GENTRY, Michael Lee, 1942-
A COMPILER FOR COMPUTER HARDWARE EXPRESSED
IN MODIFIED APL.

The University of Arizona, Ph.D., 1971
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

© COPYRIGHTED
BY
MICHAEL LEE GENTRY
1971
A Compiler For Computer Hardware
Expressed In Modified APL

by

Michael Lee Gentry

A Dissertation Submitted to the Faculty of the
Department of Electrical Engineering
In Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy
In the Graduate College
The University of Arizona

1971
I hereby recommend that this dissertation prepared under my direction by Michael Lee Gentry entitled "A Compiler For Computer Hardware Expressed in Modified APL" be accepted as fulfilling the dissertation requirement of the degree of Doctor of Philosophy

After inspection of the final copy of the dissertation, the following members of the Final Examination Committee concur in its approval and recommend its acceptance:

This approval and acceptance is contingent on the candidate's adequate performance and defense of this dissertation at the final oral examination. The inclusion of this sheet bound into the library copy of the dissertation is evidence of satisfactory performance at the final examination.
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: Michael J. Hentz
ACKNOWLEDGMENTS

The author is especially grateful to Dr. Fredrick J. Hill, whose guidance throughout the research leading to this dissertation was invaluable. Sincere thanks must also be extended to Mr. C. K. Goddard for his considerable contribution to this project.
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>High-Level Language</td>
<td>3</td>
</tr>
<tr>
<td>Illustrative Example</td>
<td>4</td>
</tr>
<tr>
<td>2. SYNTAX OF AHPL</td>
<td>15</td>
</tr>
<tr>
<td>Syntax Specification</td>
<td>17</td>
</tr>
<tr>
<td>AHPL Syntax Recognizer</td>
<td>24</td>
</tr>
<tr>
<td>3. SEMANTICS OF AHPL</td>
<td>29</td>
</tr>
<tr>
<td>Logic Employed</td>
<td>29</td>
</tr>
<tr>
<td>Control Unit Logic</td>
<td>33</td>
</tr>
<tr>
<td>Logic Outside the Control Unit</td>
<td>36</td>
</tr>
<tr>
<td>4. THE COMPILER</td>
<td>41</td>
</tr>
<tr>
<td>Signal Names</td>
<td>43</td>
</tr>
<tr>
<td>Interconnection File</td>
<td>44</td>
</tr>
<tr>
<td>Description of the Main Compiler</td>
<td>48</td>
</tr>
<tr>
<td>Programmer Defined Functions</td>
<td>52</td>
</tr>
<tr>
<td>5. COMBINATIONAL LOGIC SUBROUTINES</td>
<td>60</td>
</tr>
<tr>
<td>6. RESULTS</td>
<td>69</td>
</tr>
<tr>
<td>Machine Specification</td>
<td>69</td>
</tr>
<tr>
<td>Results</td>
<td>72</td>
</tr>
<tr>
<td>7. CONCLUSIONS</td>
<td>87</td>
</tr>
<tr>
<td>SNOBOL4</td>
<td>87</td>
</tr>
<tr>
<td>Compiler Improvements</td>
<td>90</td>
</tr>
<tr>
<td>Final Conclusions</td>
<td>93</td>
</tr>
</tbody>
</table>
**TABLE OF CONTENTS (Continued)**

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPENDIX A:</td>
<td>LISTING OF MAIN AHPL COMPILER</td>
<td>95</td>
</tr>
<tr>
<td>APPENDIX B:</td>
<td>LISTING OF AHPL COMBINATIONAL LOGIC SUBROUTINE COMPILER</td>
<td>106</td>
</tr>
<tr>
<td>APPENDIX C:</td>
<td>SNOBOL4</td>
<td>124</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>127</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Sample Machine Instructions</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Sample AHPL Program</td>
<td>8</td>
</tr>
<tr>
<td>1.3</td>
<td>Register Transfer Hardware</td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Partial Control Unit</td>
<td>13</td>
</tr>
<tr>
<td>2.1</td>
<td>Syntax Recognizer Program</td>
<td>26</td>
</tr>
<tr>
<td>2.2</td>
<td>Output from Syntax Recognizer</td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>Fundamental Mode Circuits</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Register Transfer Hardware</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Input Busing Hardware</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Assembly of AHPL Compiler Run</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Compiler Block Diagram</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>Realization of a Boolean Expression</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Ripple-carry Adder</td>
<td>63</td>
</tr>
<tr>
<td>5.2</td>
<td>Subroutines for 4-Bit Adder</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>Main AHPL Program</td>
<td>71</td>
</tr>
<tr>
<td>6.2</td>
<td>Subroutine Compiler Input Data</td>
<td>73</td>
</tr>
<tr>
<td>6.3</td>
<td>Compiler Output</td>
<td>74</td>
</tr>
<tr>
<td>7.1</td>
<td>Realization of a Simple Boolean Expression</td>
<td>91</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Transliteration of APL Symbols for Key-punching Machine Description to AHPL Compiler</td>
<td>16</td>
</tr>
<tr>
<td>2. AHPL Syntax</td>
<td>21</td>
</tr>
<tr>
<td>3. Schematic Symbols</td>
<td>32</td>
</tr>
</tbody>
</table>
ABSTRACT

A logic design translator program is developed which automatically "compiles" high-level descriptions of computer architecture into digital logic circuit designs. The high-level language used to specify machine architecture is a modified version of APL (Iverson notation). It is called A Hardware Programming Language (AHPL).

Complex structures of purely combinational logic may be described separately and compiled as subroutines to the main program. Standard AND-OR logic with SET-CLEAR flip-flops is employed. A virtual wire list is generated and printed out. The compiler is written in the SNOBOL4 programming language and run on a CDC 6400 computer.
CHAPTER 1

INTRODUCTION

The increasing computational speed and memory size of modern digital computers is resulting in their application to many design automation problems previously done exclusively by man. In the development of new computers, many of the design and production stages are now fully mechanized by most manufacturers. One task still accomplished by hand is that of logic design; i.e., preparation of detailed diagrams or equations to specify the internal structure of a computer. Because this process is expensive and time-consuming, design automation would be advantageous.

Normally, the design of a computer begins with the definition of an instruction repertoire. Next, the functional operation of each instruction is outlined. Then the logical (Boolean) equations are derived which allow implementation of the instructions. These equations can than be translated into a "wire list" specifying the interconnection of hardware elements in the machine. The goal of this dissertation is automation of the logic design step where detailed diagrams or equations are prepared.
Programs or systems which perform this task are generally referred to as logic design generators (translators).

A logic design generator accepts a high-level architectural description of a computer and generates a detailed, machine-oriented description in terms of logical equations or their equivalent. Although conventional compilers differ in that they produce a machine language program as output, the analogy between compilers and logic design generators is clear. Hence, the name "hardware compiler" may be used to refer to a logic design translator.

In addition to relieving the design engineer of uninteresting details, the hardware compiler should provide fewer design errors, lower cost, better documentation, possible simulation of the proposed machine, and ease of design alteration. The input program alone serves as a canonical description of the proposed computer. Naturally, it would be desirable to ultimately incorporate the hardware compiler into a complete design automation system.

In the early 1960's, at the Burroughs Company, the "classical" work on hardware compilers was done by Gorman and Anderson (1962), and Proctor (1964). Their three-pass compiler consisted of three major parts. The "Scan and Recognizer" translated the input into an internal representation. Then, the "Design Table Analyzer" performed
a complicated timing analysis. Finally, the "Equation Generator" produced equations for the entire system. The input consisted of microsequences for each instruction, specification of hardware blocks (adders, multipliers, etc.), and block diagram descriptions of the proposed computer.

More recently, the work reported by Friedman and Yang (1969) at International Business Machines is of interest. Their "ALERT" system goes from architectural description to logic via eight successive major transformation steps with each step supplying detail needed to complete the design. Complex structures may be represented by macro functions, arrays, algorithms, and subscripted variables. Friedman and Yang (1968) describe the quality of designs produced by ALERT.

**High-Level Language**

The high-level language used to describe the digital system being designed is called a register transfer language. It should be concise, convenient, flexible, machine-readable, and capable of expressing any feature of a computer. Several such languages have been proposed. These include Reed's register transfer language (1952), the Logic Design Translator (LDT) language of Burroughs (Proctor, 1964), Digital System Design Language (DDL)
(Duley and Dietmeyer, 1968), Schlaeppi's LOTIS (1964),
APL (Iverson notation) (Iverson, 1962), and others.

The high-level language chosen for this dissertation is based upon APL (A Programming Language). Because the original version, as developed by Iverson, contained many features not pertinent to hardware specification and lacked others, certain modifications were imperative. This version of the language so modified is referred to as A Hardware Programming Language, or AHPL. In their forthcoming book entitled Introduction to Digital Hardware Systems (John Wiley and Sons, to be published), Professors Hill and Peterson of The University of Arizona present an excellent introduction to AHPL. The language actually compiled by the hardware compiler differs only in a few minor respects from the language described in the book. A formal syntactic specification of AHPL is presented in the next chapter of this dissertation.

**Illustrative Example**

The general approach or philosophy of design employed in using AHPL and the hardware compiler is outlined in detail in Hill and Peterson (to be published). In order to quickly familiarize the reader with this method, a brief example will be presented at this point. The example consists of an AHPL program describing certain features of
a hypothetical computer and the partial results of compiling
the program into hardware. This hypothetical computer is
greatly simplified and not intended to represent an au-
thentic computer.

To begin, suppose a certain computer is a single-
address machine with only 8-bit instruction and data words.
It contains four registers: 8-bit instruction and data
words. It contains four registers: 8-bit accumulator
program counter (PC), 6-bit memory address register (MA),
and an 8-bit memory data register (MD). The instruction
words consist of a 2-bit operation code followed by a
6-bit address (except I/O and Operate instructions). The
organization of instruction words is shown in Figure 1.1
(a). The operation code is shown in Figure 1.1 (b). Bit
3 will be 1 for Input/Output instructions and 0 for
operate instructions. For operate instructions, the re-
maining bits, 4 through 8, will specify the operation. Only
5 operations are used, and each is specified by one of the
right-most five bits in the instruction word, as shown in
Figure 1.1 (c). Of course, $2^5 = 32$ separate operations
could be specified by bits 4 through 8, but this would
unnecessarily lengthen and complicate the example. Also,
the number of addressable memory words ($2^6 = 64$) is un-
realistically small.
(a) Instruction Word

<table>
<thead>
<tr>
<th>Bits</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>JMP - take next instruction from A.</td>
</tr>
<tr>
<td>0 1</td>
<td>LAC - load AC register from A.</td>
</tr>
<tr>
<td>1 0</td>
<td>DAC - deposit AC register into A.</td>
</tr>
<tr>
<td>1 1</td>
<td>I/O or Operate instruction.</td>
</tr>
</tbody>
</table>

(b) Operation Code

<table>
<thead>
<tr>
<th>4 5 6 7 8 Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 0 Clear accumulator.</td>
</tr>
<tr>
<td>0 1 0 0 0 0 Rotate AC left.</td>
</tr>
<tr>
<td>0 0 1 0 0 0 Rotate AC right.</td>
</tr>
<tr>
<td>0 0 0 1 0 0 Shift AC left, zero fill.</td>
</tr>
<tr>
<td>0 0 0 0 1 1 Shift AC right, zero fill.</td>
</tr>
</tbody>
</table>

(c) Operate Instructions

Figure 1.1. Sample Machine Instructions
The AHPL program describing this example machine is shown in Figure 1,2. This program is, of course, key-punched onto regular IBM punch cards to be read and compiled by the hardware compiler into a logic design for the machine. An asterisk (*) in column 1 indicates a comment statement. These statements are not processed, but merely printed in the output file. All statements except register declarations are labeled with an integer located within the first five columns of each statement. This permits branching within the program. Lengthy statements may be continued on successive cards by placing a plus sign (+) in column 1 of the continuation cards. Within each register, all bits are numbered sequentially from one (not zero), beginning with the left-most bit. For assignment statements (not branch or declarations), the label field is followed by a compiler key. For asynchronous operations, the key is simply the letter "A". For synchronous operations, the key is the letter "S", followed by an integer indicating the number of machine clock periods required for completion of the transfer specified. One or more blanks separate the key and the main portion of the statement, which may not contain blanks.

The first four statements declare the names of all registers and their lengths. Line 5 is a comment. The remainder of the program may be interpreted as a control
Figure 1.2. Sample AHPL Program.
program describing the events that must occur in the machine as it executes instructions. Statement 1 transfers the contents of the program counter to the MA register and statement 2 fetches the next instruction from central memory to the MD register. Statement 3 is a multiple branch. Bits 1 and 2 of the new instruction are examined and the appropriate branch taken. JMP instructions are executed by statement 4. Control then branches back to 1 for the next instruction. Statements 6 through 9 cause a data word to be fetched from memory and transferred to the accumulator. DAC instructions are accomplished by statements 10 through 13. The accumulator's contents are moved first to the MD register and then deposited in memory. Statement 14 separates I/O and operate instructions. Statement 15 is a conditional assignment which performs whichever operation is specified by the instruction.

After completing a LAC, DAC, or operate instruction, the control program arrives at 16. This statement must increment the value of the program counter. An AHPL statement can be written to accomplish this, but it is cumbersome. In effect, it would add the necessary logic to make the PC register into a binary counter. The counter, adders, etc. are examples of functional logic units which consist entirely of combinational logic but are lengthy and
unwieldy to write into single AHPL statements. The problem is more serious for machines with larger word size than 8 bits. For these units, a special feature of the hardware compiler may be employed. A combinational logic AHPL "subroutine" may be written describing these units separately. They are compiled separately and their hardware stored for use when called in the main AHPL program. Chapter 5 discusses these subroutines in more detail.

Statement 17 returns the control to statement 1 to begin the next instruction. Programming for I/O statements would follow statement 16, but is omitted here for brevity.

For compilation purposes, this same program must be interpreted as specifying the machines' hardware. For example, statement 4 requires the right-most 6 bits of the MD to be transferred to the PC. The hardware shown in Figure 1.3 (a) would accomplish this transfer upon receipt of an enabling control pulse.

Similar hardware would result for the transfers specified in statements 1, 2, 6, 7, 8, 10, 11, and 12. Statement 15 would result in bits 4 through 8 of the MD register being ANDed individually with a control pulse to generate the enabling pulse for the corresponding transfer. For example, the hardware for rotating the AC to the right is illustrated in Figure 1.3 (b).
Figure 1.3. Register Transfer Hardware

(a) Hardware for PC=SUFFIX(6)/MD

(b) Hardware for Rotate AC Right
At the same time the register transfer hardware is being built, the compiler generates the control unit hardware. The control unit, of course, must generate, at the proper time, all pulses enabling transfers. It is built around a special hardware device known as a control delay element, which generates an output pulse one clock period after receiving a pulse. This delay allows transfers to be executed.

To illustrate, the first three statements in the control program for the sample machine are reproduced in block diagram form in Figure 1.4 (a). Figure 1.4 (b) shows the control unit hardware for these statements.

A single pulse is generated to start the machine. This pulse is sustained and propagated throughout the control unit, generating control pulses for execution of all transfers. The start pulse is ORed with any pulses returning control to statement 1 to generate the enabling pulse for the MA=PC transfer. The enabling pulse also enters a delay element and emerges one clock period later to initiate the memory read cycle required by statement 2 and sets a flip-flop. The flip-flop is ANDed with the read cycle completion pulse (synchronized with the machine clock) to reset the flip-flop and propagate a pulse to statement 3. Notice that no model of central memory is supplied. This is true of all designs produced by the hardware
(a) Program Flow Chart

(b) Control Unit Hardware

Figure 1.4. Partial Control Unit
compiler. The Boolean expressions in statement 3 are hardware-realized and used to route this control pulse to the proper location. It can be seen that the control unit may be partitioned into segments (see dotted vertical lines) which are associated with individual control program statements.

These first three statements illustrate the synchronous assignment, asynchronous memory reference, and branch type statements. Every other statement in the example AHPL control program is one of these three basic types. Thus, the entire control unit for the machine can be realized by extending the hardware of Figure 1.4. Subsequent chapters will elaborate in detail upon many of the ideas presented in this example.
CHAPTER 2

SYNTAX OF AHPL

An AHPL program describing some digital system conveys two separate messages. Naturally, the system's behavior or response to some input data is described by the program. However, from the viewpoint of hardware compilation, the same AHPL program relates information about how the system could be built. This structural interpretation of system description programs imposes certain modifications and conventions on APL, which are incorporated into AHPL. A fundamental convention adopted is that variables in the AHPL programs are considered to represent physical hardware such as registers, flip-flops, gate output signals, buses, etc. Declaration statements at the beginning of each AHPL program fix the category and dimension of each variable. Accordingly, Iverson operators modifying rank and dimension of variables are not included in AHPL. As input data to the hardware compiler, the AHPL programs must be punched onto cards which are read by the CDC 6400 computer. Because of this keypunching, some symbols in Iverson notation must be transliterated as shown in Table 1. Notice the logical operators $\land$(and), $\lor$(or),
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Transliteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\land\lor\neg)</td>
<td>Logic Operations</td>
<td>(\land\lor\neg)</td>
</tr>
<tr>
<td>+</td>
<td>Assignment</td>
<td>=</td>
</tr>
<tr>
<td>→</td>
<td>Branch</td>
<td>+</td>
</tr>
<tr>
<td>x</td>
<td>Vector (Register)</td>
<td>X</td>
</tr>
<tr>
<td>(x_i)</td>
<td>Subscript</td>
<td>(x_i)</td>
</tr>
<tr>
<td>,</td>
<td>Catenation</td>
<td>,</td>
</tr>
<tr>
<td>α</td>
<td>Prefix</td>
<td>&quot;PREFIX&quot;</td>
</tr>
<tr>
<td>ω</td>
<td>Suffix</td>
<td>&quot;SUFFIX&quot;</td>
</tr>
<tr>
<td>(k\uparrow x)</td>
<td>Rotate x to left k bits</td>
<td>(k\uparrow x)</td>
</tr>
<tr>
<td>(k\downarrow x)</td>
<td>Rotate x to right k bits</td>
<td>(k\downarrow x)</td>
</tr>
<tr>
<td>(k(x;p))</td>
<td>Shift left, fill with p</td>
<td>(k(x;p))</td>
</tr>
<tr>
<td>(k^p x)</td>
<td>Shift right, fill with p</td>
<td>(k^p x)</td>
</tr>
<tr>
<td>([x_1<em>s_1+...+x_n</em>s_n,s_{n+1})]</td>
<td>Conditional Branch</td>
<td>([x_1<em>s_1+...+x_n</em>s_n,s_{n+1})]</td>
</tr>
<tr>
<td>([x_1<em>v_1+...+x_n</em>v_n,v_{n+1})]</td>
<td>Conditional Assignment</td>
<td>([x_1<em>v_1+...+x_n</em>v_n,v_{n+1})]</td>
</tr>
</tbody>
</table>

and \(\neg\) (complement). Notice also that because of the absence of "α" and "ω" from the CDC 6400 character set, these special vectors are transliterated into the strings "PREFIX" and "SUFFIX" correspondingly.
Conditional branch should be interpreted as branch to statement $S_1$ if Boolean Expression $X_1$ is true, branch to statement $S_2$ if Boolean Expression $X_2$ is true, etc. Finally, if none of the expressions $X_1$ through $X_n$ are true, branch to statement $S_{n+1}$. Thus, $S_{n+1}$ serves as a "default option" for conditional branch statements. Conditional assignment is treated analogously.

Characters available on the CDC 6400 are referred to as terminal characters. For purposes of reference, a complete list of terminal characters is given:

\[
T = \{ A B C D E F G H I J K L M N O P Q R S T U [ V W X Y Z ( ) + = / * , ; : \ - \ + \ + \ - \ < \ > \} 0 1 2 3 4 5 6 7 8 9 \leq \geq \equiv \}
\]

The symbol $\|$ stands for "blank".

Other conventions will be mentioned when appropriate.

