 5 | TRUE -> I

PROGRAM

  WHILE
    i <= 4
  DO
    x,i := x+1, i+1
  OD

PROOF

  TERM

    i is incremented regularly , so it eventually will be
    greater than 4.

  WHILETEST TRUE  (i <= 4)
```

	x	i
dopart	$x_1 = x_0 + 1$	$i_1 = i_0 + 1$
f	$x_2 = x_1 + 4$	$i_2 = 5$

derivations:

$$x_2 = x_1 + 4$$
$$= x_0 + 1 + 4$$
$$= x_0 + 5$$

```
    derived function: x,i := x+5, 5

    f ¬= f'
    FAIL

  WHILETEST FALSE (i > 4)

    f = I indicated in the specification of f
    PASS

RESULT

  FAIL
```

The latter example also illustrates two other notions about a whiledo proof. The first is that if a variable is assigned a constant in the intended function, then the mapping derived for that variable under WHILETEST TRUE always passes. The second notion is that if the intended function specifies the identity transformation for the WHILETEST FALSE condition (in this case i > 4), then WHILETEST FALSE always passes.

The fifth example illustrates a fordo proof.

```
                                  [x := x+99]
[x := x+99]                       i := 1
FOR                               [x,i := x+100-i, 101]
  i := 1 to 100    converted to   WHILE
DO                 ------------>    i <= 100
  x := x + 1                      DO
OD                                  x,i := x+1, i+1
                                  OD
```

The proof has two parts: the uninitialized whiledo is proved, and then the sequence of the initialization assignment and the whiledo intended function is proved.

First is the proof of the whiledo.

FUNCTION

```
  x,i := x+101-i, 101
  VAR i: 1 <= INTEGER <= 101  (this type specification
                               evolved from the fordo
                               loop)
```

PROGRAM

The uninitialized whiledo specified earlier

PROOF

TERM

i is incremented regularly , so i eventually will be greater than 100.

WHILETEST TRUE (i <= 100)

	x	i
dopart	$x_1 = x_0 + 1$	$i_1 = i_0 + 1$
f	$x_2 = x_1 + 101 - i_1$	$i_2 = 101$

derivations:

$$x_2 = x_1 + 101 - i_1$$
$$= (x_0 + 1) + 101 - (i_0 + 1)$$
$$= x_0 + 101 - i_0$$

derived function: x,i := x+101-i, 101
PASS

```
  WHILETEST FALSE  (i > 100)

    i > 100 and type specification of i <= 101 implies
    i = 101

    x = x + 101 - i

      = x + 101 - 101

      = x

    f = I
    PASS

RESULT

  PASS
```

Second is the proof of the sequence.

FUNCTION

```
  x := x + 100
```

PROGRAM

```
  DO
    i := 1
    x,i := x+101-i, 101  (intended function of whiledo)
  OD
```

PROOF

```
       x           i
  ---------------------
```

$x_1 = x_0$ $i_1 = 1$

$x_2 = x_1 + 101 - i_1$ $i_2 = 101$

derivations:

$$x_2 = x_1 + 101 - i_1$$
$$= x_0 + 101 - 1$$
$$= x_0 + 100$$

derived function x,i := x+100, 101
PASS

RESULT

PASS

Arrays and Anonymous Data

Arrays, sequences, stacks, queues, and sets are abstract data types used during the designing of a program. Because of their use in the design stage of program writing, these data structures have to be handled in correctness proofs; consequently, techniques for manipulating these structures in a functional correctness proof have been developed.

Array manipulation begins with the notation used to express the various aspects of an array. An array variable is indicated by a succeeding subscript enclosed in brackets like vector[3]. The range of an array is indicated in the type declaration by writing starting-index..ending-index within the brackets, vector[1..10]. In assignment statements all or part of the array may be referenced. If the whole array is being referenced, no subscripting is necessary. Some examples of array assignments are

```
1) word[1..3] := c,o,m
2) word[1,3,5,7] := c,m,u,e
3) word := c,o,m,p,u,t,e,r
4) word := c
```

1) illustrates an assignment where the consecutive elements of an array are referenced, thus ellipses, .., are used to specify the range of elements being referenced. The range indicated in the brackets must match the number of elements on the right hand side of the assignment. Also if the

beginning number of the range is greater than the ending number, then the notational implication is that no assignment is made. In 2) since the elements being referenced are not consecutive, each subscript is written out; however, the variable name does not have to be repeated each time. Once again the number of elements implied on the left hand side must match the number on the right hand side. 3) and 4) illustrate references to the whole array. In 3) the array size is assumed to be equivalent to the number of elements on the right hand side. When there is a scalar on the right hand side as in 4), each indicated element of the array assumes the value of the scalar.

The notation for multidimensional arrays is similar to one dimensional arrays except a semicolon separates the dimensions. For example mat[3;1..6] refers to elements in row 3, columns 1 through 6; mat[8,11;2] refers to elements in column 2, rows 8 and 11. The notation mat[2] refers to the whole second row of mat and mat[;3] refers to the whole third column of mat. This notation can be extended for arrays of dimensions greater than two. The array notation just described was devised by this author for use in this report. It is an aggregate of notation used in IBM [8] and Mills [9] and used by different programming languages, and it should not be considered standard notation.

The difficulties that arise with array data are that only part of the array is altered in most assignments and not only the array but also the index of the array may be affected in a program segment. An example of a proof with arrays is the best explanation of how to handle these two difficulties. (The following proof is extracted from the correctness proofs of the RISC simulator which is the substance of the next chapter. This proof is part of the correctness proof for the shift left procedure).

FUNCTION

```
  i,dest[i..(31-amt)] := 32-amt, source[(amt+i)..31]
  VAR i: INTEGER <= 32-amt  (This type declaration evolves
                              from a fordo loop)
      dest,source: ARRAY[0..31] of 0..1
```

PROGRAM

```
  WHILE
    i <= 31-amt
  DO
    i,dest[i] := i+1, source[amt+i]
  OD
```

PROOF

(in the proof d is used for dest, s for source, and a for amt)

TERM

i is incremented regularly, so it eventually will be greater than 31 - a

WHILETEST TRUE (i <= 31-a)

	i	d
dopart	$i_1 = i_0 + 1$	$d_1[0..i_0-1] = d_0[0..i_0-1]$
		$d_1[i_0+1..31] = d_0[i_0+1..31]$
		$d_1[i_0] = s[a+i_0]$
f	$i_2 = 32-a$	$d_2[0..i_1-1] = d_1[0..i_1-1]$
		$d_2[i_1..(31-a)] = s[(a+i_1)..31]$

```
    derivations:

    $d_2[0..(i_1 -1)] = d_1[0..(i_1 -1)]$

    $d_2[0..(i_0 +1)-1] = d_1[0..(i_0 +1)-1]$

    $d_2[0..i_0 ] = d_1[0..i_0 ]$

    $d_2[0..i_0 -1],d [i_0 ] = d_1[0..i_0 -1],d_1[i_0 ]$
                           $= d_0[0..i_0 -1],s[a+i_0 ]$

    $d_2[i_1 ..(31-a)] = s[(a+i_1 )..31]$

    $d_2[(i_0 +1) ..(31-a)] = s[(a+(i_0 +1))..31]$

    derived function: i,d[i..(31-a)] := 32-a, s[(a+i)..31]
    PASS
  WHILETEST FALSE (i > 31-a)

   i > 31-a combined with type i <= 32-a implies i = 32-a

   d[i..(31-a)] = s[(a+i)..31]

   d[(32-a)..(31-a)] ==> nothing is changed since the low
                         index < high index
   PASS
RESULT

  PASS
```

In the assignments of the dopart in the WHILETEST TRUE section, only one element of the array d is changed, and this is indicated by the first two assignments to d which show that the elements preceding and following element i remain the same. The fact that the element of d that is changed depends on the variable i is controlled in the trace table and in the derivation by careful subscripting of i to indicate the correct data state value at each assignment.

Mills [9] refers to data structures such as sequences, stacks, queues, and sets as anonymous data because members can be accessed without individual item names. Anonymous data manipulation is introduced, but a detailed and comprehensive look at anonymous data handling is not covered in this report because of its magnitude. More thorough explanations and descriptions of the subsequent concepts about anonymous data can be found in IBM [8] and Mills [9].

The structures that are classed as anonymous data are all list structures, and the basic idea behind their manipulation is that special list operations and functions are defined. Some list operations and functions are

operation	word description
||	concatenation
H	head of list
T	tail of list

function	list structure
PUSH	stack
DEQUE	queue
SUM	any list
MEMBER	set

List structures q, r, s, and t are defined as follows:

q = MONTIE (queue)

r = FISH (sequence)

s = ER (sequence)

t = 1,2,3,4 (set),

and are used to illustrate the use of some of the preceding functions and operations as follows:

r || s = FISHER

H(r) = F

T(s) = R

DEQUE(q) = ONTIE (DEQUE removes the first element of a queue)

SUM(t) = 10 (SUM adds up all the values in the list structure. The values in the list must be integers for SUM to be used)

Many more operations and functions for list structures have been defined and are included in the descriptions in IBM [8] and Mills [9].

The following example demonstrates the use of list functions and operations in a proof of a program that contains anonymous data [8].

```
FUNCTION

  total, que := SUM(que) + total, EMPTY
  (EMPTY is a keyword related to list structures signifying
   that there are no elements left in the list)
  VAR que: QUEUE of INTEGER

PROGRAM

  WHILE
    que ¬= EMPTY
  DO
    total := total + DEQUE(que)
  OD

PROOF

  (in the proof t is used for total, q for que)

  TERM

    An element of the queue is removed at each iteration, so
    eventually the queue will be empty.

  WHILETEST TRUE (q ¬= EMPTY)
```

	t	q
dopart	$t_1 = t_0 + H(q_0)$	$q_1 = q_0 - H(q_0)$
f	$t_2 = SUM(q_1) + t_1$	$q_2 = EMPTY$

```
    derivations:

    $t_2 = SUM(q_1) + t_1$

      $= SUM(q_0 - H(q_0)) + (t_0 + H(q_0))$

      $= t_0 + (SUM(q_0 - H(q_0)) + H(q_0))$

      $= t_0 + SUM(q_0)$

    derived function: t,q := t + SUM(q), EMPTY
    PASS

  WHILETEST FALSE (q = EMPTY)

    t = t + SUM(q) but SUM of an empty list is defined
    to be zero so t = t

    PASS

RESULT

  PASS
```

Conditional Rules

The simplest form of a conditional rule is a single condition followed by a single rule. However, conditional rules usually are not this simple. Complexity arises with conditional rule situations such as multiple path conditions, conditional rules nested within conditional rules, or sequences with conditional rules interspersed among unconditional rules. Manipulation of these cases of complex conditional rules is a methodical step by step process in a functional correctness proof.

The steps taken to verify a derived program function with multiple path conditions or nested conditional rules are:

1. reexpress the derived function so that the predicates are disjoint
2. partition the domain of the specified function according to the derived function conditions
3. compare the function rules of the derived function and intended function in each of the partitions.

These steps are enough to insure sufficient correctness. For complete correctness the domains of the two functions must be shown to be equal also [8, 9].

Disjoint conditions or predicates are conditions whose specified partitions do not overlap [8, 9]. If p1 and p2 are disjoint predicates then p1 & p2 = false. To make consecutive conditional rules such as p1 -> g1 | p2-> g2 | p3 -> g3 disjoint, one combines the negation of all preceding predicates with each predicate p1 -> g1 | ¬p1 & p2 -> g2 | ¬p1 & ¬p2 & p3 -> g3. The conditional rule

```
     x > 5 -> x := 1 | x > 3 -> x := 4 | x > 2 -> x := 5
```

can be converted so that each condition is disjoint to

```
     x > 5 -> x := 1 | x <= 5 & x > 3 -> x:= 7 |
     x <= 5 & x <= 3 & x > 2 -> x := 5
```

which is equivalent to

```
     x > 5 -> x := 1 | 3 < x <= 5  -> x:= 7 |
     2 < x <= 3 -> x := 5.
```

When there are nested predicates in a conditional rule, the predicates are converted to disjoint predicates at each level, and then the higher level or outer predicates are distributed into the inner predicates. In illustration, the conditional rule

```
     x > 9 -> (x > 18 -> a := x | x = 18 -> b := x |
               x > 12 -> c := x) | x > 3 -> d := x
```

has two levels. At one level, x > 18, x = 18, x > 12 are converted to the disjoint predicates x > 18, x <= 18 & x = 18, x <= 18 & x ¬= 18 & x > 12 which when simplified become x > 18, x = 18, 12 < x < 18. And, at the other level, x > 9 and x > 3 are converted to the disjoint predicates x > 9, x <= 9 & x > 3 or x > 9, 3 < x <= 9. Finally, the outer predicate x > 9 is distributed through the inner conditional rules

```
     x > 9 & x > 18 -> a := x | x > 9 & x = 18 -> b := x |
     x > 9 & 12 < x < 18 -> c := x | 3 < x <= 9 -> d := x
```

which simplifies to

```
x > 18 -> a := x | x = 18 -> b := x |
12 < x < 18 -> c := x |
3 < x <= 9 -> d := x  [8].
```

Once the derived program function is reexpressed as consecutive disjoint rules

```
p1 -> r1 | p2 -> r2 | p3 -> r3,
```

then the domain of the specified function is partitioned according to the disjoint predicates, and the rules of the derived function and intended function are compared in each partition

```
in p1(X) does r1(X) = f(X)?
in p2(X) does r2(X) = f(X)?
in p3(X) does r3(X) = f(X)?
```

The following proof illustrates the steps for handling the proof of a multiple path program with nested conditions.

