

1980

## PL/99, a mid-level programming language for the TMS9900 microprocessor

Kenneth B. Walkley  
*College of William & Mary - Arts & Sciences*

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Walkley, Kenneth B., "PL/99, a mid-level programming language for the TMS9900 microprocessor" (1980). *Dissertations, Theses, and Masters Projects*. Paper 1539625082.  
<https://dx.doi.org/doi:10.21220/s2-w4zf-1215>

This Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).

PL/99  
"A  
A Mid-Level Programming Language  
for the TMS9900 Microprocessor

---

A Thesis  
Presented to  
The Faculty of the Program in Applied Science  
The College of William and Mary in Virginia

In Partial Fullfillment  
Of the Requirements for the Degree of  
Master of Science

---

by  
Kenneth B. Walkley  
1980

APPROVAL SHEET

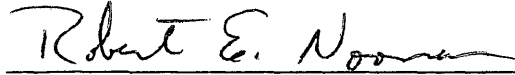
This thesis is submitted in partial fulfillment of  
the requirements for the degree of


Master of Science

  
Author

Approved, August 1980

  
John C. Knight

  
Robert E. Noonan

  
Michael K. Donegan

## TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS .....	iv
ABSTRACT .....	v
INTRODUCTION .....	2
CHAPTER 1. THE TMS9900 MICROPROCESSOR .....	5
CHAPTER 2. LANGUAGE DEFINITION .....	11
CHAPTER 3. COMPILER DEVELOPMENT AND SEMANTICS .....	38
CHAPTER 4. SAMPLE PROGRAMS .....	62
CHAPTER 5. CONCLUDING REMARKS .....	68
APPENDIX .....	70
NOTES .....	74
BIBLIOGRAPHY .....	75

## ACKNOWLEDGEMENTS

The author wishes to express his appreciation to Dr. John C. Knight for his encouragement and assistance throughout this effort, and to his wife Judy who provided substantial patience and confidence.

## ABSTRACT

A programming language for the Texas Instruments TMS9900 microprocessor and aspects of its design, development, and implementation are described. This mid-level language combines the efficiency of an assembly language with the structure and readability of a high level language, and is necessarily machine dependent. The programmer is provided with complete control of the target machine through a language framework which allows development of source programs which clearly exhibit the underlying algorithm and which are thus much easier to correct and understand than the corresponding assembly language program. Data types and operations are defined strictly in the context of the TMS9900. The PL/99 compiler is a cross-compiler which has been written in PASCAL for execution on Control Data Corporation 6600 and Cyber computers. The object module generated by the compiler is consistent with the Texas Instruments standard object module format.

PL/99

A Mid-Level Programming Language

for the TMS9900 Microprocessor

## INTRODUCTION

Until recent years, the languages available for programming most computers have been either low-level assembly languages or high-level languages such as FORTRAN, ALGOL, or PASCAL. Programs written in the machine-dependent assembly languages usually require relatively long development times and essentially preclude a clear exhibition of the structure of the underlying algorithm. The high-level languages generally overcome these deficiencies, but are machine independent and thus cannot provide explicit control of a given computer at the machine operation level.

In a classic paper, Wirth [1] presented the programming language PL360 which afforded the IBM 360 programmer the flexibility and degree of control typical of a conventional assembly language, but which exhibited the structure and readability of a high-level language. This early paper set the standard for a series of languages alternately called ALGOL-like assembly languages [2], structured assembly



languages [3,4], block-structured assembly languages [5], and mid-level assembly languages [6]. The target machines for these languages have ranged from the IBM 360 to smaller computers, minicomputers, microcomputers, and microprocessors.

Regardless of the particular title affixed, the purpose of each of these languages has been to provide the programmer with complete control of the target machine through a language structure which more nearly resembles a high-level language such as ALGOL or PASCAL. Each language provides certain high-level constructs coupled with a convenient syntax, but data types and operations are defined strictly in the context of the target machine. The resulting source programs clearly exhibit the structure of the underlying algorithm and are much easier to develop, correct, and understand than the corresponding assembly language program. The compilation of the source program to machine code is generally straightforward and no inefficiencies are introduced into the resulting machine code.

In a recent paper, Foster [6] describes the MID-level ASsembly language MIDAS which provides a basic framework for the development of a mid-level language for virtually any microprocessor. This basic MIDAS framework is extended to

obtain a dialect for a particular target microprocessor. The concepts and principles established for MIDAS have provided the basis for the development of PL/99, a mid-level assembly language for the Texas Instruments TMS9900 microprocessor.

The PL/99 compiler is a cross-compiler which has been written in PASCAL for execution on Control Data Corporation 6600 and Cyber computers. The primary goal has been to develop a prototype compiler with a sufficiently modular structure to allow convenient expansion and modification for future applications. The complete instruction set of the microprocessor has been implemented through a combination of a PASCAL-like language structure coupled with a machine instruction mnemonic capability for those instructions which are not readily incorporated into the basic PL/99 structure. The object module produced by the compiler is consistent with the format used by the Texas Instruments loaders.

The following chapters present a description of the target microprocessor, a detailed development of the syntax and semantics of PL/99, and example programs to illustrate the capabilities of the language.

## CHAPTER 1

### The TMS9900 Microprocessor

The Texas Instruments TMS9900 microprocessor [7] is a 16-bit single integrated circuit microprocessor which constitutes the basic building block for the 990 computer family including the 990/4 microcomputer and the 990/10 minicomputer. The PL/99 language has been directed toward the basic instruction set of the microprocessor, and thus the few remaining instructions which require the additional hardware of the 990/4 or 990/10 have not been considered. A brief summary of the microprocessor and its characteristics and capabilities is presented in the following sections.

## 1.1 Instruction Set

A machine instruction usually occupies one 16-bit memory word, but some instructions extend to two or three words for immediate operands and extended addresses. A word describing an instruction is decoded by the Central Processing Unit (CPU) into various fields within the 16-bits. The fields within an instruction word contain the following information:

1. The op code identifies the operation to be performed when the instruction is executed.
2. A single-bit B code indicates whether the instruction will affect a full 16-bit word or an 8-bit byte.
3. The two-bit T fields for source and destination operands specify which of the five available addressing modes is to be used (direct register, indirect register, memory address, memory address indexed, or indirect register autoincrement).
4. Source and destination register fields specify the register to be used.

5. A displacement field contains a bias to be added to the program counter to determine a target address.
6. Count fields indicate the bit shift count for shift instructions or the number of bits to be addressed by a Communications Register Unit (CRU) instruction.

## 1.2 Memory Addressing

A 16-bit memory address word is used for memory addressing. The least significant bit of this address word is used by the CPU to address bytes. All word addresses in memory are even values which range from 0000 to hexadecimal FFFE.

## 1.3 Workspace Concept and the Register File

A file of sixteen 16-bit registers (R0-R15) is available to the programmer through a workspace pointer register which points to a register file or workspace in memory. The workspace may be arbitrarily located in memory

on word (even-numbered byte) boundaries, and any number of such workspaces may be defined by the programmer. To access a particular workspace, the programmer loads the workspace pointer register with the address of the register file in memory.

#### 1.4 Addressing Modes

Approximately one-half of the instructions available on the TMS9900 allow a choice of one of five addressing modes for one or both operands. These modes allow convenient access to data in workspace registers or in memory locations.

1. Workspace Register Addressing - A specified workspace register contains the desired operand.
2. Workspace Register Indirect Addressing - A specified workspace register contains the address of the operand.
3. Workspace Register Indirect Autoincrement Addressing - A specified workspace register contains the address of the operand. Upon

completion of the operation, the workspace register contents are incremented by one for byte instructions and by two for word instructions.

4. Indexed Memory Addressing - The word following the instruction contains the base address while a specified workspace register contains the indexed value. The effective address of the operand is found by summing the base address and the indexed value.
5. Symbolic Memory Addressing (Direct and Indirect) - The operand or operand address is contained in the word following the instruction.

### 1.5 Status Register

A condition code register holds relevant information on preceding operations and is referenced by conditional branches. This status register can be loaded and stored as a single 16-bit register, or individual bits may be altered with a single instruction.

## 1.6 Communications Register Unit

The Communications Register Unit (CRU) is the I/O data interface for the 990 computer family. The CRU may be visualized as 4K consecutive bits which are addressed as independent bits or in groups of up to 16 bits by CRU instructions. The CRU is best described as a 4096-bit input register and a 4096-bit output register.

