INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
MASUD, MANZER

MODULAR IMPLEMENTATION OF A DIGITAL HARDWARE DESIGN AUTOMATION SYSTEM

The University of Arizona

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

Copyright 1981 by Masud, Manzer

All Rights Reserved
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages ______
2. Colored illustrations, paper or print ______
3. Photographs with dark background ______
4. Illustrations are poor copy ______
5. Pages with black marks, not original copy ______
6. Print shows through as there is text on both sides of page ______
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ______
11. Page(s) ___________ lacking when material received, and not available from school or author.
12. Page(s) ___________ seem to be missing in numbering only as text follows.
13. Two pages numbered __________. Text follows.
14. Curling and wrinkled pages ______
15. Other ____________________________________________________________

University
Microfilms
International
MODULAR IMPLEMENTATION OF A DIGITAL HARDWARE DESIGN AUTOMATION SYSTEM

By

Manzer Masud

A Dissertation Submitted to the Faculty of the DEPARTMENT OF ELECTRICAL ENGINEERING In Partial Fulfillment of the Requirements For the Degree of DOCTOR OF PHILOSOPHY In the Graduate College THE UNIVERSITY OF ARIZONA

1981

©Copyright 1981 Manzer Masud
THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Manzer Masud entitled Modular Implementation of a Digital Hardware Design Automation System and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

[Signatures and dates]

Dissertation Director
26 May 1981
STATEMENT BY AUTHOR

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

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

SIGNED: [Signature]
ACKNOWLEDGMENTS

Alhamdolillah (all praise is due to Allah), the research has reached the stage that it can be presented as a dissertation. I would like to express my deep appreciation to all those who participated in the research and also to those who contributed in making my stay in the United States an enjoyable experience.

I would like to thank Professor Paul Skinner of Speech and Hearing Sciences, Professor Richard Schotland of Atmospheric Sciences, and Professor Roy Mattson of the Electrical Engineering Department for giving me an opportunity to work with their respective departments.

Outside the University of Arizona, I would like to express my gratitude to Dr. Kenneth Wacks of Teradyne Inc., Boston, MA, for inviting me to work on a very rewarding summer research project. His enthusiastic support of AHPL was very encouraging. Among the people who participated in the summer project, I would like to mention the name of Peter deBryun Kops, a graduate student at Harvard who is now working with Teradyne; and Dave Trumper, an MIT undergraduate.

Among my colleagues, I must acknowledge the contribution of Zainulabedeen Navabi and Duan-Ping Chen towards the project.

Financial support from Teradyne Inc., VHSIC - Department of Defense, and General Instruments is gratefully acknowledged.
Most of the credit for the research goes to Professor Frederick J. Hill whose guidance and support played an important role in the successful completion of the project.

Finally, I would like to thank my family, particularly my parents, for their encouragement and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>x</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xi</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Objectives</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Selection of a Proper Language</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 The Generalized Language</td>
<td>5</td>
</tr>
<tr>
<td>1.1.3 User-Specific Sublanguages</td>
<td>7</td>
</tr>
<tr>
<td>1.1.4 Application-Dependent Output</td>
<td>8</td>
</tr>
<tr>
<td>1.2 Approach</td>
<td>9</td>
</tr>
<tr>
<td>1.2.1 The Multistage Compiler</td>
<td>10</td>
</tr>
<tr>
<td>1.2.2 User Parameters</td>
<td>14</td>
</tr>
<tr>
<td>2 ON EXTENDING AHPL</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Review of AHPL II</td>
<td>16</td>
</tr>
<tr>
<td>2.1.1 Basic Operations</td>
<td>18</td>
</tr>
<tr>
<td>2.1.2 The Design Philosophy</td>
<td>18</td>
</tr>
<tr>
<td>2.1.3 Modularity and Task Subdivision</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Some Limitations and Apparent Limitations of AHPL II</td>
<td>22</td>
</tr>
<tr>
<td>2.2.1 Open-Ended Language</td>
<td>23</td>
</tr>
<tr>
<td>2.2.2 AHPL's Association with a Logic Family</td>
<td>24</td>
</tr>
<tr>
<td>2.2.3 Formal Language Considerations</td>
<td>24</td>
</tr>
<tr>
<td>2.2.4 Hardware Flexibility</td>
<td>32</td>
</tr>
<tr>
<td>2.3 Summary of Added Features</td>
<td>32</td>
</tr>
<tr>
<td>3 SPECIFICATION OF UNIVERSAL AHPL</td>
<td>35</td>
</tr>
<tr>
<td>3.1 Terminal Symbols</td>
<td>35</td>
</tr>
<tr>
<td>3.1.1 Buses and Excuses</td>
<td>36</td>
</tr>
<tr>
<td>3.1.2 Memory Elements</td>
<td>37</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3. Input/Output Lines</td>
<td>39</td>
</tr>
<tr>
<td>3.1.4. CLU and FNREG</td>
<td>40</td>
</tr>
<tr>
<td>3.2. Syntax</td>
<td>41</td>
</tr>
<tr>
<td>3.3. Semantics</td>
<td>47</td>
</tr>
<tr>
<td>4. IMPLEMENTATION OF STAGE I</td>
<td>62</td>
</tr>
<tr>
<td>4.1. Stage I Output</td>
<td>62</td>
</tr>
<tr>
<td>4.1.1. Symbol Table</td>
<td>63</td>
</tr>
<tr>
<td>4.1.2. System Table (SYSTAB)</td>
<td>63</td>
</tr>
<tr>
<td>4.1.3. Symbol Declaration Table (SDT)</td>
<td>63</td>
</tr>
<tr>
<td>4.1.4. Symbol Reference Table (SRT)</td>
<td>67</td>
</tr>
<tr>
<td>4.1.5. Step QTABLE Relation Table (SQRT)</td>
<td>67</td>
</tr>
<tr>
<td>4.1.6. Quadruple Table (QTABLE)</td>
<td>67</td>
</tr>
<tr>
<td>4.1.7. Table of Temporary Symbols (TOTS)</td>
<td>67</td>
</tr>
<tr>
<td>4.1.8. Reference Table (REF)</td>
<td>67</td>
</tr>
<tr>
<td>4.1.9. Parameter Table (PARAM)</td>
<td>72</td>
</tr>
<tr>
<td>4.1.10. Argument Table (ARG)</td>
<td>72</td>
</tr>
<tr>
<td>4.1.11. FOR Table</td>
<td>72</td>
</tr>
<tr>
<td>4.1.12. IF Table</td>
<td>72</td>
</tr>
<tr>
<td>4.1.13. THUNK Table</td>
<td>77</td>
</tr>
<tr>
<td>4.1.14. PINTAB Table</td>
<td>77</td>
</tr>
<tr>
<td>4.1.15. Label Reference Table (LRT)</td>
<td>77</td>
</tr>
<tr>
<td>4.1.16. Pulse Table</td>
<td>77</td>
</tr>
<tr>
<td>4.2. Syntax Analysis</td>
<td>86</td>
</tr>
<tr>
<td>4.3. Semantic Action</td>
<td>88</td>
</tr>
<tr>
<td>4.4. Use of Stage I Output for Simulation</td>
<td>91</td>
</tr>
<tr>
<td>5. OPTIMIZATION USING LINKED LIST</td>
<td>94</td>
</tr>
<tr>
<td>5.1. Structure of a Node</td>
<td>95</td>
</tr>
<tr>
<td>5.2. Structure of I0LIST</td>
<td>96</td>
</tr>
<tr>
<td>5.3. Network Representation Using Linked List</td>
<td>96</td>
</tr>
<tr>
<td>5.4. Removal of Redundant Elements</td>
<td>96</td>
</tr>
<tr>
<td>5.5. Other Optimizations</td>
<td>99</td>
</tr>
<tr>
<td>6. IMPLEMENTATION OF STAGE 2</td>
<td>101</td>
</tr>
<tr>
<td>6.1. An Overview of Stage 2</td>
<td>101</td>
</tr>
<tr>
<td>6.2. Additional Processing for Submodules</td>
<td>111</td>
</tr>
<tr>
<td>6.2.1. CLU Processor</td>
<td>113</td>
</tr>
<tr>
<td>6.2.2. Functional Register Processor</td>
<td>113</td>
</tr>
<tr>
<td>6.3. A Demonstration Stage 3</td>
<td>114</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS — Continued

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>A COMPILER EXAMPLE</td>
<td>115</td>
</tr>
<tr>
<td>8</td>
<td>A GUIDE FOR WRITING STAGE 3</td>
<td>133</td>
</tr>
<tr>
<td>8.1</td>
<td>Defining Local Parameters</td>
<td>133</td>
</tr>
<tr>
<td>8.2</td>
<td>Accessing and Manipulating AELL</td>
<td>135</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Building Segments</td>
<td>136</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Changing Node Numbers</td>
<td>138</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Assigning PIN Numbers to MSI Parts</td>
<td>140</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Minimizing the Number of Control Elements</td>
<td>141</td>
</tr>
<tr>
<td>8.3</td>
<td>A Proposed Mask Generation System</td>
<td>143</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Meeting FANIN FANOUT Requirements</td>
<td>144</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Converting into NAND Gates</td>
<td>146</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Converting Flipflops into NAND Gates</td>
<td>148</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Output Formatting</td>
<td>150</td>
</tr>
<tr>
<td>8.3.5</td>
<td>Conclusions</td>
<td>150</td>
</tr>
<tr>
<td>9</td>
<td>CONCLUSIONS AND CURRENT APPLICATIONS</td>
<td>151</td>
</tr>
<tr>
<td>9.1</td>
<td>Current Applications</td>
<td>151</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Device Modeling System for Test Engineers</td>
<td>151</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Test Sequence Generator</td>
<td>155</td>
</tr>
<tr>
<td>9.1.3</td>
<td>SLA Implementation of VLSI</td>
<td>157</td>
</tr>
<tr>
<td>9.2</td>
<td>Evaluation of Accomplishment</td>
<td>161</td>
</tr>
<tr>
<td>9.3</td>
<td>Future Research</td>
<td>163</td>
</tr>
<tr>
<td>9.4</td>
<td>Available AHPL Software</td>
<td>164</td>
</tr>
<tr>
<td></td>
<td>LITERATURE CITED</td>
<td>166</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Illustration</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Subset languages</td>
<td>8</td>
</tr>
<tr>
<td>1.2</td>
<td>Three-stage hardware compiler</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>A simple system</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>A hardware realization of the system</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Data/control partition</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Hierarchy of description languages</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>SYNTAX interaction with the rest of the system</td>
<td>87</td>
</tr>
<tr>
<td>4.2</td>
<td>The proposed simulator</td>
<td>93</td>
</tr>
<tr>
<td>5.1</td>
<td>Node representing a network element</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>A partial network</td>
<td>97</td>
</tr>
<tr>
<td>5.3</td>
<td>Node representation of the circuit of Figure 5.2</td>
<td>97</td>
</tr>
<tr>
<td>5.4</td>
<td>A typical arrangement of SIGLISTS</td>
<td>99</td>
</tr>
<tr>
<td>6.1</td>
<td>A simplified VTOC representation of Stage 2</td>
<td>102</td>
</tr>
<tr>
<td>6.2</td>
<td>An initialized node</td>
<td>103</td>
</tr>
<tr>
<td>6.3</td>
<td>A memory element initialization</td>
<td>104</td>
</tr>
<tr>
<td>6.4</td>
<td>A control flipflop initialization</td>
<td>105</td>
</tr>
<tr>
<td>6.5</td>
<td>Sample ARGLIS entries</td>
<td>107</td>
</tr>
<tr>
<td>6.6</td>
<td>Circuit for Example 6.1</td>
<td>108</td>
</tr>
<tr>
<td>6.7</td>
<td>Circuit for Example 6.2</td>
<td>108</td>
</tr>
<tr>
<td>6.8</td>
<td>Circuit generated by Example 6.3</td>
<td>109</td>
</tr>
<tr>
<td>6.9</td>
<td>Multiple activities to a flipflop simple strategy</td>
<td>110</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.10.</td>
<td>Multiple activities by forming common control subexpression</td>
<td>110</td>
</tr>
<tr>
<td>7.1.</td>
<td>A compiler example</td>
<td>116</td>
</tr>
<tr>
<td>7.2.</td>
<td>Stage 1 tables</td>
<td>119</td>
</tr>
<tr>
<td>7.3.</td>
<td>Stage 2 Abstract Element Linked List</td>
<td>124</td>
</tr>
<tr>
<td>7.4.</td>
<td>Stage 3 output</td>
<td>126</td>
</tr>
<tr>
<td>7.5.</td>
<td>A partial network for the example</td>
<td>131</td>
</tr>
<tr>
<td>8.1.</td>
<td>Block diagram of a typical AHPL application</td>
<td>134</td>
</tr>
<tr>
<td>8.2.</td>
<td>Building segment table consecutive nodes</td>
<td>136</td>
</tr>
<tr>
<td>8.3.</td>
<td>Building segment table non-consecutive nodes</td>
<td>137</td>
</tr>
<tr>
<td>8.4.</td>
<td>Node map</td>
<td>139</td>
</tr>
<tr>
<td>8.5.</td>
<td>The proposed mask generator system</td>
<td>145</td>
</tr>
<tr>
<td>8.6.</td>
<td>Meeting FANOUT requirements</td>
<td>146</td>
</tr>
<tr>
<td>8.7.</td>
<td>A logic circuit and its NAND equivalents</td>
<td>147</td>
</tr>
<tr>
<td>8.8.</td>
<td>Replacing gates by their NAND equivalents</td>
<td>148</td>
</tr>
<tr>
<td>8.9.</td>
<td>NAND representation of a D-flipflop</td>
<td>149</td>
</tr>
<tr>
<td>9.1.</td>
<td>SCIRTS flow diagram</td>
<td>158</td>
</tr>
<tr>
<td>9.2.</td>
<td>AHPL compiler interfaced with SCIRTS</td>
<td>159</td>
</tr>
<tr>
<td>9.3.</td>
<td>Compilation of AHPL description into SLA layout</td>
<td>161</td>
</tr>
<tr>
<td>9.4.</td>
<td>A comprehensive digital design automation system</td>
<td>165</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1. AHPL operators</td>
<td>19</td>
</tr>
<tr>
<td>2.2. Basic AHPL operations</td>
<td>20</td>
</tr>
<tr>
<td>4.1. Symbol table, description of contents</td>
<td>64</td>
</tr>
<tr>
<td>4.2. System table, description of contents</td>
<td>65</td>
</tr>
<tr>
<td>4.3. Symbol declaration table, description of contents</td>
<td>66</td>
</tr>
<tr>
<td>4.4. Symbol reference table, description of contents</td>
<td>68</td>
</tr>
<tr>
<td>4.5. Step QTABLE relation table, description of contents</td>
<td>69</td>
</tr>
<tr>
<td>4.6. Quadruple table, description of contents</td>
<td>70</td>
</tr>
<tr>
<td>4.7. Table of temporary symbols, description of contents</td>
<td>71</td>
</tr>
<tr>
<td>4.8. REF table, description of contents</td>
<td>73</td>
</tr>
<tr>
<td>4.9. PARAM table, description of contents</td>
<td>74</td>
</tr>
<tr>
<td>4.10. ARG table, description of contents</td>
<td>74</td>
</tr>
<tr>
<td>4.11. FOR table, description of contents</td>
<td>75</td>
</tr>
<tr>
<td>4.12. IF table, description of contents</td>
<td>76</td>
</tr>
<tr>
<td>4.13. THUNK table, description of contents</td>
<td>78</td>
</tr>
<tr>
<td>4.14. PINTAB table, description of contents</td>
<td>79</td>
</tr>
<tr>
<td>4.15. LRT table, description of contents</td>
<td>79</td>
</tr>
<tr>
<td>4.16. PULSE table, description of contents</td>
<td>79</td>
</tr>
</tbody>
</table>
With the advent of LSI and VLSI technology, the demand and affordability of custom tailored design has increased considerably. A short turnaround time is desirable along with more credible testing techniques. For a low-production device it is necessary to reduce the time and money spent in the design process. Traditional hardware design automation techniques rely on extensive engineer interaction. A detailed description of the circuit to be manufactured must be entered manually. It is often necessary to prepare a separate description for each phase of the design process. In order to be successful, a modern design automation system must be capable of supporting all phases of design activities from a single circuit description. It must also provide an adequate level of abstraction so that the circuit may be described conveniently and concisely. Such abstraction is provided by computer hardware description languages (CHDL).

In this research, an automation system based on AHPL (A Hardware Programming Language) has been developed. The project may be divided into three distinct phases: (1) Upgrading of AHPL to make it more universally applicable; (2) Implementation of a compiler for the language; and (3) Illustration of how the compiler may be used to support several phases of design activities.

Several new features have been added to AHPL. These include: application-dependent parameters, multiple clocks, asynchronous resets, xi
functional registers and primitive functions. The new language, called Universal AHPL, has been defined rigorously.