```
FUNCTION

  z <= 10 -> y := 2 | x >= 5 -> y := 1 | TRUE -> y := 0

PROGRAM

  z > 10 -> (x < 5 -> y := 0 | x > 1 -> y := 1) |
  z <= 15 -> y := 2

PROOF

  1. make predicates disjoint

     x < 5, x > 1  become x < 5, x >= 5 & x > 1 or

                          x < 5, x >= 5

     z > 10, z <= 15 become z > 10, z <= 10 & z <= 15 or

                            z > 10, z <= 10

  2. distribute the outer condition

     z > 10 & x < 5 -> y := 0 | z > 10 & x >= 5 -> y := 1 |

     z <= 10 -> y :=2

  3. partition domain of f and compare rules

     when z > 10 & x < 5 does f = (y := 0)?     yes

     when z > 10 & x >= 5 does f = (y := 1)?    yes

     when z <= 10 does f = (y := 2)?            yes

RESULT

  PASS
```

From multiple path programs the complexity increases to sequences of conditional and unconditional rules. A case structured approach is applied to handle this occurrence. The conditions are recorded in the trace table along with the data state changes, and backward substitution is used to derive the condition as well as the rule for each case.

The cases are created by determining the possible paths of the program and labeling each case in terms of T(rue) and F(alse) according to the value of each predicate along the path. In the case of

```
IF
  x < 0
THEN
  y := 3
  IF
    x >= -4
  THEN
    x := 2
  ELSE
    z := -2
  FI
ELSE
  y,z := 0,0
FI
```

the possible paths are TT, TF, and F. So there are three cases to handle for this program .

Once the possible paths are determined, the function for each path is derived as a conditional rule in terms of the initial state of the variables. Both the rule and the condition are derived for each case. The following sequence program illustrates the case structured approach [8].

```
IF
  x < 0
THEN
  x := x + y
ELSE
  y := x + y
FI
x,y := y, x + y
IF
  y < 0
THEN
  x := x - y
ELSE
  y := x - y
FI
```

The possible paths are TT, TF, FT, FF. The derivation of the TT path proceeds as follows:

condition	x	y
$x_0 < 0$	$x_1 = x_0 + y_0$	$y_1 = y_0$
	$x_2 = y_1$	$y_2 = x_1 + y_1$
$y_2 < 0$	$x_3 = x_2 - y_2$	$y_3 = y_2$

derivations:

condition $x_0 < 0 \ \& \ y_2 < 0$

$x_0 < 0 \ \& \ x_1 + y_1 < 0$

$x_0 < 0 \ \& \ x_0 + y_0 + y_0 < 0$

$x_0 < 0 \ \& \ x_0 + 2y_0 < 0$

rule $x_3 = x_2 - y_2$ $\qquad y_3 = y_2$

$= y_1 - (x_1 + y_1)$ $\qquad = x_1 + y_1$

$= -(x_0 + y_0)$ $\qquad = (x_0 + y_0) + y_0$

$\qquad = x_0 + 2y_0$

derived function for TT case:

P1 = $x < 0 \ \& \ x + 2y < 0 \to x,y := -x - y, x + 2y$.

It should be noted that under the condition column, y is used in the condition rather than y_0 or y_1 because by the time the condition y < 0 has an influence on the program, there already have been two data state changes. The data state at the time a condition becomes active is the state used in the condition in the trace table, otherwise, the derived condition will be incorrect.

The derived functions for the other three paths are

TF: P2 = x < 0 & x+2y >= 0 -> x,y := y, -x-y

FT: P3 = x >= 0 & 2x+y < 0 -> x,y := -x, 2x+y

FF: P4 = x >= 0 & 2x+y >= 0 -> x,y := x+y, -x.

The final derived program function is

[P] = P1 | P2 | P3 | P4.

The verification step is now the same as the verification step for the multiple path case. The domain of the intended function f is partitioned according to the conditions of the derived function and the rules in each partition are compared. That is,

does f = (x,y := -x-y, x+2y) for x < 0 & x+2y < 0?

does f = (x,y := y, -x-y) for x < 0 & x+2y >= 0?

does f = (x,y := -x, 2x+y) for x >= 0 & 2x+y < 0?

does f = (x,y := x+y, -x) for x >= 0 & 2x+y >= 0?

The case structured approach is used not only for ifthen and ifthenelse structures but also for whiledo structures when the intended function is expressed as a conditional rule. The following whiledo is an example where the case structure would be applied [8]:

```
[x>y -> x,y := x-y, 0 | TRUE -> x,y := o, y-x]
WHILE
  x > 0 & y > 0
DO
  x,y := x-1, y-1
OD.
```

In the WHILETEST TRUE segment of the proof the sequence used for deriving the program function is

```
(dopart)    x,y := x-1, y-1
(f)         x>y -> x,y := x-y, 0 |
            TRUE -> x,y := 0, y-x
```

which is a sequence with both an uncondnditonal and conditional rule.

When the case structure is used in a whiledo function derivation, it is again important to be careful about the data state used in the condition in the table trace. In the WHILETEST TRUE segment there is one data state change caused by the "dopart" before the condition in the "f part" is considered. Mathematically stated, p -> f(g(X)) = f(X) is to be verified. The conditions in f are based on g(X) not X.

Summary

Program and proof syntax provide documentation and set up the framework within which a program may be proved to be correct. Specific proof forms and objectives exist for each of the control structures that are fundamental to structured programming.

With the framework and proof objectives established proofing techniques such as mental verification, table tracing, and methods of array and anonymous data handling and conditional rule manipulation can be applied to derive and verify the program function.

CHAPTER V

RISC SIMULATOR AND FUNCTIONAL CORRECTNESS

The functional correctness method has demonstrated its practicability on simple programs devised to illustrate specific aspects of this proof approach. Now the functional correctness method is applied to a functioning program that already has been coded and tested. Applying the functional proof of correctness to a working program demonstrates the performance of the method on an example not specially designed to support the method, and provides the method a chance to locate errors in the program not discovered during the testing of the program.

EXEC Module of RISC Simulator

The program to which functional correctness proofs are applied is a program that simulates the Reduced Instruction Set Computer described in chapter two. The simulator was based on Patterson [15]; however, since the article incompletely describes various details of RISC, many assumptions and additions had to be included in the simulator. The module most closely associated with RISC is the EXEC module which contains procedures that simulate the execution of the RISC instruction set. These are the procedures which are used for the correctness proofs.

In Patterson [15] twenty-six instructions are described. Eleven main procedures were written to simulate these twenty-six instructions. These procedures include arithmetic, and_vals, or_vals, xor_vals, shifts, loads, stores, jumps, calls, ret, and get_ops. Besides the eleven instruction procedures, six other procedures critical to the correct performance of the instruction procedures are included in the EXEC module. These procedures are the conversion procedures, bin_dec, dec_bin; the exception procedures, prot_excp, bndry_excp, addr_excp; and the memory access procedure, mac. Thus, the EXEC module contains seventeen main procedures. The functional proof of correctness method is applied to sixteen of them. Get_ops is not used in this report because the simulation of the GTIN and GTLPC instructions is based mainly on assumptions.

Principle Assumptions

Before the proofs of some of the EXEC procedures are presented the principal assumptions that affect the operation of the procedures in the EXEC module are described. Because of the ease of implementation and modification, the RISC simulator is table driven even though there is no indication that tables are used in the actual RISC. Figures 17 and 18 show the navigation matrix and the operation table which are used to direct the operations of the instruction procedures. The operation codes in the operation table were arbitrarily assigned for the simulator because the RISC description does not give the op codes. Another aspect of RISC not described in Patterson [15] is the program status word (PSW). For the simulator the PSW is 64 bits long. Figure 19 shows the break down of the PSW used in the simulator. Finally, 144 registers are used in the simulator register bank rather than 138 which is the number of registers indicated in the RISC descriptions [13, 14, 15]. The reason for this is that with 138 registers the window in the last procedure call before window overflow has only 16 registers instead of the usual 22; whereas with 144 registers all the windows have 22 registers.

Illustrative Proofs of EXEC Procedures

Complete proofs of three of the EXEC procedures, and_vals, bin_dec, and stores, and a proof of a subprogram of the calls procedure are presented in this chapter. Highlights of the proofs of the other EXEC procedures are presented in the appendix. The proof of the and_vals procedure is presented first.

INDEX	OPERATION	N [0]	N [1]	N [2]
0	illop	110	-	-
1	ADD	10	0	0
2	ADDC	10	0	1
3	SUB	10	1	0
4	SUBC	10	1	1
5	AND	20	0	-
6	OR	20	1	-
7	XOR	20	2	-
8	SLA	30	1	0
9	SRA	30	1	1
10	SLL	30	0	0
11	SRL	30	0	1
12	LDL	40	1	4
13	LDSU	40	1	3
14	LDSS	40	1	2
15	LDBU	40	1	1
16	LDBS	40	1	0
17	STL	40	0	0
18	STS	40	0	1
19	STB	40	0	1
20	JMP	50	1	-
21	JMPR	50	1	-
22	CALL	50	2	0
23	CALLR	50	2	1
24	RET	50	3	-
25	GTLPC	60	0	-
26	GTIN	60	1	-
*33	SUBR	10	2	0
34	SUBRC	10	2	1

*Indices 27 - 32 were used for special pseudo operations included in the simulator but not a part of RISC; consequently they are not included in the navigation matrix here.

Figure 17. Navigation Matrix for RISC Simulator

First Four Bits of Op Code

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Last	1	16	17	18	19	20	21	22	23	24	25	26	0	0	0	0	0
Three	2	0	33	34	0	0	0	0	0	0	0	0	0	0	0	0	0
Bits	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
of	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
op	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Code	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 18. Operation Table for RISC Simulator

PSW Bits	Description
0-2	current window pointer
3	window overflow
4	stack overflow
5-7	unused
8-15	exceptions
16	unused
17-20	condition code
21-31	unused
32-63	location counter

Exception Bits

8	Protection
9	address
10	data
11	Illegal op code
12	illegal return
13	boundary
14	integer overflow
15	unused

Condition Code Bits

17	negative
18	zero
19	overflow
20	carry

Figure 19. PSW for RISC Simulator

```
PROC and_vals(op1, op2, result: ARRAY[0..31] OF 0..1)
  i: 0 <= INTEGER <= 32
  [result := op1 & op2]
1  FOR
2    i := 0 to 31
3  DO [result[i] := op1[i] & op2[i]]
4    IF
5      (op1[i] = op2[i]) and (op1[i] = 1)
6    THEN [result[i] := 1]
7      result[i] := 1
8    ELSE [result[i] := 0]
9      result[i] := 0
10   FI
11 OD
CORP
```

The function specifications on the THEN and ELSE parts may be considered excessive and can be omitted.

The proof of this procedure consists of proving two subprograms. The first subprogram is the ifthenelse structure in lines 4-10 and the second subprogram is the fordo structure in lines 1-11. Mental verification could be used on this procedure because of its simplicity; however, a formal proof is given for illustration purposes.

```
FUNCTION

  result[i] := op1[i] & op2[i]
  VAR result[i],op1[i],op2[i]: 0..1

PROGRAM

  lines 4-10 of PROC and_vals

PROOF

  IFTEST TRUE  (op1[i] = op2[i] & op1[i] = 1)

    f: result[i] := op1[i] & op2[i]
                 := 1 & 1
                 := 1

    g: result[i] := 1

    f = g
    PASS

  IFTEST FALSE  (op1[i] ¬= op2[i] v op1[i] ¬= 1)

    f: result[i] := op1[i] & op2[i]
       op1[i] ¬= op2[i] and data type 0..1 implies 0 & 1
         which = 0
       op1[i] ¬= 1  and data type 0..1 implies op1[i] = 0,
         thus, giving 0 & 0 or 0 & 1, both = 0
       so result[i] := 0

    h: result[i] := 0

    f = h
    PASS

RESULT
  PASS
```

```
FUNCTION

  result := op1 & op2
  VAR result, op1, op2: ARRAY[0..31] OF 0..1

PROGRAM

  lines 1-10 of PROC and_vals

PROOF

  The fordo loop can be  expanded  into  a  sequence  of  32
  statements as follows:

    result[0] := op1[0] & op2[0]
    result[1] := op1[1] & op2[1]
       .            .        .
       .            .        .
       .            .        .
    result[31] := op1[31] & op2[31]

  derived function
    result[0..31] := op1[0..31] & op2[0..31]

RESULT
  PASS
```

The proofs for the or_vals and xor_vals procedures are very similar to the and_vals proof. The differences are the operations, or and xor operations instead of the and operation, and the condition on the if tests in the procedures.

The next proof is the proof of one of the conversion procedures, bin_dec. Special functions are employed in the specification and the proving of this procedure. This use of special functions is similar to the use of special functions and operations in anonymous data proofs. The special functions used pertain specifically to the data structures used in this procedure. Their definitions are

DEC(x,i,j) = decimal value of the binary array x from left index i to right index j, if i > j, then DEC(x,i,j) = 0

SUM(f(a),b,c) = sum from a = b to a = c of f(a)

```
PROC bin_dec(length: INTEGER <= 32,
             binval: ARRAY[0..31] OF 0..1,
             decval: 0 <= INTEGER <= 2**31 - 1)
   USE bindeclocs
   [length >= 32 -> length := 31;
    decval := DEC(binval, 32-length, 31)]
   [length >= 32 -> length := 31]
1  IF
2    length >= 32
3  THEN
4    length := 31
5  FI
   [decval := DEC(binval,32-length,1)]
6  FOR
7    i := (32-length) TO 31
8  DO [i <= 31 -> exponent,decval :=
                   0,decval + SUM(binval[j]*2**(31-j),i,31)
        | TRUE -> I]
9    exponent := 31 - i
10   decval := decval + (binval[i]*2**exponent)
11 OD
CORP