**Syntax Specification**

When describing a programming language, the syntax and semantics of the language are two properties of prime importance. The semantic discussions are concerned with the meanings attached to various structures within the language. The semantics of AHPL are the subject of Chapter 3.
present chapter focuses upon a formal presentation of the syntax of AHPL.

The syntax specification of a language is a set of definite rules or conventions on the way symbols are combined into expressions, operands, etc. The allowable combinations start with individual characters and eventually result in a statement (or sentence, program, etc.) in the given language. In discussing the syntax of a language, the peculiar problem of distinguishing between the language being described and the describing language is encountered. Care must be exercised to avoid confusion and ambiguity. The language in terms of which the description is being made is called the meta-language.

Three common such formalisms which have been developed for the representation of language syntax are:

1. Post Production Systems
2. Phrase Structure Grammars
3. Backus Normal Form, or BNF

All three are essentially equivalent. The logician Emil Post developed Post Production Systems during the 1940's as a tool in the study of Symbolic logic. During the 1950's, linguist Noam Chomsky developed Phrase Structure Grammars for use in his study of natural languages. Both of these are most useful in theoretical study of languages and their properties. Backus Normal Form was developed by
the programmer John Backus at IBM during the late 1950's as a tool in the description of programming languages. Because it is more succinct and useful in compilation, a formalism most similar to BNF was chosen to present the syntax of AHPL.

The metalanguage BNF requires a few "metasymbols" of its own in addition to those of AHPL. The metasymbols of BNF and their meanings are as follows:

::= means "is defined as"
| means "or"
(____) means "the element (____) of the language"
(____)(____) juxtaposition means "followed by"

Notice that here English is used to describe BNF. In effect, English is a meta-metalanguage used to describe the metalanguage BNF which is in turn used to describe AHPL.

A few simple examples will illustrate the formalism. Consider the following:

\[ \langle \text{DIGIT} \rangle ::= 0|1|2|3|4|5|6|7|8|9 \]
\[ \langle \text{INTEGER} \rangle ::= \langle \text{DIGIT} \rangle | \langle \text{INTEGER} \rangle \langle \text{DIGIT} \rangle \]

Here, \( \langle \text{DIGIT} \rangle \) is an example of syntactic types which consist of single characters from the set of terminal characters. The second example should be read "the element \( \langle \text{INTEGER} \rangle \) of the language is defined as a \( \langle \text{DIGIT} \rangle \) or an \( \langle \text{INTEGER} \rangle \) followed by a \( \langle \text{DIGIT} \rangle \)". Of course, the definition is left-recursive and means an \( \langle \text{INTEGER} \rangle \) is one or
more consecutive digits. The left side of each definition is called the "Defined Type", and the right side the "Definiens".

Using the metalanguage BNF, the syntactic specification of AHPL is given in Table 2. Notice that the specification is presented in "bottom-up" fashion where the lowest form of syntactic type appears first and the highest form last. The lowest form of syntactic type must, of course, consist entirely of terminal characters. On the other hand, the highest form may use any or all previously defined syntactic types. The specification could have been presented equally well in the reverse order.

Notice also that ⟨SUBROUTINE⟩, ⟨REGISTER⟩, and ⟨BUS⟩ have identical definitions; i.e., one or more consecutive letters. Obviously, any AHPL compiler must distinguish between the three types. This is possible because all registers and buses are declared and subroutines defined before compilation of the main program. The AHPL compiler generates syntax types to match registers, buses, and subroutine names as it processes declarations and subroutines.

The manner in which Boolean expressions are specified automatically establishes the rules of precedence for the three Boolean operators. Complement (¬) takes
Table 2. AHPL Syntax

\[
\langle \text{LETTERS} \rangle ::= \text{A}|\text{B}|\text{C}|\text{D}|\text{E}|\text{F}|\text{G}|\text{H}|\text{I}|\text{J}|\text{K}|\text{L}|\text{M}|\text{N}|\text{O}|\text{P}|\text{Q}|\text{R}|\text{S}|\text{T}|\text{U}|\text{V}|
\text{W}|\text{X}|\text{Y}|\text{Z}
\]
\[
\langle \text{DIGIT} \rangle ::= 0|1|2|3|4|5|6|7|8|9
\]
\[
\langle \text{INTEGER} \rangle ::= \langle \text{DIGIT} \rangle | \langle \text{INTEGER} \rangle \langle \text{DIGIT} \rangle
\]
\[
\langle \text{BLANKS} \rangle ::= \text{B} | \langle \text{BLANKS} \rangle \text{B}
\]
\[
\langle \text{LOGICAL VALUE} \rangle ::= 0|1
\]
\[
\langle \text{BOOLE CNT} \rangle ::= \langle \text{LOGICAL VALUE} \rangle | \langle \text{BOOLE CNT} \rangle \langle \text{LOGICAL VALUE} \rangle
\]
\[
\langle \text{BINARY} \rangle ::= \text{\&}|\text{|}\text{\lor}
\]
\[
\langle \text{REGISTER} \rangle ::= \langle \text{LETTERS} \rangle | \langle \text{REGISTER} \rangle \langle \text{LETTERS} \rangle
\]
\[
\langle \text{SUBROUTINE} \rangle ::= \langle \text{LETTERS} \rangle | \langle \text{SUBROUTINE} \rangle \langle \text{LETTERS} \rangle
\]
\[
\langle \text{ROT LFT} \rangle ::= \langle \text{INTEGER} \rangle \text{\uparrow} \langle \text{REGISTER} \rangle
\]
\[
\langle \text{ROT RHT} \rangle ::= \langle \text{INTEGER} \rangle \text{\downarrow} \langle \text{REGISTER} \rangle
\]
\[
\langle \text{SFT LFT} \rangle ::= \langle \text{INTEGER} \rangle \text{\uparrow} (\langle \text{REGISTER} \rangle;\langle \text{LOGICAL VALUE} \rangle)
\]
\[
\langle \text{SFT RHT} \rangle ::= \langle \text{INTEGER} \rangle \text{\downarrow} (\langle \text{REGISTER} \rangle;\langle \text{LOGICAL VALUE} \rangle)
\]
\[
\langle \text{SUFFIX} \rangle ::= \text{SUFFIX}(\langle \text{INTEGER} \rangle)/\langle \text{REGISTER} \rangle |
\text{SUFFIX}(\langle \text{INTEGER} \rangle)/\text{PREFIX}(\langle \text{INTEGER} \rangle)/\langle \text{REGISTER} \rangle
\]
\[
\langle \text{PREFIX} \rangle ::= \text{PREFIX}(\langle \text{INTEGER} \rangle)/\langle \text{REGISTER} \rangle |
\text{PREFIX}(\langle \text{INTEGER} \rangle)/\text{SUFFIX}(\langle \text{INTEGER} \rangle)/\langle \text{REGISTER} \rangle
\]
\[
\langle \text{CHARACTERISTIC} \rangle ::= \langle \text{REGISTER} \rangle \langle \langle \text{INTEGER} \rangle \rangle
\]
\[
\langle \text{REDUCE} \rangle ::= \langle \text{BINARY} \rangle/\langle \text{REGISTER} \rangle | \langle \text{BINARY} \rangle/\langle \text{PREFIX} \rangle |
\langle \text{BINARY} \rangle/\langle \text{SUFFIX} \rangle
\]
Table 2. (Continued)

\[
\langle \text{BIT} \rangle := \langle \text{LOGICAL VALUE} \rangle | \langle \text{REGISTER} \rangle (\langle \text{INTEGER} \rangle) | \langle \text{REDUCE} \rangle | (\langle \text{BOOLE EXPN} \rangle)
\]

\[
\langle \text{NEG TERM} \rangle := \langle \text{BIT} \rangle | -\langle \text{BIT} \rangle
\]

\[
\langle \text{PROD TERM} \rangle := \langle \text{NEG TERM} \rangle | \langle \text{PROD TERM} \rangle \land \langle \text{NEG TERM} \rangle
\]

\[
\langle \text{BOOLE EXPN} \rangle := \langle \text{PROD TERM} \rangle | \langle \text{BOOLE EXPN} \rangle \lor \langle \text{PROD TERM} \rangle
\]

\[
\langle \text{VARIABLE} \rangle := \langle \text{REGISTER} \rangle | \langle \text{ROT RHT} \rangle | \langle \text{ROT LFT} \rangle |
\]

\[
\quad \langle \text{SFT RHT} \rangle | \langle \text{SFT LFT} \rangle | \langle \text{BOOLE CNT} \rangle |
\]

\[
\quad \langle \text{CHARACTERISTIC} \rangle | \langle \text{PREFIX} \rangle | \langle \text{SUFFIX} \rangle |
\]

\[
\quad \langle \text{REGISTER} \rangle \langle \text{BINARY} \rangle \langle \text{REGISTER} \rangle
\]

\[
\langle \text{CATENATION} \rangle := \langle \text{VARIABLE} \rangle | \langle \text{CATENATION} \rangle, \langle \text{VARIABLE} \rangle
\]

\[
\langle \text{ASYN} \rangle := \langle \text{REGISTER} \rangle = \text{MEMORY} (\langle \text{REGISTER} \rangle) |
\]

\[
\quad \text{MEMORY} (\langle \text{REGISTER} \rangle) = \langle \text{REGISTER} \rangle |
\]

Asynchronous Subroutine Call

\[
\langle \text{COND ASSN} \rangle := \langle \text{BOOLE EXPN} \rangle \ast \langle \text{CATENATION} \rangle |
\]

\[
\quad \langle \text{COND ASSN} \rangle + \langle \text{BOOLE EXPN} \rangle \ast \langle \text{CATENATION} \rangle
\]

\[
\langle \text{SYNC} \rangle := \langle \text{CATENATION} \rangle = \langle \text{CATENATION} \rangle |
\]

\[
\quad \langle \text{CATENATION} \rangle = [\langle \text{COND ASSN} \rangle] \langle \text{CATENATION} \rangle |
\]

Synchronous Subroutine Call

\[
\langle \text{ASSN} \rangle := \langle \text{SYNC} \rangle | \langle \text{ASYN} \rangle
\]

\[
\langle \text{COND BRANCH} \rangle := \langle \text{BOOLE EXPN} \rangle \ast \langle \text{INTEGER} \rangle |
\]

\[
\quad \langle \text{COND BRANCH} \rangle + \langle \text{BOOLE EXPN} \rangle \ast \langle \text{INTEGER} \rangle
\]
Table 2, (Continued)

\[
\langle \text{BRANCH} \rangle ::= \langle \text{INTEGER} \rangle \mid +[\langle \text{COND BRANCH} \rangle] \langle \text{INTEGER} \rangle
\]

\[
\langle \text{DECLAR} \rangle ::= \langle \text{BLANKS} \rangle \langle \text{REGISTER} \rangle = \langle \text{INTEGER} \rangle
\]

\[
\langle \text{STATEMENT} \rangle ::= \langle \text{INTEGER} \rangle \langle \text{BLANKS} \rangle \langle \text{BRANCH} \rangle \mid \langle \text{DECLAR} \rangle \mid \\
\langle \text{INTEGER} \rangle \langle \text{BLANKS} \rangle \langle \text{ASSN} \rangle
\]
precedence over AND (\&), and OR (\lor) is evaluated last. Thus, the expression \(-X\&Y\lor Z\) means \(((\neg X)\& Y)\lor Z\).

The branching facility of AHPL makes it imperative that certain, if not all, statements be labeled. This is accomplished by a five-column field at the beginning of each statement in the main program. A statement must contain an integer label in this field, permitting control transfer to the statement.

Thus, the definition of \(\text{STATEMENT}\) contains the syntactic types \(\langle \text{INTEGER} \rangle \langle \text{BLANKS} \rangle\) at the beginning of each alternative to indicate this label field.

**AHPL Syntax Recognizer**

By itself this syntactic specification of AHPL does not constitute a set of rules for producing or recognizing allowable strings in the language. However, to use the specification as a "generative grammar" and produce allowable AHPL strings is a straightforward matter. For example, to produce a \(\langle \text{SUFFIX} \rangle\) special vector, first write down the string "SUFFIX" followed by "(". Then choose an \(\langle \text{INTEGER} \rangle\). Follow this with ")/". Finally, choose a \(\langle \text{REGISTER} \rangle\). A similar procedure could be applied to generate any syntactic type, including \(\langle \text{STATEMENT} \rangle\).

As is often the case, the inverse problem of testing the syntactic correctness of a given string is
more difficult, but certainly not impossible. To demonstrate, a SNOBOL4 program was written which serves as a syntax recognizer for AHPL statements. The program is shown in Figure 2.1. Strings or statements to be tested are read in from punched cards and printed in the output listing. Syntactically incorrect statements are identified by an error message. Figure 2.2 shows a sample output listing from the syntax recognizer where both syntactically correct and incorrect statements were encountered.

Although this program is not a part of the actual AHPL compiler, it does illustrate the power of the SNOBOL4 programming language when applied to problems of this nature. An examination of the syntax recognizer program reveals an interesting fact. A major portion of the program consists of a syntactic specification of AHPL almost identical to the BNF form shown in Table 2. For the program, the syntactic specification had to be cast into the language of SNOBOL4. For example, the BNF metasymbols ::= and | were transliterated into = and V respectively, and syntactic types are not enclosed in angle brackets in the program. Also, terminal characters are enclosed within up-arrows (used as quotation marks in SNOBOL). Other differences also exist, but to point all of them out would involve the discussion too deeply in the intricacies of SNOBOL.
SNOBOL4 (VERSION 2.0)

BELL TELEPHONE LABORATORIES, INCORPORATED
CDC VERSION BY PURDUE UNIVERSITY AND I.R.A.

* THIS PROGRAM IS A SYNTACTIC RECOGNIZER FOR AHPL STATEMENTS.
* FIRST A SERIES OF PATTERNS IS BUILT CULMINATING IN A PATTERN WHICH MATCHES ONLY SYNTACTICALLY CORRECT STATEMENTS. CARD IMAGES ARE THEN READ IN AND PROCESSED. INCORRECT STATEMENTS ARE IDENTIFIED BY AN ERROR MESSAGE.

LETTERS = +ABCDEFGHIJKLMNOPQRSTUVWXYZ
INTEGER = SPAN(+0123456789t)
BLANKS = SPAN(t)
LOG.VAL = ANY(+01+)
BOOLON = SPAN(+01+)
BINARY = +t+ v +tv+
REG = FAIL
SFT.ROT = INTEGER ANY(#9) (*REG v (+(+ *REG +t+ LOG.VAL +t+))
PREFIX = +PREFIX(+ INTEGER +)/* (*REG v (+SUFFIX(+ INTEGER
+))/+( +REG))
SUFFIX = +SUFFIX(+ INTEGER +)/* (*REG v (+PREFIX(+ INTEGER
+%))/+( +REG))
REDUCE = BINARY +v/* (*REG v PREFIX v SUFFIX)
BIT = (+t+ v NULL) ((*REG +t+ +INTEGER +t+) v LOG.VAL v REDUCE)
BOOEXP = (+t+ v NULL) (+(+ BIT BINARY BIT +) v BIT v
(+*BOOEXP BINARY +*BOOEXP)
L.UNIT = *REG BINARY *REG

Figure 2.1. Syntax Recognizer Program
VAR = *REG v SFT v ROT v PREFIX v SUFFIX v BOOCON v L v UNIT
CAT = ARBNO(VAR v v) VAR
ASYN = *REG ^=MEMORY(* *REG *) ^= v ^=MEMORY(* *REG ^=) ^= *REG
COND. ASSN = ARBNO(BOOEXP ^=^ CAT ^=^) BOOEXP ^=^ CAT
SYNC = CAT ^=^ ([*+ COND. ASSN ^= v NULL]) CAT
COND. BRH = ARBNO(BOOEXP ^=^ INTEGER ^=^) BOOEXP ^=^ INTEGER
BRANCH = ^=^ ([*+ COND. BRH ^= v NULL]) INTEGER
STATEMENT = INTEGER BLANKS ((BRANCH v ASYN v SYNC))
DECLAR = BLANKS SPAN(LETTERS) • VECTOR ^=INTEGER

* PATTERNS ARE NOW READY. CARD ANALYSIS PROGRAM FOLLOWS.
  
  ^ANCHOR = 1
EOF =
READI IMAGE = TRIM(INPUT)
OUTPUT = ^ IMAGE
IMAGE ^=
NEXTST IDENT(EOF)
OUTPUT = ^ LINE
IMAGE = LINE
READC LINE = TRIM(INPUT)
LINE ^=
LINE ^= =
OUTPUT = ^ LINE
IMAGE = IMAGE LINE
PRINT OUTPUT = ^ LINE
EOF = 1
ENDGAME EOF = 1
ANALYZE IMAGE DECLAR
REG = REG v VECTOR
NOTD IMAGE STATEMENT
OUTPUT = ^<<<SYNTACTIC ERROR>>>^ (NEXTST)
OK OUTPUT = ^<<<NO SYNTACTIC ERROR>>>^ (NEXTST)
END

Figure 2.1. (Continued)
NO ERRORS DETECTED DURING COMPILATION

* SAMPLE AHPL PROGRAM ILLUSTRATING THE SYNTAX RECOGNIZER.
* DECLARATION OF MACHINE REGISTERS
  AC=8
  D=8
  PC=6

* DESCRIPTION OF MACHINE ARCHITECTURE FOLLOWS:
1  AC=PC
  <<<NO SYNTACTIC ERROR>>>  
2  D=MEMORY(A)
  <<<SYNTACTIC ERROR>>>  
3  IR=0
  <<<SYNTACTIC ERROR>>>  
4  *[^PREFIX(3)/D*11],5
  <<<NO SYNTACTIC ERROR>>>  
5  AC=D
  <<<NO SYNTACTIC ERROR>>>  
6  *[^AC<4>*7],10
  <<<NO SYNTACTIC ERROR>>>  
7  AC=[PC<1]^PC<2>*D],AC&D
  <<<NO SYNTACTIC ERROR>>>  
8  D=M(A)
  <<<SYNTACTIC ERROR>>>  
9  SUFFIX(6)/AC=SUFFIX(6)/D
  <<<NO SYNTACTIC ERROR>>>  
10  *[^-AC<8>*1],11
  <<<NO SYNTACTIC ERROR>>>  
11A  AC=2+(AC:*0)
  <<<SYNTACTIC ERROR>>>  
12  *[^1]
  <<<NO SYNTACTIC ERROR>>>  

Figure 2.2. Output from Syntax Recognizer
CHAPTER 3

SEMANTICS OF AHPL

The ultimate goal of the AHPL compiler is a hardware specification or logic design of the proposed machine in terms of Boolean equations of their equivalent. Thus, in addition to the major problem of syntactic analysis of incoming statements, the compiler must "generate" the machine's hardware. To accomplish this, some meaning in terms of hardware or logic must be associated with each allowable syntactic construction. That is to say, the semantics of AHPL must be specified.

Logic Employed

At this point, the primitive hardware components assumed available by the compiler must be defined. For reasons of generality and simplicity, standard AND and OR gates with unrestricted fan-in and fan-out requirements were used by the compiler in implementing the logic. Both the true and complemented output signal is assumed available from all gates, eliminating the need for separate NOT gates. For high-speed memory, standard SET-CLEAR flip-flops were assumed with 1 and 0 outputs available. Thus,
the complement of every logic signal is assumed available by the compiler.

For some asynchronous operations, a pulse must be generated to indicate completion of the operation. Because this completion pulse returns to the control unit to become the sustaining control pulse for the machine's operation, it is necessary that this pulse be synchronized with the basic machine clock. A special asynchronous circuit is assumed available to generate this pulse. This completion pulse synchronizer circuit is shown in Figure 3.1 (a).

Finally, a special fundamental mode sequential circuit called the control delay element was assumed. This circuit is shown in Figure 3.1 (b). This element accepts an incoming control pulse and generates an outgoing pulse one machine clock period later. It is used exclusively in the machine's control unit. The example in Chapter 1 illustrates how this element is used in the control unit logic design. The design of this circuit as well as the completion pulse synchronizer circuit is given in the Appendix to Introduction to Digital Hardware Systems by Professors Hill and Peterson,

Table 3 shows the schematic symbols used for each hardware element used by the compiler.