The compiler design is modular. The parsing is done by an automatic parser generated from the SLR(1) BNF grammar of the language. The compiler produces two data bases from the AHPL description of a circuit. The first one is a tabular representation of the circuit, and the second one is a detailed interconnection linked list. The two data bases provide a means to interface the compiler to application-dependent CAD systems.

In the end, a discussion on how the AHPL compiler can be interfaced to other CAD systems is given, followed by examples from current applications and from ongoing research projects. These applications illustrate the usefulness of a CHDL-based approach to the design of digital hardware automation systems.
CHAPTER 1

INTRODUCTION

1.1. Objectives

Recent developments in integrated circuit technology have made it possible to manufacture complex digital systems at a low per-system cost. Full potential of this technological capability, however, can be utilized only if associated hardware design automation systems (HDAS) can be upgraded to the same level of low cost productivity. This is especially true of special-purpose low-production digital systems.

Computer Aided Design (CAD) systems for digital design commonly available today were developed several years ago. At that time design languages had not matured enough to support complex design problems. These CAD systems, therefore, are not based on design languages and require the design engineer to provide a very detailed description of his circuit. These systems do not provide much help to the designer in organizing his design and do not aid his thinking process. Because the descriptions are lengthy and tedious, they are prone to errors and debugging is difficult, all of which translates into high design cost and low productivity.

Design of a digital system usually involves the following processes:
1. Specification of the system structure and functional behavior.
2. Testing of the functional behavior of the system.
4. Testing the circuit for timing and loading problems.
5. Final fabrication.

CAD systems are available to aid the designer at all levels of design activities. However, CAD systems at different levels were developed individually and each has its own front end. Thus, several descriptions of the same circuit must be written to make use of these systems. Loss of time and effort is obvious; a more serious problem is that one is never sure if the descriptions at different levels represent the same circuit. In other words, there is no guarantee that the circuit being manufactured is the same as the one which was tested.

A large digital system may use several components. These components are tested individually, but yet another CAD system is needed to test the whole system. Models of individual components must be prepared again and their interconnection specified. This, again, translates into loss of time and possibility of error.

From the above discussion, it is evident that a unified approach to the design problem must be taken. A complete automation system is needed in which the circuit to be designed needs to be described only once and at an appropriate level of abstraction. Such abstraction is
provided by the design languages. It may be true to say that if the realization of special purpose low-production digital system is to be feasible, it will be through the use of design languages. The purpose of this research was to develop one such HDAS. The design automation system discussed in the following pages has these desirable features:

1. It allows the design engineer to describe his circuit at an adequate level of abstraction.
2. It supports simulation of the functional behavior of the circuit.
3. It supports various technologies.
4. It generates implementable object description.
5. It may be interfaced easily to other available CAD systems in order to support all phases of design activities.

The first step towards the design of such a system is to select a proper design language; to enhance it if necessary; and, finally, to implement it in such a way that it can support several applications. This is the methodology followed in this research. The implementation is modular, in order to make the system easy to write, easy to maintain, and adaptable to future expansion.

1.1.1. Selection of a Proper Language

A design-language-based HDAS requires the user to describe his circuit using the design language. The system processes this description and generates the desired output. Thus, a proper choice of language is essential for a good automation system.
Design languages may be classified on the basis of their abstraction. There are four major levels of abstraction:

a. System level.

b. Register transfer level.

c. Gate and component level.

d. Detailed wiring level.

System level is the most abstract level, while the wiring level is the most detailed. The less abstract the level is, the more control the designer has over the final output. As the designer gains more control, he is expected to provide more information and the description becomes more tedious.

A digital design engineer is usually not interested in details beyond gate and component level. However, gate-level description itself can become very tedious for modern LSI and VLSI devices or any large digital system. This has severely hampered the productivity of designers.

With a very high level of abstraction, e.g., the system level, the design engineer loses most of his control over the final output. He is unable to include architectural details in his description which are necessary for an efficient circuit design.

The register transfer level provides the necessary user control but not at the expense of his productivity. Designers of large digital systems visualize their systems as being composed of elements grouped together as registers or memory where the information can be stored and
and interconnecting circuitry to process this information. Therefore, the most suitable level for the purpose of describing these systems is the register transfer level. This explains why most computer hardware description languages (CHDL) use this level of abstraction [1-5].

A HDAS based on a Register Transfer Language (RTL) may use a fixed design rule to generate the final output from the circuit description. However, a design engineer would like to have some control over the design rule—some means to make technology-dependent decisions or to select components of his choice—some way to enter his judgment in the automatic design process. This problem and some solutions will be discussed in more detail in the following pages. For now it suffices to say that a register transfer language with ability to include some lower-level details is a good starting point.

AHPL [6] is one of the more widely circulated and completely documented of the existing hardware description languages [24]. It has been thoroughly tested for consistency and unambiguity. It has good software support. Furthermore, a native language always seems to be easier and more attractive. For these reasons, the automation system proposed here uses AHPL as the main user interface. This is not to say that a similar system for another language cannot be developed. The approach taken here is modular and the basic design principles can easily be used for systems based on other design languages.

1.1.2. The Generalized Language

AHPL of reference [6], called AHPL II, is a medium for clock mode description based on the assumption that most digital systems can be
partitioned into a control section and a data section. The two major parts of an AHPL description of a sequential module are the declarations and the control sequence. In the declaration section, various buses, registers, inputs, outputs and combinational logic units are declared. The control sequence is a list of numbered statements, each consisting of an action followed by a branch. All register transfer and bus connections are specified in the action part. The branch part specifies which set of actions are to be performed next. The branch may be conditional on the value of any line or register.

AHPL II is a good design language, but it does not give the designer enough control over the design rule. It assumes single-clock synchronous circuit, allows only falling-edge-triggered D flipflops as storage elements, and AND/OR type of buses. Such restrictions are difficult to follow in a real-world situation. Design practices vary from one group to another, and some flexibility is essential to allow for variations. A design automation system may also be used by test engineers. They are required to test devices developed by a number of design groups, so for them the flexibility is absolutely necessary. Reference [7] discusses an extension and formalization of AHPL II to AHPL III. This language provides for three types of structures: procedural structure, combinational logic unit, and functional registers. Furthermore, a procedural structure may either be a primitive or a non-primitive procedural structure. The primitive structure is called a module. The non-primitive structure provides a means to
express repetitive connections among modules of a system. AHPL III also permits the specification of types of a memory element using parameters. Driving clocks may also be specified.

For the purpose of this research, a superset of AHPL II has been developed. This language is called Universal AHPL. It incorporates all the features of AHPL III except for the facility to interconnect modules using non-primitive procedural structures. Several additional features have been added; these include: asynchronous data transfer, global branches (resets), and multiple clocks. Use of parameters is allowed for non-memory elements also.

The primary description segment in the Universal AHPL is the Module. To further organize the design, two additional description segments, the combinational logic unit (CLU) and the functional register (FNREG) may be invoked within a module. A CLU consists only of combinatorial logic while a FNREG includes memory elements as well as logic but no sequential control. As an example, an arithmetic-logic-unit (ALU) might be represented as a functional register, or as a CLU and a declared memory register. The Universal AHPL also allows system-defined primitive functions. These functions may be invoked like a CLU. New primitive functions may be added at any time without making any change in the language syntax. Chapter 3 gives a complete description of the language.

1.1.3. User-Specified Sublanguages

Features discussed above were added to AHPL II on the basis of a careful study of a variety of design environments. Not every feature
of Universal AHPL will be of interest in every application. The users manual of a particular application will always define a subset language. It is possible to start from a small subset and include additional features as their need arises.

Figure 1.1 shows some of the possible subset languages. Obviously, more languages will evolve as new applications are discovered.

**Figure 1.1. Subset languages.**

1.1.4. Application-Dependent Output

The output of a hardware compiler is not a set of executable instructions but a "suitable" output to drive some form of or some part of a circuit implementation process. This suitable output may be in totally different and seemingly unrelated forms, depending on the need of different users. For instance, one user may need the
output to be in terms of simple logic elements—And, Or, Inverter and flipflops—while another may want the system to generate a sequence of commands for a numeric controller for wire-wrapping or drafting. To elaborate the point, some of the applications and the required outputs are listed below.

<table>
<thead>
<tr>
<th>Application</th>
<th>Required Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function-level simulator</td>
<td>Time behavior of the circuit</td>
</tr>
<tr>
<td>Logic network generation</td>
<td>And, Or, Inverter and flipflops</td>
</tr>
<tr>
<td>OEM</td>
<td>SSI, MSI and LSI parts and their interconnections</td>
</tr>
<tr>
<td>Wire wrap prototypes</td>
<td>Commands to numeric controller</td>
</tr>
<tr>
<td>LSI and VLSI</td>
<td>Mask generation</td>
</tr>
<tr>
<td>Gate-level simulator</td>
<td>Gate-level time behavior</td>
</tr>
<tr>
<td>Automatic test sequence generation (using network equations)</td>
<td>Destination equations</td>
</tr>
<tr>
<td>Automatic test sequence generation (using D-Algorithm)</td>
<td>Gates, flipflops and interconnectors</td>
</tr>
<tr>
<td>VLSI using SLA</td>
<td>SLA Layout</td>
</tr>
<tr>
<td>Documentation</td>
<td>None</td>
</tr>
</tbody>
</table>

These seemingly unrelated outputs have one thing in common—they can all be derived from the same source program. Thus a successful general-purpose design automation system must be able to support a variety of outputs with a minimal software development cost.

1.2. Approach

Design of a software-language processor involves implementation of an algorithm to convert a source code at a higher level of
abstraction into an object code at a more detailed level. The language processor takes care of all the idiosyncracies of the target machine so that the language can be designed independently of the machine on which it is to be implemented. Similarly, the processor of a hardware language can convert an abstraction into a suitable output.

A language itself does not define the transformation from source code into final output—the object code. In fact, there could be a family of transforms with equivalent meaning. Each implementation defines one such transform.

A hardware design language may be treated as a family of closely related languages, each one of which defines a transform from source code into a particular suitable output. These languages, although syntactically similar, generate totally different object code. However, a much better approach would be to treat the language as a single language and the different outputs as target machine object codes which depend on the user's need. Following this approach, a hierarchial implementation of the language is possible. The compilation process can be divided into several stages. At each stage the user will be able to communicate with the system using application-dependent parameters.

1.2.1. The Multistage Compiler

In order to support a variety of applications with minimal duplication of software development, it is important to identify and separate the common aspects of the compilation process from those which
are application-dependent. To this end, the compiler has been partitioned into three stages, as shown in Figure 1.2. Several intermediate representations (outputs of intermediate stages) are possible. Most useful are the ones which minimize the programming effort and are easily understandable. Also, these intermediate outputs must have the same information as the original source code. That is, it should be possible to design a de-compiler to get back an equivalent to the original source code.

Source codes are considered to be equivalent if the input/output behavior of the machine they produce is identical. Thus the statement:

\[ A[0:5], B[7:9] \rightarrow C[0:5], D[3:5] \]

is equivalent to the statement:

\[ A[0:5] \leftarrow C[0:5]; B[7:9] \rightarrow D[3:5] \]

Stage 1 of the compiler decomposes the source text into a quadruple table and other tables to keep track of variables and to help in determining referencing environment. Quadruples generated by the first stage are not like those generated by the compilers of software languages. These quadruples are just a tabular representation of the original AHPL text.

The implementation of Stage 1 takes full advantage of available software. This includes a parser generator program [15]. The BNF [19] description is written in SLR(1) grammar [13]. This grammar is fed into the parser generator program to generate the parser tables. An executive routine calls appropriate routines for semantic actions.
Figure 1.2. Three-stage hardware compiler.
Output of the first stage may be used as the data base of a function-level simulator. The second stage will assign control states and translate the description to a transfer connection list for each distinctively controlled data register or bus. The list will be stored as an Abstract Element Linked List. Again, the linked list will contain all the useful information in the original source code.

The first two stages will more or less be the same for all applications while the last step will be dependent on the application. The third stage will generate the output appropriate to a particular application. It may be a wire list interconnecting MSI parts. It may be an abstract interconnection list suitable for test sequence generation, or it may be a complete design and layout for VLSI implementation [24].

Clearly, the requirements of a wide range of separate applications including those mentioned earlier cannot be accomplished by a single Stage 3. Instead, a separate Stage 3 is written for each application, as illustrated in Figure 1.2. New versions of Stage 3 may be prepared at any time a need for significantly different output becomes apparent.

The advantage of the proposed scheme is obvious. Since the more complex task of syntax recognition and decomposition of the source code into internal form is taken care of by Stages 1 and 2, it will be easier to write a custom-tailored Stage 3. Also, Stages 1 and 2 are free from application details, so they can be made modular and easily modifiable.
1.2.2. User Parameters

A hardware design language is a reflection of design philosophy of its author. AHPL II attempts to organize the design along a well-tested path which is free of hazards. It achieves the objective by restricting the designer's choice of register types, bus types, and clocking options. However, to make the language more universal, some flexibility must be provided. Making the language more flexible increases the possibility of ambiguity and may lead to incorrect hardware generation. For instance, allowing asynchronous transfer in a clock mode circuit may cause it to malfunction. However, a careful designer may use such a feature to his advantage. Furthermore, the language may also be used by test engineers, who must test what already exists. Additional flexibility has been provided for those users who need it, but those who wish to conform to the original AHPL philosophy need not worry about the added features. This flexibility is provided by means of user parameters which isolate the language from those application-dependent features which are not common to all applications. PMS and ISP [22, 23] also allow the use of parameters. However, the usage there is much more general and these parameters specify details which are inherent in AHPL syntax. AHPL hardware parameters are only for circuit details. The user parameter may be specified in two ways. First, a set of parameters enclosed by curly brackets "{}" may be included with each declared set of memory elements, bus, or functional register. This information is passed to Stage 3 after some initial processing. These parameters may, for
example, distinguish between a tristate and a wired-and bus or specify a particular flipflop realization. In addition, user input may be requested by any Stage 3 compiler or function level simulator. Layout guidance might be one example of information provided to Step 3, and this might even be supplied on an interactive basis.

User parameters supplied at subsequent stages serve to isolate the language from application-specific details. These parameters, if included in the main language, would make it bulky, incomprehensible, and expensive.

The mechanism of user parameters will help in interfacing the AHPL compiler to already-existing CAD systems at the place of application.

The method of passing parameters, as discussed above, greatly reduces the need for other application-specific features. New parameters may be defined and common parameters used in different ways by various application CAD systems. In some applications there may be no need for user parameters. The mechanism is powerful, and to some extent, makes it unnecessary to anticipate in advance all features of the various applications CAD systems which will be interfaced to the AHPL compiler.

The modularized approach has simplified the design of all three stages. It reduces the effort to adapt AHPL to a particular application. It relieves the designer of a particular Stage 3 of processing the language itself and allows him to begin with a much easier-to-manage set of tables.
CHAPTER 2

ON EXTENDING AHPL

2.1. Review of AHPL II

Of the hardware description languages currently in use, most may be classified as register transfer languages [1-5]. A register transfer language is characterized by the fact that timing is restricted to discrete clock periods.

Register transfer languages may be divided into three sublevels. On one hand, descriptions in languages such as ISP [2] do not specify details of hardware implementation of either the data structure or the control unit. At the other extreme, RTS III [5] specifies both control and data hardware. In between lies AHPL, which supports detailed specification of data-handling hardware but permits an implied realization of the control unit.

Most register transfer languages describe a machine in terms of storage elements (registers), data paths between these registers, and a set of rules which control the flow of data through this path. Figure 2.1 describes a simple system. If the control switch is in position A then the x register is loaded with signal on Inline, else the y register gets the contents of the x register. Such a system may be described at the register transfer level as:
IF (CNTRL EQ A) THEN X ← INLINE
ELSE Y ← X.

Figure 2.1. A simple system.

One possible hardware realization of the system is shown in Figure 2.2.

Figure 2.2. A hardware realization of the system.
2.1.1. Basic Operations

AHPL resembles APL [16]. It operates on vectors and matrices and, like APL, has a very straightforward "go to" type of control structure. Basic AHPL operators are shown in Table 2.1. Some of the commonly used operations are shown in Table 2.2.

A very complex expression can be written by combining these basic operations:

$$((A, B)! (C, D)) \star (A/E, \overline{A}/E) \rightarrow (F! \text{INC}(G), Z) \star (V/H[1:8], V/H[1:8])$$

The AHPL program has a one-to-one hardware correspondence. That is, each statement or step can unambiguously be translated into hardware.

AHPL statements are executed sequentially. The sequence may be altered by a branch statement or subprogram invocation.