DATA bindeclocs
  i: INTEGER >= 0
  exponent: 0 <= INTEGER <= 31
ATAD
```

The proof of this procedure consists of proofs of three subprograms, the ifthen in lines 1-5, the fordo in lines 6-11, and the sequence that results from the stepwise abstraction step of replacing the ifthen and fordo with their specified functions once they have been verified. The ifthen and the sequence can be verified by inspection because both of these structures correspond directly with their intended functions. Thus, the main part of the proof of this procedure lies in the proof of the fordo subprogram.

The abbreviations ex, dc, bv are used for exponent, decval, and binval, respectively, in the following proofs.

FUNCTION

```
  i <= 31 -> i,ex,dc := 32,0,SUM(bv[j]*2**(31-j),i,31) + dc
  | TRUE -> I
  VAR ex: 0 <= INTEGER <= 31
```

PROGRAM

  uninitialized whiledo loop that is partially equivalent to the fordo loop in lines 1-5 of PROC bin_dec

```
  WHILE
    i <= 31
  DO
    ex := 31 - i;
    i, dc := i+1, dc + (bv[i]*2**ex)
  OD
```

PROOF

  TERM

    i is incremented regularly , so eventually i will be greater than 31.

  WHILETEST TRUE (i <= 31)

	i	ex	dc
dopart	$i_1 = i_0$	$ex_1 = 31-i_0$	$dc_1 = dc_0$
	$i_2 = i_1 +1$	$ex_2 = ex_1$	$dc_2 = dc_1 + (bv[i_1]*2**ex_1)$
f	$i_3 = 32$	$ex_3 = 0$	$dc_3 = dc_2 +$ SUM(bv[j]*2**(31-j),i_2,31)

```
    derivations:
      $dc_3 = dc_2 + SUM(bv[j]*2**(31-j), i_2, 31)$
            $= (dc_1 + bv[i_1]*2**ex_1) +$
              $SUM(bv[j]*2**(31-j), i_1+1, 31)$
            $= dc_0 + bv[i_0]*2**(31-i_0) +$
              $SUM(bv[j]*2**(31-j), i_0+1, 31)$
            $= dc_0 + SUM(bv[j]*2**(31-j), i_0, 31)$

      derived function
        i,ex,dc := 32, 0, dc + SUM(bv[j]*2**(31-j),i,31)
      PASS
  WHILETEST FALSE (i > 31)
    f = I is specified in the intended function
    PASS
RESULT
  PASS
```

FUNCTION

```
ex,dc := 0, DEC(bv,32-length,31)
VAR bv: ARRAY[0..31] OF 0..1
    length: INTEGER <= 31
```

PROGRAM

initialized whiledo loop that is equivalent to the fordo loop in lines 6-11 of PROC bin_dec with the program function inserted for the uninitialized whiledo loop part

```
dc, i := 0, 32-length
i,ex,dc := 32,0,SUM(bv[j]*2**(31-j),i,31) + dc)
```

PROOF

i	dc
i_1 = 32-length	dc_1 = 0
i_2 = 32	dc_2 = SUM(bv[j]*2**(31-j),i_1,31) + dc_1

derivation:

dc_2 = SUM(bv[j]*2**(31-j),i_1,31) + dc_1

= SUM(bv[j]*2**(31-j),32-length,31) + 0

= DEC(bv,32-length,31)

The step from line 2 to line 3 in the above derivation is a result of combining the definition of the SUM function, DEC function, and the data type of bv.

Note: if length <= 0, then 32-length > 31 and the SUM function does nothing since the indicated lower bound < indicated upper bound

derived function:
ex,dc := 0, DEC(bv,32-length,31)

RESULT
PASS

The next proof is the proof of the stores procedure. There are three store instructions, STL, STS, STB (see Table 1) that are handled in the stores procedure. The differentiation of the three instructions is accomplished through use of the navigation matrix (n[2]). Also, throughout the proof when a row of the memory matrix is referenced, word addr is used instead of the parameter addr. The reason for this is that the parameter addr is a byte address which is necessary since STS and STB do not access full words of memory; however, in the simulator memory is word addressable only, thus the byte address, addr, is adjusted for fullword memory access, and the use of word addr indicates this adjustment.

```
PROC stores(addr: 0 <= INTEGER <= memsize-1,
            dest: 0 <= INTEGER <= 143,
            n: ARRAY[0..2] OF INTEGER >= -1,
            r: ARRAY[0..143] OF ARRAY[0..31] OF 0..1,
            m: ARRAY[144..memsize-1] OF
                                ARRAY[0..31] OF 0..1,
            psw: ARRAY[0..63] OF 0..1)
  USE stores_locs
  [ n[2] = 0 -> (addr MOD 4 ¬= 0 -> psw[13] := 1 |
                 TRUE -> m[word addr] := r[dest])
   |n[2] = 1 -> (addr MOD 2 ¬= 0 -> psw[13] := 1 |
                 TRUE ->
                  (addr MOD 4 = 0 ->
                    m[word addr;0..15] := r[dest;16..31] |
                   TRUE ->
                    m[word addr;16..31] := r[dest;16..31]))
   |n[2] = 2 -> ( addr mod 4 = 0 ->
                   m[word addr;0..7] := r[dest;24..31]
                 |addr mod 4 = 1 ->
                   m[word addr;8..15] := r[dest;24..31]
                 |addr mod 4 = 2 ->
                   m[word addr;16..23] := r[dest;24..31]
                 |TRUE ->
                    m[word addr;24..31] := r[dest;24..31])
   |TRUE -> I]
   [n[2] ¬= 0 -> temp := m[word addr],
                 start_bit := (addr MOD 4 = 0 -> 0 |
                               addr MOD 4 = 1 -> 8 |
                               addr MOD 4 = 2 -> 16 |
                               addr MOD 4 = 3 -> 24 )]
1  IF
2    n[2] ¬= 0
3  THEN [temp := m[word addr],
           start_bit := (addr MOD 4 = 0 -> 0 |
                         addr MOD 4 = 1 -> 8 |
                         addr MOD 4 = 2 -> 16 |
                         addr MOD 4 = 3 -> 24 )]
4    mac(addr,1,temp)
5    start_bit := (addr MOD 4) * 8
6  FI


   [ n[2] = 0 -> (addr MOD 4 ¬= 0 -> psw[13] := 1 |
                  TRUE -> temp := r[dest])
    |n[2] = 1 -> (addr MOD 2 ¬= 0 -> psw[13] := 1 |
                  TRUE -> temp[start_bit..(start_bit+15)]
                                     := r[dest;16..31])
    |n[2] = 2 -> (temp[start_bit..(start_bit+7)]
                              := r[dest;24..31])
    |TRUE -> I]
7  CASE
8    n[2]
9  PART (n[2] = 0) [addr MOD 4 ¬= 0 -> psw[13] := 1 |
                 TRUE -> temp := r[dest]]
  IF
    addr MOD 4 ¬= 0
  THEN
    psw[13] := 1
  ELSE
    temp := r[dest]
  FI
PART (n[2] = 1) [addr MOD 2 ¬= 0 -> psw[13] := 1 |
                 TRUE -> temp[start_bit..(start_bit+15]
                                   := r[dest;16..31]]
  IF
    addr MOD 2 ¬= 0
  THEN
    psw[13] := 1
  ELSE
    temp[start_bit..(start_bit+15)] := r[dest;16..31]
  FI
PART (n[2] = 2) [temp[start_bit..(start_bit+7)]
                               := r[dest;16..31]]
  temp[start_bit..(start_bit+7)] := r[dest;16..31]
ESAC
mac(addr,0,temp)
CORP
```

```
DATA stores_locs
  start_bit: INTEGER >= 0
  temp: ARRAY[0..31] OF 0..1
ATAD
```

Memsize which is used in the parameter type specifications is a constant not a variable. Using memsize is equivalent to using an integer such as 512.

Stepwise abstraction and program self-containment are two concepts of the functional correctness method that are well illustrated in the proof of the stores procedure. There are two major subprograms, an ifthen in lines 1-6 and a case in lines 7-27, that make up the procedure. Within the case subprogram, there are two ifthenelse subprograms. The proof of this procedure has three levels of abstraction. First the ifthenelse subprograms in the case structure are verified, then their intended functions are inserted in the case subprogram, and the case subprogram is verified. The ifthen subprogram is verified also at this level of abstraction. The final level contains a sequence structure consisting of the intended function of the ifthen, the intended function of the case, and the mac subroutine call in line 28. At each level of abstraction the proof can be affected only by a change in the specified functions of the lower level subprograms because it is the specified function which is used in the proof at a higher level.

Two other instances of self-containment illustrated are the treatment of global variables and the treatment of procedure calls. The use of global variables in a program implies an interdependence among procedures which is detrimental to the functional correctness method because procedure independence is a significant factor of the technique. The way global variables are handled is they are treated like parameters. This promotes self-containment and diminishes the sense of external dependence. Dest, n, r, m, and psw actually are all global variables, but they are listed in the parameter list for the stores procedure. The verification of the accurate syntactic use of the global variables is not part of proof of correctness.

Procedures called from within a procedure are treated like subprograms. The intended function of the called procedure is inserted in the main procedure so that the calling procedure can be verified. The verification of the called procedure's function occurs at a different level of abstraction. Once again the proof of the calling procedure is affected by the called procedure only if the specified function of the lower level procedure is changed; otherwise, the implementation of the called procedure is transparent and immaterial at this level. This is illustrated in the stores procedure by the mac procedure call.

In the following proofs for the stores procedure the abbreviations a, d, t, sb are used for addr, dest, temp, and start_bit, respectively.

```
FUNCTION

  a MOD 4 ¬= 0 -> psw[13] := 1 | TRUE -> t := r[d]

PROGRAM

  lines 10-16 of PROC stores

  IF
    a MOD 4 ¬= 0
  THEN
    psw[13] := 1
  ELSE
    t := r[d]
  FI

PROOF

  mental verification

RESULT
  PASS
```

```
FUNCTION

  a MOD 2 ¬= 0 -> psw[13] := 1 |
  TRUE -> t[sb..(sb+15)] := r[d;16..31]

PROGRAM

  lines 18-24 of PROC stores

  IF
    a MOD 2 ¬= 0
  THEN
    psw[13] := 1
  ELSE
    t[sb..(sb+15)] := r[d;16..31]
  FI

PROOF

  mental verification

RESULT
  PASS
```

```
FUNCTION

    n[2] = 0 -> (a MOD 4 ¬= 0 -> psw[13] := 1 |
                 TRUE -> t := r[d])
  | n[2] = 1 -> (a MOD 2 ¬= 0 -> psw[13] := 1 |
                 TRUE -> t[sb..(sb+15)] := r[d;16..31])
  | n[2] = 2 -> (t[sb..(sb+7)] := r[d;24..31])
   |TRUE -> I

PROGRAM

  lines 7-27 of PROC stores with intended functions replacing
  subprograms

  CASE
    n[2]
  PART (n[2] = 0)
    a MOD 4 ¬= 0 -> psw[13] := 1 |
    TRUE -> t := r[d]
  PART (n[2] = 1)
    a MOD 2 ¬= 0 -> psw[13] := 1 |
    TRUE -> t[sb..(sb+15)] := r[d;16..31]
  PART (n[2] = 2)
    t[sb..(sb+7)] := r[d;24..31]
  ESAC

PROOF

  mental verification

RESULT
  PASS
```

FUNCTION

```
  n[2] ¬= 0 -> t := m[word addr],
               sb := (a MOD 4 = 0 -> 0  |
                      a MOD 4 = 1 -> 8  |
                      a MOD 4 = 2 -> 16 |
                      a MOD 4 = 3 -> 24 )
```

PROGRAM

```
  lines 1-6 of PROC stores with intended functions replacing
  subprograms

  IF
    n[2] ¬= 0
  THEN
    2nd parameter of mac = 0 -> m[word addr] := 3rd parameter
    | TRUE -> 3rd parameter := m[word addr];
    sb := (a MOD 4) * 8
  FI
```

PROOF

  mental verification

RESULT
  PASS

```
FUNCTION

    n[2] = 0 -> (a MOD 4 ¬= 0 -> psw[13] := 1 |
                 TRUE -> m[a] := r[d])
  | n[2] = 1 -> (a MOD 2 ¬= 0 -> psw[13] := 1 |
                 TRUE ->
                   (a MOD 4 = 0 ->
                      m[word addr;0..15] := r[d;16..31] |
                    TRUE ->
                      m[word addr;16..31] := r[d;16..31]))
  | n[2] = 2 -> (  a mod 4 = 0 ->
                     m[word addr;0..7] := r[d;24..31]
                 | a mod 4 = 1 ->
                     m[word addr;8..15] := r[d;24..31]
                 | a mod 4 = 2 ->
                     m[word addr;16..23] := r[d;24..31]
                 | TRUE ->
                     m[word addr;24..31] := r[d;24..31])
   | TRUE -> I (sb and t are not part of the domain)

  VAR a: INTEGER

PROGRAM

  lines 1-28 of PROC stores with intended functions
  replacing subprograms

  n[2] ¬= 0 -> t := m[word addr],
               sb := (a MOD 4 = 0 -> 0 |
                      a MOD 4 = 1 -> 8 |
                      a MOD 4 = 2 -> 16 |
                      a MOD 4 = 3 -> 24 )
    n[2] = 0 -> (a MOD 4 ¬= 0 -> psw[13] := 1 |
                 TRUE -> t := r[d])
  | n[2] = 1 -> (a MOD 2 ¬= 0 -> psw[13] := 1 |
                 TRUE -> t[sb..(sb+15)] := r[d;16..31])
  | n[2] = 2 -> (t[sb..(sb+7)] := r[d;24..31])
   |TRUE -> I;
  2nd parameter = 0 -> m[word addr] := 3rd parameter |
  TRUE -> 3rd parameter m[word addr];
```

The last condition is the program function for the mac procedure which is called at line 28. Since it is verified easily that the second parameter of the mac call is a 0, the last condition can be replaced with the single assignment statement m[word addr] := t.