## 1.7 Register Conventions

Conventions in certain instructions require that specified registers within the workspace be reserved for particular functions. Program storage requirements are reduced by conserving space in the instruction word through the use of these conventions. Register R11 is used as a link register (branch and link), R12 is used as a base register for CRU I/O addresses, and R13, R14, and R15 are used as link registers for status exchange operations. R0 is inhibited from use as an index register through a zero default in index register identification which generates the immediate address mode ( direct symbolic memory addressing ).



## CHAPTER 2

### Language Definition

PL/99 mimics PASCAL to some extent in its basic structure and capabilities, but the machine dependency of a mid-level language results in certain differences and restrictions not usually associated with a high-level language. The following sections present a detailed definition of PL/99.

#### 2.1 Lexical Elements

The lexical structure of PL/99 is patterned after PASCAL, but tends to be more restrictive in several aspects.

### 2.1.1 Character Set and Special Symbols

The PL/99 compiler will recognize the twenty-six upper case letters A through Z and the digits 0 through 9. Other symbols recognized are:

+	:	(
-	=	)
/	<>	[
*	<	]
:=	<=	>
.	>	&
,	>=	\$
	-	

### 2.1.2 Free Format Input

PL/99 source programs may be written in a free format which allows the programmer to structure the code in a manner which exhibits the underlying algorithm in an easily readable style. A statement may begin and end at any point on a card; multiple statements may appear on a single card; and statements may extend to more than one card. Any number of blanks may appear between distinct identifiers, numbers, or other symbols, but no blanks may occur within an identifier, number, or other symbol.

### 2.1.3 Identifiers

Identifiers are names associated with constants, variables, types, procedures, and labels. They must begin with a letter which may be followed by any combination of letters, digits, or underscores. PL/99 permits identifiers of any length, but only the first ten characters are used. Identifiers denoting distinct objects must therefore be unique in the first ten characters.

Examples: COUNTER, FIRST\_ID, TEMP1

### 2.1.4 Numbers

Positive integers composed of strings of the digits 0 through 9 and the letters A through F are allowed in PL/99. If the digit string is preceded by the character \$, the integer is interpreted as hexadecimal, otherwise the base is assumed to be ten. The target microprocessor is a 16-bit machine, and thus the integers must be in the range 0 to 65535.

Examples: 123, \$ABC1

### 2.1.5 Comments

A comment statement may be inserted at any point where a blank is allowed (e.g., between any two identifiers, numbers, or special symbols) to serve as documentation or to enhance the readability of the code. The delimiters (\* and \*) bracket the comment which may be any sequence of symbols not containing \*). A comment has no effect on the code generated.

Example: (\* THIS IS A COMMENT. \*)

### 2.1.6 Reserved Words

Certain identifiers in PL/99 are classified as reserved words, and the programmer may not use them in any context other than that set forth in the language definition. In particular, reserved words may not be used as identifiers. The PL/99 reserved words are:

ABS	ENDP	PROGRAM
AND	ENDPR	REPEAT
ARRAY	ENDW	THEN

BEGIN	GOTO	UNTIL
BYTE	IF	VALUE
CALL	INV	VAR
CONST	LABEL	WHILE
DO	OF	WORD
ELSE	OR	XOR
ENDIF	PROCEDURE	

In addition to these reserved words, the register identifiers R0,R1,...R15 are predeclared identifiers which are used to reference workspace registers. The user may optionally declare any of these register identifiers to designate a label, constant, variable, or procedure identifier and thus override the predeclaration.

#### 2.1.7 Mnemonics

A number of the TMS9900 instructions do not readily lend themselves to implementation through the PL/99 structure, and thus a limited mnemonic capability has been included in the language. The following mnemonics have been implemented:

AB	JOP	SRA
B	LDCR	SRC
BL	LIMI	SRL

BLWP	LWPI	STCR
CB	MOVB	STST
CLR	RTWP	STWP
COC	SB	SWPB
CZC	SBO	SZC
JH	SBZ	SZCB
JL	SETO	TB
JNC	SLA	X
JNO	SOC	
JOC	SOCB	

Detailed discussions of the mnemonic capability are presented in sections 2.9.4 and 3.6.7.4.

## 2.2 Input/Output

The concept of input/output for the target machine has not been rigorously addressed in PL/99, but access to the Communications Register Unit (CRU) has been provided through the mnemonic facility described in section 2.9.4. The five CRU instructions may be directly generated through the use of the appropriate mnemonic.

### 2.3 Grammar Specification

The context free syntax for PL/99 is formally described using the traditional Backus-Naur Form (BNF). As presented in the appendix, these BNF productions collectively describe the permissible forms of a PL/99 program.

### 2.4 PL/99 Program Structure

A program written in PL/99 will be composed of the following parts:

```
program header
    label declaration section
    constant declaration section
    variable declaration section
    value section
    procedure definitions
BEGIN
    program body
ENDP.
```

Each program section will appear in the order given above. The label, constant, or variable declarations may be

empty. Value initializations and procedure definitions are likewise optional.

This program structure offers the advantage of a clear organization with the label, constant, variable, and initial values conveniently located at the beginning of the program. The declaration sections facilitate the construction of the symbol table and avoid errors which can occur when extraneous identifiers are inadvertently introduced due to coding errors. The procedure definitions and program body relative positions are easily incorporated into the TMS9900 memory organization.

## 2.5 Program Header

The program header is the first statement in every PL/99 program. It is composed of the reserved word PROGRAM followed by an identifier and a semicolon. The program header assigns a name to the program which is subsequently used in the object module.

Example: PROGRAM PL\_PROG;



## 2.6 Declarations and Types

All labels, constants, and variables must be declared in PL/99. Initial values for selected variables may be indicated in the VALUE section. A declaration section may be empty and thus not appear in a particular PL/99 program. Note that it is not necessary to declare the register identifiers R0-R15, but initial values may be preset for them.

The allowed data types include two of simple type and one structured type. These data types are defined to reflect the TMS9900 instruction types and addressing capabilities. Programmer defined types such as found in PASCAL are not permitted.

### 2.6.1 Labels

The label declaration section is composed of the reserved word LABEL followed by a list of label identifiers separated by commas. The last label identifier is followed by a semicolon. A label is an identifier associated with a given source statement and is usually the target of one or more GOTO or mnemonic jump statements.

Example: LABEL LAB1, LAB2, LAB3;

### 2.6.2 Constants

The constant declaration allows association of an integer constant value with a constant identifier. Constant definitions separated by commas follow the reserved word CONST. Each constant definition has the syntax identifier=constant value. A semicolon completes the CONST section.

Example: CONST MAX\_HEX = \$FFFF, PROCESSOR = 9900;

The use of constant identifiers allows the grouping of program constants in a convenient location where they can be easily noted for documentation purposes or changed for a particular execution case.

### 2.6.3 Variables

Variable names other than the predeclared register identifiers are declared in the variable declaration. The variable declaration section is composed of the reserved

word VAR followed by a series of variable declarations. Each variable declaration is composed of a list of identifiers followed by a colon followed by a type. The type may be a simple type or a structured type. The simple types are WORD and BYTE, and the structured type is the ARRAY type. Variables of type BYTE are packed two per word of storage. For the ARRAY type, the colon is followed by the reserved word ARRAY, the array bound specification in brackets, the reserved word OF, and the BYTE or WORD type specification. The first index of every array is assumed to be zero, and the bound specification denotes the maximum index to be used for the array, i.e., the number of array elements plus one.

```
Example:  VAR V1,V2: BYTE;
          V3,V4: WORD;
          V5 : ARRAY [3] OF BYTE;
```

Note that the array V5 is composed of the four elements V5 [0], V5 [1], V5 [2], and V5 [3].

#### 2.6.4 Value Initialization

A VALUE declaration section has been implemented in PL/99 to provide a convenient means of initializing program

variables and registers at load time. This capability provides a compact method for program initialization without resorting to assignment statements, and thus results in more efficient machine code as well as a more readable source program. The VALUE declaration ultimately results in loader directives to perform the required initialization.

The VALUE declaration section is indicated by the reserved word VALUE. Each initialized variable name is equated to the appropriate numerical value in a manner paralleling the CONST declarations.