2.1.2. The Design Philosophy

AHPL provides a different and efficient approach to the representation of sequential circuits. Firstly, it concentrates on clock-mode circuits only. Then it divides the network into a control part and a data part, as shown in Figure 2.3. The control section issues signals on a set of control lines, causing register transfers to take place in the data section. Sequencing of control may be influenced by branching information fed back from the data section. The data unit is composed of registers, flipflops, buses, and other elements declared in the declaration section. The control sequencer is automatically built from the timing and branching information given in the sequence part of the module description.
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>And</td>
</tr>
<tr>
<td>V</td>
<td>Or</td>
</tr>
<tr>
<td>®</td>
<td>Eor</td>
</tr>
<tr>
<td>A/</td>
<td>All bits And</td>
</tr>
<tr>
<td>V/</td>
<td>All bits Or</td>
</tr>
<tr>
<td>T</td>
<td>Encode</td>
</tr>
<tr>
<td>BAR</td>
<td>Complement</td>
</tr>
<tr>
<td>+</td>
<td>Transfer</td>
</tr>
<tr>
<td>=</td>
<td>Connection</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Branch</td>
</tr>
<tr>
<td>A</td>
<td>Simple register or memory</td>
</tr>
<tr>
<td>A,B</td>
<td>Column catenation</td>
</tr>
<tr>
<td>A!B</td>
<td>Row catenation</td>
</tr>
<tr>
<td>A[J]</td>
<td>Jth bit (column) of A</td>
</tr>
<tr>
<td>A[M:N]</td>
<td>Bits M thru N of A</td>
</tr>
<tr>
<td>A&lt;J&gt;</td>
<td>Jth row of A</td>
</tr>
<tr>
<td>A&lt;M:N&gt;</td>
<td>Rows M thru N of A</td>
</tr>
</tbody>
</table>
Table 2.2. Basic AHPL operations

<table>
<thead>
<tr>
<th>Transfer and Connection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ← B</td>
<td>Register A is loaded with the contents of register B</td>
</tr>
<tr>
<td>X,Y ← A,B</td>
<td>Multiple source and destination registers</td>
</tr>
<tr>
<td>X ← (A</td>
<td>B)*((f,\overline{f})</td>
</tr>
<tr>
<td>Z = A</td>
<td>Bus connection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>← 10</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>← (f,\overline{f})/(10,20)</td>
<td>Conditional branch</td>
</tr>
<tr>
<td>← (10,15,20)</td>
<td>Multiple branch/diverge for parallel execution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subprogram Invocation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ← INC (x)</td>
<td>New value of X is the result of performing the &quot;INC&quot; operation on X</td>
</tr>
</tbody>
</table>
2.1.3. Modularity and Task Subdivision

A large task such as the CPU of a Parallel Processor, or a microprocessor-based system using several DMAs, I/O controllers and other independent subsystems can be subdivided into smaller tasks by taking advantage of the facility of intermodule communication. Communication between different modules is established by means of input/output lines and buses. This helps to define sophisticated synchronous and asynchronous handshake protocols among modules of a system.

AHPL also provides a way to describe combinational logic units such as adders, incrementers, decoders, etc. These units can be designed separately from the module which is going to use it. A module can invoke a CLunit in a simple transfer or connection statement. The approach is similar to FORTRAN functions, with the necessary restrictions for hardware realization. An example of the Combinational logic unit description may be found in Chapter 4.
Modules are the highest form of intelligence, and so are entitled to self-rule and some privacy. They can communicate with each other only through proper I/O channels. Regardless of its intelligence or authority, one module cannot look into the internal registers or buses of another module. This allows for proper partitioning and easy diagnosis in case of failure.

AHPL is a powerful design tool. Systems ranging from simple combinational logic to highly structured super computers and parallel processors can be described in AHPL with little effort. The language forces the designer to design well-organized systems. It employs a rigid and safe design rule. For this reason, circuits generated by AHPL are, to a great extent, free from timing problems. However, this rigidness is rather unpractical and limits the scope of application of AHPL.

2.2. Some Limitations and Apparent Limitations of AHPL II

The rigidity of AHPL II is both its source of strength and weakness. On one hand, it helps the compiler to generate efficient, fast and safe hardware. But on the other hand, it cannot accommodate variations in design practices. Shortcomings of AHPL can be grouped into four major categories:

1. It is not open-ended.
2. Implementation details cannot be specified. Only hardware translation to bipolar logic family has been demonstrated [25].
3. It lacks flexibility offered by most software languages.
4. It follows a very strict design rule.

Ways to overcome these limitations are discussed below.

2.2.1. Open-Ended Language

One desirable feature of a general-purpose hardware description language would be to support descriptions at several levels. Such a language may start from a very rudimentary level where basic symbols of the language are defined. These symbols may be basic operators and data types. More abstract constructs may be defined using this 'base level' language. As need for more refined abstraction arises, higher-level languages may be defined in terms of previously defined lower-level languages. Thus a tree-like structure would emerge, as depicted in Figure 2.5.

![Diagram of hierarchy of description languages]

Figure 2.5. Hierarchy of description languages.

Such a language will have the following advantages:

1. Universal communication at all levels of design automation.
2. Richness of constructs.
3. Logical and well-defined relationship between several levels of abstraction.

4. Analysis at different lower levels from the same source program.

This is the approach of the language CONLAN which has been under development for the past 6 years [8]. This issue will not be addressed by the Universal AHPL.

2.2.2. AHPL's Association with a Logic Family

When AHPL was in its development phase, bipolar logic was the most commonly available logic. AHPL subconsciously adopted notions which were most useful for clocked bipolar logic. This is not to say that AHPL is not useful for other logic families. The language itself is more or less technology-independent, but the relationship between its constructs and final hardware were based on bipolar logic. This relationship must be reexamined. AHPL language constructs must be expressed in terms which are independent of technology. Means to specify implementation details not given by the language constructs must be provided. Implementation details for a bipolar circuit would be quite different from that of a MOS VLSI.

2.2.3. Formal Language Considerations

This topic needs a very detailed and careful analysis. Such an analysis is beyond the scope of this research. However, several aspects of the language from the software point of view will be discussed very briefly.

AHPL is based on APL. APL itself is a very unusual language. It is praised for its brevity and blamed for its obscurity. Celebrated
one-liners usually need a two-page explanation. Like APL, AHPL too is a very expressive language with its powerful vector-oriented constructs. However, unlike APL, AHPL programs are self-documenting and very clear. AHPL imposes restrictions which would seem to be arbitrary. It allows constructs which may seem to be redundant. Its typing, its scope rule, and its procedure mechanism are all unique and can bewilder any software designer. AHPL is a hardware programming language, and it gives up software generalities in order to generate efficient and reliable hardware. The software structure of AHPL will be analyzed in eleven areas. Improvements will be suggested wherever necessary.

**AHPL Procedure Mechanism.** AHPL is very different from software programming languages in its treatment of procedures. It classifies procedures according to their intelligence, a module being more intelligent than a CLU. All modules are autonomous and they cannot be invoked. A CLU is meaningful only when invoked by a module, either directly or indirectly through another CLU. Parameters are passed by name. The syntax of module description is different from that of a CLU description. Similarly, the semantics of the two types of procedure are not the same. The AHPL procedure mechanism may not appeal to a software designer, but one should remember that AHPL is a hardware language. The AHPL program partitions a large system into independent autonomous modules. These modules can communicate only through proper channels. Their operation must be explicitly synchronized. If module invocation were to be allowed, the semantics of the language would have
to be changed significantly. The scope rule would have to be modified; a step would no longer be executable in a single clock period; implicit synchronization of modules would be necessary. Such a change will not be useful. The present scheme of intermodule communication is a powerful one. Different sections of a module can be activated or deactivated using I/O lines. All modules are simultaneously active like parallel processes. Such parallelism is hard to find in ordinary programming languages.

CLUs perform logical operations only. They have no storage elements and no sequential control. Assuming gate delays to be much smaller than the clock period, any number of CLUs can be executed during a single clock period. Thus invocation of a CLU does not cause any timing problem in a well-designed system. This explains why a CLU can be invoked while a module cannot be invoked. The prime reason for allowing CLU is to facilitate the description of large iterative combinational logic blocks, such as Adders and Multiplexers. The syntax of CLU is very helpful in describing such a block.

A device which has storage elements and logic gates but no sequential control can also be executed in a single clock period. Thus it can be invoked by a module without causing any timing problems. Universal AHPL introduces the device as FUNCTIONAL REGISTER. This concept will enable the AHPL user to develop even more structured and organized design.

Explicit Naming of Modules. Like most programming languages, AHPL requires that all modules be named and defined explicitly. It
does not have the facility to generate copies of similar modules implicitly from a single description. Such a facility would help in defining large parallel processors and other such systems in an elegant and concise manner. Research in this direction is in progress [7]. Future implementations of AHPL may incorporate this facility.

**AHPL Scope Rule.** AHPL uses static scope rule [9]. The referencing environment is determined by the type of the variable. Thus Memory, Buses and CLU are local; Inputs and Outputs are semilocal; and Exinput and Exbuses are global [10]. Though seemingly artificial, the scope rule helps AHPL to achieve proper hardware partitioning and to establish well-defined communication protocols among modules of a system. Many languages are block-structured. A subprogram in an inner block may refer to data elements in the outer block. If AHPL allowed this, it would be possible for a module to look into internal registers of an enclosing module and to alter its contents. Such a module may be useful for testing purposes. However, the same result can easily be achieved by adding extra I/O lines. The change in scope rule would mean extensive semantic changes in the language. In particular, modules would no longer be autonomous. For this reason, the universal AHPL will not attempt to change the scope rule.

**Data State Types.** AHPL supports binary and integer data types (values). Integers are used for indexing, dimensioning, and to specify control state numbers. Actual circuit elements can only take a binary value. Naturally, within a module a data element will always be either
0 or 1. Tristate buses must be allowed in the circuit realization, but the high impedance value is not represented in AHPL.

**Hardware Data Type.** Most software languages use typing to restrict the choice of values of a variable. Such typing often leads to more efficient implementation in terms of memory management and access time. More advanced languages, like PASCAL [11] even allow user-defined types. The AHPL data type, however, is based on hardware. Thus an element may be a storage element or memory, a bus, an input line and so on. Regardless of its type, an element can take only binary values. AHPL uses type for two purposes: one, to define the proper referencing environment and two, to check certain kinds of errors. For instance, an element declared as input must not occur on LHS of a replacement statement. Thus, hardware type is a useful concept but it should not be confused with general software usage of the word.

**Control Structure.** AHPL allows only 'go to' type of control statements in a module description.

\[ (+n_1, n_2 \ldots n_m) \]

\[ (+a_1, a_2 \ldots a_n)/(n_1, n_2 \ldots n_n) \]

The first statement is an unconditional branch statement like Go To \((n_1, n_2 \ldots n_m)\). The second one is conditional branch like if \(a_1\) go to \(n_1\); if \(a_2\) then go to \(n_2\) \ldots if \(a_n\) then go to \(n_n\). The familiar If-Then-Else or For or Do Loop type of control is not available. The reason for this is AHPL's insistence that a module description must be translatable into hardware on a one-to-one basis.
An elaborate control structure would necessarily need implicit hardware. For instance a FOR type of control statement would imply a counter, but not indicate how to connect one. CLU description, however, allows a more sophisticated control structure. The reason is that CLU control variables are only used for loop counting and indexing by the Compiler. They are not translated directly into hardware. Old AHPL allowed APL-type control statements. But Algol-type control statements seem to be more clear. Universal AHPL, therefore, allows IF-then-Else and FOR constructs in a CLU description.

**Array Dimension and Indexing.** AHPL arrays cannot be more than two-dimensional. To allow for more than two dimensions, major changes in AHPL semantics would be necessary. Universal AHPL generalizes the meaning of some of the AHPL operators so that multidimensional arrays may be added later. Available memory modules are usually organized as a one- or two-dimensional array. Therefore, at present, there is no need to include higher-dimensional constructs.

AHPL allows only 0-origin indexing, with bit 0 being the most significant bit. Although 0-origin indexing is a common hardware practice, opinions on bit 0 being the MSB differ. The reason for this difference is centuries old, when the Arabic numerals were grafted into English language without making proper modifications. Arabic is written right to left so that the digits are weighed accordingly, with left-most digit being the most significant one. This convention has been retained for English numbers also, although the language itself is
written left to right. Following the number convention, the left-
most bit of a register is called the most significant bit. In numbering
bits of a register, AHPL follows the left-to-right writing convention;
thus \( \theta \) became the most significant bit. On the other hand, some design-
ers followed the convention (right-to-left) for writing numbers, and
for them bit \( \theta \) is the least significant bit. Preference of one conven-
tion over another is simply a matter of personal taste. It may seem
that allowing both conventions would be better. However, a careful
analysis would reveal that such intermixing of conventions may lead to
ambiguities. Similarly, nonzero-origin indexing would also cause
ambiguities. In hardware, register bits and memory locations, always
start from \( \theta \); so the indexing convention of AHPL needs no modification.

**Limited Set of Operators.** AHPL allows only vector and boolean
operators. For indexing, dimensioning and CLU loop control, arithmetic
operators are very desirable. Universal AHPL permits the use of arith-
metic operators for these purposes.

**Multiple Meaning of Some Operators.** Square and angular brackets
are used for dimensioning as well as indexing. A \([15]\) in the declara-
tion section defines variable \( A \) to be a 15-bit vector. In other parts
of the program, the same statement would mean 16th bit (0-origin
indexing) of \( A \). This does not cause any confusion because the declara-
tion section is distinctively apart from the rest of the description.
Another operator which has two meanings is the conditional operator \( '\#' \).
On the left-hand side of a transfer statement it controls the selection
of destination variables, whereas on the right-hand side it controls the source selection. Thus the expression

\[(A!B)\ast (a,b) \leftrightarrow (C!D)\ast (c,d)\]

is functionally the same as

\[A \leftrightarrow (C!D/o,o,..o/!A)\ast (a\&c,a\&d,a\&c\&d,\bar{a});\]

\[B \leftrightarrow (C!D/o,o,..o/!B)\ast (b\&c,b\&d,b\&c\&d,\bar{b}).\]

If both source control variables (c and d) are zero, then the source equation becomes zero, and zero may be loaded into the destination variables. However, if both destination control variables (a and b) become zero, the net effect is as if the statement was not executed. Left-hand-side asterisks controls enable input of destination vectors; so if LHS conditionals are zero, nothing is loaded into the destination vectors. It is felt that the dual use of \(\ast\) will not cause any confusion to a design engineer.

Arithmetic expressions in Universal AHPL may use five symbols: \(\dagger, /, \ast, +, -\). Four of these symbols are used in the language for other purposes, also. However, an arithmetic expression is permitted only for indexing, dimensioning and loop control. Boolean and vector operators are not permitted for these purposes, hence no confusion arises.

**Multiple Operators for Similar Operation.** Two types of replacement are allowed: transfer and connection. From the hardware point of view, it is necessary to differentiate between the two types of replacement statements. Destination of a transfer statement is a storage
element whereas that of a connection statement is a non-storage element. It is possible to use one generic operator for both types of statements. Proper interpretation can be made by looking at the type of the destination variable. However, use of a separate operator has the following advantages:

1. It improves the readability.
2. It emphasizes that the two types of replacement statements are different.
3. It helps in error-checking.

2.2.4. Hardware Flexibility

AHPL forces the user to design his system using a very small subset of building blocks. Also, it assumes that all the registers of a module are driven by a single clock edge. The first restriction only causes some inconvenience or inefficiency whereas the second restriction makes it impossible to design and simulate MOS circuits. Universal AHPL removes both restrictions. It allows specification of clocking options and element types as parameters. Universal AHPL also allows asynchronous transfer and set and reset operations.

2.3. Summary of Added Features

Features necessary to make AHPL more universal have been discussed in the previous section. AHPL with these features is defined as Universal AHPL. For a quick reference, a summary of the features which will be added is given below:
1. Declaration statement may have parameters. These parameters may specify the name of the driving clock for a memory element. They also specify subtype of the variable; for instance, whether it is a D-flipflop or a latch, OR bus or a Tristate bus. In absence of parameters, default type is consistent with AHPL of reference [6].

2. Option of user-specified node number for declared variables. This will allow assignment of Pin numbers to Input Output lines and external buses.

3. Underscore is accepted as a letter. This will improve program readability. For instance, PROGRAM_COUNTER is more readable than PROGRAMCOUNTER.

4. The following replacement operators are permitted:

   <= clocked transfer
   < unclocked transfer (D latches)
   <S- unclocked set invocation
   <R- unclocked reset invocation
   = logic or bus connection
   := bidirectional connection of busses

5. Arithmetic expression allowed for indexing, dimensioning and loop control.

6. If then Else and FOR constructs are the bases for the approach to CLU description.

7. Functional registers are included.
8. Multiple clocks allowed, but only one clock will drive the control sequence.

9. Always-active global asynchronous branches (reset) included.

10. Variable dimension CLU and functional registers allowed.

11. '?' allowed in bitstring to specify don't care.

12. '??' allowed as argument to specify no connection.

13. Multiple invocation of a CLU would imply multiple copy of the CLU whereas multiple invocation of a functional register would generate implied busing for a single copy.