```
VAR t: ARRAY[0..31] OF 0..1
```

PROOF

The following two pages contain the trace table for the proof of this procedure. Even though it appears as if there are two tables, actually it is one table divided into two sections because of the size of the table. The rows are numbered to connect the two sections. [13] is used for psw[13] in the table and derivations. Also data state changes are not shown for a, r, d, and word addr because their values do not change.

	condition	psw[13]	sb
1.	$n_0[2] = 0$ & a MOD 4 ¬= 0	$[13]_1 = 1$	$sb_1 = sb_0$
2.	$n_0[2] = 0$ & a MOD 4 = 0	$[13]_1 = [13]_0$	$sb_1 = sb_0$
3.	$n_0[2] = 1$ & a MOD 4 = 0	$[13]_1 = [13]_0$ $[13]_2 = [13]_1$	$sb_1 = 0$ $sb_2 = sb_1$
4.	$n_0[2] = 1$ & a MOD 4 = 1	$[13]_1 = [13]_0$ $[13]_2 = 1$	$sb_1 = 8$ $sb_2 = sb_1$
5.	$n_0[2] = 1$ & a MOD 4 = 2	$[13]_1 = [13]_0$ $[13]_2 = [13]_1$	$sb_1 = 16$ $sb_2 = sb_1$
6.	$n_0[2] = 1$ & a MOD 4 = 3	$[13]_1 = [13]_0$ $[13]_2 = 1$	$sb_1 = 24$ $sb_2 = 1$
7.	$n_0[2] = 2$ & a MOD 4 = 0	$[13]_1 = [13]_0$ $[13]_2 = [13]_1$	$sb_1 = 0$ $sb_2 = sb_1$
8.	$n_0[2] = 2$ & a MOD 4 = 1	$[13]_1 = [13]_0$ $[13]_2 = [13]_1$	$sb_1 = 8$ $sb_2 = sb_1$
9.	$n_0[2] = 2$ & a MOD 4 = 2	$[13]_1 = [13]_0$ $[13]_2 = [13]_1$	$sb_1 = 16$ $sb_2 = sb_1$
10.	$n_0[2] = 2$ & a MOD 4 = 3	$[13]_1 = [13]_0$ $[13]_2 = [13]_1$	$sb_1 = 24$ $sb_2 = sb_1$
11.	($n_0[2]$ ¬= 0 & 1 & 2) & (a MOD 4 = 0 | 1 | 2 | 3)	$[13]_1 = [13]_0$	sb_1 = 0 | 8 16 | 24
12.		$[13]_3 = [13]_2$	$sb_3 = sb_2$

	t	m[word addr]
1.	$t_1 = t_0$	$m_1 = m_0$
2.	$t_1 = r_0[d_0]$	$m_1 = m_0$
3.	$t_1 = m_0$[word addr]	$m_1 = m_0$
	$t_2[0..(sb_1 -1)] = t_1[0..(sb_1 -1)]$	$m_2 = m_1$
	$t_2[sb_1..(sb_1 +15)] = r_1[d_1;16..31]$	
	$t_2[(sb_1 +16)..31] = t_1[(sb_1 +16)..31]$	
4.	$t_1 = m$ [word addr]	$m_1 = m_0$
	$t_2 = t$	$m_2 = m_1$
5.	$t_1 = m_0$[word addr]	$m_1 = m_0$
	$t_2[0..(sb_1 -1)] = t_1[0..(sb_1 -1)]$	$m_2 = m_1$
	$t_2[sb_1..(sb_1 +15)] = r_1[d_1;16..31]$	
	$t_2[(sb_1 +16)..31] = t_1[(sb_1 +16)..31]$	
6.	$t_1 = m_0$[word addr]	$m_1 = m_0$
	$t_2 = t_1$	$m_2 = m_1$
7.	$t_1 = m_0$[word addr]	$m_1 = m_0$
	$t_2[0..(sb_1 -1)] = t_1[0..(sb_1 -1)]$	$m_2 = m_1$
	$t_2[sb_1..(sb_1 +7)] = r_1[d_1;24..31]$	
	$t_2[(sb_1 +7)..31] = t_1[(sb_1 +24)..31]$	
8.	$t_1 = m_0$[word addr]	$m_1 = m_0$
	$t_2[0..(sb_1 -1)] = t_1[0..(sb_1 -1)]$	$m_2 = m_1$
	$t_2[sb_1..(sb_1 +7)] = r_0[d_0;24..31]$	
	$t_2[(sb_1 +7)..31] = t_1[(sb_1 +24)..31]$	
9.	$t_1 = m_0$[word addr]	$m_1 = m_0$
	$t_2[0..(sb_1 -1)] = t_1[0..(sb_1 -1)]$	$m_2 = m_1$
	$t_2[sb_1..(sb_1 +7)] = r_1[d_1;24..31]$	
	$t_2[(sb_1 +7)..31] = t_1[(sb_1 +24)..31]$	
10.	$t_1 = m_0$[word addr]	$m_1 = m_0$
	$t_2[0..(sb_1 -1)] = t_1[0..(sb_1 -1)]$	$m_2 = m_1$
	$t_2[sb_1..(sb_1 +7)] = r_1[d_1;24..31]$	
	$t_2[(sb_1 +7)..31] = t_1[(sb_1 +24)..31]$	
11.	$t_1 = m_0$[word addr]	$m_1 = m_0$
12.	$t_3 = t_2$	m_3[word addr] $= t_2$

(the last data state change in row 12 actually follows each of the above conditional changes, and for 1, 2, and 11, it is the second data state change not the third as indicated) derivations:

Since the derivations for several of the cases are similar and lengthy, only two representative derivations are given.

1.condition:

$n_0[2] = 1$ & a MOD $4 = 2$

rule:

$[13]_3 = [13]_2 \qquad sb_3 = sb_2$

$\quad = [13]_1 \qquad\quad = sb_1$

$\quad = [13]_0 \qquad\quad = 16$

$t_3[0..(sb_2-1)] = t_2[0..(sb_2-1)]$

$t_3[0..(sb_1-1)] = t_1[0..(sb_1-1)]$

$t_3[0..15] = t_1[0..15]$

$= m_0[\text{word addr};0..15]$

$t_3[(sb_2+16)..31] = t_2[(sb_2+16)..31]$

$t_3[(sb_1+16)..31] = t_1[(sb_1+16)..31]$

$t_3[32..31] = t_1[32..31]$

lower bound > upper bound implies no change

$t_3[sb_2..(sb_2+15)] = t_2[sb_2..(sb_2+15)]$

$t_3[sb_1..(sb_1+15)] = t_2[sb_1..(sb_1+15)]$

$t_3[16..31] = r_1[d_1;16..31]$

$= r_0[d_0;16..31]$

m_3[word addr] = t_2

m_3[word addr;0..15], m_3[word addr;16..31] =

t_2[0..15], t_2[16..31]

m_3[word addr;0..15], m_3[word addr;16..31] =

t_1[0..15], r_1[d_1;16..31]

m_3[word addr;0..15], m_3[word addr;16..31] =

m_0[0..15], r_0[d_0;16..31]

In the above derivations the size compatibility of the arrays on the left and right hand sides of an assignment should be mentally verified during the backward substitution. For instance, if during the substitution the following resulted, t_3[8..24] = r_0[d_0;24..31], then this is an error in the array assignment, and the proof fails.

derived function:

n[2] = 1 & a MOD 4 = 2 ->

m[word addr;16..31] := r[d;16..31]

does the intended function =

m[word addr;16..31] := r[d;16..31]

in the partition n[2] = 1 & a MOD 4 = 2?

It does, therefore this part of the derived function passes.

(recognizing that if a MOD 4 = 2, then a MOD 2 = 0 is necessary in the verification of this partition)

2.condition:

n_0[2] = 1 & a MOD 4 = 1

rule:

$[13]_3 = [13]_2$ $\quad sb_3 = sb_2$ $\quad t_3 = t_2$

$\quad = 1$ $\quad\quad = sb_1$ $\quad\quad = t_1$

$\quad\quad = 8$ $\quad\quad = m_0$[word addr]

m_3[word addr] $= t_2$

$= t_1$

$= m_0$[word addr]

derived function:

n[2] = 1 & a MOD 4 = 2 -> psw[13] := 1

does the intended function = (psw[13] := 1) in the partition n[2] = 1 & a MOD 4 = 1?

It does, therefore this part of the derived function passes.

(recognizing that if a MOD 4 = 1, then a MOD 2 ¬= 0 is necessary in the verification of this partition)

The complete derived function is as follows:

```
n[2] = 0 & a MOD 4 ¬= 0 -> psw[13] := 1 |
n[2] = 0 & a MOD 4 = 0 -> m[word addr] := r[d] |
n[2] = 1 & a MOD 4 = 0 -> m[word addr;0..15] :=
                                        r[d;16..31] |
n[2] = 1 & a MOD 4 = 1 -> psw[13] := 1 |
n[2] = 1 & a MOD 4 = 2 -> m[word addr;16..31] :=
                                        r[d;16..31] |
n[2] = 1 & a MOD 4 = 3 -> psw[13] := 1 |
n[2] = 2 & a MOD 4 = 0 -> m[word addr;0..7] :=
                                        r[d;24..31] |
n[2] = 2 & a MOD 4 = 1 -> m[word addr;8..15] :=
                                        r[d;24..31] |
n[2] = 2 & a MOD 4 = 2 -> m[word addr;16..23] :=
                                        r[d;24..31] |
n[2] = 2 & a MOD 4 = 3 -> m[word addr;24..31] :=
                                        r[d;24..31] |
(n[2] ¬= 0 & 1 & 2 & 3) & (a MOD 4 = 0 | 1 | 2 | 3) -> I
```

When the rule of the derived function of each partition is compared with the intended function in that partition, the result is that the functions agree in each partition.

RESULT
  PASS

The last proof is a case where the proof process detected an error in a procedure, the calls procedure. Only the proof of the subprogram with the error is given. The type specifications of the the variables used in the subprogram are

```
psw: ARRAY[0..63] OF 0..1,
r: ARRAY[0..143] OF ARRAY[0..31] OF 0..1,
sc, sl: INTEGER,
calladdr: INTEGER,
disp: INTEGER
regval: INTEGER.
```

One constant, base_addr, is also referenced in the subprogram. A special function is used in the program definition and in the function specification. This function is DEC(a) = the decimal value of the binary array a.

The abbreviations ca, d, rv, ba are used for the variables calladdr, disp, regval, and base_addr, respectively, in the following proof.

FUNCTION

```
  r[s1;0] ¬= 1 & addr_excp(DEC(r[s1]) = 0 ->
          ca := sc + DEC(r[s1])
  | TRUE -> psw[9], ca := 1, (ba-d)*4
```

  (regval is not in the function specification because it is not an element of the domain)

PROGRAM

  subprogram of the calls procedure

```
  IF
    r[s1,0] ¬= 0
  THEN
    bin_dec(32,r[s1],regval)
    psw[9] := 0
    IF
      psw[9] = 0
    THEN
      ca := sc + rv
    FI
  ELSE
    psw[9] := 1
    ca := (ba-d)*4
  FI
```

PROOF

  The table in this proof is arranged in two sections like the table in the stores proof.

([9] is used for psw[9] in the table and derivations)

	condition	r[sl]	ca
1.	$r_0[sl_0;0] \neg= 1$	$r_1[sl_0] = r_0[sl_0]$ $r_2[sl_1] = r_1[sl_1]$	$ca_1 = ca_0$ $ca_2 = ca_1$
1a.	$[9]_2 = 0$	$r_3[sl_2] = r_2[sl_2]$	$ca_3 = sc_2 + rv_2$
1b.	$[9]_2 = 1$	$r_3[sl_2] = r_2[sl_2]$	$ca_3 = ca_2$
2.	$r_0[sl_0;0] = 1$	$r_1[sl_0] = r_0[sl_0]$	$ca_1 = (ba-d_0)*4$

	psw[9]	rv
1.	$[9]_1 = [9]_0$ $[9]_2 = addr\_excp(rv_1)$	$rv_1 = DEC(r_0[sl_0])$ $rv_2 = rv_1$
1a.	$[9]_3 = [9]_2$	$rv_3 = rv_2$
1b.	$[9]_3 = [9]_2$	$rv_3 = rv_2$
2.	$[9]_1 = 1$	$rv_1 = rv_0$

derivations:
(sc, sl, and d do not appear in the table because their values remain the same in every data state)

condition:

$r_0[sl_0;0] \neg= 0 \ \& \ [9]_2 = 0$

$r_0[sl_0;0] \neg= 0 \ \& \ addr\_excp(rv_1) = 0$

$r_0[sl_0;0] \neg= 0 \ \& \ addr\_excp(DEC(r_0[sl_0])) = 0$

rule:

$ca_3 = sc_2 + rv_2$

$= sc_1 + rv_1$

$= sc_0 + DEC(r_0[sl_0])$