The VALUE declaration for array variables assumes that the values declared correspond to the ordered elements of the array beginning with index zero, but does not require that all elements be initialized.

Example: VALUE HEX\_DIGIT = \$B, XARRAY = [4,2,\$A,3];

Note that XARRAY[0]=4,XARRAY[1]=2,etc., but there may be more than four elements in XARRAY.

## 2.7 Procedure Definition and Usage

PL/99 allows the use of procedures, but only in a restricted sense. A procedure is a block of code beginning with the reserved word PROCEDURE coupled with the procedure name, and followed by an arbitrary number of source statements which constitute the procedure body. The procedure is terminated with the reserved word ENDPR. Only global variables are allowed. There is no parameter list, and there are no local variables as in PASCAL. There may be any number of procedures, but they may not be nested. This limited procedure capability does provide the programmer with the capability to execute a fixed block of code repeatedly from any point within the program body. Additionally, this form for procedures is readily incorporated into the memory organization and provides direct access to the TMS9900 branch and link instructions. The CALL statement is used to branch to the desired procedure and is described in section 2.9.3.

```
Example: PROCEDURE ADD_ONE;
          A:=A+1;
          B:=B+1;
          ENDPR;
```

## 2.8 Addressing Modes

The five addressing modes available on the TMS9900 for source and destination operands have been implemented directly in PL/99. Four addressing modes are directly expressible by the programmer, and two are activated through the choice of non-register variable names or the ARRAY data structure.

### 2.8.1 Workspace Register Addressing

As previously noted, the workspace register identifiers R0,R1,...,R15 are predeclared in PL/99. If the contents of a workspace register are to be used as an operand, the appropriate register is indicated simply by name, e.g., R5:=R5+1.

### 2.8.2 Workspace Register Indirect Addressing

If a workspace register contains the address of an operand, the appropriate register identifier is followed by the symbol  $\wedge$ , e.g., R5 $\wedge$ :=R5 +2. Note that no blanks are

permitted between the register name and mode specification.

### 2.8.3 Workspace Register Indirect Autoincrement Addressing

This mode functions in the same manner as indirect addressing, but upon completion of the operation the workspace register contents are incremented by one for byte instructions and by two for word instructions. This mode is indicated by suffixing the symbol & to the register name and ^, e.g.,  $R5\wedge := R5\wedge + R4\wedge$ . Note that this addressing mode may be used to step through a table of values without recourse to the ARRAY structure.

### 2.8.4 Symbolic Memory Addressing

Direct symbolic memory addressing has been implemented for variable names which do not correspond to register identifiers. Variable names descriptive of the problem being programmed may thus be used to further enhance the readability of the algorithm. The two words which follow a machine instruction are reserved for the addresses of the variable names which designate the source and destination operands, respectively. The programmer simply employs the

variable name in a statement to activate this mode, e.g.,  
R5^:=DATA.

### 2.8.5 Indexed Memory Addressing

This mode allows the use of any register as an index register for determination of a required effective address. Both explicit and implicit forms of this addressing mode have been implemented.

In the explicit form, a variable identifier is followed by parentheses enclosing an integer constant or constant identifier in the range 1-15 which specifies the register to be used as the index register. The TMS9900 instruction set requires that register R0 cannot be used for indexing. The word following the machine instruction will contain the base address, i.e., the address of the variable identifier, which when added to the contents of the index register yields the effective address of the operand.

Examples: TABLE(\$A), XVAL(3), YVAL(ONE)

The ARRAY data structure utilizes this addressing mode, but in an implicit sense. An array variable identifier is



followed by brackets enclosing the array index which may be an integer constant, a constant identifier, or a register identifier:

Examples: A [2], A [TWO], A [R2]

The base address corresponds to the address of the first element of the array A[0]. The array index simply becomes a specification of the indexed value used to obtain the address of a given element within the array. If one of the constant forms for the index is used, however, the addressing mode reverts to direct symbolic memory addressing because the address of the specified array element may be determined directly from the array base address and constant. When a register identifier is employed, the contents of that register are added to the array base address to obtain the address of the desired element. Note that the entire array may be accessed by incrementing the indexing register contents in a loop.

```
Example: R1 := 0;
         REPEAT
           A[R1] := 0;
           R1 := R1 + 1;
         UNTIL R1 = 5;
```

## 2.9 Statements

The program and procedure bodies are composed of a list of statements separated by semicolons. Statements available in PL/99 include assignment, GOTO, CALL, mnemonic, IF, REPEAT, WHILE, and the empty statement. Statements may be unlabeled, or a label identifier followed by a colon may precede a statement for reference by a GOTO or mnemonic jump statement.

### 2.9.1 Assignment Statement

The assignment statement is used to assign the value of an expression to a register or variable identifier. As in PASCAL, the symbol := is the assignment operator, which separates the destination operand on the left-hand-side from an expression on the right-hand-side of the assignment statement. Several forms for the expression are allowed, but all reduce to a single machine instruction and must reflect the source and destination operand ordering required by the instruction set of the microprocessor.

Expressions may be composed of a single constant, a single identifier, a single identifier preceded by the monadic operators + or -, or an identifier-identifier or an identifier-constant pair separated by any of the arithmetic operators +, -, \*, or /, or by any of the Boolean operators AND, OR, or XOR. Also permitted are the ABS and INV expressions described below. Each of these forms may be directly translated into the appropriate TMS9900 machine instructions.

Expressions in PL/99 are restricted to a single operator form. This restriction is not generally serious, however, and should provide most programmers with an assignment statement syntax which will be sufficiently powerful for most applications. The adoption of this limited expression follows from two additional considerations. Firstly, a primary goal of the language is to provide the programmer with control of register allocation and use. Thus there are no registers available for the storage of intermediate results obtained during a more lengthy expression evaluation. Secondly, expressions are evaluated in a simple left to right scan with no parentheses or operator precedence constraints. The single operator syntax precludes any difficulties arising from these considerations and simultaneously results in a one-to-one correspondence between machine instructions and

the assignment statements.

The concept of source and destination operands is fundamental to the TMS9900 instruction set and must be reflected in the specification of expressions in PL/99. Expressions involving the arithmetic operators +, -, \*, or / must have the syntax  $d := d \text{ operator } s$  where  $d$  and  $s$  are the destination and source operands respectively.

```
Examples: R1:=R1+R2;
          R3^:=R3^*R1;
          DATA:=DATA-VAL1;
          VAL1:=VAL1/DATA;
```

In addition, the TMS9900 instructions for AND and OR operations perform the indicated operation between the contents of a workspace register and an immediate operand (i.e., a constant or constant identifier). The XOR operation, however, allows the use of a workspace register destination operand and a source operand which is resident in the workspace or in memory. These peculiarities of the TMS9900 must be strictly observed and reinforces the machine dependency of PL/99.

```
Examples: R1:=R1 AND 1;
          R3:=R3 XOR V1;
          R2:=R2 OR $A;
```

When register identifiers are employed in any expression, the addressing mode may be indicated as previously described. Note also that any variable identifier may be an array element.

The last two forms for an expression involve the reserved words ABS and INV. These expressions are monadic and result in the absolute value and inverted value of a given operand.

Examples: R1 := INV(R1); V1 := ABS(V1);

### 2.9.2 GOTO Statement

The GOTO statement results in an unconditional branch to the statement preceded by the indicated statement label, and is composed of the reserved word GOTO followed by a label identifier. A GOTO statement may be used within a procedure, but transfer into or out of a procedure using a GOTO is not permitted.

Example: GOTO LAB1;

### 2.9.3 CALL Statement

The CALL statement is used to branch to a procedure. The reserved word CALL is followed by the procedure name.

Example: CALL PROC1;

### 2.9.4 Mnemonic Statement

Several of the TMS9900 instructions do not readily lend themselves to incorporation into the PL/99 structure. A mnemonic capability has thus been included to provide convenient access to certain jump, shift, compare, branch, and CRU instructions. Also included are several byte operand instructions which may be used when operations involving the left byte of a register are required. A mnemonic statement may occur at any point where any other statement may occur. Mnemonics are denoted by a pair of periods enclosing a mnemonic identifier. The operands for the instruction follow the second period by at least one blank and are separated by a comma if more than one operand occurs. The following table summarizes the available mnemonics and gives examples of their usage and operand form.