Now, the AHPL compiler is a one-pass compiler in which each statement is processed as it is read in and then
Figure 3.1. Fundamental Mode Circuits
Table 3. Schematic Symbols

<table>
<thead>
<tr>
<th>Element</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND gate</td>
<td><img src="image1" alt="AND gate diagram" /></td>
</tr>
<tr>
<td>OR gate</td>
<td><img src="image2" alt="OR gate diagram" /></td>
</tr>
<tr>
<td>Control Delay</td>
<td><img src="image3" alt="Control Delay diagram" /></td>
</tr>
<tr>
<td>Completion Pulse Synchronizer</td>
<td><img src="image4" alt="Completion Pulse Synchronizer diagram" /></td>
</tr>
</tbody>
</table>
discarded. Thus, all machine hardware, both within the control unit and elsewhere, is generated simultaneously by the compiler. Exceptions to this statement are the logic units which are actually compiled by a separate compiler program. Although control unit and register logic is parallel generated, it is convenient for purposes of explanation to break the discussion into two parts. The first section discusses the control unit hardware. The second section covers hardware configurations outside the control unit.

**Control Unit Logic**

To speed compilation of the hardware for the control unit, the AHPL programmer must supply keys for each assignment statement (synchronous or asynchronous). No key is necessary for branch statements. For asynchronous assignment statements, the letter "A" is inserted between the five column label field and the remainder of the statement. The "A" indicates to the compiler that this assignment statement is asynchronous. For synchronous assignment statements, the letter "S" followed by an integer is similarly inserted. The integer indicates to the compiler the number of clock periods required to complete the transfer. The use of these compiler keys is made more clear in the following paragraphs.
Within the control unit there are basically three hardware configurations corresponding to the three basic types of statements. First, when the compiler key indicates asynchronous assignment statement, the following timing device is generated in the hardware of the control unit:

Examples of asynchronous transfers include memory reference instructions and combinational logic units with varying execution times.

Suppose now that the compiler key is of the form "S"i, where i is an integer. This would, of course, indicate a synchronous assignment statement which would require i clock periods to execute. For this statement, the compiler would insert into the control unit i control delay elements concatenated end-to-end. Most commonly, i is equal to 1. To illustrate, suppose the AC and DA are declared 16-bit registers and the following AHPL statement is encountered:

50    S1    AC=2+(DA;1)
This would result in the following control unit hardware:

A transfer requiring multiple clock periods for execution would generate similar hardware with the corresponding number of delay elements concatenated.

The processing of direct branch statements results in no new hardware. The control pulse is merely routed to the proper section of the control unit. However, conditional branch statements require some gating within the control unit. The logic generated for conditional branch statements is of the general form shown below:
$S_1$ through $S_n$ are statements to which control is passed if the corresponding Boolean expression $X_1$ through $X_n$ is true. In the event $X_1$ through $X_n$ are all false, the control is passed to statement $S_{n+1}$.

**Logic Outside the Control Unit**

Outside the control unit, hardware generation centers upon register transfers. The gating required for simple register transfers is illustrated in Figure 3.2. Transfers involving concatenated bits from several registers result in hardware configurations analogous to that shown in this figure.

When Boolean expressions are encountered, the compiler processes them in such a fashion that the logic necessary to implement the expressions is generated as they are analyzed. No attempt is made on the part of the compiler to minimize the expressions before their hardware realization is generated. Thus, the logic generated is a direct reflection of the expression analyzed and it becomes important that the machine designer write all Boolean expressions in minimal form. As a simple example, the expression $\overline{AB} + \overline{AB}$, if encountered by the compiler, would be realized by the following hardware:
Figure 3.2. Register Transfer Hardware
Of course, the expression should be reduced to $\overline{E}$, which requires no new hardware to realize.

When a logic unit is specified by a combinational logic subroutine, it is normal to specify declared registers as permanent inputs to the logic unit. Similarly, the output from the logic unit is normally gated into a single register or concatenation of registers. However, for large and expensive (in terms of gates) logic units such as adders or multipliers, it is sometimes desirable to be able to switch various inputs into them and route their outputs to several locations. This avoids duplicating the units in situations where they find multiple use in a machine. Input and output buses are introduced for this purpose.

Input buses are generated at the time their declaring statements are processed. To illustrate, suppose the section of declarative statements shown in Figure 3.3 (a) is encountered. Registers named AC, DA, and X are declared along with a bus named ABUS. Notice that X is a single flip-flop which will be used exclusively for controlling which registers are gated into ABUS. If X is 1,
D AC=4
D DA=4
D X=1
D ABUS=[X(1)*AC]DA

(a) Declarative Statements

(b) Hardware Generated

Figure 3.3: Input Busing Hardware
then the AC register is enabled into the bus, otherwise
the DA register is enabled into ABUS.

Figure 3.3 (b) is a schematic diagram showing the
hardware generated by this sequence of declarative state­
ments. Similar gating is generated for each bus declared.
Notice that any legitimate Boolean expression may be used
to enable registers into buses. Appropriate hardware to
realize the Boolean expression is generated as the de­
clarative statement is processed.

Output buses need not be declared. They are
handled automatically by the compiler. Combinational
logic unit outputs are gated with the proper control
pulse and routed to the correct register in the same
fashion as an ordinary register transfer. All registers
with multiple inputs are supplied with a bank of OR gates
on their input.
CHAPTER 4

THE COMPILER

The AHPL compiler actually consists of two programs. The first program reads in the processes all declarative statements, combinational logic functions, and subroutines. This subroutine compiler then passes its results to the second program, the main compiler. This data is passed to the main compiler via a scratch file stored on the disk file. The main compiler accepts the main AHPL architectural program as well as the subroutine data from the disk, and completes compilation of the entire proposed machine. Figure 4.1 illustrates how a typical run is assembled. The main compiler is discussed in the present chapter and the subroutine compiler in Chapter 5. Both programs were written in the SNOBOL programming language. A brief discussion of SNOBOL4 is given in Appendix C. Also see Griswold, Poage and Polonsky, 1968.

A syntax-directed compiler is a general program which accepts the semantic and syntactic specification of any language and translates any program in that language into machine code. The AHPL compiler incorporates the semantic and syntactic specifications of AHPL into the
Figure 4.1. Assembly of AHPL Compiler Run
program and cannot compile any other language. Therefore, strictly speaking, the AHPL compiler is not a syntax-directed compiler. However, the AHPL compiler more nearly resembles a syntax-directed compiler than any other commonly recognized general type of compiler.

**Signal Names**

If the compiler had been written in an algebraic language such as FORTRAN, it would have been necessary to use only integers for signal names, with some corresponding scheme for relating hardware elements to integer signal names. However, SNOBOL is a string manipulation language whose basic data element is a string of characters. There is no inherent advantage to using integers alone for signal names when programming in SNOBOL. For this reason, the compiler assigns to each hardware signal a descriptive name consisting of alphanumeric characters. A simple assignment scheme characterized by the following rules is employed,

1. Flip-flops within each register, all gates, control delay elements, and pulse generators are each numbered sequentially from 1.

2. Gate signal names are of the form "T"k or "F"k, where k is an integer. "T"k is the true output of gate number k, and "F"k the complemented output of gate number k.
3. Delay element signals are of the form "D"k, where k is an integer.

4. Flip-flop signals are of the form RNk or -RNk. Here, RN is the name of the register with which the given flip-flop is associated, and k is again an integer. Also, -RNk indicates the complemented or \( \overline{Q} \) output of flip-flop RNk. Flip-flops generated for use within the control unit have signal names "CUFF"k or "-CUFF"k, where CUFF implies control unit flip-flop and k is an integer.

5. Output signals from the special pulse generating elements used in asynchronous operations are of the form "PG"k, where k is an integer. It is obvious that registers may not be named "T", "F", "PG", or "CUFF".

**Interconnection File**

As the compiler proceeds with hardware generation, signal names are assigned according to the above scheme and the interconnections generated are stored in a special file. This "interconnection file" is composed of several sequences of variables which are operated as if they were arrays, and store input signals to all hardware elements. The interconnection file could have been constructed from true SNOBOL arrays. However, version 2.0 of SNOBOL 4 requires that all arrays be dimensioned prior to their use.
For the arrays used in the interconnection file, this is a serious drawback since any predefined array dimension would limit the number of corresponding hardware elements.

A special technique was employed to circumvent this problem. The necessary arrays were simulated by use of the special monadic indirectness operator ($) of SNOBOL. When applied to a name or string, the operator $ generates a variable that is the value of its operand. To illustrate its use in simulating arrays, suppose the variable "DLY" is to be a one-dimensional array which stores the input signals to all control delay elements used in designing a machine. Thus, if the "third" element of the "array" DLY has the value "T31", it would indicate that the third delay element contains as its single input signal the true output of gate 31. To simulate this array, a unique name is first generated for each array element. This unique name is generated by concatenating the "array" name with a subscripting integer separated by a colon. The colon serves no purpose other than to delimit segments of the string to be converted to a variable. In SNOBOL (not AHPL), the up-arrow (↑) is used for a single quotation mark and strings separated by a blank are concatenated. Do not confuse the SNOBOL quotation mark with the AHPL shift or rotate operator. Thus, "↑DLY:↑ 3" in SNOBOL generates a string "DLY:3". By applying the indirectness operator to
this string, a variable is generated which represents one
element in the simulated array DLY. Thus the SNOBOL state-
ment $(DLY:3) = T31$ assigns to the variable "DLY:3"
the string value "T31". To perform general subscripting,
of course, an index variable is kept. Suppose the variable
K has a current value of 3. Then, the SNOBOL statement
$(DLY:K) = T31$ would generate a string "DLY:3", convert
it to a variable, and assign as its value the string "T31".
This technique simulates a one-dimensional array named DLY
with no limit on the subscript. All entries to the inter-
connection file were stored using this technique.

Gate input signals are stored in a sequence of vari-
ables with names beginning "G:" followed by an integer.
The value of each of these variables is a string of input
signals. For example, the variable "G:47" may have the
value "T18 AC4". This indicates that gate number 47 has
two input signals, which are the true output of gate 18
and the one output of the fourth bit in the AC register.
This sequence of variables has associated with it a com-
ppanion sequence of variables whose names begin with "GT:",
indicating gate type. To illustrate, the variable "GT:47"
may contain the string "\wedge". This would indicate that gate
47 is an AND gate. The symbol "\vee" would have indicated
OR gate.
The variable sequences beginning with "DLY:" and "PG:" are similar to "G:". Entries are signal names representing the input signal to each delay or pulse generating element respectively. Contrary to the case of "G:" entries to these sequences must be single signals due to the nature of the associated hardware elements. For example, the variable "PG:21" may contain "T18", indicating that the twenty-first pulse synchronizer's input comes from gate 18.

For each register declared, a separate sequence of variables is generated. Each variable name begins with the register name and is followed by not one, but two indices. First, it is necessary to distinguish between the SET and CLEAR inputs on all flip-flops. Second, it is desirable to be able to reference individual flip-flops within each register. Thus, suppose a register named AC had been declared, and K currently is equal to four. Then, the statement $(-t-AC:+ K +SET+) = +F14+$ would assign the false output of gate 14 to the SET input of the fourth flip-flop of the AC register.

In summary, there are "arrays" named "G", "DLY", and "PG", plus an "array" for each register declared, which are used to store input signals to associated hardware elements. In addition, the "array" "GT" stores a gate type indicator for each gate employed,
Description of the Main Compiler

Figure 4.2 is a rough block diagram of the main compiler program. A complete listing is given in Appendix A. The remainder of the chapter describes how the compiler works.

The first 28 statements define functions and patterns and initialize parameters used in the compiler. Patterns are special character structures to be searched for in various strings by SNOBOL. Statement 29 through 50 read data from the disk file and store it for future reference. This data was, of course, generated and deposited on the disk by the subroutine compiler. The next three statements initialize the card reading process.

The statement labeled "ORIGIN" marks the beginning of processing on each incoming statement. If the input card file is empty, the program branches to the statement labeled "LIST". Otherwise, as many cards as necessary to form the next complete statement are read in. Analysis of incoming statements begins at the statement labeled "ANALYZE".

Analysis of incoming statements is accomplished by use of the pattern-matching facility of SNOBOL. A desired pattern or character structure is defined and matched with the string to be analyzed. From this pattern-matching statement, SNOBOL allows program branching to locations
Figure 4.2. Compiler Block Diagram
dependent upon the success or failure of the pattern match. Another feature of SNOBOL, deferred variable assignment, allows segments of the string under analysis to be assigned to variables upon successful pattern matching. Using this feature, fractions of strings may be separated and stored for further analysis. Thus, by applying a succession of pattern matches to an incoming statement, the compiler proceeds to work down a syntactic tree until sufficient information is available to generate the hardware specified by the statement.

By examination of the compiler key, the statement's principal syntactic type is determined and the appropriate branch taken (to DCLR, BNCH, SYNC, or ASYN). In the same process, the incoming statement's label number if stored and, in the case of assignment statements, the strings on either side of the equals sign are stored. Statements 74 through 80 cause an appropriate message to be printed upon detection of certain errors. Any incoming declarative statements encountered by the main program are processed by statements 81 (DCLR) through 84. Of course, declarative statements are normally processed by the subroutine compiler. However, it is possible to declare a register (not a bus) at any time during the program prior to using any bit of the register in other statements.
Statements 85 (BNCH) through 94 process branch type statements. An unconditional branch requires only that the control pulse be routed to the correct location. A conditional branch requires more analysis, including the hardware realization of associated Boolean expressions, before routing the gated control pulses. Asynchronous assignment statements are processed by statements 95 (ASYN) to 113. Memory read, memory write, and asynchronous subroutine calls are the statements of this type allowed, and the control unit hardware for all three is identical (see Chapter 3). For subroutine calls, necessary interconnections between subroutine logic and the rest of the machine are then formed.

Statements 114 (SYNC) through 140 process synchronous assignment statements, the most involved of the four basic types of statements. After generating the control unit hardware (specified number of delay elements concatenated), the register transfer, conditional transfer, or subroutine call is processed. Subroutine calls again require only that necessary interconnections between subroutine logic and the rest of the machine be formed. Conditional transfers require the realization of several possible register transfers. Each of these is enabled by the output of the hardware realization of the corresponding Boolean expression. The essence of processing register
transfers lies in two programmer-defined functions (TRANSFER and BLDVEC) which are discussed below.

Upon completion of processing of any statement, the programs branches back to "ORIGIN" where the process begins anew for each incoming statement. The next large segment of the compiler, statements 141 through 258, is comprised of six programmer-defined functions. Their names are CXSUB, ROUTE, GEN, BLDVEC, DISK, TRANSFER, and BOOEXP. These functions are analogous to subroutines in other high level programming languages such as FORTRAN. They are discussed individually in the paragraphs below. The remainder of the program, statements 259 through 309, is basically a listing routine which prints the interconnection file. In addition, any required buses on the inputs to registers are generated as the register inputs are printed. The compiler branches to LIST upon completion of processing on the final statement of the input AHPL program.

**Programmer Defined Functions**

The brief, non-recursive function GEN will generate a single gate when called; i.e., information regarding the new gate is entered into the interconnection file. This function is called with two arguments separated by a comma. The first specifies gate type and must be either "A" or "V" (inverters are not used in the AHPL compiler system since the true and complemented version of every signal
is available). The second argument is a string specifying input signals to the new gate. For example, the function call GEN(\uparrow V, T31 F73) would generate an OR with two input signals, T31 and F73. GEN returns the number of the new gate.

The function ROUTE is also brief and non-recursive. It is called when processing control branch type AHPL statements and is used to route the control pulse to the proper location within the control unit hardware. The first argument specifies the statement to which control is to be passed, and the second argument is the representing the signal control pulse itself. If control is to branch forward to some statement not yet processed, it is necessary to store this information for use when the upcoming statement is processed. Using the same technique employed for the interconnection file arrays, an array named "I.P" is simulated to store input control pulses to upcoming statements. ROUTE generates no hardware when called.

The five-statement function DISK reads a single piece of data from the disk file each time it is called. The subroutine compiler deposits data on the disk in 50 character records. However, some data strings exceed 50 characters and therefore require multiple records. This necessitated the use of data delimiters. The symbol "§" was added to the end of each piece of data before being
loaded onto the disk file. DISK reads and concatenates records from the disk file until the delimiter is encountered. DISK then returns the new piece of data to the calling statement.

When a combinational logic subroutine call is encountered in the main AHPL program, the function CXSUB is called. If the subroutine called has output signals which must be properly connected into the rest of the machine, CXSUB forms these connections. Otherwise, CXSUB takes no action whatsoever. The function is called with two arguments, the first of which specifies the register to which the logic unit's output is to be gated. The second argument is the string of output signals itself. CXSUB merely AND's the logic unit output signals with an enabling control pulse and then routes them into the correct register.

The process of generating hardware to execute a transfer specified by a synchronous assignment statement is broken into two stages. Once completely analyzed, these statements result in the bits specified by the right-hand side expression being transferred to the flip-flops specified by the left-hand side expression. Each of these expressions, however, may be complicated concatenations of certain bits selected by AHPL variables. Thus, the first step is to determine precisely which bits are to be transferred to which flip-flops. Then, the hardware to execute
the transfer may be built. Two companion functions named TRANSFER and BLDVEC perform this processing.

The non-recursive function TRANSFER accepts three arguments. The first is the left-hand side of the equals sign; i.e., the concatenation of variables to which transfer is to be made. The second is the right-hand side of the equals sign; i.e., the concatenation of variables specifying bits to be transferred. The third argument is the enabling control pulse for the transfer. TRANSFER strips off individual variables from the concatenated expressions and calls BLDVEC to analyze them. BLDVEC returns a string which consists of the signal names of the selected bits separated by asterisks. BLDVEC's operation is discussed below. With this bit-wise representation of each variable, TRANSFER proceeds to generate the hardware shown in Figure 3.2 by taking each individual bit, ANDing it with the enable pulse, and sending the output to the correct flip-flop input. Finally, if a transfer appears in an AHPL program more than once, the hardware for this transfer is generated only at the first occurrence of the transfer. All future occurrences merely result in the enabling control pulse being ORed into existing hardware generated at the first occurrence.

As can be seen, the actual analysis of AHPL variables occurs in the function BLDVEC. It is called with
only one argument: the variable to be analyzed. It returns a string which is a bit-wise representation of the variable analyzed. The string consists of the signal names of the correct bits separated by asterisks. In effect, BLDVEC builds a vector specified by a variable. For example, suppose a four-bit AC register had been declared. Then, BLDVEC would return the string "AC1*AC2*AC3*AC4*" upon analysis of the variable "AC". Whereas, the AHPL variable "2↑AC" would be returned from BLDVEC as "AC3*AC4*AC1*AC2*".

In order to handle certain variables in a more elegant and faster manner, BLDVEC may operate recursively. To illustrate, suppose the variable analyzed is "SUFFIX(2)/PREFIX(3)/AC", where again the AC register is 4 bits in length. This variable specifies the center 2 bits of the AC register. The first call of BLDVEC stores "SUFFIX(2)" and calls BLDVEC again to analyze the variable "PREFIX(3)/AC". This time, BLDVEC stores "PREFIX(3)" and calls itself again to analyze "AC". The third call generates the string "AC1*AC2*AC3*AC4*" and passes it back up. At the second level, the first 3 bits are stripped and passed up again. Finally, at the first level the last 2 bits of "AC1*AC2*AC3*" are stripped and the resulting string "AC2*AC3*" is passed back to the original calling statement.
To illustrate how an entire statement is processed, suppose the AHPL statement

```
31 S1 IR=SUFFIX(2)/PREFIX(3)/AC
```

were encountered with a four-bit AC and a two-bit IR registers declared. After adding a delay element to the control unit, TRANSFER would be called to build the transfer hardware. TRANSFER would first call BLDVEC to analyze "IR". The result, of course, would be "IR1*IR2*". Next, BLDVEC would analyze the right-hand side of the equals sign. As shown above, the result would be "AC2*AC3*". Using the hardware of Figure 3.2, TRANSFER would then transfer AC2 to IR1 and AC3 to IR2.

