NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator

Mario R. Barbacci
Department of Computer Science
Carnegie-Mellon University
Pittsburgh Pa.
August 2, 1976

This project is supported in part by the Advanced Research Projects Agency (ARPA) of the Department of Defense, under contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research, and by the National Science Foundation, under grant GJ 32758X.

TABLE OF CONTENTS

SECTION		PAGE
1	Introduction	3
2	Declarations	6
	2.1 Memories and Registers	6
	2.2 Macros	7
	2.3 Identifiers and Constants	8
	2.4 Comments	9
3	Register Transfers	10
	3.1 Structure Selectors	10
	3.2 Transfers	12
	3.3 Shift Operators	12
	3.4 Arithmetic Expressions	14
	3.5 Relational Expressions	15
4	Register Transfer Sequences	17
	4.1 Blocks	17
	4.2 Conditional Statements	17
	4.3 Labelled Statements	19
	4.4 The BAILOUT Operation	19
	4.5 Statement-Lists	20
5	ISPL Programs	21

6 The Compiler Output 24

6.1 Running the Compiler 24

6.2 Example I - Listing 25

6.3 Example I - Symbol Table 25

6.4 Example I - Cross Reference 28

6.5 Example I - Statement Table 29

7 References 32

8 Appendix I - The Minicomputer Listing 33

9 Appendix II - ISPL Reserved Keywords 41

10 Appendix III - The XTOP10.REQ File 42

11 Appendix IV - The Multiplier MACRO10 Format 43

12 Appendix V - The SIMISP.REQ File 46

12.1 The Statement Table 46

12.2 The Symbol Table 47

12.3 Table Diagrams 48

A User's Guide to the ISPL Compiler

Mario R. Barbacci

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

ABSTRACT

The compiler described in this manual will translate programs written in a subset of ISP [Bell, 1971] into register transfer level instructions. The code thus generated could be used for the implementation of wiring list generators, simulators, or other Computer Aided Design applications. This manual describes the syntax and semantics of the language (ISPL) accepted by the compiler.

ACKNOWLEDGEMENTS

The compiler described here is an improved version of an original system implemented by S. Goldman and R. Scroggs. The syntax graph driving the compiler is generated using a program (GRPGEN) written by P. Karlton and R. Scroggs. This version of the manual reflects the modifications and improvements suggested by the users during the preparation of the ISP description of the candidate architectures for the Army/Navy CFA project. Special thanks are due to H. Elovitz (NRL), R. Gordon (NUSC), R. Howbrigg (NUSC), D. Siewiorek (CMU), and S. Zuckerman (NRL).

The Symbolic Manipulation of Computer Descriptions:
ISPL Compiler and Simulator

The Department of Computer Science at Carnegie-Mellon University is currently engaged in a research project exploring the uses of computer description languages in the automatic design of both software and hardware systems. This document describes a language, ISPL, based on the Instruction Set Processor notation of Bell&Newell [Bell,1971]. The language was designed as a tool for the description of instruction sets i.e. the architecture of a computer, and has been used extensively in a design automation project at CMU [Siewiorek,1976] and in the Army/Navy Computer Family Architecture Project.

Traditional computer description languages have been designed primarily for human communication and/or simulation. The SMCD [Barbacci,1974] project has the more ambitious goal of developing design automation tools which would permit the generation of machine-relative software, documentation, hardware modular design, program verification, simulation, and generation of microcode. As in any evolutionary project, preliminary results are necessarily short of the ultimate goal; thus at this point we can present two concrete systems: a compiler and a simulator. A machine-relative compiler-compiler is being investigated by a group under W. Wulf. An automatic generator of hardware modular specifications is being developed by a group under D. Siewiorek and A. Parker. Further studies of computer descriptive languages are being carried out by this author and others.

As indicated above, the systems described in this report have been used as part of the Army/Navy CFA project, sponsored by the Army Electronics Command and the Naval Research Laboratory. Part of the project involved the description, in ISPL, of three commercial architectures: The DEC PDP-11, the IBM /360,370, and the Interdata

8/32. These descriptions were used to collect statistics on the execution of a set of benchmark programs under the ISPL simulator. Although the simulator is not particularly fast, its interactive facilities allow very extrict control and detailed analysis of the register transfer operations being performed during the fetch/decode/execute cycle of the machines. The simulator was not meant to be used as a software development tool (although in fact, some CFA benchmarks for the Interdata 8/32 were debugged under the simulator, it being more accessible at CMU than the real machine), it is rather an Architectural Design tool that allows the user to explore alternative instruction sets and to collect statistics on the performance of the architectures.

Mario R. Barbacci
August 2, 1976

[Barbacci,1974] Barbacci, M.R. and D.P. Siewiorek: "Some Aspects of the Symbolic Manipulation of Computer Descriptions", Department of Computer Science, Carnegie-Mellon University, July, 1974.

[Bell,1971] Bell, C.G. and A. Newell: "Computer Structures: Readings and Examples", McGraw-Hill Book Company, New York, 1971.

[Siewiorek,1976] Siewiorek, D.P., and M.R. Barbacci: "The CMU RT-CAD System: An Innovative Approach to Computer Aided Design", National Computer Conference, NCC76, New York, June 1976.

1. Introduction

The ISP (for Instruction Set Processor) notation was developed for a text [Bell, 1971] to precisely describe the programming level of a computer in terms of its memory, instruction format, data types, data operations, and a set of interpretation rules.

The behavior of a processor is determined by the nature and sequence of its operations. This sequence is given by a set of bits in primary memory (a program) and a set of interpretation rules (usually in the central processor). Thus if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends on the initial conditions and a particular program.

Although the above format is commonly used to describe a digital computer, ISPL is not intended to force the user into a given description style; ISPL can be used to describe register transfer systems in general (digital computers are a subset of such systems, namely those systems that interpret an instruction set).

The subset of ISP implemented by the compiler under discussion contains a number of features that allow the user to describe a wide variety of digital systems: Pseudo register declarations, macros, and compound statements. For efficiency reasons, certain other features described in [Bell, 1971] are not implemented. Among these are: multidimensional memory arrays, parameterized procedures, multiple word access, and scattered bit access. However byte access is implemented.

An ISPL program consists of a description of the memory components (memories and registers) and a description of the behavior of the system. Memory components are defined in ISPL by a name and a description of their structure using brackets to

group the subcomponents along a given dimension. In the current implemention the only subcomponents allowed are memory words and bits (as subcomponents of memory words and registers). The behavior of the system is given by a set of register transfer statements. These statements can be performed in sequence or concurrently. In ISPL, concurrency of actions is the rule rather than the exception, and it is reflected in the use of ";" as a delimiter for lists of concurrent actions. Sequencing is expressed by using the term "next" as a delimiter for lists of sequential actions. Complex concurrent and sequential activities can be described in terms of simpler activities using "next", ";", "(", and ")" in an Algol-like block structure.

The ISPL compiler produces code for an idealized Register Transfer Machine. There are two types of instructions in the RTM: Data and Control instructions. Control instructions are used to sequence the operation of the machine. They contain instructions to START, STOP, BRANCH, DIVERGE into concurrent execution paths, etc. The Data instructions are used to define the Arithmetic and Logical operations among the registers of the machine. They are described in terms of a 3-address format:

destination* ← *source1 operation source2

The RTM code produced by the compiler is presented in two formats. The first format is simply a tabular listing intended primarily for human use. The second format is intended primarily for machine consumption. The human intended tabular representation could be digested by suitable string manipulating programs and stored into a more convenient machine format. Several reasons argued against this approach: depending on the language used, writting these interface programs might involve a non trivial amount of work. Worse yet, any format modification intended to help human readers will render these programs obsolete. The solution adopted was to produce

another copy of the RTM code directly into a machine understandable format. Thus the version of the RTM code intended for machine use is created as a "program" using MACRO-10 as the intermediate language. The format of these programs is described in the appendices.

2. Declarations

There are two types of declarations in ISPL: Memory Declarations (explained in this section) are used to describe the structure of the registers and memories in a machine; Procedure Declarations (explained in later sections) are used to describe the behavior of the functional units in a machine.

2.1. Memories and Registers

Memory components are defined in ISPL by a name and a description of their structure. The number of subcomponents at each level of decomposition is given by a bracketed list of constants, much like an array declaration in Algol.

```
declaration-part ::=          DECLARE declaration-list ERALCED
declaration-list ::=          declaration |
                              declaration-list ; declaration
declaration ::=               memory-declaration |
                              memory-declaration := memory-declaration |
                              procedure-declaration
procedure-declaration ::=     identifier := ( statement-list )
memory-declaration ::=        identifier structure-declaration
structure-declaration ::=     [ word-list ] < bit-list > |
                              [ word-list ] < > |
                              < bit-list > |
                              < >
word-list ::=                 name-list
bit-list ::=                  name-list
name-list ::=                 element-range |
                              name-list , element-range
element-range ::=             number | number : number
```

The declarations are given by a list of individual component declaration using ";" as delimiter. There are two types of memory declarations: 1) A definition of a physical component (physical declaration), and 2) A definition of a logical component (logical declaration) in terms of a previously declared (physical or logical) component. A logical declaration uses the ":=" operator to make an equivalence between two components.

<u>Examples</u>

A<15:0>	Declares A as the name of a register 16 bits wide, named 15, ... 0 (from left to right). The ":" or range operator is used to denote an abreviated list of subcomponent names.
Mp[0:4095]<0:11>	Square brackets are used to specify those dimensions where the accessing is done through some "addressing" (switching) schema. The memory, Mp, consists of 4096 words, each of 12 bits, named (from left to right) 0,1,...11.
R<15,13,11,9:10>	In general, the list of subcomponents along any dimension is given by a list of "names" for the individual subcomponents. Numbers used to name individual elements do not indicate relative position.
Mw[32767:0]<15:0>;	
Mb[65535:0]<7:0>:=Mw[32767:0]<15:0>;	Now the designer can use either Mw (the "word" memory) or Mb (the "byte" memory).

The only concession to the use of numbers as both names and position indicators is by using the range (":") operator, whereby the abreviated list consists of the bounds and all integers in between, with the implication that these consecutive numbers also name consecutive (from left to right) elements. The use of an empty bit-list (<>) indicates a single, unnamed bit.

Undeclared variables or multiple declarations of a variable are, usually, non-fatal errors. The compiler will warn the user it this situation arises. The compiler compares the lengths (Nwords*Nbits) of the left and right hand sides of a logical declaration; if the lengths do not match a warning is issued.

2.2. Macros

A different type of declaration, the MACRO declaration, allows the designer to

abreviate the description by naming often used strings of characters. The macro name can then be used instead of the full string. The format of a macro declaration is the following:

MACRO *identifier* := any-string-of-characters-not-containing-a-$-sign $

Macros are handled in its entirety by the lexical phase, thus the parser never "sees" a macro expansion. Macros can, therefore, be declared at any point in the description, not necessarily in the declaration part, and remain in effect until the end of the description.

Examples

MACRO SIGNBIT := ACC<0> $	The use of SIGNBIT some time later in the description is equivalent to using ACC<0>. Macros are strictly in-line string substitutions.

A macro can be defined in terms of other macros and the user should be careful to avoid a recursive definition which would create a non-terminating string replacement loop.

There are implementation dependent limits on the size of a macro string. If a macro declaration exceeds this limit (1000 characters at present) a warning will be issued. Results might be unpredictible if this situation occurs.

2.3. Identifiers and Constants

An identifier in ISPL is a string of letter, digits, and "."'s, beginning with a letter; the "." is included as an identifier character for readability purposes. In the current implementation only the first 6 characters of an identifier are kept by the compiler. Identifiers must, therefore, differ in the first 6 characters for the compiler to distinguish them. The lexical phase accepts upper and lower case ASCII characters but

they are converted and stored internally as upper case characters. This is another limitation of the implementation.

For readability purposes, identifiers can be followed by a larger and more descriptive version of the identifier. This secondary identifier is treated like an inline comment by the lexical phase. The syntax for this extended identifier use is:

short.identifier\this.is.a.long.identifier

An extended identifier can be appended to a short identifier using the "\" character. Such compound identifiers are valid wherever an identifier is valid. Notice that this is not the same thing as an "alias", as described in the full language [Bell, 1971]. The secondary name is stripped by the lexical phase and the designer must use the primary name for identification purposes.

