THE AHPL COMBINATIONAL LOGIC LANGUAGE
AND COMPILER

by
Charles Kent Goddard

A Thesis Submitted to the Faculty of the
COMPUTER SCIENCE COMMITTEE
In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE
WITH A MAJOR IN COMPUTER SCIENCE
In the Graduate College
THE UNIVERSITY OF ARIZONA

1971
STATEMENT BY AUTHOR

This thesis 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 thesis 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 head of the major department or the Dean of the Graduate College when in his judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Charles K. Maddux

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Frederick Hill
Professor of Electrical Engineering

15 May 1971
PREFACE

This study had as its purpose the implementation of the formalized digital hardware system design concepts proposed by Professors Frederick Hill and Gerald Peterson. Through these formalized concepts the design and production of electronic digital computer systems will be greatly simplified. The funds for the development of the compiler program were provided by both the University of Arizona Computer Science Committee and the University of Arizona Department of Electrical Engineering.

The author wishes to thank Mr. Michael Topliff of the University of Arizona Computer Center and Mr. Michael Gentry of the Electrical Engineering Department for their continuous aid and advice.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vi</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>2. AHPL HARDWARE</td>
<td>11</td>
</tr>
<tr>
<td>Class One Hardware</td>
<td>11</td>
</tr>
<tr>
<td>Class Two Hardware</td>
<td>14</td>
</tr>
<tr>
<td>Class Three Hardware</td>
<td>15</td>
</tr>
<tr>
<td>3. THE AHPL COMBINATIONAL LOGIC LANGUAGE</td>
<td>22</td>
</tr>
<tr>
<td>General Subprogram Format</td>
<td>23</td>
</tr>
<tr>
<td>Basic Elements</td>
<td>25</td>
</tr>
<tr>
<td>AHPL Operands</td>
<td>27</td>
</tr>
<tr>
<td>Register Declarations</td>
<td>33</td>
</tr>
<tr>
<td>Subprogram Definition Statements</td>
<td>34</td>
</tr>
<tr>
<td>Bookkeeping Statements</td>
<td>35</td>
</tr>
<tr>
<td>Terminal Statements</td>
<td>41</td>
</tr>
<tr>
<td>Hardware Assignment Statements</td>
<td>44</td>
</tr>
<tr>
<td>Function Call</td>
<td>44</td>
</tr>
<tr>
<td>Recursive List Expander</td>
<td>45</td>
</tr>
<tr>
<td>Recursive List Expander with Function Call</td>
<td>45</td>
</tr>
<tr>
<td>Boolean Equation</td>
<td>47</td>
</tr>
<tr>
<td>AHPL Functions</td>
<td>48</td>
</tr>
<tr>
<td>Illustrative Example</td>
<td>49</td>
</tr>
<tr>
<td>4. DESCRIPTION OF THE COMPILER</td>
<td>56</td>
</tr>
<tr>
<td>SNOBOL 4</td>
<td>56</td>
</tr>
<tr>
<td>Compiler Strategy</td>
<td>59</td>
</tr>
<tr>
<td>Phase One</td>
<td>59</td>
</tr>
<tr>
<td>Phase Two</td>
<td>62</td>
</tr>
<tr>
<td>Phase Three</td>
<td>63</td>
</tr>
</tbody>
</table>
## TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler Organization</td>
<td>63</td>
</tr>
<tr>
<td>Data Section</td>
<td>66</td>
</tr>
<tr>
<td>Origin</td>
<td>66</td>
</tr>
<tr>
<td>Hardware Generation Section</td>
<td>68</td>
</tr>
<tr>
<td>Output Vectors</td>
<td>75</td>
</tr>
<tr>
<td>Subroutine Name and Status Table</td>
<td>76</td>
</tr>
<tr>
<td>Subroutine Definition Section</td>
<td>76</td>
</tr>
<tr>
<td>Function Definition Section</td>
<td>79</td>
</tr>
<tr>
<td>Translator Section</td>
<td>82</td>
</tr>
<tr>
<td>Error Diagnostic Section</td>
<td>96</td>
</tr>
<tr>
<td>5. IMPLEMENTATION</td>
<td>98</td>
</tr>
<tr>
<td>Decoder Unit</td>
<td>98</td>
</tr>
<tr>
<td>Incrementor Unit</td>
<td>105</td>
</tr>
<tr>
<td>Adder Unit</td>
<td>105</td>
</tr>
<tr>
<td>6. CONCLUSION</td>
<td>119</td>
</tr>
<tr>
<td>APPENDIX: THE AHPL COMBINATIONAL LOGIC COMPILER</td>
<td>122</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>145</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1.</td>
<td>Primary Specifications of a Hypothetical Digital Computer</td>
</tr>
<tr>
<td>2.</td>
<td>Activity Program</td>
</tr>
<tr>
<td>3.</td>
<td>Derived Control Unit</td>
</tr>
<tr>
<td>4.</td>
<td>Class One Hardware Devices</td>
</tr>
<tr>
<td>5.</td>
<td>AHPL Delay Input-Output Patterns</td>
</tr>
<tr>
<td>6.</td>
<td>AHPL Pulse Generator</td>
</tr>
<tr>
<td>7.</td>
<td>Indirect Access Mechanism</td>
</tr>
<tr>
<td>8.</td>
<td>Sample AHPL Combinational Logic Language Statement Format</td>
</tr>
<tr>
<td>9.</td>
<td>AHPL Combinational Logic Language Character Set</td>
</tr>
<tr>
<td>11.</td>
<td>Examples of Register Declaration Statements</td>
</tr>
<tr>
<td>12.</td>
<td>AHPL Subprogram Definition Statements</td>
</tr>
<tr>
<td>13.</td>
<td>AHPL Bookkeeping Statements</td>
</tr>
<tr>
<td>14.</td>
<td>Subroutine Terminal Declaration Statement</td>
</tr>
<tr>
<td>15.</td>
<td>Interpretation of Recursive List Expander</td>
</tr>
<tr>
<td>16.</td>
<td>Subroutine XOR</td>
</tr>
<tr>
<td>17.</td>
<td>Logic Produced by XOR</td>
</tr>
<tr>
<td>18.</td>
<td>Subroutine CMPR and Function EQL</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>19</td>
<td>CMPR Produced Logic</td>
</tr>
<tr>
<td>20</td>
<td>General Operation of AHPL Combinational Logic Subroutine Compiler</td>
</tr>
<tr>
<td>21</td>
<td>SRSIG Flow Chart</td>
</tr>
<tr>
<td>22</td>
<td>CONNECT Flow Chart</td>
</tr>
<tr>
<td>23</td>
<td>EXPAND Flow Chart</td>
</tr>
<tr>
<td>24</td>
<td>Subroutine Definition Section Flow Chart</td>
</tr>
<tr>
<td>25</td>
<td>Function Definition Section Flow Chart</td>
</tr>
<tr>
<td>26</td>
<td>Examples of Intermediate Code Generated from AHPL Source Statements</td>
</tr>
<tr>
<td>27</td>
<td>Flow Chart of FUN</td>
</tr>
<tr>
<td>28</td>
<td>The Evaluation of a Boolean Expression by FUN</td>
</tr>
<tr>
<td>29</td>
<td>Function FOUR</td>
</tr>
<tr>
<td>30</td>
<td>Subroutine DECODE</td>
</tr>
<tr>
<td>31</td>
<td>Compiler Output from DECODE</td>
</tr>
<tr>
<td>32</td>
<td>Logic Produced by DECODE</td>
</tr>
<tr>
<td>33</td>
<td>Function PROP</td>
</tr>
<tr>
<td>34</td>
<td>Subroutine INCR</td>
</tr>
<tr>
<td>35</td>
<td>Compiler Output from INCR</td>
</tr>
<tr>
<td>36</td>
<td>Logic Produced by INCR</td>
</tr>
<tr>
<td>37</td>
<td>Subroutine ADD</td>
</tr>
<tr>
<td>38</td>
<td>Compiler Output from ADD</td>
</tr>
<tr>
<td>39</td>
<td>Logic Produced by ADD</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Basic Elements of the AHPL Combinational Logic Language</td>
<td>28</td>
</tr>
<tr>
<td>2. Production Rules for AHPL Combinational Logic Language</td>
<td>36</td>
</tr>
<tr>
<td>3. Patterns and Special Variables used in AHPL Combinational Logic Subroutine Compiler</td>
<td>60</td>
</tr>
<tr>
<td>4. AHPL Combinational Logic Compiler Sections</td>
<td>65</td>
</tr>
<tr>
<td>5. Subprograms of Compiler Hardware Generation Section</td>
<td>69</td>
</tr>
<tr>
<td>6. Line Format of Subroutine Name and Status Table</td>
<td>77</td>
</tr>
<tr>
<td>7. Function Subprograms of the Translator Section</td>
<td>84</td>
</tr>
<tr>
<td>8. AHPL Combinational Logic Compiler Error Messages</td>
<td>97</td>
</tr>
</tbody>
</table>
The initial design of combinational logic networks for an electronic digital system can be totally automated. The automated system proposed consists of a high level programming language which is called the AHPL Combinational Logic Subroutine Language, and a compiler program which compiles programs written in the language into combinational logic networks. The output from such compilation is in the form of hardware input lists which are suitable for use as input to subsequent automated design or production procedures. The language described is completely general but contains a feature which allows for the minimum allocation of hardware consistent with the program description of the logic desired.

The combinational logic design system described was developed concurrently with the development of an automated control unit logic generator. Together, the two systems represent the means for the automatic generation of the initial logic required in the design of electronic digital computer hardware systems.
CHAPTER 1

INTRODUCTION

The design of an electronic digital computer proceeds, generally speaking, in accordance with several clearly defined stages. The first of these is usually that of initial logic design. The objective of the initial logic design stage is to provide a source document to be used by most of the subsequent stages. This source document generally contains all of the basic specifications of the hardware system under consideration and describes what is termed its basic architecture.

Most of the subsequent design stages have been automated to various degrees of sophistication. Indeed, we see that automation is involved not only in further design but also in the actual production of the hardware itself. All of the software involved in these automated design and production procedures will utilize input which, ultimately, can be traced back to the document specifying the basic architecture of the system.

The initial logic design stage, because of the high level of creativity required, is still largely accomplished by hand. In spite of the fact that the task involves an
enormous quantity of dull and boring bookkeeping activities, experienced and clever human minds are, in general, a prerequisite to the production of an optimal, or even acceptable, initial logic design.

We feel that it is possible to automate the initial logic design stage of electronic digital systems and yet not transcend upon that area which requires those, primarily human, creative attributes. Such an automated system, moreover, would free the designer from the fatigue resulting from the prodigious clerical tasks and enable him to concentrate on activities more worthy of his intellectual effort. What is proposed is a system which utilizes a high level programming language to describe digital hardware circuitry. We would allow such hardware to be described in arbitrary detail. A program, consisting of an arbitrary number of statements, would be interpreted by the system as a complete architectural description of some digital computer.

Some experimentation with this idea has been reported. The ALERT system (Friedman and Yang, 1969) has been well documented and is apparently finding some use with the International Business Machine Corporation. The authors of this software report that the logic produced by their automatic logic generator is less than optimal. In a test which compared the logic produced by the automated
system against that which actually existed in the hand designed IBM 1130 computer, it was found that 300 percent more hardware was produced by the logic generator. This value was significantly reduced in later comparisons but never achieved as economical a design as that which was produced by hand.

The ALERT system makes use of the language APL. This language was invented by the mathematician Kenneth E. Iverson in 1962 and was reported in a book called APL (Iverson, 1962). It is generally involved with the manipulation of matrices and vectors and seems particularly well suited for the task of designing digital hardware networks. Few, if any, other programming languages can as efficiently convey the type of information required for this task or in such a diverse range of detail.

The details concerning the use of a programming language in the design of electronic digital computers are exhaustively treated in the text Introduction to the Organization and Design of Digital Hardware Systems (Hill and Peterson, 1971). A practical digital computer logic design system based upon the use of a programming language is described in "A Hardware Compiler" (Gentry, 1971b). Since this paper deals with one particular aspect of this design concept we must at this point familiarize the reader with some of the basic ideas involved.
The solution to a digital computer design problem generally begins with an agreement concerning the overall features to be possessed by the desired system. The number of registers, the size of the memory, and the word length are among the basic features that must be initially established. In addition to these, we must agree upon the number and type of instructions that will be included in the system's command repertoire.

After the basic machine architecture has been so established, we may proceed to model in symbolic notation those activities that must occur in the system as it processes information. The model of which we speak can be similar to a mathematical program where each assignment statement denotes the transfer of information among the registers (and memory) included in the design. Moreover, the compare branch statements can be viewed as the regulatory means by which control (in this case, our attention) is transferred to various segments of the program.

To illustrate this, let us consider a hypothetical computer the basic attributes of which are outlined in Figure 1. For the sake of simplicity, we will omit the consideration of all but the "memory reference" instructions. These are composed of a left justified three-bit operation code and a right justified eight-bit operand address. The bits are numbered from the left. Bit four
### REGISTERS:

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>12</td>
<td>Accumulator</td>
</tr>
<tr>
<td>AD</td>
<td>12</td>
<td>Memory Address Register</td>
</tr>
<tr>
<td>IR</td>
<td>12</td>
<td>Instruction Register</td>
</tr>
<tr>
<td>MD</td>
<td>12</td>
<td>Memory Data Register</td>
</tr>
<tr>
<td>PC</td>
<td>12</td>
<td>Program Counter Register</td>
</tr>
</tbody>
</table>

### MEMORY:

\[(M) (256 \times 12)\]

### MEMORY REFERENCE INSTRUCTIONS:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>or Function</td>
</tr>
<tr>
<td>000</td>
<td>Halt</td>
</tr>
<tr>
<td>001</td>
<td>Load Accumulator from MD register</td>
</tr>
<tr>
<td>010</td>
<td>Two's complement add of AC and MD</td>
</tr>
<tr>
<td>011</td>
<td>Store contents of AC</td>
</tr>
<tr>
<td>100</td>
<td>Logical (boolean) AND of AC and MD</td>
</tr>
<tr>
<td>101</td>
<td>Unconditional jump</td>
</tr>
<tr>
<td>110</td>
<td>Jump if AC is zero</td>
</tr>
</tbody>
</table>

Figure 1. Primary Specifications of a Hypothetical Digital Computer.

is called an "I/O-bit" and will not be treated in this discussion.

Figure 2 contains an APL-like program which describes the activities that must be supported by the computer specified in Figure 1. The program is presented for purposes of illustration and no attempt was made to render
start
AD ← FC
MD ← M\textsuperscript{↑AD}
IR ← MD
→ (IR\textsubscript{4}*50+IR\textsubscript{4}*6)
→ ((IR\textsubscript{1}∧IR\textsubscript{2}∧IR\textsubscript{3})*27+(IR\textsubscript{1}∧IR\textsubscript{2}∧IR\textsubscript{3})*7)
→ ((IR\textsubscript{1}∧IR\textsubscript{2}∧IR\textsubscript{3})*8+(IR\textsubscript{1}∧IR\textsubscript{2}∧IR\textsubscript{3})*17+(IR\textsubscript{1}∧IR\textsubscript{2} ∧ IR\textsubscript{3})*17+(IR\textsubscript{1}∧IR\textsubscript{2} ∧ IR\textsubscript{3})*9+(IR\textsubscript{1} ∧ IR\textsubscript{2} ∧ IR\textsubscript{3})*9+(IR\textsubscript{1} ∧ IR\textsubscript{2} ∧ IR\textsubscript{3})*9)
stop
w\textsuperscript{8}/AD←w\textsuperscript{8}/IR
→ ((IR\textsubscript{2}∧IR\textsubscript{3})*11+(IR\textsubscript{2}∧IR\textsubscript{3})*14+(IR\textsubscript{2}∧IR\textsubscript{3})*16)
MD ← AC
M\textsuperscript{↑AD}←MD
→ (25)
PC ← AD
→ (3)
→ ((∧/¬AC)*14+(∨AC)*25 )
w\textsuperscript{8}/AD←w\textsuperscript{8}/IR
MD ← M\textsuperscript{↑AD}

Figure 2. Activity Program.
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>((\neg IR_2 \lor IR_3) \times 20 + (\neg IR_2 \lor IR_3) \times 22 + (IR_2 \lor IR_3) \times 24)</td>
</tr>
<tr>
<td>20</td>
<td>AC \leftarrow AND(AC, MD)</td>
</tr>
<tr>
<td>21</td>
<td>\rightarrow (25)</td>
</tr>
<tr>
<td>22</td>
<td>AC \leftarrow MD</td>
</tr>
<tr>
<td>23</td>
<td>\rightarrow (25)</td>
</tr>
<tr>
<td>24</td>
<td>AC \leftarrow ADD(AC, MD)</td>
</tr>
<tr>
<td>25</td>
<td>PC \leftarrow INC(PC)</td>
</tr>
<tr>
<td>26</td>
<td>\rightarrow (2)</td>
</tr>
<tr>
<td>27</td>
<td>... operate instructions begin ...</td>
</tr>
</tbody>
</table>

Figure 2. (Continued).
it in as clever or efficient a manner as possible. Nevertheless the program should be sufficient to convey to the reader who possesses a reasonable understanding of digital computer systems the basic idea of the concept being entertained.

The interesting, indeed exciting attribute of such activity programs is that they can be transformed into unique hardware configurations on a statement by statement basis. To illustrate this, the reader is referred to the diagrams presented in Figure 3. Only part of the program is represented, but each statement of the program so diagrammed is indicated by the presence of a statement number between two vertical dashed lines. The diagram consists of lines, blocks and arrows. The lines represent signal paths among the blocks which are either gates, flip-flops, or delays. The arrows leading from the lines are also signal paths and carry enabling pulses which effect the operation indicated. A whole mapping of the program into such a diagram would result in a skeleton description of the computer's central processing unit. We could also include the actual configurations involved in the operations enabled by the control unit and thus be supplied with a primary description of the architecture of the present machine.
Figure 3. Derived Control Unit.
For the sake of brevity all of this will not be done. As a matter of fact, although a complete initial logic design source document will require such a total picture, this picture will be in a symbolic but not graphic form. If the reader is not yet convinced that we may both describe computer architecture from an activity program and map such programs onto a set of hardware, he is urged to reference the sources cited (Hill and Peterson, 1971; Gentry, 1971b).

This paper addresses itself to the problem implied by the occurrence of such statements as those numbered 20, 24, and 25 in the program of Figure 2. These statements reference what will be termed combinational logic subroutines. As will be seen, it is possible to describe combinational logic with the same language used to describe activity.

The problem of combinational logic subroutines is, of course, part of a larger problem: to establish a context free language with which digital computer architecture can be completely described and to develop a hardware compiler which will effect the mapping of programs written in this language onto a unique set of hardware. The output of such a compilation should be a document suitable as input to software in higher stages of computer development.
CHAPTER 2

AHPL HARDWARE

Before engaging in a discussion of the language, we must state in rather precise terms just what is included in the AHPL hardware domain. This domain can be said to consist of three levels. The first, or primary level includes two species of gates, a single species of flip-flop, and a logic level inverter. For the sake of simplicity, AHPL lumps the two gate types and the logic level inverter into the category of GATE and assumes that for each gate so allocated, one of the three boolean functions (AND, OR, and NOT) are assigned to it. The second class of hardware includes two devices called DELAYS and PULSE GENERATORS which are composed of devices from class one. Their composition is implicit, however, and they are allocated simply as devices with no further thought to their make-up. The third class of hardware supported by AHPL consists of those entities defined or declared in an AHPL program. An example of a class three hardware unit would be a register.

Class One Hardware

The AHPL gate appears to exist in an unlimited quantity. The AHPL compiler systems impose no limit to
the number of gates that can be allocated to a particular design. We will see, however, that there does exist a practical limit to the number of devices which can be handled during the course of any particular run. All gates remain in a central gate pool and are allocated as needed. Each gate is known primarily by its identifying number. The boolean function accomplished by a gate is stored as a function of its number. The output from a gate is also a function of its number.