## PL/99 Mnemonics

Add Bytes	.AB.	R2,R1
Branch	.B.	ADDR1
Branch and Link	.BL.	R4
Branch and Load Workspace Pointer	.BLWP.	R5^
Compare Bytes	.CB.	R2,R1
Clear Operand	.CLR.	V1
Compare Ones Corresponding	.COC.	R2,V1
Compare Zeroes Corresponding	.CZC.	R2,V1
Jump if No Carry	.JNC.	LAB1
Jump if No Overflow	.JNO.	LAB2
Jump on Carry	.JOC.	LAB3
Jump if Odd Parity	.JOP.	LAB4
Jump if High	.JH.	LAB5
Jump if Low	.JL.	LAB6
Load Interrupt Mask Immediate	.LIMI.	\$000A
Load Communication Register	.LDCR.	R1,5
Load Workspace Pointer Immediate	.LWPI.	WPADDR
Move Byte	.MOVB.	R2,R1
Return with Workspace Pointer	.RTWP.	
Subtract Bytes	.SB.	R2,R1
Set Bit to One	.SBO.	1
Set Bit to Zero	.SBZ.	\$AF
Set To One	.SETO.	A2 [4]

Shift Left Arithmetic	.SLA.	R3,4
Set Ones Corresponding	.SOC.	R2,V1
Set Ones Corresponding, Byte	.SOCB.	V2,V3
Shift Right Arithmetic	.SRA.	R4,SHIFTCOUNT
Shift Right Circular	.SRC.	R4,\$A
Shift Right Logical	.SRL.	R4,0
Store Communication Register	.STCR.	R2,2
Store Status	.STST.	R3
Store Workspace Pointer	.STWP.	R5
Swap Bytes	.SWPB.	R3
Set Zeroes Corresponding	.SZC.	A1 [1] ,V1
Set Zeroes Corresponding, Byte	.SZCB.	V2,V3
Test Bit	.TB.	6
Execute	.X.	R5

The use of these mnemonics must always adhere to the basic definitions and usage prescribed for the instruction set [7]. Additional comments regarding the mnemonic capability may be found in section 3.6.7.4.

### 2.9.5 Conditions



The three remaining statements - IF, WHILE, and REPEAT - all involve the evaluation of a condition which is composed of an identifier-identifier or identifier-constant pair separated by one of the relational operators =, <>, <, <=, >, or >=. This syntax maps directly into the TMS9900 compare instructions.

If the indicated relationship holds between two identifiers or between an identifier and a constant, the condition is assigned the Boolean value TRUE, and the appropriate bits in the machine status register set to one. Otherwise the condition is FALSE. Examples of the use of conditions are presented in the discussions of the IF, WHILE, and REPEAT statements below.

Restrictions on instruction format peculiar to the TMS9900 previously noted for assignment statements and Boolean expressions apply similarly to conditions. If an identifier and constant are to be compared, the identifier must be a register identifier.

#### 2.9.6 IF Statement

The IF statement is available in two forms. The syntax is IF condition THEN statement list ENDIF or IF condition THEN statement list ELSE statement list ENDIF with the usual semantics in both cases.

Examples: IF R1>V1 THEN V1:=R1; ENDIF;

```
IF R1<0 THEN R1:=-R1
ELSE
  R1:=R1+R2;
  R2:=V1;
ENDIF;
```

A semicolon preceding the ENDIF or ELSE is optional. The compiler will insert an empty statement following the semicolon to complete the parse.

#### 2.9.7 Repetitive Statements

The WHILE and REPEAT statements are used to specify that a block of statements be executed while or until a condition is true. The syntax of the WHILE statement is WHILE condition DO statement list ENDW. The REPEAT statement evaluates the condition after the statement list has been executed and has the syntax REPEAT statement list UNTIL condition.

In each case the number of iterations to be performed is not specified directly. Repetition of the statement list continues until the condition becomes false. The REPEAT statement will always be executed at least once, but the WHILE statement will result in no execution of the statement list if the condition is initially false.

```
Example:  WHILE R1 > 0 DO
           B := B + 1 ;
           R1 := R1 - 1;
           ENDW;

           REPEAT A := A - 1 UNTIL A = 0;
```

#### 2.9.8 Empty Statement

An empty statement contains no symbols, causes no action, and results in no machine code.

## CHAPTER 3

### Compiler Development and Semantic Actions

The PL/99 compiler has been constructed using a system of compiler writing codes developed jointly at the NASA-Langley Research Center and the College of William and Mary. A fairly complete compiler may be generated through the use of these programs in conjunction with a grammar specification for the target language and appropriate semantic information. The development and implementation of the PL/99 compiler, its major components and semantic actions are discussed in this chapter.

### 3.1 Compiler Generation

Development of the PL/99 compiler has made use of a series of programs which facilitate the construction of a given compiler by providing a number of automatic features which result in a fairly complete compiler. The basis for the generation of a compiler via this approach is a set of grammar rules for the target language. These rules or productions are expressed in a form similar to the Backus-Naur Form (BNF) and may optionally include interspersed semantic information. This basic definition of the target language is analyzed by the parser generator (PARGEN) which produces optimized LALR(1) parser tables and parsing procedures. These results are incorporated into a skeletal compiler which additionally includes a symbol scanner (NEXTSYM), a procedure SYNTHESIZE which contains a case label for each grammar rule and its associated semantics, and dummy procedures for subsequent incorporation of a variety of functions including error processing, symbol table entry, searching and printing, and object code generation. This basic compiler is written in PASCAL 6000 Version 3 and all LABEL, CONST, VAR, and VALUE declarations are specified automatically. The user must then construct and incorporate specific procedures required for a particular application.

Subsequent modifications to the grammar may be incorporated by executing the various components of the compiler writing system with the revised grammar and semantics. The existing version of the compiler may also be specified such that the new skeletal compiler will be merged with the old compiler to produce a new program containing the revised grammar and semantics as well as the specific procedures and other code previously incorporated into the old compiler. This capability provides a convenient process for evolving a final compiler through several stages of development with a minimum of repetitive effort or cumbersome text editing between old and new versions of the compiler.

### 3.2 Compiler Structure

The PL/99 compiler is a one-pass syntax directed cross-compiler written in PASCAL for execution on Control Data Corporation 6600 and Cyber computers. The following table presents the primary components of the compiler:

#### PL/99 Compiler Structure

Procedure Name	Function
PARSE	Parses PL/99 source program.
SYNTHESIZE	Semantic actions and object code generation control.

NEXTSYM	Returns next token.
PROC_IDENT	Constructs identifier.
GET_NUMBER	Constructs constant.
GET_MNEMONIC	Controls generation of machine instruction for mnemonics.
NEXTCH	Gets next input character.
ENTERST	Enters identifier and attributes into symbol table.
GET_STORAGE	Allocates storage for variables.
SEARCHST	Searches symbol table for given identifier name.
GEN1,GEN2,..., GEN9,GENCRU1, GENCRU2	Generate machine code for various instruction formats.
ERROR	Writes error messages.
PRINTSYMTAB	Writes output listing of symbol table.
WRITE_LISTING	Writes output listing.
LOAD_VALUES	Initializes storage as per VALUE declaration.
WRITE_OBJECT	Writes object module.
WRITE_VALUES	Writes output listing of VALUE initializations.

A series of secondary procedures supports the functions of the main components listed above. Modularity in the compiler has been emphasized for future expansion or revision.

### 3.3 Memory Organization

A simple linear memory organization has been assumed for the object code generated by the PL/99 compiler. The initial workspace pointer is set at byte address 0000 and the first sixteen words of storage allocated for the workspace. This initial workspace is always provided, but the user may designate any number of additional workspaces through the Load Workspace Pointer Immediate (LWPI) mnemonic. Variables defined in the VAR declaration are allocated storage beginning at byte address hexadecimal 0020. If a procedure is defined, its initial instruction address is adjusted to begin on an even-numbered byte address. The program entry point is the first even-numbered byte address following all declarations and procedure definitions. Subsequent instructions and data are assigned sequential addresses through the remainder of the source program compilation.