14. Parameters to specify implementation details are permitted. These parameters are passed directly to stage 3.

15. Facility to include primitive functions.
CHAPTER 3

SPECIFICATION OF UNIVERSAL AHPL

A language provides a means to convey ideas using phrases. The meaning of a phrase depends on its syntactic structure and on the meaning of its constituents [12]. The syntax of a context-free language [13] may be specified by a grammar having a finite set of rules. Grammar of a context-free language is a quadruple [12] G where:

\[ G = (V, T, P, S) \]

- \( V \) is the finite non-empty vocabulary
- \( T \subseteq V \) is the terminal alphabet
- \( S \in (V-T) \) is the axiom
- \( P \) is the finite non-empty set of grammar rule called productions. It has the form \( A \rightarrow B \) for \( A \in (V-T) \) and \( B \in V^* \)

AHPL syntax has been rigorously defined in Backus-Naur form [19]. The first section of this chapter will deal with the meaning of the terminal alphabet of the AHPL grammar. The complete grammar will be presented in the second section. The meaning of productions of the grammar will be discussed in the third section.

3.1. Terminal Symbols

Terminal symbols are used as operators, delimiters, and data elements. Operators are used to perform scalar or vector operations on
data elements. Delimiters are used to signal the end of a clause, phrase, sentence, or a paragraph, etc. The meaning of operators and delimiters will become clear on studying the syntax and semantics of the language. In this section meaning of data elements will be given.

AHPL data elements may be classified into four major categories: (1) Buses and Exbuses; (2) Memory Elements; (3) Input Output lines; (4) Submodules—function registers and combinational logic units.

3.1.1. Buses and Exbuses

a. Definition of a Bus element.

i) A bus element has a list of (data bit, enable line) pairs as inputs.

ii) It has one and only one output.

iii) If only one enable line is 1 then the output will correspond to the data bit of the (data bit, enable line) pair.

iv) If all enables are zero, the output can only be inferred from hardware parameters (see below) which specify the type of the bus element.

v) If more than one enable is 1, then the output can only be determined by using the hardware parameters (see below) in conjunction with corresponding data bits.

b. Declared and Implicit Buses

i) A declared bus is a named vector of bus bits.

ii) A bus may be generated internally by the compiler; these are called implicit buses.
iii) Contents of declared buses may be explicitly manipulated by the user. Implicit buses, on the other hand, are not directly accessible.

iv) A bus declaration yields a bus local to a module, whereas an EXBUS declaration yields a bus which interconnects modules and the outside world.

v) The realization of a bus can be dictated by a hardware parameter which is passed directly to stage 3.

c. Treatment of Special Cases

i) Pair (data=0, enable) is deleted from the Input pair list of an implicit bus, or a declared AND/OR type bus. The pair is passed to stage 3 for other type of buses.

ii) Pair (data=1, enable) is replaced by enable on an implicit or AND/OR type bus. The pair is passed to stage 3 for other type of buses.

iii) If only one pair remains in a bus bit, it is treated as normal for EXBUS. However, for internal buses it is replaced by AND gate.

iv) If no pair remains, the bus is replaced by logical 0.

3.1.2. Memory Elements

a. Definition of Memory Elements

i) Two types of memory elements are allowed: (1) D-type synchronous flipflop; and (2) D-type asynchronous latches. Other types of memory elements may be included later by modification of the stage 2 compiler, or by defining them
in terms of D flipflops as functional registers. A J-K flipflop is easily expressed as a functional register.

ii) A synchronous memory element has five inputs:
   a) Data
   b) Clock
   c) Enable
   d) Set
   e) Reset

iii) An asynchronous memory element has four inputs:
   a) Data
   b) Enable
   c) Set
   d) Reset

iv) A memory element has one and only one output.

b. Declaration of Memory Elements

i) A memory element must be explicitly declared. No implicit memory element is generated. Several memory elements may be grouped together and declared as a single vector or matrix.

ii) Memory elements are local to the module where they are declared.

iii) A driving clock may be optionally specified in the memory declaration of the synchronous memory element. This clock will be connected to the clock input of all the elements of the declared vector (or matrix). In the absence of the
clock specification, the system clock will be the driving clock.

iv) Realization of a memory element can be dictated by a parameter which is passed directly to stage 3.

3.1.3. Input/Output Lines

a. Definition

i) Input lines are wires coming from other modules to the module in which they are declared as INPUTS. If the wires are coming from the outside world it is declared as EXINPUT.

ii) A module must not use input lines as a destination of a transfer or connection statement.

iii) Input lines may be used as arguments for CLU or FNREG invocation, but the user must make sure that such invocation would not change the state of these lines.

iv) Output lines are wires going out of the module to other modules or to the outside world.

v) The characteristics of output lines are similar to those of AND/OR buses.

b. Declaration

i) Input Output lines must explicitly be declared. Several lines may be grouped together and declared as a single vector.

ii) The output of one module may be declared as input to several modules.
iii) The output of one module must not be declared as output of any other module.

iv) Several modules in a system may refer to the same EXINPUT. An EXINPUT must not be declared as output of any module.

v) Each input declaration of a module must correspond to an output declaration of another module.

3.1.4. CLU and FNREG

a. Definition

i) CLU and FNREG are invocable submodules. They are not autonomous and are activated only when invoked by a module, either directly or indirectly.

ii) CLU consist only of combinational logic whereas FNREG has memory elements also. Neither of the two can include a control sequence.

iii) CLU and FNREG are completely executed at the point of invocation. CLU does not retain any information regarding the previous execution. The contents of a functional register, on the other hand, may change as a result of an invocation.

iv) Each CLU invocation statement generates a new copy of the unit. Copies with identical arguments may be merged together by an optimizer program.

v) Only one FNREG is generated for each declaration. The compiler generates implied buses if the same FNREG is invoked more than once.
b. Declaration

i) FNREG and CLU are local to the module in which they are declared.

ii) A driving clock may be optionally specified for the functional register. In the absence of a clock specification the system clock will be the driving clock.

iii) Declaration is a mapping between local name and generic name. Same generic submodules may be used by more than one module. However, a generic description is used only as a template to generate local submodules.

iv) In the absence of a description for the generic submodule, a black box will be connected.

3.2. Syntax

AHPL syntax is given below.
Grammar

1.01 $S^*$ ::= ! - $S$ - !

2.01 $S$ ::= <AHPLPROGRAM> .

3.01 <AHPLPROGRAM> ::= <AHPLPROGRAM> . <DESCRIPTIONS>
3.02 ::=: <DESCRIPTIONS>

4.01 <DESCRIPTIONS> ::= <MODULEDESC>
4.02 ::= <CLUDESC>
4.03 ::= <FNREGDESC>

5.01 <MODULEDESC> ::= <MODHEAD> . <MODDECLS> . <MODSEQ>

6.01 <CLUDESC> ::= <CLUHEAD> . <CLUDECLS> . BODY <CLUACTS> . END

7.01 <FNREGDESC> ::= <FNHEAD> . <MODDECLS> . BODY <RELATION> . END

8.01 <MODHEAD> ::= MODULE : ID

9.01 <MODDECLS> ::= <MODDECLS> . <МОDECL>
9.02 ::= <MODDECL>

10.01 <MODSEQ> ::= BODY SEQUENCE : <SLRM> . <PROCPART> . ENDSEQUENCE <NOPROC> . END

11.01 <МОDECL> ::= <TYPE1> : <ID_DIM_LIST> <REF1>
11.02 ::= <TYPE1> : <ID_DIM_LIST>
11.03 ::= <TYPE2> : <ID_DIM_LIST> <REF2>
11.04 ::= <TYPE2> : <ID_DIM_LIST>
11.05 ::= PINS : <PIN_LIST>
11.06 ::= LABELS : <LABEL_LIST>

12.01 <TYPE1> ::= BUSES
12.02 ::= EXBUSES
12.03 ::= EXINPUTS
12.04 ::= INPUTS
12.05 ::= MEMORY
12.06 ::= OUTPUTS
12.07 ::= PULSES

13.01 <ID_DIM_LIST> ::= <ID_DIM_LIST> ; <ID_DIM>
13.02 ::= <ID_DIM>
14.01 <REF1> ::= <. <PAR_LIST> .>
15.01 <TYPE2> ::= CLUNITS
15.02 ::= FNREGS
16.01 <REF2> ::= <. ID <REF1> <DIMENSION>
16.02 ::= <. ID <REF1>
16.03 ::= <. ID <DIMENSION>
16.04 ::= <. ID
17.01 <PIN_LIST> ::= <PIN_LIST> ; <PIN_NUM>
17.02 ::= <PIN_NUM>
18.01 <LABEL_LIST> ::= <LABEL_LIST> ; <LABID>
18.02 ::= <LABID>
19.01 <ID_DIM> ::= ID <DIMENSION>
19.02 ::= ID
20.01 <DIMENSION> ::= <. <AE> > [ <AE> ]
20.02 ::= [ <AE> ] <. <AE> >
20.03 ::= <. <AE> >
20.04 ::= [ <AE> ]
21.01 <AE> ::= <EXPR>
22.01 <EXPR> ::= <EXPR> + <TERM>
22.02 ::= <EXPR> - <TERM>
22.03 ::= <TERM>
23.01 <TERM> ::= <TERM> * <FACTOR>
23.02 ::= <TERM> / <FACTOR>
23.03 ::= <FACTOR>
24.01 <FACTOR> ::= <FACTOR> ^ <PRIMARY>
24.02 ::= <PRIMARY>
25.01 <PRIMARY> ::= ( <EXPR> )
25.02 ::= INTEGER
25.03 ::= ID
25.04 ::= - <PRIMARY>
25.05 ::= + <PRIMARY>
26.01 <PAR_LIST> ::= <PAR_LIST> ; <PARAM>
26.02 ::= <PARAM>
27.01 <PARAM> ::= <AE>
26.01 <LABID> ::= ID = <SLRM>
29.01 <SLRM> ::= ID <SUBS_RANGE>
29.02   ::= ID

30.01 <PIN_NUM> ::= ID ( <NUMB_STRING> )

31.01 <NUMB_STRING> ::= ( <NUMB_STRING> )
31.02     ::= <NUMB_STRING> , INTEGER
31.03     ::= <NUMB_STRING> , ?
31.04     ::= INTEGER
31.05     ::= ?

32.01 <PROCPART> ::= <PROCPART> . INTEGER <STEPS>
32.02     ::= INTEGER <STEPS>

33.01 <NOPROC> ::= <STARTSTEP> ; <RELATION>
33.02     ::= <STARTSTEP>

34.01 <STEPS> ::= NODELAY <ACTION>
34.02     ::= <ACTION>
34.03     ::= NULL
34.04     ::= DEADEND

35.01 <ACTION> ::= <RELATION> ; <BRANCH>
35.02     ::= <RELATION>
35.03     ::= <BRANCH>

36.01 <RELATION> ::= <RELATION> ; <RELATION1>
36.02     ::= <RELATION1>

37.01 <BRANCH> ::= => <GLRM> / <NUMB_STRING>
37.02     ::= => <NUMB_STRING>

38.01 <RELATION1> ::= <INVOCATION>
38.02     ::= <TRANSFER>
38.03     ::= <CONNECTION>

39.01 <INVOCATION> ::= ID ( <INVOK_LIST> ) * <BGLRM> <=
39.02     ::= ID ( <INVOK_LIST> ) <=

40.01 <TRANSFER> ::= <SYNCTR>
40.02     ::= <ASYNCTR>

41.01 <CONNECTION> ::= <DLRM> = <GLRM>
41.02     ::= <DLRM> ::= <DLRM>
41.03     ::= <DLRM> ::= <CLHS>

42.01 <INVOK_LIST> ::= <INVOK_LIST> ; <CGLRM>
42.02     ::= <CGLRM>
43.01 <BGLRM> ::= <BGLRM> ! <GLRM1>
43.02  ::= <GLRM1>

44.01 <CGLRM> ::= <BGLRM>
44.02  ::= ??

45.01 <SYNCTR> ::= <DLRM> <= <GLRM>
45.02  ::= <CLHS> <= <GLRM>

46.01 <ASYNCTR> ::= <DLRM> <= <GLRM>
46.02  ::= <CLHS> <= <GLRM>
46.03  ::= <DLRM> <S> <GLRM>
46.04  ::= <DLRM> <R> <GLRM>

47.01 <DLRM> ::= <DLRM> ! <DLRM1>
47.02  ::= <DLRM1>

48.01 <GLRM> ::= <BGLRM> * <BGLRM>
48.02  ::= <BGLRM>

49.01 <CLHS> ::= <DLRM> * <BGLRM>
49.02  ::= <CLHS1>

50.01 <CLHS1> ::= ( <CLHS> )

51.01 <DLRM1> ::= <DLRM1> ; <DLRM2>
51.02  ::= <DLRM2>

52.01 <DLRM2> ::= ( <DLRM> )
52.02  ::= <SLRM>

53.01 <GLRM1> ::= <GLRM1> ; <GLRM2>
53.02  ::= <GLRM2>

54.01 <GLRM2> ::= <GLRM2> @ <GLRM3>
54.02  ::= <GLRM3>

55.01 <GLRM3> ::= +< <GLRM3>
55.02  ::= <GLRM4>

56.01 <GLRM4> ::= <GLRM4> + <GLRM5>
56.02  ::= <GLRM5>

57.01 <GLRM5> ::= 5/ <GLRM5>
57.02  ::= <GLRM6>

58.01 <GLRM6> ::= <GLRM6> & <GLRM7>
58.02  ::= <GLRM7>

59.01 <GLRM7> ::= ^ <ELEMENT>
59.02  ::= <ELEMENT>
60.01 <ELEMENT> ::= ID ( <INVOK_LIST> )
60.02 ::= ID <SUBS_RANGE> ( <INVOK_LIST> )
60.03 ::= INTEGER $ INTEGER
60.04 ::= \ <NUMB_STRING> \ 
60.05 ::= ( <BGLRM> )
60.06 ::= <SLRM>

61.01 <SUBS_RANGE> ::= < <RANGE> > [ <RANGE> ]
61.02 ::= [ <RANGE> ] < <RANGE> >
61.03 ::= < <RANGE> >
61.04 ::= [ <RANGE> ]

62.01 <RANGE> ::= <AE> : <AE>
62.02 ::= <AE>

63.01 <STARTSTEP> ::= CONTROLRESET ( <GLRM> ) /
63.02 ::= CONTROLRESET ( <NUMB_STRING> )

64.01 <CLUHEAD> ::= CLU : ID ( <INVOK2_LIST> ) <REF1>
64.02 ::= CLU : ID ( <INVOK2_LIST> )

65.01 <CLUDECLS> ::= <CLUDECLS> . <CLUDECL>
65.02 ::= <CLUDECL>

66.01 <CLUACTS> ::= <CLUACT2>

67.01 <INVOK2_LIST> ::= <INVOK2_LIST> ; ID
67.02 ::= ID

68.01 <CLUDECL> ::= INPUTS : <ID_DIM_LIST>
68.02 ::= OUTPUTS : <ID_DIM_LIST>
68.03 ::= CLUNITS : <ID_DIM_LIST>
68.04 ::= CLUNITS : <ID_DIM_LIST> <REF2>
68.05 ::= CTERMS : <ID_DIM_LIST>

69.01 <CLUACT2> ::= <CLUACT2> ; <CLUACT>
69.02 ::= <CLUACT>

70.01 <CLUACT> ::= <CONNECTION>
70.02 ::= <IFSTAT>
70.03 ::= <FORSTAT>

71.01 <IFSTAT> ::= IF <CLUREL> <THEN_CLAUS>
71.02 ::= IF <CLUREL> <THEN_CLAUS> FI
72.01 <FORSTAT> ::= <FORHEAD> = <AE> TO <AE> STEP <AE>
                CONSTRUCT <CLUACTS> ROF
72.02       ::= <FORHEAD> = <AE> TO <AE> CONSTRUCT
                <CLUACTS> ROF
73.01 <CLUREL> ::= ID <RELOP> <AE>
74.01 <THEN_CLAUS> ::= THEN <CLUACTS>
75.01 <ELSE_CLAUS> ::= ELSE <CLUACTS>
76.01 <FORHEAD> ::= FOR ID
77.01 <RELOP> ::= =
77.02       ::= <
77.03       ::=>
77.04       ::=<> 
77.05       ::=>< 
77.06       ::=>=
78.01 <FNHEAD> ::= FREG : ID ( <INVOK2_LIST> ) <REF1>
78.02       ::= FREG : ID ( <INVOK2_LIST> )
3.3. Semantics

The semantics of the Universal AHPL can be best understood by an analysis of its BNF. A brief analysis of each production is given below.

**Productions 1 to 4:** A system (AHPL program) is an unnamed list of modules, CLU units and functional registers.

**Productions 5 to 7:** A module description has three parts—a header part, a declaration part, and a sequence part. Similarly, CLU and Functional Register descriptions have three parts—header, declaration, and CLUacts or relation.

**Production 8:** A module header specifies the name of the module. Every module in a system must have a unique name.

**Production 9:** The declaration part of a module may have any number of declaration statements, each separated by a period.

**Production 10:** A sequence has two parts. Statements appearing before the keyword ENDSEQUENCE belong to the "procedural" part, while those appearing after the keyword belong to the "non-procedural" or "always active" part. The first statement in a sequence is the clock declaration. For example, the statement SEQUENCE:Phi declares Phi as the master clock which controls activities of the module. Phi will drive all the control sequence flipflops and also those data flipflops which do not have an explicit clock declaration.