As was already stated, an AHPL gate may perform the AND, OR or NOT functions. In addition, both the true and false outputs are available from every gate allocated.

No limitation on the number of inputs that can be assigned to any particular gate is imposed by the system. It is felt that a subsequent routine may be used to rectify any problems which may arise due to fan-in restrictions.

The truth tables and labeling conventions for the three types of AHPL gates is shown in Figure 4.

The AHPL flip-flop is a bistable storage device, conceptually equivalent to the common R-S or SET-RESET flip-flop. There are two categories of flip-flop in the AHPL system. These are (1) the control flip-flop (CFF), and (2) the register flip-flop. Each control flip-flop is known by a number. The register flip-flop, however, is known by either the number of the register and the bit
Figure 4. Class One Hardware Devices.
position, or the register name and bit position. The former is generally used when referencing one of the two input locations of the flip-flop and the later designation when the output of the flip-flop is being referenced.

Control flip-flops are not available to the combinational logic subsystem. These devices are for the exclusive use of the control unit compiler. An example of the use of the control unit flip-flop is shown in the diagram in Figure 3 of the introduction.

No restriction is placed upon the number of input signals that can be assigned to either the SET or RESET inputs of a flip-flop. A clean-up routine will be used to examine both input lists of every flip-flop allocated and to assign these input strings as inputs to an OR-gate when necessary.

The truth table and labeling conventions for the AHPL register flip-flop are also shown in Figure 4.

Class Two Hardware

The primary unit of this class is the delay. Although this device is not available to the combinational logic subsystem we will for the sake of completeness give it some attention. The delay is a two input single output device responsible for synchronizing enabling pulses from the control unit with the computer's central clock-pulse
train. Control pulses, generally from other delays, are reshaped, restored to the proper logic level and fired in unison with a central clock pulse. Figure 5 shows the labeling convention for this device together with a graph describing its input-output characteristics. A complete discussion of this device can be found in the dissertation written by Mr. Gentry (1971).

Pulse Generators constitute the remainder of Class Two hardware. These devices are similar to delays but are used to propagate a "complete" signal for asynchronous operations. A block diagram of this device is shown in Figure 6. The device is fed by two inputs one of which is the clock-pulse train maintained as the central timing reference of the computer. The other input originates in some fashion from the unit maintaining the asynchronous operation.

**Class Three Hardware**

This is the highest class of hardware supported by the system. Each unit in this class is constructed from designer specified numbers of elements from the previous two classes. A 16-bit register is an example. In general, the system being described fashions class three hardware units on the basis of user supplied program models. This class is unique in that it utilizes established class three
Figure 5. AHPL Delay Input-Output Patterns.
Figure 6. ANPL Pulse Generator.
units in the production of other class three units. For example, a central processing unit is composed of registers, bus's, logic units and one or more control units, all of which are of the same class.

We will limit our discussion in this section to the convention that will be observed concerning a feature called indirect access to logic units. This convention will pertain only to the subsystem being discussed in this paper. The main Hardware Compiler will utilize the same feature but with entirely different mechanics.

An array of OR-gates can serve, from the viewpoint of a logic unit, as a register image. This feature is of prime importance when we desire to use one logic unit from a variety of registers. The BUS concept being established here is illustrated in Figure 7. Depending upon the setting of CFF-1 and CFF-2, either register A or register B can access the logic unit from argument position "A". Likewise, either register C or register D can access the logic unit from argument position "B". Since both the true and false output from each OR-gate is available, only the Q-outputs need be propagated through the switches. The switches are enabled by the setting of control flip-flops from the control unit.
Figure 7. Indirect Access Mechanism.
The alternative to indirect access to a logic unit is of course to propagate the flip-flop outputs directly into the logic unit.

Since we are concerned here with the compilation of AHPL combinational logic subroutines and cannot avail ourselves of the machinery of the control unit compiler, we will establish the means of BUS construction on a tentative basis. The actual manner in which the total system will effect this description is at the present time open to speculation. We will, however, formulate our tentative mechanism in such a way as to be completely realistic.

Looking forward to the time when the combinational logic system is interfaced with the control unit system, we will make it possible for the control unit compiler to recognize that a multi-element switch is required and that the outputs from that switch must be propagated into certain elements of a BUS.

Our convention on the array of output signals from a logic unit can be summarized as follows: only those signals which require propagation into the SET inputs of register flip-flops will be rendered immediately available to the control unit system. However, since these signals will contain information pertaining to the complement output of the device from which they arose, that signal which is
to be propagated to the RESET inputs of the same flip-flops can be recovered rather easily.
The AHPL Combinational Logic Language was formulated in accordance with what were considered to be several a priori characteristics. As its development progressed, the criteria set was expanded to include those restraints imposed by the compiler system and new features as the complexities of present day digital systems was increasingly recognized.

First of all, the combinational logic language had to contain sufficient capacity for the unambiguous description of combinational logic. In addition, arbitrarily complex combinational logic units should be definable in relatively few program statements. The language had to be "bit" oriented rather than "vector" oriented: one should be able to specify combinational logic on a wire-by-wire basis. Finally, the language had to resemble as closely as possible that which is used to describe control units. Both language subsets, of course, are to be subsets of Kenneth Iverson's APL,

We have, it is felt, met these basic requirements. The language about to be discussed is both concise and
simple. It is completely general within the framework of combinational logic design. The only restriction that can be said to exist is that the designer must orient his design strategy so as to incorporate the subroutine-logic unit concept.

**General Subprogram Format**

An AHPL subprogram consists of a definition statement, a body, and an END statement. The body consists of an arbitrary number of executable statements. An executable statement can be of the bookkeeping, hardware assignment, or terminal declaration variety. Each of these statement types will be discussed in this chapter. There exists two species of subprograms in the AHPL language. One of these is, of course, the combinational logic subroutine. The other will be called the AHPL function. The AHPL function is used by the combinational logic subroutine in much the same way as a FORTRAN function is used by a Fortran main program.

Each combinational logic subroutine is considered to be a description of some combinational logic unit. There exist no facilities for linking the hardware produced by one subroutine to that being produced by another. Such linkage is in the domain of the control unit compiler. Each subroutine is treated in sequence. All register names and
functions referenced from a combinational logic subroutine body must be defined prior to the occurrence of the subroutine from which those references are made. Once defined, functions and registers are available to all subsequent subroutines.

With the exception of subprogram definition statements, all other AHPL statements must possess both a statement number and a statement key. Two sample AHPL statements are shown in Figure 8.

Register declarations are not a proper part of the combinational logic language but since they must precede any subroutine in which they are referenced, the topic of their declaration will be developed in a formal fashion in the next section. It must be pointed out, however, that this statement was one of the first that was established in the control unit language. As a consequence, its syntax is unlike that of any of the purely combinational logic language statements we will discuss.

```
242  B    K = K + 1
16   S    C[J] = 0
```

Figure 8. Sample AHPL Combinational Logic Language Statement Format.
The manner of subprogram presentation is quite important. The source deck should be structured so that all registers that will be required by the subroutines are declared. These declarations are normally followed by those AHPL functions that will be referenced from subsequent subroutines, and these are then followed by the subroutines themselves. No hard rules are presented. It is important only that all class three hardware units or configurations be defined or constructed before their use is called for by a subroutine.

Comments may be freely used and are denoted with the symbol * punched in column one. Statements may continue onto as many cards as necessary and continuation is denoted with a + punched in column one.

Basic Elements

Throughout the remainder of this chapter we will employ the facilities of a meta-language to define the AHPL Combinational Logic Language. That which will be used is a variation of Backus Normal Form as employed by T. E. Cheatham and Kirk Sattley in their paper on syntax directed compiling (1964). Since no fixed rules in the use of this meta-language are immediately available, we will take some liberties with its format. It is hoped that in so doing we will prepare the reader for the compiling methods to be
discussed in the next chapter. In addition, we will present in this discussion a complete set of rules for the generation of statements in the AHPL Combinational Logic Language as a by-product.

The characters ::= in the meta-language should be read: "is defined to be." A vertical slash is interpreted as an 'exclusive-or.' The symbols < and > are used to enclose what are termed nonterminal characters of the language being demonstrated. The reader is warned that a nonterminal character has nothing really to do with his first ideas of 'character.' A nonterminal character is a syntactic element that may be defined entirely in terms of other, previously defined nonterminal characters. Since our source language is grounded, there must ultimately occur some nonterminal which is defined completely in terms of terminal characters but the relationship is not always immediate.

We will use braces in the meta-language to enclose alternative terminal or nonterminal characters in a concatenated string. These braces should not be confused with brackets. Brackets are terminal characters of the language and are used to enclose subscripts. Also, one should be careful not to confuse the vertical bar of the meta-language with the slash (/).
The set of terminal characters available to the combinational logic language are shown in Figure 9. From these we present in Table 1, the formal definition of what are considered to be the basic syntactic elements of the language. The definitions in the table not only define a portion of the grammar of the language, but in addition establish the means by which certain elements of the language can be recognized. To illustrate this, we provide in Figure 10 a graphic representation of the parsing of the string "A C R [6]". The result is an inverted "tree", with the various "limbs" being labeled with nonterminal characters from the table. The string is "recognized" to be that element which will arise, discounting a grammatical error, at the root of the tree.

**AHPL Operands**

An AHPL operand may be an integer or a symbolic variable. The compiler will classify all variables in accordance with the type of statement in which they exist.

```
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
( ) * - + / , ^ V -> [ ] =
```

Figure 9, AHPL Combinational Logic Language Character Set.
Table 1. Basic Elements of the AHPL Combinational Logic Language.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;NULL&gt;</code></td>
<td>::=</td>
</tr>
<tr>
<td><code>&lt;LETTERS&gt;</code></td>
<td>::= A</td>
</tr>
<tr>
<td><code>&lt;DIGITS&gt;</code></td>
<td>::= 0</td>
</tr>
<tr>
<td><code>&lt;BLANKS&gt;</code></td>
<td>::=  ( b )</td>
</tr>
<tr>
<td><code>&lt;INTEGER&gt;</code></td>
<td>::= <code>&lt;DIGITS&gt;</code></td>
</tr>
<tr>
<td><code>&lt;IDENTIFIER&gt;</code></td>
<td>::= <code>&lt;LETTERS&gt;</code></td>
</tr>
<tr>
<td><code>&lt;UNOP&gt;</code></td>
<td>::= <code>1</code></td>
</tr>
<tr>
<td><code>&lt;FACTOR&gt;</code></td>
<td>::= <code>&lt;INTEGER&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ADDOP&gt;</code></td>
<td>::= <code>+</code></td>
</tr>
<tr>
<td><code>&lt;MULOg&gt;</code></td>
<td>::= <code>/</code></td>
</tr>
<tr>
<td><code>&lt;BINOP&gt;</code></td>
<td>::= <code>\&amp;</code></td>
</tr>
<tr>
<td><code>&lt;PREFIX&gt;</code></td>
<td>::= PREFIX [ <code>&lt;FACTOR&gt;</code> ]</td>
</tr>
<tr>
<td><code>&lt;SUFFIX&gt;</code></td>
<td>::= SUFFIX [ <code>&lt;FACTOR&gt;</code> ]</td>
</tr>
<tr>
<td><code>&lt;PART.VECTOR&gt;</code></td>
<td>::= <code>{ </code>&lt;PREFIX&gt;`</td>
</tr>
<tr>
<td><code>&lt;REDUCE.VECTOR&gt;</code></td>
<td>::= <code>( </code>&lt;UNOP&gt;<code> </code>&lt;BINOP<code>/</code>{ <code>&lt;UNOP&gt;</code> <code>&lt;IDENTIFIER&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;BIT&gt;</code></td>
<td>::= <code>&lt;UNOP&gt;</code> <code>&lt;IDENTIFIER</code> [ <code>&lt;FACTOR</code> ]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 10. Sample PARSE of ACR[6].
and with their location in that statement. As will be later explained, there exist two basic categories of variables. These are the "bookkeeping" and "hardware assignment" variables, respectively. No "bookkeeping" variable may possess a subscript. On the other hand, all "hardware assignment" variables are assumed to be subscripted and if no subscript explicitly exists, one is provided by the compiler.

In spite of the fact that all "hardware assignment" operand variables are assumed to be array elements, no formal array declarations are made except in the case of registers and indirect access subroutine definition arguments (dummy vectors). The declaration of registers and input bus's will be explained in the following sections. In general, the programmer may use subscripts in any manner he desires, so long as this manner conforms to the production rules of the language.

The compiler will associate a hardware device with every hardware assignment operand which either occurs to the left of an equal sign or as an indirect access subroutine argument. The compiler "knows" all of the devices associated with a register or a bus but does not remember this information in the case of other arrays. For this reason, only register names and bus names are allowed to exist in the REDUCE.VECTOR bit specification. As will
be seen, the recursive list expansion statement provides a reasonable alternative in the case of other arrays.

Rather than exclude the use of register element operands as left-hand assignment variables, we have provided the non-combinational logic facility of "loading" register flip-flops from within a combinational logic subroutine. Our flip-flop loading scheme will affect the connection of a right-hand expression signal (device output) with the "set" input of the specified flip-flop if the operand does not possess the "negation" symbol (\( \neg \)), and the "reset" input if the "negation" symbol is possessed. Since all such flip-flop loadings are under the jurisdiction of the control unit and must be "gated" in, we have provided through the special variable "CPULSE", the means by which this must be done. When CPULSE is first encountered in a right-hand expression, the combinational logic compiler will allocate an OR-gate, the true output of which will be associated with the operand. The special status of this gate will be preserved even if CPULSE itself is redefined. At some later time, when the subroutine in which CPULSE occurs is referenced by the control unit program, the control unit compiler will propagate a signal from the control unit into the INPUT of that special OR-gate. Thus, even though the flip-flop loading was defined in the subroutine, it is still actually controlled by the
control unit. Although this feature is never really required in a strictly combinational logic sense, it appeared to be useful in that the design of some logic units was simplified, and for that reason, the feature was included.

There are instances when the designer wishes to describe a logic unit for which no definite period of delay can be associated. Such units are said to be asynchronous. In the design of asynchronous logic units must be included the means by which the end of the operation can be detected and a "complete" signal returned to the control unit. The AHPL combinational logic design system uses the pulse generator described in Chapter 2 as the device from which the "complete" signal is initiated. The programmer may effect the allocation of such a device through the use of the special operand "RPULSE". When RPULSE is encountered as a left-hand assignment operand, the combinational logic compiler will associate with it the next available pulse generator. In addition, the identity of the device is preserved and is made available to the control unit compiler at the time the subroutine, in which this operand occurred, is referenced in the control unit program.

In conclusion it must be pointed out that although the combinational logic compiler places no explicit restrictions on the number of characters that may be used to represent an operand, this number should be limited to a
maximum of four characters. The reason for this is that there is no intrinsic difference between those variables specified in a combinational logic subprogram and those contained in the compiler. This fact will be made apparent in the next chapter. Since all of the compiler variables are either of a length of five or more characters, or contain a special character, the limiting of program variable names to strings of less than five characters will insure that no confusion will occur during compilation.

Register Declarations

Registers are declared by means of declaration statements the production rules for which are stated below:

\[
\text{DECLARE REGISTER} ::= D \text{ BLANKS} \text{ IDENTIFIER} = \text{INTEGER}
\]

The alphanemic string representing the IDENTIFIER portion of the rule will be known throughout a compilation run as a register name. The INTEGER portion will be interpreted as the length or number of bits of the register.

In practice the statement is presented to the compiler with the "D" punched in column one of a standard 80-column Hollerith card. The appearance of the first character of the remainder of the declaration may appear anywhere on the card. Imbedded blanks within the expression are not allowed. Sample declaration statements are
shown in Figure 11. It should be noted that no length restrictions are imposed. Registers may be defined with any arbitrary length.

Subprogram Definition Statements

As was stated at the beginning of this chapter, every AHPL subprogram is recognized by its definition statement. The definition of both species of subprogram always includes an argument list. The subroutine definition argument list may consist of no more than four arguments. Each such argument may be either the name of some previously defined register, or may be a subscripted operand, the identifier portion of which is not equal to any previously defined register name. The latter type of argument will be considered to be a bus and will be termed a dummy vector.

The subscript portion of the dummy vector type argument must be an integer. Furthermore, this integer will be interpreted to be the length of the desired bus. When this type of argument is encountered, a bus of the

<table>
<thead>
<tr>
<th>D</th>
<th>ACR=18</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>PCR=13</td>
</tr>
<tr>
<td>D</td>
<td>MDR=18</td>
</tr>
</tbody>
</table>

Figure 11. Examples of Register Declaration Statements.
desired length will be created. Subsequent reference in
the subroutine to the dummy vector will result in the use
of elements of that bus.

No other form of argument is considered legal in
the subroutine argument list. If an argument not conforming
to these specifications is encountered, an error message
will be issued and all processing terminates.

The AHPL function argument list may consist of up
to 8 elements. Any AHPL operand is a legal function
argument. It is permissible to specify subscripts and
names as separate arguments to be "joined" within the
function.

The formal production rules for both definition
statements are presented in Table 2. It should be noted that
individual arguments are separated by means of commas. Ex­
amples of typical definition statements are shown in
Figure 12.