Constants are strings of digits, interpreted as a number in some base. The default base is 10 (i.e., constants are decimal numbers unless otherwise specified). Constants in base 8 (octal numbers) must be tagged with the character #, as in #100 (decimal 64). Constants in base 2 (binary numbers) must be tagged with the character ', as in '100 (decimal 4). Constants in base 16 (hexadecimal numbers) must be tagged with the character ", as in "A1 (decimal 161). The length of a constant is the minimum number of bits needed to represent it (i.e. leading 0's are stripped). The constant 0 is 1 bit long. The current implementation of the compiler limits constants to a maximum size of 35 bits.

2.4. Comments

Comments can be inserted in a description by preceeding the comment string with the character "!". All characters following the "!" until the end of the line are ignored.

3. Register Transfers

Register Transfers are used to describe the data operations on the memories and registers (the data components) of the system. The syntax of a transfer follows very closely that of most programming languages. The main difference is the use of some special operators and the use of a non-standard operator precedence to accomodate these new operators.

The operators act upon the components of the system by taking the data stored in some components (the inputs), operating (i.e., transforming) on the data, and storing the resulting data in some component (the output).

The data used by the operators is defined in terms of the components that contain it. Since the memories and registers are declared as structured components made out of words and bits, a structure selector is needed in order to access or store data.

3.1. Structure Selectors

```
structure-selector ::=      term | term < selector-range >
term ::=                    number | memory-access | ( expression )
memory-access ::=           identifier |
                            identifier [ arithmetic-expression ]
                            identifier [ element-name ]
element-name ::=            number
selector-range ::=          bit | bit : bit
bit ::=                     number
```

The ***terms*** are the building blocks used in a register transfer expression. A ***term*** can be a constant, a ***memory-access*** (to select data stored in a memory or register), or an ***expression*** in parenthesis (thus allowing large and complex register transfer expressions).

A ***structure-selector*** is used to select parts of a ***term*** (i.e. to select bits of a register, a constant, or an expression). The nature of the register transfer operators requires that the operands be of homogeneous type (i.e., register-like) and length. Thus multiword memories must be accessed using an ***arithmetic-expression*** (the address calculation) enclosed in "[" and "]" to select one and only one word of the array.

The compiler compares the maximum value that the result of an address computation can have with the number of words declared for a memory. If the former exceeds the latter, a warning is issued.

When a ***selector-range*** is applied to a memory or register access term it must use the bit names used in the declaration. When it is applied to other types of term, whose structure has not been declared (i.e., constants and expressions), the bits of the term are implicitly named n, n-1, ..., 1, 0 (from left to right).

Examples

ACC	Select the entire ACC register
Mp[Pc]	Select the word whose address is contained in register Pc
ACC<5>	Select bit 5 of register ACC
Mp[R[INDEX]+DISPLACEMENT]<0>	Select bit 0 of the word whose address is given by the effective address calculation expression
(A<7:0>+B<7:0>)<5:4>	Select the 5th and 6th bits (from the right) of the result of the addition

Attempting to access undeclared bits of a register or memory word will result in a warning message. The compiler will then default the erroneous bit name to the leftmost bit of the declaration. When the selector range of a register or memory word attempts to switch the relative position of two bits, the compiler will switch the

selector range boundaries and issue a warning message. For instance, if X is declared as X<0:5>, both X<2:3> and X<3:2> are equivalent terms but in the second case a warning is issued.

3.2. Transfers

Register transfers are used to modify the contents of the registers and memories. The syntax of a transfer is the following:

transfer ::=	***memory-access* ← *arithmetic-expression*** |
	memory-access* <*selector-range*> ← *arithmetic-expression

The use of a ***selector-range*** on the left hand side of the "←" specifies a partial register (or memory word) modification; the non-selected bits are not disturbed. If the right hand side is shorter than the left hand side, the result is stored right justified and 0's are concatenated to its left to clear the high order bits of the left hand side. If the right hand side is larger than the left hand side truncation of the high order bits will occur (the compiler will issue a warning if this situation occurs).

The right hand side of a transfer is always an ***arithmetic-expression.*** The difference between an ***arithmetic-expression*** and an ***expression*** properly is in the use of relational operators, which are not allowed in the former. We will give more details in the subsection dealing with expressions.

3.3. Shift Operators

shift ::=	***structure-selector*** |
	structure-selector shift-op structure-selector
shift-op :=	↑SL | ↑SR | ↑SL0 | ↑SR0 | ↑SL1 | ↑SR1 | ↑RL | ↑RR |
	concatenation
concatenation ::=	@

A ***shift*** is the first step in the hierarchy of register transfer operations, shift operators have the highest binding power (precedence). A ***shift*** always takes the following form:

left.operand ***shift-op*** right.operand

The meaning of the operators (all of them have the same precedence) is the following:

OPERATOR	MEANING
↑SL	Shift left the left.operand, one position, and insert the (rightmost bit of the) right.operand into the vacant position, dropping the leftmost bit of the left.operand. The length of the result is the same as the length of the left.operand. The result can be stored in a register or used as an operand when building complex expressions. The operator does not modify the left.operand, only the transfer operator ("←") can perform side effects.
↑SR	Shift right the left.operand, one position, and insert the (rightmost bit of the) right.operand into the vacant position, dropping the rightmost bit of the left.operand. The length of the result is the same as the length of the left.operand.
↑SLO	Shift left the left.operand the number of positions indicated by the value of the right.operand inserting 0's in the vacant positions and dropping the righmost bits of the left.operand. The right.operand is treated as an unsigned integer. The result has the same length as the left.operand..
↑SRO	Similar to ↑SLO but shifting right.
↑SL1	Similar to ↑SLO but inserting 1's into the vacant positions.
↑SR1	Similar to ↑SL1 but shifting right.
↑RR	Rotate towards the right the left.operand by the number of positions indicated by the value of the right.operand. The length of the result is the same as the length of the left.operand.
↑RL	Similar to ↑RR but rotating left.
@	Concatenate the left.operand with the right.operand. This operator is included among the shift operators for symmetry reasons. The length of the result is the sum of the lengths of the operands.

3.4. Arithmetic Expressions

complement ::=	*shift* | NOT *shift*
conjunction ::=	*complement* |
	conjunction AND *complement* |
	conjunction EQV *complement*
disjunction ::=	*conjunction* |
	disjunction OR *conjunction* |
	disjunction XOR *conjunction*
negation ::=	*disjunction* |
	- *disjunction* |
	MINUS *disjunction* |
	+ *disjunction*
factor ::=	*negation* |
	factor * *negation* |
	factor / *negation*
sum ::=	*factor* |
	sum - *factor* |
	sum MINUS *factor* |
	sum + *factor*
arithmetic-expression ::=	*sum*

All logical operators (NOT, AND, EQV, OR, and XOR) operate on a bit by bit basis. If the operands have unequal lengths the shortest operand is expanded (on the left) with 0's.

The arithmetic operators, with the exception of MINUS, operate on unsigned (pure magnitude) operands, the MINUS operator assumes a Two's Complement represention with a sign bit in the leftmost position. The main difference is in the padding used to match the length of their operands. The MINUS operator extends the sign of the shortest operand, the other operators use 0 as the padding character.

The length of the result of the infix operators "+", "-", and "MINUS" is one bit larger that the largest operand. The length of the result of the "*" operator is the sum of the lengths of the operands. The length of the result of the "/" operator is the same as the length of the left operand (the dividend).

3.5. Relational Expressions

In order to describe non-trivial systems, ISPL provides certain facilities to control the execution of the transfers. Thus certain transfers may or may not be executed depending on the result of some previous operation. These conditional activities are described in more detail in the following section. Here we are concerned with the basic data operators of the language, among which we include the relational operators used to build conditional expressions.

relation ::=	***arithmetic-expression*** | ***arithmetic-expression relop arithmetic-expression***
relop ::=	EQL | NEQ | LSS | LEQ | GEQ | GTR | TST
expression ::=	***relation***

Relational operators perform a test between their left and right operands. The result for all these operators, with the exception of TST, is a boolean value (TRUE or FALSE) which can be tested by one of the control operations defined in the following section. All relational operators treat the operands as unsigned integers. A 2's complement representation of a negative number will therefore look greater than a positive number of the same length.

The TST operator performs a logical subtraction of its operands and produces a result of 0, 1, or 2, indicating that the left operand is less than, equal to, or greater than the rigth operand, respectively.

Beware that relational operators have less precedence than logical and arithmetic operators, thus, the expression: A LSS B AND C GEQ D is parsed as: A LSS (B AND C) GEQ D which is syntactically incorrect. The proper way of writting the expression is: (A LSS B) AND (C GEQ D)

It was indicated before that the right hand side of a register transfer operation

(←) must be an *arithmetic* expression. This does not allow the use of relational operators. In order to use them on the right hand side of a transfer, the (relational) expression must be enclosed in parenthesis. This in effect transforms the (relational) *expression* into a *term*, a valid *arithmetic-expression*, e.g.:

```
FLAG←(A NEQ B); ! Yields 0 or 1
TVAL←1+(D TST E); ! Yields 1,2, or 3
```

4. Register Transfer Sequences

The behavior of a digital system is described in ISPL by a list of statements. These statements can be build up from register transfers by using two special delimiters to indicate sequential or concurrent execution. Statement lists can be nested using parenthesis to build more complex statement lists. The syntax of the register transfer sequences is as follows:

statement-list ::=	*parallel-statement-list* |
	BAILOUT ***identifier*** |
	statement-list NEXT ***parallel-statement-list***
parallel-statement-list ::=	*labelled-statement* |
	parallel-statement-list ; ***labelled-statement***
labelled-statement ::=	*statement* |
	identifier := ***statement***
statement ::=	*conditional-execute* |
	conditional-decode |
	block |
	transfer |
	identifier
conditional-execute ::=	(IF *expression* => ***statement-list***)
conditional-decode ::=	(DECODE *expression* => ***parallel-statement-list***)
block ::=	(***statement-list***)

4.1. Blocks

Blocks are the simplest building tools to define complicated statements. A ***block*** is a ***statement-list*** enclosed in parenthesis:

(A←0 NEXT A←A OR B[X]<7:0> ; C←C+1)

4.2. Conditional Statements

There are two ways of specifying conditional activities. These are the ***conditional-decode*** and the ***conditional-execute*** statements:

(condition => statement(s)),

where the conditions and their interpretation are as follows:

CONDITION	INTERPRETATION
DECODE ***expression***	The value of the expression is interpreted as an integer and used to select one out of n possible statements, given as a list of alternatives. These alternatives are separated by ";", but in this case they are not considered to be concurrent activities; only one of them will be executed. The statements in the list are numbered 0 through n-1, from left to right. The ith statement is executed if the value of the expression is equal to i.
IF ***expression***	This is a special case of the conditional-decode statement. The statement-list following the => operator is initiated if the logical value of the expression is TRUE, otherwise it is bypassed.

For simplicity, the ***expressions*** used in the ***conditional-execute*** statement do not have to be ***relational-expressions,*** yielding a TRUE or FALSE value. An ***arithmetic-expression*** can be used, with the implication that the result of the expression is tested against 0. The ***statement-list*** is executed if the expression is not equal to 0, it is bypassed otherwise. In other words, the expression is interpreted as (expression NEQ 0). For similar reasons, the ***conditional-decode*** statement accepts a ***relation*** as the conditional expression, with the implication that the logical values FALSE and TRUE are interpreted as the numbers 0 and 1, respectively.

The language does not provide an IF ... THEN ... ELSE type of conditional statement. They are trivially described using a 2-way DECODE statement. The user should be careful to write the alternative statements in the proper order: the 0th case (logical FALSE) first and the 1st case (logical TRUE) second. Thus the statements are reversed from the normal Algol-like order.

Do not forget the ";"s after each alternative, except the last one, of a DECODE statement. A missing ";" in this context is a fatal error that is sometimes detected several lines after the offending alternative. The compiler will complain about a "missing action list".

4.3. Labelled Statements

The statements described above can be identified with a label. This label is used to designate the starting point of the statement. The label of a statement can be used wherever a statement is valid. The interpretation given to the use of a label in the middle of a ***statement-list*** is the following:

1) If the label is associated with a procedure definition, it is interpreted as a call (invocation) of the procedure, unless the invocation occurs inside the definition of the procedure, in which case the invocation is interpreted as a jump to the starting point of the sequence (i.e. there are no recursive calls in ISP).

2) Other invocations are treated as jumps to the starting point of the sequence. In the current implementation, labels (and their sequences) need not be declared before they are used. Thus we can jump forward in the description.

A reserved label, STOP, is predeclared in the compiler. It can be used to indicate a jump to the end of the description.