### 3.4 Symbol Table

The PL/99 symbol table has been implemented as a PASCAL linked list data structure and contains various attributes



associated with identifiers declared in the PL/99 source program. The structure of the symbol table is as follows:

#### Symbol Table Entry

NEXT	Pointer to next entry in table.
IDENTIFIER	Symbol name.
ARRAY_SIZE	Upper bound plus one for array.
ADDRESS	Relative address of variable identifier or procedure entry point.
INITIALED	Boolean field indicating VALUE initialization.
VALUE_PTR	Pointer to initial value set for this identifier.
IDTYPE	Type associated with this identifier.

The last field, IDTYPE, designates a PASCAL tag field for the symbol table record and provides additional information depending on the type: LABEL types include specification of addresses where forward references to the label were detected; CONST types have the associated integer value specified; and REGISTER types include the integer equivalent of the register name.

During parsing of the LABEL, CONST, VAR, and PROCEDURE declarations, the procedure ENTERST is used to enter each identifier into the symbol table. The table is first

searched to verify that the symbol has not previously been declared, and, if it has, an error message is issued. Otherwise all attributes available are set and dummy values substituted for any fields which will be determined at a later point in the compilation. Note that the register identifiers R0,R1,...,R15 are the first identifiers entered into the symbol table push-down list and the pointer to R15 saved. The programmer thus may choose to declare any of these identifiers as a label, constant, variable, or procedure identifier, and that declaration will always be found first in the symbol table search, thus overriding the predeclaration.

A summary of the symbol table is printed along with the source and object code listings on completion of the PL/99 source program compilation.

### 3.5 Parser Actions

The procedure PARSE is automatically constructed by the compiler writing system and causes tokens to be read and directs the processing of the semantic actions through calls to the procedure SYNTHESIZE. PARSE determines the actions required given a symbol and the current state of the parse.

If a shift is required the state symbol is pushed onto a stack and the next token read. Successive tokens are read if necessary. If a reduction sequence is required, the appropriate semantic actions are invoked as the stack is popped for the required states. This process continues until an accept state is reached or an error state occurs from which recovery is not possible. Successful completion of the parse results in the final accept state.

### 3.6 Semantic Actions

As noted previously, the procedure SYNTHESIZE is generated by the compiler writing system and contains a case statement label for each rule of the input grammar. Each label has appropriate semantic actions specified and thus SYNTHESIZE provides ultimate control over the compiler code generation. The semantic actions are processed at the conclusion of the parsing of a grammar rule as is always the case with a syntax directed compiler. The following sections present discussions of the semantic actions associated with the various components of the PL/99 language.

### 3.6.1 Program Header

Parsing of the program header results in the saving of the program name. This identifier is used during the generation of the object module.

### 3.6.2 Label Declarations

Each declared label is entered into the symbol table and the number of forward references to the label initialized to zero.

### 3.6.3 Constant Declarations

A symbol table entry is generated for each declared constant identifier. The integer value associated with the constant is also entered so that the integer value may be substituted whenever the constant identifier occurs in the source program.

### 3.6.4 Variable Declarations

Parsing of the VAR declarations results in the allocation of memory for each declared variable. The initial workspace for registers R0,R1,...,R15 is allocated storage and the workspace pointer WP is set to 0000 just prior to the parsing of the variable declarations.

#### 3.6.4.1 Simple Types

Variables of simple types BYTE and WORD are entered into the symbol table and allocated single byte or word storage as required. WORD variables have their most significant bits at an even-numbered byte address.

#### 3.6.4.2 Array Types

The array name is entered into the symbol table along with the bound specification (i.e., the largest array index to occur) plus one. The amount of storage required is computed from the bound and allocated. The address entered for the array name corresponds to the first element. Arrays

of type WORD begin on an even-numbered byte address.

### 3.6.5 Value Declarations

Variable and register identifiers may be preset in any order using integer values or constant identifiers. When a value is initialized for an identifier, the symbol table field is flagged, a pointer to the appropriate entry in the list of initial values saved, and the initial value entered into the list. When all or part of an array is initialized, the initialization flag in the symbol table record indicates how many elements of the array have been preset beginning with the first element (index 0). All initial values will subsequently be included in the load module while dummy values will be set for identifiers not initialized. A summary of all preset values is printed as part of the compiler output.

### 3.6.6 Procedure Declarations

Definition of a procedure results first in a symbol table search to verify that the procedure name has not previously been declared. If it has not, the procedure name

is entered along with the entry address for the procedure. If this address is not an even-numbered byte address, the program counter is incremented by one prior to entry in the symbol table. The procedure body is then compiled until the ENDPR reserved word is detected. The ENDPR statement results in the generation of a branch instruction (B) which assumes that workspace register R11 contains the branch address for return to the calling program. Additional discussion of this point is included in the CALL statement section below.

### 3.6.7 Statements

Generation of PL/99 machine instructions for the various source statements which may occur makes use of the information stored in the symbol table during parsing of both declarations and the program body. The semantic actions associated with the various statement forms are discussed below.

#### 3.6.7.1 Labeled Statements

When a labeled statement is encountered, the symbol table is searched for the label name. The location of the label is then entered so that displacements relative to the label may be determined as required for use in GOTO or mnemonic jump statements which reference the label.

### 3.6.7.2 GOTO Statement

A GOTO statement first results in a search of the symbol table for the referenced label. If the label has previously occurred, an unconditional jump instruction (JMP) is generated using a displacement computed from the current program counter contents and the address where the label occurred. Otherwise the GOTO makes a forward reference to the label, and the symbol table entry for the label is updated to indicate the address at which a jump instruction with an incomplete displacement field has occurred. These incomplete displacements are subsequently filled when the label is found and its relative address determined.

It should be noted that the TMS9900 instruction set provides 8-bit displacement fields. Forward jumps must therefore not exceed 127 words and backward jumps must be less than 128 words. An appropriate error message is generated if these restrictions are not satisfied.



### 3.6.7.3 CALL Statement

The occurrence of a CALL statement initiates a symbol table search for the name of the referenced procedure and its entry point. A branch and link instruction (BL) is generated which automatically saves the current program counter contents in workspace register R11 and then branches to the procedure entry point. As previously noted, the ENDPR statement results in a branch instruction which references R11 for the branch address for return to the calling program.

### 3.6.7.4 Mnemonic Statement

Use of a mnemonic statement first results in the determination of the mnemonic employed, and then the processing of the required operands.

The AB, CB, SB, and MOV B mnemonics have been implemented primarily to allow byte operations for register resident quantities where the left or most significant byte is involved. These mnemonics thus expect both destination and source register operands. The SOC, SOCB, SZC, and SZCB

mnemonics are of the same instruction type and require specification of destination and source operands which may be register or memory resident. Operands for COC and CZC mnemonics must be a register identifier for the destination operand and either a register or memory resident identifier for the source.

The B, BL, and BLWP mnemonics require specification of a register or memory resident source operand which will contain the branch address. A memory or register resident source operand must also be specified for CLR, SETO, SWPB, and X.

The jump mnemonics JNC, JNO, JOC, JOP, JH, and JL require a label operand from which the jump displacement may be determined.

The shift mnemonics SLA, SRA, SRC, and SRL require specification of a register identifier which must be followed by the shift count. The shift count must be in the range 0-15 and can be an integer value or a constant identifier.

The immediate operand for the LWPI and LIMM mnemonics must be a constant or constant identifier. STST and STWP require specification of a workspace register identifier,

and RTWP has no operands.

#### 3.6.7.5 Assignment Statements

Various forms of the assignment as well as the availability of both byte and word types results in the generation of a variety of machine instructions. In each case the symbol table is searched to provide the necessary data for a given identifier and the addressing mode is set.

Type checking is performed to ensure that only word-word or byte-byte operands are combined. These restrictions are required because both word and byte instructions are available in the TMS9900 instruction set.

The source statement is also checked to confirm that the proper ordering of source and destination operands has been observed. If operands are out of order, or, if the destination operand on the left-hand-side of an assignment does not correspond to the first operand on the right-hand-side, an instruction format error message is written. Similar messages are issued if register identifiers are not employed for operations where they are required, or if index registers are improperly used.

Eight basic assignment statements may occur where register, variable identifier, and constants (or constant identifiers) are employed in the expression on the right-hand-side of the assignment. Addressing modes are determined from the syntax used in the source statement. The various expression forms and associated machine instructions are as follows:

1. Constant. This syntax results in the generation of a load immediate (LI) instruction.
2. Identifier or + identifier. Either a move (MOV) or move byte (MOVB) instruction is generated.
3. Identifier or - identifier. The negate instruction (NEG) is generated.
4. Identifier + or - constant. In this case the second operand is a constant. If this constant has a value of 1 or 2, then an increment (INC) or decrement (DEC), or increment by two (INCT) or decrement by two (DECT) instruction is generated, respectively. If the constant is 3 or greater, an add immediate (AI) is generated for the + operator, but no subtraction is available in the TMS9900 instruction set.