**Bookkeeping Statements**

Bookkeeping statements constitute a major portion
of the language being discussed. It is through them that
we enable the language to be "bit" oriented in a very
efficient fashion. The designer need not specify each
gate or each register flip-flop directly but may merely
specify some general element in terms of some index or
Table 2. Production Rules for AHPL Combinational Logic Language.

\[
\langle \text{NULL} \rangle ::= \\
\langle \text{LETTER} \rangle ::= A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z \\
\langle \text{DIGIT} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \text{BLANKS} \rangle ::= \text{b} \mid \langle \text{BLANKS} \rangle \text{b} \\
\langle \text{INTEGER} \rangle ::= \langle \text{DIGIT} \rangle \mid \langle \text{INTEGER} \rangle \langle \text{DIGIT} \rangle \\
\langle \text{IDENTIFIER} \rangle ::= \langle \text{LETTER} \rangle \mid \langle \text{IDENTIFIER} \rangle \langle \text{LETTER} \rangle \\
\langle \text{MULOP} \rangle ::= \ast / \\
\langle \text{ADDOP} \rangle ::= + - \\
\langle \text{FACTOR} \rangle ::= \langle \text{IDENTIFIER} \rangle \mid \langle \text{INTEGER} \rangle \\
\langle \text{MULFAC} \rangle ::= \langle \text{FACTOR} \rangle \mid (\langle \text{AREX} \rangle) \\
\langle \text{TERM} \rangle ::= \langle \text{MULFAC} \rangle \mid \langle \text{TERM} \rangle \langle \text{MULOP} \rangle \langle \text{MULFAC} \rangle \\
\langle \text{AREX} \rangle ::= \langle \text{TERM} \rangle \mid \langle \text{AREX} \rangle \langle \text{ADDOP} \rangle \langle \text{TERM} \rangle \\
\langle \text{ARITH.EQ} \rangle ::= \langle \text{IDENTIFIER} \rangle = \langle \text{AREX} \rangle \\
\langle \text{UNCOND.BRANCH} \rangle ::= \rightarrow (\langle \text{INTEGER} \rangle) \\
\langle \text{COMPARE} \rangle ::= \langle \text{FACTOR} \rangle : \langle \text{FACTOR} \rangle \\
\langle \text{RESULT} \rangle ::= (\text{EQ}, \text{NEQ}) \mid (\text{LT}, \text{GE}) \mid (\text{LE}, \text{GT}) \\
\langle \text{DOUBLE.BRANCH} \rangle ::= (\langle \text{INTEGER} \rangle, \langle \text{INTEGER} \rangle) \\
\langle \text{CONDITIONAL BRANCH} \rangle ::= \langle \text{COMPARE} \rangle \langle \text{RESULT} \rangle \langle \text{DOUBLE.BRANCH} \rangle \\
\langle \text{B-STATEMENT} \rangle ::= \langle \text{INTEGER} \rangle \langle \text{BLANKS} \rangle B \langle \text{BLANKS} \rangle \{ \langle \text{ARITH.EQ} \rangle \\
\langle \text{UNCOND.BRANCH} \rangle \mid \langle \text{CONDITIONAL BRANCH} \rangle \}
Table 2. (Continued)

<DECLARE REGISTER> ::= D<BLANKS> <IDENTIFIER>=<INTEGER>

<SUBROUTINE ARG> ::= <IDENTIFIER> | <IDENTIFIER>[<INTEGER>]

<SUBR.ARG.LIST> ::= <SUBROUTINE ARG> | <SUBR.ARG.LIST>,

<SUBROUTINE DEF> ::= <BLANKS><SUBROUTINE>(<IDENTIFIER>(

<SUBR.ARG.LIST>))

<UNOP> ::= <NULL> | 7

<BINOP> ::= A | V

<PREFIX> ::= PREFIX[<INTEGER>|<IDENTIFIER>]

<SUFFIX> ::= SUFFIX[<INTEGER>|<IDENTIFIER>]

<PART:VECTOR> ::= {<PREFIX><SUFFIX>} / <UNOP><IDENTIFIER>

<REDUCE.VECTOR> ::= <UNOP><BINOP> / {<UNOP><IDENTIFIER> | 

<BIT> ::= <UNOP><IDENTIFIER>[<FACTOR>] | <UNOP><IDENTIFIER> 

| 0 | 1 | CPULSE | RPULSE | <REDUCE.VECTOR>

<LHS-BIT> ::= <UNOP><IDENTIFIER>[<FACTOR>] | <UNOP>

<IDENTIFIER> | CPULSE | RPULSE

<FUNC.ARG> ::= <BIT> | <FACTOR>

<FUNC.ARG.LIST> ::= <FUNC.ARG> | <FUNC.ARG.LIST>, <FUNC.ARG>

<FUNCTION DEF> ::= <BLANKS> FUNCTION ( <IDENTIFIER>

{<FUNC.ARG.LIST>|<NULL>})

<FUNCTION CALL> ::= <IDENTIFIER>(<FUNC.ARG.LIST>)

<END> ::= END
Table 2, (Continued)

\[\langle \text{OUTPUT DECL} \rangle \ ::= \langle \text{IDENTIFIER} \rangle \ (\langle \text{SUBR.ARG.LIST} \rangle ) = \langle \text{IDENTIFIER} \rangle \ | \ \langle \text{IDENTIFIER} \rangle , \langle \text{IDENTIFIER} \rangle \]

\[\langle \text{TERM.STATE} \rangle \ ::= \langle \text{INTEGER} \rangle \ \langle \text{BLANKS} \rangle \ T \ \langle \text{BLANKS} \rangle \ {\langle \text{END} \rangle } \ |
\langle \text{OUTPUT DECL} \rangle \}

\[\langle \text{BOOL.TERM} \rangle \ ::= \langle \text{BIT} \rangle \ | \ \langle \text{UNOP} \rangle \ (\langle \text{BOOL.EXPR} \rangle ) \]

\[\langle \text{BOOL.EXPR} \rangle \ ::= \langle \text{BOOL.TERM} \rangle \ | \ \langle \text{BOOL.EXPR} \rangle \ \langle \text{BINOP} \rangle \ \langle \text{BOOL.TERM} \rangle \]

\[\langle \text{BOOL.EQ} \rangle \ ::= \langle \text{LHS-BIT} \rangle \ = \ (\langle \text{BOOL.EXPR} \rangle ) \]

\[\langle \text{FUNC.CALL.ST} \rangle \ ::= \langle \text{LHS-BIT} \rangle \ = \ (\langle \text{FUNCTION CALL} \rangle ) \]

\[\langle \text{RECUR.LIST.EXP} \rangle \ ::= \langle \text{LHS-BIT} \rangle \ = \ (\langle \text{LHS-BIT} \rangle \ \langle \text{BINOP} \rangle
\{ \langle \text{BIT} \rangle \ | \ (\langle \text{FUNCTION CALL} \rangle )\}) \]

\[\langle \text{S-STATEMENT} \rangle \ ::= \langle \text{INTEGER} \rangle \ \langle \text{BLANKS} \rangle \ S \ {\langle \text{BOOL.EQ} \rangle | \ \langle \text{FUNC.CALL.ST} \rangle \ | \ \langle \text{RECUR.LIST.EXP} \rangle } \]

\[\langle \text{COMMENT} \rangle \ ::= * \]
SUBROUTINE DEFINITION STATEMENTS:

SUBROUTINE(ADD(ZPR[18], ZPD[18], OFL))

SUBROUTINE(BLGC(ACR, MDR, CTRL))

SUBROUTINE(INC(XQP[15]))

***ZPR, ZPD, and XQP are BUS arrays; ACR, OFL, MDR, and CTRL are registers

FUNCTION DEFINITION STATEMENTS:

FUNCTION(XOR(A, B, C, D, F, H, PP, TX))

FUNCTION(NGT())

FUNCTION(FVR(A))

Figure 12. AHPL Subprogram Definition Statements.

bookkeeping variable that will be repeatedly evaluated with respect to imposed restraints. Hence, each flip-flop of a 60-bit register may be referenced with but one "hardware" statement within a structure of arithmetic, compare, and branch statements.

There exist three fundamental types of bookkeeping statements allowed in the AHPL Combinational Logic Language. These are the arithmetic, unconditional branch, and
conditional branch statements. The production rules for each of these are presented in Table 2.

Every statement of this category must possess a statement key in the form of the character "B" and this key must be preceded by, and followed by, at least one blank. Each such statement must possess a statement number which is unique with respect to the subprogram in which it exists.

Arbitrarily complex arithmetic assignment statements are allowed. The only restriction imposed is that no real numbers are accepted; all arithmetic is in integer mode. Although the exponentiation operator is not offered by the language, it will be accepted by the compiler.

The conditional branch statement possess a syntactic structure which is clearly obvious from Table 2. It is composed of three parts, each of which must be separated from the other by at least one blank. The integers of the \langle DOUBL, BRANCH \rangle must of course be statement numbers to statements which exist in the subprogram. As with APL this statement is interpreted as follows: if the first of the relations specified is true then transfer will be made to the first statement number, and if is false transfer will be made to the second statement number.

We have used the notation \langle LE, GT \rangle, \langle EQ, NEQ \rangle, and \langle LT, GE \rangle to mean \langle , \rangle, \langle =, \rangle, and \langle , \rangle. The reason for this transliteration is that the second set of symbols were in
use for other tasks when this portion of the language was being formulated.

The \(<\text{UNCOD,BRANCH}>\) should by this time be self explanatory.

We will see examples of the use of statements of this type in the chapter on implementation. Meanwhile, however, it is perhaps illustrative to examine some standard examples presented in Figure 13. Each of the forms discussed above is represented.

**Terminal Statements**

As can be seen by referencing Table 2, there exist two classes of terminal statement. The first and most simple of these is the \(<\text{END}>\) statement which consists merely of the characters "END". The other is slightly more complicated and will be discussed below. Both forms, however, share the requirement of a statement number and the character "T" as a statement key.

The \(<\text{OUTPUT.DECL}>\) statement appears to be similar for both species of subprogram. The similarity is, however, only superficial. In the case of functions the statement is concerned with but a single device or signal whereas the subroutine category is concerned with up to two arrays of signals. To be specific, consider the example in Figure 14. The statement shown occurs in a subroutine where "S"
13  B  J=0
14  B  J:10 (LT,GE) → (15,17)
15  B  J=J+1
16  B  → (14)
17  T  END

1  B  JOG=0
2  B  4096:JOG (LE,GT) → (3,19)
3  B  JOG=(JOG*JOG/(2-JOG/(2-JOG))) - JOG*(-1)+1
4  B  → (2)

Figure 13. AHPL Bookkeeping Statements.

33  T  ADD(A,B)=S
45  T  FULAD(A,B,L)=G,S

Figure 14. Subroutine Terminal Declaration Statement.
is an array name. Because of the presence of this statement the device associated with every defined element of the S-array will be stored in a special location. When the subroutine is referenced by some statement in the control unit program, each of the devices associated with the S-array will be handed to the control unit compiler as the value of the subroutine.

It will be noted that facilities are provided for the declaration of up to two such output vectors. It is usually the case that one of these is a single element vector. However, either of the vectors can exist with a maximum of 60 elements. The order in which these vectors occur is important and will be preserved. The programmer must, when he writes the control unit program statement which references the subroutine, maintain this order relationship among the assignment variables placed to the left of the equal sign. The first of the declared vectors will be assigned to the first of the main program assignment variables, and the other to the second. With the exception of logic unit input arguments and control pulses, these vectors will be the only existing representatives of a combinational logic subroutine from the control unit compiler's viewpoint.

The function variety of this statement type causes the assignment of some device defined in the function as
the value of that function. We will defer further dis-
cussion of this and direct the reader to examine the ex-
amples presented in the chapter on implementation.

The argument list required by the statement is of no importance. Its inclusion is due to an attempt to match
the statement form to that which was presented in an early
version of the manuscript being written by Professors Hill
and Peterson.

Hardware Assignment Statements

All hardware assignment statements must possess,
in addition to a statement number, the character "S" as
a statement key. A hardware assignment statement consists
of a left-hand-side, also called a left part or left-hand
bit, an equal sign, and a right-hand expression. The
left-hand bit must be an element of some register or of
some gate array. The right-hand expression must be of one
of the four types to be discussed below. These are, as
defined in Table 2, a function call, a recursive list
expander, a recursive list expander with function call,
or a boolean expression.

Function Call

The right-hand expression of this statement type
consists of a character string previously defined as the
name of a function and a parentheses enclosed list of
arguments, the items of which are separated by commas. The entire right-hand expression so defined must be also enclosed in parentheses.

The argument list may consist of any entities which have been defined previous to the call. The identifier part of gate array element names are also valid arguments.

Recursive List Expander

The right-hand expression of this statement type consists of three items the first of which must be identical to the operand which occurred on the left of the equal sign. The second item must be a boolean operator and the third the name of some previously established device. The statement will cause the system to attempt to recover the device which is associated with the left-hand bit and to augment its input with the device associated with the remaining operand. If no recovery is possible, a gate of the type specified by the boolean operator will be allocated and associated with the name used as the left-hand bit. To the input of this gate will be assigned the device associated with the right-most operand. We illustrate in Figure 15 the manner in which this statement is interpreted.

Recursive List Expander
with Function Call

This statement type is similar to that which was just described. The only difference is that the right-most
Figure 15. Interpretation of Recursive List Expander.
entity is a function call rather than a bit operand. The production rules for this and the previous statement type are presented in Table 2.

Boolean Equation

The description of this statement is clearly not necessary. The rules for its production are present in Table 2.

It should be noted that we have assigned no precedence value to either operator. The system will attempt to assign as many inputs to a device as possible. All subexpressions are reduced to a single gate before the remainder of the expression is treated. The user may, through the use of parentheses, control the manner in which hardware is allocated. Eventually, the entire boolean expression will be reduced to the output of a single gate. The left-hand bit will then be assigned this gate as its value.

It was not mentioned above but is generally the case that the presence of a register element name as a left-hand bit will render the reduced value of an expression unrecoverable. This is because the software will introduce the gate output directly into the input lists associated with the particular flip-flop referenced. (This "register loading" feature was previously discussed.) The
gate representing the right-hand expression will then be unrecoverable to the program because no association between the left-hand bit and the device is made.

**AHPL Functions**

In addition to the definition and end statements already mentioned, an AHPL function subprogram may consist of any of the bookkeeping or hardware assignment statements so far discussed. We have already mentioned the use of a terminal declaration in this type of subprogram.

The operands occurring within a function subprogram may be classified as either local or global to that subprogram. Only those which are listed as arguments in the definition argument list are local. Note that this classification is unlike that which is commonly the case in other programming languages. The reason that this is so in the present language is that the compiler will supply a special name for every operand occurring in the argument definition list. In addition, if an operand appearing in the function body consists of an identifier and a subscript either or both of which appear also as arguments, the system will generate instructions to effect the joining of these entities before the statement containing the operand is processed. This is all quite complicated and can render the use of such subprograms quite inefficient.
It also makes it impossible to use anywhere in the sub-
program body a left-hand bit which itself occurs as an
argument. Such use will result in the system associating
hardware with system generated names. An example of the
use of function subprograms will be offered in the next
section.

Illustrative Example

To illustrate the language just described we will
present two sample design subroutines and indicate graph-
ically how they are associated with the logic configuration
they produce.

The first of the subprograms is shown in Figure 16.
It is called XOR and the logic unit produced by it will
effect the bit-by-bit exclusive OR function on the two
registers gated into it. It will be noted that indirect
access arguments are declared in the subroutine definition.
These will lead to the production of OR-gate bus's and the
names will from that point on be classified as dummy vec-
tors. In Figure 17 the logic produced from the subroutine
is shown in an anotated fashion. The program name associated
with each device is shown where this is possible. In addi-
tion, each device is numbered with an integer which indi-
cates the order of its allocation.
SUBROUTINE (XOR(A[6], B[6]))

1 B J=7
2 B J:1 (EQ, NEQ) → (6, 3)
3 B J=J-1
5 B →(2)
6 T XOR(P, Q)=X
7 T END

Figure 16. Subroutine XOR.

The second illustrative example is shown in Figure 18. This is subroutine CMPR which produces a configuration that will effect the comparison of the contents of two three-bit registers called AC and MD. Note that these registers are declared before the subroutine is defined. The third register declared will serve as the means of storing the results of a comparison. It will contain a binary "00" if register AC contains a number smaller than that contained by MD, a binary "01" is that number is equal to that contained by MD, and a binary "10" otherwise. This register will be loaded by circuitry defined in the subroutine. No output vector is declared.
Figure 17. Logic Produced by XOR.
D AC=3
D MD=3
D SWT=2

FUNCTION(EQL(A,B,C))
2 T EQL(S,V)=TX
3 T END

* *

SUBROUTINE(CMPR(AC,MD))
1 B J=0
2 B J: 3 (LT,GE) → (3,13)
3 B K=J
4 B J=J+1
5 B J: 1 (EQ,NEQ) → (6,8)
6 S XA[J]=(AC[J]∧¬MD[J])
7 B → (9)
8 S XA[J]=(AC[J]∧¬MD[J]∧¬XA[K])
9 S XZ[J]=(EQL(AC,MD,J))
10 S LFT = (LFT∧XA[J])
11 S RGT = (RGT∧XZ[J])

Figure 18. Subroutine CMPR and Function EQL.
Figure 18. (Continued).

12 B  \rightarrow \ (2)
13 S  SWT[1]= (CPULSE\ΛLFT)
16 S  \neg SWT[1]= (CPULSE\neg\Lambda LFT)
19 S  SWT[2]= (CPULSE\neg\Lambda RGT)
20 S  \neg SWT[2]= (CPULSE\neg\Lambda RGT)
21 T  END
CMPR utilizes the AHPL function subprogram called EQL which is also defined before the subroutine. This function, when supplied with values by the subroutine call, will develop that gate configuration necessary to effect the loading of "01" into the SWT register when the contents of both the AC and MD registers are identical. In other words, the function describes the logic required to detect the equality of the contents of the two input registers.

The actual loading of register SWT will be effected when a control pulse is received from the control unit. Statements 13 through 20 in the subroutine indicate the manner in which this is done. It should be noticed that both the "set" and "reset" inputs of the SWT register are referenced separately. As was previously mentioned, the register loading facility is an optional facility not actually required by a combinational logic system.

A diagram of the logic produced by the above subprograms is presented in Figure 19. Again, most of the devices are labeled with program names.
Figure 19. CMPR Produced Logic.
CHAPTER 4

DESCRIPTION OF THE COMPILER

The AHPL combinational logic subroutine compiler was written in the SNOBOL 4 programming language (Griswold, Poage and Polonsky, 1968) and implemented on the Control Data 6400 computer. The program consists of only 516 statements. Although one version of the compiler is being modified to interface with the total hardware compiler system, the configuration which will be discussed in this paper is a self contained independent compiler which requires no "main program" compiler for its successful operation.

SNOBOL 4

SNOBOL 4 is a recently developed character string manipulation language possessing features which make it highly attractive for experimental compiler writing. The basic data element of the language is the character string. There are provisions for concatenating, splitting, and matching these character strings. In addition, the language provides for the creation of complex structures called patterns which are very similar to the BNF production system introduced in Chapter 2.
In addition to the normal arithmetic operations, SNOBOL 4 offers the operations of post-matching value assignment, conditional value assignment, and indirect value assignment. Conditional value assignment is used throughout the compiler to assign character substrings from AHPL source statements as the value of various compiler variables. These variables and the conditional value assignment operation are built into the pattern structures. Upon a successful pattern match with a source statement, AHPL syntactic elements are available for further processing without intervening programming steps.

Once such a compiler variable has been assigned a character string as value, the character string itself may be assigned some value through the use of the indirect value assignment operation. This feature, too, is used throughout the compiler. As an example consider a register declaration statement. The whole source statement is matched with a pattern containing the production rules for a valid register declaration. Upon a successful match, the variable VECTOR will have as value those characters which represent the register name. The value of the variable LENGTH will be the integer defining the register's length. We may assign the value of a register counter to those characters representing the register name by setting $VECTOR equal to the value of that counter. SNOBOL 4 also allows
for the creation of variables during object time. We use this feature to link source attributes in a unique way. For example, in the register declaration example we create a unique variable containing the register name and assign it the value of LENGTH. This is done with the assignment statement: ("RL." $VECTOR) = LENGTH.

In addition to the above operations, we often create patterns during compilation of source statements. Again utilizing the above example, we would store those characters representing the register name in a pattern called REGISTER. This pattern eventually contains the names of all of the registers declared in a particular run. Character strings declared as register names will match this pattern successfully. As a consequence, both the register number and the register length for any particular register can be recovered throughout the compilation.

The above operations have enabled us to omit using symbol tables. It must be added, however, that the interpretative SNOBOL 4 compiler resides in memory throughout the compilation of AHPL source subprograms and it furnishes the symbol table and symbol table search machinery.

The last operation frequently used by the Combinational Logic compiler is that of post-matching replacement. Briefly explained, this operation allows us to replace that part of a string which successfully matches a pattern or
reference string with yet another string. We usually, but not always, use the null string for replacement. Thus, for example, when analyzing an argument list, we assign the character string representing an argument as the value of some compiler variable and replace the original with the null string. In this way, each argument is removed from the list and from further consideration.

Illustrated in Table 3 are some of the important patterns and variables maintained by the Combinational Logic Subroutine compiler.

Compiler Strategy

AHPL combinational logic subroutines are compiled into symbolic hardware lists in three distinct phases. These are (1) the translation phase, (2) an execution phase, and (3) a listing phase. Each subroutine is completely compiled before the next subroutine is read. Functions and registers must be defined before being referenced from within a subroutine. Functions and registers are global: once defined they are available to any of a sequence of subroutines.