4.4. The BAILOUT Operation

The BAILOUT operation provides a way to describe the handling of exceptional conditions that might occur during the fetching, decoding, and execution of instructions. This operation is in effect a super RETURN from a procedure when an exceptional condition arises. The BAILOUT operator is used together with the label of the procedure whose context we want to leave, i.e., BAILOUT returns accross multiple levels of (dynamically) nested procedures. For instance:

Examples

```
p1 := (...NEXT (IF x => y←z NEXT BAILOUT p2) NEXT ...)

p2 := (...NEXT p1 NEXT ...)

Main := (...NEXT p2 NEXT ....)
```

In the above example, procedure MAIN invokes procedure P2 which starts execution of procedure P1. At some point, P1 decides that some error has occurred (IF X => ...) and that only MAIN can handle the situation. The effect of "BAILOUT P2" is to terminate the execution of P1 and P2 and return to procedure MAIN, at the point were it invoked P2.

4.5. Statement-Lists

Statements, labelled or otherwise, can be used to describe a list of concurrent activities, a ***parallel-statement-list***, using the ";" as delimiter. ***Parallel-statement-lists*** can be used to build sequences of activities or ***statement-lists***, using the "next" operator as delimiter. Notice that the ";" when used to indicate concurrency has a higher precedence than the "next" used to indicate sequentiality. For instance, in the following statement-list: A←B ; C←D NEXT E←F the transfers A←B and C←D are executed concurrently, and only when they are both completed will the locus of control pass to the next statement, the transfer E←F.

One detail to keep in mind is that ISPL is a statement language, not an expression language (in the BLISS sense). In particular, there is no such thing as an empty or null sequence, thus sequences like: (A←B;) or A←B; NEXT C←D are invalid (the ";" must be followed by a statement). In some cases the compiler is capable of detecting the extra ";" and will eliminate it after warning the user.

5. ISPL Programs

As mentioned in the Introduction, an ISPL description consists of a set of component declarations, together with a description of the behavior of the (main) system:

ispl-program *::=* ***identifier*** *:= (* ***declaration-part statement-list*** *)*

The above syntax indicates that ISPL programs look like labelled blocks, with a declaration-part, local to the body of the block.

EXAMPLE

```
MULT:=
        (DECLARE
        MPD<15:0>;
        P<15:0>;
        C<15:0>;
        STEP := (DECODE P<0> => P←P ↑SR 0; P←(P+MPD)<15:0> ↑SR 0)
         ERALCED
        L0:=    (
                C←8 NEXT
                L1:=    (
                        STEP NEXT
                        C←(C-1)<15:0> NEXT
                        (IF C NEQ 0 => L1)
                        )
                )
        )
```

The first example presents the ISPL description of a simple 8-bit multiplier using the shift-and-add algorithm. The multiplicand resides in the leftmost 8 bits of the MPD register. The multiplier resides in the rightmost 8 bits of the P register. The partial product is developed using all 16 bits of the P register. Additional details about the algorithm can be found in [Bell, 1972].

The description begins with the specification of the label for the program (MULTIPLIER). Labels are used in ISPL to identify activities so that they can be branched to, or used as subroutines.

The program itself is enclosed in parenthesis, and consists of two parts. The declarations and the specification of the behavior. The former are specified as a list of individual component declarations (multiplicand, multiplier/product, and step counter), and one procedure (STEP) which performs the basic multiplication operation, using the reserved identifiers DECLARE and ERALCED as brackets. The specification of the activities of the system is given as a list of two sequential steps. The first step (C←8) initialises the counter and the second is given by a labelled (L1) block of activities. this consists of a sequence of three steps. The first one performs the basic multiplication operation by calling the procedure; the second step decrements the counter; the third step tests the counter to see if the operation has been completed. If the value of the counter has not reached 0 then a jump to the label is indicated by using the label (L1) as an activity. If the counter is 0 then control flows out of the labelled statement and reaches the end of the program.

The basic multiplication operation is described using the DECODE control operation. It implements a 2-way branch depending on the value of the expression P<0>. The alternative paths selected by this operation are given as a list using the ";" as delimiter. The first path (P←P ↑SR 0) is selected if the value of the controlling expression (P<0>) is 0; the second path (P←(P+MPD) ↑SR 0) is selected if the value is 1. The operator ↑SR 0 represents a shift right inserting zero in the vacant position.

EXAMPLE

```
MINI:=(DECLARE  !MEMORY AND REGISTERS
        M[0:#377]<11:0>;        !MAIN MEMORY
        Z<7:0>; !EFFECTIVE ADDRESS REGISTER
        CACC<12:0>;      ! 13 BIT ACCUMULATOR WITH CARRY POSITION
                CARRY.BIT<> := CACC<12>;
                SIGN.BIT<> := CACC<11>;
                ACC<11:0> := CACC<11:0>;
        IR<11:0>;       !INSTRUCTION REGISTER
                OP<11:9> := IR<11:9>;
                I.BIT<> := IR<8>;
                ADDRESS<7:0> := IR<7:0>;
                IO.BITS<7:0> := IR<7:0>;
                UCLASS<> := IR<7>;
        L<7:0>; !RETURN REGISTER
        PC<7:0>;        !PROGRAM COUNTER
        IO.REG<7:0>;    !INPUT-OUTPUT REGISTER
        RUN<>;  !RUN MODE
    ! PROCEDURE TO INCREMENT PROGRAM COUNTER
    INCRPC:=( PC←(PC+1)<7:0>)   ! NOTE THAT PC WILL WRAP
  ERALCED
START:=     (DECODE RUN =>
                STOP;           ! If run=0
            (   IR←M[PC] NEXT INCRPC NEXT
                (DECODE I.BIT => Z←ADDRESS ; Z←M[ADDRESS]<7:0>) NEXT
                (DECODE OP =>   !INSTRUCTION DECODING
                        ACC←ACC AND M[Z];       !AND
                        CACC←ACC + M[Z];                !TAD (SETS CARRY BIT)
                        (M[Z]←(M[Z]+1)<11:0> NEXT (IF M[Z] EQL 0 => INCRPC) );  !ISZ
                        (M[Z]←ACC NEXT ACC←0);  !DCA
                        (L←PC NEXT PC←Z);               !JSR
                        PC←Z;                   !JUMP
                        IO.REG←IO.BITS;         !IOT
                        (DECODE UCLASS =>
                            (   (IF IR<6> => INCRPC) NEXT
                                (IF IR<5> => ACC← NOT ACC) NEXT
                                (IF IR<4> => ACC←0) NEXT
                                (IF IR<3> => CACC←ACC+1) NEXT   ! (SETS CARRY BIT)
                                (IF IR<2> => CACC←ACC-1) NEXT   ! (SETS CARRY BIT IF BORROW)
                                (IF IR<1> => ACC← ACC ↑SR0 1) NEXT
                                (IF IR<0> => ACC← ACC ↑SL0 1)   ); !END OF UCLASS=0
                            (   (IF IR<6> => INCRPC) NEXT
                                (IF IR<5> => PC←L) NEXT
                                (IF IR<4> => PC←CACC<7:0>) NEXT
                                (IF IR<3> => RUN←0) NEXT
                                (IF (IR<2> AND SIGN.BIT) OR
                                        (IR<1> AND (ACC EQL 0)) OR
                                        (IR<0> AND (NOT SIGN.BIT)) => INCRPC)
                            )
                        )       !END OF UCLASS DECODING
                )       !END OF INSTRUCTION DECODING
            )   !END OF RUN=1 MODE
            ) NEXT      !END OF INSTRUCTION CYCLE
        START
)
```

6. The Compiler Output

The compiler produces a listing file (with extension LST) and an "object code" file (with extension RTM). The latter extension stands for Register Transfer Machine. In other words, the compiler produces code for some idealized machine which executes register transfer operations.

6.1. Running the Compiler

The following example shows a typical execution. The actual calling procedure may change from installation to installation. When the compiler starts executing it prompts the user for the ISP source file name. If there are any error messages they are printed on the user's terminal as well as in the listing file. When the compilation is done (the compiler types messages indicating the current phase it is executing) it automatically calls the MACRO10 assembler and passes to it the name of the RTM file. At the end of the assembly the user should have the following files (assume the ISP source is called X.ISP): X.LST, X.RTM, X.REL, as well as the X.ISP file, of course.

```
ru isp
Input File: mult.isp

ISP COMPILER Thursday 29 Jul 76 23:42:13 MULT.ISP[N655MB25]     PAGE 1

Parse Completed.
Optimization Completed.
Semantic Check and Output Follows
ISP:NO ERRORS DETECTED
23:43:57
MACRO:  .MAIN

EXIT
```

6.2. Example I - Listing

The listing file reproduces the ISPL source program together with any warning and error messages. The listing file is organized in 4 parts: 1) The listing proper, 2) A cross-reference listing indicating the places in the RTM object code were the registers, memories, and labels are being used, 3) A symbol table listing containing all the user and system declared entities, together with their attributes, and 4) A statement table listing containing a readable version of the RTM object code.

```
[001] MULT:=
[001]    (DECLARE
[002]    MPD<15:0>;
[002]    P<15:0>;
[002]    C<15:0>;
[002]    STEP := (DECODE P<0> => P←P ↑SR 0; P←(P+MPD)<15:0> ↑SR 0)
[003]     ERALCED
[003]    L0:=    (
[003]            C←8 NEXT
[004]            L1:=    (
[004]                    STEP NEXT
[005]                    C←(C-1)<15:0> NEXT
[005]                    (IF C NEQ 0 => L1)
[005]                    )
[005]            )
[005]    )
[005]
```

6.3. Example I - Symbol Table

The compiler produced symbol table for the multiplier example is shown below. There is an entry (1 line) for each user or compiler declared component. These include memory components, labels, and constants. The INDEX column indicates the position in the symbol table of the entity. This index is used to represent the variables in the statement table.

```
ISP COMPILER Thursday 29 Jul 76 22:09:14 TEMP.TMP[N655MB25]      PAGE 2

    0     0 10000000    0    0    0    0         0 'e      '
    1     2 10000000    0    0    0   20         1 'C      '<0(17):17(0)>
    2     4 10000100    0    0   16    0         0 'L0     '
    3     4 10000100    0    0   21    0         0 'L1     '
```

```
 4    2 10000000   0   0   0  20        1 'MPD    '<0(17):17(0)>
 5    4 10000100   0   0   1   0        0 'MULT   '
 6    2 10000000   0   0   0  20        1 'P      '<0(17):17(0)>
 7    4 10001100   0   0   3   0        0 'STEP   '
10    4 10000101   0   0  35   0        0 'STOP   '
11    3 10000001   0   0   0   1        0        0
12    5 10000001   0   0   0   1        0    0,, 0
13    3 10000001   0   0   0   1        0        1
14    3 10000001   0   0   0   4        0       10
15    5 10000001   0   0   0  20        0   17,, 0
16   10 10000001   0   0   0   1        0 '%TFAAA'
17    7 10000001   0   0   0  21        0 '%TRAAA'
20    7 10000001   0   0   0  20        0 '%TRAAB'
21    7 10000001   0   0   0   1        0 '%TRAAC'
```

The TYPE column describes the type of "variable" stored in a given entry of the symbol table. The valid types are: Memory Array (TYPE=1), Register (2), Constant (3), Label (4), Mask (5), Flag (6), Temporary register (7), and Temporary flag (10). The last two are used for compiler declared variables (for instance, temporary registers are declared in order to store partial results when evaluating expressions).

The FLAGS field contains information used by the compiler. It is displayed as part of the output mainly for debugging purposses (i.e. they show the status of the symbol table entry).

The DEF field is used to store a pointer to an associated symbol table entry. It is used when a memory component, say a register, is defined in terms of a previously declared memory component. For instance, we can declare:

INSTRUCTION.REGISTER<15:0>;

OP.CODE<3:0> := INSTRUCTION.REGISTER<15:12>;

In the symbol table listing, the DEF field for OP.CODE will point to a pseudo register declaration entry, corresponding to INSTRUCTION.REGISTER<15:12>. The DEF field for the latter will point to the main declaration of INSTRUCTION.REGISTER<15:0>. If INSTRUCTION.REGISTER had been mapped on top of another register or memory

declaration, the DEF fields will chain these definitions. (DEF defines a chain of definitions, the last entry of which is always the main declaration).

The LBL (LaBeL) field associates with every user declared label, an integer used by the compiler. This integer constitutes an internal label.

The BCNT and WCNT (Bit CouNT and Word CouNT, respectively) indicate the number of bits and words for each memory and constant. (The count is given as an octal number).