Finally, all Boolean expressions encountered during compilation are analyzed and the necessary logic to implement them generated by the function BOOEXP. This is a highly recursive function which has a single argument the expression to be analyzed. It returns the signal name of the gate whose output assumes the function value. BOOEXP recognizes the following precedence order for the three Boolean operators:

1. - (Complement)
2. \( \land \) (AND)
3. \( \lor \) (OR)

However, parentheses may be employed in the usual fashion to influence precedence.
To illustrate the operation of this function, suppose the following Boolean expression is to be realized:

\[ f = \neg(\text{IR}(3) \land \text{IR}(4)) \lor (\neg\text{IR}(1) \land \text{IR}(2) \land \neg\text{IR}(3)) \]

Figure 4.3 shows how the logic evolves as this function is analyzed. Every occurrence of the logical operators \( \land \) and \( \lor \) in an expression results in a gate being generated and BOOEXP calling itself to generate input signals to the gate. So, an OR gate is generated first, as shown in Figure 4.3(a). Next, BOOEXP is recursively called to analyze \(-\text{IR}(3) \land \text{IR}(4)\)

This results in the logic of Figure 4.3(b). Then BOOEXP begins analysis of \(\neg\text{IR}(1) \land \text{IR}(2) \land \neg\text{IR}(3)\). The first \( \land \) symbol results in the logic of Figure 4.3(c), and BOOEXP being recursively called down another level to analyze the remainder of the expression. Finally, the final AND gate is generated and the logic is complete as shown in Figure 4.3(d).

Notice that the logic realization of any expression depends entirely upon the expression as presented to the compiler by the AHPL programmer. No minimization techniques are applied to incoming expressions.
(a) Step 1

(b) Step 2

(c) Step 3

(d) Complete Logic

Figure 4.3. Realization of a Boolean Expression
CHAPTER 5

COMBINATIONAL LOGIC SUBROUTINES

As mentioned earlier, any logical computation with the result transferred into a register can be written in a single AHPL statement. However, when describing large and complex functions, the resulting statements are unwieldy and cumbersome. Combinational logic subroutines, as introduced in Chapter 7 of Hill and Peterson (to be published), provide notation for representing logic in a sequence of AHPL statements. These subroutines are typically used to generate logic units such as counters, adders, decoders, etc.

The writing of the combinational logic subroutine compiler was a major subtask of the AHPL compiler project. Details are given in a M. S. Thesis by Charles K. Goddard (1971). For completeness, a list of the subroutine compiler is reproduced in Appendix B. Programming necessary to integrate the subroutine compiler into the AHPL compiler system, as well as input bus programming, was added by the author. The remainder of this chapter will center upon the use of these subroutines.

There are actually two distinct types of subroutines which may be used to specify a logic unit. In addition to
the regular subroutines, which are called from the main compiler, the system also provides for use of function subroutines which are called from regular subroutines. Function subroutines are used to specify blocks of logic which are repeated many times in the construction of a logic unit. The calling subroutine supplies arguments which specify each time the proper signals to be used as the function generates logic. The use of function subroutines is illustrated in an example below.

Combinational logic subroutines may specify hardware designed to operate in either a synchronous or asynchronous fashion with regard to the machine's control unit. Based upon the speed of the logic employed, the designer may be able to predict the maximum propagation delay for a given logic unit. This unit would then be called from the main program with a synchronous assignment statement, and the control unit appropriately specified. On the other hand, a given logic unit may require an uncommonly large delay or have a propagation delay time which varies over a wide range of values depending upon the input vectors. In these situations, an asynchronous subroutine should be specified and called from the main program with an asynchronous assignment statement.

The use of combinational logic subroutines is best illustrated with an example. One of the most common
logic units in digital computers is the parallel binary adder. A simple approach to the design of such adders is the so-called "ripple-carry" adder. For adding two n-bit binary vectors, this combinational logic unit consists of n full adder units interconnected as in Figure 5.1. For position i, the full adder (FA) is an implementation of the equations

\begin{align*}
(1) \quad S_i &= \overline{x_i} \overline{y_i} c_{i+1} \lor x_i y_i \overline{c_i} \lor x_i \overline{y_i} c_{i+1} \lor x_i y_i c_{i+1} \\
(2) \quad c_i &= x_i y_i \lor x_i c_{i+1} \lor y_i c_{i+1}
\end{align*}

Here, as in the figure, X and Y are input-bit strings. S is the sum string and C the carry-bit string.

An AHPL subroutine which specifies a simple 4-bit adder may be found in Figure 5.2. The function subroutine specifies the logic for a single full adder unit. This function is then called as required by a regular subroutine which generates the complete adder unit.

Figure 5.2(a) is the full adder function. The initial statement identifies what follows as a function whose name is FULLADD with three arguments. As in the main routine, statements are labeled for branching purposes with an integer in the first five columns. A compiler key follows the label, but the keys differ from those in the main routine. The letter S implies signal flow statement, B implies bookkeeping statement, and T a termination.
Figure 5.1. Ripple-carry Adder

(a) FULLADD Function

<table>
<thead>
<tr>
<th>FUNC</th>
<th>FULLADD(A,B,C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>S</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

SO=\((-A\land B\lor C)\)  
CO=(A\land B)\lor (A\land C)\lor (B\land C)  
FULLADD(A,B,C)=CO,SO

(b) ADDER Subroutine

<table>
<thead>
<tr>
<th>SUBR</th>
<th>ADDER(AC,MD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

C[5]=0  
I=4  
K=I+1  
C[I],S[I]=FULLADD(AC[I],MD[I],C[K])  
I=I-1  
i:1    (LT,GE) +\(7,3\)  
ADDER(AC,MD)=S  
END

Figure 5.2. Subroutines for 4-Bit Adder
statement. As in the main routine, one or more blanks separate the key and the remainder of the statement.

Statement 1, a signal flow statement specifies the hardware for the sum bit of the unit. Statement 2 specifies the hardware for the carry bit. In order to return them to the calling subroutine, the sum and carry bits are assigned to be the value of FULLADD in statement 3. Statement 4 ends the function. The last two statements are examples of termination statements.

Figure 5.2(b) is the adder subroutine. It generates the hardware to add the contents of the AC and MD registers. Again, the initial statement identifies what follows as a subroutine named ADDER with two arguments. The purpose of statement 1 is to zero the carry input to the first full adder unit. Statement 2 initializes an index variable, and is an example of a bookkeeping statement. Statement 3 also, initializes an index variable. In general, these indices and bookkeeping statements are used to construct and control program loops. Statement 4 calls the function FULLADD with proper arguments and assigns the returned sum and carry bits to $S_I$ and $C_I$ respectively. The index variable $I$ is decremented in statement 5. In statement 6, $I$ is compared with zero and the program branches back to 3 if $I$ is still greater than zero or to 7 otherwise. Statement 7 assigns the bits of the vector
S to ADDER for storage until ADDER is called in the main routine. The subroutine ends at statement 8.

A ripple carry adder of any dimension could be specified with these routines by merely initializing I correctly in statement 2 and $C_{I+1}$ in statement 1. The subroutine could be called from the main compiler with the following statement.

31 SI AC=ADD(AC,MD)

In this example, inputs were taken directly from declared registers and no control pulse was required to enable the subroutine. Of course, input busing may be used to gate several different vectors into a given logic unit. In these cases, the bus is declared along with the registers and the bus name is used in writing the subroutines instead of a particular register name. For example, suppose an input bus named ABUS had been declared for use with the adder unit specified in Figure 5.2. Then, if AC were replaced with ABUS throughout Figure 5.2(b), the logic unit so specified would add the contents of the MD register with the contents of whatever register or vector were gated into ABUS. For busing, of course, a separate synchronous assignment statement is required in the main program to set the control flip-flop for the bus.

Normally, the statement switching inputs to a bus directly precedes the subroutine call statement using the
bus. The compiler key for this statement should reflect the proper control delay necessary for the propagation delay of the logic unit whose input vectors were switched. For situations where this multiple control delay is necessary but cannot be specified via the statement preceding the subroutine call, a special 'no operation' statement should be inserted into the AHPL program just prior to the subroutine call. This special statement has the following syntactic definition:

\[
\text{NO OPERATION} := \langle \text{INTEGER} \rangle \langle \text{BLANKS} \rangle \cdot \langle \text{INTEGER} \rangle \langle \text{BLANKS} \rangle \cdot \text{NO OP}
\]

The first \langle \text{INTEGER} \rangle is, of course, the label. The compiler key \langle \text{INTEGER} \rangle specifies the number of control delays desired.

Frequently a logic unit requires a pulse from the control unit to activate it. For example, a counter may be specified which will be incremented upon receipt of the control pulse. For these situations, the subroutine compiler recognizes the reserved variable CPULSE as being a pulse from the control unit which will be supplied when the main AHPL program is compiled. Thus, the variable CPULSE may be used as a logic signal in any subroutine requiring a control pulse.

For asynchronous subroutines, a completion pulse must be generated. The variable RPULSE is reserved for these cases. When used, RPULSE results in a pulse generator
(synchronizer) being allocated whose output is passed to the main compiler for use when required. The input to RPULSE is specified within the subroutine by the designer via an assignment statement.

The subroutine compiler itself is a large SNOBOL program, as can be seen from Appendix B. It analyzes incoming statements in a fashion basically similar to the main compiler, however, it cannot generate the hardware for each statement as it is read in. This is due primarily to the looping facility of the subroutines as well as function calls. Instead, the subroutine compiler converts each incoming AHPIL statement into a SNOBOL statement or sequence of statements. When compilation of a function or subroutine ends, the so generated sequence of SNOBOL statements is converted to SNOBOL code. When executed, this new code generates the required hardware. Because the conversion to SNOBOL code is performed only once, the principal merit of this approach is its speed for subroutines with loops which are traversed a large number of times.

Finally, it should be noted that the subroutines are compiled before the main program and their hardware stored in the interconnection file. Thus, when called from the main program, it is necessary only to connect the logic unit into the central processing unit's hardware. Actually, function subroutines are converted into SNOBOL
code and stored for execution when called by a subroutine. The regular subroutines, however, are executed immediately after conversion to SNOBOL code.
CHAPTER 6

RESULTS

This chapter will consist of specifying in AHPL a machine of fairly realistic size and capability. This architectural program is then compiled by the AHPL compiler system and the results shown. An adder and incrementing logic units are specified with combinational logic subroutines to illustrate their usefulness.

**Machine Specification**

Suppose a certain machine has a 15-bit word length and eight basic instruction types. The first three bits of the instruction word indicate the instruction according to the following code.

<table>
<thead>
<tr>
<th>Bit</th>
<th>IR(1)</th>
<th>IR(2)</th>
<th>IR(3)</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>LAC (Load Accumulator)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>TAD (Twos Complement Add)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AND</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>DAC (Deposit Accumulator)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>JMP (Jump)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>INPUT</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>OUTPUT</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>OPERATE</td>
</tr>
</tbody>
</table>
Bit IR 4 = 1 indicates indirect addressing. The machine contains program counter, accumulator, instruction register, memory data, and memory address registers. They are abbreviated PC, AC, IR, MD, and MA respectively. The AHPL main program describing this machine may be found in Figure 6.1.

The first three statements fetch the next instruction from memory and load it into the IR. Statement 4 is a conditional branch to 23 for input instructions, 24 for output instructions, or 25 for operate instructions. This branch depends upon the value of bits one, two, and three of the IR as specified above. If indirect addressing is indicated, statements 6 through 8 accomplish the replacement of the last 11 bits of the IR. Statement 9 causes control to branch to 16 for DAC instructions, 19 for JMP instructions, and 10 otherwise. LAC, TAD, and AND instructions all require another memory reference executed in statement 10. For ADD, control then branches to 14 where the sum of the MD and AC registers as supplied by the adder unit is transferred to the AC. Statement 12 causes either the MD or the results of ANDing the MD and AC registers to be placed in the AC. Statements 16 and 17 deposit the contents of the AC into memory. Statement 19 transfers the MA to the PC register.
READ IN FROM DISK COMPLETED AT 8357

* AHPL ARCHITECTURAL DESCRIPTION OF 15-BIT MACHINE.

1 S1 MA=PC
2 A MD=MEMORY(MA)
3 S1 IR=MD
4 *25

5

6 S1 MA=SUFFIX(11)/IR
7 A MD=MEMORY(MA)
8 S1 SUFFIX(11)/IR=SUFFIX(11)/MD
9 *[-IR<1>^IR<2>^IR<3>*16+IR<1>^IR<2>^IR<3>*19]10
10 A MD=MEMORY(MA)
11 *[-IR<2>^IR<3>*14]12
12 S1 AC=[-IR<2>^IR<3>*MD]AC=MD
13 *21
14 S1 AC=ADD(MD,AC)
15 *21
16 S1 MD=AC
17 A MEMORY(MA)=MD
18 *21
19 S1 PC=SUFFIX(11)/IR
20 *1
21 S1 PC=INC(PC)
22 *1

*23 INPUT PROGRAMMING SEQUENCE Follows HERE.
*24 OUTPUT PROGRAMMING SEQUENCE Follows HERE.
*25 OPERATE PROGRAMMING SEQUENCE Follows HERE.

Figure 6.1. Main AHPL Program
Except for the JMP instruction, all instructions eventually result in control passing to statement 21 where the program counter is incremented before branching back to statement 1 for the next instruction.

The input data to the subroutine compiler is given in Figure 6.2. First, all machine registers are declared along with their lengths. Then, the subroutine ADD specifies the hardware for the adder unit. The adder specified is a ripple-carry unit similar to that described in Chapter 5. No input bus is employed. The adder receives inputs directly from the AC and MD registers. Finally, the subroutine INC specifies necessary hardware about the PC register to increment it upon receipt of a control pulse. Note that this input data is actually processed by the subroutine and the results passed to the main compiler before the main AHPL program is compiled. These subroutine listings were generated as they were compiled and the gates used in each subroutine also shown.

Results

The results of compilation of this machine may be found in Figure 6.3. The inputs to all registers are listed first. This is followed by a list of all gates employed. The first 120 gates comprise the adder and the next 24 gates are used in the incrementing logic for the PC register. The remaining 422 gates are used in transfers,
D  AC=15
D  MD=15
D  IR=15
D  MA=11
D  PC=11
SUBR  ADD(MD,AC)
1  S  C[16]=0
2  B  J=15
3  B  K=J+1
6  B  J=J-1
7  B  J=0 (LE,GT) *(B,3)
8  T  ADD(MD,AC)=SUM
9  T  END

GATES 0-120 IN ADD
SUBR  INC(PC)
1  B  J=1
2  S  TOG=CPULSE
3  S  PC[J]=(TOG∧PC[J])
4  S  ¬PC[J]=(TOG∧PC[J])
5  S  TOG=(PC[J]∧TOG)
6  B  J=J+1
7  B  J=11 (LE,GT) *(B,3)
8  T  END

GATES 120-154 IN INC

Figure 6.2. Subroutine Compiler Input Data
<table>
<thead>
<tr>
<th>REGISTER F/F</th>
<th>SET</th>
<th>CLEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUFF1</td>
<td>T179</td>
<td>T180</td>
</tr>
<tr>
<td>CUFF2</td>
<td>T251</td>
<td>T252</td>
</tr>
<tr>
<td>CUFF3</td>
<td>T285</td>
<td>T286</td>
</tr>
<tr>
<td>CUFF4</td>
<td>T422</td>
<td>T423</td>
</tr>
<tr>
<td>AC1</td>
<td>T452</td>
<td>T453</td>
</tr>
<tr>
<td>AC2</td>
<td>T454</td>
<td>T455</td>
</tr>
<tr>
<td>AC3</td>
<td>T456</td>
<td>T457</td>
</tr>
<tr>
<td>AC4</td>
<td>T458</td>
<td>T459</td>
</tr>
<tr>
<td>AC5</td>
<td>T460</td>
<td>T461</td>
</tr>
<tr>
<td>AC6</td>
<td>T462</td>
<td>T463</td>
</tr>
<tr>
<td>AC7</td>
<td>T464</td>
<td>T465</td>
</tr>
<tr>
<td>AC8</td>
<td>T466</td>
<td>T467</td>
</tr>
<tr>
<td>AC9</td>
<td>T468</td>
<td>T469</td>
</tr>
<tr>
<td>AC10</td>
<td>T470</td>
<td>T471</td>
</tr>
<tr>
<td>AC11</td>
<td>T472</td>
<td>T473</td>
</tr>
<tr>
<td>AC12</td>
<td>T474</td>
<td>T475</td>
</tr>
<tr>
<td>AC13</td>
<td>T476</td>
<td>T477</td>
</tr>
<tr>
<td>AC14</td>
<td>T478</td>
<td>T479</td>
</tr>
<tr>
<td>AC15</td>
<td>T480</td>
<td>T481</td>
</tr>
<tr>
<td>MD1</td>
<td>T392</td>
<td>T393</td>
</tr>
<tr>
<td>MD2</td>
<td>T394</td>
<td>T395</td>
</tr>
<tr>
<td>MD3</td>
<td>T396</td>
<td>T397</td>
</tr>
<tr>
<td>MD4</td>
<td>T398</td>
<td>T399</td>
</tr>
<tr>
<td>MD5</td>
<td>T400</td>
<td>T401</td>
</tr>
<tr>
<td>MD6</td>
<td>T402</td>
<td>T403</td>
</tr>
<tr>
<td>MD7</td>
<td>T404</td>
<td>T405</td>
</tr>
</tbody>
</table>

Figure 6.3. Compiler Output
<table>
<thead>
<tr>
<th>MD8</th>
<th>T406</th>
<th>T407</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD9</td>
<td>T408</td>
<td>T409</td>
</tr>
<tr>
<td>MD10</td>
<td>T410</td>
<td>T411</td>
</tr>
<tr>
<td>MD11</td>
<td>T412</td>
<td>T413</td>
</tr>
<tr>
<td>MD12</td>
<td>T414</td>
<td>T415</td>
</tr>
<tr>
<td>MD13</td>
<td>T416</td>
<td>T417</td>
</tr>
<tr>
<td>MD14</td>
<td>T418</td>
<td>T419</td>
</tr>
<tr>
<td>MD15</td>
<td>T420</td>
<td>T421</td>
</tr>
<tr>
<td>IR1</td>
<td>T183</td>
<td>T184</td>
</tr>
<tr>
<td>IR2</td>
<td>T185</td>
<td>T186</td>
</tr>
<tr>
<td>IR3</td>
<td>T187</td>
<td>T188</td>
</tr>
<tr>
<td>IR4</td>
<td>T189</td>
<td>T190</td>
</tr>
<tr>
<td>IR5</td>
<td>T482</td>
<td>T483</td>
</tr>
<tr>
<td>IR6</td>
<td>T484</td>
<td>T485</td>
</tr>
<tr>
<td>IR7</td>
<td>T486</td>
<td>T487</td>
</tr>
<tr>
<td>IR8</td>
<td>T488</td>
<td>T489</td>
</tr>
<tr>
<td>IR9</td>
<td>T490</td>
<td>T491</td>
</tr>
<tr>
<td>IR10</td>
<td>T492</td>
<td>T493</td>
</tr>
<tr>
<td>IR11</td>
<td>T494</td>
<td>T495</td>
</tr>
<tr>
<td>IR12</td>
<td>T496</td>
<td>T497</td>
</tr>
<tr>
<td>IR13</td>
<td>T498</td>
<td>T499</td>
</tr>
<tr>
<td>IR14</td>
<td>T500</td>
<td>T501</td>
</tr>
<tr>
<td>IR15</td>
<td>T502</td>
<td>T503</td>
</tr>
<tr>
<td>MA1</td>
<td>T504</td>
<td>T505</td>
</tr>
<tr>
<td>MA2</td>
<td>T506</td>
<td>T507</td>
</tr>
<tr>
<td>MA3</td>
<td>T508</td>
<td>T509</td>
</tr>
<tr>
<td>MA4</td>
<td>T510</td>
<td>T511</td>
</tr>
<tr>
<td>MA5</td>
<td>T512</td>
<td>T513</td>
</tr>
<tr>
<td>MA6</td>
<td>T514</td>
<td>T515</td>
</tr>
<tr>
<td>MA7</td>
<td>T516</td>
<td>T517</td>
</tr>
<tr>
<td>MA8</td>
<td>T518</td>
<td>T519</td>
</tr>
<tr>
<td>MA9</td>
<td>T520</td>
<td>T521</td>
</tr>
<tr>
<td>MA10</td>
<td>T522</td>
<td>T523</td>
</tr>
</tbody>
</table>