5. Identifier +,-,\*, or / identifier. The appropriate add (A), add bytes (AB), subtract (S), subtract bytes (SB), multiply (MPY), or divide (DIV) instruction is generated when the second operand is an identifier.
6. Identifier AND or OR constant. The corresponding and immediate (ANDI) or or immediate (ORI) instruction is generated.
7. Identifier XOR identifier. The exclusive or (XOR) instruction is generated.
8. Identifier ABS (identifier) or INV (identifier). The absolute value (ABS) or invert (INV) instruction is generated.

#### 3.6.7.6 Conditions

The evaluation of the appropriate condition is an integral part of the execution of the IF, WHILE, and REPEAT conditional statements. Machine instructions to perform the required compare operations are generated in two forms:

1. Identifier <, <=, =, <>, >, or >= constant. A compare immediate (CI) instruction is generated.
2. Identifier <, <=, =, <>, >, or >= identifier. The appropriate compare words (C) or compare bytes (CB) instruction is generated.

Type checking is performed to verify that the identifier is a register identifier in 1 above, and that the identifier types are equivalent in 2 above.

#### 3.6.7.7 WHILE Statement

Detection of the reserved word DO results in the initial translation of the WHILE statement. The program counter is pushed onto a stack for subsequent reference, and the relational operator specified for the condition is complemented. A jump instruction related to the complemented condition is then generated to jump around the body of the WHILE if the condition is false. Note that at this point, however, the jump displacement is not known. For example, if the original condition specified the = relational operator, then the complemented operator would be <> and a jump not equal (JNE) instruction would be

generated. The body of the WHILE is then compiled until the ENDW reserved word is encountered. This statement list comprising the body of the WHILE may include additional control structures including WHILE statements. At this point the address at the head of the WHILE is popped from the stack and an unconditional jump (JMP) instruction generated to return to the compare at the head of the WHILE. The address of the last statement in the WHILE is now known, and appropriate measures may be taken to update the previously incomplete jump instruction.

#### 3.6.7.7.1 REPEAT Statement

The reserved word REPEAT causes the program counter contents to be pushed onto a stack, and the remainder of the REPEAT statement compiled. As in the case of the WHILE statement, the body of the REPEAT may contain any valid PL/99 statement. The appropriate compare instruction is then generated as specified by the UNTIL part of the REPEAT statement. The stack is popped for the address of the head of the REPEAT, the condition relational operator complemented, and a conditional jump instruction generated to return to the first statement in the body of the REPEAT.

### 3.6.7.8 IF Statement

Two forms of the IF statement are available, and thus additional jump instructions must be generated if the ELSE form is used. For the IF-THEN part the condition is evaluated and the appropriate compare instruction generated. The program counter is then stacked, the relational operator complemented, a conditional jump instruction with an incomplete displacement generated to jump around the THEN part of the IF if the condition is false, and the statement list comprising the THEN compiled. The next symbol is then either the ELSE or ENDIF. In either case, the address previously stacked is popped and used with the current program counter to fix the displacement for the jump around the THEN. If the current symbol is the ENDIF, the statement compilation is complete.

Detection of the ELSE causes the current program counter to be stacked and an unconditional jump instruction (JMP) around the ELSE to be generated with an incomplete displacement field. The ELSE statement list is then compiled until the ENDIF is encountered. The required address is popped from the stack and the previous jump instruction around the ELSE completed.



### 3.7 Code Generation Completion

During the compilation sequence described above, the generated machine code instructions are written in hexadecimal notation to a PASCAL text file. These instructions are complete except for the forward displacement fields of jump instructions associated with the WHILE, IF, GOTO, and mnemonic branch and jump statements. The addresses at which these forward references have occurred have been maintained along with the corresponding displacements as they were determined. These address-displacement pairs are now sorted into increasing relative address order and used to complete the jump instructions as the machine code is transferred one instruction at a time from the original text file to a new text file. The program listing, symbol table, and VALUE summary are then printed and the object module constructed.

### 3.8 Object Module Generation

The PL/99 object module is produced in the Texas Instruments standard object module format [8]. Each object module record may be up to 71 ASCII characters in length,

and consists of a number of tag characters followed by one or two fields of data. The following table summarizes the tag characters and data fields used for the PL/99 object module.

#### OBJECT MODULE TAG CHARACTERISTICS

TAG	First Field	Second Field	Meaning
0	Length of all Relocatable Code	8 character Program Identifier	Program Start
2	Entry Address	None	Relocatable Entry Address
7	Checksum for Current Record	None	Checksum
A	Load Address	None	Relocatable Load Address
B	Data	None	Absolute Data
C	Data	None	Relocatable Data
F	None	None	End of Record

The first tag character of the object module is always 0. Each succeeding tag character follows the last field of the preceding tag character. A record is ended by a tag 7 followed by a checksum and the tag F. The last record of the object module is indicated by a colon in column one.

Note that all numerical values in the module are in hexadecimal notation.

The initial tag 0 is followed by the length of the module relocatable code and an eight character program name. The module entry address is preceded by a tag 2 while the module relocatable load address follows the tag A. Absolute data (i.e., instructions and constants) are preceded by tag B, and a word that contains a relocatable address is preceded by tag C. The checksum is preceded by a tag 7 and consists of the two's complement sum of the 8 bit ASCII values of the characters in a record from the first tag through the 7. The checksum is then followed by the tag F.

The object module contains a data field for every byte address generated during the compilation including the addresses assigned to variable and register identifiers. The VALUE declarations cause the required initial values to be written in the appropriate byte or word data field of the object module while hexadecimal FF or FFFF is used as a dummy value for identifiers not initialized. The final object module is written on the file OBJECT which is printed and which may be saved on disc or tape or punched for future use.

## CHAPTER 4

### Sample Programs

Two sample programs have been written and compiled using the PL/99 compiler. The resulting object module was then executed on a TM 990/100M microcomputer for verification.

#### 4.1 Listing Format

Upon successful completion of the parsing of a PL/99 source program, an output listing is produced which contains the PL/99 source code, addresses and machine code, a summary of the symbol table, a listing of all initial values set using the value declaration, and a listing of the object module. All addresses, machine instructions, and object

module data are in hexadecimal form.

To the left of each PL/99 source line in the program listing is the relative word address of the instruction generated for the source line. Following the address is the machine instruction generated for that source line. If several instructions or data fields were generated for a given line, they are printed on succeeding lines prior to the printing of the next source line. Following the address and machine instruction columns are the source statement line number and the source statement.

The symbol table listing contains the name of each variable, label, constant, and procedure used in the source program as well as the predeclared register identifiers. The type of each identifier is given and its address unless it is a constant identifier for which no storage is allocated. If the variable name is an array, the array size is listed.

The VALUE declaration listing provides a convenient check on program initialization. All initialized variables are printed with the corresponding value. Only that portion of an array initialized is printed.

The object module listing represents the final output of the compiler and is formatted in the manner previously described.

#### 4.2 Program Execution Procedure

The two sample programs have been executed on a Texas Instruments TM 990/100M microcomputer to verify the generated machine code. The object module proper was not used in this process.

A TM 990/301 microterminal connected to the 990/100M was used as the I/O communication interface. This terminal resembles a hand-held calculator and provides direct data entry and communication with the 990/100M. Keys are provided for direct hexadecimal-decimal conversions and vice-versa, and for entering and displaying the program counter (PC), status register (ST), and workspace pointer (WP). CRU data entry and display is similarly available. Data and instructions are entered into specified memory locations using the EMA (enter memory address), EMD (enter into memory address displayed), and EMDI (EMD with autoincrementing of address).

The H/S key allows the suspension of program execution and display of the next address and contents. It also provides single step execution where each keystroke causes the next instruction to be executed and the contents of that address to be displayed.

Pressing the RUN key initiates program execution using the current contents of the WP, PC, and ST registers.