The PNAME (Print NAME) contains an identifier for each entry. For user declared variables and labels it contains the identifier used in the program (truncated to six characters). Constants are identified by their numeric value (octal). Masks are represented as a pair of octal numbers. These indicate the left and rightmost bit positions of the mask with respect to the right edge of the word (for instance, a binary mask like 00011000 will appear as 4,,3). System declared registers and flags are given compiler generated names.

The last field of the symbol table, WORDS;BITS, contains the list of subcomponents for each user declared memory or register. The list contains the bit (word) names given in the declaration as well as the internal bit (word) names generated and used for the compiler. The compiler generates a position dependent internal bit (word) name which can be used to generate the proper subcomponent accessing code. These position identifiers are indicated in parenthesis, next to the user specified bit (or word) names.

6.4. Example I - Cross Reference

```
INDEX   VAR      STATEMENTS


    1   'C      '
           20       24       25       26
    2   'L0     '
           33
    3   'L1     '
           30       32
    4   'MPD    '
           11
    5   'MULT   '
           34
    6   'P      '
            5        7       11       13
    7   'STEP   '
           15       23
   10   'STOP   '
   11           0
            7       13       26
   12      0,,  0
            5
   13           1
   14          10
           20
   15     17,,  0
           12       25
   16   '%TFAAA'
           26       27
   17   '%TRAAA'
           11       12       24       25
   20   '%TRAAB'
           12       13
   21   '%TRAAC'
            5        6
```

6.5. Example I - Statement Table

```
INDEX  LABEL    FLAG    OPCODE  DEST     SOURCE1 SOURCE2 MERGE   PATHS

    0              0    'START '                          16
    1  'MULT  '    1    'SMERGE'
       (     5)
    2              0    'ISP    '                          2
    3  'STEP  '    1    'SMERGE'
       (     7)
    4              0    'ISP    '                          3
    5              0    'RBYTE ''%TRAAC''P      '  0,, 0
                                (    21)(     6)(    12)
    6              0    'BRANCH'        '%TRAAC'          14   7,11
                                        (    21)
    7              0    'RSHFT ''P      ''P      '      0
                                (     6)(     6)(    11)
   10              0    'JOIN  '                          14
   11              0    'ADD   ''%TRAAA''P      ''MPD    '
                                (    17)(     6)(     4)
   12              0    'RBYTE ''%TRAAB''%TRAAA'  17,, 0
                                (    20)(    17)(    15)
   13              0    'RSHFT ''P      ''%TRAAB'       0
                                (     6)(    20)(    11)
   14              0    'SMERGE'
   15              0    'RETURN'        'STEP   '          3
                                        (     7)
   16  'L0    '    1    'SMERGE'
       (     2)
   17              0    'ISP    '                          4
   20              0    'MOVE  ''C      '      10
                                (     1)(    14)
   21  'L1    '    1    'SMERGE'
       (     3)
   22              0    'ISP    '                          5
   23              0    'CALL  '        'STEP   '          3
                                        (     7)
   24              0    'DECR  ''%TRAAA''C      '
                                (    17)(     1)
   25              0    'RBYTE ''C      ''%TRAAA'  17,, 0
                                (     1)(    17)(    15)
   26              0    'NEQ   ''%TFAAA''C      '       0
                                (    16)(     1)(    11)
   27              0    'IF    '        '%TFAAA'          31   31,30
                                        (    16)
   30              4    'JOIN  '        'L1     '         21
                                        (     3)
   31              0    'SMERGE'
   32              0    'NOOP  '        'L1     '         21
                                        (     3)
   33              0    'NOOP  '        'L0     '         16
                                        (     2)
   34              0    'NOOP  '        'MULT   '          1
                                        (     5)
   35  'STOP  '    1    'STOP  '
       (    10)
```

The LABEL field is used to identify the individual statements.

The FLAGS field, as in the symbol table, is used internally by the compiler. In this particular example, the only flag shown indicates whether the label associated with the instruction was declared by the user (1) or by the compiler(0).

The OPR field contains the operation name. The meaning of most operations should be obvious from their names. Data operations are described as a 3-address assembly-like instruction. The source operands and the destination operand are indicated by their index into the symbol table (columns SRC1, SRC2, and DEST). The RBYTE operation is used to extract a byte from a register. The interpretation of the operation is the following: DESTINATION←SOURCE1<SOURCE2> where destination and source1 are of type register and source2 is a mask. Other non-obvious data operations (not shown in the example) are:

WBYTE (DESTINATION<SOURCE1>←SOURCE2),

READ (DESTINATION←SOURCE1[SOURCE2]), and

WRITE (DESTINATION[SOURCE1]←SOURCE2).

The RTM code uses at most three operands, thus an ISP statement like: A←B[C]<1> compiles into two RTM operations. The first is a READ operation that loads a (compiler generated) temporary register with B[C]. The second operation is a RBYTE that extracts bit 1 of this temporary (the position of this bit is deduced from the declaration of B) and stores it into A. Control operations are slightly more complex. Serial Merge (SMERGEOP) operations are used as merging points for non-concurrent sequences. Parallel merge (PMERGEOP) operations are used as merging points for concurrent sequences. Branch (BRANCHOP) operators select one out of many alternative sequences. These sequences are identified by a list of the labels of their

entry points, given in the same order as the conditional statement in the original ISP. Diverge (DIVERGEOP) operations are used to initiate simultaneous, concurrent paths. These paths are, as in the branch operations, indicated by a list of labels.

Branch and Diverge operations also specify the label of the statement following the alternative or concurrent paths. That statement is the "merge" point for the different paths.

The join (JOIN) operator is used as an unconditional jump statement. It generally appears as the last statement of a path, and jumps to the appropriate merging point (a serial or parallel merge). The NOOP operation is used as a control operation. It is generated by the compiler to indicate the end of a block. The statement points to the entry point of the block.

7. References

[Barbacci, 1973] Barbacci, M. R. and D. P. Siewiorek: "The Automated Exploration of the Design Space for Register Transfer (RT) Systems". First Annual Symposium on Computer Architecture, University of Florida, Gainesville, Florida, December 1973.

[Bell, 1971] Bell, C. G. and A. Newell: "Computer Structures: Readings and Examples". McGraw Hill Book Company, New York, 1971.

[Bell, 1972] Bell, C. G., J. Grason, and A. Newell: "Designing Computers and Digital Systems". Digital Press, Digital Equipment Corporation, 1972.