Figure 6.3, Compiler Output (Continued)
<table>
<thead>
<tr>
<th>GATE TYPE</th>
<th>INPUT SIGNALS</th>
<th>GATE TYPE</th>
<th>INPUT SIGNALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>^ MD15 AC15</td>
<td>2</td>
<td>^ MD15 ZERO</td>
</tr>
<tr>
<td>3</td>
<td>^ AC15 ZERO</td>
<td>4</td>
<td>v T1 T2 T3</td>
</tr>
<tr>
<td>5</td>
<td>v MD15 AC15 ZERO</td>
<td>6</td>
<td>^ F4 T5</td>
</tr>
<tr>
<td>7</td>
<td>^ MD15 AC15 ZERO</td>
<td>8</td>
<td>v T6 T7</td>
</tr>
<tr>
<td>9</td>
<td>^ MD14 AC14</td>
<td>10</td>
<td>^ MD14 T4</td>
</tr>
<tr>
<td>11</td>
<td>^ AC14 T4</td>
<td>12</td>
<td>v T9 T10 T11</td>
</tr>
<tr>
<td>13</td>
<td>v MD14 AC14 T4</td>
<td>14</td>
<td>^ F12 T13</td>
</tr>
<tr>
<td>15</td>
<td>^ MD14 AC14 T4</td>
<td>16</td>
<td>v T14 T15</td>
</tr>
<tr>
<td>17</td>
<td>^ MD13 AC13</td>
<td>18</td>
<td>^ MD13 T12</td>
</tr>
<tr>
<td>19</td>
<td>^ AC13 T12</td>
<td>20</td>
<td>v T17 T18 T19</td>
</tr>
<tr>
<td>21</td>
<td>v MD13 AC13 T12</td>
<td>22</td>
<td>^ F20 T21</td>
</tr>
<tr>
<td>23</td>
<td>^ MD13 AC13 T12</td>
<td>24</td>
<td>v T22 T23</td>
</tr>
<tr>
<td>25</td>
<td>^ MD12 AC12</td>
<td>26</td>
<td>^ MD12 T20</td>
</tr>
<tr>
<td>27</td>
<td>^ AC12 T20</td>
<td>28</td>
<td>v T25 T26 T27</td>
</tr>
<tr>
<td>29</td>
<td>v MD12 AC12 T20</td>
<td>30</td>
<td>^ F28 T29</td>
</tr>
<tr>
<td>31</td>
<td>^ MD12 AC12 T20</td>
<td>32</td>
<td>v T30 T31</td>
</tr>
<tr>
<td>33</td>
<td>^ MD11 AC11</td>
<td>34</td>
<td>^ MD11 T28</td>
</tr>
</tbody>
</table>

Figure 6.3. Computer Output (Continued)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 35 | ^ | AC11 T28 | 36 | v | T33 T34 T35 |
| 37 | v | MD11 AC11 T28 | 38 | ^ | F36 T37 |
| 39 | ^ | MD11 AC11 T28 | 38 | v | T38 T39 |
| 40 | v | MD10 AC11 T28 | 40 | v | T41 T42 T43 |
| 42 | v | MD10 AC11 T28 | 42 | ^ | F44 T45 |
| 44 | v | MD10 AC11 T28 | 44 | v | F46 T47 |
| 46 | v | MD10 AC11 T28 | 46 | v | T49 T50 T51 |
| 48 | v | MD10 AC11 T28 | 48 | v | T52 |
| 50 | v | MD9 AC9 | 50 | v | T54 T55 |
| 52 | v | MD9 AC9 | 52 | v | T56 T57 T58 T59 |
| 54 | v | MD9 AC9 | 54 | v | T60 T61 |
| 56 | v | MD9 AC9 | 56 | v | T62 T63 |
| 58 | v | MD9 AC9 | 58 | v | T65 T66 T67 |
| 60 | v | MD9 AC9 | 60 | v | T68 T69 |
| 62 | v | MD9 AC9 | 62 | v | T70 T71 |
| 64 | v | MD9 AC9 | 64 | v | T73 T74 T75 |
| 66 | v | MD9 AC9 | 66 | v | T76 T77 |
| 68 | v | MD9 AC9 | 68 | v | T78 T79 |
| 70 | v | MD9 AC9 | 70 | v | T81 T82 T83 |
| 72 | v | MD9 AC9 | 72 | v | T84 T85 |
| 74 | v | MD9 AC9 | 74 | v | T86 T87 |
| 76 | v | MD9 AC9 | 76 | v | T89 T90 T91 |
| 78 | v | MD9 AC9 | 78 | v | T92 T93 |
| 80 | v | MD9 AC9 | 80 | v | T94 T95 |
| 82 | v | MD9 AC9 | 82 | v | T96 T97 T98 T99 |

Figure 6.3. Computer Output (Continued)
Figure 6.3. Computer Output (Continued)
Figure 6.3. Computer Output (Continued)
Figure 6.3, Computer Output (Continued)
Figure 6.3. Computer Output (Continued)
Figure 6.3. Computer Output (Continued)
<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>431</td>
<td>IR7 T426</td>
</tr>
<tr>
<td>432</td>
<td>IR7 T426</td>
</tr>
<tr>
<td>433</td>
<td>IR8 T426</td>
</tr>
<tr>
<td>434</td>
<td>IR8 T426</td>
</tr>
<tr>
<td>435</td>
<td>IR9 T426</td>
</tr>
<tr>
<td>436</td>
<td>IR9 T426</td>
</tr>
<tr>
<td>437</td>
<td>IR10 T426</td>
</tr>
<tr>
<td>438</td>
<td>IR10 T426</td>
</tr>
<tr>
<td>439</td>
<td>IR11 T426</td>
</tr>
<tr>
<td>440</td>
<td>IR11 T426</td>
</tr>
<tr>
<td>441</td>
<td>IR12 T426</td>
</tr>
<tr>
<td>442</td>
<td>IR12 T426</td>
</tr>
<tr>
<td>443</td>
<td>IR13 T426</td>
</tr>
<tr>
<td>444</td>
<td>IR13 T426</td>
</tr>
<tr>
<td>445</td>
<td>IR14 T426</td>
</tr>
<tr>
<td>446</td>
<td>IR14 T426</td>
</tr>
<tr>
<td>447</td>
<td>IR15 T426</td>
</tr>
<tr>
<td>448</td>
<td>IR15 T426</td>
</tr>
<tr>
<td>449</td>
<td>DE8</td>
</tr>
<tr>
<td>450</td>
<td>T372 T389 T424</td>
</tr>
<tr>
<td>451</td>
<td>DE9</td>
</tr>
<tr>
<td>452</td>
<td>T309 T342 T374</td>
</tr>
<tr>
<td>453</td>
<td>T310 T343 F374</td>
</tr>
<tr>
<td>454</td>
<td>T311 T344 T375</td>
</tr>
<tr>
<td>455</td>
<td>T312 T345 F375</td>
</tr>
<tr>
<td>456</td>
<td>T313 T346 T376</td>
</tr>
<tr>
<td>457</td>
<td>T314 T347 F376</td>
</tr>
<tr>
<td>458</td>
<td>T315 T348 T377</td>
</tr>
<tr>
<td>459</td>
<td>T316 T349 F377</td>
</tr>
<tr>
<td>459</td>
<td>T317 T350 T378</td>
</tr>
<tr>
<td>461</td>
<td>T318 T351 F378</td>
</tr>
<tr>
<td>462</td>
<td>T319 T352 T379</td>
</tr>
<tr>
<td>463</td>
<td>T320 T353 F379</td>
</tr>
<tr>
<td>464</td>
<td>T321 T354 T380</td>
</tr>
<tr>
<td>465</td>
<td>T322 T355 F380</td>
</tr>
<tr>
<td>466</td>
<td>T323 T356 T381</td>
</tr>
<tr>
<td>467</td>
<td>T324 T357 F381</td>
</tr>
<tr>
<td>468</td>
<td>T325 T358 T382</td>
</tr>
<tr>
<td>469</td>
<td>T326 T359 F382</td>
</tr>
<tr>
<td>470</td>
<td>T327 T360 T383</td>
</tr>
<tr>
<td>471</td>
<td>T328 T361 F383</td>
</tr>
<tr>
<td>472</td>
<td>T329 T362 T384</td>
</tr>
<tr>
<td>473</td>
<td>T330 T363 F384</td>
</tr>
<tr>
<td>474</td>
<td>T331 T364 T385</td>
</tr>
<tr>
<td>475</td>
<td>T332 T365 F385</td>
</tr>
<tr>
<td>476</td>
<td>T333 T366 T386</td>
</tr>
<tr>
<td>477</td>
<td>T334 T367 F386</td>
</tr>
<tr>
<td>478</td>
<td>T335 T368 T387</td>
</tr>
<tr>
<td>479</td>
<td>T336 T369 F387</td>
</tr>
<tr>
<td>480</td>
<td>T337 T370 T388</td>
</tr>
<tr>
<td>481</td>
<td>T338 T371 F388</td>
</tr>
<tr>
<td>482</td>
<td>T191 T255</td>
</tr>
<tr>
<td>483</td>
<td>T192 T256</td>
</tr>
<tr>
<td>484</td>
<td>T193 T257</td>
</tr>
<tr>
<td>485</td>
<td>T194 T258</td>
</tr>
<tr>
<td>486</td>
<td>T195 T259</td>
</tr>
<tr>
<td>487</td>
<td>T196 T260</td>
</tr>
<tr>
<td>488</td>
<td>T197 T261</td>
</tr>
<tr>
<td>489</td>
<td>T198 T262</td>
</tr>
<tr>
<td>490</td>
<td>T199 T263</td>
</tr>
<tr>
<td>491</td>
<td>T200 T264</td>
</tr>
<tr>
<td>492</td>
<td>T201 T265</td>
</tr>
<tr>
<td>493</td>
<td>T202 T266</td>
</tr>
<tr>
<td>494</td>
<td>T203 T267</td>
</tr>
<tr>
<td>495</td>
<td>T204 T268</td>
</tr>
</tbody>
</table>

**Figure 6.3.** Computer Output (Continued)
AHPL.VER.3 04/14/71 EXECUTE TIME IN MSEC = 83355

Figure 6,3. Computer Output (Continued)
03/31/71 SCOPE 3.2.0 VER. 0 UNIVERSITY OF ARIZONA

21.54.41. GENTRUC
21.54.41. GENTRY, T200, CM140000, BN3900332F.
21.54.41. ATTACH, SNOBOL, SNOBOL.
21.54.42. CYCLE **, SNOBOL
21.54.42. PFN FOUND IN SD 007
21.54.42. CYCLE 01, SNOBOL
21.54.42. FILE HAS BEEN ATTACHED
21.54.42. SNOBOL.
21.54.45. ROLLOUT COMPLETED. (FL 140000)
21.56.58. ROLLIN COMPLETED.
21.57.00. ROLLOUT COMPLETED. (FL 140000)
22.02.24. ROLLIN COMPLETED.
22.03.04. END SNOBOL4
22.03.04. REWIND, SCRATCH.
22.03.04. RFL 100.
22.03.05. FL= 140000, CPU= 035.239, PPU= 005.270
22.03.05. RFL 140000.
22.03.05. FL= 000100, CPU= 035.239, PPU= 005.270
22.03.11. SNOBOL.
22.04.41. END SNOBOL4
22.04.41. FL= 140000, CPU= 119.376, PPU= 010.540

Figure 6.3. Computer Output (Continued)
buses, and the control unit. The input signals to the nine delay elements required are listed last.

The total CDC 6400 central processor time required to compile the entire machine including logic units was 120 seconds (see last page of Figure 6.3). The number of gates exceeds the hand design number for several reasons, as discussed in the following chapter.
CHAPTER 7

CONCLUSIONS

This chapter will suggest several improvements to the AHPL compiler and conclusions which may be drawn about the system. First, the use of SNOBOL as the high-level language in which to write the compiler is discussed. In this discussion, SNOBOL refers to SNOBOL4, Version 2.0, which is being run on the University of Arizona's CDC 6400 computer with an available memory of 140000 words. Following this, improvements to the compiler itself are mentioned before final conclusions are drawn.

SNOBOL4

When SNOBOL was originally developed, research in compiler techniques was one of its intended purposes. As mentioned in Chapter 2, it is very powerful when applied to syntax analysis problems. Also, the compilation of AHPL programs into hardware may be viewed almost entirely as a string manipulation problem and, of course, SNOBOL stands for StriNg Oriented symBOlic Language. Furthermore, because SNOBOL is a very high-level language, it afforded rapid compiler development and relieved programmers of
many tedious details. However, certain disadvantages must be pointed out.

First of all, SNOBOL4 is an interpreter and not a compiler. This means that no machine language version of the AHPL compiler is ever produced. Instead, it is run interpretively each time. SNOBOL runs in two distinct phases. It passes through a program once during the "compilation" phase converting the program into some internal representation. This is followed by the "execution" phase. The "compilation" phase cannot be bypassed. Although the AHPL compiler program remains unaltered from run to run, it is still "compiled" by SNOBOL each time. Running on the University of Arizona's CDC 6400 with a field length of 140000\textsubscript{8} words, the AHPL subroutine and main compiler programs combined consume approximately 10 seconds of central processor time merely "compiling" during each run. This is obviously an unproductive use of expensive computer time.

The SNOBOL compiler itself requires a large field length of approximately 70000\textsubscript{8} words. The memory area from the end of the SNOBOL compiler to the end of the declared field length is operated as a dynamic regenerative storage area by SNOBOL. All strings, patterns, arrays, data, and programming is stored within this storage area. In executing a program, when the end of the dynamic storage area
is reached, SNOBOL automatically calls a "garbage collection" routine. This routine attempts to compress the contents of the dynamic storage area by discarding any strings, patterns, etc. temporarily generated during execution. Experience with the AHPL compiler and a declared field length of 1400008 words indicates that each call of the collection routine consumes nearly one second of central processor time. This indicates a tradeoff between time saved by permanently storing patterns instead of generating them at each statement as needed, and the time lost by more frequent garbage collection calls due to memory lost by storing the patterns. Furthermore, it is entirely possible to completely exhaust all available memory, causing program termination. Another consideration is that compilation of large digital systems requires the AHPL compiler's interconnection file to grow proportionally large. All of these factors combined to produce a fairly severe memory restriction. As a result, the subroutine and main compiler were separated into two programs with the results of subroutine compilation passed to the main compiler via a scratch file on disk storage. This method consumes a small amount of computer time putting data out onto the disk and retrieving it. However, when the subroutine and main compilers were combined, the resulting program was huge and slow running due to frequent calls of the regenerative storage collection
routine. Furthermore, the interconnection file was more limited. A version of the main compiler which stands alone was produced. It is significantly smaller and faster than the complete system. It should therefore be used whenever subroutines are not required.

Finally, a glance at the output listings of Chapter 6 for compilation of a sample machine indicates that execution time is not especially short. Although every attempt was made to speed the AHPL compiler system, SNOBOL is relatively slow running.

In conclusion, it would appear that a production version of the AHPL compiler should be written in assembly language or some other language which produces a machine language version of the compiler. This should result in a smaller, faster compiler with larger capacity due to increased memory for the interconnection file.

Compiler Improvements

There are several improvements which could be made to the present compiler system which would result in a savings of gates. Perhaps the simplest is elimination of gates with single input signals. There are several situations in which the compiler allocates a gate with a single input signal in anticipation of a possible additional signal being added in the future. When this eventuality fails to occur, the design is completed with a single input
to such a gate. For example, when the hardware for a register transfer is generated, the enabling control pulse is routed into an OR gate the output of which is used to execute the transfer. Should the identical transfer be requested again, the only action necessary would be to route the second enable pulse into the OR gate generated above.

The highly recursive function which analyzes Boolean expressions operates by scanning for Boolean operators and generating a gate each time one is encountered. Given the expression \( f = AC(1) \land MD(7) \land MQ(3) \) to realize, it would generate the hardware shown in Figure 7.1(a). Obviously, \( f \) could be realized as in Figure 7.1(b) at a savings of one gate. A Boolean expression analyzer function was written which alleviated this problem, but it proved to be larger, slower, and less eloquent.

![Figure 7.1. Realization of a Simple Boolean Expression](image)
Also, once an input program has been completely compiled, the multitude of techniques for minimizing combinational logic could be applied throughout the machine. This would involve drastic amounts of computer time, however.

Another possible improvement to the compiler is to provide more extensive error-checking capability. When the compiler was written, no special attempt was made to check for all possible errors in the AHPL program being compiled. However, the compiler does detect and provide error messages for most of the common errors as a natural product of processing the incoming statements. One contemplated improvement would have the compiler keep account of all register transfers built. Then, as each new transfer is specified, the compiler would examine existing transfer paths to determine if the newly requested transfer could be effected via a combination of existing transfers. It would notify the designer of any such possibilities for his evaluation. Also, it should be noted that all timing considerations are left to the designer.

The assumption of AND-OR logic and SET-CLEAR flip-flops is obviously somewhat restricting. This limitation might be eliminated in several ways. Of course, one method would be to write the compiler in such a fashion that different hardware components were assumed available. Another
would be to convert the present compiler's output to Boolean equations and then implement them in different logic. This is not within the true domain of the compiler, however.

Finally, many features of APL which are useful for hardware specification are not implemented in the AHPL compiler. This was due to speed and memory restrictions. Obviously, the more features available to the designer, the more convenient the language becomes to him. AHPL as outlined in Hill and Peterson (to be published) contains every useful feature of APL for hardware specification plus a few additional features.