**Productions 11 to 19:** A declaration statement has three parts. It specifies type, dimension and, optionally, parameters of a data
element (variable). If there are two or more variables of the same type and having identical parameters, they may be grouped together in a single declaration statement. These variables are separated by semicolons. A period terminates the statement. Type specifiers PIN, LABEL and PULSE do not generate new circuit elements; they just assist the compiler in making proper wiring decisions. These statements may be called compiler directives.

Two types of parameters are allowed. The first type (Prod 14) is simply a list of identifiers and arithmetic expressions separated by semicolons. This type of parameter is used with Type 1 (Prod 12) elements. The use and interpretation of parameters is application-dependent. One such use may be to specify subtype, clocking option, and implementation detail of a flipflop. For example:

\[
\text{MEMORY: } A[16]; B[8] \{\text{DFF; Phasel; n}\}.
\]

where DFF is the subtype, Phasel is the driving clock, and n is an integer to be used by stage 3 for application-dependent details.

Type 2 (Prod 16) parameter specifier gives the generic name of the submodules. This is used with type 2 (Prod 15) elements. Along with the generic name, it can optionally specify type 1 parameters and the number of submodules used. For instance, the statement:

\[
\text{FNREG: } A[16] <\text{:JKFF\{Phl\}[16]}
\]

specifies that A is a 16-bit function register composed of 16 JKFF. Phl may be interpreted as the driving clock of JKFF.
If no generic name is given or it is not defined in the program, the submodule is treated as a black box.

**Production 20:** Dimension specifies the number of bits of the declared identifier. 0-origin indexing is used. Column dimension is enclosed in angular brackets, and row dimension in square brackets. The arithmetic expression (Prod 21) may be used to specify the dimension. In case of a module description, these expressions must not contain any variable. However, variables are permitted to satisfy variable-size arrays in CLU and FNREG descriptions.

**Productions 21 to 25:** An expression may be a simple operand (ID or integer), or an operand preceded by a monadic operator, or an expression enclosed in parenthesis, or two expressions separated by a dyadic operator.

Evaluation of an expression is based on usual precedence of arithmetic operator and is done left-to-right. For example, 10*3/4-6-2+1=0. Arithmetic expression is allowed only for indexing, dimensioning, loop control and If statements.

**Productions 26 and 27:** See production 14 and explanation of Productions 15 to 19.

**Production 28:** Portions of a variable may be renamed using a label statement. A label acts just like any other identifier, except that it must not be re-labelled. Examples:

**LABELS:** ADDL = ADDRESS [8:15]  
ADDH = ADDRESS [0:7]
Bits 0 through 7 of ADDRESS are redefined as ADDH. Similarly, bits 8 through 15 are redefined as ADDL.

Production 29: SLRM, an acronym for Simple Line Register or Memory, is the most fundamental data element of the language. Any declared variable is a SLRM. In an expression, if the variable name is used without any subscript (Prod 29.01), then all the bits of the variable are used. Subs-range (Prod 61) may be specified to use only a portion of the variable.

Production 30: See Prod 11.05 and 17.01. This construct allows the user to specify node numbers for declared elements. This is useful for specifying pin numbers for external I/O lines.

Production 31: A string of integers is formed by separating several integers by commas. '?' specifies don't care. Parentheses may be used for clarity.

Production 32: The procedural part of a module description may have several statements called step. Each step begins with a step number followed by details regarding the task to be performed by the step. Each step is assigned a control state level (CSL). The module is said to be in a particular step when the corresponding CSL is active.

Production 33: The non-procedural or always active part begins with global reset statement (Prod 63) followed by zero or more Relations (Prod 36). Statements in this section are always active. Thus $x = Y$ will mean that the values of $x$ will always be the same as that of $Y$. In the case of transfers like $A \leftarrow B$, the register $A$ will get the contents of $B$ every time the driving clock of $A$ goes from high to low.
Production 34: A regular step gives timing and action information. A null step is simply the one which does nothing. It is used for synchronization when the designer feels that the activities initiated in the previous step will take more than one clock period. A DEADEND step is the one from where the control signal cannot go to any other step in the circuit. It is used as a loop terminator.

If the timing part of a step is 'NODELAY' then the corresponding control state level becomes active at the same time as the previous one. CS flipflop is not generated for a NODELAY step.

Production 35: Action may have a Relation part or a Branch part, or both.

Production 36: The Relation part may have one or more statements (Prod 38), each separated by a semicolon. All the statements are simultaneously active.

Production 37: A branch may be conditional or unconditional. An unconditional branch to more than one step would always result in parallel loops. It is desirable to make such parallel loops, wherever possible, to speed up the system. If more than one condition is true, then parallel loops will be formed. If no branch is specified or none of the conditions are true, then the next step in the sequence would be executed.

Example 1:

+(10, 20, 30)

Go to step 10, 20, and 30 simultaneously (parallel executions).
Example 2:

\[ \rightarrow(x, y, z)/(10, 20, 30) \]

x, y, and z may be any boolean expression. If x is true, then go to 10; if y is true, then go to 20; if z is true, then go to 30.

Example 3:

10

\[ \rightarrow(x)/(5) \]

20

: 

If x is true, then go to step 5, or else go to step 20.

The operation of parallel loops must be properly synchronized, otherwise circuit behavior will be unpredictable.

Branch from a nodelay step to itself or to any previous nodelay step without an intervening ordinary step is not permitted.

Production 38: Three types of Relation statements are permitted: Invocation, Transfer, and Connection.

Production 39: This type of statement is used to invoke a functional register. The invocation may be conditional (Prod 39.01) or unconditional. The general statement is of the form:

local name (List of Arguments) * Boolean Expression <=

The binding between local name and generic name takes place according to the declaration. Each bit of an argument is connected to the
corresponding input bit of the functional register. If the statement is in the procedural part, then the corresponding CSL is used to control the connection between formal and actual arguments. If the same functional register is invoked in more than one step, an implied busing of actual arguments would result. The optional conditional controls the clocking of the functional register. If the condition is false, clocking will not take place. In this case, the contents of the function register would not be altered.

**Production 40:** A transfer may be synchronous (Prod 45) or asynchronous (Prod 46).

**Production 41:** A connection may be unidirectional or bidirectional. The destination of a connection statement is a non-memory element. A connection statement in the non-procedural part implies permanent connection from the source to the destination. If used in the procedural part, then the source is available at the output of the destination immediately after the rising edge of the corresponding CSL. A connection between bidirectional buses is called a bidirectional connection. Thus:

\[
x :=: y
\]

means \( y \) is connected to \( x \) and also \( x \) is connected to \( y \).

**Production 42:** The actual argument list for a submodule invocation is a list of CGLRMs (Prod 44) separated by semicolons.

**Production 43:** See Prod 48.
Production 44: Elements of an actual argument list may be a BGLRM (Prod 43) or '??'. '??' means that no actual argument is to be connected to the corresponding formal argument.

Production 45: The destination of a synchronous transfer must be synchronous inputs of storage elements. If the destination is enabled, then the transfer will take place on the falling edge of its driving clock. If the driving clock is other than the master clock, then the designer must ensure a proper phase relationship between the two clocks (also see Prod 47, 48, and 50).

Production 46: The asynchronous transfer (Prod 46.01 and 46.02) takes place as soon as the destination is enabled. Productions 46.03 and 46.04 specify which bits of the destination element should be Set or Reset. Set and Reset are performed regardless of the status of enable line.

Example:

\[
A <s- /1,0,0,1,1,0/ \\
\] Bits 0, 3, and 4 of A would be set to 1

\[
B <R- /0,1,1,0,0,0/ \\
\] Reset bit 1, 2, and 3; leave bits 0, 4 and 5 unaltered

\[
C <- /1,0,1,0,0,1/ \\
\] Old value of C will be replaced by 101001.

Also, see Prod 47, 48 and 50.
Production 47: DLRM, or a destination line register or memory, may be composed of several registers catenated by row or column. They can be grouped together using parentheses. In its simplest form, a DLRM is a single register or line. The facility to catenate is merely a convenience, and the statement can always be rewritten in its fully expanded form. Thus:

\[ A!B \leftarrow C!D \]

is the same as

\[ A\oplus C; B\leftarrow D \]

if all of them have equal number of rows.

Production 48: GLRM, or a general line register or memory, is the most general AHPL element. It is the result of scalar and vector operations performed on one or more elements. For example,

\[ ((A\&B)!((C+D),E))\ast F \]

is a GLRM.

Production 49: The Destination elements of a transfer statement may be conditionally enabled. If the condition is true, then the transfer will take place or else the corresponding element will be left unaltered.

Example:

Let B and C be equidimensional vectors; and x a scalar

\[ (B \oplus C) * (x,\oplus x) \leftarrow E \]

B will get E if x is true, C stays unchanged

C will get E if x is false, B stays unchanged.
Only one selection operator is permitted in an expression. Thus:

The expression \((B \star x) \neq (C \star y)\) is illegal.

**Production 50:** A CLHS may be enclosed in parentheses.

**Production 51:** See Prod 47.

**Production 52:** See Prod 47.

**Productions 53 to 59:** See Prod 48.

**Productions 60.01 and 60.02:** Used to invoke a CLU or a primitive function. Terms within the parentheses are actual arguments. They are connected to formal arguments by positional correspondence. See Prod 42.

Time needed to perform combinational logic function is assumed to be very short compared to the clock period; hence, there is no timing problem.

**Productions 60.03 and 60.04:** These productions enable the user to specify a vector of constants (ROM).

Example:

/0,0,0,1,0,1/ creates a 6-bit vector of constants

The encode statement (Prod 60.04) has two arguments. The first argument gives the number of columns of the generated vector, and the second argument gives its decimal value. Thus the statement 6$11 is the same as /0,0,1,0,1,1/.

**Production 60.05:** AHPL expressions may be parenthesized. Parentheses are used to make an expression more legible or to override the precedence of operators.
Production 60.06: See Prod 29.

Production 61: AHPL allows the user to select subset of a vector or matrix by means of an indexing operation. Row indices are placed in angular brackets and column indices in square brackets. Thus the statement $A[3:7]<2:6>$ selects column number 3 to 7 and rows 2 through 6 of the matrix $A$. Recall that 0-origin indexing is used; $lb$ must be numerically less than $ub$ in a statement like $x[lb:ub]$.

Production 62: $lb$ and $ub$, discussed above, may be any arithmetic expression. If used in a module description, then the expression must not contain any variable.

Production 63: A control reset statement is globally active. The module is reset to a specified state asynchronously. It stays in that state as long as the corresponding reset signal is active (high).

Example:

```
CONTROLRESET (a1, a2, a3, a1)/(10, 20, 30, 40)
```

If $a_1$ is high, go to step #10 and 40 simultaneously
If $a_2$ is high, go to step #20
If $a_3$ is high, go to #30.

More than one signal may be active at a time. A module must not be reset to a nodelay step.

Production 64: A CLU header gives the generic name of the CLU, a list of formal arguments, and optionally, a list of parameters. The formal arguments are bound to the actual arguments by positional correspondence at the time of invocation. Formal parameters are bound to
the actual parameters at the time of declaration. Parameters may be used in the CLU description as a variable in arithmetic expressions. CLU description is used as a template. Each invocation generates a new copy of the CLU. Identical copies may be merged into one if they have the same input lines.

Production 65: A CLU description may include several declarations separated by periods.

Production 66: See Prod 70

Production 67: Formal arguments must be simple identifiers separated by semicolons.

Production 68: A CLU must declare input output lines. It may declare other CLUs. Local variables are declared as CTERMS. These translate into simple wires.

Production 69: The main part of a CLU description consists of several activities separated by semicolons.

Production 70: Three types of activities are permitted: connection (Prod 41), IF statement, and FOR statement. IF and FOR statements may be nested.

Production 71: An IF statement begins with the keyword IF followed by a relational expression. The expression is evaluated; if it is true then the THEN-clause is executed. If the expression is false, then the Else-clause (if any) is executed.

Production 72: FOR statement is of the form

$$\text{FOR } l = a_1 \text{ to } a_2 \text{ step } a_3 \text{ CONSTRUCT}$$
1 is assigned the value of a1 for the first iteration. For each successive iteration, a3 is added to it. The loop terminates when 1 becomes equal to a2. a3 may be negative; in this case a2 must not be greater than a1. If "STEP a3" is omitted, a step size of unity is assumed.

i is an index variable and not a data element. Similarly, a1, a2, and a3 are arithmetic expression and not data elements. FOR statement simply provides a way to abbreviate long descriptions. For instance, the statement:

```
FOR 1 = 1 to 3 construct
ROF;
```

is the same as the following statements:

```
```

**Production 73:** A relational expression is an index variable followed by a relational operator followed by an arithmetic expression. For example:

```
x >= 5*1
```

The expression is true if the current value of x is greater than or equal to 51. x and I must not be data elements.
Productions 74 and 75: Then-clause begins with the keyword "Then" followed by one or more activities. Similarly, Else-clause begins with the keyword "Else" followed by one or more activities. Activities may include other If statements (Prod 70).

Production 76: See Prod 72.

Production 77: Six types of relational operators are permitted: equal to, less than, greater than, not equal to, less than or equal to, greater than or equal to.

Production 78: Functional registers are also submodules. Its header begins with the key word FREG. The rest of the header is similar to a CLU header (Prod 64).

A functional register description, like the CLU description, is also used as a template. However, each invocation does not generate a new copy. One, and only one functional register is generated for each locally declared FNREG. If the same FNREG is invoked in more than one step, implied busing is assumed. See also Prod 39.
CHAPTER 4

IMPLEMENTATION OF STAGE 1

4.1. Stage 1 Output

The State 1 output consists of sixteen tables. Fifteen of these are stored in a large one-dimensional array called Store. The one remaining, called Symbol table, is stored in a four-column array, Symtab.

The common storage, Store, is therefore shared by fifteen dynamically growing tables. Sharing is done by means of director arrays, one array for each table. Two functions, RECEIV and LOCATE [17], use these director arrays to access Store. Given the row number and column number of a table, these functions return a pointer to Store where the entry of interest can be found or stored. Details regarding the approach may be found in reference [14].

A brief discussion of all the tables will be given, followed by a short example. Note that several tables have one or more "unused" columns. This augmentation facilitates the design of stage 2.

4.1.1. Symbol Table

This table is used to store the name of declared symbols. Each entry may hold up to 40 characters. It must be noted that this table stores only the name of a symbol and not its attributes. For this reason,
several unrelated symbols may point to the same symbol table entry. For convenience and clarity, a table entry pointing to the symbol table is replaced by the actual name in the print-out (see Table 4.1).

4.1.2. System Table (SYSTAB)

Stage 1 tables may be shared by several modules and submodules. The SystemTable provides important partitioning information. Entries in the table are explained in Table 4.2. Columns 10 and 11 show nodes allocated for the module. Node numbers assigned to both declared and implicitly generated network elements which may be referred to by stage 2 as a numbered network point. Allocation of nodes for CLU and Functional Register is not done by the stage 1 compiler; hence columns 10 and 11 are zero for submodules.