8. Appendix I - The Minicomputer Listing

```
[001] MINI:=(DECLARE    !MEMORY AND REGISTERS
[002]    M[0:#377]<11:0>;         !MAIN MEMORY
[002]    Z<7:0>; !EFFECTIVE ADDRESS REGISTER
[002]    CACC<12:0>;      ! 13 BIT ACCUMULATOR WITH CARRY POSITION
[002]             CARRY.BIT<> := CACC<12>;
[002]             SIGN.BIT<> := CACC<11>;
[002]             ACC<11:0> := CACC<11:0>;
[002]    IR<11:0>;        !INSTRUCTION REGISTER
[002]             OP<11:9> := IR<11:9>;
[002]             I.BIT<> := IR<8>;
[002]             ADDRESS<7:0> := IR<7:0>;
[002]             IO.BITS<7:0> := IR<7:0>;
[002]             UCLASS<> := IR<7>;
[002]    L<7:0>; !RETURN REGISTER
[002]    PC<7:0>;         !PROGRAM COUNTER
[002]    IO.REG<7:0>;     !INPUT-OUTPUT REGISTER
[002]    RUN<>;  !RUN MODE
[002]      ! PROCEDURE TO INCREMENT PROGRAM COUNTER
[002]      INCRPC:=( PC←(PC+1)<7:0>)      ! NOTE THAT PC WILL WRAP
[003]    ERALCED
[003] START:=       (DECODE RUN =>
[004]             STOP;           ! If run=0
[004]         (   IR←M[PC] NEXT INCRPC NEXT
[004]             (DECODE I.BIT => Z←ADDRESS ; Z←M[ADDRESS]<7:0>) NEXT
[004]             (DECODE OP =>  !INSTRUCTION DECODING
[004]                     ACC←ACC AND M[Z];       !AND
[004]                     CACC←ACC + M[Z];                !TAD (SETS CARRY BIT)
[004]                     (M[Z]←(M[Z]+1)<11:0> NEXT (IF M[Z] EQL 0 => INCRPC) );  !ISZ
[004]                     (M[Z]←ACC NEXT ACC←0);  !DCA
[004]                     (L←PC NEXT PC←Z);               !JSR
[004]                     PC←Z;                   !JUMP
[004]                     IO.REG←IO.BITS;         !IOT
[004]                     (DECODE UCLASS =>
[004]                         (   (IF IR<6> => INCRPC) NEXT
[004]                             (IF IR<5> => ACC← NOT ACC) NEXT
[004]                             (IF IR<4> => ACC←0) NEXT
[004]                             (IF IR<3> => CACC←ACC+1) NEXT   ! (SETS CARRY BIT)
[004]                             (IF IR<2> => CACC←ACC-1) NEXT   ! (SETS CARRY BIT IF BORROW)
[004]                             (IF IR<1> => ACC← ACC ↑SR0 1) NEXT
[004]                             (IF IR<0> => ACC← ACC ↑SL0 1)   ); !END OF UCLASS=0
[004]                         (   (IF IR<6> => INCRPC) NEXT
[004]                             (IF IR<5> => PC←L) NEXT
[004]                             (IF IR<4> => PC←CACC<7:0>) NEXT
[004]                             (IF IR<3> => RUN←0) NEXT
[004]                             (IF (IR<2> AND SIGN.BIT) OR
[004]                                     (IR<1> AND (ACC EQL 0)) OR
[004]                                     (IR<0> AND (NOT SIGN.BIT)) => INCRPC)
[004]                         )
[004]                     )       !END OF UCLASS DECODING
[004]             )       !END OF INSTRUCTION DECODING
[004]         )   !END OF RUN=1 MODE
[004]         ) NEXT      !END OF INSTRUCTION CYCLE
[004]    START
[004] )
```

```
INDEX TYPE    FLAGS  DEF  BLK  LBL BCNT       WCNT PNAME   WORDS;BITS:NAME(POSITION)
    0    0 10000000    0    0    0    0          0 'e      '
    1    2 10010000    4    0    0   14          1 'ACC    '<0(13):13(0)>
    2    2 10010000   16    0    0   10          1 'ADDRES'<0(7):7(0)>
    3    2 10100000    5    0    0    1          1 'CACC   '<13(0)>
    4    2 10100000    5    0    0   14          1 'CACC   '<0(13):13(0)>
    5    2 10000000    0    0    0   15          1 'CACC   '<0(14):14(0)>
    6    2 10100000    5    0    0    1          1 'CACC   '<14(0)>
    7    2 10010000    6    0    0    1          1 'CARRY.'
   10    2 10010000   15    0    0    1          1 'I.BIT '
   11    4 10001100    0    0    3    0          0 'INCRPC'
   12    2 10010000   20    0    0   10          1 'IO.BIT'<0(7):7(0)>
   13    2 10000000    0    0    0   10          1 'IO.REG'<0(7):7(0)>
   14    2 10100000   17    0    0    3          1 'IR     '<11(2):13(0)>
   15    2 10100000   17    0    0    1          1 'IR     '<10(0)>
   16    2 10100000   17    0    0   10          1 'IR     '<0(7):7(0)>
   17    2 10000000    0    0    0   14          1 'IR     '<0(13):13(0)>
   20    2 10100000   17    0    0   10          1 'IR     '<0(7):7(0)>
   21    2 10100000   17    0    0    1          1 'IR     '<7(0)>
   22    2 10000000    0    0    0   10          1 'L      '<0(7):7(0)>
   23    1 10000000    0    0    0   14        400 'M      '[377(377):0(0)]<0(13):13(0)>
   24    4 10000100    0    0    1    0          0 'MINI   '
   25    2 10010000   14    0    0    3          1 'OP     '<11(2):13(0)>
   26    2 10000000    0    0    0   10          1 'PC     '<0(7):7(0)>
   27    2 10000000    0    0    0    1          1 'RUN    '
   30    2 10010000    3    0    0    1          1 'SIGN.B'
   31    4 10000100    0    0   10    0          0 'START '
   32    4 10000101    0    0  161    0          0 'STOP   '
   33    2 10010000   21    0    0    1          1 'UCLASS'
   34    2 10000000    0    0    0   10          1 'Z      '<0(7):7(0)>
   35    5 10000001    0    0    0    1          0    0,, 0
   36    3 10000001    0    0    0    1          0        0
   37    3 10000001    0    0    0    1          0        1
   40    5 10000001    0    0    0    1          0    1,, 1
   41    5 10000001    0    0    0    1          0    2,, 2
   42    5 10000001    0    0    0    1          0    3,, 3
   43    5 10000001    0    0    0    1          0    4,, 4
   44    5 10000001    0    0    0    1          0    5,, 5
   45    5 10000001    0    0    0    1          0    6,, 6
   46    5 10000001    0    0    0   10          0    7,, 0
   47    5 10000001    0    0    0   14          0   13,, 0
   50   10 10000001    0    0    0    1          0 '%TFAAA'
   51    7 10000001    0    0    0   14          0 '%TRAAA'
   52    7 10000001    0    0    0   15          0 '%TRAAB'
   53    7 10000001    0    0    0   14          0 '%TRAAC'
   54    7 10000001    0    0    0    1          0 '%TRAAD'
   55    7 10000001    0    0    0    1          0 '%TRAAE'
   56    7 10000001    0    0    0    1          0 '%TRAAF'
   57    7 10000001    0    0    0   11          0 '%TRAAG'
   60    7 10000001    0    0    0    1          0 '%TRAAH'
```

```
INDEX   VAR      STATEMENTS

   1    'ACC   '
           27      32      46      47      67      73      77     103
                  107     113     140
   2    'ADDRES'
           20      22
   5    'CACC  '
           32      77     103     130
   7    'CARRY.'
  10    'I.BIT '
           17
  11    'INCRPC'
            7      16      43      63     120     151
  12    'IO.BIT'
           56
  13    'IO.REG'
           56
  17    'IR    '
           15      61      65      71      75     101     105     111
                  116     122     126     132     136     141     145
  22    'L     '
           51     124
  23    'M     '
           15      22      26      31      34      37      40      46
  24    'MINI  '
          160
  25    'OP    '
           25
  26    'PC    '
            5       6      15      51      52      54     124     130
  27    'RUN   '
           12     134
  30    'SIGN.B'
          137     144
  31    'START '
          156     157
  32    'STOP  '
           13
  33    'UCLASS'
           60
  34    'Z     '
           20      23      26      31      34      37      40      46
                   52      54
  35      0,, 0
          111     145
  36          0
           41     140
  37          1
          107     113
  40      1,, 1
```

```
        105       141
41        2,,  2
        101       136
42        3,,  3
         75       132
43        4,,  4
         71       126
44        5,,  5
         65       122
45        6,,  6
         61       116
46        7,,  0
          6        23      130
47       13,,  0
         36
50     '%TFAAA'
         41        42      140      142
51     '%TRAAA'
         22        23       26       27       31       32       34       35
                   40       41
52     '%TRAAB'
         35        36
53     '%TRAAC'
         36        37
54     '%TRAAD'
         61        62       65       66       71       72       75       76
                  101      102      105      106      111      112      116      117
                  122      123      126      127      132      133      136      137
                  143      147      150
55     '%TRAAE'
        141       142      143
56     '%TRAAF'
        144       146
57     '%TRAAG'
          5         6
60     '%TRAAH'
        145       146      147
```

```
INDEX  LABEL    FLAG    OPCODE  DEST      SOURCE1 SOURCE2 MERGE   PATHS

   0               0    'START '                            10
   1  'MINI  '     1    'SMERGE'
      (     24)
   2               0    'ISP    '                            2
   3  'INCRPC'     1    'SMERGE'
      (     11)
   4               0    'ISP    '                            3
   5               0    'INCR   ''%TRAAG''PC      '
                                 (     57)(     26)
   6               0    'RBYTE ''PC      ''%TRAAG'   7,, 0
                                 (     26)(     57)(     46)
   7               0    'RETURN'          'INCRPC'           3
                                          (     11)
  10  'START '     1    'SMERGE'
      (     31)
  11               0    'ISP    '                            4
  12               0    'BRANCH'          'RUN    '         155  13,15
                                          (     27)
  13               0    'JOIN   '         'STOP   '         161
                                          (     32)
  14               0    'JOIN   '                           155
  15               0    'READ   ''IR      ''M       ''PC      '
                                 (     17)(     23)(     26)
  16               0    'CALL   '         'INCRPC'           3
                                          (     11)
  17               0    'BRANCH'          'I.BIT '          24  20,22
                                          (     10)
  20               0    'MOVE   ''Z       ''ADDRES'
                                 (     34)(      2)
  21               0    'JOIN   '                           24
  22               0    'READ   ''%TRAAA''M       ''ADDRES'
                                 (     51)(     23)(      2)
  23               0    'RBYTE ''Z       ''%TRAAA'   7,, 0
                                 (     34)(     51)(     46)
  24               0    'SMERGE'
  25               0    'BRANCH'          'OP     '         154  26,31,34,46,51,54,56,
               60
                                          (     25)
  26               0    'READ   ''%TRAAA''M       ''Z       '
                                 (     51)(     23)(     34)
  27               0    'AND    ''ACC     ''ACC     ''%TRAAA'
                                 (      1)(      1)(     51)
  30               0    'JOIN   '                           154
  31               0    'READ   ''%TRAAA''M       ''Z       '
                                 (     51)(     23)(     34)
  32               0    'ADD    ''CACC    ''ACC     ''%TRAAA'
                                 (      5)(      1)(     51)
  33               0    'JOIN   '                           154
  34               0    'READ   ''%TRAAA''M       ''Z       '
                                 (     51)(     23)(     34)
  35               0    'INCR   ''%TRAAB''%TRAAA'
                                 (     52)(     51)
  36               0    'RBYTE ''%TRAAC''%TRAAB'  13,, 0
```

```
                         (    53)(    52)(    47)
37          0   'WRITE ''M      ''Z      ''%TRAAC'
                         (    23)(    34)(    53)
40          0   'READ  ''%TRAAA''M      ''Z      '
                         (    51)(    23)(    34)
41          0   'EQL   ''%TFAAA''%TRAAA'        0
                         (    50)(    51)(    36)
42          0   'IF    '        '%TFAAA'              44  44,43
                                (    50)
43          0   'CALL  '        'INCRPC'               3
                                (    11)
44          0   'SMERGE'
45          0   'JOIN  '                             154
46          0   'WRITE ''M      ''Z      ''ACC    '
                         (    23)(    34)(     1)
47          0   'CLEAR ''ACC    '
                         (     1)
50          0   'JOIN  '                             154
51          0   'MOVE  ''L      ''PC     '
                         (    22)(    26)
52          0   'MOVE  ''PC     ''Z      '
                         (    26)(    34)
53          0   'JOIN  '                             154
54          0   'MOVE  ''PC     ''Z      '
                         (    26)(    34)
55          0   'JOIN  '                             154
56          0   'MOVE  ''IO.REG''IO.BIT'
                         (    13)(    12)
57          0   'JOIN  '                             154
60          0   'BRANCH'        'UCLASS'             153  61,116
                                (    33)
61          0   'RBYTE ''%TRAAD''IR     '    6,, 6
                         (    54)(    17)(    45)
62          0   'IF    '        '%TRAAD'              64  64,63
                                (    54)
63          0   'CALL  '        'INCRPC'               3
                                (    11)
64          0   'SMERGE'
65          0   'RBYTE ''%TRAAD''IR     '    5,, 5
                         (    54)(    17)(    44)
66          0   'IF    '        '%TRAAD'              70  70,67
                                (    54)
67          0   'NOT   ''ACC    ''ACC    '
                         (     1)(     1)
70          0   'SMERGE'
71          0   'RBYTE ''%TRAAD''IR     '    4,, 4
                         (    54)(    17)(    43)
72          0   'IF    '        '%TRAAD'              74  74,73
                                (    54)
73          0   'CLEAR ''ACC    '
                         (     1)
74          0   'SMERGE'
75          0   'RBYTE ''%TRAAD''IR     '    3,, 3
                         (    54)(    17)(    42)
```

```
INDEX  LABEL    FLAG    OPCODE   DEST     SOURCE1 SOURCE2 MERGE   PATHS

  76               0    'IF     '         '%TRAAD'          100  100,77
                                          (    54)
  77               0    'INCR  ''CACC   ''ACC    '
                                 (     5)(      1)
 100               0    'SMERGE'
 101               0    'RBYTE ''%TRAAD''IR      '    2,, 2
                                 (    54)(     17)(     41)
 102               0    'IF     '         '%TRAAD'          104  104,103
                                          (    54)
 103               0    'DECR  ''CACC   ''ACC    '
                                 (     5)(      1)
 104               0    'SMERGE'
 105               0    'RBYTE ''%TRAAD''IR      '    1,, 1
                                 (    54)(     17)(     40)
 106               0    'IF     '         '%TRAAD'          110  110,107
                                          (    54)
 107               0    'RSHFT0''ACC    ''ACC    '        1
                                 (     1)(      1)(     37)
 110               0    'SMERGE'
 111               0    'RBYTE ''%TRAAD''IR      '    0,, 0
                                 (    54)(     17)(     35)
 112               0    'IF     '         '%TRAAD'          114  114,113
                                          (    54)
 113               0    'LSHFT0''ACC    ''ACC    '        1
                                 (     1)(      1)(     37)
 114               0    'SMERGE'
 115               0    'JOIN  '                            153
 116               0    'RBYTE ''%TRAAD''IR      '    6,, 6
                                 (    54)(     17)(     45)
 117               0    'IF     '         '%TRAAD'          121  121,120
                                          (    54)
 120               0    'CALL   '         'INCRPC'            3
                                          (    11)
 121               0    'SMERGE'
 122               0    'RBYTE ''%TRAAD''IR      '    5,, 5
                                 (    54)(     17)(     44)
 123               0    'IF     '         '%TRAAD'          125  125,124
                                          (    54)
 124               0    'MOVE  ''PC     ''L      '
                                 (    26)(     22)
 125               0    'SMERGE'
 126               0    'RBYTE ''%TRAAD''IR      '    4,, 4
                                 (    54)(     17)(     43)
 127               0    'IF     '         '%TRAAD'          131  131,130
                                          (    54)
 130               0    'RBYTE ''PC     ''CACC   '    7,, 0
                                 (    26)(      5)(     46)
 131               0    'SMERGE'
 132               0    'RBYTE ''%TRAAD''IR      '    3,, 3
                                 (    54)(     17)(     42)
 133               0    'IF     '         '%TRAAD'          135  135,134
                                          (    54)
 134               0    'CLEAR ''RUN
                                 (    27)
```

```
INDEX  LABEL    FLAG    OPCODE  DEST     SOURCE1 SOURCE2 MERGE    PATHS

  135              0    'SMERGE'
  136              0    'RBYTE ''%TRAAD''IR     '    2,, 2
                                (    54)(    17)(    41)
  137              0    'AND    ''%TRAAD''%TRAAD''SIGN.B'
                                (    54)(    54)(    30)
  140              0    'EQL    ''%TFAAA''ACC     '        0
                                (    58)(     1)(    36)
  141              0    'RBYTE ''%TRAAE''IR     '    1,, 1
                                (    55)(    17)(    40)
  142              0    'AND    ''%TRAAE''%TRAAE''%TFAAA'
                                (    55)(    55)(    58)
  143              0    'OR     ''%TRAAD''%TRAAD''%TRAAE'
                                (    54)(    54)(    55)
  144              0    'NOT    ''%TRAAF''SIGN.B'
                                (    56)(    30)
  145              0    'RBYTE ''%TRAAH''IR     '    0,, 0
                                (    60)(    17)(    35)
  146              0    'AND    ''%TRAAH''%TRAAH''%TRAAF'
                                (    60)(    60)(    56)
  147              0    'OR     ''%TRAAD''%TRAAD''%TRAAH'
                                (    54)(    54)(    60)
  150              0    'IF     '        '%TRAAD'            152  152,151
                                         (    54)
  151              0    'CALL   '        'INCRPC'              3
                                         (    11)
  152              0    'SMERGE'
  153              0    'SMERGE'
  154              0    'SMERGE'
  155              0    'SMERGE'
  156              0    'NOOP   '        'START '             10
                                         (    31)
  157              0    'JOIN   '        'START '             10
                                         (    31)
  160              0    'NOOP   '        'MINI  '              1
                                         (    24)
  161  'STOP  '    1    'STOP   '
       (    32)
```

9. Appendix II - ISPL Reserved Keywords

The following keywords and identifiers are reserved in the language:

AND
BAILOUT
DECLARE
DECODE
DELAY (not described in this manual)
EQL
EQV
ERALCED
GEQ
GTR
IF
LSS
LEQ
MACRO
MINUS
NEQ
NEXT
NOT
OR
STOP
TST
WAIT (not described in this manual)
XOR

```
XTTESTOP=#200,
XTEQLOP=#201,
XTNEQOP=#202,
XTLSSOP=#203,
XTLEQOP=#204,
XTGEQOP=#205,
XTGTROP=#206,
XTMOVEOP=#210,
XTCLEAROP=#211,
XTNOOP=#212,
XTWBYTEOP=#213,
XTRBYTEOP=#214,
XTREADOP=#220,
XTWRITEOP=#221,
XTLROTOP=#226,
XTRROTOP=#227,
XTNOTOP=#230,
XTINCROP=#231,
XTDECROP=#232,
XTLSHFTOP=#233,
XTRSHFTOP=#234,
XTANDOP=#235,
XTOROP=#236,
XTXOROP=#241,
XTEQVOP=#242,
XTADDOP=#243,
XTSUBOP=#244,
XTLSHFT1OP=#245,
XTRSHFT1OP=#246,
XTLSHFT0OP=#247,
XTRSHFT0OP=#250,
XTCONCOP=#251,
XTNEGOP=#252,
XTSUBTWOOP=#253,          !Two's Complement Subtract
XTMULTOP=#300,
XTDIVOP=#301,
XTIFOP=#350,
XTRETURNOP=#351,
XTISPOP=#352,
XTPJOINOP=#353,
XTBAILOUTOP=#361,
XTCALLOP=#363,
XTJOINOP=#365,
XTBRANCHOP=#371,
XTDIVERGEOP=#372,
XTSMERGEOP=#373,
XTPMERGEOP=#374,
XTSTARTOP=#376,
XTSTOPOP=#377,
```

11. Appendix IV - The Multiplier MACRO10 Format

Another version of the RTM code intended for machine consumption consists of a MACRO10 program in which all the information in the symbol and statement tables is encoded as MACRO10 statements (all of which are in fact, data definition statements).