**Final Conclusions**

The choice of modified APL as the register transfer language seems completely satisfactory. Combined with the design philosophy described by Hill and Peterson (to be published), AHPL presents no apparent restrictions or clumsiness when specifying computer architecture. The logic designs produced by machine compare very favorably with those produced by hand compilation of the same AHPL programs. Although the compiler system is seemingly slow running and requires a large field length, this is primarily due to the fact that the compiler is written in the SNOBOL4 programming language. The use of combinational logic subroutines provides convenience and flexibility to the system. However, a subroutine should be written only when
the desired transfer results in an extremely cumbersome assignment statement in the main program. The advantages afforded by automatic logic design translators would seem to warrant their consideration as a viable alternative to logic design by man.
APPENDIX A

LISTING OF MAIN AHPL COMPILER
# MAIN COMPILER FOR A HARDWARE PROGRAMMING LANGUAGE.

```plaintext
OUTPUT(*CONTROL*"2",*(132A1)*)
INPUT(*SCRATCH*,4,50)
TRIM(SCRATCH) *0K* ? F(ERR2)
DEFINE(*DISK()STR»TMP«)  
DEFINE(*ROUTE(A.1,A.2)T,I,BK,WD«)  
DEFINE(*GEN(A.1,A.2)«)  
DEFINE(*BLOVEC(VARB)T.I,NA,NO,FLG«)  
DEFINE(*TRANSFER(A.1,A.2,A.3)«)  
DEFINE(*BOOEXP(B.E)«)  
DEFINE(*CXSUB(A.1,A.2)C.V,T.N«)  
INPULSE = ARRAY(200)
G.I = 0
D.I = 0
FF.I = 0
C.P = *START*
REG = FAIL
SRVEC = FAIL
RNAME = *CUFF«
BLANKS = SPAN(* *)
LETTERS = *ABCDEFGHIJKLMNOPQRSTUVWXYZ*
DIGITS = *0123456789*
ALPHANUM = LETTERS DIGITS
IDENTIFIER = SPAN(LETTERS)
```
INTEGER = SPAN(DIGITS)
BINOP = ANY(↑↓↑↓)
EOF =
ANCHOR = 1
NA = DISK()
IDENT(NA,↑EOR↑)
REG = REG v NA
RNAME = RNAME NA ↑↑
SNA = DISK()
T. I = 1

SET.R
$$(NA ↑↑ T. I ↑SET↑) = DISK()$$  
$$(NA ↑↑ T. I ↑CLR↑) = DISK()$$
T. I = LT(T. I, SNA) T. I + 1

SUB.R
DATA = DISK()
IDENT(DATA,↑EOR↑)
SNAME = DATA ↑↑
SRVEC = SRVEC v DATA
$$(DATA ↑↑1↑) = DISK()$$
$$(DATA ↑↑2↑) = DISK()$$
$$(DATA ↑↑3↑) = DISK()$$
$$(DATA ↑↑4↑) = DISK()$$

GAT.R
T. I = 1

TYP.R
$$(↑GT↑↑ T. I) = DISK()$$
$$(↑G↑↑ T. I) = DISK()$$
T. I = T. I + 1

R.END
G. I = T. I - 1
OUTPUT = READ IN FROM DISK COMPLETED AT TIME()

READI
IMAGE = TRIM(INPUT)
OUTPUT = IMAGE

ORIGIN
IDENT(EOF)
OUTPUT = LINE
IMAGE = LINE
READC
LINE = TRIM(INPUT)
LINE →→
LINE →→ = :F(ENDGAME)
OUTPUT = → LINE
IMAGE = IMAGE LINE :S(PRINT)
PRINT
OUTPUT = → LINE :F(ANALYZE)
ENDGAME
EOF = 1 :S(DCLR)
ANALYZE
IMAGE →O SPAN(→ →) = :F(ERR7)
IMAGE LEN(S) → STNO =
TRIM(STNO) ARBNO(→ →) INTEGER → STNO :F(ERR7)
ENABLE = → T → GEN(→V →,CP $:F(ERR7)
$($:F(ERR7)
CnP =
IMAGE →S INTEGER → DCOUNT BLANKS REM → STAT :F(SYSC)
IMAGE ARBNO(→ →) →→ REM → GOTO :S(BNCH)
IMAGE →A SPAN(→ →) BREAK(→→) → LSIDE →→ REM → RSIDE :S(SYSC)
ERR1
OUTPUT = →***<SYNTACTIC ERROR>> DETECTED AT → $:E(LASTNO \( :E(N)
ERR2
OUTPUT = →***SUBROUTINE COMPILER ERROR.*** → END
ERR3
OUTPUT = →***END OF DISC FILE ENCOUNTERED AT → $:E(LASTNO \( :E(N
ERR4
OUTPUT = →*** → VAR8 → IS ILLEGAL VARIABLE.*** → (ORIGIN)
ERR5
OUTPUT = →*** COMPILER ERROR. STNO = → $:E(LASTNO \( :E(LIST)
ERR6
OUTPUT = →***VECTORS INCOMPATIBLE.*** → END
ERR7
OUTPUT = →***STATEMENT LABEL MISSING.*** → END
DCLR
IMAGE IDENTIFIER → VECTOR → INTEGER → LENGTH :F(ERR1)
$VECTOR = LENGTH
RNAME = RNAME VECTOR →→
REG = REG → VECTOR :F(ORIGIN)
BNCH
GOTO →[→ BREAK(→→) → TEST → INTEGER → FGOTO :S(CBNCH)
ROUTE(GOTO, ENABLE)
CBNCH
DFLT = GEN(→→, ENABLE)
ROUTE(FGOTO, → T → G →)
TEST = TEST →→
LOOPA
TEST BREAK(→→) → OPTN →→ = :F(ORIGIN)
OPTN BREAK(↑↑↑) • BEXP ↑↑ INTEGER • GOTO :F(ERR1)
T.I = GEN(↑↑↑,booEXP(BEXP) ENABLE)
ROUTE(GOTO,↑ T↑ T.I)
$↑(G↑ DFLT) = $↑(G↑ DFLT) ↑ F↑ T.I :
(LOOPA)

ASYN
ANCHOR = 0
LSIDE ←MEMORY←
RSIDE ←MEMORY←
ANCHOR = 1
RSIDE ←SRVEC • T.V
$(T.V ↑↑ T.I) ARBNO(↑ T.I) INTEGER • T.I :
F(ASYN.2)
$↑(G↑ T.I) = $↑(G↑ T.I) ENABLE

ASYN.2
T.S = $(T.V ↑↑ T.I)
LSIDE BREAK(↑↑) • NA LEN(1) = :
F(ASYN.3)
CXSUB(NA,$(T.V ↑2↑))
CXSUB( LSIDE,$(T.V ↑3↑)) :
(DROPCAN)
CXSUB( LSIDE,$(T.V ↑2↑)) :
(DROPCAN)

ASYN.3

WCC
T.S = ↑ WCC↑
RCC
T.S = ↑ RCC↑

DROPCAN
ANCHOR = 1
FF.I = FF.I ↑ 1
$↑(↑CUFF↑ FF.I ↑SET↑) = ENABLE
C.P = ↑ T ↑ GEN(↑↑↑,↑(CUFF↑ FF.I T.S))
$↑(↑CUFF↑ FF.I ↑CLR↑) = C.P :
(ORIGIN)

SYNC
T.I = 1
SIG = ENABLE

MDELAY
D.I = D.I ↑ 1
$↑(↑DLY↑ D.I) = SIG
SIG = ↑ DE↑ D.I
T.I = LT(T.I,DCOUNT) T.I ↑ 1 :
S(MDELAY)
C.P = SIG
STAT ↑NO OP↑ :
S(ORIGIN)
STAT BREAK(↑↑) • LSIDE ↑→ REM • RSIDE ↑F(ERR1)

RSIDE ↑∥ BREAK(↑∥) • INSIDE ↑∥ REM • RVEC :
:S(C.ASSN)
RSIDE *SRVEC * T.V ARB :S(SUBR)
TRANSFER(LSIDE,RSIDE,ENABLE) :(ORIGIN)
CASSN
DFLT = GEN(ENABLE)
TRANSFER(LSIDE,RVEC,GEN T.V DFLT)
INSIDE = INSIDE ARM
LOOPB
INSIDE BREAK(MAYTRAN) • MAYTRAN BEXP T.V DFLT
MAYTRAN BREAK(MAYTRAN) • BEXP REM RVEC :F(ERR1)
T.V = BOOEXP(BEXP)
$(*G* DFLT) = $(*G* DFLT) T.V T.V
TRANSFER(LSIDE,RVEC,GEN(ENABLE T.V)) :(LOOPB)
SUBR
IDENT($(*T.V *:*)) *F(ERR5)
$(*T.V =$(*T.V) T.V)
SUB.1
LSIDE BREAK(MAYTRAN) • NA LEN(1) = :F(SUB.2)
CXSUR(NA,$(*T.V *:*))
CXSUB(LSIDE,$(*T.V =$(*T.V) T.V)
SUB.2
CXSUB(LSIDE,$(*T.V =$(*T.V) T.V)
C.V = 1
MAX = SA.1
CXSUB.1
A.2 ARBNO(ANY) (ANY(TF) INTEGER) • R.BIT = :F(RETURN)
T.N = GEN(ENABLE R.BIT)
$(*A.1 =$(*C.V SET) = $(*A.1 =$(*C.V SET) T.V T.N
$(*A.1 =$(*C.V CLR) = $(*A.1 =$(*C.V CLR) T.V T.N
C.V = LT(C.V MAX) C.V + 1 :F(RETURN)$S(CXSUB.1)
ROUTE
T.I = $(*T.I =$(*T.I)
IDENT(T.I) :S(FWD)
INTEGER(T.I) :F(FWD)
$(*T.I =$(*T.I) A.2 :F(RETURN)
FWD
$(*A.1 =$(*A.1) A.2 :F(RETURN)
DISK = DISK
FET
STR = TRIM(SCRATCH) :F(RETURN)
STR RTAB(1) • TMP <$* = TMP :S(LASTR)
DISK = DISK STR :S(FET)

100
LASTR  DISK = DISK STR
GEN  G.i = G.i + 1
$(\uparrow GT\uparrow G.i) = A.1
$(\uparrow G.i \uparrow G.i) = A.2
GEN = G.i

BLDVEC
BLDVEC =
T.i = 1
VARB \#\# REG . NA RPOS(0)
VARB \#\# REG . NA \(\uparrow \uparrow\) INTEGER \#\# NO \(\uparrow \uparrow\)
VARB SPAN(\(\uparrow 01\uparrow\)) RPOS(0)
VARB \#\# REG . NA BINOP . FLG \# REG . NO
VARB (\(\uparrow\) PREFIX\(\uparrow\) \(\uparrow\) SUFFIX\(\uparrow\)) . FLG \(\uparrow\) INTEGER \#\# NO \(\uparrow\) / / REM . NA
VARB INTEGER \#\# NO ANY(\#\#\#) . FLG \# REG . NA RPOS(0)
VARB INTEGER \#\# NO ANY(\#\#\#) . FLG \(\uparrow\) REG . NA \(\uparrow\) \(\uparrow\)
VARB SPAN(\(\uparrow 01\uparrow\)) . FIL \(\uparrow\) / / REM . NA
VARB INTEGER \#\# NO ANY(\#\#\#) . FLG \(\uparrow\) BREAK(\(\uparrow\) \(\uparrow\)) . T.V LEN(1)
VARB INTEGER \#\# NO ANY(\#\#\#) . FLG \(\uparrow\) BREAK(\(\uparrow\) \(\uparrow\)) . T.V LEN(1)
VARB INTEGER \#\# NO ANY(\#\#\#) . FLG \(\uparrow\) BREAK(\(\uparrow\) \(\uparrow\)) . T.V LEN(1)

S1  MAX = $NA
S2  BLDVEC = BLDVEC NA T.i \(\uparrow\uparrow\)
T.i = LT(T.i, MAX) T.i + 1
S3  BLDVEC = NA NO \(\uparrow\uparrow\)
S4  BLDVEC = NA RPOS(0)
S5  IDENT(\(\uparrow\) PREFIX\(\uparrow\) \(\uparrow\) FILG)
S5B  NA (ARB \(\uparrow\uparrow\)) . T.W (SPAN(ALPHANUM) \(\uparrow\uparrow\)) . T.B RPOS(0) = T.W
+  IDENT(FLG\#1\#)

S6  BLDVEC = T.B BLDVEC
T.i = LT(T.i, NO) T.i + 1
S5B  NA (BREAK(\(\uparrow\uparrow\)) LEN(1)) . T.B =
BLDVEC = BLDVEC T.B
T.i = LT(T.i, NO) T.i + 1
S6  VARB LEN(1) . FLG =
IDENT(FLG\#1\#)
BLDVEC = BLDVEC †ZERO**  
(S6) 186
BLDVEC = BLDVEC †ONE**  
(S6) 187
BLDVEC = BLDVEC(NA)  
(S10) 188
VARB = T.V †,†  
(S11) 189
VARB BREAK(†,†) • T.V LEN(1) =  
(S98) 190
BLDVEC = BLDVEC BLDVEC(T.V)  
(S98) 191
MAX = NO  
T.I = 1  
IDENT(FLG,†,†)  
(S14) 200
BLDVEC (BREAK(†,†) †,†) • T.WA REM • T.WB = T.WB T.WA :F(ERR5)  
T.I = LT(T.I,MAX) T.I + 1  
(S11) 193
(S12) 194
BLDVEC (ARB †,†) • T.WA (SPAN(ALPHANUM) †,†) • T.WB RPOS(0) =  
T.WB T.WA  
(S12) 195
T.I = LT(T.I,MAX) T.I + 1  
(F(ERR5) 196
MAX = $NA  
T.I = 1  
IDENT(FLG,†,†)  
(S14) 200
BLDVEC = BLDVEC NA T.I †,†  
(S14) 201
T.I = LT(T.I,MAX) T.I + 1  
(S15) 202
IDENT(FIL,†,†)  
(F(ERR5) 203
FIL = †ONE**  
(S16) 204
FIL = †ZERO**  
(S17) 205
MAX = NO  
T.I = 1  
IDENT(FLG,†,†)  
(S19) 207
BLDVEC (BREAK(†,†) †,†) REM • T.WB = T.WB FIL :F(ERR5)  
T.I = LT(T.I,MAX) T.I + 1  
(S18) 208
(S19) 209
BLDVEC (ARB †,†) • T.WA SPAN(ALPHANUM) †,† RPOS(0) = FIL T.WA  
T.I = LT(T.I,MAX) T.I + 1  
(S19) 210
(S19) 211
IDENT($NA,$NO)  
(F(ERR6) 212
BLDVEC = BLDVEC † T.† GEN(FLG,NA T.I † NO T.I) †,†  
T.I = LT(T.I,$NA) T.I + 1  
(S21) 213
(S21) 214
TRANSFER TV.L =  
TV.R =  
DTRAN = A.2 †,† A.1
IDENT($SDTRAN) $(*G: + $SDTRAN) = $(*G: + $SDTRAN) A . 3

T1
A . 1 = A . 1 ↑ ↑
A . 2 = A . 2 ↑ ↑
$SDTRAN = GEN(↑ ↑ ↑ A . 3)
TRIG . P = ↑ T↑ $SDTRAN

T2
A . 1 BREAK(↑ ↑ ↑) · T . V ↑ ↑ ↑ = : F(T3)
TV . L = TV . L BLDVEC(T . V)

T3
A . 2 BAL · T . V ↑ ↑ ↑ = : F(T4)
TV . R = TV . R BLDVEC(T . V)

T4
TV . L BREAK(↑ ↑ ↑) · TO . BIT ↑ ↑ ↑ = : F(RETURN)
TV . R BREAK(↑ ↑ ↑) · FRM . BIT ↑ ↑ ↑ = : F(ERR6)
TO . BIT * REG · NA INTEGER · NO RPOS(0) : F(ERR5)

T5
$(NA ↑ ↑ ↑ NO ↑ SET↑) = $(NA ↑ ↑ ↑ NO ↑ SET↑) ↑ T↑ GEN(↑ ↑ ↑ FRM . BIT TRIG . P)
$(NA ↑ ↑ ↑ NO ↑ CLR↑) = $(NA ↑ ↑ ↑ NO ↑ CLR↑) ↑ T↑

B00EXP
B . E ARB · P . TRM ↑ ↑ ↑ (NOTANY(↑ ↑ ↑) REM) · B . EXP : S(B10)
B . E ARB · N . TRM ↑ ↑ ↑ (NOTANY(↑ ↑ ↑) REM) · P . TRM : S(B11)
B . E ↑ ↑ ↑ REM · B . T : S(B12)
B . E ↑ 0↑
B00EXP = ↑ ZERO↑
B . E ↑ 1↑
B00EXP = ↑ ONE↑

B1
B . E * REG · NA ↑ 0↑ INTEGER · NO ↑ 0↑
B00EXP = ↑ ↑ NA NO

B2
B . E ↑ (↑ ARB · B . EXP ↑ )
B00EXP = BO0EXP(B . EXP)

B3
SGNL =
B . E ANY(↑ ↑ 0↑) · OP ↑ 0↑ * REG · NA RPOS(0) : F(B6)
T . I = 1

B4
SGNL = SGNL ↑ ↑ NA T . I
T . I = LT(T . I, SNA) T . I + 1 : S(B5) F(B9)

B5
B . E ANY(↑ ↑ 0↑) · OP ↑ 0↑ (↑ PREFIX↑ v ↑ SUFFIX↑) · FLAG
\[
\begin{align*}
\text{\texttt{\textasciitilde\rangle (t INTEGER \ No \ \texttt{\langle reg \ RA RPOS(0) :F\texttt{(ERR1)}})}} & \quad \text{249} \\
\text{\texttt{\textasciitilde\rangle IDENT\texttt{(\textasciitilde\rangle FLAG,\texttt{\langle PREFIX\rangle)}}} & \quad \text{250} \\
\text{\texttt{T.I = \textasciitilde\rangle NA \ No \ \texttt{\langle + 1}}) & \quad \text{251} \\
\text{\texttt{\texttt{\textasciitilde\rangle B7}}) & \quad \text{252} \\
\text{\texttt{\texttt{\textasciitilde\rangle B8}}) & \quad \text{253} \\
\text{\texttt{\texttt{\textasciitilde\rangle B9}}) & \quad \text{254} \\
\text{\texttt{\texttt{\textasciitilde\rangle B10}}) & \quad \text{255} \\
\text{\texttt{\texttt{\textasciitilde\rangle B11}}) & \quad \text{256} \\
\text{\texttt{\texttt{\textasciitilde\rangle B12}}) & \quad \text{257} \\
\text{\texttt{\texttt{\textasciitilde\rangle LIST}}) & \quad \text{258} \\
\text{\texttt{\texttt{\textasciitilde\rangle P1}}) & \quad \text{259} \\
\text{\texttt{\texttt{\textasciitilde\rangle P2}}) & \quad \text{260} \\
\text{\texttt{\texttt{\textasciitilde\rangle P3}}) & \quad \text{261} \\
\text{\texttt{\texttt{\textasciitilde\rangle P4}}) & \quad \text{262} \\
\text{\texttt{\texttt{\textasciitilde\rangle P5}}) & \quad \text{263} \\
\end{align*}
\]

\text{\texttt{\textasciitilde\rangle B7}}
\text{\texttt{T.I = 1}}
\text{\texttt{\texttt{\textasciitilde\rangle B8}}}
\text{\texttt{SGNL = SGNL \ \texttt{\langle NA \ T.I}})
\text{\texttt{\texttt{\textasciitilde\rangle B9}}}
\text{\texttt{BOOEXP = \texttt{\langle T+ \ GEN\texttt{(OP,SGNL)}}}
\text{\texttt{\texttt{\textasciitilde\rangle B10}}}
\text{\texttt{BOOEXP = \texttt{\langle T+ \ GEN\texttt{(\langle\rangle,BOOEXP\langle P.TRM\rangle} BOOEXP\langle B.EXP\rangle)}
\text{\texttt{\texttt{\textasciitilde\rangle B11}}}
\text{\texttt{BOOEXP = \texttt{\langle T+ \ GEN\texttt{(\langle\rangle,BOOEXP\langle N.TRM\rangle} BOOEXP\langle P.TRM\rangle)}
\text{\texttt{\texttt{\textasciitilde\rangle B12}}}
\text{\texttt{BOOEXP = \texttt{\langle T+ \ BOOEXP\langle B.T\rangle} \ (\texttt{RETURN))}}
\text{\texttt{\texttt{\textasciitilde\rangle LIST}}}
\text{\texttt{\texttt{\textasciitilde\rangle P1}}}
\text{\texttt{\texttt{\textasciitilde\rangle P2}}}
\text{\texttt{\texttt{\textasciitilde\rangle P3}}}
\text{\texttt{\texttt{\textasciitilde\rangle P4}}}
\text{\texttt{\texttt{\textasciitilde\rangle P5}}}
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle CONTROL = 1}}}
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle CONTROL = 0}}}
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ REGISTER \ F/F \ SET \ CLEAR} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})

\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle CONTROL = 1}}}
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle CONTROL = 0}}}
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ REGISTER \ F/F \ SET \ CLEAR} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})

\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle CONTROL = 1}}}
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle CONTROL = 0}}}
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ REGISTER \ F/F \ SET \ CLEAR} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})
\text{\texttt{\texttt{\textasciitilde\rangle \texttt{\langle OUTPUT = \texttt{\langle T+ \ TEMPLATE PROGRAM COMPILED INTO THE}}} \}})
OUTPUT = '---' '---' '---' '---' '---' '---' '---' '---' '---' '---' '---' '---' '---' '---' '---'
  T.I = 1

P6
STR1 = ' ' T.I ' ' $('GT' ' T.I) ' ' $('GI' ' T.I)
LT(SIZE(STR1) + 40) ' ' IF(P8)
T.I = LT(T.I, G.I) T.I + 1 ' ' IF(P9)
STR1 = STR1 B.L
STR1 LEN(41) * STR2
OUTPUT = STR2 T.I ' ' $('GT' ' T.I) ' ' $('GI' ' T.I)

P7
T.I = LT(T.I, G.I) T.I + 1 ' ' S(P6) F(P10)
P8
OUTPUT = STR1 ' ' (P7)
P9
OUTPUT = STR1
P10
CONTROL = 0
OUTPUT = ' ' INPUT ' ' DELAY ELEMENT NO. ' ' T.I
LT(T.I, D.I) T.I = 1 ' ' IF(P12)
STR1 = STR1 B.L
STR1 LEN(30) * STR2
STR1 = STR2 $('DLY' ' T.I) ' ' T.I
T.I = LT(T.I, D.I) T.I + 1 ' ' IF(P12)
STR1 = STR1 B.L
STR1 LEN(60) * STR2
OUTPUT = STR2 $('DLY' ' T.I) ' ' T.I
T.I = LT(T.I, D.I) T.I + 1 ' ' S(P11) F(P13)

P11
STR1 = $('DLY' ' T.I) ' ' T.I

P12
OUTPUT = STR1
P13
CONTROL = 0
OUTPUT = 'AHPL.VER.3' ' ' DATE() ' ' EXECUTE TIME IN MSEC = '
TIME()
APPENDIX B

LISTING OF AHPL COMBINATIONAL LOGIC

SUBROUTINE COMPILER
SNOBOL (VERSION 2.0)

---------

BELL TELEPHONE LABORATORIES, INCORPORATED
CDC VERSION BY PURDUE UNIVERSITY AND I.O.A.