4.1.3. Symbol Declaration Table (SDT)

Pertinent type and dimension information regarding symbols declared in an AHPL program are stored in this table. These symbols may be identifiers (variables), submodule names, or certain parameters and index variables. Explanation of table entries is given in Table 4.3. The code for symbol type, in column 2, is derived from the BNF. Thus memory has a code of 125, which is computed by the formula code = Prod * 10 + subpro. The only exception is functional register input and output which are given a code of 681 and 682, respectively. A type code of 0 is assigned to undeclared elements such as index variable. Columns 6 and 7 are zero if the symbol occurs in a module description. For a CLU or FNREG description, however, these columns are used by stage 2 to compute columns 4, 5, and 8 (see Table 4.3).
Table 4.1. Symbol table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>First ten characters of the symbol</td>
</tr>
<tr>
<td>2</td>
<td>Next ten characters of the symbol</td>
</tr>
<tr>
<td>3</td>
<td>Next ten characters of the symbol</td>
</tr>
<tr>
<td>4</td>
<td>Last ten characters of the symbol</td>
</tr>
</tbody>
</table>

Unused space is filled with blanks
Table 4.2. System table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Name of the module or submodule</td>
</tr>
<tr>
<td>2</td>
<td>Code for the type: 81 for module 641 or 642 for CLUNINT 781 or 782 for FNREG</td>
</tr>
<tr>
<td>3</td>
<td>SRT row number for the master clock of the module</td>
</tr>
<tr>
<td>4</td>
<td>SDT row number where the declaration for the module begins</td>
</tr>
<tr>
<td>5</td>
<td>SDT row number where the declaration ends</td>
</tr>
<tr>
<td>6</td>
<td>SQRT row number where steps for the module begins</td>
</tr>
<tr>
<td>7</td>
<td>SQRT row number where steps for the module end</td>
</tr>
<tr>
<td>8</td>
<td>REF row number where the reference entries for the module begin</td>
</tr>
<tr>
<td>9</td>
<td>REF row number where the reference entries for the module end</td>
</tr>
<tr>
<td>10</td>
<td>Lower end of nodes allocated for the module by the Stage 1 compiler</td>
</tr>
<tr>
<td>11</td>
<td>Upper end of allocated nodes</td>
</tr>
<tr>
<td>12</td>
<td>Unused</td>
</tr>
<tr>
<td>13</td>
<td>Unused</td>
</tr>
</tbody>
</table>
Table 4.3. Symbol declaration table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Symbol table row number for the name of the symbol.</td>
</tr>
<tr>
<td>2</td>
<td>Code for the type of the symbol.</td>
</tr>
<tr>
<td>3</td>
<td>Row number of the REF table, describing the subtype of the symbol.</td>
</tr>
<tr>
<td>4</td>
<td>Total number of bits in each row of the symbol; that is, number of columns of the symbol.</td>
</tr>
<tr>
<td>5</td>
<td>Total number of bits in each column of the symbol; that is, number of rows of the symbol.</td>
</tr>
<tr>
<td>6</td>
<td>Row number of Thunk table which can be executed to give number of columns of the symbol. Useful for CLUNIT and FNREG where the dimensions cannot be computed by Stage 1.</td>
</tr>
<tr>
<td>7</td>
<td>Row number of Thunk table which can be executed to give number of rows of the symbol.</td>
</tr>
<tr>
<td>8</td>
<td>An unique number assigned to each declared symbol. To be used by Stage 2 as node number for circuit elements linked list. A negative entry points to another SDT row where the symbol has actually been stored. Used for semilocal and global declaration.</td>
</tr>
<tr>
<td>9</td>
<td>Row number of TOTS which heads the list of user-defined node numbers for the symbol. This column is zero if user does not wish to define his own node number. Normally used for external symbols.</td>
</tr>
<tr>
<td>10</td>
<td>Unused.</td>
</tr>
</tbody>
</table>
4.1.4. Symbol Reference Table (SRT)

This table explains how a declared symbol is used. Columns 6 to 9 are filled only for submodules. These columns are used to compute columns 2 to 6 by the stage 2 program (Table 4.4).

4.1.5. Step QTABLE Relation Table (SQRT)

This table shows the relationship between steps of a module and QTABLE entries. A module description has only entry for each step plus one for the non-procedural part. A CLU or FNREG description has only one entry in the table. A 0 in column 2 implies that the QTABLE entries referred to by columns 3 and 4 either belong to the non-procedural part of a module or they belong to submodule description (Table 4.5).

4.1.6. Quadruple Table (QTABLE)

All activities of an AHPL description are broken up into quadruples and stored in QTABLE. The production number given in column 1 is computed by the formula PROD \* 10 + Subprod. Details are given in Table 4.6.

4.1.7. Table of Temporary Symbols (TOTS)

TOTS is used to store additional information regarding operands of a quadruple. A detailed description is given in Table 4.7.

4.1.8. Reference Table (REF)

Information contained in REFl and REF2 (Prods 14 and 16, Section 3.1) portion of a symbol declaration is stored in REF table. The table also stores the information contained in CLU and FNREG
Table 4.4. Symbol reference table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Row number of SDT where the symbol is declared.</td>
</tr>
<tr>
<td>2</td>
<td>Lower column subscript.</td>
</tr>
<tr>
<td>3</td>
<td>Upper column subscript.</td>
</tr>
<tr>
<td>4</td>
<td>Lower row subscript.</td>
</tr>
<tr>
<td>5</td>
<td>Upper row subscript.</td>
</tr>
<tr>
<td>6</td>
<td>Row number of Thunk table which can be executed to give lower column subscript. Useful for CLUNIT and FNREG where the subscripts must be computed by Stage 2.</td>
</tr>
<tr>
<td>7</td>
<td>Row number of Thunk table which can be executed to give upper column subscript.</td>
</tr>
<tr>
<td>8</td>
<td>Row number of Thunk table which may be executed to give lower row subscript.</td>
</tr>
<tr>
<td>9</td>
<td>Row number of Thunk table which may be executed to give upper row subscript.</td>
</tr>
</tbody>
</table>

A negative entry in columns 6-9 means that they have been copied from the corresponding SDT columns.
Table 4.5. Step QTABLE relation table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Row number of the system table where the module is defined.</td>
</tr>
<tr>
<td>2</td>
<td>Current step number.</td>
</tr>
<tr>
<td>3</td>
<td>QTABLE row number where the quadruples of the current step begin.</td>
</tr>
<tr>
<td>4</td>
<td>QTABLE row number where the quadruples of the current step end.</td>
</tr>
<tr>
<td>5</td>
<td>A '1' in this column indicates that it is a NODELAY step.</td>
</tr>
</tbody>
</table>
Table 4.6. Quadruple table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Production number, defining the operation.</td>
</tr>
<tr>
<td>2</td>
<td>Row number of TOTS containing the second operand. Zero for a single operand instruction.</td>
</tr>
<tr>
<td>3</td>
<td>Row number of TOTS containing the first operand.</td>
</tr>
<tr>
<td>4</td>
<td>Row number of TOTS containing the result. This column is zero for transfer and connection operations, and for FNREG invocation. Zero for branch and control reset operations also.</td>
</tr>
</tbody>
</table>

For CLU and FNREG invocations, column 2 contains the row number of TOTS which specifies the name of the sub-module, and column 3 contains the row number of TOTS which point to the actual argument list.

For productions 711 and 712, column 2 contains the row numbers of IF table where details regarding the IF statement have been stored. Similarly, for productions 721 and 722, column 2 contains the row number of FOR table.
<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A negative number in column 1 is one of the four possible codes discussed below. The code indicates the type of information contained in the TOTS row. A positive number refers to an appropriate QTABLE entry. Used for LHS catenations and LHS conditional productions.</td>
</tr>
<tr>
<td>2</td>
<td>Number of columns of the operand.</td>
</tr>
<tr>
<td>3</td>
<td>Number of rows of the operand.</td>
</tr>
<tr>
<td>4</td>
<td>Depends on the code in column 1. Discussed below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column 1 Code</th>
<th>Column 4 Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Contains SRT row number describing the declared operand.</td>
</tr>
<tr>
<td>-2</td>
<td>Contains starting node numbers for undeclared operand. These nodes may be referred to by Stage 2 as a numbered network point.</td>
</tr>
<tr>
<td>-3</td>
<td>Contains row number of PINTAB in case the operand is a bitstring or branch or reset destination step numbers. If the TOTS entry is pointed at by a SDT entry, then the PINTAB will contain user-defined node number for the symbol. Column 2 gives number of PINTAB entries for the operand.</td>
</tr>
<tr>
<td>-4</td>
<td>Pointer to ARGUMENT table where list of actual arguments (for submodule invocation) starts. Column 2 in this case gives number of arguments.</td>
</tr>
</tbody>
</table>

Note that in case of CLU a TOTS row may be placeholder for several nodes which will be generated by Stage 2.
headers. The Reference table greatly facilitates the interface between module and submodules (Table 4.8).

4.1.9. Parameter Table (PARAM)

There are three types of parameters: (1) subtype specifier for memory or bus element; (2) formal parameter for submodules; (3) actual parameters for submodules. If actual numeric parameters are supplied by a module, then they must be Integer Constants. In this case they are placed in column 2 and column 1 is zero. For all other cases, column 2 is zero. If the parameter is a subtype specifier, then the negative entry in column 1 points to the SDT entry where the subtype symbol is stored. For all other cases, column 1 is positive and points to the THUNK table (Table 4.9).

4.1.10. Argument table (ARG)

This table has only one column. It points to SDT if the argument is formal argument; otherwise it points to TOTS (Table 4.10).

4.1.11. FOR Table

This table is used by QTABLE to store additional information regarding a FOR statement (Table 4.11).

4.1.12. IF Table

The IF table is used by QTABLE to store additional information regarding an IF statement (Table 4.12).
Table 4.8. REF table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If the reference is to a CLUNIT or a FNREG, this column contains their name; otherwise it is blank.</td>
</tr>
<tr>
<td>2</td>
<td>Row number of system table where the referred submodule is defined. This column is zero if the submodule is not defined or the reference is made to something other than a module.</td>
</tr>
<tr>
<td>3</td>
<td>Row number of ARGUMENT table where the list of the formal argument begins.</td>
</tr>
<tr>
<td>4</td>
<td>Row number of ARGUMENT table where the list of the formal argument ends.</td>
</tr>
<tr>
<td>5</td>
<td>Row number of parameter table where the list of the actual or formal parameter begins.</td>
</tr>
<tr>
<td>6</td>
<td>Row number of parameter table where the list of the actual or formal parameter ends.</td>
</tr>
<tr>
<td>7</td>
<td>Unused.</td>
</tr>
<tr>
<td>8</td>
<td>Unused.</td>
</tr>
</tbody>
</table>
Table 4.9. PARAM table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A negative number in column 1 gives SDT entry where the parameter is described. A positive number contains the row number of THUNK table where the parameter is stored</td>
</tr>
<tr>
<td>2</td>
<td>Value of the parameter, if it can be determined; 0, otherwise.</td>
</tr>
</tbody>
</table>

Table 4.10. ARG table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Row number of SDT (for formal arguments), or row number of TOTS (for actual argument) where the argument is described.</td>
</tr>
</tbody>
</table>
### Table 4.11. FOR table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pointer to SDT, where the index variable is described.</td>
</tr>
<tr>
<td>2</td>
<td>Pointer to THUNK table's row which may be executed to give the initial value of the index variable for FOR loop control.</td>
</tr>
<tr>
<td>3</td>
<td>Pointer to THUNK table's row which may be executed to give the final value of the index variable for FOR loop control.</td>
</tr>
<tr>
<td>4</td>
<td>Pointer to THUNK table's row which may be executed to give the step size.</td>
</tr>
<tr>
<td>5</td>
<td>QTABLE row number where Productions for the FOR loop begin.</td>
</tr>
<tr>
<td>6</td>
<td>QTABLE row number where Productions for the FOR loop end.</td>
</tr>
</tbody>
</table>
Table 4.12. IF table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Row number of SDT where the index variable is described.</td>
</tr>
<tr>
<td>2</td>
<td>Relational operators &lt;, =&lt;, =, &gt;, =&gt;, &lt;&gt;.</td>
</tr>
<tr>
<td>3</td>
<td>Pointer to THUNK table's row which may be executed to get the arithmetic expression.</td>
</tr>
<tr>
<td>4</td>
<td>QTABLE row number where Production for SUCCESS begins.</td>
</tr>
<tr>
<td>5</td>
<td>QTABLE row number where Production for SUCCESS ends.</td>
</tr>
<tr>
<td>6</td>
<td>QTABLE row number where Production for FAILURE begins.</td>
</tr>
<tr>
<td>7</td>
<td>QTABLE row number where Production for FAILURE ends.</td>
</tr>
</tbody>
</table>
4.1.13. THUNK Table

Information regarding arithmetic expressions is stored in this table. The production number in column 1 is calculated by the formula Prod number = Prod * lo + Suprod. Since all the arithmetic expressions used in a module must either be a constant or an id, this table is recovered at the end of a module description (Table 4.13).

4.1.14. PINTAB Table

PINTAB stores integers. These may be branch or control reset step numbers, bitstrings, or user-defined node numbers. This table is referred by TOTS table (Table 4.14).

4.1.15. Label Reference Table (LRT)

LRT makes linkage between a label and the original symbol (Table 4.15).

4.1.16. Pulse Table

This points to SDT entry of symbols declared as pulse in a module (Table 4.16).

Example 4.1 is given below which illustrates how the tables are used. The example is specially designed to exercise many of the available features of the language. The example is just for illustration purposes, and the described system does not perform any meaningful operation. The example should be studied in conjunction with Tables 4.1 through 4.6 and the BNF given in Section 3.2.
### Table 4.13. THUNK table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Production number defining the arithmetic operation.</td>
</tr>
<tr>
<td>2</td>
<td>Second operand, zero for single operand operations. If the production is ID (253), then this column contains the SDT row number where ID has been stored.</td>
</tr>
<tr>
<td>3</td>
<td>First operand.</td>
</tr>
<tr>
<td>4</td>
<td>Result of operation. In case of CLU and FNREG, result is not computed by Stage 1, and this column is zero.</td>
</tr>
</tbody>
</table>

If column 1 is 212, production for arithmetic expression result, then column 2 contains the THUNK row number where productions of the arithmetic expression begin and column 3 contains the row number where it ends. The fourth column in this case stores the result of the whole expression when it is executed.
### Table 4.14. PINTAB table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>List of user-defined node numbers, branch or control reset numbers, or bitstrings.</td>
</tr>
</tbody>
</table>

### Table 4.15. LRT table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Name of the label.</td>
</tr>
<tr>
<td>2</td>
<td>Pointer to SDT where the label is described.</td>
</tr>
<tr>
<td>3</td>
<td>Pointer to SRT where the original symbol is described.</td>
</tr>
</tbody>
</table>

### Table 4.16. PULSE table, description of contents

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Description of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pointer to SDT where the pulse (or clock) is described.</td>
</tr>
</tbody>
</table>
Example 4.1.
Example 4.1 -- Continued
# Table of Temporary Symbols

<table>
<thead>
<tr>
<th>Code</th>
<th>Oue</th>
<th>Code</th>
<th>Road</th>
<th>PNTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
</tbody>
</table>

# Parameter Table

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

# Example 4.1 -- Continued
### Example 4.1. -- Continued
Example 4.1. -- Continued
4.2. Syntax Analysis

Syntax analysis is done by a bottom-up table-driven parser. The scheme used here is similar to the one discussed in chapter 13 of reference [18]. SLR(1) BNF of the language is given to an automatic parser generator program [15]. The program generates the parse table. A modified sparse matrix technique is used to store these tables. Two stacks are used to do the parsing (syntax analysis). STACK 1 contains the parser states and STACK 2 contains either the value of the symbol (subtoken) being processed or the result of previous reductions.

The main subroutine of the syntax analysis module is called SYNTAX. This routine calls SCANER to get a token and a subtoken (value). The token is translated into the SYNTAX internal code, called term. This code, along with top of STACK 1 is used to access a proper entry of the parse table. This entry gives two quantities--ACT and COD. Based on COD, one of the five actions is taken. These actions are listed below.

Error: An error has been encountered in the source code. Take appropriate error recovery action.

Accept: The source program has been successfully parsed and the parser has reached the goal symbol. Return to the caller.

Exchange: Pop out an element from stacks and push ACT on the stack.

Shift: Push Term and Act on STACK 1 and the subtoken followed by a zero on STACK 2.
Reduce: An LHS has been encountered. Pop contents of STACK 1 and STACK 2 into Buff 1 and Buff 2. These buffers communicate between syntax and semantic routines. Call SEMANT to take appropriate semantic actions. The result is returned by SEMANT in a variable called Newpnt, push this on STACK 2. Determine the next state by using present state and reduced LHS to access a proper GOTO table entry.

Scanning is performed by the subroutine SCANER. It reads the input text, one character at a time. The character is converted into an internal code and a value. The value may be, for example, actual value for a numeral, ASCII code for an alphabetic character, etc. Using the internal code and scanner's current state, a proper entry of the scanner table is accessed. This entry gives three quantities—action, subact, and next state. An action may be like forming an integer or an identifier, looking for a delimiter, returning to SYNTAX with a token and subtoken, reading more input, etc. Subact specifies scanner code for multicharacter symbols.

Figure 4.1 gives input output interface between the syntax analysis module and the rest of the system.
4.3. Semantic Action

The Semantic Action Module produces the tables discussed in Section 4.1. The main semantic routine is called SEMANT. It is invoked by subroutine SYNTAX when the latter discovers a Left Hand Side (LHS)--Production. Inputs to the subroutine are the production number, subprod number, a number specifying number of RHS elements for the production, and the two buffers--Buff1 and Buff2. In addition, a variable Newpnt is used which contains the result of a previous semantic action. At the end of the action, the variable Newpnt is given the new results. The result is usually the row number of one of the tables where information regarding the production is stored. For several Productions, no result is needed.

Buff2 contains details about the right-hand side (RHS) of the production. For terminals appearing on RHS, it contains their value code--symbol table entry for ID, actual value for integer, zero for keywords and delimiters. For non-terminals appearing on RHS, it contains the result of semantic action when the non-terminal appeared on the left-hand side.

As an example, consider the production:

8.01 <MODHEAD> ::= MODULE:ID

The production number will be 8, subprod will be 1, the number giving number of RHS will be 3, and Buff2 will contain 0 0 and n where n is a pointer to the symbol table where the ID is stored.

Refer to the example given in Section 4.1. A printout showing Buff2 and other arguments passed to Semant for the statement
ALPHA <= BETA; => (1) is shown below. A brief explanation is also given.