```
  condition:
    $r_0[sl_0;0]$ ¬= 0 & $[9]_3 = 1$
    $r_0[sl_0;0]$ ¬= 0 & addr_excp($rv_2$) = 1
    $r_0[sl_0;0]$ ¬= 0 & addr_excp(DEC(r [$sl_1$])) = 1
  rule:
    $ca_3 = ca_2$
        $= ca_1$
        $= ca_0$

  condition:
    $r_0[sl_0;0] = 1$
  rule:
    $ca_1 = (ba-d_0)*4$
    $[9]_1 = 1$

derived function:
  r[sl;0] ¬= 1 & addr_excp(DEC(r[sl])) = 0 ->
                    ca,psw[9] := sc + DEC(r[sl]), 0 |
  r[sl;0] ¬= 1 & addr_excp(DEC(r[sl])) = 1 ->
                                   ca,psw[9] := ca, 1 |
  r[sl;0] = 1 -> ca,psw[9] := (ba-d)*4, 1
```

The derived function and the intended function do not agree in the partition r[sl;0] ¬= 1 & addr_excp(DEC(r[sl]) = 1. The derived function = (ca,psw[9] := ca, 1) and the intended function = (ca,psw[9] := (ba-d)*4, 1).

RESULT
  FAIL

This last example demonstrates a failure in a case structured proof. In at least one of the partitions that result from the derivation of the program function, the intended function and the derived function do not have the same rule. In this example r[s1;0] ¬= 1 & addr_excp(DEC(r[s1]) = 1 is the partition where the intended function and derived function do not agree.

Summary

Using the functional correctness method to prove the correctness of the procedures of the EXEC module in the RISC simulator provides an example of the application of the method to an operating program. Since the procedures to which the method is applied are not designed specifically as illustrative models for the functional correctness method and since they are not small procedures as the examples are, applying the functional correctness method to these procedures provides a realistic and rigorous test of the method's capability and usefulness. In the case of the stores procedure, the proof is more extensive than the proofs presented as examples, and in the case of the calls procedure, the proof points out an error in the procedure that was not found when the procedure was tested.

CHAPTER VI

FUNCTIONAL CORRECTNESS APPLIED TO NEW PROCEDURE

In the last chapter functional proof of correctness was applied to the procedures of the EXEC module of the RISC simulator. These were procedures already coded and tested, and proving their correctness was a test of the effectiveness of functional proof of correctness when it is applied to realistic procedures. In this chapter the effectiveness of functional proof of correctness is tested by applying the proving techniques to a new procedure of the RISC simulator. An arithmetic procedure was freshly designed and implemented for the simulator. It includes two subtract instructions, SUBR (subtract register) and SUBRC (subtract register with carry), which were not part of the original RISC instruction set or the original simulator. The functional correctness method was used to prove the correctness of the new procedure, and then the procedure was inserted into the RISC simulator. The objective was to determine whether proving the program's correctness eliminates logic errors.

PDL of the Arithmetic Procedure

The PDL of the top level of the arithmetic procedure is presented first.

```
PROC arithmetic(psw[14],scc: 0..1,n[1]:0..2)
  op1,op2: INTEGER
  [define operands;
   (psw[14] = 0 -> (n[1] = 0 -> add operands
                   | TRUE -> subtract operands));
   (scc = 1 & psw[14] = 0 -> set mask)]
  setops(op1,op2)
  [n[1] = 1 -> psw[14] = 0 -> add operands
   | TRUE -> psw[14] = 0 -> sub operands]
  IF
    n[1] = 0
  THEN [psw[14] = 0 -> add operands]
    IF
      psw[14] = 0
    THEN
      adds(op1,op2)
    FI
  ELSE [psw[14] = 0 -> sub operands]
    IF
      psw[14] = 0
    THEN
      subs(op1,op2)
    FI
  FI
  [scc = 1 & psw[14] = 0 -> set mask]
  IF
    scc = 1 & psw[14] = 0
  THEN
    setmask
  FI
CORP
```

The intended function specification for the preceding procedure illustrates the concept of deferring details. Rather than having the details of the program's function specified at the top level of design, the abstract functions, define operands, add operands, subtract operands, and set mask, were used in the intended function. The details are contained in the procedures implied by these abstract functions. Delaying intended function details for lower level procedures sharpens the basic functional objective of the program at the top level which is the level of greatest abstraction.

This method of deferring details is consistent with the concepts of self-containment, stepwise refinement, and stepwise abstraction. In stepwise refinement a procedure is designed by beginning with abstract functions and recursively replacing an abstract function with a more specific function (refer to chapter 3). In the case of the arithmetic procedure, the four abstract functions of the high level procedure are expanded at lower levels. Then in the reverse process of stepwise abstraction used in proving the program's correctness, the detailed and rigorous proofs occur at the lower levels. Because of self-containment of the lower level procedures, their details do not affect the high level procedure, and consequently, the general function references in the high level procedure are possible. Delaying details in this manner improves the clarity of the program, but does not diminish the accuracy of the proof.

The PDL of the four abstract functions in the arithmetic procedure is given on the following pages. Two special functions are used in the intended functions and later in the proofs of these procedures. These two special functions are DEC(x) = the decimal value of the 32 bit array x and TWOSBIN(m) = the 32 bit twos complement binary value of the integer m. Also, UND stands for undefined. When UND is assigned to a variable, it means that the value of the variable could be anything.

```
PROC setops(op1,op2: INTEGER,
            r: ARRAY[0..143] OF ARRAY[0..31] OF 0..1,
            s1,s2: 0 <= INTEGER <= 143,
            sc: INTEGER,
            psw[14],imm: 0..1,
            n[1]: 0..2)
  temp: INTEGER
  [n[1] = 0 v
   n[1] = 1 -> ((DEC(r[s1])=0 & r[s1;0]=1) v
               (DEC(r[s2])=0 & r[s2;0]=1 & imm=0)) ->
                 psw[14],op1,op2 := 1,UND,UND
               |TRUE ->
                op1 := DEC(r[s1]) - 2**32*r[s1;0],
                op2 := (imm=0 -> DEC(r[s2])-2**32*r[s2;0]
                        |TRUE -> sc)
   |TRUE    -> ((DEC(r[s1])=0 & r[s1;0]=1) v
               (DEC(r[s2])=0 & r[s2;0]=1 & imm=0)) ->
                 psw[14],op1,op2 := 1,UND,UND
               |TRUE ->
                op2 := DEC(r[s1]) - 2**32*r[s1;0],
                op1 := (imm=0 -> DEC(r[s2])-2**32*r[s2;0]
                        |TRUE -> sc]

  [DEC(r[s1]) = 0 & r[s1,0]=1 -> psw[14] := 1
   |TRUE -> op1 := DEC(r[s1]) - 2**32*r[s1;0]]
  twos_comp(s1,op1)
  [DEC(r[s2])=0 & r[s2,0]=1 & imm=0 -> psw[14] := 1
   |TRUE -> op2 := (imm=0 -> DEC(r[s2]) - 2**32*r[s2;0]
                    |TRUE -> sc)]
  IF
    imm = 0
  THEN
    twos_comp(s2,op2)
  ELSE
    op2 := sc
  FI
  [n[1]=2 & psw[14]=0 ->
        op2 := DEC(r[s1]) - 2**32*r[s1;0],
        op1 := (imm=0 -> DEC(r[s2]) - 2**32*r[s2;0]
                |TRUE -> sc)]
  IF
    n[1] = 2 & psw[14] = 0
  THEN [op1,op2 := op2,op1]
    temp := op1
    op1 := op2
    op2 := temp
  FI
CORP
```

```
PROC twos_comp(s: 0 <= INTEGER <= 143,
               op: INTEGER,
               r: ARRAY[0..143] OF ARRAY[0..31] OF 0..1,
               psw[14]: 0..1)
  regval: INTEGER
  [DEC(r[s])=0 & r[s;0]=1 -> psw[14] := 1
   |TRUE -> op := DEC(r[s]) - 2**32 * r[s;0]]
  bin_dec(32,r[s],regval)
  IF
    regval = 0 & r[s;0] = 1
  THEN
    psw[14] := 1
  ELSE
    op := regval - r[s;0]*2**31 - r[s;0]*2**31
  FI
CORP
```

The constant maxint is used in both the adds and subs procedures.

```
PROC adds(op1,op2: INTEGER,
          dest: 0 <= INTEGER <= 143,
          r: ARRAY[0..143] OF ARRAY[0..31] OF INTEGER,
          psw[14,20],n[2]: 0..1)
  result: INTEGER
  [n[2] = 0 -> (((op1>0 & op2>0) v
                 (op1<0 & op2<0)) &
                (ABS(op1) > maxint - ABS(op2))) ->
                  psw[14],r[dest] := 1,UND
               |TRUE -> r[dest] := TWOSBIN(op1 + op2)
   |TRUE   -> (((op1>0 & op2>0) v
                 (op1<0 & op2<0)) &
                (ABS(op1) > maxint-ABS(op2)-psw[20])) ->
                  psw[14],r[dest] := 1,UND
               |TRUE -> r[dest] := TWOSBIN(op1+op2+psw[20])
  bin_dec(32,r[dest],result)
  IF
    (((op1 > 0 & op2 > 0) v
      (op1 < 0 & op2 < 0)) &
     (ABS(op1) > maxint - ABS(op2)))
  THEN
    psw[14] := 1
    ELSE [n[2]=1 & (op1 & op2 different signs v
                  ABS(op1+op2) <= maxint-psw[20]) ->
            result := op1 + op2 + psw[20]
          |n[2]=1 & ABS(op1+op2) > maxint-psw[20] ->
            psw[14],r[dest] := 1,UND
          |TRUE -> result := op1 + op2]
      result := op1 + op2
      IF
        n[2] = 1
      THEN
        IF
          ((op1>0 & op2>0) v (op1<0 & op2<0)) &
          ABS(result) > maxint - psw[20]
        THEN
          psw[14] := 1
        ELSE
          result := result + psw[20]
        FI
      FI
    FI
    [r[d] := TWOSBIN(result)]
    place_in_reg(result)
  CORP
```

```
PROC subs(op1,op2: INTEGER,
          dest: 0 <= INTEGER <= 143,
          r: ARRAY[0..143] OF ARRAY[0..31] OF INTEGER,
          psw[14,20],n[2]: 0..1)
  result: INTEGER
  [n[2] = 0 -> (((op1<0 & op2>0) v
                 (op1>0 & op2<0)) &
                (ABS(op1) > maxint - ABS(op2))) ->
                 psw[14],r[dest] := 1,UND
               |TRUE -> r[dest] := TWOSBIN(op1 - op2)
   |TRUE   -> ((op1<=0 & op2>=0) &
               (ABS(op1) > maxint - op2 - psw[20])) ->
                 psw[14],r[dest] := 1,UND
               |TRUE -> r[dest] := TWOSBIN(op1-op2-psw[20])
  bin_dec(32,r[dest],result)
  IF
    (((op1<0 & op2>0) v
      (op1>0 & op2<0)) &
     (ABS(op1) > maxint - ABS(op2)))
  THEN
    psw[14] := 1
  ELSE [n[2]=1 & (op1>0 v op2<0 v
                ABS(op1)+op2 <= maxint-psw[20]) ->
        result := op1 - op2 - psw[20]
       |n[2]=1 & op1<=0 & op2>=0 &
                ABS(op1)+op2 > maxint-psw[20] ->
        psw[14] := 1
       |TRUE -> result := op1 - op2]
    result := op1 - op2
    IF
      n[2] = 1
    THEN
      IF
        (op1 <= 0 & op2 >= 0) &
        (ABS(result) > maxint - psw[20])
      THEN
        psw[14] := 1
      ELSE
        result := result - psw[20]
      FI
    FI
  FI
  place_in_reg(result)
CORP
```

```
PROC place_in_reg(result: INTEGER,
                  r: ARRAY[0..143] OF ARRAY[0..31] OF 0..1,
                  dest: 0 <= INTEGER <= 143)
  i: 0 <= INTEGER <= 32
  [r[dest] := TWOSBIN(result)]
  IF
    result >= 0
  THEN
    dec_bin(32,result,r[dest])
  ELSE
    result := ABS(result + 1)
    dec_bin(32,result,r[dest])
    [r[dest] := ¬r[dest]]
    FOR
      i := 0 to 31
    DO [r[dest;i] := ¬ r[dest;i]]
      IF
        r[dest;i] = 1
      THEN
        r[dest;i] = 0
      ELSE
        r[dest;i] = 1
      FI
    OD
  FI
CORP
```

i[19] is the 19th bit of the instruction register and the sign bit of the second operand if the second operand is an immediate value.