Phase One

Each AHPL source statement is analyzed and translated into one or more compiler instructions. These, of course, are in SNOBOL 4 syntax. For AHPL hardware assignment
Table 3. Patterns and Special Variables used in AHPL Combinational Logic Subroutine Compiler.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTER</td>
<td>Contains as alternatives, all register names declared during run.</td>
</tr>
<tr>
<td>FUNVEC</td>
<td>Contains as alternatives, all function names declared during run.</td>
</tr>
<tr>
<td>SRVEC</td>
<td>Contains as alternatives, all subroutine names declared during run.</td>
</tr>
<tr>
<td>GATEVEC</td>
<td>Contains all AHPL defined gate names or array element names (including subscript). This vector is initialized at the start of each subroutine compilation.</td>
</tr>
<tr>
<td>DUMVEC</td>
<td>Contains the names of indirect access subroutine arguments.</td>
</tr>
<tr>
<td>OUTVEC</td>
<td>Contains the unsubscripted names of gate arrays which are defined as logic unit output vectors.</td>
</tr>
<tr>
<td>ARGVEC</td>
<td>Contains each of the arguments declared in an AHPL function definition argument list. This pattern is reinitialized for each AHPL function.</td>
</tr>
</tbody>
</table>

**Special Variables (created variables)**

<table>
<thead>
<tr>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;RL.&quot; N</td>
</tr>
<tr>
<td>&quot;RN.&quot; N</td>
</tr>
<tr>
<td>&quot;DA,&quot; M</td>
</tr>
</tbody>
</table>
Table 3. (Continued)

<table>
<thead>
<tr>
<th>Special Variables (created variables)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GT.&quot; N</td>
<td>Equal to type of Nth gate ($\land$, $\lor$, OR $\neg$).</td>
</tr>
<tr>
<td>&quot;G.&quot; N</td>
<td>Contains list of input signals to gate N.</td>
</tr>
<tr>
<td>&quot;OS.&quot; M</td>
<td>Contains list of inputs to the Mth pulse generator.</td>
</tr>
<tr>
<td>&quot;SET.&quot; M &quot;.&quot; L</td>
<td>Contains input list to &quot;SET&quot; input of Lth flip-flop of Mth register.</td>
</tr>
<tr>
<td>&quot;RESET.&quot; M &quot;.&quot; L</td>
<td>Contains input list to &quot;RESET&quot; input of Lth flip-flop of Mth register.</td>
</tr>
</tbody>
</table>
statements, these instructions always involve the reference (or nested reference) to one or more of the SNOBOL 4 function subprograms which constitute the compiler's hardware generation section. AHPL subroutine terminal assignment statements are translated into a null instruction (which is valid in SNOBOL 4). AHPL arithmetic statements are moved to the list of translated instructions essentially unchanged.

The list of compiler instructions generated for each statement of an AHPL subprogram is called the list of intermediate code and is the value of the compiler variable IWORD. Under normal circumstances, this list is not available to the AHPL programmer.

Phase Two

The processing of a combinational logic subroutine END statement initiates a transition into phase two of the compilation. A request is made of the resident SNOBOL 4 compiler to compile the recently constructed intermediate code list. If this is successful a direct jump to the first instruction of the resultant machine code is effected. In effect we have appended the Combinational Logic compiler with a set of new instructions which make extensive use of its hardware generation section.
This transition occurs only when a subroutine END statement is encountered. Function subprogram END statements direct the storage of the intermediate code list as the value of a variable created from the characters 'FUNCTION' and the value of the function counter FN_CT. Once this is done, control is passed back to 'ORIGIN'.

Phase Three

Combinational logic subroutine END statements are translated into an intermediate code list command to effect a direct jump to CALL5. This is the label of a statement in the compiler which initiates the joining of each of the signals associated with the declared output vector of the source subroutine. There may be two of these and each set of signals is treated separately. The resultant concatenation strings are stored in appropriate locations of the Subroutine Name and Status Table. This done, the current SUBRNST line, the register inputs and the assigned gates are printed in the output listing. The three phases of operation are summarized in the flow chart of Figure 20.

Compiler Organization

The Combinational Logic Subroutine compiler is composed of the 10 modules or sections described in Table 4. Some of the sections are entirely comprised of SNOBOL 4
Figure 20. General Operation of AHPL Combinational Logic Subroutine Compiler.
Table 4, AHPL Combinational Logic Compiler Sections.

<table>
<thead>
<tr>
<th>Section</th>
<th>Function Subprograms Contained</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>NONE</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>NONE</td>
</tr>
<tr>
<td>REGISTER DECLARATION</td>
<td>NONE</td>
</tr>
<tr>
<td>SUBROUTINE DEFINITION</td>
<td>NONE</td>
</tr>
<tr>
<td>FUNCTION DEFINITION</td>
<td>NONE</td>
</tr>
<tr>
<td>TRANSLATION</td>
<td>BTYPE()</td>
</tr>
<tr>
<td></td>
<td>STYPE()</td>
</tr>
<tr>
<td></td>
<td>TTYPE()</td>
</tr>
<tr>
<td></td>
<td>PASIG(STRING)</td>
</tr>
<tr>
<td></td>
<td>FNCALL()</td>
</tr>
<tr>
<td></td>
<td>FUN(XRAY)</td>
</tr>
<tr>
<td>HARDWARE GENERATION</td>
<td>COMP(ELEMENT)</td>
</tr>
<tr>
<td></td>
<td>CONNECT(TO, PLACE, FROMSIG)</td>
</tr>
<tr>
<td></td>
<td>EXPAND(OLD.SIG, NEWSIG, FUNC.)</td>
</tr>
<tr>
<td></td>
<td>FACTVAL(INDEXVAR)</td>
</tr>
<tr>
<td></td>
<td>GEN(X, Y)</td>
</tr>
<tr>
<td></td>
<td>REDUCE(B.NOP, RED, OP, LNG, NA, 1, NEG.1, NEG, 2)</td>
</tr>
<tr>
<td></td>
<td>SRSIG(BT.SG)</td>
</tr>
<tr>
<td>ERROR DIAGNOSTIC</td>
<td>NONE</td>
</tr>
<tr>
<td>SOURCE DOCUMENT INPUT</td>
<td>READ()</td>
</tr>
<tr>
<td>HARDWARE OUTPUT</td>
<td>NONE</td>
</tr>
</tbody>
</table>
function subprograms and access to these sections is therefore only by means of function calls.

Data Section

The Combinational Logic compiler is essentially a syntax directed compiler. The identification of source statements is accomplished by matching them to pattern structures which have been defined in much the same way as we defined the production rules of Chapter 2. Not all of the syntax rules of the AHPL combinational logic language are defined in the Data Section, however. To do this would tie up an enormous segment of memory and actually render the compilation process quite impractical (Gentry, 1971a). Some of the more complicated or less frequently used syntactic entities must be recreated in the Translator Section as they are required.

Also established in this section are the initial values of many of the counters and patterns used in the compiler.

Origin

The master control of the compiler resides in a short segment of instructions the center of which is labeled; 'ORIGIN'. The actual compilation process begins here. The input buffer is initialized so that the READ routine can establish a "look-ahead" into the source file.
READ will then always return one full source statement from this file. A call to READ is made from ORIGIN and a scan is made of the statement returned to ascertain if it is a register declaration, a function definition, or a subroutine definition. If it is none of these, an error message is issued and compilation ceases. Otherwise control is transferred to an appropriate section.

Regardless of where control is passed, it will always be returned to ORIGIN. The one exception will be when READ is called and the source file is empty: a direct jump to the end of the compiler program is effected.

Connected with ORIGIN is that segment of the compiler which establishes those registers declared in the source program. This activity was to be in the domain of the control unit compiler and as a consequence the section was merely "plugged" into the Combinational Logic Sub-routine compiler as a temporary measure. We do not include it in the hardware generation section. Most of the activities which occur when a register is declared were discussed at the beginning of the present chapter. The register name is added to the pattern REGISTER and both the name and the length of the register are stored as functions of the register counter: REGCT.
Hardware Generation Section

This section is composed of the seven function subprograms listed in Table 5. All of the logic produced by a combinational logic subprogram is produced by the units in this section. The reader should, as he glances down the table, recognize the purpose of most of the functions and its correspondence to one or another of the features of the combinational logic language as discussed in the preceding chapter. To reinforce this recognition we will briefly discuss each of the functions listed.

**GEN.** — Except for those gates allocated for an input bus to a subroutine, all of the gates allocated from a given source program are done so from this routine. GEN is very short. All that is required is that the gate counter 'GATCT' be incremented by unity and that the first and second arguments of the call be stored as the value of two variables created from the value of GATCT. Specifically, the first argument specifies the gate type and is stored as the value of: $( 'GT,' GATCT)$. The other is an input (or string of inputs) and is stored as the value of: $( 'G.' GATCT)$. GEN then assumes the value GATCT and returns.

**SRSIG.** — SRSIG is responsible for finding and returning as its value the device output label that has
been associated with the AHPL operand name sent to it as an argument. SRSIG will return "ZERO" and "ONE" for arguments which are equal to 0 or (0) and 1 or (1), respectively. When the call argument is equal to the string "CPULSE", SRSIG will request an OR gate from the function GEN. The output label of this gate will be stored in the first location of the current line of the Subroutine Name and Status Table. If the call argument to SRSIG is the null string, a value "DEFAULT" is returned. A flow chart of this subprogram is presented in Figure 21.

**REDUCE.** -- AHPL source operands which are syntactically equivalent to reduced or partially reduced vectors are translated into calls to this subprogram. The arguments to the function are stated in Table 3. Each of these can be recognized as those substrings of the operand presented in Table 1.

**CONNECT.** -- This subprogram carries out a function which is almost opposite that carried out by SRSIG. The second call argument is assumed to be an output label of some hardware device (generated through the compilation of some expression on the right of an equal sign). This device label is associated with the string-valued first argument by means of indirect value assignment. If, however, the argument TO,PLACE contains the name of a register, that is
Figure 21. SRSIG Flow Chart.
TO, PLACE is the name of some flip-flop, the device label is added to the SET or RESET input lists of that flip-flop. The decision as to which list to augment is determined by the presence or absence of a unary operator prefix to the bit identifier.

If the first call argument to function CONNECT is equal to the reserved operand "RPULSE", a PULSE GENERATOR is allocated and the value of the second call argument assigned as its input. The output label of this device is stored in the last location of the current line of the Subroutine Name and Status Table.

A flow chart of this subprogram is shown in Figure 22.

EXPAND. -- Recursive list expansion statements will be translated into a call to this subprogram. If the first call argument to the function references a register flip-flop, the same action is taken as described for the function CONNECT. Otherwise EXPAND attempts to find a device which has been previously assigned as the value of this argument. If successful, the value of the second call argument is assigned to the input list of this device. If the attempt is unsuccessful, the function GEN is referenced and the gate allocated is associated with the string-valued first call argument.

A flow chart of EXPAND is presented in Figure 23.
Figure 22. CONNECT Flow Chart.
Figure 23. EXPAND Flow Chart.
**COMPLEMENT.** -- As illustrated in Table 3, COMPLEMENT returns the logical inverse of its call argument. Usually this argument will be a gate output label. The complementation of this label entails changing the character "T" to the character "F", and vice versa. For any other input call argument, the unary operator is either added or removed.

**FACTVAL.** -- All AHPL operands are sent to hardware generation routines as strings. Variable subscripts, if present, must be evaluated. FACTVAL tests its argument against the syntactic definition for an integer. If this test is successful, the argument is returned as the value of the function. If the test is unsuccessful, the argument is evaluated and this value is returned.

Output Vectors

Both CONNECT and EXPAND test their association operands against the pattern OUTVEC. When this test is successful, an element of one of the two possible output vectors is generated and assigned the device output label contained in the second argument. These created elements utilize the value of the subscript to specify the position of the vector element. The last task of phase three of the compilation is to sort these signals and concatenate them.
into long strings in preparation to storing them in certain locations of the Subroutine Name and Status Table.

Subroutine Name and Status Table

The logic generated through the compilation of combinational logic subroutines is made available to the control unit compiler through a dimensioned array of 15 rows and eight columns called the Subroutine Name and Status Table. All of the input-output and control information pertaining to any single logic unit is stored in one row of the table. The row pointer is in fact the subroutine counter SR,CT. Consequently that subroutine name which is associated with a certain logic unit is itself equal to row number of the appropriate line in the table. The row format of the table, referred to as SUBRNST in the program, is presented in Table 6.

Subroutine Definition Section

ORIGIN passes control to this section whenever a valid subroutine definition statement is encountered. Upon assuming control, this section begins by incrementing the subroutine counter SR,CT and initializing the patterns INDEXVAR, GATEVEC, and OUTVEC. The subroutine name is assigned the value of SR,CT and the pattern SRVEC is assigned the value SRNAME as an alternative.
Table 6. Line Format of Subroutine Name and Status Table.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp</td>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>o₁</td>
<td>o₂</td>
<td>rp</td>
</tr>
</tbody>
</table>

cp:  
- a. Null; no entry.  
- b. Control pulse (CPULSE) requested; gate output label is stored.

aᵢ:  
- b. Alphabetic: iᵗʰ argument is a register.  
- c. Numeric: iᵗʰ argument was a bus; number of first gate of bus is stored.

oᵢ:  
- b. Alphanumeric: iᵗʰ output vector exists; string of device output labels are stored.*

rp:  
- b. Output label of PULSE GENERATOR.

*Each device label of output vector is that line which is to be propagated to the "SET" input lists of some register. The complement of each device label must also be gated into the "RESET" input lists of each register flip-flop.
Once the various parameters and patterns have been redefined, the subroutine argument list is processed. Each argument, beginning with the left-most is removed from the argument list. An isolated argument is examined to see if it is a register or a bus. If it is the latter, the OR gates constituting this bus are allocated at this time and the number of the first gate of this bus is stored in the proper location of the Subroutine Name and Status Table. The location is computed as one plus the position count of the argument in the list. A register argument is stored unchanged in the proper table location.

Finally, after the last argument of the argument list has been encountered, control passes to a subsegment of this section which effects the sequential reading of the remainder of the source statements contained in the subroutine. As each source statement is read, its statement number, key, and leading blanks are removed. A statement label, called LB.L is fashioned by prefixing the statement number with the subroutine name. The statement key is then tested and an appropriate compiler translator subprogram called.

The last statement of any combinational logic subroutine must be the END statement. The processing of this statement is detected by examining a flag called EOF.2. When EOF.2 is "on", it is reset to "off" and a request is
made of the SNOBOL 4 compiler to compile IWORD. The successful compilation of IWORD is always followed by a direct jump to the first instruction of the newly compiled code.

A flow chart of this section is presented in Figure 24.

Function Definition Section

The structure and function of this section is very similar to that of the section just discussed. The function counter FN.CT is incremented and its value is assigned to the character string representing the function name. In addition, this character string is added as an alternative to the pattern FUNVEC.

Each of the arguments of the function definition argument list is removed and assigned an integer equal to its occurrence in that list with the left-most argument being the first. The pattern ARGVEC is repeatedly redefined so that it will possess, as alternatives, each of the arguments.

A function indicator FFLAG is set to some non null value. Then, each of the statements belonging to the function are sequentially read, and stripped of statement number, key, and all leading blanks. As directed by KEY, one of the three basic translator routines is called. The last statement of the function, an END statement is not
Figure 24. Subroutine Definition Section Flow Chart.
Figure 24. (Continued).
translated but causes the storage of the intermediate code list as the value of a variable formed from concatenating the word "FUNCTION" with the value of the function counter, FN.CT. This done, control is transferred to ORIGIN.

Figure 25 contains a flow chart of this section.

Translator Section

The translation of an AHPL source statement must of course be preceded by the recognition of that state­ment. The combinational logic compiler is essentially syntax directed. However, instead of storing the syntax specifications for the source language in a table and utilizing generalized recognition algorithms, the compiler limits its recognition capability to just those statement forms established in Chapter 2. As has already been mentioned, some of the basic source language specification were established as patterns in the Data Section. The remainder are repeatedly created as needed within three main translator routines. Consequently, some of the syntax of the source language is embedded within the structure of these routines.

Table 7 summarizes the names and arguments of the six SNOBOL 4 function subprograms which constitute the translator section.
Figure 25. Function Definition Section Flow Chart.
Table 7. Function Subprograms of the Translator Section.

<table>
<thead>
<tr>
<th>Compiler Function Name</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTYPE</td>
<td>NONE</td>
</tr>
<tr>
<td>STYPE</td>
<td>NONE</td>
</tr>
<tr>
<td>TTYPE</td>
<td>NONE</td>
</tr>
<tr>
<td>PASIG</td>
<td>STRING</td>
</tr>
<tr>
<td>FNCALL</td>
<td>NONE</td>
</tr>
<tr>
<td>FUN</td>
<td>XRAY</td>
</tr>
</tbody>
</table>

**BTYPE.** -- Every AHPL source statement with the character "B" as a statement key will illicit the service of the compiler subprogram BTYPE(). This routine, then is responsible for recognizing and translating AHPL bookkeeping statements. Arithmetic statements require only editing: at least one blank must separate each operator from its preceding and succeeding operands. The assignment variable in arithmetic statements is added to the list of alternatives of the pattern INDEXVEC.

Conditional and unconditional branch statements require slightly more extensive treatment. Their translation into SNOBOL 4 is clearly shown in Figure 26.
AHPL SOURCE STATEMENT  |  INTERMEDIATE CODE
---|---
A. Bookkeeping statements from SUBROUTINE ADD:
1   B  J=1   |   ADD1  J=1 ;
3   B  J:8 (EQ,NEQ) \rightarrow (14,5)  |   ADD3  EQ(J,8)  :S(ADD14)F(ADD5) ;
4   B  \rightarrow (3)  |   ADD4  :(ADD3) ;
10  B  L=L+J/2-1  |   ADD10  L=L+J/2-1 ;

B. Hardware assignment statements from SUBROUTINE BNQ:
4   S  C[K] = 0  |   BNQ4  DEST. = ↑C[K]↑ ;
              |   CONNECT(DEST.,SRSIG(↑0↑)) ;
              |   T.3=↑T↑GEN(↑V↑)SRSIG(↑T.1↑)SRSIG(↑T.2↑) ;
              |   CONNECT(DEST.,SRSIG(T.3)) ;
              |   EXPAND(DEST.,SRSIG(↑G[J]↑),↑V↑) ;

Figure 26. Examples of Intermediate Code Generated from AHPL Source Statements.
### AHPL Source Statement

**C.** Hardware assignment statement from `FUNCTION TXL`: (A, B, and M are function arguments)

<table>
<thead>
<tr>
<th>AHPL Source Statement</th>
<th>Intermediate Code</th>
</tr>
</thead>
</table>
| | `TEMP1 = ↑FTAB[LEVCT,1]↑↑[↑FTAB[LEVCT,3]]↑↑↑ ;`
| | `TEMP2 = ↑FTAB[LEVCT,2]↑↑[↑FTAB[LEVCT,3]]↑↑↑ ;`
| | `T.1=↑T↑GEN(↑↑↑↑,SRSIG(TEMP1)SRSIG(TEMP2)) ;`
| | `CONNECT(DEST.,COMP(SRSIG(↑↑↑↑))) ;` |

### D. Call of TXL from SUBROUTINE CLIP:

<table>
<thead>
<tr>
<th>AHPL Source Statement</th>
<th>Intermediate Code</th>
</tr>
</thead>
</table>
| 6  $ GR[K] = (TXL(AC,DA,L)) $ | **CLIP6** `DEST. = ↑GR[K]↑ ;`
| | `NLEVCT = LEVCT ; LEVCT = LEVCT + 1 ;`
| | `FTAB[LEVCT,1] = ↑AC↑ ;`
| | `FTAB[LEVCT,2] = ↑DA↑ ;`
| | `FTAB[LEVCT,3] = FACTVAL(↑L↑) ;`
| | `FCODE = CODE(↑FUNCTION1.C↑) :S[FCODE] ;`
| | **CLIP6.R** `CONNECT(↑GR[K]↑↑,SRSIG(↑FR↑↑)) ;`
| | `LEVCT = LEVCT - 1 ;`
| | `FCODE = ;` |

*Figure 26. (Continued)*
**STYPE.** -- When the character "S" is extracted as a statement key, the statement is translated by the routine called STYPE(). This routine creates patterns which will allow it to recognize three statement forms. These are the function call, the recursive list expansion, and the recursive list expansion with function call statements. If the source statement is not recognized as one of these three, it is assumed to be a boolean expression and the function FUN, to be discussed below, is called with the boolean expression as an argument. In every case, STYPE separates that string located to the left of an equal sign and assigns it as the value of LHSIDE. This string will be the first argument of either CONNECT or EXPAND.