Consult the BNF given in Section 3.2 in conjunction with the listing of subroutine SEMANT [17] to understand various entries.

\[
\begin{array}{llll}
\text{PROD} & 2; \text{CNT} & 1\text{BUFF2} & 15 \\
29 & & 15 & 7 \\
52 & & 16 & 10 \\
51 & & 10 & 10 \\
47 & & 16 & 15 \\
52 & & 16 & 16 \\
60 & & 8 & 8 \\
59 & & 16 & 11 \\
58 & & 11 & 11 \\
57 & & 11 & 11 \\
56 & & 11 & 11 \\
55 & & 11 & 11 \\
54 & & 11 & 11 \\
53 & & 11 & 11 \\
43 & & 11 & 11 \\
48 & & 11 & 11 \\
45 & & 0 & 10 \\
40 & & 11 & 11 \\
38 & & 11 & 11 \\
36 & & 11 & 11 \\
37 & & 0 & 11 \\
35 & & 1 & 11 \\
34 & & 1 & 11 \\
32 & & 2 & 11 \\
17 & & 1 & 11 \\
\end{array}
\]

\[
\begin{align*}
16 & \cdots & 2 & \text{ALPHA} \leq \text{BETA}; \\
29 & \cdots & 4; \text{CNT} & 1\text{BUFF2} = 1 \\
31 & \cdots & 1; \text{CNT} & 3\text{BUFF2} = 0 \ 0 \\
31 & \cdots & 2; \text{CNT} & 2\text{BUFF2} = 0 \ 1 \\
37 & \cdots & 1; \text{CNT} & 3\text{BUFF2} = 1 \ 0 \\
35 & \cdots & 2; \text{CNT} & 1\text{BUFF2} = 1 \\
32 & \cdots & 1; \text{CNT} & 4\text{BUFF2} = 1 \ 0 \ 2 \ 1
\end{align*}
\]

ALPHA, appearing on LHS of the transfer statement, is accepted as Prod 29 subprod 2 (see the BNF). The buffer contains the row number of symbol table (not shown), where the name 'ALPHA' is stored. The proper area of SDT is searched to find the entry in column 2. A new row of SRT (see row 7 of SRT) is allocated; columns of the SRT entry are filled in; the row number of SRT is returned in Newpnt as the result.
of the semantic action. The next production, 52.02, is processed by allocating a TOTS row (see row 10) and filling its columns appropriately. Notice that the RHS of the production is a non-terminal <SLRM> which was processed during the previous call. Recall that Buff2(l) now contains the result of the semantic action when this non-terminal appeared on the LHS. Thus Buff2(l) contains SRT row number where ALPHA is stored. This row number is stored in column 4 of TOTS (see Table 4.8). The next production results in a call to PDLRM. Notice that the subroutine SEMANT processes simple productions by itself and calls other subroutines to process more difficult productions. Since the subprod is 2, no processing is needed and Buff2(l) is returned as the result. Next, the RHS of the expression is parsed. For 29.02, as before, an SRT row is allocated. The row number is returned as the result. For 60.06 <Element>::=<SLRM>, a TOTS entry is allocated and its row number is returned. For the next 9 productions, no processing is needed other than simply copying Buff2(l) into Newpnt. Production 45.01 is for synchronous transfer.

SYNCTR>::=<DLRM><=GLRM>

Notice that the three Buff2 elements correspond to the three RHS quantities. A new QTABLE row is allocated (see row #5) and the quadruple is stored there. For the next 2 productions, no semantic action is needed. The third one is used to count the number of activities in the step. Production 31.04 causes the destination step number to be stored in the PINTAB table. Production 37.02 is processed by allocating a
TOTS row (see row 12 of TOTS) and storing pertinent information in the row. Then a QTABLE row is allocated where the quadruple is stored (see row 6 of QTABLE). For Production 34.02, column 5 of the SQRT row for the current step is filled to indicate whether it is a nodelay step or not. The next production gives the step number of the current step. It is stored in SQRT, column 2. See row 2 of SQRT. This completes the processing of the step. The rest of the program is processed in the similar fashion.

4.4. Use of Stage 1 Output for Simulation

Stage 1 output is a tabular representation of the circuit at the functional level. Several uses of this output are possible. One, of course, is to feed it to the Stage 2 processor to get an internal representation of gate and memory element interconnection. Other uses may be, for example, fault isolation, block diagram drawing, or function level simulation.

A function level simulator for AHPL II is available. Simple modifications in the program would enable it to use the Stage 1 output. Input to the simulator should be broken into two parts: one, the Stage 1 tables; and two, the user-supplied parameters for clock limits, initial input vectors, and other useful information for simulation. A separate processor for these parameters may be designed. The complete arrangement of the system is given in Figure 4.2.
The example of Figure 4.2 clearly illustrates how application-dependent parameters may be supplied at a subsequent stage and processed independently of the language.
Figure 4.2. The proposed simulator.
CHAPTER 5

OPTIMIZATION USING LINKED LIST

Tables produced by stage 1 are converted into an abstract network. The abstract network is an interconnection of declared elements as well as implicitly generated elements. The network representation is abstract in the sense that the final interpretation depends on the application. Stage 3 program for one application may use the linked list to produce the actual hardware, while that for another application may interpret it as a set of boolean expressions describing the circuit. A doubly linked structure may be used to store the network. Such a structure allows for quick insertion and deletion [20, 21]. Since several elements may be connected to one element, the structure used here is more complex than a simple doubly linked list.

Allocation and deallocation procedure is very simple. Elements are allocated from a large linear array. The element number corresponds to the index of the array; thus a sorted list is always available. Deletion occurs only during the final optimization. The deallocated nodes are not returned to the storage pool. This makes allocation and deallocation very fast.

The most time-consuming phase in any network synthesis program is the optimization process. In particular, the removal of redundant elements is expensive. If special provisions are not made in the data
structure itself, the process will be of the order of \( (n^2) \) or even \( (n^3) \) in the worst case. This means that a network of modest complexity would take several minutes of a mainframe CPU time. A novel scheme has been used to make the optimization an order \( (n) \) process. The data structure will be discussed first, followed by a discussion on the optimization technique.

5.1. Structure of a Node

The structure of a node is depicted in Figure 5.1.

- ELEMENT TYPE
- SIGINP
- SIGLNK
- ILINK
- OLINK
- SYMLNK

Figure 5.1. Node representing a network element.

The first cell gives the type of the element. An element type may be AND, OR, NAND, DFF, etc.

SIGINP gives the sum of all the inputs to the element. Thus if an element has node #200, node #256, and node #68 as its input, then SIGINP of the element would be 524. SIGLNK points to another element which has either the same SIGINP number or a SIGINP number which is the same number + a multiple of 1000. SIGINP and SIGLNK are used to speed up the optimization.

ILINK, OLINK, and SYMLNK are pointers to IOLIST. ILINK heads the input list; OLINK the output list; and SYMLNK the symbolic name list, explained later.
A node is stored in six simultaneous arrays, each array storing one particular attribute. Word-packing to store several attributes in a single word has been avoided for the sake of speed.

5.2. Structure of I0LIST

I0LIST is used to store additional information regarding a node. Each node of I0LIST has three elements. Two of these are information elements, and the third points to the next I0LIST node which contains similar information. The third element is zero if there is no successor node. Nodes of I0LIST are allocated sequentially. No explicit deallocation is made. I0LIST is stored in a \( n \times 3 \) array; \( n \) at present is 6000 and can easily be increased.

5.3. Network Representation Using Linked List

The complete network is stored using the nodes of Section 5.1 and I0LIST of Section 5.2. A partial network is shown in Figure 5.2. Figure 5.3 shows how this network is stored using the linked list, Element 250 is of type NAND. Its Ilink heads a list which contains numbers 100, 185, 222 and 300. These elements are connected to the input of element 250. Similarly, OLINK points to the output list. SIGINP is \( 100 + 185 + 222 + 300 = 807 \).

5.4. Removal of Redundant Elements

An element is redundant if there is another element in the network which has the same type and exactly the same inputs. If two elements have exactly the same inputs, then their SIGINP must be the
Figure 5.2. A partial network.

Figure 5.3. Node representation of the circuit of Figure 5.2.
same. Elements with the same SIGINP may be linked together in a linear linked list. Thus the search to find redundant elements is confined to a very small list.

A hash table (HSHSIG), in conjunction with SIGINP and SIGLNK, is used to make the list. At present the size of the hash table is 1000. All the elements with the same SIGINP MOD 1000 are linked together in a linear linked list. The header of this list is stored in SIGINP MOD 1000^th location of the hash table. Elements with SIGINP=0 are also linked together with their header stored at HSHSIG(1). Clearly, increasing the size of HSHSIG will further reduce the search time.

The Siglist is maintained by three routines—SIGADJ, SIGPRE, and HASH. SIGADJ is called whenever SIGINP of a gate changes. It is given the gate number and its new SIGINP. Using SIGPRE and HASH, the routine removes the gate from its previous list and inserts it into the new list. The operation is fairly simple. HASH function at present computes SIGINP MOD 1000. This means that the elements having different types may be linked together. The present scheme may be improved by taking type of the element into account while computing the Hash function or by having one hash table for each element type. These schemes would further limit the search to most likely candidates.

A typical Siglist is shown in Figure 5.4. Only SIGINP and SIGLNK are shown; other attributes of the element node are omitted for clarity.
The search for redundant elements starts from the first entry in HSHSIG. The entire list headed by this element is searched, and if redundant elements are found they are removed. After the first list is completely searched, the process moves to the next list. Removal of a redundant element may create other redundant elements. For this reason, several passes over HSHSIG are made until all redundant elements are removed.

![Diagram showing the arrangement of SIGLISTS.](image)

Figure 5.4. A typical arrangement of SIGLISTS.

5.5. Other Optimizations

Other optimizations include:

1. Removal of elements with no input (except for declared elements).

2. Removal of elements with no output (except for declared elements).
3. Removal of single-input elements, if possible.
4. Merging of smaller gates to form larger ones.
5. Removal of VCC and GND connections, if possible.
6. Removal of duplicate inputs from gate-type elements

Linked-list structure, because of its quick and easy insertion and deletion, proved to be very helpful for these optimizations also.
CHAPTER 6

IMPLEMENTATION OF STAGE 2

Stage 2 produces an abstract representation of the system described by the AHPL source program. This network is stored in the Abstract Element Linked List (AELL) discussed in the previous chapter. Input to the Stage 2 processor is the set of tables produced by Stage 1. Its output is the Abstract Element Linked List, along with some Stage 1 tables, including the symbol table, system table, and SDT. These three tables help to organize the Stage 3 algorithm. A Stage 2 table CLUTB [35] is also outputed to help the Stage 3 in processing submodules.

6.1. An Overview of Stage 2

Figure 6.1 is a VTOC representation of Stage 2; details have been omitted.

The output of Stage 1 is read and stored in a large array called Store. The storage method is similar to the one discussed in Section 4.1. These tables are processed to build the abstract network. Finally, the network, along with some of the Stage 1 tables and with CLUTB, is outputed to a file called STAGE2.OUT.

The individual modules of the network are built separately, one module at a time. Submodules are built if they are invoked by the module being compiled. The strategy is to simply look at each row
Figure 6.1. A simplified VTOC representation of Stage 2.
of the System table. Column 2 of the table specifies the type. If the type code is 81, implying a module, then the row is processed; otherwise it is ignored.

Columns 4 and 5 of the System table give the range of SDT row numbers where declared symbols for the module are stored. Column 2 of SDT gives the type of these elements. A type of 121 through 126 means that these are data elements (see Section 3.2). A negative entry in column 8 implies that the element has been described in another module. An entry in column 9 means that user has defined node numbers for the element. Using these last two columns, nodes from AELL are allocated, one node for each data element bit specified in the module. As an example, see SDT row 12 of the previous example. The SDT entry specifies that the symbol AOUT is of type OUTPUTS, it has six columns and one row; that is, six elements. Column 8 gives the starting point of node numbers. Thus six elements from AELL are allocated (nodes 128 through 133), and their type is set as OUTPUTS. Symbolic information about the element is stored as shown in Figure 6.2.

![Figure 6.2. An initialized node.](image)

Note that the first entry of the IOLIST cell depicted in Figure 6.2 gives the index of the symbol table where the symbol "AOUT" is stored.
The second entry is a packed integer giving both row number (0 in this case) and the column number. The third entry of the I0LIST cell would link to another cell if additional information is to be stored.

If the declared element is of type Memory, then appropriate gates are connected to its input. This is symbolically shown in Figure 6.3.

![Figure 6.3. A memory element initialization.](image)

The next initialization task is to assign AELL nodes for control sequence flipflops (or OR gates for a nodelay step). Columns 6 and 7 of system table give the range of SQRT rows belonging to the Module. One control element is generated for each step of the module. A node number is assigned to each control element and is stored in column 6 of the corresponding SQRT row. If the control element is a flipflop, then gates are connected to its input, as shown below. Notice that no OR gate is connected to the Enable input, since the control
flipflops are always enabled, VCC is directly connected to the input. If a Stage 3 compiler decides to use a 4-input flipflop for a control element, it can do so by ignoring the connection to the enable line. The module number and the step number given in columns 1 and 2 of SQRT are the only symbolic information about the control element. They are stored using SYMLINK of the assigned node, as discussed above. Assignment of control elements concludes the initialization phase.

The complete module is built by processing its individual steps in sequence, one step at a time. Recall that actions performed by each step have already been converted into quadruples and stored in the QTABLE by the Stage 1 compiler. Columns 3 and 4 of SQRT give the range of QTABLE entries belonging to the step. Each of these QTABLE entries are processed in sequence.
Column 1 of QTABLE gives the production code of the quadruple. For some productions, no processing is necessary in that they are processed in conjunction with some other productions. For instance, LHS catenation is processed with the transfer or connection quadruple. The most difficult to process is the quadruple for submodule invocation. It is discussed in the next section. Compilation of most of the quadruple is done in two steps. The first step involves the setting up of arguments, and the second generates appropriate circuitry as needed. A three-column array, ARGLIS, is used to set up the arguments. The number of useful entries in each of the columns of ARGLIS is recorded in NIN(3). A quadruple usually has two operands and a result. Node numbers representing the operand in the second column of QTABLE are stored in the first column of ARGLIS. Similarly, node numbers of the operand in the third column are stored in the second ARGLIS column, and the result in the third column. Figure 6.5 shows the ARGLIS entries for the statement A<= /0/,A[1:17] of the example given in Section 4.1. Note that a gnd is internally represented as -2.

Some productions, for example RHS catenations, do not generate any circuitry. They are used just to set up proper arguments for transfer or connection productions which follow. Quadruple which do generate hardware need further processing. If new nodes are needed, then they are allocated from AELL. Then the connections specified by the quadruple are made. A few examples are given below to illustrate the circuit generated by some of the AHPL statements. Newly generated elements and connections are shown by dotted lines.
<table>
<thead>
<tr>
<th>SUBR CMPQTB: PROD = 604</th>
<th>NIN STACK = 10 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUBR CMPQRB: PROD = 531</th>
<th>NIN STACK = 17 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>-2</td>
</tr>
<tr>
<td>103</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>0</td>
</tr>
<tr>
<td>105</td>
<td>0</td>
</tr>
<tr>
<td>106</td>
<td>0</td>
</tr>
<tr>
<td>107</td>
<td>0</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
</tr>
<tr>
<td>109</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>112</td>
<td>0</td>
</tr>
<tr>
<td>113</td>
<td>0</td>
</tr>
<tr>
<td>114</td>
<td>0</td>
</tr>
<tr>
<td>115</td>
<td>0</td>
</tr>
<tr>
<td>116</td>
<td>0</td>
</tr>
<tr>
<td>117</td>
<td>0</td>
</tr>
<tr>
<td>118</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUBR CMPQTB: PROD = 451</th>
<th>NIN STACK = 18 18 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>102</td>
</tr>
<tr>
<td>102</td>
<td>103</td>
</tr>
<tr>
<td>103</td>
<td>104</td>
</tr>
<tr>
<td>104</td>
<td>105</td>
</tr>
<tr>
<td>105</td>
<td>106</td>
</tr>
<tr>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>107</td>
<td>108</td>
</tr>
<tr>
<td>108</td>
<td>109</td>
</tr>
<tr>
<td>109</td>
<td>110</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>111</td>
<td>112</td>
</tr>
<tr>
<td>112</td>
<td>113</td>
</tr>
<tr>
<td>113</td>
<td>114</td>
</tr>
<tr>
<td>114</td>
<td>115</td>
</tr>
<tr>
<td>115</td>
<td>116</td>
</tr>
<tr>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td>117</td>
<td>118</td>
</tr>
<tr>
<td>118</td>
<td>119</td>
</tr>
</tbody>
</table>

Figure 6.5. Sample ARGLIS entries.
Example 6.1 (see Figure 6.6):

\[ 10 \Rightarrow a/5 \]

![Figure 6.6. Circuit for Example 6.1.](image)

Example 6.2 (see Figure 6.7):

\[ 10 \cdots <= B \cdot a; \]

![Figure 6.7. Circuit for Example 6.2.](image)
Example 6.3 (see Figure 6.8):

\[ 10 \times b \leq C + D. \]

The expression would generate 3 QTABLE entries which are symbolically shown below:

\[ \times b \ A \ T1 \]
\[ + 0 \ C \ T2 \]
\[ \leq T2 \ T1 \ 0 \]

Figure 6.8. Circuit generated by Example 6.3.