In order to understand the RTM file (the ISP and listing files associated with this example were described previously, in the section describing the compiler output), the reader should have a working knowledge of BLISS10, enough to understand the SIMISP.REQ file describing the structure of the MACRO10 statements. The SIMISP.REQ file is given after the example.

```
;ARF ISP COMPILER - JUNE 1976
        TWOSEG
        INTERN  SYTABL,STTABL,SYTOP,STTOP,ISPTIT
        INTERN  ISPFNM,ISPEXT,ISPDAT,ISPTIM,ISPPPN,ISPVER
        RELOC 400000
%B0005: EXP     0,17,17,0,-1
%B0003: EXP     0,17,17,0,-1
%B0001: EXP     0,17,17,0,-1
$00025: EXP     27,26
$00006: EXP     7,11
        RELOC   0
SYTABL:
        BYTE    (9)0,200(18)0,0,0,0,0(36)'@      ',0
        BYTE    (9)2,200(18)0,0,20,0,%B0001(36)'C      ',1
        BYTE    (9)4,204(18)0,17,0,0,0(36)'L1     ',0
        BYTE    (9)2,200(18)0,0,20,0,%B0003(36)'MPD    ',1
        BYTE    (9)4,204(18)0,1,0,0,0(36)'MULT  ',0
        BYTE    (9)2,200(18)0,0,20,0,%B0005(36)'P      ',1
        BYTE    (9)4,214(18)0,3,0,0,0(36)'STEP  ',0
        BYTE    (9)4,205(18)0,32,0,0,0(36)'STOP  ',0
        BYTE    (9)5,201(18)0,0,1,0,0(36)       0,0
        BYTE    (9)3,201(18)0,0,1,0,0(36)       0,0
        BYTE    (9)3,201(18)0,0,1,0,0(36)       1,0
        BYTE    (9)3,201(18)0,0,4,0,0(36)      10,0
        BYTE    (9)5,201(18)0,0,20,0,0(36)17000000,0
        BYTE    (9)10,201(18)0,0,1,0,0(36)'%TFAAA',0
        BYTE    (9)7,201(18)0,0,21,0,0(36)'%TRAAA',0
        BYTE    (9)7,201(18)0,0,20,0,0(36)'%TRAAB',0
        BYTE    (9)7,201(18)0,0,1,0,0(36)'%TRAAC',0
STTABL:
        BYTE    (9)0,376(18)2361(12)0,0,0(18)0,16,0,0
        BYTE    (9)1,373(18)5261(12)0,0,0(18)0,0,0,4
        BYTE    (9)0,352(18)4301(12)0,0,0(18)0,2,0,0
        BYTE    (9)1,373(18)5261(12)0,0,0(18)0,0,0,6
        BYTE    (9)0,352(18)4301(12)0,0,0(18)0,3,0,0
```

```
        BYTE    (9)0,214(18)11221(12)20,5,10(18)0,0,0,0
        BYTE    (9)0,371(18)13461(12)0,20,0(18)2,14,$00006,0
        BYTE    (9)0,234(18)7070(12)5,5,11(18)0,0,0,0
        BYTE    (9)0,365(18)2341(12)0,0,0(18)0,14,0,0
        BYTE    (9)0,243(18)7121(12)16,5,3(18)0,0,0,0
        BYTE    (9)0,214(18)11221(12)17,16,14(18)0,0,0,0
        BYTE    (9)0,234(18)7070(12)5,17,11(18)0,0,0,0
        BYTE    (9)0,373(18)5261(12)0,0,0(18)0,0,0,0
        BYTE    (9)0,351(18)1421(12)0,6,0(18)0,3,0,0
        BYTE    (9)0,210(18)10021(12)1,13,0(18)0,0,0,0
        BYTE    (9)1,373(18)5261(12)0,0,0(18)0,0,0,2
        BYTE    (9)0,352(18)4301(12)0,0,0(18)0,4,0,0
        BYTE    (9)0,363(18)2401(12)0,6,0(18)0,3,0,0
        BYTE    (9)0,232(18)10024(12)16,1,0(18)0,0,0,0
        BYTE    (9)0,214(18)11221(12)1,16,14(18)0,0,0,0
        BYTE    (9)0,202(18)7042(12)15,1,11(18)0,0,0,0
        BYTE    (9)0,350(18)6241(12)0,15,0(18)2,27,$00025,0
        BYTE    (9)4,365(18)2341(12)0,2,0(18)0,17,0,0
        BYTE    (9)0,373(18)5261(12)0,0,0(18)0,0,0,0
        BYTE    (9)0,212(18)212(12)0,2,0(18)0,17,0,0
        BYTE    (9)0,212(18)212(12)0,4,0(18)0,1,0,0
        BYTE    (9)1,377(18)1441(12)0,0,0(18)0,0,0,7
SYTOP:  EXP     20
STTOP:  EXP     32
ISPTIT: ASCIZ   'Friday 23 Jul 76 19:22:58 TEST.ISP[N655M825]'
ISPFNM: SIXBIT  'TEST  '
ISPEXT: SIXBIT  'ISP   '
ISPDAT: ASCIZ   '23 Jul 76'
ISPTIM: ASCIZ   '19:22:58'
ISPPPN: EXP     32540,334165
ISPVER: EXP     0,0,0,0
        END
```

The MACRO10 program starts by declaring certain symbols to be accessible to separately compiled modules. This is done with the INTERN MACRO10 operator. The symbols in question are the base address for the symbol and statement tables and the number of entries in each table (actually the index of the last entry, the first entry has index 0). The user therefore can access the symbol table entries between SYTABL[0,<fieldname>] and SYTABL[@SYTOP,<fieldname>] and the statement table entries between STTABL[0,<fieldname>] and STTABL[@STTOP,<fieldname>].

The MACRO10 program is divided in two segments, the high segment contains the bit and word lists of the symbol table, as well as the label lists of the statement table. The low segment contains the symbol and statement tables properly.

The bit and word lists are declared as a list of expressions, using the EXP MACRO10 operation, each element of the list takes a full word on the PDP-10. Each bit and word list is identified by a label of the form: %Bnnnn for bit lists and %Wnnnn for word lists were nnnn is the index of the symbol table associated with the bit/word list. Every element of a bit/word list appears as a pair of consecutive elements in the EXP statement. The first (odd) element is the bit/word name. The second (even) element is the bit/word position. The bit/word list ends with a -1 as a bit/word name element.

The statement table label lists appear as lists of expressions, again using the EXP operation. These lists are identified by a label of the form $nnnnn were nnnnn is the index of the statement table associated with the label list. There is no need for a special list terminator, the statement table entry contains a count or vector length for its label list, if any.

12. Appendix V - The SIMISP.REQ File

12.1. The Statement Table

```
MACRO
        STFLAGS=0,27,9$,           !ASSORTED FLAGS FOR THE STATEMENT
        STOPERATION=0,18,9$,       !OPERATION CODE. SEE XTOP.REQ
        STARFOP=0,0,18$,           ! ARF OPERATION CODE
        STDESTINATION=1,24,12$,    !DESTINATION VARIABLE SYMBOL TABLE INDEX
        STSOURCE1=1,12,12$,        !SOURCE1 VARIABLE SYMBOL TABLE INDEX
        STSOURCE2=1,0,12$,         !SOURCE2    "        "      "     "
        STSCOUNT=2,18,18$,         !NUMBER OF ELEMENTS IN STSLIST.
        STLABEL=3,0,18$,           !SYMBOL TABLE INDEX OR 0.
        STMERGELABEL=2,0,18$,      !LABEL OF THE ASSOC. MERGE STATEMENT FOR
                                   !XTDIVERGE ,XTBRANCH  AND XTCALL OPS.
                                   !LABEL OF ASSOC. STATEMENT FOR XTCALLOP.
        STSLIST=3,18,18$;          !POINTER TO VECTOR OF SUCCESSOR STATEMENTS.
                                   !STSUCSTRUCT IS MAPPED ONTO THE VECTOR

BIND    !THE STTABLE FLAGS
        STUSERLAB=1↑0,             !STATEMENT LABEL WAS DECLARED BY USER
        STBREAK=1↑1,               !BREAK FLAG. SIMULATOR BREAKS AFTER FLAGGED
                                   !STATEMENTS ARE EXECUTED
        STTRACE=1↑2,               !TRACE FLAG. SIMULATOR WILL PRINT VARIABLES
                                   ! AFTER EXECUTION.
        STRECORD=1↑3,              !RECORD THE SIMULATED TIME OF EACH EXECUTION
        STIGNORED=1↑4,             !FLAGS DIVERGE,PMERGE AND ASSOC. JOINS AS
                                   !DELETED STATEMENTS!!
        STOPAQUE=1↑5,              ! DISABLES READ/WRITE/ACCESS TALLY
        STETCETC=0;                !ADD ANY OTHER FLAGS YOU LIKE

BIND
        STENTRYSIZE=4;             !4 WORDS/ENTRY

STRUCTURE STSTRUCTURE[INDEX,WORD,P,S]=(.STSTRUCTURE+.INDEX*STENTRYSIZE+.WORD)<.P,.S>;

EXTERNAL STSTRUCTURE STTABLE;   !THE STATEMENT TABLE
EXTERNAL STTOP;          !THE INDEX OF THE LAST STTABLE ENTRY (STARTING FROM 0)


MACRO
        STSUCLABEL=18,18$,         !THE SUCCESSOR LABEL
        STSUCINDEX=0,18$;          !THE SUCCESSOR INDEX

STRUCTURE STSUCSTRUCT[WORD,P,S]=(.STSUCSTRUCT+.WORD)<.P,.S>;
```