Also shown in Figure 8 is the intermediate code generated from the three simple statement forms recognized by STYPE. The actual mechanics of the intermediate code generation and statement recognition can best be understood by referencing both the compiler program listing and a SNOBOL 4 manual.

**TTYPE.** -- All terminal statement forms result in the transfer of control to the subprogram called TTYPE. This routine translates subroutine END statements into a direct jump to SCALL5 and sets the flag EOF.2 to some non null value. Function subprogram END statements are ignored.
Both subroutine and function terminal assignment statements are treated by this routine. The function terminal assignment statement is translated into a command to assign the right hand operand as the value of a return variable formed by prefixing the function number of the current AHPL function with the alphanumeric characters "FR". These variables are recreated by STYPE when a function call is recognized. This "FR" variable will of course be an argument to the function SRSIG. When the call is effected during the execution phase, the value of the FR variable is sent to the function SRSIG. Since its value is a character string defined in the AHPL function, that is a character string which has been associated by either CONNECT or EXPAND with some device output label, it is the device label that is returned.

Subroutine terminal assignment statements, it will be recalled, may contain a list of up to two vector names as the right hand expression. These vector names are removed from the list and assigned an integer. The pattern OUTVEC is assigned the vector names as alternatives. The device label of the \( i^{th} \) element of either the first or the second output vector will be stored as the value of the variable \( j \)." \( i \) where \( j \) can be either one or two.

**PASIG.** -- Every AHPL operand encountered during the translation phase of compilation is sent to PASIG for
editing. This routine effects some rather complicated but important operations on each directly or as a result of recognizing them to be of certain classes. First of all, PASIG recognizes two basic categories of operands and two categories of operand environment. There are bookkeeping and hardware variables and they may occur in either subroutines or functions.

AHPL subroutine hardware operands are returned imbedded in the Control Data 6000 Series version of single quotes: up-arrows. Bookkeeping variables are returned unchanged.

PASIG too carries on some activities in intermediate code generation. Such activities can occur whenever the operand it is to edit occurred in some AHPL function subprogram and either the entire operand or one or more of its parts has previously appeared in the argument list of that function's definition. In such cases the actual operand will not be available until the function is called during the execution phase of compilation. When it is available and the function is called there must exist an instruction in the function which will effect the fetching of the operand from the function call table. This task is made complicated by the fact that both the operand name and the operand index, or subscript, could have been declared arguments in the definition. Not only must we fetch these
items from the function call table but must in addition join them together to create the intended operand.

PASIG will effect the generation of such instructions when necessary. A counter called TP.T is maintained and incremented by unity whenever an operand requires creation from at least two parts. The concatenation of these parts is assigned as the value of a temporary variable formed by concatenating the characters 'TEMP' with the value of TP.T. This character string, moreover, is returned to the caller as the value of PASIG.

**FUN.** — All AHPL boolean expressions not recognized by STYPE are sent as the value of XRAY to the recursive boolean expression translator named FUN. A sequence of intermediate instructions are generated as a result of the analysis of the expression. The operands and operators of the expression as well as the precedence relations implied by the organization of the expression are reflected in the intermediate code sequence.

The manner in which FUN translates a boolean expression can best be seen by examination of the flow chart of Figure 27 and the example of Figure 28. This subprogram makes use of the recursive capability of the SNOBOL 4 function subprogram. When FUN ascertains that its argument contains parenthesized subexpressions it merely calls
Figure 27. Flow Chart of FUN.
Figure 27. (Continued).
Figure 28. The Evaluation of a Boolean Expression by FUN.
itself with an argument made equal to the subexpression less the leading and trailing parentheses. Eventually \textsc{fun} will be called with an argument which contains only \textsc{ahpl} operands and boolean operators. When this occurs, it sets about generating an instruction which will effect the execution phase call to the function \textsc{gen} for as many different types of boolean operators as exist in the unparenthesized expression.

No precedence is assigned to the boolean operators. An operator status is maintained and when this status differs from the most recently encountered operator, a new instruction is generated.

In addition to being sent to \textsc{pasig}, most of the \textsc{ahpl} operand encountered in the boolean expression are imbedded in the characters 'SRSIG(' and ')' before being stored as a new entry in the local \textsc{list}. This variable contains not just the operands of the expression but a sequence of calls to the function \textsc{srsig} with the operands as arguments. Every element of this list will be written as the second argument to \textsc{gen} when an instruction referencing this function is generated. Thus, in Figure 28 we see this argument position being taken up with:

\textsc{srsig}(\text{At}) \textsc{srsig}(\text{Bt}).
One of the exceptions occurs when the encountered operand is a reduced vector. Matching this operand implies its breakdown into various parts and we use these parts as arguments to the function REDUCE. The function is not called at this time. Instead the call is loaded as one of the elements of LIST just as occurred above.

Finally, if a unary operator is encountered adjacent to a left parenthesis, either of the above discussed items is imbedded in the characters 'COMP(' and ')'.

FUN returns as its value the contents of LIST. If the original boolean expression was well-formed, LIST at this time should contain but one entity.

**FCALL.** -- Whenever STYPE encounters a function call in an AHPL source statement, the services of the subprogram FNCALL are obtained. This routine is responsible for generating the instructions in the intermediate code list which will effect the execution phase loading of the function call push down stack FTAB with the operands specified by the function call argument list, the execution phase request for SNOBOL 4 compilation of the intermediate code list established from the phase one compilation of the named AHPL function, and the direct jump to the first instruction of this code. The original intermediate code from the AHPL function is not used. A copy of it is
made and assigned a unique name. This copy is augmented with a statement which will effect a return to the next statement in the list from which the call is being made.

FTAB is a dimensioned array of four rows and eight columns. Each row corresponds to a particular level of function call. The arguments are loaded into columns starting with the left-most. In addition to the above instructions, additional instructions are generated which effect the incrementation of the row pointer LEVCT. The variable NLEVCT is assigned the value of LEVCT before it is incremented. In this way there are pointers to the current and last row being used in the table. Since there are only four rows, only four nested function calls are allowed. These calls, however, may be to the same function from which the call is made.

Error Diagnostic Section

The combinational logic compiler maintains a rather small list of error messages. These are summarized in Table 8. In several cases, the message will be augmented with the output of the variables or operands which caused the trouble.

It is possible that the SNOBOL 4 compiler itself will provide execution diagnostics. So far it has been experienced that this facility is limited to cases where a
Table 8. AHPL Combinational Logic Compiler Error Messages.

***UNRECOGNIZED ARGUMENT IN SRSIG.
   ARGUMENT WAS:

***SYNTAX ERROR. THIS STATEMENT NOT RECOGNIZED.

***ATTEMPT TO EXCEED MAXIMUM ALLOWABLE NUMBER OF SUBROUTINES.

***ATTEMPT TO EXCEED THE MAXIMUM ALLOWABLE NUMBER OF ARGUMENTS IN SUBROUTINE OR FUNCTION.

***LENGTH OF INPUT BUS FOR DUMMY SUBROUTINE ARGUMENT OMITTED.

***UNRECOGNIZABLE KEY.

***SNOBOL 4 COMPILE ERROR IN OBJECT.
   (Failure to compile intermediate code list.)

***DGOTO NOT RECOGNIZED,

***TROUBLE WITH XRAY BY FUN-FUNCTION,
   TROUBLE STARTED WITH:

***TROUBLE IN EXPAND. NEW SIGNAL; OLD SIGNAL;
   HARDWARE NAMED COULD NOT BE FOUND.

***DESTINATION NOT RECOGNIZED BY CONNECT, DESTINATION WAS:
   SOURCE WAS:

statement number referenced in a GOTO was inadvertently mispunched or omitted. Such cases can be immediately spotted by checking the source program listing.
CHAPTER 5

IMPLEMENTATION

To illustrate the usefulness of the combinational logic compiler system we will present three sample logic unit design programs. Due to the expense involved we find it impossible to avail ourselves of all of the features offered by the compiler system. The examples offered were written so as to minimize the time required for their compilation without sacrificing their effectiveness in demonstrating the power of the compiler and the sufficiency of the source language.

Decoder Unit

The AHPL function FOUR and the AHPL subroutine DECODE describe the initial logic configuration of a unit which will effect the translation of the binary outputs of the eight bit ACR register into the 20 signals required to enable the filaments of a three digit octal display. This display will be capable of indicating the octal numbers 000 through to 377.

DECODE works by breaking up the register into two groups of three bits and one group of two bits. A bookkeeping variable is initialized and then incremented to a
maximum value of four. For each value of this variable the function FOUR is called. Four possible gate configurations can be chosen in the function and the desired one is obtained through the value of the bookkeeping variable. The gate configurations are developed from the left-most two bits of the group specified in DECODE. It should be noted that FOUR was written for the explicit use of DECODE and refers explicitly to bit positions of ACR. It was written this way for purposes of economy only. We might just as well have established this function so that the register name and group index were to be sent to it as arguments. If this had been done, however, several more complex SNOBOL 4 statements would necessarily be executed each time the function was called.

The computer listing of function FOUR and subroutine DECODE are shown in Figures 29 and 30, respectively. A listing of the pertinent line of the subroutine name and status table, current register inputs, and the gates assigned is shown in Figure 31. Finally, to graphically demonstrate the architecture established by the compiler, we present in Figure 32 a block diagram of the gates listed in Figure 31.
FUNCTION(FOUR(J))

NN = J

NN: 1  (EQ, NEQ) r > (5, 3)
NN: 2  (EQ, NEQ) r > (7, 4)
NN: 3  (EQ, NEQ) r > (9, 11)

S FOUR = (¬ACR[K] ∧ ¬ACR[N])
B r > (12)
S FOUR = (¬ACR[K] ∧ ACR[N])
B r > (12)
S FOUR = (ACR[K] ∧ ¬ACR[N])
B r > (12)
S FOUR = (ACR[K] ∧ ACR[N])

T FOUR(A, B) = FOUR
T END

Figure 29. Function FOUR.
AMPL COMBINATIONAL LOGIC SUBROUTINE LISTING

SUBROUTINE(DECODER(ACR)):

* * * * *

DECODER DESCRIBES THE LOGIC CONFIGURATION REQUIRED TO
DECODER THE ACR REGISTER INTO OCTAL.

* * * * *

0 B M=0
1 B L=8
2 B N=L-1
3 B K=L-2
4 B J=0
5 B M=M+1
6 B J=J+1
7 S X[J]=(FOUR(J))
8 S D[M]=(X[J] AND ACR[L])
9 B M=M+1
10 S D[M]=(X[J] AND ACR[L])
11 B J:4 (EQ,NEQ) \(\rightarrow (12,5)\)
12 B L:5 (LE,GT) \(\rightarrow (15,13)\)
13 B L=L-3
14 B \(\rightarrow (2)\)
15 B N = 2
16 B K=1
17 B J=0
18 B M=M+1
19 B J=J+1
20 S D[M]=(FOUR(J))
21 B J:4 (EQ,NEQ) \(\rightarrow (22,18)\)
22 T DECODER(A,B)=D
23 T END

Figure 30. Subroutine DECODE.
<table>
<thead>
<tr>
<th>SUBROUTINE NAME AND STATUS TABLE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUHNSI(1,1) =</td>
</tr>
<tr>
<td>SUHNSI(1,2) = ACR</td>
</tr>
<tr>
<td>SUHNSI(1,3) =</td>
</tr>
<tr>
<td>SUHNSI(1,4) =</td>
</tr>
<tr>
<td>SUHNSI(1,5) =</td>
</tr>
<tr>
<td>SUHNSI(1,6) = T2 T3 T5 T6 T8 T9 T11 T12 T14 T15 T17 T18 T20 T21 T23 T24 T25 T26 T27 T28</td>
</tr>
<tr>
<td>SUHNSI(1,7) =</td>
</tr>
<tr>
<td>SUHNSI(1,8) =</td>
</tr>
</tbody>
</table>

Figure 31. Compiler Output from DECODE.
### Gates Used in Logic Unit

<table>
<thead>
<tr>
<th>Gate</th>
<th>Type</th>
<th>Input List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>^</td>
<td>¬ACR6 ¬ACR7</td>
</tr>
<tr>
<td>2</td>
<td>^</td>
<td>T1 ¬ACR8</td>
</tr>
<tr>
<td>3</td>
<td>^</td>
<td>T1 ACR8</td>
</tr>
<tr>
<td>4</td>
<td>^</td>
<td>¬ACR6 ACR7</td>
</tr>
<tr>
<td>5</td>
<td>^</td>
<td>T4 ¬ACR8</td>
</tr>
<tr>
<td>6</td>
<td>^</td>
<td>T4 ACR8</td>
</tr>
<tr>
<td>7</td>
<td>^</td>
<td>ACR6 ¬ACR7</td>
</tr>
<tr>
<td>8</td>
<td>^</td>
<td>T7 ¬ACR8</td>
</tr>
<tr>
<td>9</td>
<td>^</td>
<td>T7 ACR8</td>
</tr>
<tr>
<td>10</td>
<td>^</td>
<td>ACR6 ACR7</td>
</tr>
<tr>
<td>11</td>
<td>^</td>
<td>T10 ¬ACR8</td>
</tr>
<tr>
<td>12</td>
<td>^</td>
<td>T10 ACR8</td>
</tr>
<tr>
<td>13</td>
<td>^</td>
<td>¬ACR3 ¬ACR4</td>
</tr>
<tr>
<td>14</td>
<td>^</td>
<td>T13 ¬ACR5</td>
</tr>
<tr>
<td>15</td>
<td>^</td>
<td>T13 ACR5</td>
</tr>
<tr>
<td>16</td>
<td>^</td>
<td>¬ACR3 ACR4</td>
</tr>
<tr>
<td>17</td>
<td>^</td>
<td>T16 ¬ACR5</td>
</tr>
<tr>
<td>18</td>
<td>^</td>
<td>T16 ACR5</td>
</tr>
<tr>
<td>19</td>
<td>^</td>
<td>ACR3 ¬ACR4</td>
</tr>
<tr>
<td>20</td>
<td>^</td>
<td>T19 ¬ACR5</td>
</tr>
<tr>
<td>21</td>
<td>^</td>
<td>T19 ACR5</td>
</tr>
<tr>
<td>22</td>
<td>^</td>
<td>ACR3 ACR4</td>
</tr>
<tr>
<td>23</td>
<td>^</td>
<td>T22 ¬ACR5</td>
</tr>
<tr>
<td>24</td>
<td>^</td>
<td>T22 ACR5</td>
</tr>
<tr>
<td>25</td>
<td>^</td>
<td>¬ACR1 ¬ACR2</td>
</tr>
<tr>
<td>26</td>
<td>^</td>
<td>¬ACR1 ACR2</td>
</tr>
<tr>
<td>27</td>
<td>^</td>
<td>ACR1 ¬ACR2</td>
</tr>
<tr>
<td>28</td>
<td>^</td>
<td>ACR1 ACR2</td>
</tr>
</tbody>
</table>

Figure 31. (Continued).
Figure 32. Logic Produced by DECODE.
**Incrementor Unit**

Subroutine INC and function PROP together represent the AHPL software machinery required to describe a logic unit which can effect the incrementation by unity of the PCR register. This register has been declared as being eight bits in length previous to the occurrence of either the function or the subroutine. Rather than describe the programming involved in these subprograms we direct the reader to reference Figures 33, 34, 35, and 36 which contain listings of the programs and gate assignments and a block diagram of the logic developed, respectively.

**Adder Unit**

Subroutine ADD contains the rules for the development of an eight bit asynchronous ripple adder unit built around the ACR, MDR, and OVF registers. This unit was constructed during a different run than was used to construct the previous two units. Consequently it will be noted that the first gate allocated is numbered with one. All together 148 gates were assigned in the development of the adder unit. To graphically depict the configuration among all of these is clearly unnecessary. Instead only the first stage of the adder is shown in Figure 39. Figures 37 and 38 contain a listing of the program and of the gates assigned, respectively.
FUNCTION(PROP(KK,MAX))
L=KK

PROP IS A UTILITY FUNCTION THAT WILL BE CALLED FROM THE INCR SUBROUTINE. IT EFFECTS THE GENERATION OF THE LOGIC REQUIRED FOR THE PROPOGATION OF THE CARRY SIGNAL IN THE INCREMENTING UNIT BEING DESIGNED.

L IS THE CURRENT STAGE VALUE
MAX IS THE HIGHEST FLIP FLOP VALUE IN THE GROUP

Figure 33. Function PROP.
AHPL COMBINATIONAL LOGIC SUBROUTINE LISTING

SUBROUTINE(INCR(PCR))

0 B MXL=9
1 B J=9
2 S C[J]=1
3 B J:1 (EQ,NEQ) (12,4)
4 R K=J
5 H J=J-1
6 S SP[J]=(C[K]∧¬PCR[J])
7 S RP[J]=(¬C[K]∧PCR[J])

* *
* *
* CPU PULSE IS A CONTROL PULSE THAT WILL BE PROPAGATED FROM THE
* CONTROL UNIT. ITS MERE PRESENCE AS A RSH-BIT EFFECTS AN AHPL
* COMPILER REQUEST FOR SUCH A SIGNAL. IT WILL BE ALLOCATED
* WHEN THE CONTROL UNIT PROGRAM IS BEING INTERPRETED.
* *
8 S PCR[J]=(CPULSE∧SP[J])
9 S ¬PCR[J]=(CPULSE∧¬RP[J])

* *
* *
* THE CARRY SIGNAL (C[K]) FOLLOWS A PSEUDO-LOOK AHEAD STRUCTURE.
* NO ATTEMPT IS MADE TO LIMIT THE NUMBER OF INPUTS INTO THE
* GATE BEING ALLOCATED FOR THIS SIGNAL.
* *
* THIS COULD BE WRITTEN SO THAT THE LOOK-AHEAD FEATURE WAS
* STRUCTURED IN TERMS OF GROUPS OF FOUR FLIP-FLOP UNITS.
* *
10 S C[J]=(PROP(J,MXL))
11 B (3)
12 T END

Figure 34. Subroutine INCR.
1. SUBROUTINE NAME AND STATUS TABLE.

| SUBRNST (2,1) | = | T31 |
| SUBRNST (2,2) | = | PCR |
| SUBRNST (2,3) | = |
| SUBRNST (2,4) | = |
| SUBRNST (2,5) | = |
| SUBRNST (2,6) | = |
| SUBRNST (2,7) | = |
| SUBRNST (2,8) | = |

2. REGISTER INPUTS.

<table>
<thead>
<tr>
<th>OVF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVF SET = RESET =</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACR:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACR SET = RESET =</td>
</tr>
<tr>
<td>ACR SET = RESET =</td>
</tr>
<tr>
<td>ACR SET = RESET =</td>
</tr>
<tr>
<td>ACR SET = RESET =</td>
</tr>
<tr>
<td>ACR SET = RESET =</td>
</tr>
<tr>
<td>ACR SET = RESET =</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MDR:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDR SET = RESET =</td>
</tr>
<tr>
<td>MDR SET = RESET =</td>
</tr>
<tr>
<td>MDR SET = RESET =</td>
</tr>
<tr>
<td>MDR SET = RESET =</td>
</tr>
<tr>
<td>MDR SET = RESET =</td>
</tr>
<tr>
<td>MDR SET = RESET =</td>
</tr>
<tr>
<td>MDR SET = RESET =</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCR:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCR SET = T67  RESET = T68</td>
</tr>
<tr>
<td>PCR SET = T62  RESET = T63</td>
</tr>
<tr>
<td>PCR SET = T57  RESET = T58</td>
</tr>
<tr>
<td>PCR SET = T52  RESET = T53</td>
</tr>
<tr>
<td>PCR SET = T47  RESET = T48</td>
</tr>
<tr>
<td>PCR SET = T42  RESET = T43</td>
</tr>
<tr>
<td>PCR SET = T37  RESET = T38</td>
</tr>
<tr>
<td>PCR SET = T32  RESET = T33</td>
</tr>
</tbody>
</table>