AHPL.VER.3

# APL COMBINATIONAL LOGIC SUBROUTINE COMPILER

MAXLENGTH = 7700
OUTPUT(\+CONTROL\+,2,\+(132A1)\+)
DEFINE(\+GEN(A,1,A,2)\+)
DEFINE(\+BOOEXP(B,E)\+)
DEFINE(\+READ()\+,\+NXTST\+)
DEFINE(\+FUN(XRAY)OP1,OP2,OPA,LIST\#,\+FUNB\#)
DEFINE(\+TTYPE()\+)
DEFINE(\+STYPE()\+)
DEFINE(\+BTYPE()\+)
DEFINE(\+COMP(ELEMENT)\+)
DEFINE(\+PASIG(STRING)\+)
DEFINE(\+SRSIG(B,T,SG)\+)
DEFINE(\+FNCALL()\+)
DEFINE(\+EXPAND(OLD.SIG,NEW.SIG,FUNC,)\#)
DEFINE(\+CONNECT(PLACE,FROMSIG)\+)
DEFINE(\+REDUCE(B,NOP,RED.OP,LNG,NA,1,NEG,1,NEG,2)\+)
LETTERS = \+ABCDEFGHIJKLMNOPQRSTUVWXYZ\+
DIGITS = \+0123456789\+
ALPHANUM = LETTERS DIGITS
BINOP = ANY(\+\+\+)
BLANKS = SPAN(\+\+)
INTEGER = SPAN(DIGITS)
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENTIFIER = SPAN(LETTERS)</td>
<td></td>
</tr>
<tr>
<td>GATESIG = (↑ T↑ v ↑ F↑) . TRUTHCODE INTEGER . POSITN</td>
<td></td>
</tr>
<tr>
<td>UNARYOP = ↑¬↑ v NULL</td>
<td></td>
</tr>
<tr>
<td>FACTOR = IDENTIFIER v INTEGER</td>
<td></td>
</tr>
<tr>
<td>UNARYOP = IDENTIFIER IDENTIFIED↑ [↑ FACTOR ↑]↑ v UNARYOP IDENTIFIER↑</td>
<td></td>
</tr>
<tr>
<td>SIGNALMATCH = UNARYOP . NEG↓ IDENTIFIER . NA↑ <em>[↑ (IDENTIFIER v INTEGER) . NUMBER ↑]</em></td>
<td></td>
</tr>
<tr>
<td>SREDUCE = ↑↑ UNARYOP . NEG↓ BINOP . B↑ BINOP ↑↑↑</td>
<td></td>
</tr>
<tr>
<td>SREDUCE = ↑↑ UNARYOP . NEG↓ BINOP . B↑ BINOP ↑↑↑</td>
<td></td>
</tr>
<tr>
<td>FINDFN = POS(O) *FUNVEC RPOS(O)</td>
<td></td>
</tr>
<tr>
<td>FUNVEC = FAIL</td>
<td></td>
</tr>
<tr>
<td>LEVCT = 0</td>
<td></td>
</tr>
<tr>
<td>OS.CT = 0</td>
<td></td>
</tr>
<tr>
<td>FTAB = ARRAY(↑4↑ 8↑)</td>
<td></td>
</tr>
<tr>
<td>G↑ I = 0</td>
<td></td>
</tr>
<tr>
<td>SR.CT = 0</td>
<td></td>
</tr>
<tr>
<td>SUBRNST = ARRAY(↑15↑ 8↑)</td>
<td></td>
</tr>
<tr>
<td>REG = FAIL</td>
<td></td>
</tr>
<tr>
<td>BUS = FAIL</td>
<td></td>
</tr>
<tr>
<td>SRVEC = FAIL</td>
<td></td>
</tr>
<tr>
<td>EOF↑ 2 =</td>
<td></td>
</tr>
<tr>
<td>FFLAG =</td>
<td></td>
</tr>
<tr>
<td>EOF =</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
# ANCMUX = 1
READI
LINE = TRIM(INPUT)
LINE =>
OUTPUT = + + LINE : (READI)
ORIGIN
READ(
IMAGE +D+ SPAN(+ +) =
IMAGE SRDEF : S(SUBRI)
IMAGE FNDEF : S(FNCT)
ERR1
OUTPUT = <<<SYNTAX ERROR>>> AT + ALASTNO : (END)
ERR5
OUTPUT = +EXPAND* HARDWARE NAMED OLD-SIGNAL NOT FOUND.* + (END)
ERR6
OUTPUT = UNRECOGNIZED ARG IN SRSIG = * SRSIG : (END)
ERR7
OUTPUT = SYM*CT EXCEEDED MAXIMUM.* : (END)
ERR8
OUTPUT = UNRECOGNIZABLE SIGNAL.* : (END)
ERR9
OUTPUT = EXCESSIVE NUMBER OF SUBROUTINES DECLARED.* : (END)
ERR10
OUTPUT = EXCESSIVE NUMBER OF ARGUMENTS USED.* : (END)
ERR11
OUTPUT = LENGTH OF INPUT BUSS NOT DECLARED.* : (END)
ERR12
OUTPUT = UNABLE TO LOCATE STATEMENT LABEL.* : (END)
ERR13
OUTPUT = NO INPUT DATA.* : (END)
ERR14
OUTPUT = SNOBOL4 COMPILATION NOT RECOGNIZED.* : (END)
ERR15
OUTPUT = COMPARE RELATION NOT RECOGNIZED.* : (END)
ERR16
OUTPUT = DGOTO NOT RECOGNIZED.* : (END)
ERR17
OUTPUT = XRAY = XRAY : (END)
ERR18
OUTPUT = COMPILED GENERATOR SUPPLY EXHAUSTED.* : (END)
ERR19
OUTPUT = DESTINATION= TO PLACE SOURCE= FROMSIG : (END)
DCLR
IMAGE IDENTIFIER = VECTOR => INTEGER = LENGTH IF(BUSD)
$VECTOR = LENGTH
RNAME = RNAME VECTOR =>
REG = REG => VECTOR : (ORIGIN)
BUSD
IMAGE BREAK(=>) = B.N <=[= BREAK(=>)] = INSIDE
* => REM * R.VEC : (ERR1)
BUS = BUS => B.N
MAX = $R.VEC + G.1
```
\$B.N = G.I + 1
G*B
  GEN(\\u2192\\u2192,)
  LT(G*I,MAX)
  T.I = 1
  DFLT = GEN(\\u2192\\u2192,)
C.V1
  $(\langle G: \langle \$B.N + T.I - 1 \rangle \rangle = \langle T.G: \langle \$B.N + T.I - 1 \rangle \rangle
  + (R.VEC T.I)
  T.I = LT(T.I + \$R.VEC) T.I + 1
  INSIDE = INSIDE \u2192
N.V
  INSIDE BREAK(\u2192, \u2192) • MAY.BUS \u2192 = \u2192 F(C.0)
  MAY.BUS BREAK(\u2192, \u2192) • BEXP \u2192 REM • R.VEC
  T.E = B00EXP(BEXP)
  \$\langle G: \langle DFLT \rangle \rangle = \$\langle G: \langle DFLT \rangle \rangle \rightarrow T.E
  T.I = 1
C.V2
  $(\langle G: \langle \$B.N + T.I - 1 \rangle \rangle = \$\langle G: \langle \$B.N + T.I - 1 \rangle \rangle
  + (R.VEC T.I)
  T.I = LT(T.I + \$R.VEC) T.I + 1
C.O
  $(\$B.N \u2192) = \$B.N \u2192 MAX
SUBR1
  EOF.2 = \$G = G.I
  GATEVEC = FAIL
  DUMVEC = FAIL
  INDEX.VEC = FAIL
  OUTVEC =
  SR.CT = LT(SR.CT,15) SR.CT + 1
  $SRNAME = SR.CT
  ALPHAPAT = SRNAME
  CPULSE =
  SNAME = SNAME SRNAME \u2192
  IWORD = SRNAME \u2192
  ARG.CT = 1
  LIST = LIST \u2192
SUBR2
  LIST BREAK(\u2192, \u2192) • ARGUE LEN(1) = \u2192 F(SUBR6)
\begin{verbatim}
ARG•CT = ARG•CT + 1
ARGUE *REG RPOS(0)
ARGUE *BUS RPOS(0)
DUMVEC = DUMVEC • ARGUE
TEMP = ARGUE \rightarrow \rightarrow
$TEMP$ INTEGER • FIRST \rightarrow INTEGER • LAST
$\{\rightarrow DA.\rightarrow ARG•CT\} = (LAST \rightarrow FIRST)
SUBRNST[SR.CT, ARG•CT] = FIRST
$ARGUE = ARG•CT$
SUBR4 SUBRNST[SR.CT, ARG•CT] = ARGUE
SUBR6 READ()
IMAGE (INTEGER • NULL) • LB•L BLANKS ANY(*SRT*) • KEY BLANKS =
+ 
LB•L = ALPHAPAT LB•L
IDENT(KEY, \rightarrow S) :S(SUBR7)
IDENT(KEY, \rightarrow B) :S(SUBR8)
IDENT(KEY, \rightarrow T) :F(ERR1) S(SUBR9)
SUBR7 STYPE()
SUBR8 BTYPE()
SUBR9 TTYPE()
IDENT(EOF.2) :S(SUBR6)
OUTPUT =
NEWCODE = CODE(IWORD)
SCALL5 K = 0
GROUP2 =
GROUP1 =
SUBR32 K = LT(K,64) K + 1 :F(SUBR33)
GROUP1 = GROUP1 \{1 \# K\}
GROUP2 = GROUP2 \{2 \# K\}
$\{1 \# K\} =
$\{2 \# K\} =
SUBR33 SUBRNST[SR.CT, 6] = GROUP1
SUBRNST[SR.CT, 7] = GROUP2
\end{verbatim}
CONTROL = 0
OUTPUT = GATES G.1 IN SRNAME (:ORIGIN)
FNCT
FUNVEC = FUNVEC FNAME
$FNAME = FN.CT
FN.CT2 = FUNCTION FN.CT
ALPHAPAT = FNAME
IWORD = IWORD FNAME ;
ARGVEC = FAIL
ARG.CT = 0
FNC.1 IDENT(LIST) ;S(FNC.4)
LIST BREAK(↑↑) ARGUE LEN(1) = ;F(FNC.3)
FNC.2 ARG.CT = LT(ARG.CT 8) ARG.CT + 1 ;F(ERR10)
ARGVEC = ARGVEC ARGUE
$ARGUE = ARG.CT (FNC.1)
FNC.3 ARGUE = LIST
LIST = NULL ;(FNC.2)
FNC.4 READ()
IMAGE BLANKS (INTEGER NULL) LB.L BLANKS ANY(↑SRT↑) KEY
+ BLANKS = ;F(ERR5)
LB.L = ALPHAPAT LB.L
FFLAG = 1
IDENT(KEY,↑S↑) ;S(FNC.5)
IDENT(KEY,↑T↑) ;S(FNC.7)
IDENT(KEY,↑B↑) :F(ERR1)
BTYPE() ;S(FNC.4) ;F(ERR5)
FNC.5 STYPE() ;S(FNC.4) ;F(ERR5)
FNC.7 IMAGE ↑ = ;S(FNC.7)
IMAGE #END# ;S(FNC.9)
TTYPE() ;(FNC.4)
FNC.9 FFLAG =
OUTPUT = COMPILING COMPLETE AT TIME = TIME()
$FN.CT2 = IWORD
IWORD = ;(:ORIGIN)
BT.22
PLACE = PASIG(L.SIDE)
L.SIDE = TRIM(L.SIDE)
INDEX.VEC = INDEX.VEC v L.SIDE

BT.22
IMAGE BREAK(=+=) * L.SIDE LEN(1) = :F(RT.5)
L.SIDE = TRIM(L.SIDE)
INDEX.VEC RPOS(0) *INDEX.VECK RPOS(0) = :S(RT.22)

BT.11
PLACE = PASIG(L.SIDE)
IMAGE SPAN(=+=) =
IDENT(IMAGE) = :S(BT.13)
IMAGE BREAK(=+=/)()= VAR.1 LEN(1) * OP.OR = :F(RT.12)
ARI.STR = ARI.STR + PASIG(TRIM(VAR.1)) + + OP.OR : (BT.11)

BT.12
ARI.STR = ARI.STR ≠ PASIG(IMAGE)

BT.13
IWORD = IWORD LB.L + + PLACE + = + ARI.STR + : (RETURN)

BT.5
IMAGE + + INTEGER * GO.T0 + +
GO.T0 = ALPHAPAT GO.T0

BT.6
IWORD = IWORD LB.L + +:(GO.T0 + +) + + + : (RETURN)

BT.8
IMAGE BREAK(=+=) * VAR.1 LEN(1) = :F(ERR1)
VAR.1 = PASIG(VAR.1)
IMAGE BREAK(=+=) * VAR.2 =
VAR.2 = PASIG(VAR.2)
IMAGE SPAN(=+=) =
IMAGE + (EQ,NEQ) + = + EQ+ :S(RT.6)
IMAGE + (LE,GT) + = + LE+ :S(RT.8)
IMAGE + (LT,GE) + = + LT+ :F(ERR16)

BT.9
IMAGE TAB(2) * LOGIC =
IMAGE SPAN(=+=) =
IMAGE + (INTEGER * S.1 + + INTEGER * S.2 + +) + :F(ERR17)
S.1 = ALPHAPAT S.1
S.2 = ALPHAPAT S.2

BT.10
WORD = LB.L ≠ # LOGIC #(# VAR.1 #, # VAR.2 #) :S(# S.1
#)F(# S.2 #) ;
IWORD = IWORD WORD + : (RETURN)

SType
TP.T = 0
IMAGE BREAK(=+=) * L.SIDE LEN(1) = :F(ERR1)
IMAGE BLANKS =
WORD = LB.L # DEST.* = # PASIG(L.SIDE) ≠ ;
IWORD = IWORD WORD
IMAGE POS(0) UNARYOP . NEG.1 ↑(↑ IDENTIFIER . FNAME3 ↑(↑
BREAK(↑↑) . LIST ↑)↑ RPOS(0) :F(ST.B)
FNAME3 FINDFN :F(ERR18)
FNCALL()
NEG.1 ↑= = #COMP(SRSIG(($↑FR↑ ≠ $FNAME3 ≠ ))≠ :S(ST.A)
NEG.1 = #SRSIG($↑FR↑ ≠ $FNAME3 ≠ )≠
ST.A WORD = LB.L ↑.R↑ ≠ CONNECT(# PASIG(L.SIDE) ≠ , # NEG.1
#LEVCT = LEVCT - 1 ;:
↑ FCODE = ↑↑
ST.B IMAGE POS(0) ≠ (* L.SIDE BINOP . BOP M.BIT . NEWSIG ≠ )≠
RPOS(0) :F(ST.C)
WORD = # EXPAND(DEST.* SRSIG(# PASIG(NEWSIG ≠ )≠ .*#
BOP #↑) ;:
ST.C IMAGE ↑(↑ * L.SIDE BINOP . BOP UNARYOP . NEG.1 ↑(↑ IDENTIFIER .
FNAME3 ↑(↑ BREAK(↑↑) . LIST ↑))↑ RPOS(0) :F(ST.E)
FNAME3 FINDFN :F(ERR18)
FNCALL()
NEG.1 ↑= = #COMP(SRSIG(($↑FR↑ ≠ $FNAME3 ≠ ))≠ :S(ST.D)
NEG.1 = #SRSIG($↑FR↑ ≠ $FNAME3 ≠ )≠
ST.D WORD = LB.L ↑.R↑ ≠ EXPAND(# PASIG(L.SIDE) ≠ , # NEG.1 ≠ , # BOP
↑ #↑ ; LEVCT = LEVCT - 1 ;:
↑ FCODE = ↑↑
ST.E T.CT = 0
WORD = # CONNECT(DEST.* # FUN(IMAGE) ≠ ) ;:
ST.X IWORD = IWORD WORD
TTYPE IMAGE ↑END↑ :F(TT.3)
IWORD = IWORD LB.L ↑ :SCALL5) ;:
TT.1 EOF.2 = 1 :F(RETURN)
TT.3 IMAGE BREAK(↑=↑) LEN(1) = :F(ERR18)
IMAGE BLANKS =
IDENT(FLAG) ; (TI.4)
IMAGE POS(0) IDENTIFIER . R1 => IDENTIFIER . R2 RPOS(0)
* 1S(TT.5)
OUTVEC = OUTVEC v IMAGE
$( $0.# IMAGE) = 1
TT.38
WORD = LB.L $ = #
TT.2
IWORD = IWORD WORD : (RETURN)
TT.4
WORD = LB.L $( FR # FN . CT $) = # PASIG (IMAGE)
* $ $ $ (TT.2)
TT.5
OUTVEC = OUTVEC v R1 v R2
$( #0.# R1) = 1
$( #0.# R2) = 2 : (TT.38)
FUNB
OP1 =
LIST =
FUN0.5
XRAY # # = : S(FUN0.5)
XRAY SRREDUCE = : S(FUNA.2)
XRAY SRPREDUCE = : S(FUNA.3)
XRAY M.BIT . OPA = : S(FUN2.0)
XRAY ANY (# $) . OP2 = : S(FUN2.2)
XRAY $ $ . BAL . OPA # = : S(FUN7.0)
XRAY $ ( BAL . OPA $) $ = : S(FUN8.0) F(ERR1)
FUN2.0
LIST = LIST # SRSIG($ PASIG(OPA ) $)#
FUN2.1
IDENT(XRAY) : F(FUN0.5)
IDENT(OP1) : F(FUN5.1) S(FUN6.0)
FUN2.2
IDENT(OP1) : F(FUN5.0)
FUN2.3
OP1 = OP2
OP2 = : (FUN0.5)
FUN5.0
IDENT(OP1*OP2) : F(FUN5.1)
OP2 = : (FUN0.5)
FUN5.1
T.CT = T . CT + 1
BLIM = # T . T . CT
FUN5.9
LIST $ # = : S(FUN5.9)
WORD = # # BLIM $ = $ T . GEN($ $ OP1 $ $ LIST $) ; ;

115
IW0RD = IWORD WORD
LIST = ≠ SRSIG(≠≠ BLIM ≠≠)≠
IDENT(OP2) : F(FUN8,1)
FUN6.0 LIST ≠ ≠ = S(FUN6,9)
FUN = LIST : (RETURN)
FUN7.0 LIST = LIST ≠ COMP(≠ FUN(OPA) ≠≠) ≠ (FUN2,1)
FUN8.0 LIST = LIST ≠ ≠ FUN(OPA) : (FUN2,1)
FUN8.1 OP1 = OP2 : (FUN2,1)
FUN8.2 LIST = LIST ≠ REDUCE(≠≠ B N0P ≠≠, ≠≠ NA.1 ≠≠, ≠≠ NEG.1
+ ≠≠, ≠≠ NEG.2 ≠≠) ≠ (FUN2,1)
FUN8.3 LIST = LIST ≠ REDUCE(≠≠ B N0P ≠≠, ≠≠ RED.0P ≠≠, ≠≠ NUMBER ≠≠, ≠≠
+ NA.1 ≠≠, ≠≠ NEG.1 ≠≠, ≠≠ NEG.2 ≠≠) ≠ (FUN2,1)
FNCALL IWORD = IWORD ¬ NLEVCT = LEVCT ; LEVCT = LEVCT + 1 ¬
ARG.0T = 0
FNCL.1 IDENT(LIST) IS(FNCL.6)
LIST BREAK(≠≠, ≠) ¬ ARGUE LEN(1) = : F(FNCL.5)
FNCL.2 ARG.0T = LT(ARG.0T ¬ 8) ARG.0T + 1 : F(ERP11)
IDENT(FFLAG) : F(FNCL.4)
FNCL.3 ARGUE POS(0) ¬ INDEX.VEC RPOS(0) = S(FNCL.7)
WORD ≠ Site FTAB[LEVCT, ≠ ARG.0T ≠] ¬ ≠ ARGUE ≠ ≠ ¬
FNCL.9 IWORD = IWORD WORD : (FNCL.1)
FNCL.7 WORD ¬ Site FTAB[LEVCT, ≠ ARG.0T ≠] ¬ FACTVAL(≠≠ ARGUE ≠≠) ≠
+ : (FNCL.9)
FNCL.4 ARGUE POS(0) ¬ ARGUE POS(0) : F(FNCL.3)
WORD ≠ Site FTAB[LEVCT, ≠ ARG.0T ≠] ¬ FTAB[NLEVCT, ¬ ARGUE ≠] ¬
+ : (FNCL.9)
FNCL.5 ARGUE = LIST
LIST = : (FNCL.2)
FNCL.6 INSERT = ≠ FUNCTION≠ $FUNCTION3
CAL.CNR = CAL.CNR + 1
INSERT2 = INSERT ≠ C.¬ CAL.CNR
$INSERT2 = $INSERT ≠ ¬ L .¬ R ¬ ¬
WORD = ≠ FCODE = CODE(≠≠ INSERT2 ≠≠) ¬ [FCODE] ¬
IW0RD = IWORD WORD
CONNECT