12.2. The Symbol Table

```
MACRO
        SYTYPE=0,27,9$,              !THE ENTRY TYPE (1=MEMORY,2=REGISTER,3=CONSTANT,
                                     !4=LABEL, 5=MASK, 6=FLAG, 7=TREGISTER, #10=TFLAG)
        SYFLAGS=0,18,9$,                    !ASSORTED FLAGS FOR THE ENTRY
        SYDEFINITION=0,0,18$,        !INDEX OF ASSOCIATED ENTRY. USED FOR REG-DEFINITIONS
        SYLABEL=1,18,18$,            !INTERNAL STATEMENT TABLE INDEX FOR ENTRIES OF TYPE=4
        SYBITCNT=1,0,18$,            !NUMBER OF BITS/WORD OR CONSTANT LENGTH
        SYWRDPTR=2,18,18$,           !POINTER TO WORD LIST (ONLY FOR TYPE=1)
        SYBITPTR=2,0,18$,            !POINTER TO BIT LIST (ONLY FOR TYPE=1 OR 2)
        SYPNAME=3,0,36$,             ! SIXBIT STRING FOR VARIABLES, VALUE FOR
                                     !CONSTANTS AND MASKS (LEFTBIT,,RIGHTBIT)
        SYWRDCNT=4,0,36$;            !NUMBER OF WORDS (ONLY FOR TYPE=1)

BIND
        SYENTRYSIZE=5,               !5 WORDS/ENTRY
        SYSYSTEMVAR=1↑0,             !SYSTEM DECLARED VAR. (TYPE=3,5,7,#10)
        SYBREAK=1↑1,                 !BREAK FLAG. USED ONLY FOR LABELS.
        SYTRACE=1↑2,                 !TRACE FLAG. SIMULATOR TELLS AFTER VARIABLE IS WRITTEN INTO.
        SYPRIMARY=1↑4,               !INDICATES VAR. IS LEFT HALF OF REG-DEFINITION
        SYSECONDARY=1↑5,             !INDICATES VAR. IS RIGHT HALF OF REG-DEFINITION
        SYBITADDRESS=1↑6,            !INDICATES STORAGE IS BIT ADDRESABLE
        TYPEMEMORY=1,                !FOR SYTYPE ABOVE
        TYPEREGISTER=2,              ! "     "      "
        TYPECONSTANT=3,              ! "     "      "
        TYPELABEL=4,                 ! "     "      "
        TYPEMASK=5,                  ! "     "      "
        TYPEFLAG=6,                  ! "     "      "
        TYPETREGISTER=7,             ! "     "      "
        TYPETFLAG=8;                 ! "     "      "

STRUCTURE SYSTRUCTURE[INDEX,WORD,P,S]=(.SYSTRUCTURE+.INDEX*SYENTRYSIZE+.WORD)<.P,.S>;

EXTERNAL SYSTRUCTURE SYTABLE;   !THE SYMBOL TABLE
EXTERNAL SYTOP; !THE NUMBER OF ENTRIES -1 (I.E. MAX INDEX)

STRUCTURE IVECTOR[NDX]=[1](..IVECTOR+.NDX)<0,36>;

EXTERNAL ISPTIT,ISPFNAM,ISPEXT,ISPPPN,ISPDAT,ISPTIM,ISPVER;
```

12.3. Table Diagrams

```
%

-----------------------------------------------------------
| STFLAGS       | STOPERATION   |         STARFOP         |
-----------------------------------------------------------
|      STDESTINATION |   STSOURCE1      |   STSOURCE2     |
-----------------------------------------------------------
|     STSCOUNT                  |   STMERGELABEL          |
-----------------------------------------------------------
|    STSLIST                    |  STLABEL                |
-----------------------------------------------------------

-------------------------------
|  STSUCINDEX  |  STSUCLABEL  |        1ST SUCCESSOR
-------------------------------
|          ............       |
-------------------------------
|  STSUCINDEX  |  STSUCLABEL  |        "STSCOUNT"TH SUCCESSOR
-------------------------------
|        -1                   |
-------------------------------

------------------------------------------------------
| SYTYPE      | SYFLAGS    |        SYDEFINITION      |
------------------------------------------------------
| SYLABEL                  | SYBITCNT                 |
------------------------------------------------------
| SYWRDPTR                 | SYBITPTR                 |
------------------------------------------------------
|                 SYPNAME                             |
------------------------------------------------------
|                 SYWRDCNT                            |
------------------------------------------------------

--------------------------------
| FIRST WORD/BIT NAME          |
--------------------------------
| FIRST WORD/BIT POSITION      |
--------------------------------
|          ..........          |
--------------------------------
| LAST WORD/BIT NAME           |
--------------------------------
| LAST WORD/BIT POSITION       |
--------------------------------
|            -1                |
--------------------------------
%
```

A User's Guide to the ISPL Simulator

Mario R. Barbacci
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

TABLE OF CONTENTS

	SECTION	PAGE
1	Introduction	3
2	From ISPL to RTM and Beyond	4
3	The Command Language	6
3.1	START and CONTinue	7
3.2	EXIT	7
3.3	READ and DUMP	7
3.4	ECHO and DECHO	8
3.5	RADIX	8
3.6	CTR, SETCTR, and OUTCTR	8
3.7	OPAQUE and DOPAQUe	9
3.8	VALUE and SETVALue	9
3.9	TRACE, UTRACE, DTRACE, and TELLTRace	10
3.10	BREAK, DBREAK, and TELLBReak	10
3.11	SBREAK, DSBREAK, and TELLSBreak	10
3.12	ICONNEct and OCONNEct	10
3.13	HELP	11
4	Storage Mapping	12
4.1	Allowable Types of Mapping	14
5	Examples	15

5.1 Linking the Compiler Output with the Simulator 15

5.2 Running the Simulator 16

5.3 Executing Selected Procedures 18

5.4 Reading Command Files 19

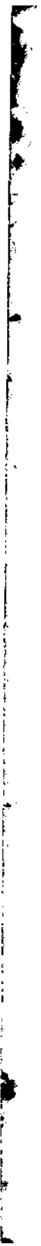
ISPL Simulator: User's Manual

Abstract

The simulator described in this manual will interpret the output of the ISPL compiler, the RTM code, thus allowing the users a generalized computer architecture simulation facility. This manual describes the commands available to the users.

Acknowledgements

The ISP simulator is a much improved version of a primitive system developed by S. Rodkey at CMU during the spring of 1975. The system was modified and expanded by Greg Lloyd of the Naval Research Laboratory during the Fall of 1975. The system was further enhanced by the author during the Winter and Spring of 1976. Many commands and features were added to the system as part of the Army/Navy CFA project. Special thanks are due to the users of the system for their comments and suggestions, among them: H. Elovitz (NRL), R. Gordon (NUSC), R. Howbrigg (NUSC), D. Siewiorek (CMU), and S. Zuckerman (NRL).



1. Introduction

The ISPL compiler translates Computer Architecture (Instruction Set) Descriptions written in a subset of ISP [Bell71] into instructions for an idealized Register Transfer Machine (RTM) which can perform the primitive Register Transfer Operations needed to fetch, decode, and execute instructions. The ISP simulator is in effect an implementation of the Register Transfer Machine.

Some effort has been put into isolating the user from the low level detail of the RTM code. Under normal circumstances, the user will interact with the simulator using the names of registers, memories, procedures, etc, as declared in the ISPL description.

The simulator follows the convention of the ISPL compiler with regard to number representation, it uses an unsigned (pure magnitude) representation. Internally, the simulator uses multiple precision operations on the PDP-10 to execute the data operations and transfers. A current implementation limitation sets a limit of 140 bits for the length of the variables used in the register transfer operations (beware that the ISPL compiler will allow the user to declare registers and memories of arbitrary length - the simulator will warn the user if any attempt is made to operate on variables larger than 140 bits).

Although concurrency is easily described in ISPL, the simulator makes no attempt to provide this facility. It will execute concurrent operations in sequence and the user should avoid writting order-dependent parallel ISP statements.

2. From ISPL to RTM and Beyond

The process of obtaining a running simulator given a syntactically correct ISP description is rather simple. The ISPL compiler, in the abscence of serious errors, will produce a MACRO10 program containing the RTM object code. This program should be assembled in order to produce a relocatable PDP-10 binary file. This process is also handled by the ISPL compiler (i.e. it will generate the RTM file and then invoke the MACRO10 assembler). At the end of the compilation the user has the following files (assume that the original ISP files was X.ISP):

X.ISP (The source file)
X.LST (The listing file, described in the ISPL compiler manual)
X.RTM (The object file, described in the ISPL compiler manual)
X.REL (The relocatable binary version of the X.RTM file)

At this point you can get rid of the X.LST and X.RTM files, as far as the simulator is concerned, they are not needed at all. Hold onto the X.REL file for dear life, unless cycles are cheap at your installation and can afford to run the ISPL compiler as often as you please....

The simulator consists of a group of (currently 7) binary files that must be linked, using one of the standard PDP-10 CUSPs, with the X.REL file. Once this is done, you can save the core image and you are all set to go. The exact procedure might change from installation to installation, depending on whether you use LOAD or LINK10.

A typical procedure might look like:

```
EXECUTE X.REL,@ISPSIM.CMD

<or alternative, if you have the LINK-10 loader in your system:>

R LINK
*x.REL
*@ISPSIM.CMD
*<any switches you want>
*/SSAVE x
```

*/GO

The above sequence will produce two files: X.SHR and X.LOW. These are your ISPL description compiled, linked, saved, and ready to run:

RU x < and off you go!, good luck!>

3. The Command Language

The simulator accepts a small number of commands, using a fixed format:

<keyword> <parameter> <parameter> ....

Only one command is accepted per line. Commands might be typed, in upper or lower case, directly from the user's terminal or can be retrieved from command files (the latter can be done recursively, up to 16 levels of nested command files). Comments can be inserted in the command stream by typing a "!" followed by any arbitrary string. The command scaner will ignore anything between the "!" and the end of the line. Most parameters represent ISPL variable names or numeric values. The latter can be typed in several modes (Binary, Octal, Decimal, and Hexadecimal) and there are facilities to set up a proper default value of the type-in type-out radix.

All variables, labels, and constants defined in the ISP source program have activity counters associated with them. This allows the user to collect statistical data when running benchmark programs under the simulator. There are commands to clear, preset, and interrogate the value of these counters.

The command language include a group of commands to trace variables, start, break, and continue a simulation run, as well as commands to set and interrogate the values of the register and memories of the target machine.

When the simulator is running and the user suspects an infinite loop of instructions, typing a $ (Altmode) will break the execution. Actually, any type ahead will produce an interruption. $ is the preferred mode.

3.1. START and CONTinue

START <label> is the command used to begin the simulation of an ISP procedure or main program. <label> is the name of a procedure declared in the ISP description. The START command is valid only at the top level of simulation. Thus, after a breakpoint in the simulation the user must use the command CONT to proceed.

3.2. EXIT

EXIT is the command used to finish a simulation run. It allows an orderly return to the PDP-10 monitor. EXIT closes the files that might have been created with the OCONNECT command. Typing ↑C will return to the monitor but CONNECTed files will be lost.

3.3. READ and DUMP

READ <dev:filename.ext[ppn]> allows the user to specify a file containing simulation commands. Essentially, READ substitutes the user terminal with the file and proceeds to read and execute commands until the end of the file is found, at which point the user terminal is again the command input device. Defaults are DSK (device), SIM (extension) and current user's PPN. Command files can contain comments. A comment is anything between a ! and the end of a line.

DUMP is used to save the status of a simulation run. DUMP creates a file containing the values of each variable (if non-zero), trace/break flags, read/write counters, etc. The file created by DUMP can be read by the READ command, thus allowing a simple way of reinitializing a simulation at the point the DUMP command was issued.

3.4. ECHO and DECHO

ECHO and DECHO are commands used to set an internal flag that controls the ECHOing of the commands being read from a command file onto the user terminal. After the ECHO command is issued, the execution of a READ command will type onto the user"s terminal the command lines as they appear in the command file. DECHO disables this type-out. ECHO and DECHO can be issued from inside the command file thus allowing a selective type-out.

3.5. RADIX

RADIX <base> is used to set the numeric base to be used for typing in and out. <base> is one of the following strings: BINARY, OCTAL, DECIMAL, or HEX. If base is ommited the command simply types the name of the current base without altering it. The current base setting might be bypassed on input by prefixing the constant with one of the following: ' (binary), # (octal) or " (hex). Regardless of the current radix, HEX constants which begin with a letter MUST be prefixed with " (this is a requirement that will be lifted in a future release).

3.6. CTR, SETCTR, and OUTCTR

CTR <name> displays the value of the counter(s) associated with <name>. These counters are tagged with R, W, or L to indicate whether they are the Read, Write, or Label count respectively. SETCTR <name> <readcounter> <writecounter> allows the user to specify the setting of these counters. If <name> is a label, then the <readcounter> plays the role of label count. If the <...counter> values are ommitted they default to 0. Instead of <name> the user may specify ALL and the command is

applied to all the variables and labels. All read/write counts are expressed in terms of 8-bit bytes. Thus, reading a 16 bit register increments the R counter by 2. The register lengths are rounded up to the next multiple of 8 before incrementing the counter: A 19 bit register counts as 3.

OUTCTR <filename.ext[ppn]> is a subset of the DUMP command. It creates a file (default extension CTR) with the values of of all non-zero counters.

3.7. OPAQUE and DOPAQUe

OPAQUE <label-list> and DOPAQUE <label-list> are used to inhibit or enable the variable and label activity counters. The parameters to these two commands are labels or procedure names. If a procedure is OPAQUEd then no activity counts are incremented during its execution. The DOPAQUE command re-enables the activity counting. These two commands affect only those procedures named in the parameter list. Procedures called by OPAQUEd or DOPAQUEd procedures are not affected.