Figure 35. Compiler Output from INCR.
<table>
<thead>
<tr>
<th>GATE</th>
<th>TYPE</th>
<th>INPUT LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>^</td>
<td>ONE -PCR8</td>
</tr>
<tr>
<td>30</td>
<td>^</td>
<td>ONE PCR8</td>
</tr>
<tr>
<td>31</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>^</td>
<td>T31 T29</td>
</tr>
<tr>
<td>33</td>
<td>^</td>
<td>T31 T30</td>
</tr>
<tr>
<td>34</td>
<td>^</td>
<td>ONE PCR6</td>
</tr>
<tr>
<td>35</td>
<td>^</td>
<td>T34 ~PCR7</td>
</tr>
<tr>
<td>36</td>
<td>^</td>
<td>T34 PCR7</td>
</tr>
<tr>
<td>37</td>
<td>^</td>
<td>T31 T35</td>
</tr>
<tr>
<td>38</td>
<td>^</td>
<td>T31 T36</td>
</tr>
<tr>
<td>39</td>
<td>^</td>
<td>ONE PCR7 PCR8</td>
</tr>
<tr>
<td>40</td>
<td>^</td>
<td>T39 ~PCR6</td>
</tr>
<tr>
<td>41</td>
<td>^</td>
<td>T39 PCR6</td>
</tr>
<tr>
<td>42</td>
<td>^</td>
<td>T31 T40</td>
</tr>
<tr>
<td>43</td>
<td>^</td>
<td>T31 T41</td>
</tr>
<tr>
<td>44</td>
<td>^</td>
<td>ONE PCR6 PCR7 PCR8</td>
</tr>
<tr>
<td>45</td>
<td>^</td>
<td>T44 ~PCR5</td>
</tr>
<tr>
<td>46</td>
<td>^</td>
<td>T44 PCR5</td>
</tr>
<tr>
<td>47</td>
<td>^</td>
<td>T31 T45</td>
</tr>
<tr>
<td>48</td>
<td>^</td>
<td>T31 T46</td>
</tr>
<tr>
<td>49</td>
<td>^</td>
<td>ONE PCR5 PCR6 PCR7 PCR8</td>
</tr>
<tr>
<td>50</td>
<td>^</td>
<td>T49 ~PCR4</td>
</tr>
<tr>
<td>51</td>
<td>^</td>
<td>T49 PCR4</td>
</tr>
<tr>
<td>52</td>
<td>^</td>
<td>T31 T50</td>
</tr>
<tr>
<td>53</td>
<td>^</td>
<td>T31 T51</td>
</tr>
<tr>
<td>54</td>
<td>^</td>
<td>ONE PCR4 PCR5 PCR6 PCR7 PCR8</td>
</tr>
<tr>
<td>55</td>
<td>^</td>
<td>T54 ~PCR3</td>
</tr>
<tr>
<td>56</td>
<td>^</td>
<td>T54 PCR3</td>
</tr>
<tr>
<td>57</td>
<td>^</td>
<td>T31 T55</td>
</tr>
<tr>
<td>58</td>
<td>^</td>
<td>T31 T56</td>
</tr>
<tr>
<td>59</td>
<td>^</td>
<td>ONE PCR3 PCR4 PCR5 PCR6 PCR7 PCR8</td>
</tr>
<tr>
<td>60</td>
<td>^</td>
<td>T59 ~PCR2</td>
</tr>
<tr>
<td>61</td>
<td>^</td>
<td>T59 PCR2</td>
</tr>
<tr>
<td>62</td>
<td>^</td>
<td>T31 T60</td>
</tr>
<tr>
<td>63</td>
<td>^</td>
<td>T31 T61</td>
</tr>
<tr>
<td>64</td>
<td>^</td>
<td>ONE PCR2 PCR3 PCR4 PCR5 PCR6 PCR7 PCR8</td>
</tr>
<tr>
<td>65</td>
<td>^</td>
<td>T64 ~PCR1</td>
</tr>
</tbody>
</table>

Figure 35. (Continued).
3. GATES USED IN LOGIC UNIT.

<table>
<thead>
<tr>
<th>GATE</th>
<th>TYPE</th>
<th>INPUT LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>∧</td>
<td>T64, PCR1</td>
</tr>
<tr>
<td>67</td>
<td>∧</td>
<td>T31, T65</td>
</tr>
<tr>
<td>68</td>
<td>∧</td>
<td>T31, T66</td>
</tr>
<tr>
<td>69</td>
<td>∧</td>
<td>ONE, PCR1, PCR2, PCR3, PCR4, PCR5, PCR6, PCR7, PCR8</td>
</tr>
</tbody>
</table>

Figure 35. (Continued).
Figure 36. Logic Produced by INCR.
THE FOLLOWING INITIAL LOGIC DESIGN PROGRAM DESCRIBES A SIMPLE Ripple ADDER WITH A CARRY-COMPLETE SIGNAL. THEORETICALLY, THIS ADDER WILL OPERATE IN AN ASYNCHRONOUS Manner.

A HIGH LOGIC LEVEL WILL BE RETURNED TO THE CONTROL UNIT VIA A ONE-SHOT WHEN THE ADDITION PROCESS IS COMPLETE. THIS FEATURE IS IMPLEMENTED IN THE PROGRAM BY REFERENCING THE COMPILER VARIABLE #PULSE. A CONTROL PULSE IS REQUESTED FROM THE THE CONTROL UNIT BY REFERENCING THE VARIABLE #Pulse, THIS PROGRAM ALSO USES THE OEUING FEATURE WHICH WAS IMPLEMENTED AS AN AID IN FOLLOWING THE HARDWARE ASSIGNMENT SEQUENCE OF THE COMPILER.

Figure 37. Subroutine ADD.
### 1. SUBROUTINE NAME AND STATUS TABLE

| SUBRNS1 (1,1) | = T197 |
| SUBRNS1 (1,2) | = OVF  |
| SUBRNS1 (1,3) | = ACR  |
| SUBRNS1 (1,4) | = ADR  |
| SUBRNS1 (1,5) |  |
| SUBRNS1 (1,6) | = T96  |
| SUBRNS1 (1,7) | = T93 T91 T89 T87 T45 T33 T21 T9 |
| SUBRNS1 (1,8) | = OS.1 |

Figure 38. Compiler Output from ADD.
### Gates and Logic Units

<table>
<thead>
<tr>
<th>Gate</th>
<th>Type</th>
<th>Input List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>ACR8 MDR8</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>ACR8 MDR8</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>ACR8 MDR8</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>ACR8 MDR8</td>
</tr>
<tr>
<td>5</td>
<td>v</td>
<td>T1 T2</td>
</tr>
<tr>
<td>6</td>
<td>v</td>
<td>ZERO15</td>
</tr>
<tr>
<td>7</td>
<td>v</td>
<td>T3 T4</td>
</tr>
<tr>
<td>8</td>
<td>v</td>
<td>ONE T7</td>
</tr>
<tr>
<td>9</td>
<td>v</td>
<td>T6 T8</td>
</tr>
<tr>
<td>10</td>
<td>v</td>
<td>T3 T4</td>
</tr>
<tr>
<td>11</td>
<td>v</td>
<td>ZERO110</td>
</tr>
<tr>
<td>12</td>
<td>v</td>
<td>T1 T11</td>
</tr>
<tr>
<td>13</td>
<td>v</td>
<td>ACR7 MDR7</td>
</tr>
<tr>
<td>14</td>
<td>v</td>
<td>ACP7 MDR7</td>
</tr>
<tr>
<td>15</td>
<td>v</td>
<td>ACR7 MDR7</td>
</tr>
<tr>
<td>16</td>
<td>v</td>
<td>ACR7 MDR7</td>
</tr>
<tr>
<td>17</td>
<td>v</td>
<td>T13 T14</td>
</tr>
<tr>
<td>18</td>
<td>v</td>
<td>T12 T17</td>
</tr>
<tr>
<td>19</td>
<td>v</td>
<td>T15 T16</td>
</tr>
<tr>
<td>20</td>
<td>v</td>
<td>F12 T119</td>
</tr>
<tr>
<td>21</td>
<td>v</td>
<td>T18 T20</td>
</tr>
<tr>
<td>22</td>
<td>v</td>
<td>T15 T16</td>
</tr>
<tr>
<td>23</td>
<td>v</td>
<td>T12 T22</td>
</tr>
<tr>
<td>24</td>
<td>v</td>
<td>T13 T23</td>
</tr>
<tr>
<td>25</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>26</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>27</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>28</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>29</td>
<td>v</td>
<td>T25 T26</td>
</tr>
<tr>
<td>30</td>
<td>v</td>
<td>T24 T29</td>
</tr>
<tr>
<td>31</td>
<td>v</td>
<td>T27 T28</td>
</tr>
<tr>
<td>32</td>
<td>v</td>
<td>F24 T31</td>
</tr>
<tr>
<td>33</td>
<td>v</td>
<td>T30 T32</td>
</tr>
<tr>
<td>34</td>
<td>v</td>
<td>T27 T28</td>
</tr>
<tr>
<td>35</td>
<td>v</td>
<td>T24 T34</td>
</tr>
<tr>
<td>36</td>
<td>v</td>
<td>T25 T35</td>
</tr>
<tr>
<td>37</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>38</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>39</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>40</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>41</td>
<td>v</td>
<td>T37 T39</td>
</tr>
<tr>
<td>42</td>
<td>v</td>
<td>T36 T41</td>
</tr>
<tr>
<td>43</td>
<td>v</td>
<td>T39 T40</td>
</tr>
<tr>
<td>44</td>
<td>v</td>
<td>F36 T43</td>
</tr>
<tr>
<td>45</td>
<td>v</td>
<td>T47 T44</td>
</tr>
<tr>
<td>46</td>
<td>v</td>
<td>T39 T40</td>
</tr>
<tr>
<td>47</td>
<td>v</td>
<td>T36 T46</td>
</tr>
<tr>
<td>48</td>
<td>v</td>
<td>T37 T47</td>
</tr>
<tr>
<td>49</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>50</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
<tr>
<td>51</td>
<td>v</td>
<td>ACR6 MDR6</td>
</tr>
</tbody>
</table>

Figure 38. Compiler Output from ADD (Continued).
<table>
<thead>
<tr>
<th>GATE</th>
<th>TYPE</th>
<th>INPUT LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
<td>A</td>
<td>ACM4 MDR4</td>
</tr>
<tr>
<td>93</td>
<td>V</td>
<td>T49 T50</td>
</tr>
<tr>
<td>94</td>
<td>A</td>
<td>T46 T53</td>
</tr>
<tr>
<td>55</td>
<td>V</td>
<td>T51 T52</td>
</tr>
<tr>
<td>56</td>
<td>A</td>
<td>F48 T55</td>
</tr>
<tr>
<td>57</td>
<td>V</td>
<td>T54 T56</td>
</tr>
<tr>
<td>58</td>
<td>V</td>
<td>T51 T52</td>
</tr>
<tr>
<td>59</td>
<td>A</td>
<td>T48 T54</td>
</tr>
<tr>
<td>60</td>
<td>V</td>
<td>T49 T59</td>
</tr>
<tr>
<td>61</td>
<td>A</td>
<td>ACM3 MDR3</td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>ACM3 MDR3</td>
</tr>
<tr>
<td>63</td>
<td>A</td>
<td>ACM3 MDR3</td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>ACM3 MDR3</td>
</tr>
<tr>
<td>65</td>
<td>v</td>
<td>T61 T62</td>
</tr>
<tr>
<td>66</td>
<td>A</td>
<td>T60 T65</td>
</tr>
<tr>
<td>67</td>
<td>V</td>
<td>T63 T64</td>
</tr>
<tr>
<td>68</td>
<td>A</td>
<td>F66 T67</td>
</tr>
<tr>
<td>69</td>
<td>V</td>
<td>T66 T68</td>
</tr>
<tr>
<td>70</td>
<td>V</td>
<td>T63 T64</td>
</tr>
<tr>
<td>71</td>
<td>A</td>
<td>T60 T70</td>
</tr>
<tr>
<td>72</td>
<td>V</td>
<td>T61 T71</td>
</tr>
<tr>
<td>73</td>
<td>A</td>
<td>ACM2 MDR2</td>
</tr>
<tr>
<td>74</td>
<td>A</td>
<td>ACM2 MDR2</td>
</tr>
<tr>
<td>75</td>
<td>A</td>
<td>ACM2 MDR2</td>
</tr>
<tr>
<td>76</td>
<td>A</td>
<td>ACM2 MDR2</td>
</tr>
<tr>
<td>77</td>
<td>V</td>
<td>T73 T74</td>
</tr>
<tr>
<td>78</td>
<td>V</td>
<td>T72 T77</td>
</tr>
<tr>
<td>79</td>
<td>v</td>
<td>T75 T76</td>
</tr>
<tr>
<td>80</td>
<td>A</td>
<td>F72 T79</td>
</tr>
<tr>
<td>81</td>
<td>V</td>
<td>T78 T80</td>
</tr>
<tr>
<td>82</td>
<td>v</td>
<td>T75 T76</td>
</tr>
<tr>
<td>83</td>
<td>A</td>
<td>T72 T82</td>
</tr>
<tr>
<td>84</td>
<td>v</td>
<td>T73 T83</td>
</tr>
<tr>
<td>85</td>
<td>A</td>
<td>ACM1 MDR1</td>
</tr>
<tr>
<td>86</td>
<td>A</td>
<td>ACM1 MDR1</td>
</tr>
<tr>
<td>87</td>
<td>A</td>
<td>ACM1 MDR1</td>
</tr>
<tr>
<td>88</td>
<td>A</td>
<td>ACM1 MDR1</td>
</tr>
<tr>
<td>89</td>
<td>V</td>
<td>T85 T86</td>
</tr>
<tr>
<td>90</td>
<td>A</td>
<td>T84 T84</td>
</tr>
<tr>
<td>91</td>
<td>V</td>
<td>TK7 T88</td>
</tr>
<tr>
<td>92</td>
<td>A</td>
<td>F84 T91</td>
</tr>
<tr>
<td>93</td>
<td>V</td>
<td>T90 T92</td>
</tr>
<tr>
<td>94</td>
<td>v</td>
<td>T87 T88</td>
</tr>
<tr>
<td>95</td>
<td>A</td>
<td>T84 T94</td>
</tr>
<tr>
<td>96</td>
<td>v</td>
<td>T85 T95</td>
</tr>
<tr>
<td>97</td>
<td>A</td>
<td>F12 T14</td>
</tr>
<tr>
<td>98</td>
<td>V</td>
<td>T77 T84</td>
</tr>
<tr>
<td>99</td>
<td>A</td>
<td>T96 T101 T102 T103 T104</td>
</tr>
<tr>
<td>100</td>
<td>A</td>
<td>T26 T30 T35 T37 T44</td>
</tr>
<tr>
<td>101</td>
<td>V</td>
<td>T39 T40</td>
</tr>
<tr>
<td>102</td>
<td>V</td>
<td>T51 T52</td>
</tr>
</tbody>
</table>

Figure 38. Compiler Output from ADD (Continued).
3. GATES USED IN LOGIC UNIT.

<table>
<thead>
<tr>
<th>GATE TYPE</th>
<th>INPUT LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>T63 T64</td>
</tr>
<tr>
<td>104</td>
<td>T75 T76</td>
</tr>
<tr>
<td>105</td>
<td>T99 T100</td>
</tr>
<tr>
<td>106</td>
<td>T97 T105</td>
</tr>
<tr>
<td>107</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>T107 T106</td>
</tr>
<tr>
<td>109</td>
<td>F166 T167</td>
</tr>
<tr>
<td>110</td>
<td>T168 T169</td>
</tr>
<tr>
<td>111</td>
<td>F24 T26</td>
</tr>
<tr>
<td>112</td>
<td>T39 T40</td>
</tr>
<tr>
<td>113</td>
<td>T112 T115</td>
</tr>
<tr>
<td>114</td>
<td>T138 T150</td>
</tr>
<tr>
<td>115</td>
<td>T152</td>
</tr>
<tr>
<td>116</td>
<td>T63 T64</td>
</tr>
<tr>
<td>117</td>
<td>T75 T76</td>
</tr>
<tr>
<td>118</td>
<td>T113 T114</td>
</tr>
<tr>
<td>119</td>
<td>T111 T110</td>
</tr>
<tr>
<td>120</td>
<td>T109 T119</td>
</tr>
<tr>
<td>121</td>
<td>F119 T109</td>
</tr>
<tr>
<td>122</td>
<td>F36 T38</td>
</tr>
<tr>
<td>123</td>
<td>T51 T52</td>
</tr>
<tr>
<td>124</td>
<td>T123 T126</td>
</tr>
<tr>
<td>125</td>
<td>T50 T62 T74</td>
</tr>
<tr>
<td>126</td>
<td>T63 T64</td>
</tr>
<tr>
<td>127</td>
<td>T75 T76</td>
</tr>
<tr>
<td>128</td>
<td>T124 T125</td>
</tr>
<tr>
<td>129</td>
<td>T122 T128</td>
</tr>
<tr>
<td>130</td>
<td>T121 T129</td>
</tr>
<tr>
<td>131</td>
<td>F129 T121</td>
</tr>
<tr>
<td>132</td>
<td>F48 T50</td>
</tr>
<tr>
<td>133</td>
<td>T63 T64</td>
</tr>
<tr>
<td>134</td>
<td>T133 T136</td>
</tr>
<tr>
<td>135</td>
<td>T62 T74</td>
</tr>
<tr>
<td>136</td>
<td>T75 T76</td>
</tr>
<tr>
<td>137</td>
<td>T134 T135</td>
</tr>
<tr>
<td>138</td>
<td>T132 T137</td>
</tr>
<tr>
<td>139</td>
<td>T131 T133</td>
</tr>
<tr>
<td>140</td>
<td>F138 T131</td>
</tr>
<tr>
<td>141</td>
<td>F63 T62</td>
</tr>
<tr>
<td>142</td>
<td>T75 T76</td>
</tr>
<tr>
<td>143</td>
<td>T142</td>
</tr>
<tr>
<td>144</td>
<td>T74</td>
</tr>
<tr>
<td>145</td>
<td>T143 T144</td>
</tr>
<tr>
<td>146</td>
<td>T141 T145</td>
</tr>
<tr>
<td>147</td>
<td>T140 T146</td>
</tr>
<tr>
<td>148</td>
<td>F146 T140</td>
</tr>
</tbody>
</table>

Figure 38. Compiler Output from ADD (Continued).
A. LAST TWO STAGES OF ADDER

Figure 39. Logic Produced by ADD.
B. LAST TWO STAGES OF "COMPLETE SIGNAL" LOGIC

Figure 39. (Continued).
CHAPTER 6

CONCLUSION

We have discussed in reasonable detail the establishment of the AHPL Combinational Logic Language and the Combinational Logic compiler. Three sample design programs were submitted and the results of their compilation discussed. It is believed that so far as combinational logic is concerned the language and software developed constitute a viable if not immediately useful tool for the initial logic design of the electronic digital computer. It is felt that the AHPL Combinational Logic Language possesses sufficient capacity for not only describing any logic unit but for describing it in a way that requires the minimum allocation of hardware.

At the present time the combinational logic subroutine compiler is being interfaced with the control unit compiler preparatory to the establishment of a total initial logic design system. Its success in this environment is not yet known but initial reports have been favorable. Certain changes in the manner of hardware allocation are being effected. The details concerning the total
system and the hardware allocation changes will be reported in the dissertation by Mr. Michael Gentry (1971b).

It is believed that the combinational logic compiler could be definitely improved by developing the hardware allocation section in a language other than SNOBOL 4. Though the task of retranslating the intermediate code lists into a macro-assembly language would be difficult, it is by no means impossible. It is thought that the form assumed by the intermediate code structure could be changed to facilitate its use by other compilers or interpreters written in some other language.