```
PROC setmask(s1,s2,dest: 0 <= INTEGER <= 143,
             psw[17],psw[18],
             psw[19],psw[20]: 0..1,
             n[1]: 0..2,
             imm,i[19]: 0..1)
  USE setmask_locs

  [(r[dest;0] = 1 -> psw[17,18] := 1,0 |
   DEC(r[dest]) = 0 -> psw[17,18] := 0,1 |
   TRUE -> psw[17,18] := 0,0),
   (n[1]=0 -> ((r[s1;0]=1 &
               ((imm=0 & r[s2;0]=1) v (imm=1 & i[19]=1))) v
               (r[s1;0]=1 & r[dest;0]=0) v
               (r[dest;0]=1 &
               ((imm=0 & r[s2;0]=0) v (imm=1 & i[19]=0))) ->
                 psw[19] := 1
              |TRUE -> psw[19] := 0),
              ((r[s1;0]=0 & r[dest;0]=1 &
               ((imm=0 & r[s2;0]=1) v (imm=1 & i[19]=1))) v
              (r[dest;0]=0 & r[s1;0]=0 &
               ((imm=0 & r[s2;0]=1) v (imm=1 & i[19]=1))) ->
                 psw[20] := 1
              |TRUE -> psw[20] := 0)

   |n[1]=1 -> ((r[s1;0]=1 & r[dest;0]=0 &
               ((imm=0 & r[s2;0]=0) v (imm=1 & i[19]=0))) v
               (r[s1;0]=0 & r[dest;0]=1 &
               ((imm=0 & r[s2;0]=1) v (imm=1 & i[19]=1))) ->
                 psw[19] := 1
              |TRUE -> psw[19] := 0),
              ((r[dest;0]=1 &
               ((imm=0 & r[s1;0]=r[s2;0]) v
                (imm=1 & r[s1;0]=i[19]))) v
               (r[dest;0]=1 & r[s1;0]=0 &
               ((imm=0 & r[s2;0]=1) v (imm=1 & i[19]=1))) v
               (r[dest;0]=0 & r[s1;0]=0 &
               ((imm=0 & r[s2;0]=1) v (imm=1 & i[19]=1))) ->
                  psw[20] := 1
               |TRUE -> psw[20] := 0)
```

```
 |TRUE   -> ((r[s1;0]=0 & r[dest;0]=0 &
             ((imm=0 & r[s2;0]=1) v (imm=1 & i[19]=1))) v
             (r[s1;0]=1 & r[dest;0]=1 &
             ((imm=0 & r[s2;0]=0) v (imm=1 & i[19]=0))) ->
               psw[19] := 1
            |TRUE -> psw[19] := 0),
            ((r[dest;0]=1 &
             ((imm=0 & r[s1;0]=r[s2;0]) v
              (imm=1 & r[s1;0]=i[19]))) v
            (r[dest;0]=1 & r[s1;0]=1 &
             ((imm=0 & r[s2;0]=0) v (imm=1 & i[19]=0))) ->
            (r[dest;0]=0 & r[s1;0]=1 &
             ((imm=0 & r[s2;0]=0) v (imm=1 & i[19]=0))) ->
               psw[20] := 1
            |TRUE -> psw[20] := 0)]

psw[17..20] := 0
[r[dest;0] = 0 -> psw[17] := 1 |
 DEC(r[dest]) = 0 -> psw[18] := 1]
IF
  r[dest;0] = 0
THEN
  psw[17] := 1
ELSE
  bin_dec(32,r[dest],regval)
  IF
    regval = 0
  THEN
    psw[18] := 1
  FI
FI
bit1,bit3 := r[s1;0],r[dest;0]
IF
  imm = 0
THEN
  bit2 := r[s2;0]
ELSE
  bit2 := i[19]
FI
IF
  n[1] = 0
THEN [set overflow and carry for addition]
  IF
    (bit1=0 & bit2=0 & bit3=0) v
    (bit1=1 & bit2=1 & bit3=0)
  THEN
    psw[19] := 1
  FI
```

```
    IF
      (bit1=0 & bit2=1 & bit3=1) v
      (bit1=0 & bit2=1 & bit3=0)
    THEN
      psw[20] := 1
    FI
  ELSE
    IF
      n[1] = 2
    THEN [bit1,bit2 := bit2,bit1]
      temp := bit1
      bit1 := bit2
      bit2 := temp
    FI
    [set overflow and carry bits for subtraction]
    IF
      (bit1=1 & bit2=0 & bit3=0) v
      (bit1=0 & bit2=1 & bit3=1)
    THEN
      psw[19] := 1
    FI
    IF
      (bit1=bit2 & bit3=1) v
      (bit1=0 & bit2=1 & bit3=1) v
      (bit1=0 & bit2=1 & bit3=0)
    THEN
      psw[20] := 1
    FI
  FI
CORP

DATA setmask_locs
  bit1,bit2,bit3: 0..1,
  temp: 0..1
  regval: INTEGER
ATAD
```

Proof Examples

Two of the functional correctness proofs resulting from proving the correctness of the new arithmetic procedure are presented in this section. Even though the two proofs are very similar in their function specifications and function derivations, they were chosen as examples because of their illustrative outcomes and because their derivations were not extremely lengthy and complex. Both of the proofs had results of failure, and consequently instigated a review of both the intended function and program implementation. In the first failure the intended function was modified, and in the second, the program implementation was corrected. In both the proofs the abbreviations mi, d, rs, [14], [20] are used for maxint, dest, result, psw[14], and psw[20] respectively. Also variables such as op1, op2, d, and n[2] whose values do not change within the procedure are treated like constants in the trace table and derivations, that is, their state changes are not included in the table, and they are not subscripted.

The first proof presented is the proof of the last level of abstraction of the adds procedure. Because of the level of abstraction, intended functions of lower level subprograms are used in the program specification.

FUNCTION

```
  [n[2] = 0 -> (((op1 > 0 & op2 > 0) v
                  (op1 < 0 & op2 < 0)) &
                 (ABS(op1) > maxint - ABS(op2))) ->
                   psw[14],r[dest] := 1,UND
                |TRUE -> r[dest] := TWOSBIN(op1 + op2)
   |TRUE    -> (((op1 > 0 & op2 > 0) v
                  (op1 < 0 & op2< 0)) &
                 (ABS(op1) > maxint-ABS(op2)-psw[20])) ->
                   psw[14],r[dest] := 1,UND
                |TRUE -> r[dest] := TWOSBIN(op1+op2+psw[20])
```

PROGRAM

```
  rs := DEC(r[d]);
  ((op1>0 & op2>0) v (op1<0 & op2<0)) &
  (ABS(op1) > mi-ABS(op2)) -> psw[14] := 1
  |TRUE -> (n[2]=1 & ((ABS(op1+op2)<=mi-psw[20]) v
                      (op1 & op2 different signs)) ->
              rs := op1+op2+psw[2]
           |n[2]=1 & ABS(op1+op2)>mi-psw[20] &
                      op1 & op2 same signs ->
              psw[14] := 1
           |TRUE -> rs := op1 + op2));
  r[d] := TWOSBIN(rs);
```

PROOF

cond	rs	r[d]	[14]
	rs_1 = DEC(r_0[d])	$r_1 = r_0$	$[14]_1 = [14]_0$
c1	$rs_2 = rs_1$	$r_2 = r_1$	$[14]_2 = 1$
c2	rs_2 = op1+op1+[20]	$r_2 = r_1$	$[14]_2 = [14]_1$
c3	$rs_2 = rs_1$	$r_2 = r_1$	$[14]_2 = 1$
c4	rs_2 = op1+op2	$r_2 = r_1$	$[14]_2 = [14]_1$
	$rs_3 = rs_2$	r_3 = TWOSBIN(rs_2)	$[14]_3 = [14]_2$

The conditions symbolically represented in the table are

```
c1 = ((op1>0 & op2>0) v (op1<0 & op2<0)) &
     ABS(op1) > mi-ABS(op2)

c2 = ((op1<=0 & op2>=0) v (op1>=0 & op2<=0) v
     ABS(cp1) <= mi-ABS(op2)) & n[2]=1 &
     (ABS(op1+op2) <= mi-[20] v op1 & op2 diff signs)

c3 = ((op1<=0 & op2>=0) v (op1>=0 & op2<=0) v
     ABS(op1) <= mi-ABS(op2)) & n[2]=1 &
     ABS(op1)+op2 > mi-[20] & (op1 & op2 same signs)
```

The above condition simplifies to

```
c3 = ((op1=0 & op2=0) v
     ABS(op1) <= mi-ABS(op2)) & n[2]=1 &
     ABS(op1+op2) > mi-[20] & (op1 & op2 same signs)

c4 = ((op1<=0 & op2>=0) v (op1>=0 & op2<=0) v
     ABS(op1) <= mi-ABS(op2)) & n[2]¬=1
```

derivations:

1. r_3 = TWOSBIN(rs_2) $\quad [14]_3 = [14]_2$
   = TWOSBIN(rs_1) $\quad = 1$
   = TWOSBIN(DEC(r_0[d]))

2. r_3 = TWOSBIN(rs_2) $\quad [14]_3 = [14]_2$
   = TWOSBIN(op1+op2+[20]) $\quad = [14]_1$
   $= [14]_0$

3. r_3 = TWOSBIN(rs_2) $\quad [14]_3 = [14]_2$
   = TWOSBIN(rs_1) $\quad = 1$
   = TWOSBIN(DEC(r_0[d]))

4. r_3 = TWOSBIN(rs_2) $\quad [14]_3 = [14]_2$
   = TWOSBIN(op1+op2) $\quad = [14]_1$
   $= [14]_0$

derived function:

```
c1 -> r[d], psw[14] := TWOSBIN(DEC(r[d])), 1
c2 -> r[d], psw[14] := TWOSBIN(op1+op2+psw[20]), psw[14]
c3 -> r[d], psw[14] := TWOSBIN(DEC(r[d])), 1
c4 -> r[d], psw[14] := TWOSBIN(op1-op2), psw[14]
```

In the partitions c1 and c2 the derived function is r[d], psw[14] := TWOSBIN(DEC(r[d])), 1; however the intended function for this partition is psw[14] := 1 which implies that r[d] does not change. Since TWOSBIN and DEC are not inverse functions TWOSBIN(DEC(r[d])) is not equivalent to r[d], so the functions do not agree.

RESULT
  FAIL

The nature of the failure in the above proof caused a reevaluation of the intended function. The question was asked if it were necessary for r[d] to retain its value in the partitions where the failures occurred. The value of r[d] is not important in those partitions, so the intended function was modified to indicate this fact. The modification was the addition of r[d] := UND in the two places of the intended function where the assignment psw[14] := 1 is located.

The next proof is the proof of the top level abstraction of the subs procedure. Once again the intended functions of the lower level subprograms are used in the program specification.

FUNCTION

```
   n[2] = 0 -> (((opl < 0 & op2 > 0) v
                  (opl > 0 & op2 < 0)) &
                 (ABS(opl) > maxint - ABS(op2))) ->
                   psw[14],r[dest] := 1,UND
                |TRUE -> r[dest] := TWOSBIN(opl - op2)
   |TRUE    -> ((opl <= 0 & op2 >= 0) &
                (ABS(opl) > maxint - op2 - psw[20])) ->
                   psw[14],r[dest] := 1,UND
                |TRUE -> r[dest] := TWOSBIN(opl-op2-psw[20])
```

PROGRAM

```
  rs := DEC(r[d]);
  ((opl>0 & op2<0) v (opl<0 & op2>0)) &
  (ABS(opl) > mi-ABS(op2)) -> psw[14],r[d] := 1,UND
  |TRUE -> (n[2]=1 & ABS(opl)+op2<=mi-psw[20] ->
              rs := opl-op2-psw[2]
            |n[2]=1 & ABS(opl)+op2>mi-psw[20] ->
              psw[14] := 1
            |TRUE -> rs := opl - op2));
  r[d] := TWOSBIN(rs);
```

PROOF

cond	rs	r[d]	[14]
	rs_1 = DEC(r_0[d])	$r_1 = r_0$	$[14]_1 = [14]_0$
c1	$rs_2 = rs_1$	$r_2 = r_1$	$[14]_2 = 1$
c2	rs_2 = opl-opl-[20]	$r_2 = r_1$	$[14]_2 = [14]_1$
c3	$rs_2 = rs_1$	$r_2 = r_1$	$[14]_2 = 1$
c4	rs_2 = opl-op2	$r_2 = r_1$	$[14]_2 = [14]_1$
	$rs_3 = rs_2$	r_3 = TWOSBIN(rs_2)	$[14]_3 = [14]_2$

The conditions symbolically represented in the table are

```
c1 = ((op1>0 & op2<0) v (op1<0 & op2>0)) &
     ABS(op1) > mi-ABS(op2)

c2 = ((op1<=0 & op2<=0) v (op1>=0 & op2>=0) v
     ABS(op1) <= mi-ABS(op2)) & n[2]=1 &
     ABS(op1)+op2 <= mi-[20]

c3 = ((op1<=0 & op2<=0) v (op1>=0 & op2>=0) v
     ABS(op1) <= mi-ABS(op2)) & n[2]=1 &
     ABS(op1)+op2 > mi-[20]

c4 = ((op1<=0 & op2<=0) v (op1>=0 & op2>=0) v
     ABS(op1) <= mi-ABS(op2)) & n[2]¬=1
```

derivations:

1. r_3 = TWOSBIN(rs_2) $\quad [14]_3 = [14]_2$
   = TWOSBIN(rs_1) $\quad = 1$
   = TWOSBIN(DEC(r_0[d]))

2. r_3 = TWOSBIN(rs_2) $\quad [14]_3 = [14]_2$
   = TWOSBIN(op1-op2-[20]) $\quad = [14]_1$
   $= [14]_0$

3. r_3 = TWOSBIN(rs_2) $\quad [14]_3 = [14]_2$
   = TWOSBIN(rs_1) $\quad = 1$
   = TWOSBIN(DEC(r_0[d]))

4. r_3 = TWOSBIN(rs_2) $\quad [14]_3 = [14]_2$
   = TWOSBIN(op1-op2) $\quad = [14]_1$
   $= [14]_0$

derived function:

```
c1 -> r[d], psw[14] := TWOSBIN(DEC(r[d])), 1
c2 -> r[d], psw[14] := TWOSBIN(op1-op2-psw[20]), psw[14]
c3 -> r[d], psw[14] := TWOSBIN(DEC(r[d])), 1
c4 -> r[d], psw[14] := TWOSBIN(op1-op2), psw[14]
```

The derived and intended function differ in partition c3. In partition c3 the derived function is