3.8. VALUE and SETVALue

VALUE and SETVALue are the commands used to set and interrogate the contents of the ISP variables. The valid formats are:

VALUE <regname> (displays the value of a single register)

VALUE <memname> [<fromword> {: <toword>}] (displays the values stored in a memory).

SETVAL <regname> = <value> (stores <value> into the register)

SETVAL <memname> [<fromword>] = <value-list> (stores into the memory. If more than one value is specified, they are stored in succesive memory positions, starting at <fromword>).

3.9. TRACE, UTRACE, DTRACE, and TELLTRace

{TRACE | UTRACE | DTRACE} <variable-list> are the commands used to enable or disable the tracing of variables during the simulation. If the identifier ALL is specified instead of a variable list, the command applies to all variables. TRACE and UTRACE differ in that the former applies to all variables (including compiler declared temporary registers and flags) while the latter only applies to user declared variables (registers and memories). DTRACE is used to disable the tracing.

TELLTRace will type on the user's terminal the list of variables currently being traced.

3.10. BREAK, DBREAK, and TELLBReak

{BREAK | DBREAK} <label-list> are the commands used to enable or disable the setting of breakpoints during the simulation. The parameters are either ISP procedure names or labels. TELLBR displays on the user's terminal the list of breakpoint names.

3.11. SBREAK, DSBREAK, and TELLSBreak

These commands are similar to BREAK, DBREAK, and TELLBReak but instead of using ISP labels as parameters they take RTM statement numbers. Thus allowing a finer degree of control on the placement of the breakpoints. These commands are not particularly useful for the normal user, who should not be concerned with the RTM code.

3.12. ICONNEct and OCONNEct

ICONNEct <identifier>,<channel-number>,<variable-name>

OCONNEct <identifier>,<channel-number>,<variable-name>

These commands are used to "connect" ISP variables to PDP-10 ASCII files which will act as potentially infinite sources/sinks for variable values. When a variable is connected to an input file, each time the variable is accessed, the value will be obtained from the file instead of the simulated storage allocated to the variable. Similarly, writting into a variable that has been connected to an output file results in the value being written into the file (as well as into the storage allocated to the variable). The format for both input and output files is the same: one number/line.

The file names are created by the simulator and consist of the first parameter to the command (the <identifier>) as the file name, with extension ICn (ICONNEct) or OCn (OCONNEct), where n is the user specified channel number. The current implementation only allows up to three input and three output channels open simultaneoulsy. Thus the only valid channel numbers are 1, 2 and 3.

3.13. HELP

HELP tells the user about the command names and their format. HELP <commandname> tells the user about a specific command.

4. Storage Mapping

The simulator allocates space for the registers and memories declared in the RTM symbol table using contiguous storage on the memory of the PDP-10. The fact that the PDP-10 is a 36 bits/word, 2's complement machine is completely transparent to the user. All RTM operations are interpreted rather than compiled into PDP-10 instructions. Moreover, the simulator does not impose any limitations derived from the word length; ISPL registers and memories are allocated contiguous bit strings on the PDP-10.

The use of logical register/memory declarations in the ISPL description presents the following problem: The ISPL compiler allows the user to define arbitrary mappings between bits of the left and right hand sides of the logical declaration, the only check made at that point is that the number of bits is the same. From the simulator point of view, it could be posible to implement arbitrary bit mappings at a tremendous degradation in performance (accessing a bit of a register or memory word that is mapped onto some other component implies searching a table of bit name/position equivalences; having to follow this procedure bit by bit, even for full register/word accesses could be hard to justify). The simulator makes a compromise between convenience to the ISPL writer and efficiency of simulation. The solution adopted is to restrict the types of mappings that the simulator can handle: _all the bits of the right hand side of a logical declaration must be contiguous_. Continuity is defined in terms of the word/bit naming convention used in the main declaration of the register/memory used on the right hand side of the logical declaration. There are no limitations as to what can appear on the left hand side of the logical declaration, these bits are _by definition_ contiguous.

Specifically, the following are the valid types of mappings allowed by the simulator:

1) If the right hand side of a mapping was declared as a register, the structure of the right hand side must specify a contiguous string of bit names as specified in the main declaration. The number of bits may range from 1 to the entire register length and, for proper subsets of the main declaration, may be located anywhere in the register.

2) If the right hand side of a mapping consists of a single memory word, the valid mappings are those defined as above.

3) If the right hand side of a mapping consists of a set of memory words, the structure of the right hand side must specify a contiguous string of full words as specified in the main declaration. The number of words may range from 1 to the entire memory range and, for proper subsets of the main declaration, may be located anywhere in the memory.

4.1. Allowable Types of Mapping

The following list of memory maps gives a good coverage of the allowable cases:

```
M[#777777:#770000,#7777:0]<7:0>;              !THE ADDRESSING SPACE
MB[#7777:0]<7:0>                      :=      M[#7777:0]<7:0>;
MBIO[#777777:#770000]<7:0>            :=      M[#777777:#770000]<7:0>;
MW[#3777:0]<15:0>                     :=      M[#7777:0]<7:0>;
MWIO[#377777:#370000]<15:0>           :=      M[#777777:#770000]<7:0>;

A0N0N[0:255]<0:15>;
A0NN0[0:255]<15:0>;
AN00N[255:0]<0:15>;
AN0N0[255:0]<15:0>;
R0N<0:15>;
RN0<15:0>;

MAP11[0:15]<0:15>        :=      A0N0N[100:115]<0:15>;
MAP12[0:15]<0:15>        :=      A0NN0[100:115]<15:0>;
MAP13[0:15]<0:15>        :=      AN00N[115:100]<0:15>;
MAP14[0:15]<0:15>        :=      AN0N0[115:100]<15:0>;
MAP15[0:2]<0:2>          :=      R0N<5:13>;
MAP16[0:2]<0:2>          :=      RN0<13:5>;

MAP21[0:15]<15:0>        :=      A0N0N[100:115]<0:15>;
MAP22[0:15]<15:0>        :=      A0NN0[100:115]<15:0>;
MAP23[0:15]<15:0>        :=      AN00N[115:100]<0:15>;
MAP24[0:15]<15:0>        :=      AN0N0[115:100]<15:0>;
MAP25[0:2]<2:0>          :=      R0N<5:13>;
MAP26[0:2]<2:0>          :=      RN0<13:5>;

MAP31[15:0]<0:15>        :=      A0N0N[100:115]<0:15>;
MAP32[15:0]<0:15>        :=      A0NN0[100:115]<15:0>;
MAP33[15:0]<0:15>        :=      AN00N[115:100]<0:15>;
MAP34[15:0]<0:15>        :=      AN0N0[115:100]<15:0>;
MAP35[2:0]<0:2>          :=      R0N<5:13>;
MAP36[2:0]<0:2>          :=      RN0<13:5>;

MAP41[15:0]<15:0>        :=      A0N0N[100:115]<0:15>;
MAP42[15:0]<15:0>        :=      A0NN0[100:115]<15:0>;
MAP43[15:0]<15:0>        :=      AN00N[115:100]<0:15>;
MAP44[15:0]<15:0>        :=      AN0N0[115:100]<15:0>;
MAP45[2:0]<2:0>          :=      R0N<5:13>;
MAP46[2:0]<2:0>          :=      RN0<13:5>;

MAP51<0:5>       :=      A0N0N[100]<4:9>;
MAP52<0:5>       :=      A0NN0[100]<9:4>;
MAP53<5:0>       :=      R0N<5:10>;
MAP54<5:0>       :=      RN0<10:5>;
```

5. Examples

This section contains the transcript of several actual runs. The first example is based on the small ISPL example described in the ISPL manual. The transcript for the compilation phase of the multiplier example appears in the ISPL compiler manual. We start from the point right after the MACRO10 assembler has generated the *.REL file.

5.1. Linking the Compiler Output with the Simulator

```
r link
*mult
*@ispsim
*/ssave mult
*/go

EXIT
```

MULT.REL is the name of the file created by the ISPL compiler. ISPSIM.CMD is the name of the command file containing the list of files that make up the simulator. It also contains commands to load the BLISS10 run time library. The use of the SSAVE switch instead of the SAVE switch creates a shareable version of the program. Thus the result of the LINK10 execution will be named MULT.SHR+MULT.LOW.

5.2. Running the Simulator

Here we run the program that was created in the previous transcript. The example makes use of a few simple commands that set initial values in the variables, selects some variables for tracing and then starts the execution at the main entry point of the description. The example is simple and self explanatory.

```
ru mult
ISP SIMULATOR V3 - NRL ARF STAGE 2
Thursday 29 Jul 76 23:42:13 MULT.ISP[N655MB25]
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

>radix octal
>setval p←2
>setval mpd←3000                      ! #6 on left half of mpd
>trace mpd,p,c
>start 10

@ L0     +#2    C      =#10
@ STEP   +#4    P      =#1
@ L1     +#4    C      =#7
@ STEP   +#10   P      =#1400
@ L1     +#4    C      =#6
@ STEP   +#4    P      =#600
@ L1     +#4    C      =#5
@ STEP   +#4    P      =#300
@ L1     +#4    C      =#4
@ STEP   +#4    P      =#140
@ L1     +#4    C      =#3
@ STEP   +#4    P      =#60
@ L1     +#4    C      =#2
@ STEP   +#4    P      =#30
@ L1     +#4    C      =#1
@ STEP   +#4    P      =#14
@ L1     +#4    C      =#0
SIMULATION COMPLETED

RUN TIME(10 usec units)=45259
RTM OPS EXECUTED=136

>value p
P       =#14
>value mpd
MPD     =#3000
>exit
EXIT
```

When the simulator starts it performs two preliminary operations: 1) It transforms the RTM statement table eliminating the DIVERGE/PMERGE operations that define concurrent operations, and 2) It allocates space for the registers and memories declared in the RTM symbol table. The simulator then types two messages advising the user of the existance of the HELP command and of the use of the <ESC> (AltMode) to break the execution of the simulator from the user's terminal.

The tracing of variables indicates the place in the ISPL program where an assignment to the variable has occurred. The location is identified by printing the nearest ISPL label together with a displacement (in RTM operations) from this label. The name of the variable affected by the transfer is printed, together with the new value. The run time printed at the end of the simulation is obtained from a fast 10us. clock available at CMU. Some installations might now have this feature.

In the above example we initialize the multiplier (P) to 2 and the multiplicand (MPD) to 6. According to the algorithm, the multiplicand is stored in the left half of the MPD register. In the current implementation of the simulator we can not specify partial register initialization, thus, we have to load the right half of MPD with a suitable value (initialization of variables in the command language implies full register modification, with zeroes on the left of the value). At the end of the run, the contents of the P register contains the result of the multiplication (6*2=12 or #14 given that we set the type out radix to OCTAL).

5.3. Executing Selected Procedures

In the following example we show a few more commands and features of the simulator:

```
ru mult
ISP SIMULATOR V3 - NRL ARF STAGE 2
Thursday 29 Jul 76 23:42:13 MULT.ISP[N655MB25]
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

>radix octal

>setval p←3

>setval mpd←400          ! Multiplicand=1

>utrace all

>start step

@ STEP   +#10   P      =#201
RUN TIME(10 usec units)=3001
RTM OPS EXECUTED=9
```

The above sequence shows how the simulator can be used to execute selected procedures from the ISPL description. In fact, the simulator treats ALL labels and procedure names as potential entry points. It does not assign any special meaning to the label of the main body of the ISPL description.

5.4. Reading Command Files

The following example shows the use of the READ command. In this particular case we are not only initializing the variables and setting trace flags, but we are also starting the simulation automaticaly from the command file. The number of ">" character used to prompt the input stream (a user or a command file) indicates the level of nesting of the command stream. One ">" is the mark of the top level.

```
>dtrace all

>read ml.sim

>>! this is a command file

>>setval p←2

>>setval mpd←2000
>>       ! multiplicand=4

>>trace p

>>start 10

@ STEP   +#4    P      =#1
@ STEP   +#10   P      =#1000
@ STEP   +#4    P      =#400
@ STEP   +#4    P      =#200
@ STEP   +#4    P      =#100
@ STEP   +#4    P      =#40
@ STEP   +#4    P      =#20
@ STEP   +#4    P      =#10
SIMULATION COMPLETED

RUN TIME(10 usec units)=32120
RTM OPS EXECUTED=136

>>!end of command file

>>7 LINES READ

>exit

EXIT
```