The time required for hardware allocation by SNOBOL 4 is, of course, a vast improvement over that which was required for the hand preparation of the architectural documents. It has been estimated that approximately 200 milliseconds are required for each gate allocation by the compiler. The compilation of a subroutine as complicated as that which developed the adder unit described in the last chapter required about 45 seconds of central processing unit time on the CDC 6400 computer. A problem arises, however, when we attempt to compile several subroutines in sequence. As the computer allocates more and more memory for the storage of names and lists, less memory remains for the large working storage areas required by the SNOBOL 4 operations. When this state is approached, the SNOBOL 4
compiler spends more time in establishing these working storage areas than was the case at the beginning. The time required for the allocation of a single gate can then rise to several seconds.

It should be pointed out that there exist a number of strategies that, if implemented, would downgrade the seriousness of the memory problem. None of these were actively investigated during the current study for reasons of economy and overall objective. One of these, probably the strongest, has been discussed. Others involve the dumping of all of the information to be stored onto a magnetic tape and starting fresh with each new subroutine. If the compiler were to find use, say in an industrial environment, one of these methods would be required,
APPENDIX

THE AHPL COMBINATIONAL LOGIC COMPILER

A computer listing of the Combinational Logic Sub-routine Compiler is presented in the following pages. For implementation on the Control Data 6400 computer installation at the University of Arizona Computer Center the following sequence of control cards are required:

JOB CARD (with requests for 140 K memory and 200 seconds CPU time)
ATTACH, SNOBOL, SNOBOL.
SNOBOL.

COMPILER PROGRAM
AHPL DESIGN PROGRAM

122
**SNOBOL4 (VERSION 2.0)**

---

**Bell Telephone Laboratories, Incorporated**

**CDC Version by Purdue University and IU-A.**

*AHPL COMBINATIONAL LOGIC SUBROUTINE COMPILER*

**VERSION 3.0**

**MAXLENGTH = 7700**

**ADUMP = 1**

---

**DATA SECTION**

```
DEFINE(*GEN(X,Y)*)
DEFINE(*REDUCE(B,NOP,RED,OP,LANG,NA,NEG,1,NEG,2)*)
DEFINE(*DEBUG(DBG1,DBG2,DBG3)*)
DEFINE(*READ(TXT,NXTST)*)
DEFINE(*FACTVAL(INDEX)*)
DEFINE(*FUN(XRAY)OP1,OP2,OPA,LIST,FUN#)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
DEFINE(*TTYPE(T)*)
```

**OUTPUT(*ERROR*,2,(*,1X,130A1)*)**

**OUTPUT(*TITLE*,2,(*,1H1,25X,70A1,/)*)**

**OUTPUT(*HEAD*,2,(*,13X,80A1,/)*)**
ELEMENTARY DATA

LETTERS = ABCDEFGHIJKLMNOPQRSTUVWXYZ
DIGITS = 0123456789
ALPHANUM = LETTERS DIGITS
BINOP = ^^^^ v ^^^^ 
BLANKS = SPAN( ^ ^ )
GTE = #^#
INTEGER = ANY(DIGITS) OPTION(SPA%N(DIGITS))
IDENTIFIER = ANY(LETTERS) OPTION(SPA%N(ALPHANUM))
GATESIG = OPTION(BLANKS) ANY(TF) TRUTHCODE
INTEGER , POSTN
UNARYOP = v UNARYOP NUMBER 
FACTOR = IDENTIFIER v INTEGER
M^BIT = UNARYOP IDENTIFIER [+ FACTOR [+ v UNARYOP IDENTIFIER ] + ] v GATESIG
FACTOR = FACTOR NUMBER [+]
LIST^BIT = IDENTIFIER [+ INTEGER [+ ] + ] v IDENTIFIER
ARG^LIST = LIST^BIT ARBN0( + + LIST^BIT)
SUBROUTINE = IDENTIFIER + SRNAME [+]
ARG^LIST = + + LIST^BIT
FUNCTION = IDENTIFIER , FUNCTION(+ IDENTIFIER + FUNCTION( + ) +)
ARG^LIST = + + LIST^BIT
SRPREDUCE = + + UNARYOP + UNARYOP + BINOP + 8 + UNARYOP [+]+ [+](
#PREFIX # #SUFFIX ) + RED.OP [+ FACTOR + NUMBER [+ ] #)
UNARYOP : NEG^2 IDENTIFIER + NA^1 [+]
SREDUCE = + + UNARYOP + NEG^1 BI^OP + 8 + UNARYOP [+]+ [+]
NEG^2 IDENTIFIER + NA^1 [+]
PARTSIGNAI = UNARYOP + NEG^1 IDENTIFIER + NA^1
SIGN = OPTION(BLANKS) [# INTEGER + TURN0 #]
SGOTO = #^[INTEGER + S^1 ][INTEGER + S^2 ]#
ENNER = #^[EQUAL# NA^1 ]#
LEGT = #^[LEG# ]#
LTGE = #^[LT#GE]#
DECLAR = ^[VECTOR + INTEGER + LENGTH
F^ARG = UNARYOP IDENTIFIER [+ FACTOR + ] v UNARYOP
IDENTIFIER
F^ARG^LIST = F^ARG ARBNO( + + F^ARG)
STRIPOFF = (INTEGER v NULL) + BLANKS ANY( #STBC# ) + KEY
BLANKS
FINDREG = POS(0) REGISTER RPOS(0)
FINDM = POS(0) +DUHVEC RPOS(0)
FINDFN = POS(0) +FUHVEC RPOS(0)
FNUVEC = FAIL
REG.CT = 0
CAL.CNR = '0
GATCT = 0
OS.CT = 0
FTAB = ARRAY(4*8)
LEVCT = 0
FN.CT = 0
SR.CT = 0
SUBRINST = ARRAY(15*8)
REGISTER = FAIL
SRVEC = FAIL
EOF*2 =
EOF =
FFLAG =
COMMENT = •
CONTINUE = •
^ANCHOR = 1

ORIGIN SECTION

TITLE = ampl combinational logic subroutine listing
READI LINE = TRIM (INPUT) ;
LINE COMMENT = {ORIGIN}
OUTPUT = " " LINE ;(READI)
ORIGIN READ() ;
IMAGE DECLAR ;S(DECLAR)
IMAGE SRDEF ;S(SUBR1)
IMAGE FNCDEF ;S(FNCT)F(SYNERR)

REGISTER DECLARATION SECTION

DECLR REG.CT = REG.CT + 1
\$\#RN\# REG.CT = VECTOR
\$VECTOR = REG.CT
REGISTER = REGISTER \@ VECTOR
\$\#RL\# REG.CT = LENGTH ;(ORIGIN)
**AMPL SUBROUTINE DEFINITION SECTION**

```
** SUBR1 **
IMAGE SRDEF :F(FNCT)
EOF,2 =
FIRST = GATCT
GATEVEC = FAIL
DUMVEC = FAIL
INDEXVEC = FAIL
OUTVEC =
SR,CT = LT(SR,CT,15) SR,CT + 1 :F(SMCTERR)
$SRNAME = SR,CT
ALPHAPAT = SRNAME
SRVEC = SRVEC v SRNAME
CPULSE =
WORD = SRNAME 5 #
IWORD = WORD
NEWCODE =
ARG,CT = 1

******
BEGIN EXAMINATION OF THE SUBROUTINE DEFINITION ARGUMENT LIST
EXAMINE EACH FOR PRESENCE OF SUBSCRIPT AND TRY TO
MATCH THE ALPHAMERIC PART TO THE SYMBOLOGY OF DECLARED
REGISTERS. IF SUBSCRIPT APPEARS WITH LABEL THAT MATCHES
SOME DECLARED REGISTER, IGNORE SUBSCRIPT. OTHERWISE
CONSTRUCT AN OR-8US FOR THIS ARGUMENT. IF THERE IS NO
SUBSCRIPT AND THE ARGUMENT DOES NOT MATCH THE LIST OF
DECLARED REGISTERS, RETURN TO ERROR ROUTINE AND TERMINATE.

******
** SUBR2 **
IDENT(LIST) :S(SUBR6)
LIST BREAK(#+,+) ARGUE LEN(1) 3 :F(SUBR5)

** SUBR3 **
ARG,CT = LT(ARG,CT,5) ARG,CT + 1 :F(ACTERR)
ARGUE IDENTIFIER , NAME, 4 INTEGER , LENGTH 4 ) :F(SUBR4)
NAME, FINDREG :S(SUBR4,1)
DUMVEC = DUMVEC v NAME
$NAME, = ARG,CT
$( #DA,# ARG,CT) = LENGTH
SUBRNS(TSR,CT,ARG,CT) = GEN($V,*)
```

* I26 *
THE OR-BUS CONSTRUCTION FOR AN INDIRECT ACCESS 
SUBROUTINE ARGUMENT BEGINS HERE.
THE FIRST GATE NUMBER IS PLACED IN SUBRNST AS A BASE
FOR THE CALCULATION OF AN INPUT GATE NUMBER OF
AN ELEMENT OF THIS INPUT VECTOR.

COUNT = 1
BLOOP
COUNT = LT(COUNT,LENGTH) COUNT + 1 IF(SUBR2)
GATCT = GATCT + 1
S (#GATCT) = #V# :I(BLOOP)
SUBR4
ARGUE FINDREG :F(ARGERR)
SUBRNST[SR,CT,ARG,CT] = ARGUE :I(SUBR2)
SUBR4,1
SUBRNST[SR,CT,ARG,CT] = NAME, :I(SUBR2)
SUBR5
ARGUE = LIST
LIST = NULL :I(SUBR3)
SUBR6
READ() IMAGE STRIPOFF =
LB, L = ALPHAPAT LB, L
IDENT(KEY, #C#) :I(SR,60)
IDENT(KEY, #S#) :I(SUBR7)
IDENT(KEY, #B#) :I(SUBR8)
IDENT(KEY, #T#) :F(KEYERR)I(SUBR9)
SUBR7
STYPE() :F(RECOVER)I(SUBR6)
SUBR8
STYPE() :F(RECOVER)I(SUBR6)
SUBR9
STYPE() :F(RECOVER)
IDENT(EOF#2) :I(SUBR6)
NEWCODE = CODE(IWORD) :F(COMPErr)
I[NEWCODE]
SR,60
WORD = SRNAME LB, L # IMAGE # ;
IWORD = IWORD WORD :I(SUBR6)
COMBINE OUTPUT ARRAYS INTO SINGLE LISTS.

SCALL5
K = 0
GROUP2 = GROUP1 =

SUBK32
K = LT(K,64) K + 1 IF(SUBR33)
GROUP1 = GROUP1 $(1 \# \# K)
GROUP2 = GROUP2 $(2 \# \# K)
$(1 \# \# K) =
$(2 \# \# K) = IF(SUBR32)

SUBR33
SUBRNST[SRTCT,6] = GROUP1
SUBRNST[SRTCT,7] = GROUP2

BEGIN LISTING THE HARDWARE ASSIGNED.

TITLE = SRNAME \# WAS COMPILED AS THE FOLLOWING LOGIC UNIT.
TITLE = \#LOGIC PRODUCED FROM \# SRNAME \#
HEAD = \#1. SUBROUTINE NAME AND STATUS TABLE.
K = 0

SUBR34
K = LT(K,8) K + 1 IF(SUBR35)
OUTPUT = \#SUBRNST[\# SRTCT \#\# \# \#] = \# SUBRNST[SRTCT,K]

SUBR35
TITLE = \#LOGIC PRODUCED FROM \# SRNAME \#
HEAD = \#2. REGISTER INPUTS.
K = 0

SUBR36
K = LT(K,REG.CT) K + 1 IF(SUBR50).
J = 0
OUTPUT = \# \# $(\#RN\# K) \#

SUBR37
J = LT(J,$(\#RL\# K)) J + 1 IF(SUBR36)
OUTPUT = \# \# $(\#RN\# K) \# SET $ $(\#SET\# K \# J)

\# RESET $ $(\#RESET\# K \# J) IF(SUBR37)

SUBR50
J = FIRST

SUBR12
TITLE = \#LOGIC PRODUCED FROM \# SRNAME \#
HEAD = \#3. GATES USED IN LOGIC UNIT.
MMM = 0

HEAD = \#GATE TYPE INPUT LIST

SR20
J = LT(J,GATCt) J + 1 IF(SUBR21)
HEAD = \# \# \# J \# \# \# $(\#GT\# J) \# \# \# $(\#G\# J)

MMM = MMM + 1

GT(MMM,36) \#S(SUBR40)F(SR20)
SUBR21  OUTPUT =
OUTPUT =
OUTPUT =
HEAD = 'INPUT LIST OF LATEST PULSE GENERATOR:
$(↑OS.↑ OS.CT)
TITLE = ↑AMPL COMBINATIONAL LOGIC SUBROUTINE LISTING:
↑
↑(ORIGIN)

168
169
170
171
171
172
172
**AHPL FUNCTION DEFINITION SECTION**

```
FNCT IMAGE FNCDEF :F(SYNNERR) 173
  FN_CT = FN_CT + 1 174
  FUNVEC = FUNVEC ∨ FNAME 175
  $FNAME = FN_CT 176
  FN_CT2 = #FUNCTION# FN_CT 177
  ALPHAPAT = FNAME 178
  WORD = FNAME # ;# 179
  IWORD = WORD 180
  ARGVEC = FAIL 181
  ARG_CT = 0 182
  FNC.1 IDENT(LIST) :S(FNC.4) 183
  LIST BREAK(,+) ARGUE LEN(1) = :F(FNC.3) 184
  FNC.2 ARG_CT = LT(ARG_CT,8) ARG_CT + 1 :F(KEYERR) 185
  ARGVEC = ARGVEC ∨ ARGUE 186
  $ARGUE = ARG_CT : (FNC.1) 187
  FNC.3 ARGUE = LIST 188
  LIST = NULL :(FNC.2) 189
  FNC.4 READ() IMAGE STRIPOFF 190
  LB_1 = ALPHAPAT LB_L 191
  FFLAG = 1 192
  IDENT(KEY,+) :S(FNC.5) 193
  IDENT(KEY,++) :S(FNC.7) 194
  IDENT(KEY,+) :F(KEYERR) 195
  TYPE() :S(FNC.4)F(RECOVER) 196
  FNC.5 STYPE() :S(FNC.4)F(RECOVER) 197
  FNC.6 IMAGE * = :S(FNC.7) 198
  IMAGE END# :S(FNC.9) 199
  TYPF() :FNC.4 200
  FNC.9 FFLAG = 201
  TITLE = "AHPL COMBINATIONAL LOGIC SUBROUTINE LISTING" 202
  $FN_CT2 = IWORD 203
  IWORD = :ORIGIN 204
```

---

130
STYPE: STYPE ANALYSES THE CURRENT SOURCE STATEMENT AND GENERATES APPROPRIATE INSTRUCTIONS TO EFFECT THE IMPLIED HARDWARE ASSIGNMENT.

*******************************************************************************

STYPE TP, T = 0
IMAGE BREAK(=>+), L.SIDE LEN(1) = IF(SYNEHR)

ST.A
IMAGE # = #S(IIST+1)
WORD = LH.L # DEST* = # PASIG(L.SIDE) # #
IWORD = lWORD WORD
IMAGE POS(0) UNARYOP # NEG1 (# IDENTIFIER # FNAM3 (#)
+ FARG. LIST ( LIST #)) RPOS(0) IF(ST.B)
FNAM3 FINDFN IF(SYNEHR)
FNCALL()
NEG1 # = #COMP(SRSIG(S($FR(# $FNAM3 #))) # :S(ST.A))
NEG1 = #SRSIG(S($FR(# $FNAM3 #)) #)

ST.A
WORD = LB.L +,# # CONNECT(# PASIG(L.SIDE) #,# NEG1 #,)#
+ # LEVCT = LEVCT - 1 ;#
+ # FCODE = IF(ST.AA)
++ IWORD = IWORD WORD (RETURN)
ST.B
IMAGE POS(0) (# S.LSIDE BINOP # BOP M=BIT # NEWSIG )#
RPOS(0) IF(ST.C)
WORD = # EXPAND(DEST]).SRSIG(# PASIG(NEWSIG ) #)##
+ BOP #) # # (ST.AA)
# IDENTIFIER ; FNAME3 # (# FARG. LIST ( LIST #)) # RPOS(0)

ST.C
IMAGE POS(0) (# S.LSIDE BINOP # BOP UNARYOP # NEG1
+ (# IDENTIFIER ; FNAME3 (# FARG. LIST ( LIST #)) # RPOS(0)
+ IF(ST.E)
FNAM3 FINDFN IIF(SYNEHR)
FNCALL()
NEG1 # = #COMP(SRSIG(S($FR(# $FNAM3 #))) # :S(ST.A))
NEG1 = #SRSIG(S($FR(# $FNAM3 #)) #)

ST.D
WORD = LH.L #,# # EXPAND(# PASIG(L.SIDE) #,# NEG1 #,## BOP
+ #, # LEVCT = LEVCT - 1 ;#
+ # FCODE = #, # (ST.AA)
ST.E
T.CT = 0
+ WORD = # CONNECT(DEST,#,# FUN(IMAGE) #) #, # (ST.AA)

*******************************************************************************
**BTYPE**

**THIS ROUTINE TRANSLATES BOOK KEEPING STATEMENTS**

**INTO SUITABLE INTERMEDIATE CODE STATEMENTS**

---

```plaintext
BTYPE
AR1.STR =
IMAGE BREAK(*=* ) * L.SIDE LEN(1) = IF(BT.5)
L.SIDE = THIM(L.SIDE)
L.SIDE POS(0) = INDEX.VEC RPOS(0)
INDEX.VEC = INDEX.VEC V L.SIDE
BT.22 PLACE = PASIG(L.SIDE)
BT.1 IMAGE * * = :S(BT.1)
BT.11 IDENT(L.SIDE) = S(BT.13)
IMAGE BREAK(/=0/0/) * VAR.1 LEN(1) = OP.OR = IF(BT.12)
PL.2 = PASIG(TRIM(VAR.1))
ARI.STR = ARI.STR + + PL.2 + + OP.OR = (BT.11)
ARI.STR = ARI.STR = # PASIG(IMAG)
BT.13 WORD = LB.L # PLACE = # # ARI.STR = #
BT.16 IWORD = IWORD WORD = (RETURN).
BT.5 IMAGE SGOTO = IF(BT.8)
GO.TO = ALPHAPAT GO.TO
BT.6 WORD = LB.L # : (# GO.TO #) # : (BT.16)
BT.8 IMAGE BREAK(*=* ) * VAR.1 LEN(1) = IF(SYNERR)
VAR.1 = PASIG(VAR.1)
IMAGE BREAK(* =) * VAR.2 =
VAR.2 = PASIG(VAR.2)
IMAGE SPAN(* =) =
IMAGE EQEQ = *EQ* ;S(BT.9)
IMAGE LEGT = *LT* ;S(BT.9)
IMAGE LTE = *LT* ;F(LUCERR)
BT.9 IMAGE TAB(2) * LOGIC =
IMAGE SPAN(* =) =
IMAGE Dgoto :F(DGTER)
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 #) #
# = (BT.16)
```

---

*Image 0x0 to 743x539*
TTYPE ANALYSES THE CURRENT STATEMENT AND ON THE BASIS OF THE ANALYSIS, GENERATES A RETURN STATEMENT (FUNCTION) OR A DIRECT GOTO (SUBROUTINES) IF AN END STATEMENT IS ENCOUNTERED. IF THE CURRENT STATEMENT IS A TERMINAL SPECIFICATION STATEMENT, THE VALUE ASSIGNMENTS ARE MADE.