```
    r, psw[14] := TWOSBIN(DEC(r [d])), 1
```

whereas, the intended function is

```
    op1>0 v op2<0 ->
      r, psw[14] := TWOSBIN(op1-op2-psw[20]), psw[14]
|TRUE -> r, psw[14] := UND, 1.
```

RESULT
  FAIL

In this case the program rather than the intended function was modified to correct the error detected by the proof. The correction made was the if condition

```
  IF
    (ABS(result) > maxint - psw[20])
```

was expanded to

```
  IF
    (ABS(result) > maxint - psw[20]) &
    (op1 <= 0 & op2 >= 0)
```

within the subs procedure.

Programming Results

The purpose of writing and proving the new procedure was to test the effectiveness of the functional proof of correctness approach. After the new procedure was proved to be correct, it was inserted into the RISC simulator and tested with a goal of zero logic errors. The result of inserting the procedure into the simulator was that after the minor programmer errors, such as miscopying lines and misusing a nested ifthenelse statement, were corrected, the procedure executed correctly according to the specified function. There was one significant mistake - an incorrect specification of the intended function. This mistake, however, is one which can not be detected by functional proof of correctness.

Summary

A new arithmetic procedure was designed and its correctness was proved by the functional correctness method. Some of the proofs of the subprograms of the procedure resulted in failures, and thus, the intended function and program implementation were reviewed and a modification was made to correct the cause of the failure. After the proving of the program correctness, the procedure was inserted into the RISC simulator for testing. The result of the testing was that the proof process satisfactorily eliminated the logic errors.

CHAPTER VII

SUMMARY, CONCLUSIONS, AND SUGGESTED FUTURE RESEARCH

Summary

Functional proof of correctness is one approach of mathematically verifying the correctness of a program function. In a functional correctness proof the intended program function is compared to the derived program function, and if the intended function is equal to or a subset of the derived function, then the program is correct.

Structured programming is an important aspect of a functional correctness proof. To use the functional correctness approach on a program, the program must be a structured program because the fundamental control structures of structured programmming are also the fundamental structures used in proving the program's correctness. The methods of verifying the six basic control structures of structured programs, function, sequence, ifthen, ifthenelse, whiledo, dountil, and the two structures, fordo and case, that are extensions to the basic six are derived from and supported by the Correctness

Theorem. The Correctness Theorem provides a proof form and proof objective for each of the control structures, and various techniques are applied in the proof body to accomplish the proof objective and obtain a result of pass or fail. These techniques include mental verification, table tracing, special data structure handling, and complex conditional handling.

Inherent to structured programming are the concepts of hierarchical levels of program detail and program self-containment which form the basis for stepwise abstraction. Stepwise abstraction is the process of verifying the correctness of a program in a bottom-up method by verifying the correctness of a low level subprogram and replacing the subprogram with its intended function, thus, advancing the program to a higher level of abstraction. The control structures of structured programming form the subprograms used in stepwise abstraction. Because of the self-containment of the control structures, the proof of one subprogram does not affect the proof of another subprogram.

Applying the functional correctness method to the procedures of the EXEC module in the RISC simulator demonstrates the the proof method's performance on realistic procedures. The proofs of these procedures verified their correctness, and in some cases, errors that were not detected during the testing of the program were detected by the proofs.

Applying the functional correctness method to a new RISC procedure tested the effectiveness of functional correctness in detecting and eliminating logic errors before coding and testing of the procedure. The result of inserting the new procedure into the simulator after the procedure had been proved to be correct was that the procedure correctly executed its specified function. The logic errors were detected during the proof process and removed prior to the procedure's insertion into the simulator.

Conclusions

The conclusion drawn from applying functional proof of correctness to the old procedures and a new procedure of the RISC simulator is that functional proof of correctness increases the potential of having a program with zero logic errors. Also, even though the specification of the intended program function is not verified by proof of correctness, the intended function can be refined during the proving of a program because oftentimes when an error is found, the intended function is reviewed to see if what it specifies is what is actually desired. Furthermore, the guidelines outlining program and proof form generate pratical program structuring and beneficial documentation, and the proof process enforces a methodical verification of a program that is more rigorous and thorough than the prevailing freestyle desk checking.

Functional proof of correctness does have two drawbacks that detract its validity. One drawback of functional proof of correctness occurs during the proving of a program with a complex conditional structure. The numerous and lengthy paths become tedious and difficult to trace, and the conditions and rules become difficult to derive. The validity of the proof deteriorates in relation to the complexity of the conditions. Another drawback, which also is encountered in desk checking, testing, and debugging a program, is program familiarity. If one is familiar with a program, errors pointed out by the proofs tend to be missed because the foreknowledge of what the program is suppose to do influences one to believe that the program accomplishes what is specified. The solution to program familiarity is to have someone not familiar with the program design do the proving of the program.

Overall, however, the functional correctness method is useful in proving the correctness of a program's function and is effective in eliminating logic errors.

Suggested Further Research

Proof of correctness is only part of a larger design and verification process constructed to promote zero defect code. Another part of this process is module refinement and verification [8]. Whereas, a procedure provides a rule for a function, a module provides a rule for a state machine [8, 9]. Thus in module refinement and verification the module specification state machine (similar to a procedure's intended function) is compared to the module design state machine (similar to a procedure's derived function). It is during module refinement and verification that variable correctness is verified. Possible further research into module verification and refinement techniques and their application in conjunction with functional proof of correctness is suggested. Module verification and refinement techniques and functional proof of correctness techniques can be applied in the design and implementation of a program to test the effectiveness of these techniques in providing zero defect code, or in other words, providing a program that runs flawlessly (discounting compilation errors) from the beginning.

SELECTED BIBLIOGRAPHY

[1] Basili, V.R., and Dunlop, D.D., "A Comparative Analysis of Function Correctness." Computing Surveys, Vol. 14, No. 2 (June 1982), 229-244.

[2] Basili, V.R., and Noonan, R.E., "A Comparison of the Axiomatic and Functional Models of Structured Programming." IEEE Transactions on Software Engineering, Vol. SE-6, No. 5 (September 1980), 454-464.

[3] Basu, S.K., and Misra, J., "Proving Loop Programs." IEEE Transactions on Software Engineering, Vol. SE-1, No. 1 (March 1980), 76-86.

[4] Buxton, J.N., and Randell, B., eds., Software Engineering Techniques. Petrocelli/Charter, New York, N.Y., 1976, 222-226.

[5] Dijkstra, E.W., "Go-To Statement Considered Harmful." Communications of the ACM, Vol. 11, No. 3 (March 1968), 447-448.

[6] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming." Communications of the ACM, Vol. 12, No. 10 (October 1969), 576-583.

[7] Hughes, J.K., and Michtom, J.I., A Structured Approach to Programming. Prentice-Hall, Englewoood Cliffs, N.J., 1977.

[8] IBM Software Engineering Institute for Software Engineering Workshop Lecture Notes, January 10, 1983.

[9] Linger, R.C., Mills, H.D., and Witt, B.I., Structured Programming Theory and Practice. Addison-Wesley, Reading, Mass., 1979.

[10] Manna, Z., "Mathematical Theory of Partial Correctness." Journal of Computer Systems and Science, Vol. 5, No. 6 (June 1971), 239-253.

[11] Mills, H.D., "The New Math of Computer Programming." Communications of the ACM, Vol. 18, No. 1 (January 1975),43-48.

[12] Misra, J., "Some Aspects of the Verification of Loop Computations." IEEE Transactions on Software Engineering, Vol. SE-4, No. 6 (November 1978), 478-486.

[13] Patterson, D.A. and Sequin, C.H., "A VSLI RISC." Computer, Vol. 15, No. 9 (September 1982), 8-20.

[14] Patterson, D.A. and Sequin, C.H., "Design and Implementation of RISC I, report no. UCB-CSD 82-106, University of California, Berkeley, CA, (October 1982).

[15] Patterson, D.A. and Sequin, C.H., "RISC: A Reduced Instruction Set VLSI Computer." Proceedings of the Eighth International Symposium on Computer Architecture, (May 1981), 444-457.

[16] Yourdan, E., Techniques of Program Structure and Design. Prentice-Hall, Englewood Cliffs, N.J., 1975.

APPENDIX

Appendix Contents

The four proofs presented in this appendix are highlights of the proofs of the RISC procedures that were completed for this report. The proof for every procedure is not given because the proofs of several of the procedures were very similar such as the logical operator procedures, loads and stores procedures, conversion procedures, shift left and shift right procedures, and calls and return procedures; therefore, only one sample from a group of comparable proofs is presented throughout the report. Also, the proofs of the mac procedure, the setmask procedure, and the exception procedures consisted of no more than mental verification, so only one of these procedures' proofs is outlined.

Proof of Mac Procedure

```
FUNCTION

  psw[8] = 0 -> check protection exception,
  psw[9] = 0 -> check address exception,
  psw[13] = 0 -> check boundary exception;
  psw[8,9,13] = 0,0,0 ->
      addr := addr/4 + disp;
      in_out_flag = 0 -> m[addr] := memrow
      |TRUE -> memrow := m[addr]

PROGRAM

  IF
    psw[8] = 0
  THEN
    psw[8] := prot_excep(addr)
  FI
  IF
    psw[9] = 0
  THEN
    psw[9] := addr_excep(addr)
  FI
  IF
    psw[13] = 0
  THEN
    psw[13] := bndry_excep(addr)
  FI
  IF
    psw[8,9,13] = 0,0,0
  THEN
    addr := addr/4 + disp
    IF
      in_out_flag = 0
    THEN
      m[addr] := memrow
    ELSE
      memrow := m[addr]
    FI
  FI

PROOF

  mental verification

RESULT
  PASS
```

Proof Failure in Jumps Procedure

```
FUNCTION

  psw[8,9,13] := n[1] = 0 -> prot_excp(sc+DEC(r[sl])),
                             addr_excp(sc+DEC(r[sl])),
                             bndry_excp(sc+DEC(r[sl]))
                |n[1] = 1 -> prot_excp(y+DEC(psw[32..63])),
                             addr_excp(y+DEC(psw[32..63])),
                             bndry_excp(y+DEC(psw[32..63]))

  VAR n[1]: 0..1

PROGRAM

  IF
    n[1] = 0
  THEN
    bin_dec(32,r[sl],rg)
    psw[9] := addr_excp(rg)
    IF
      psw[9] = 0
    THEN
      c := sc + rg
    FI
  ELSE
    bin_dec(32,psw[32..63],lc)
    c := lc + y
  FI
  psw[8,9,13] := prot_excp(c),addr_excp(c),bndry_excp(c)
```

PROOF

(the abbreviations [8], [9], [13], pe, ae, be are used for psw[8], psw[9], psw[13], prot_excp, addr_excp, and bndry_excp, respectively, in the trace table and derivations)

	condition	[8]	[9]	[13]
1.	n[1] = 0	I	I	I
2.		I	$[9]_2 = ae(rg_1)$	I
3.	$[9]_2 = 0$	I	I	I
4.		$[8]_4 = pe(c_3)$	$[9]_4 = ae(c_3)$	$[13]_4 = be(c_3)$
3.	$[9]_2$ ¬= 0	$[8]_3 = pe(c_2)$	$[9]_3 = ae(c_2)$	$[13]_3 = be(c_2)$
1.	n[1] ¬= 0	I	I	I
2.		I	I	I
3.		$[8]_3 = pe(c_2)$	$[9]_3 = ae(c_2)$	$[13]_3 = be(c_2)$

	rg	c	lc
1.	rg_1 = DEC(r[sl])	I	I
2.	I	I	I
3.	I	$c_3 = sc + rg_2$	I
4.	I	I	I
3.	I	I	I
1.	I	I	lc_1 = DEC(psw[32..63])
2.	I	$c_2 = lc_1 + y$	I
3.	I	I	I

derivations:

```
  condition
    n[1] = 0 & [9]₂ = 0
    n[1] = 0 & ae(rg₁) = 0
    n[1] = 0 & ae(DEC(r[sl])) = 0

  rule
    [8]₄ = pe(c₃)
         = pe(sc + rg₂)
         = pe(sc + rg₁)
         = pe(sc + DEC(r[sl]))
     derivations for [9] and [13] are identical to [8]

  condition
    n[1] = 0 & [9]₂ ¬= 0
    n[1] = 0 & ae(rg₁) ¬= 0
    n[1] = 0 & ae(DEC(r[sl])) ¬= 0
  rule
    [8]$_3$ = pe(c$_2$)
          = pe(c$_1$)
          = pe(c$_0$)
     derivations for [9] and [13] are identical to [8]

  condition
    n[1] ¬= 0 (with data type 0..1 implies n[1] = 1)

  rule
    [8]$_3$ = pe(c$_2$)
          = pe(lc$_1$ + y)
          = pe(DEC(psw[32..63] + y))
     derivations for [9] and [13] are identical to [8]

Derived function:

  1. n[1] = 0 & ae(DEC(r[sl])) = 0 ->
       psw[8,9,13] := pe(DEC(r[sl])+sc),
                      ae(DEC(r[sl])+sc),
                      be(DEC(r[sl])+sc)

  2. n[1] = 0 & ae(DEC(r[sl])) ¬= 0 ->
       psw[8,9,13] := pe(c),ae(c),be(c)

  3. n[1] = 1 -> psw[8,9,13] := pe(DEC(psw[32..63)+y),
                                ae(DEC(psw[32..63)+y),
                                be(DEC(psw[32..63)+y)

  derived function and intended function do not agree in 2.