#### 4.3 Description of Sample Programs

The first example program provided, ORDER\_PAIRS, orders each of N data pairs. Two one-dimensional arrays, P and Q, are compared one element at a time, and, if the P-element is greater than the corresponding Q-element, the elements are swapped. On completion of the program execution, all P-elements are less than or equal to all corresponding Q-elements.

The declaration section defines a constant N=20, declares a word variable TEMP and the P and Q arrays, and initializes the P and Q arrays with arbitrary positive integer values.

Program execution begins at address 0066 where register R1 is loaded with value zero. A WHILE statement is then used to step through the P and Q arrays. An IF statement compares corresponding elements of the P and Q arrays using R1 as the array index. If the P-element is greater than the Q-element procedure SWAP is called to interchange the elements using TEMP to temporarily hold the P-value.

The array index R1 is then incremented by 2 to step to the next word in the P and Q arrays. The WHILE statement is then repeatedly executed until all elements have been compared and interchanged if necessary.

The second program, SUM\_COUNT, sums a list of positive integers and counts the number of zeroes encountered. The declaration section defines the constant N=40 and declares the word variables SUM and COUNT. An array NUMBER with 21 elements is declared and then initialized using the VALUE declaration.

The .CLR. mnemonic is used to set SUM, COUNT, and register R2 to 0. R2 is to be used as the index for array NUMBER.

A REPEAT statement is used to examine each element of NUMBER. If the element value is zero, the COUNT total is



incremented by one; otherwise, the SUM is updated. The IF statement which tests each element of NUMBER compares the element with zero. The element thus must first be loaded into a register, R3, to comply with the requirements of the TMS9900 compare immediate instruction format. This cycle defined by the REPEAT continues until all elements of NUMBER have been examined.

Both of these sample programs were successfully executed on the TM 990/100M microcomputer and verified through examination of specific memory addresses after program execution.

ADDRESS	CONTENTS	LINE	SOURCE STATEMENT
1.			PROGRAM ORDER_PAIRS;
2.			(* ORDER EACH OF N DATA PAIRS *)
3.			
4.			
5.			CONST N=20;
6.			
7.			VAR TEMP : WORD;
8.			P,Q : ARRAY [10] OF WORD;
9.			
10.			VALUE P = [0,3,2,4,1,8,11,9,4,1,0],
11.			Q = [2,2,1,0,8,3,2,0,3,2,1];
12.			
13.			PROCEDURE SWAP;
14.			TEMP:=P[R1];
15.			P[R1]:=Q[R1];
16.			Q[R1]:=TEMP;
17.			ENDPR;
18.			
19.			BEGIN
20.			R1:=0;
21.			
22.			WHILE R1<=N DO
23.			
24.			IF P[R1] > Q[R1] THEN CALL SWAP; ENDIF;
25.			
004E	02E0		
0050	00C0		
0052	C821		
0054	0022		
0056	0020		
0058	CF61		
005A	0038		
005C	0022		
005E	C860		
0060	0020		
0062	0038		
0064	045B		
0066	0201		
0068	0000		
006A	0281		
006C	0014		
006E	1508		
0070	8E61		
0072	0022		
0074	0038		
0076	1202		
0078	06A0		
007A	0052		

ADDRESS	CONTENTS	LINE	SOURCE STATEMENT
007C	05C1	26.	R1:=R1+2;
007E	10F5	27.	
		28.	ENDW;
		29.	
		30.	ENDP.

SYMBOL TABLE

IDENTIFIER	TYPE	ADDRESS	ARRAY SIZE
SWAP	PROCEDURE	0052	
Q	WORD	0038	11
P	WORD	0022	11
TEMP	WORD	0020	
N	CONSTANT		
R15	REGISTER	001E	
R14	REGISTER	001C	
R13	REGISTER	001A	
R12	REGISTER	0018	
R11	REGISTER	0016	
R10	REGISTER	0014	
R9	REGISTER	0012	
R8	REGISTER	0010	
R7	REGISTER	000E	
R6	REGISTER	000C	
R5	REGISTER	000A	
R4	REGISTER	0008	
R3	REGISTER	0006	
R2	REGISTER	0004	
R1	REGISTER	0002	
R0	REGISTER	0000	





ADDRESS	CONTENTS	LINE	SOURCE STATEMENT
1.		1.	PROGRAM SUM_COUNT;
2.		2.	
3.		3.	(* SUM A LIST OF NUMBERS AND
4.		4.	COUNT THE NUMBER OF ZEROES *)
5.		5.	
6.		6.	CONST N=40;
7.		7.	
8.		8.	VAR SUM, COUNT : WORD;
9.		9.	NUMBER : APRAY [20] OF WORD;
10.		10.	
11.		11.	VALUE NUMBER = [0,1,2,3,0,4,5,6,0,0,
12.		12.	7,8,9,10,0,11,12,0,0,13];
13.		13.	
14.		14.	BEGIN
15.		15.	.CLR. SUM;
16.		16.	.CLR. R2;
17.		17.	.CLR. COUNT;
18.		18.	
19.		19.	REPEAT
20.		20.	
21.		21.	R3:=NUMBER[R2];
22.		22.	IF R3=0 THEN COUNT:=COUNT+1;
23.		23.	
24.		24.	ELSE
25.		25.	SUM:=SUM+NUMBER[R2];
26.		26.	ENDIF;
27.		27.	R2:=R2+2;
28.		28.	
29.		29.	UNTIL R2=N;
004E	02E0		
0050	00C0		
0052	04E0		
0054	0020		
0056	04C2		
0058	04F0		
005A	0022		
005C	C0E2		
005E	0024		
0060	0283		
0062	0000		
0064	1603		
0066	05A0		
0068	0022		
006A	1003		
006C	A822		
006E	0024		
0070	0C20		
0072	05C2		
0074	0282		

ADDRESS    CONTENTS    LINE    SOURCE STATEMENT

0078       16F1

30.

31.

ENDP.



SYMBOL TABLE

IDENTIFIER	TYPE	ADDRESS	ARRAY SIZE
NUMBER	WORD	0024	21
COUNT	WORD	0022	
SUM	WORD	0020	
N	CONSTANT		
R15	REGISTER	001E	
R14	REGISTER	001C	
R13	REGISTER	001A	
R12	REGISTER	0018	
R11	REGISTER	0016	
R10	REGISTER	0014	
R9	REGISTER	0012	
R8	REGISTER	0010	
R7	REGISTER	000E	
R6	REGISTER	000C	
R5	REGISTER	000A	
R4	REGISTER	0008	
R3	REGISTER	0006	
R2	REGISTER	0004	
R1	REGISTER	0002	
R0	REGISTER	0000	

VALUE DECLARATIONS

NUMBER	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	7	8	9	10	11	12	0	0	0	0	0	0	0	0



## CHAPTER 5

### Concluding Remarks

The PL/99 language and compiler described in the preceding chapters offers a viable alternative for the TMS9900 programmer. Algorithms may be clearly expressed in a form which is superior to the corresponding assembly language program, require less creation and testing effort, and result in machine code which is no less efficient. The structure of PL/99 has been patterned after PASCAL and should be readily learned by most programmers familiar with the TMS9900. The compiler has a modular structure and may be revised or extended in a straightforward manner.

PL/99 in its present form will provide satisfactory solutions for many programming problems, but the absence of certain features will no doubt lead to further revision and

expansion. The inclusion of record types with the additional ability to define arrays of records may be desirable. Similarly, future revisions would probably include user defined types and CASE and FOR statements as in PASCAL. The limited procedure capability could also be improved. It is felt, however, that these deficiencies are not serious at this point in the development of PL/99. A strong framework for evolution of the language has been established.