TTYPE  IMAGE END+ :F(TT,3)  259
   WORD = LB,L ≠ :(SCALL5) ≠  260
   IWORD = IWORD WORD  261
TT.1  EOF*2 = 1 :(RETURN)  262
TT.3  IMAGE BREAK(≠≠) LEN(1) = :F(SYNERR)  263
TT.3A IMAGE ↑↑ = :S(TT,3A)  264
   IDENT(FFLAG) :F(TT,4)  265
   IMAGE POS(0) IDENTIFIER * R1 ↑↑ IDENTIFIER * R2 RPOS(0)  266
   :S(TT,5)  266
   OUTVEC = OUTVEC v IMAGE  267
   $(≠0,# IMAGE) = 1  268
TT.3B  WORD = LB,L ≠  ≠  269
TT.2  IWORD = IWORD WORD :(RETURN)  270
TT.4  WORD = LB,L ≠ $:FR↑ ≠ FN,CT ≠ ≠ PASIG(IMAGE)  271
   ≠ ≠ :(TT,2)  271
TT.5  OUTVEC = OUTVEC v R1 v R2  272
   $(≠0,# R1) = 1  273
   $(≠0,# R2) = 2 :(TT,3B)  274
FUN IS A RECURSIVE FUNCTION WHICH TRANSLATES
BOOLEAN EXPRESSIONS INTO AMPL HARDWARE GENERATING STATEMENTS.

FUN0 5
XRAY ≠ # = :S(FUN0.5)
XRAY REDUCE = :S(FUN8.2)
XRAY REDUCE = :S(FUN8.3)
XRAY H.UNIT * OPA = :S(FUN2.0)
XRAY ANY H.UNIT * OPA = :S(FUN2.2)
XRAY #¥# BAL * OPA #¥# = :S(FUN7.0)
XRAY #¥# BAL * OPA #¥# = :S(FUN8.0)
XRAY H.UNIT * OPA #¥# = :S(FUN7.0)
XRAY H.UNIT * OPA #¥# = :S(FUN8.0)

FUN1 0 LIST = LIST # SRSIG(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
F OP2 = ;F(FUN0.5)
FUN5 0 IDENT(OP1 OP2) ;F(FUN5.1)
OP2 = ;F(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 ≠ #)
IWORD = IWORD WORD
LIST ≠ # SRSIG(# BLIM ≠ #)
IDENT(OP2) ;F(FUN8.1)
FUN6 0 LIST ≠ # = ;S(FUN6.0)
FUN = LIST ;F(RETURN)
FUN7 0 LIST = LIST # COMP(# FUN(OPA) ≠ #) :F(FUN2.1)
FUN8 0 LIST = LIST ≠ # FUN(OPA) :F(FUN2.1)
FUN8 1 OP1 = OP2 :F(FUN2.1)
FUN8 2 LIST = LIST # REDUCE(≠# B NOP ...# NA 1 ... NEG 1
≠# ...# NEG 2 ≠ #) :F(FUN2.1)
FUN8 3 LIST = LIST # REDUCE(≠# B NOP ...# REU OP ...# NUMBER ≠...#
≠# NA 1 ...# NEG 1 ...# NEG 2 ≠ #) :F(FUN2.1)
FNDCALL LINKS THE VARIABLES IN THE BODY OF THE FUNCTION WITH THOSE OF THE FUNCTION CALL.

```
FNCALL
    WORD = ↑ NLEVCT = LEVCT ; LEVCT = LEVCT + 1 ↑
    IWORD = IWORD ↑ WORD
    ARG.CT = 0

FNCL.1
    IDENT(LIST) ; S(FNCL.6)
    LIST BREAK(*,*) ARGUE LEN(1) = IF(FNCL.5)

FNCL.2
    ARG.CT = LT(ARG.CT,8) ARG.CT + 1 IF(ARGENR)
    IDENT(FFLAG) ; F(FNCL.4)

FNCL.3
    ARGUE POS(0) *INDEX.VEC RPOS(0) ; S(FNCL.7)
    WORD = # FTAB(LEVCT,# ARG.CT #) = ↑# ARGUE #; #

FNCL.9
    IWORD = IWORD WORD ; (FNCL.1)
    FNCL.7
    WORD = # FTABLEVCT,# ARG.CT # = FACIVAL(# ARGUE #) ; #
    ; (FNCL.9)

FNCL.4
    ARGUE POS(0) ARGVEC RPOS(0) ; F(FNCL.3)
    WORD = # FTAB(LEVCT,# ARG.CT #) = FTAB(NLEVCT,# ARGUE #) ; #
    ; (FNCL.9)

FNCL.5
    ARGUE = LIST
    LIST = ↑(FNCL.2)
    GET NAME OF BLOCK OF FUNCTION CODE.

FNCL.6
    INSERT = #FUNCTION# #FNAME3
    CAL.CNR = CAL.CNR + 1
    ADD TO THIS NAME A CALL SYMBOL
    INSERT2 = INSERT #C.# CAL.CNR
    EQUATE THE BLOCK OF CODE TO THE CALL NAME
    APPEND THE CODE WITH A RETURN TO NEXT STATEMENT
    COMMAND.
    INSERT2 = INSERT # ; (:# LB. # #R) #
    WRITE AN INSTRUCTION WHICH WILL EFFECT THE COMPILING
    AND EXECUTION OF THE APPENDED CODE LIST.
    WORD = # FCODE = CODE(# # INSERT2 #) ; [FCODE] ; #
    IWORD = IWORD WORD ; (RETURN)
```
PASIG EDITS THE CURRENT SOURCE ELEMENT AND
REturns THE ELEMENT IN SUitaBLE OUTPUT FORM.

PASIG IDENT(FFLAG) IF(PA3.0)
IDENT(KEY,#B#) IS(PA2.0)
PA1.0 PASIG = ### STRING ### : (RETURN)
PA2.0 PASIG = STRING : (RETURN)
PA3.0 IDENT(KEY,#B#) IS(PA8.0)
STRING SIGNALMATCH IF(PA7.0)
TP.T = TP.T + 1
TEMP = #TEMP# TP.T
INITIAL = # # TEMP # = # NEG,1 # # #
NA.1 POS(0) #ARGVEC RPOS(0) :F(PA6.0)
NUMBER POS(0) #ARGVEC RPOS(0) :F(PA5.0)
WORD = INITIAL # FTAH(LEVCT,# $NA.1 #) ^[ ]#
# FTAH(LEVCT,# #NUMBER #) ^[ ]#
PA4.0 IWORD = IWORD WORD
PA4.1 PASIG = TEMP : (RETURN)
PA5.0 WORD = INITIAL # FTAH(LEVCT,# $NA.1 #) ^[ ]#
# NUMBER # [ ] # I(PA4.0)
PA6.0 NUMBER POS(0) #ARGVEC RPOS(0) :F(PA1.0)
WORD = INITIAL # # NA.1 # # [ ] FTAB(LEVCT,# $NUMBER #)
# I [ ] # [] (PA4.0)
PA7.0 STRING PARTSIGNAL :F(PA1.0)
NA.1 POS(0) #ARGVEC RPOS(0) :F(PA1.0)
PA7.1 WORD = # # TEMP # = # NEG,1 # # FTAB(LEVCT,# $NA.1 #) ;# #
# I(PA4.0)
PA8.0 STRING POS(0) #ARGVEC RPOS(0) :F(PA2.0)
PASIG = #FTAB(LEVCT,# $STRING #) ;# # (RETURN)

136
CONNECT ADDS TO THE INPUT LIST OF THE HARDWARE
ELEMENT SPECIFIED ON THE LEFT OF AN ASSIGNMENT
STATEMENT THE SIGNAL WHICH IS THE RESULT OF THE
EXPRESSION ON THE RIGHT SIDE OF THE EQUAL SIGN.

CONNECT TO PLACE SIGNAL MATCH :F(CNC7)
NUMBER = FACTVAL(NUMBER)
NA1 FINDMEG :F(CNC2)
CNC0 NEG1 # = #RESET# :S(CNC1)
NEG1 = #SET#
CNC1 $ (NEG1 $NA1 # = # NUMBER) =$ (NEG1 $NA1 # = # NUMBER)
FROMSIG :(RETURN)
CNC2 P.SIG = NA1 NUMBER
GATEVEC = GATEVEC v NA1 NUMBER
SP.SIG = FROMSIG
NA1 POS0 (#OUTVEC RPOS0) :F(RETURN)
$ ( ( #0 = MA1 ) # = # NUMBER) = FROMSIG :(RETURN)
CNC7 TO PLACE PARTSIGNAL :F(CNC10)
IDENT(NA1 #RPULSE#) :F(CNC12)
OS CT = OS CT < 1
SUBRST[SCCT, 8] = # OS # OS CT
$ (#OS # OS CT) = FROMSIG
:(RETURN)
CNC12 NA1 FINDREG :F(CNC8)
NUMBER = 1 :F(CNC0)
CNC8 NUMBER = 1
CNC9 IDENT(NA1 #CPULSE#) :F(CNC2)
CPULSE = FROMSIG :(RETURN)
EXPAND ADDS A SIGNAL TO THE GATE SPECIFIED
ON THE LEFT HAND SIDE OF A RECURSIVE ASSIGNMENT
STATEMENT. IF NO GATE IS SO LABELED, ONE IS
OBTAINED AND GIVEN THE LABEL.

EXPAND OLD_SIG SIGNAL MATCH IF(EXP4,0)
NUMBER = FACTVAL(NUMBER)
NA1 FINDNEG :F(EXP2,0)
EXP1.0 NEG1 = ## = #RESET# :S(EXP4,0)
NEG1 = #SET#
EXP2.0 EXP = $NEG1 $NA1 ## NUMBER = $(NEG1 $NA1 ## NUMBER)
* NEWSIG = {RETURN}
EXP3.0 EXPAND = NA1 NUMBER
EXP2.0 EXPAND POS0 GATEVEC RPOS0 :S(EXP2.1)
GATEVEC = GATEVEC EXPAND
EXP4.0 $EXPAND = # Func *NEWSIG IF EXP7.0
EXP2.1 $EXPAND POS0 GATESIG RPOS0 :F(EXP6.2)
EXP7.0 NA1 POS0 GOUTVEC RPOS0 IF(RETURN)
EXP4.0 $EXPAND PARTSIG IF EXP6.2
EXP5.0 NA1 FINDNEG :F(EXP4.2)
NUMBER = 1 :F(EXP1.0)
EXP6.0 NUMBER = 1 :F(EXP2.0)

COMPLEMENT RETURNS THE LOGICAL INVERSE OF
THE CURRENT SIGNAL.

COMPLEMENT ELEMENT POS0 GATEVEC RPOS0 IF(COMP2.0)
TRUTHCODE = TRUTHCODE
COMP = TRUTHCODE POS0 IF(COMP2.0)
TRUTHCODE = TRUTHCODE
COMP1.0 COMP = TRUTHCODE POS0 IF(COMP1.0)
COMP2.0 ELEMENT = $C(COMP2.0)
CUMP = ELEMENT
CUMP = ZERO :S(RETURN)
CUMP = ONE :S(RETURN)
CUMP = CUMP :S(RETURN)
CUMP = $C(RETURN)
SRSIG returns the real name of the current hardware element.

SRSIG (BT.5G # ARB * SRSIG #) = \$ (SRSIG6.5)
SRSIG0.5 IDENT (SRSIG) = \$ (SRSIG6.6)
SRSIG (T.5G \+ F.5G) INTEGER \# NUMBER IF (SRSIG1.0) = SRSIG \+ (RETURN) 0.02
SRSIG1.0 SRSIG \+ 0 IF (SRSIG2.0) = SRSIG \+ (RETURN) 0.03
SRSIG \+ 0 IF (SRSIG3.0) = SRSIG \+ (RETURN) 0.04
SRSIG \+ 0 IF (SRSIG4.0) = SRSIG \+ (RETURN) 0.05

SRSIG2.0 SRSIG POS(0) GATESIG RPOS(0) IF (RETURN) 0.06
SRSIG SIGNALMATCH IF (SRSIG6.0) \# NUMBER FACTVAL (NUMBER) 0.07
SRSIG2.1 SRSIG \+ 0 NA.1 NUMBER IF (RETURN) 0.08

SRSIG3.0 NA.1 FINDREG IF (SRSIG5.0)
SRSIG3.1 PL. GT = SUBRHS (SRSIG.0. SNA.1) \+ 0 \+ NUMBER IDENT (NA.1) \+ IF (RETURN) 0.09
SRSIG4.0 SRSIG = \+ T. \+ PL. GT \+ (RETURN) 0.10
SRSIG5.0 SRSIG = \+ 0 \+ PL. GT \+ (RETURN) 0.11
SRSIG5.1 SRSIG = \+ 0 \+ GATEVEC RPOS(0) \+ (RETURN) 0.12
SRSIG = \# PL. GT \+ (RETURN) 0.13
SRSIG5.2 SRSIG = \# PL. GT \+ (RETURN) 0.14
SRSIG6.0 SRSIG PARSSIGNAL IF (RETURN) 0.15
SRSIG = \# 0 \+ 0 IF (RETURN) 0.16
SRSIG8.0 \# NUMBER IF (RETURN) 0.17
SRSIG8.1 \# NUMBER IF (RETURN) 0.18
SRSIG8.2 \# NUMBER IF (RETURN) 0.19
REDUCE IS RESPONSIBLE FOR GENERATING THE LOGIC IMPLIED BY THE REDUCE OPERATION.

**REDUCE**

Q.STR =
NEG.1 y = 0 F# $3(REDO,5)
NEG.1 = T
REDO.5 IS(RD,OP) $4(RDC.2)
SRSIG7.1 NA.1 FNA=NEG $4(SRSIG7.4)
IN.1 = 0 434
MAX.1 = $3(RL,NA.1)
SRSIG7.2 IN.1 = LT(IN.1,MAX.1) IN.1 + 1 $4(SRSIG7.3)
Q.STR = Q.STR $4(NEG,2 NA.1 IN.1 $4(SRSIG7.2)
SRSIG7.3 REDUCE = NEG.1 GEN(B,NOP,Q.STR) $4(RETURN)

*******
(SRSIG7.4) THE CURRENT ARGUMENT CALLS FOR THE REDUCTION OF AN OR-BUS VECTOR (TO THE CURRENT LOGIC UNIT) INTO A SINGLE GATE.
(SRSIG7.5) SINGLE GATE, OBTAIN FIRST GATE NUMBER FROM SUBINST.
THE LENGTH OF THE BUS IS GIVEN BY $3(#DA, NA.1) WHERE
SNA.1 IS THE INDIRECT OPERATOR VALUE OF THE ALPHAMERIC LABEL USED AS THE ARGUMENT.
*SRSIG7.4 NA.1 FINDM $ IF(REDERR)  
  NEG.2 = +T$  
  :S(SRSIG7.5)  
  SRSIG7.5 IN.1 = SUBRNST[SR.CT,SNA.11] - 1  
  MAX.1 = $((DA,#$NA.1) + IN.1)  
  IN.1 = LT(IN.1,MAX.1) IN.1 + 1  :F(SRSIG7.3)  
  Q.STR = Q.STR NEG.2 IN.1  :F(SRSIG7.6)  
  RDC.2 NUMBER = FACTVAL(LNG)  
  SRSIG8.1 NA.1 FINDREG IF(SRSIG8.3)  
  IDENT(REO.OP,#PREFIX#) :S(SRSIG8.2)  
  IN.1 = 0  
  MAX.1 = NUMBER  
  IDENT(REO.OP,#PREFIX#)  
  SRSIG8.2 IN.1 = (#RL.#$SA.11) - NUMBER  
  MAX.1 = IN.1 + NUMBER  
  SRSIG8.3 NA.1 FINDM  
  IF(REDERR)  
  NEG.2 = +T$  
  :S(SRSIG8.4)  
  REDERR ERROR = #**REDUCE OPERATION UNRECOGNIZED**#  
  REDUCE = # DEFAULT#  
  :F(RETURN)
DEBUG

OUTPUT = # DBG1 = # DBG1
OUTPUT = # DBG2 = # DBG2
OUTPUT = DBG3 # DEBUG# I(RETURN)

SRS1A IDENT(CPULSE) #F(SRS1a)
CPULSE = # GEN(#F,)
SUBRIST(SR,C,11) = CPULSE

SRS1B SNSIG = CPULSE I(RETURN)
SRS1G,B SNSIG = #DEFAULT #I(RETURN)

ERROR SECTION

SRSIGERR ERROR = #****RECOGNIZED ARGUMENT IN SNSIG,#
OUTPUT = # ARGUMENT WAS: # SNSIG I(RECOVER)

EXP5-2 ERROR = #****TROUBLE IN EXPAND, NEW-SIGNAL = # NEW-SIG
+ # OLD-SIGNAL = # OLD-SIG
OUTPUT = # HARDWARE NAMED OLD-SIGNAL COULD NOT BE FOUND,#
I(RECOVER)

CNC10 ERROR = #****DESTINATION NOT RECOGNIZED BY CONNECT,#
OUTPUT = # DESTINATION WAS: # TO,PLACE
OUTPUT = # SOURCE WAS: # FROMSIG I(RECOVER)

SYNERR ERROR = #****SYNTAX ERROR, THIS STATEMENT NOT RECOGNIZED,#
I(RECOVER)

SYNERR2 ERROR = #****RECOGNIZED SIGNAL,# I(RECOVER)

SIGNERR ERROR = #****ATTEMPT TO EXCEED MAXIMUM ALLOWABLE NUMBER,#
+ # OF SUBROUTINES,# I(RECOVER)

ACTERR ERROR = #****ATTEMPTED TO EXCEED MAXIMUM ALLOWABLE NUMBER,#
+ # OF ARGUMENTS IN SUBROUTINE OR FUNCTION,#
I(RECOVER)

ARGERR ERROR = #****LENGTH OF INPUT BUS FOR DUMMY SUBROUTINE,#
+ #ARGUMENT OMITTED,# I(RECOVER)

KEYERR ERROR = #****RECOGNIZABLE KEY,# I(RECOVER)

COMPERR ERROR = #****SWOBOL4 COMPILE ERROR IN OBJECT# I(RECOVER)

LOCERR ERROR = #****COMPARE RELATION NOT RECOGNIZED,# I(RECOVER)

DGTERR ERROR = #****GOTO NOT RECOGNIZED,# I(RECOVER)

SYNERR3 ERROR = #****TROUBLE WITH XRAY BY FUN-FUNCTION,#
OUTPUT = # TROUBLE STARTED WITH XRAY I(RECOVER)

OPTION OPTION = PATTERN # NULL I(RETURN)
GEN IS A FUNCTION WHICH FETCHES THE NEXT AVAILABLE GATE AND ASSIGNS TO ITS INPUT LIST THE VALUE OF THE SECOND ARGUMENT OF THE CALL. THE GATE-TYPE OF THE CURRENT GATE IS SET EQUAL TO THE FIRST ARGUMENT.

GEN GATCT = GATCT + 1
GEN = GATCT
$(!G.* GATCT ) = Y
$( !G.* GATCT ) = X ; (RETURN)

FACTVAL RETURNS THE VALUE OF THE IDENTIFIER

FACTVAL FACTVAL = INDEXVAR
FACTVAL INTEGER ; (RETURN)
FACTVAL $FACTVAL ; (RETURN)

RECOVER OUTPUT = ^ FATAL ERROR DETECTE D IN AMPL PROGRAM^ AT TIME = ^ TIME()^ OUTPUT = ^ LAST GATE ASSIGNED WAS ^ GATCT ^^ OUTPUT = ^ FIRST GATES WERE ASSIGNED^ OUTPUT = ^ AMPL PROCESSING TERMINATED^ OUTPUT = ^ AMPL COMPILER (VER 3.0.8), DEP I OF ELECTRICAL # ENGINEERING^ WORD = ^UNIVERSITY OF ARIZONA COMPUTER CENTER^ (END)
NXTST IDENT(EOF) :F(END) 506
COND = . 507
OUTPUT = + 508
IMAGE = LINE 509
READC LINE = TRIM(INPUT) :F(ENDGAME) 510
LINE COMMENT :S(PRINT) 511
LINE CONTINUE = :F(RETURN) 512
IMAGE = IMAGE LINE ; (READC) 513
PRINT OUTPUT = + 514
ENDGAME EOF = 1 :F(RETURN) 515
END 516
LIST OF REFERENCES