TO PLACE SIGNALMATCH : F(CNC7)
NUMBER = -INTEGER(NUMBER) $NUMBER
NA.1 $REG RPOS(0) : F(CNC2)

CNC

NEG.1 $CLR = $SET
NEG.1 = $SET#

CNC1

$(NA.1 $UP NUMBER NEG.1) = $(NA.1 $UP NUMBER NEG.1) FROMSIG

CNC2

P_SIG = NA.1 NUMBER
GATEVEC = GATEVEC $ NOT NA.1 NUMBER
SP_SIG = FROMSIG
NA.1 POS(0) $OUTVEC RPOS(0) : F(RETURN)
$( $( #0 $ NOT NA.1) $ NOT # NUMBER) = FROMSIG : F(RETURN)

CNC7

TO PLACE UNARYOP $ NEG.1 IDENTIFIER $ NA.1 : F(ERR21)
IDENT(NA.1 $RPULSE#) : F(CNC12)
OS.CT = LT(OS.CT + 20) OS.CT + 1 : F(ERR20)
SUBSNST[SR.CT,8] $ NOT OS.#$ OS.CT
$(#OS.#$ OS.CT) = FROMSIG : F(RETURN)

CNC12

NA.1 $REG RPOS(0) : F(CNC0)
NUMBER = 1 : F(CNC0)

CNC8

NUMBER = 1

CNC9

IDENT(NA.1 $RPULSE#) : F(CNC2)
CPULSE = FROMSIG : (RETURN)

BOOEXP

B.E AUTO P.TRIM $NOTANY($UP$) REM) $ B.EXP $S(B10)
B.E AUTO N.TRIM $NOTANY($UP$) REM) $ P.TRIM $S(R11)
B.E $UP REM $ B.T : S(R12)
B.E $UP $F(R1)
BOOEXP = $ZERO$ : (RETURN)
B.E $UP $F(R2)
BOOEXP = $ONE$ : (RETURN)
B.E $REG $ NOT NA $<>$ INTEGER $ NOT NO $<>$ : F(R3)
BOOEXP = $UP$ NA NO : (RETURN)
B3

B.E $UP$ ARB $ B.EXP $UP$ : F(R4)
BOOEXP = BOOEXP(B.EXP)

SGNL =
B.E ANY(\uparrow\uparrow\uparrow\uparrow) • OP \leftrightarrow \#REG • NA RPOS(0)  :(RETURN)
T.I = 1

B5
SGNL = SGNL \uparrow \uparrow NA T.I
T.I = LT(T.I;:N: NA) T.I \leftrightarrow 1  :S(R5)F(R6)

B6
B.E ANY(\uparrow\uparrow\uparrow\uparrow) • OP \leftrightarrow ANY(\uparrow\uparrow\uparrow\uparrow) • FLAG \leftrightarrow \#INTEGER • NO \leftrightarrow \uparrow
*REG • NA RPOS(0)
IDENT(FLAG,\uparrow\uparrow\uparrow\uparrow)  :F(ERR)
T.I = $NA \rightarrow NO + 1

B7
T.I = 1

B8
SGNL = SGNL \uparrow \uparrow NA T.I
T.I = LT(T.I;:N: NO) T.I \leftrightarrow 1

B9
BOOEXP = \uparrow \uparrow GEN(OP;SGNL)

B10
BOOEXP = \uparrow \uparrow GEN(\uparrow\uparrow\uparrow\uparrow,BOOEXP(P.TRM) BOOEXP(B.EXP))  :(RETURN)

B11
BOOEXP = \uparrow \uparrow GEN(\uparrow\uparrow\uparrow\uparrow,BOOEXP(N.TRM) BOOEXP(P.TRM))  :(RETURN)

B12
BOOEXP = \leftrightarrow BOOEXP(B.T)

COMP
ELEMENT POS(0) GATESIG RPOS(0)

TRUTHCODE \leftrightarrow F\leftrightarrow = \leftrightarrow T\leftrightarrow
TRUTHCODE = \leftrightarrow F\leftrightarrow

COMP1.0
COMP = TRUTHCODE POSITN

COMP2.0
ELEMENT \leftrightarrow \leftrightarrow

COMP = ELEMENT
COMP \leftrightarrow ZERO\leftrightarrow = \leftrightarrow ONE\leftrightarrow
COMP \leftrightarrow ONE\leftrightarrow = \leftrightarrow ZERO\leftrightarrow
COMP \leftrightarrow \leftrightarrow = \leftrightarrow
COMP = \leftrightarrow \leftrightarrow COMP

EXPAND
OLD.SIG SIGNALMATCH  :F(EXP4.0)
NUMBER = INTEGER(NUMBER) SNUMBER

NA.1 \leftrightarrow REG RPOS(0)  :F(EXP2.0)

EXP1.0
NEG.1 \leftrightarrow \leftrightarrow = \leftrightarrow CLR\leftrightarrow
NEG.1 = \#SET\leftrightarrow

EXP.A
$(NA.1 \leftrightarrow \leftrightarrow NUMBER NEG.1) = $(NA.1 \leftrightarrow \leftrightarrow NUMBER NEG.1) NEWSIG
EXP2.0  
EXPAND = NA,1 NUMBER : (RETURN) 357
EXPAND POS(0) *GATEVEC RPOS(0) : S(EXP2.1) 359
GATEVEC = GATEVEC V EXPAND 360
$( EXPAND ) = T# GEN(FUNC.,NEWSIG) : (EXP7.0) 361
EXP2.1  
$(EXPAND) POS(0) GATESIG RPOS(0) : F(ERR5) 362
$( #G. # POSITN ) = $( #G. # POSITN ) NEWSIG 363
EXP7.0  
NA,1 POS(0) *OUTVEC RPOS(0) : F(RETURN) 364
$( #0. # NA,1 ) #. # NUMBER ) = #EXPAND : (RETURN) 365
EXP4.0  
OLD.SIG UNARYOP * NEG.1 IDENTIFIER * NA,1 : F(ERR5) 366
NA,1 *REG RPOs(0) : F(EXP4.2) 367
NUMBER = 1 : (EXP1.0) 368
EXP4.2  
NUMBER = 1 : (EXP2.0) 369
SRSIG  
SRSIG = RT.SIG 370
IDENT(SRSIG) : S(SRSIG9.6) 371
SRSIG = #SRSIG ; (RETURN) 372
SRSIG1.0  
SRSIG +(0)+ v +0+ = +ZERO+ : S(RETURN) 373
SRSIG +(1)+ v +1+ = +ONE+ ; S(RETURN) 374
SRSIG #CPULSE# : S(SRSIG1A) 375
SRSIG +RPULSE+ ; F(SRSIG2.0) 376
SRSIG = + OS.+ OS.CT : (RETURN) 377
SRSIG2.0  
SRSIG *GATESIG RPOS(0) ; S(RETURN) 378
SRSIG SIGNALMATCH : F(SRSIG6.0) 379
NUMBER = -INTEGER(NUMBER) #NUMBER 380
NA,1 *REG RPOS(0) ; F(SRSIG3.0) 381
SRSIG2.1  
SRSIG = + T+ NEG.1 NA,1 NUMBER ; (RETURN) 382
SRSIG3.0  
NA,1 FINDM - F(SRSIG5.0) 383
SRSIG3.1  
PL.GT = SUBRNST[S.R.CT, $NA,1] + NUMBER 384
IDENT(NEG.1) ; F(SRSIG4.0) 385
SRSIG = + T+ PL.GT ; (RETURN) 386
SRSIG4.0  
SRSIG = + F+ PL.GT ; (RETURN) 387
SRSIG5.0  
PL.GT = NA,1 NUMBER ; (RETURN) 388
SRSIG5.1 PL·GT *GATEVEC RPOS(0) :F(ERRA)
IDENT(NEG.1) :F(SRSIG5.2)
SRSIG = $PL·GT : (RETURN)
SRSIG5.2 SRSIG = COMP($PL·GT)
SRSIG6.0 SRSIG UNARYOP * NEG.1 IDENTIFIER NA.1 :F(SRSIG7.0)
NA.1 *REG RPOS(0) :F(SRSIG6.1)
NUMBER = 1 : (SRSIG6.1)
SRSIG6.1 NA.1 FINDM :F(SRSIG6.2)
NUMBER = 1 : (SRSIG6.1)
REDUCE Q·STR =
NEG.1 ≠ ≠ ≠ ≠ F≠
NEG.1 = + T↑
RED0.5 IDENT(REO.OP) :F(RDC.2)
SRSIG7.1 NA.1 *REG RPOS(0) :F(SRSIG7.4)
IN.1 = 0
MAX.1 = $NA.1
SRSIG7.2 IN.1 = LT(IN.1·MAX.1) IN.1 + 1 :F(SRSIG7.3)
Q·STR = Q·STR ↑ ↑ NEG.2 NA.1 IN.1 : (SRSIG7.2)
SRSIG7.3 REDUCE = NEG.1 GEN(B·NOP, Q·STR) : (RETURN)
SRSIG7.4 NA.1 FINDM
NEG.2 ↑ ↑ = ↑ F≠ :S(SRSIG7.5)
NEG.2 = + T↑
SRSIG7.5 IN.1 = SUBRNST[SRC·CT, $NA.1] - 1
MAX.1 = $(DA, ≠ $NA.1) * IN.1
SRSIG7.6 IN.1 = LT(IN.1·MAX.1) IN.1 + 1 :F(SRSIG7.3)
Q·STR = Q·STR NEG.2 IN.1 : (SRSIG7.6)
RDC.2 NUMBER = INTEGER(LNG) LNG
NUMBER = $LNG
SRSIG8.1 NA.1 *REG RPOS(0) :F(SRSIG8.2)
IDENT(REO.OP, ↑↑) :S(SRSIG8.2)
IN.1 = 0
MAX.1 = NUMBER : (SRSIG7.2)
SRSIG8,2 IN,1 = $NA,1 - NUMBER
MAX,1 = IN,1 + NUMBER
*: (SRSIG7,2)
SA
SRSIG8,3 NA,1 FINDM
NEG,2 = = = = = F
*: (SRSIG8,4)
NEG,2 = = T
SRSIG8,4 IDENT (RED, OP, # ≤ #)
*: (SRSIG8,5)
IN,1 = SUBRNST [SR,CT, $NA,1] - 1
MAX,1 = IN,1 + NUMBER
*: (SRSIG7,6)
SRSIG8,5 MAX,1 = $ DA, $ NA,1 ) + SUBRNST [SR,CT, $NA,1] - 1
IN,1 = MAX,1 - NUMBER
*: (SRSIG7,6)
SRS1A IDENT (CPULSE)
CPULSE = ≠ T* GEN (≠ v≠ )
*: (RETURN)
SRS1B SRSIG = CPULSE
*: (RETURN)
SRSIG9,6 SRSIG = =DEFAULT =
*: (RETURN)
PASIG IDENT (FFLAG)
*: (PA3,0)
IDENT (KEY = ≠ #)
*: (PA2,0)
PA1,0 PASIG = ≠ ≠ STRING ≠ ≠
*: (RETURN)
PA2,0 PASIG = STRING
*: (RETURN)
PA3,0 IDENT (KEY = ≠ #)
*: (PA2,0)
STRING SIGNAL MATCH
*: (PA7,0)
TP,T = TP,T + 1
TEMP = ≠ TEMP ≠ TP,T
INITIAL = ≠ ≠ TEMP ≠ = ≠ NEG,1 ≠ ≠ ≠
NA,1 POS(0) * ARGVEC RPOS(0)
*: (PA6,0)
NUMBER POS(0) * ARGVEC RPOS(0)
*: (PA5,0)
WORD = INITIAL ≠ FTAB [LEVCT, ≠ $NA,1 ≠ ] ≠ ≠ ≠ ≠
*: (PA4,0)
WORD = INITIAL ≠ FTAB [LEVCT, ≠ $NUMBER ≠ ] ≠ ≠ ≠ ≠
*: (PA4,0)
NUMBER ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ ≠ =

121
WORD = INITIAL ≠ ≠ NA.1 ≠ NEFTAB[LEVCT, ≠ $NUMER
#] ≠ NEFTAB[LEVCT, ≠ $NUMER
+ ≠ (PA4.0)
PA7.0
STRING UNARYOP NEFTAB[LEVCT, ≠ $NUMER
NA.1 POS(0) ≠ ARGVEC RPOS(0) :F(PA1.0)
PA7.1
WORD = ≠ TEMP ≠ NEFTAB[LEVCT, ≠ $NUMER
+ ≠ (PA4.0)
PA8.0
STRING POS(0) ≠ ARGVEC RPOS(0) :F(PA2.0)
PASIG = FTM[LEVCT, ≠ $STRING ≠ ] ≠ (RETURN)
GEN
G,I = G.I + 1
$G(!G(I) = A.1
$G(!G(I) = A.2
GEN = G.I
:NXTST
IDENT(EOP)
COND =
OUTPUT = ≠ LINE
IMAGE = LINE
READC
LINE = TRIM INPUT) :F(ENDGAME)
LINE = LINE
LINE = LINE
IMAGE = IMAGE LINE :F(READC)
PRINT
OUTPUT = ≠ LINE :F(READC)
ENDGAME
EOF = 1
:DUMP
OUTPUT(SCRATCH, 4, (50A1))
SCRATCH = OK
REG.D
RNAME BREAK(↑ ↑ ↑) ≠ NA LEN(1) =
SCRATCH = NA ↑$↑
SCRATCH = $NA ↑$↑
T.I = 1
SET.D
SCRATCH = $NA ↑↑ T.I ↑SET↑) ↑$↑
SCRATCH = $NA ↑↑ T.I ↑CLR↑) ↑$↑
T.I = LT(T.I $NA) T.I + 1
:E,REG
SCRATCH = EOR↑$↑
SUB.D
SNAME BREAK(↑ ↑ ↑) ≠ NA LEN(1) =
:F(E, SUB)
SCRATCH = NA ↑$\uparrow$
SCRATCH = SUBRNST[$NA,1$] ↑$\uparrow$
SCRATCH = SUBRNST[$NA,6$] ↑$\uparrow$
SCRATCH = SUBRNST[$NA,7$] ↑$\uparrow$
SCRATCH = SUBRNST[$NA,8$] ↑$\uparrow$

E.SUB
SCRATCH = ↑EORS↑
$T*I = 1$

GAT.D
SCRATCH = $S(↑GT↑ T*I) ↑$\uparrow$
SCRATCH = $S(↑G↑ T*I) ↑$\uparrow$
$T*I = LT(T*I,G,I) T*I + 1$

END FILE(4)

NO ERRORS DETECTED DURING COMPILATION
APPENDIX C

SNOBOL4

SNOBOL is an acronym for String Oriented symbolic Language. SNOBOL was developed at Bell Telephone Laboratories in 1962 and evolved into SNOBOL4, the programming language used to write the AHPL compiler.

The basic data element is a string of characters; i.e., a finite ordered sequence of symbols chosen from the available character set. String manipulation operations inherent in SNOBOL include separation, concatenation, replacement, and testing of contents. Strings may appear as literals or values of variables.

The most powerful feature of SNOBOL is the operation of examining strings for desired character structures, called patterns. Patterns may be simple or extremely complicated expressions. Like strings, patterns may appear in literal or variable form.

Arithmetic facilities for both fixed and floating point numbers are provided, but they are not extensive. A flexible array capability is also provided. Contrary to most languages, data type declarations are not necessary for SNOBOL variables. The last value assigned to a variable determines its data type.
The typical SNOBOL statement consists of three parts separated by blanks. Normally a statement is contained on a single card, but continuation onto successive cards is permitted. The three basic parts of a statement are:

1. LABEL - A name for the statement.

2. RULE - The string manipulation part of the statement. It is used for pattern matching, string formation, replacement, etc. It consists of four parts:
   a. String reference - giving the name of the string to be manipulated. Always present.
   b. Left side - specifying a pattern. Present if a pattern match is required.
   c. Equals sign - present if string formation or replacement.
   d. Right side - specifies a formation or replacement.

3. BRANCH - Specifies the label of the next statement to be executed. There are two types:
   a. Unconditional branch - of the form ;(NEXT). Control is always transferred to the statement labeled NEXT.
   b. Conditional branch - of the form :S(NEXT) F(GOTO). Control is transferred to NEXT if
the current rule succeeds or to GOTO if the current rule fails.

A SNOBOL program consists of a sequence of statements, the last of which is labeled END. Statements are executed in sequence unless a branch specifies otherwise. The program terminates when the END statement is reached. Branching to the END statement then terminates program execution.

SNOBOL4 programs are executed interpretively. The compiler passes through the program once and translates it into an internal representation which is then executed by the interpreter. At no time is an "object" program consisting of actual machine language instructions generated. Compiling interpretively affords greater flexibility and tracing capabilities.

Other significant features of SNOBOL4 include:

1. Error diagnostics are excellent.
2. Programs can modify themselves.
3. Recursive operations are very easy to define and use in SNOBOL4.
4. The free storage area is dynamic. When it is exhausted, it is automatically regenerated, if possible, by calling a "garbage collection" routine.
REFERENCES