## APPENDIX

## The PL/99 Grammar

This appendix presents the PL/99 grammar in standard BNF notation.

```

[ 1] < compilation > ::= < program > < eofsym >
[ 2] < program > ::= < programhead > < programbody >
[ 3] < programhead > ::= PROGRAM < identifier > ;
[ 4] < programbody > ::= < labeldecl > < constdecl >
                        < vardecl > < valuedecl >
                        < proceduredecl >
                        BEGIN < statementlist > ENDP .

[ 5] < labeldecl > ::= LABEL < labellist > ;
[ 6] < labeldecl > ::= < epsilon >
[ 7] < labellist > ::= < labelelem >
[ 8] < labellist > ::= < labellist > < labelelem >
[ 9] < labelelem > ::= < identifier >

[10] < constdecl > ::= CONST < constlist > ;
[11] < constdecl > ::= < epsilon >
[12] < constlist > ::= < constelem >
[13] < constlist > ::= < constlist > < constelem >
[14] < constelem > ::= < identifier > = < constant >

[15] < vardecl > ::= VAR < varlist >
[16] < vardecl > ::= < epsilon >
[17] < varlist > ::= < varelem >
[18] < varlist > ::= < varlist > < varelem >
[19] < varelem > ::= < identifierlist > :
                        < simpletype > ;
[20] < identifierlist > ::= < identifier >
[21] < identifierlist > ::= < idlisthead >
                        < identifierlist >
[22] < idlisthead > ::= < identifier > ,
[23] < simpletype > ::= WORD
[24] < simpletype > ::= BYTE

```

```

[ 25] < varelem > ::= < identifierlist > :
                        ARRAY [ < bound > ]
                        OF < simpletype > ;
[ 26] < bound > ::= < constant >

[ 27] < valuedecl > ::= VALUE < valuelist > ;
[ 28] < valuedecl > ::= < epsilon >
[ 29] < valuelist > ::= < valueelem >
[ 30] < valuelist > ::= < valuelist > < valueelem >
[ 31] < valueelem > ::= < identifier > = < values >
[ 32] < values > ::= < constant >
[ 33] < values > ::= [ < arrayvalues > ]
[ 34] < arrayvalues > ::= < constant >
[ 35] < arrayvalues > ::= < arrayvaluehead >
                        < arrayvalues >
[ 36] < arrayvaluehead > ::= < constant > ,

[ 37] < proceduredecl > ::= < procedurelist >
[ 38] < proceduredecl > ::= < epsilon >
[ 39] < procedurelist > ::= < proceduredefn >
[ 40] < procedurelist > ::= < procedurelist >
                        < proceduredefn >
[ 41] < proceduredefn > ::= < procedurehead >
                        < statementlist > < endpr > ;
[ 42] < procedurehead > ::= PROCEDURE < identifier > ;
[ 43] < endpr > ::= ENDPR

[ 44] < statementlist > ::= < statement >
[ 45] < statementlist > ::= < statement > ; < statement >
[ 46] < statement > ::= < label > < unlabstatement >
[ 47] < statement > ::= < unlabstatement >
[ 48] < label > ::= < identifier > :
[ 49] < unlabstatement > ::= < assignstatement >
[ 50] < unlabstatement > ::= GOTO < identifier >
[ 51] < unlabstatement > ::= < epsilon >
[ 52] < unlabstatement > ::= CALL < identifier >
[ 53] < unlabstatement > ::= < mnemonicstatement >
[ 54] < unlabstatement > ::= < strucstatement >
[ 55] < strucstatement > ::= < ifstatement >
[ 56] < strucstatement > ::= < whilestatement >
[ 57] < strucstatement > ::= < repeatstatement >

[ 58] < assignstatement > ::= < leftside > < becomes >
                        < rightside >
[ 59] < leftside > ::= < identifier >
[ 60] < becomes > ::= :=
[ 61] < rightside > ::= < expression >

[ 62] < expression > ::= < constant >
[ 63] < expression > ::= < operand1 >
[ 64] < expression > ::= < addop > < operand1 >

```

```

[ 65] < expression > ::= < operand1 > < arithop >
                                < operand2 >
[ 66] < expression > ::= < operand1 > < boolop >
                                < operand2 >
[ 67] < expression > ::= ABS ( < identifier > )
[ 68] < expression > ::= INV ( < identifier > )

[ 69] < operand1 > ::= < identifier >
[ 70] < operand2 > ::= < identifier >
[ 71] < operand2 > ::= < constant >

[ 72] < arithop > ::= < addop >
[ 73] < arithop > ::= < mulop >
[ 74] < addop > ::= +
[ 75] < arithop > ::= -
[ 76] < mulop > ::= *
[ 77] < mulop > ::= /
[ 78] < boolop > ::= AND
[ 79] < boolop > ::= OR
[ 80] < boolop > ::= XOR

[ 81] < mnemonicstatement > ::= < mnemonic > < operands >
[ 82] < mnemonic > ::= . < identifier > .
[ 83] < operands > ::= < epsilon >
[ 84] < operands > ::= < identifier >
[ 85] < operands > ::= < constant >
[ 86] < operands > ::= < operandhead >
                                < operands >
[ 87] < operandhead > ::= < identifier > ,

[ 88] < ifstatement > ::= IF < condition > < then >
                                < statementlist > ENDIF
[ 89] < ifstatement > ::= IF < condition > < truepart >
                                < statementlist > ENDIF
[ 90] < condition > ::= < operand1 > < relop >
                                < operand2 >
[ 91] < then > ::= THEN
[ 92] < truepart > ::= < then > <statementlist > ELSE

[ 93] < relop > ::= =
[ 94] < relop > ::= <
[ 95] < relop > ::= <=
[ 96] < relop > ::= >
[ 97] < relop > ::= >=
[ 98] < relop > ::= <>

[ 99] < whilestatement > ::= < whilehead >
                                < statementlist > ENDW
[100] < whilehead > ::= WHILE < condition > DO
[101] < repeatstatement > ::= < repeat > < statementlist >

```



< untilpart >

[102] < repeat > ::= REPEAT  
[103] < untilpart > ::= < UNTIL > < condition >

## Notes

[1] Wirth, Nicklaus, "PL360, A Programming Language for the 360 Computers," Journal of the Association for Computing Machinery, Vol. 15, No. 1, January 1968, pp. 37-74.

[2] Bell, D. A. and Wichmann, B. A., "An Algol-like Assembly Language for a Small Computer," Software- Practice and Experience, Vol. 1, 61-72 ( 1972 ).

[3] Pleban, Uwe Frederik, "Design and Implementation of the Structured Assembly Language PL/85," Masters Thesis, University of Kansas, 1976.

[4] Mowday, Barry L., "PL/STAR, A Structured Assembly Language for the CDC STAR 100," Masters Thesis, The College of William and Mary, 1979.

[5] Gray, Lawrence, "What type of programming language best suits OEM design?," EDN, June 20, 1978, pp. 78-84.

[6] Foster, Victor S., "MIDAS : A Mid-level Language for Microprocessors," ( University of Virginia, undated ).

[7] Texas Instruments Incorporated, "990 Computer Family System Handbook," Manual No. 945250-9701, 1976.

[8] Texas Instruments Incorporated, "TM 990/100M Microcomputer User's Guide", August 1977.

## Bibliography

Aho, Alfred V. and Ullman, Jeffrey D. Principles of Compiler Design, Reading, Mass. : Addison-Wesley, 1977.

Bell, D. A. and Wichmann, B. A. "An ALgol-like Assembly Language for a Small Computer." Software- Practice and Experience, Vol. 1, 61-72 ( 1972 ).

Foster, Victor S. "MIDAS: A Mid-level Language for Microprocessors." University of Virginia, undated.

Gray, Lawrence. "What type of programming language best suits OEM design?" EDN, June 20, 1978, pp. 78-84.

Jensen, Kathleen and Wirth, Nicklaus. Pascal User Manual and Report. New York : Spring Verlag, 1974.

Mowday, Barry L. "PL/STAR, A Structured Assembly Language for the CDC STAR 100." Masters Thesis, The College of William and Mary, 1979.

Pleban, Uwe F. "Design and Implementation of the Structured Assembly Language PL/85." Masters Thesis, University of Kansas, 1976.

Texas Instruments Incorporated. "990 Computer Family System Handbook." Manual No. 945250-9701, 1976.

Texas Instruments Incorporated, "TM 990/100M Microcomputer User's Guide", August 1977.

Wirth, Nicklaus. " PL360, A Programming Language for the 360 Computers." Journal of the Association for Computing Machinery 15 (January 1968) 37-74.

## VITA

Kenneth B. Walkley

Born in Montgomery, Alabama, January 27, 1948. Graduated from Jackson High School in Jackson, Alabama in May, 1966; B.A.E., Auburn University, 1971; M.S., Auburn University, 1973. The author has been employed by the LTV Corporation since February 1974, and presently serves as Supervisor of the Aerodynamics Unit at the Hampton Technical Center. He entered the program in Applied Science in January 1976, and has continued his graduate studies as a part-time student.