RESULT
  FAIL
```

Proof of Ret Procedure

```
FUNCTION

  r[dest;0] ¬= 1 ->
    (psw[8] := prot_excp(DEC(r[dest])+sc),
     psw[9] := addr_excp(DEC(r[dest])+sc),
     psw[13] := bndry_excp(DEC(r[dest])+sc);
     psw[8,9,13] = 0,0,0 ->
        psw[4], psw[32..63] := 0, BIN(DEC(r[dest)+sc),
        (psw[3] = 1 -> handle window ov on return
         |TRUE -> (DEC(psw[0..2]) < 7 ->
                     psw[0..2] := BIN(DEC(psw[0..2]) + 1)
                  |TRUE -> psw[12] := 1))
  |TRUE -> psw[9] := 1

PROGRAM

  bin_dec(32,r[dest],retaddr)
  retaddr := retaddr + sc
  IF
    r[dest;0] = 1
  THEN
    psw[9] := 1
  ELSE
    psw[9] := addr_excp(retaddr)
  FI
  psw[8],psw[13] := prot_excp(retaddr),
                          bndry_excp[retaddr]
  IF
    psw[8,9,13] = 0,0,0
  THEN
    psw[4] := 0,
    dec_bin(32,retaddr,loc_cntr);
    psw[32..63] := loc_cntr,
    bin_dec(3,psw[0..2],cwp);
    IF
      psw[3] = 1
    THEN
      handle window overflow on return
    ELSE
      IF
        cwp < 7
      THEN
        cwp := cwp + 1
        dec_bin(3,cwp,psw[32..63]
      ELSE
        psw[12] := 1
      FI
    FI
  FI
```

PROOF

In the table and derivations the abbreviations d, ra, lc, pe, ae, and be are used for dest, retaddr, loc_cntr, prot_excp, addr_excp, and bndry_excp. Also the psw is omitted from all references of psw values, i.e. psw[8] is [8].

	condition	[8]	[9]	[13]
1.		I	I	I
2.		I	I	I
3.	r[d;0] ¬= 1	$[8]_3$ = pe(ra_2)	$[9]_3$ = ae(ra_2)	$[13]_3$ = be(ra_2)
4.	$[8,9,13]_3$ = 0,0,0	I	I	I
5.		I	I	I
6.	psw[3] = 1	I	I	I
6.	psw[3]¬=1 & cwp_5<7	I	I	I
7.		I	I	I
6.	psw[3]¬=1 &cwp_5>=7	I	I	I
4.	$[8,9,13]_3$ ¬= 0,0,0	I	I	I
3.	r[d;0] = 1	I	$[9]_3$ = 1	I

	[0..2]	[4]	[12]	[32..63]
1.	I	I	I	I
2.	I	I	I	I
3.	I	I	I	I
4.	I	$[4]_4$ = 0	I	I
5.	I	I	I	$[32..63]_5$ = lc_4
6.	I	I	I	I
6.	I	I	I	I
7.	$[0..2]_7$ = BIN(cwp_6)	I	I	I
6.	I	I	$[12]_6$ = 1	I
4.	I	I	I	I
3.	I	I	I	I

	ra	cwp	lc	window ov
1.	$ra_1 = DEC(r[d])$	I	I	-
2.	$ra_2 = ra_1 + sc_1$	I	I	-
3.	I	I	I	-
4.	I	I	$lc_4 = BIN(ra_3)$	-
5.	I	$cwp_5 = DEC(psw[0..2]_4)$	I	-
6.	I	I	I	handled
6.	I	$cwp_6 = cwp_5 + 1$	I	-
7.	I	I	I	-
6.	I	I	I	-
4.	I	I	I	-
3.	I	I	I	-

derivations:

1 condition

$r[d;0] \neg= 1$ & $[8,9,13]_3 = 0,0,0$ & $psw[3] = 1$

$r[d;0] \neg= 1$ & $pe(ra_2) = 0$ & $ae(ra_2) = 0$ &
$be(ra_2) = 0$ & $psw[3] = 1$

$r[d;0] \neg= 1$ & $pe(ra_1 + sc) = 0$ & $ae(ra_1 + sc) = 0$ &
$be(ra_1 + sc) = 0$ & $psw[3] = 1$

$r[d;0] \neg= 1$ & $pe(DEC(r[d]) + sc) = 0$ &
$ae(DEC(r[d]) + sc) = 0$ & $be(DEC(r[d]) + sc) = 0$
$psw[3] = 1$

rule
visual derivation -
[4] ,[12] ,[0..2] = 0,[12] ,[0..2]
and window overflow is handled

$[8]_6 = [8]_5$	$[32..63]_6 = [32..63]_5$
$= [8]_4$	$= lc_4$
$= [8]_3$	$= BIN(ra_3)$
$= pe(ra_2)$	$= BIN(ra_2)$
$= pe(ra_1 + sc)$	$= BIN(ra_1 + sc)$
$= pe(DEC(r[d]) + sc)$	$= BIN(DEC(r[d] + sc))$

[9],[13] derivations same as [8]

```
2 condition
    r[d;0] ¬= 1 & $[8,9,13]_3$ = 0,0,0 & psw[3] ¬= 1 & $cwp_5$ < 7
    derivation of [8,9,13]. same as in condition 1
                 $cwp_5$ < 7
                 DEC($psw[0..2]_4$ ) < 7
                 DEC($psw[0..2]_0$ ) < 7

    r[d;0] ¬= 1 & pe(DEC(r[d]) + sc) = 0 &
    ae(DEC(r[d]) + sc) = 0 & be(DEC(r[d]) + sc) = 0 &
    psw[3] ¬= 1 & DEC(psw[0..2] ) < 7

  rule
    [8],[9],[13],[32..63],[4],[12] derived as in rule 1

    $[0..2]_7$ = BIN($cwp_6$)
             = BIN($cwp_5$ + 1)
             = BIN(DEC($psw[0..2]_4$ ) + 1)
             = BIN(DEC($psw[0..2]_0$ ) + 1)

3 condition
    r[d;0] ¬= 1 & $[8,9,13]_3$ = 0,0,0 & psw[3] ¬= 1 & $cwp_5$ >= 7
    derivations as in conditions 1 and 2

    r[d;0] ¬= 1 & pe(DEC(r[d]) + sc) = 0 &
    ae(DEC(r[d]) + sc) = 0 & be(DEC(r[d]) + sc) = 0 &
    psw[3] ¬= 1 & DEC(psw[0..2] ) >= 7

  rule
    [8],[9],[13],[32..63],[4],[0..2] derived as in rule 1
    [12] = 1

4 condition
    r[d;0] ¬= 1 & $[8,9,13]_3$ ¬= 0,0,0

    r[d;0] ¬= 1 & pe($ra_2$) ¬= 0 & ae($ra_2$) ¬= 0 &
    be($ra_2$) ¬= 0

    r[d;0] ¬= 1 & pe($ra_1$ + sc) ¬= 0 & ae($ra_1$ + sc) ¬= 0 &
    be($ra_1$ + sc) ¬= 0

    r[d;0] ¬= 1 & pe(r[d] + sc) ¬= 0 &
    ae(DEC(r[d]) + sc) ¬= 0 & be(DEC(r[d]) + sc) ¬= 0

  rule
    $[8]_4$ = $[8]_3$
         = pe($ra_2$)
         = pe($ra_1$ + sc)
         = pe(DEC(r[d]) + sc)
    [9],[13] derivations same as [8]
```

```
5 condition
    r[d;0] = 1

   rule
     [9] = 1  [8],[13],[4],[12],[0..2],[32..63] = I
```

In the comparison of the derived function and intended function in each of the 5 partitions, it is seen that the two functions agree.

RESULT
  PASS

Proof Failure in Shifts Procedure

FUNCTION

```
  (amt := (imm=0 -> DEC(r[s2]) |TRUE -> sc));
  (amt < 0 -> amt := 0 |amt > 32 & n[1] = 1 -> amt := 31
   |amt > 32 & n[1] ¬= 1 -> amt := 32)
  (n[2] = 0 -> shift_left(amt) | TRUE -> shift_right(amt));
  (scc = 1 ->
    (setmask;
     n[1] = 1 & r[s1;0] ¬= r[s1;1..amt] -> psw[19] := 1))
```

PROGRAM

This program is at the top level of abstraction.

```
  1 amt := (imm=0 -> DEC(r[s2]) | TRUE -> sc)
  2 n[1] = 1 & scc = 1 ->
      (psw19 := (r[s1;0] ¬= r[s1;1..amt] -> 1
                |TRUE -> 0)
  3 amt > 32 -> amt := 32
  4 n[2] = 0 -> shift_left(amt) |TRUE shift_right(amt)
  5 scc = 1 -> (setmask (includes psw[19] := 0);
                n[1] = 1 -> psw[19] := psw19)
```

PROOF

This proof is worked partially by mental verification and reasoning and partially by table tracing.

By inspection:

Line 1 of program corresponds directly with the first line of the FUNCTION. Also line 4 of the program corresponds directly with the fifth line of the FUNCTION. If n[1] ¬= 1, then the program simplifies to lines 1, 3, 4, and 5 where line 5 is simplified to scc = 1 -> setmask. This simplification corresponds directly with the FUNCTION for the case n[1] ¬= 1. If scc ¬= 1, then the program simplifies to lines 1, 3, 4 and again corresponds directly with the FUNCTION for this case. Finally, by inspection it can be seen that line 3 is not thorough enough to agree with the FUNCTION: in partition amt < 0, f = (amt := 0) while [P] = (amt := amt), and in partition amt> 32, f = (amt := (n[1] = 1 -> 31|TRUE -> 32)) while [P] = (amt := 32) So there is one failure found by mental verification.

FAIL

In all the cases in the following table n[1] = 1 and scc = 1 because the cases where either n[1] ¬= 1 or scc ¬= 1 has been mentally verified above.

	condition	psw19	psw[19]	amt
1.	$r[s1;0] \neg= r[s1;0..amt_0]$	$psw19_1 = 1$	I	I
2.	$amt_1 <= 32$	I	I	I
3.		I	$psw[19]_3 = psw19_2$	I
2.	$amt_1 > 32$	I	I	$amt_2 = 32$
3.		I	$psw[19]_3 = psw19_2$	I
1.	$r[s1;0] = r[s1;0..amt_0]$	$psw19_1 = 0$	I	I
2.	$amt_1 <= 32$	I	I	I
3.		I	$psw[19]_3 = psw19_2$	I
2.	$amt_1 > 32$	I	I	$amt_2 = 32$
3.		I	$psw[19]_3 = psw19_2$	I

derivations:

1 condition

$r[s1;0] \neg= r[s1;0..amt_0]$ & $amt_1 <= 32$
$r[s1;0] \neg= r[s1;0..amt_0]$ & $amt_0 <= 32$

rule

$psw[19]_3 = psw19_2$
$= psw19_1$
$= 1$

2 condition

$r[s1;0] \neg= r[s1;0..amt_0]$ & $amt_0 > 32$
$r[s1;0] \neg= r[s1;0..amt_0]$ & $amt_1 > 32$

rule

$psw[19]_3 = psw19_2$
$= psw19_1$
$= 1$

3 condition

$r[s1;0] = r[s1;0..amt_0]$ & $amt_1 <= 32$
$r[s1;0] = r[s1;0..amt_0]$ & $amt_0 <= 32$

rule

$psw[19]_3 = psw19_2$
$= psw19_1$
$= 0$

4 condition
  $r[sl;0] = r[sl;0..amt_0]$ & $amt_1 > 32$
  $r[sl;0] = r[sl;0..amt_0]$ & $amt_0 > 32$

 rule
  $psw[19]_3 = psw19_2$
  $= psw19_1$
  $= 0$

r[sl;0] = r[sl;0..amt] & amt <= 32 -> psw[19] := 0 PASS

r[sl;0] = r[sl;0..amt] & amt > 32 -> psw[19] := 0 Pass

r[sl;0] ¬= r[sl;0..amt] & amt <= 32 -> psw[19] := 1 PASS

r[sl;0] ¬= r[sl;0..amt] & amt > 32 -> psw[19] := 1
 This breaks down into two cases
 1. r[sl;0] ¬= r[sl;0..31]
 2. r[sl;0] = r[sl;0..31] & r[sl;0] ¬= r[sl;32..amt]

In case one the intended function and derived function agree, but in case two, f = (psw[19] := 0) while [P] = (psw[19] := 1), so this case FAILs.

RESULT
 FAIL

VITA

Marjorie Hyatt Turner

Candidate for the Degree of

Master of Science

Thesis: FUNCTIONAL PROOF OF CORRECTNESS TECHNIQUES APPLIED TO RISC SIMULATOR

Major Field: Computer Science

Biographical:

Personal Data: Born in Cincinnati, Ohio, August 16, 1959, the daughter of Edward T. and Phyllis D. Turner.

Education: Graduated from South Vigo High School, Terre Haute, Indiana, in May, 1977; received Bachelor of Science degree in Mathematics from Indiana State University in May, 1981; completed requirements for the Master of Science degree at Oklahoma State University in December, 1983.

Professional Experience: Actuarial Trainee at William M. Mercer; Chicago, Illinois, May, 1980 to August, 1980. Programmer at Weston Paper and Manufacturing Company; Terre Haute, Indiana, May, 1982 to August, 1982. Graduate Teaching Assistant, Department of Mathematics, Oklahoma State University, Stillwater, Oklahoma, August, 1981 to May, 1982, and August, 1982 to May, 1983.