Multiple activities to a data element are compiled separately and are ORed together at the input of the element. However, before ORing a new activity, the algorithm checks to see if the same source has been connected before. If so, then a common control gate for the two activities is formed. The example below illustrates the process.
Example 6.4 (see Figure 6.9):

1  A[0]<=C[0]
5  A[0]<=B[0]

: 
10  A[0]<=B[0]
15  A[0]<=B[0]

Figure 6.9. Multiple activities to a flipflop simple strategy.

Figure 6.10. Multiple activities by forming common control subexpression.

Control expressions are usually common to several bits of a register; thus the above method saves an appreciable amount of logic elements.

A transfer statement also generates control logic for the enable input of the memory element. Newly generated control logic is ORed with the ones generated earlier. The control logic generated for
the enable input is different from the one generated for the data input because the LHS asterisk is used to control the enable input only. The partial network of Figure 6.8 illustrates this difference.

Each data element bit is compiled individually. After the complete network is built and redundant gates are removed, one may easily form partitioned segments based on several criteria. For instance, all the bits with common enable logic may be grouped together as a segment (see Section 8.2.1). Non-consecutive bits of a register or even bits from different registers may be grouped together to make a segment.

Processing of a module is complete when all the steps belonging to it have been processed. If the user has a request for optimization, then the network is optimized. The process consists of removing unnecessary gates and unnecessary connections. For instance, a gate with no output is eliminated, any VCC input to a multiple input AND gate is removed. Single-input AND and OR gates are replaced by their inputs. Smaller gates are merged to form larger one. Redundant gates are removed.

After compiling one module, the program moves to the next one until no more modules remain.

6.2. Additional Processing for Submodules

There are important differences between a module and a submodule—FNREG and CLU. However, the process of network generation is largely the same. Most of the subroutines used by the module compiler to process QTABLE quadruples are also used by the submodule compiler.
Significant differences, from the point of view of the compiler designer, between module and submodule are:

1. Submodules are declared in modules or other submodules. Information regarding the submodule dimension and parameters must be extracted from this declaration.

2. A submodule description has a header statement which gives formal arguments and parameters. Unlike modules, input output lines of submodules are simply formal arguments and do not represent any hardware.

3. Submodules are not autonomous. They are activated only when invoked. The invocations give the actual arguments which replace the formal arguments.

4. Index variables, FOR constructs and IF constructs are allowed in CLU description.

        Submodule processors are invoked when an invocation quadruple is encountered. Major activities of a submodule processor are:

1. Using the SDT entry for the local name of the submodule, find its generic name and actual parameters.

2. Check to see if the submodule has been compiled before. If so, skip Step 3.

3. Compile the complete submodule using its formal arguments.

4. Invocation quadruple gives the actual arguments. Replace formal arguments of the compiled module by actual arguments.
6.2.1. CLU Processor

A detailed description of the processor may be found in reference [35].

6.2.2. Functional Register Processor

Major differences between the compilation process of a combinational logic unit and that of a functional register are:

1. A functional register may have several types of internal data elements, e.g., buses, memory elements, other functional registers and CLUS.
2. Multiple invocation of a functional register implies a busing network at its input whereas that for a CLU generates multiple copies of the CLU.
3. Clocks may be specified as a parameter in the functional register. The clock line must be controlled by the CSL of the step in which the functional register is invoked.
4. Local name of the functional register may appear on the right-hand side of a transfer or a connection statement.

Modifications in the CLU compiler [35] were made to handle the functional register also. Additional routines were written to handle the above-mentioned differences between the two type of submodules. A detailed example is being worked out. The example and program listing will become a part of Reference [17].
6.3. A Demonstration Stage 3

Stage 2 stores the compiled network in the Abstract Element Linked List. This form of storage is most convenient for further processing by Stage 3 application programs. However, it is not well-suited for human inspection. The purpose of the demonstration Stage 3 is to print the circuit in a human readable form.

The program uses the first three tables of Stage 1, the CLUTB, and the linked list. Modules are printed out one at a time. First the control section is printed, then the declared data elements, and finally the undeclared gates. A summary of number of elements of each type used by the module is also given.

The demonstration package also includes programs which allow the user to perform interactive editing. It allows the user to inspect a portion of the circuit, insert and delete gates, and add or remove gate inputs. This package may be used to combat timing problems by inserting gates along critical path. It can also be used to conveniently debug the circuit without altering the source code and recompiling. This demonstration Stage 3 was used in conjunction with the example to be discussed in Chapter 9, to aid the reader.
A COMPILER EXAMPLE

An example is presented in this chapter to illustrate the output of various stages of the multi-stage AHPL compiler. AHPL description of the module is given in Figure 7.1. The description defines a Module MULTISHIFT and a Combinational Logic Unit INCR. The Module description consists of two parts: a declaration part, and a sequence part. In the declaration portion, various data elements are declared. The declaration specifies element name, its type, and its dimension. Optionally parameters may be included. The first statement declares memory elements and is similar to AHPL II. Next a CLUNIT is declared. Its local name is INC, it is of the type INCR, and has a dimension of 3. Unlike most programming languages, AHPL requires the user to declare submodules. This is in accordance with AHPL's philosophy that all data elements which are to be manipulated explicitly must be declared. Of course, CLUNITs and FUNCTIONAL REGISTERS are data elements. The declaration of submodules helps to generate different local copies from the same generic template. These copies even are different from one another depending on parameters appearing within curly brackets. The next statement declares two external lines, RESET and CLOCK. These are just names of the lines and not AHPL keywords. CLOCK is later declared as PULSES; it is this statement which directs the compiler to
Figure 7.1. A compiler example.
treat the line as a clock. The Universal AHPL permits several clocks in a module. In the next statement it is specified that the line 'CLOCK' will drive all the control sequence flipflops. The RESET line is used in the globally active CONTROLRESET statement appearing towards the end of the Module description. This indicates that a '1' on this line would cause the module to reset to the control state 1. The next section of the Module description is the sequence part. Except for the changed syntax of the CONTROLRESET statement, all the statements in this section are similar to AHPLI. The CONTROLRESET statement in the Universal AHPL allows one to specify signals which would cause the Module to reset.

The combinational Logic Unit description follows the module description. The Universal AHPL does not impose any restriction on the order in which Modules and Submodules must appear. The first statement in the unit description is the header. The header begins with the statement CLU: followed by the generic name of the Submodule --INCR. Then the list of formal arguments appears within parentheses followed by a list of formal parameters appearing within curly brackets. The parameter 'I' appearing within curly brackets may be used as an integer variable in the CLU description. In this example it has been used to: (1) specify the dimension of formal arguments; (2) for indexing within the program; and (3) to control the FOR loop. Using the powerful mechanism of parameters in this manner, it is possible to generate INCR units of different dimensions from the same generic
description. The reader may verify for himself that the unit described here is indeed a modulo 2 ** 1 incremener.

Figure 7.2 shows the Stage 1 output tables. Refer to Section 4.1 for a detailed discussion of these tables. The SYMBOL table is not printed; instead the SYMBOL table entries appearing in other tables are replaced by actual symbols.

Stage 2 converts the tabular output of Stage 1 into a Linked List representation of the circuit. Figures 7.3a and 7.3b show the internal structure of the Linked List. Figure 7.3a depicts the element nodes and 7.3b depicts the IOLIST. Only a partial listing is given. ILINK, OLINK, and SYMLNK of element nodes point to the IOLIST. For example, element 101 is a D-flipflop, its ILINK, OLINK, and SYMLNK point to entries 25, 570, and 24 of the IOLIST. Following the IOLIST, one can find out that elements 262, 125, 264, VCC, and VCC are connected to the input of element #101; the output of #101 is connected to elements 391, 483, and 598; and it has the symbolic name A[1].

The Linked List output of Stage 2 is most suitable for further manipulation by subsequent stages. However, it is not suitable for human inspection. For this reason, a demonstration Stage 3 has been prepared. The purpose of this stage is simply to convert the Stage 2 output into a more readable form. Figure 7.4 shows the output of Stage 3. First the Control elements are listed, then the data flip-flops, followed by outputs, inputs, and CLUNITs. Finally the rest of the elements are listed.
### SUPAP : SYSTEM TABLE

<table>
<thead>
<tr>
<th>RNUM</th>
<th>HATL</th>
<th>TYPE</th>
<th>CLUT#</th>
<th>LSI</th>
<th>JS1</th>
<th>LSQ1</th>
<th>USQ1</th>
<th>LRFF</th>
<th>UREF</th>
<th>S1NL</th>
<th>S1NH</th>
<th>S2NL</th>
<th>S2NH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111</td>
<td>R1</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

### SUPDL : DATECLARATION TABLE

<table>
<thead>
<tr>
<th>RNUM</th>
<th>SPREC</th>
<th>TYPE</th>
<th>SUBTYPE</th>
<th>COPS</th>
<th>ROWS</th>
<th>CTUN</th>
<th>RTUN</th>
<th>PTHN</th>
<th>PIN</th>
<th>UNUSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>125</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>B</td>
<td>121</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>121</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>C</td>
<td>133</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>D</td>
<td>123</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>E</td>
<td>123</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>F</td>
<td>123</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>G</td>
<td>123</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>H</td>
<td>123</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>I</td>
<td>123</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>J</td>
<td>123</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### SKETL : REFERENCE TABLE

<table>
<thead>
<tr>
<th>RNUM</th>
<th>YTDL</th>
<th>STATE</th>
<th>LARG</th>
<th>UARG</th>
<th>LRPARA</th>
<th>UPARA</th>
<th>NCJLS</th>
<th>NROWS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### SKN1 : SYMBOL REFERENCE TABLE

<table>
<thead>
<tr>
<th>RNUM</th>
<th>SYDL</th>
<th>CLL</th>
<th>CUL</th>
<th>RLL</th>
<th>RUL</th>
<th>CLTN</th>
<th>CUTCN</th>
<th>RLTCN</th>
<th>RTUCN</th>
<th>RTUN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 7.2. Stage 1 tables.
**CONTINUED**

<table>
<thead>
<tr>
<th>RS#</th>
<th>TOT1</th>
<th>TOT2</th>
<th>TOT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
</tbody>
</table>

**TABLE 11: QUARKS -- Continued**

<table>
<thead>
<tr>
<th>RS#</th>
<th>TOT4</th>
<th>TOT5</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
</tbody>
</table>

**TABLE 12: IDENTIFICATION OF SYMBOLS**

<table>
<thead>
<tr>
<th>RS#</th>
<th>CODE</th>
<th>COLS</th>
<th>ROWS</th>
<th>PNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Figure 7.2. -- Continued*
Figure 7.2. -- Continued
### Parameter Table

<table>
<thead>
<tr>
<th>No.</th>
<th>Func. Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

### Student Table

<table>
<thead>
<tr>
<th>No.</th>
<th>J Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Final Table

<table>
<thead>
<tr>
<th>No.</th>
<th>J Test</th>
<th>J Final</th>
<th>Step</th>
<th>Low-J</th>
<th>Up-J</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>23</td>
<td>1</td>
<td>44</td>
</tr>
</tbody>
</table>

Figure 7.2. -- Continued
Figure 7.3. Stage 2 Abstract Element Linked List.

a. Element nodes.

b. IOLIST.
Figure 7.3b.
### Output Summary

#### Stage 3 Output

<table>
<thead>
<tr>
<th>NAME</th>
<th>COLUMNS</th>
<th>COLS</th>
<th>TYPE</th>
<th>INPUT LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>d 1 13</td>
<td>134</td>
<td>117</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 7.4. Stage 3 output.*
Figure 7.4. -- Continued
Figure 7.4. -- Continued
Stage 3 output is easy to read. An explanation of one entry from each of the sections would suffice. Element #367 appears in the Control section. It is the Control flipflop for Step 1 of the Module Multishift. Its Data input is connected to Element #368, Element 125 is the driving clock, and Element #369 is connected to its Set input. Reset and Enable input of the flipflop are connected to Vcc. As an example from the Memory section, consider element #101. The element has the symbolic name A<0>[0], its D, C, and E inputs are connected to element number 262, 125, and 264, respectively. Its Set and Reset inputs are connected to Vcc. The output line Z<0>[0] appearing in the next section is assigned element #134 and its input is connected to element #117. Since the statement Z=A[17] appears in the non-procedural part of the module description (see Figure 7.1), the output line Z is permanently connected to the flipflop A[17]. The next section describes input lines. Only external inputs are used in this example. The external line RESET<0>[0] is assigned element #124; since it is an external signal it has no input. Details regarding the CLUNIT is given in the next section. The generic name of the CLU is INCR, its local name is INC, it is invoked by the module MULTISHIFT in step #4. Elements 532 to 534 are inputs to the CLU, elements 535 to 537 are its output, and elements 538 to 542 are internal to the Submodule. ST3PAR is 0, indicating that the CLU must be built and not connected as a black box. Details regarding CLU elements are given with the rest of the gatelist which follows. The gatelist simply lists the element number,
its type, and its input list. For instance, element #262 is an OR gate, elements 379, 469, 566, and 567 are its input. Gate elements of type CLUIN and CLUOUT may be treated as simple wires. Gate of type CND has only two inputs; the first one is the data input and the second one is the control. These gates may be replaced by AND gates in TTL technology and by Pass transistors in MOS.

As a visual aid to the reader, a partial network of the circuit is shown in Figures 7.5a through 7.5c. It is a simple matter to draw the rest of the circuit from Stage 3 output.

Although each data element bit is individually represented in the Stage 2 Linked List, it is possible to form larger data blocks. Frequently a partition based on Enable input of data flipflop is desirable. Such a partition can readily be formed by assembling all flipflops with common Enable control gate into a block. Similarly, partitions based on other criteria may be formed.
Figure 7.5. A partial network for the example.

a. A partial control unit.

b. Transfer into a flipflop.

c. The combinational logic unit.
(a)

(b)
CHAPTER 8

A GUIDE FOR WRITING STAGE 3

In most of the real-world applications, Stage 3 will be an interface between AHPL Stage 2 and already-existing software at the place of application. One such example is discussed in Section 9.1.1. Although there is no such thing as a general-purpose Stage 3 processor, some useful guidelines for preparing a particular Stage 3 can be given.

The person responsible for building a Stage 3 processor must be thoroughly familiar with Stage 2 data base and the front end of the local software (if any) with which AHPL Stage 2 is to be interfaced. However, it is not necessary for the designer to have an understanding of either the Stage 1 or the Stage 2 algorithms.

8.1. Defining Local Parameters

The first task of the designer is to decide what kind of local parameters are needed and how they are to be entered.

Figure 8.1 gives a block diagram showing how the multistage compiler would be coupled with local CAD systems. Stage 3 parameters can directly be entered in the source code by using curly brackets, as discussed in Chapter 3. Parameters can also be entered, independent of the source code, to the Stage 3 compiler directly. For example, if the purpose is to build the circuit in some particular technology, then
Figure 8.1. Block diagram of a typical AHPL application.
some design rules will be needed to systematically convert the abstract network produced by Stage 2 into real hardware. Then rules can be stored in a file and invoked by the Stage 3 processor when needed.

8.2. Accessing and Manipulating AELL

Structure of the Abstract Element Linked List has been discussed in Chapter 5. Recall that a node of AELL has six cells. These are stored in six simultaneous arrays: GATTYP, SIGINP, SIGLNK, OLINK, SYMLNK, and ILINK. Thus, attributes of link list elements can be accessed directly. For instance, type of element N can be accessed by the statement GATTYP(N). Stage 2 has a package of subroutines which it uses to access and manipulate AELL. The package consists of 24 subroutines and functions and gives the user a good handle over the linked list. Depending on his need, the designer of a Stage 3 processor may choose to copy the entire package, to copy only a part of it, or to write a similar package of his own.

Along with the linked list, Stage 2 outputs some Stage 1 tables as well as the CLUTB table. These tables can effectively be used to organize Stage 3 design. Refer to Table 4.1 through 4.3 and 4.14 for details regarding these tables. The symbol table and PINTAB table do not undergo any change. Columns 12 and 13 of the system table now give the range of nodes allocated by the stage 2 compiler for undeclared network points of the module. Column 9 of SDT now points to PINTAB table instead of pointing to TOTS. Stage 1 tables are still available
in the file STAGE 1·OUT; they may be copied if needed. The tables will not grow dynamically, so a simple storage method may be employed. Stage 2 output is stored on a file called STAGE2·OUT. Refer to subroutine ST2OUT to see how the data is arranged. A simple subroutine may be written to read the data back into the core.

The system table defines the boundaries of individual modules. Whether these boundaries have any significance or not depends entirely on the application. SDT gives a convenient way to access individual elements of declared data symbols. Control elements have type DFCS or ORCS. These can conveniently be found by searching elements of the linked list; other circuit elements do not have any structural significance and may be accessed directly.

8.2.1. Building segments

Each bit of a data element is individually represented in the abstract network produced by Stage 2. It is often desirable to combine several bits, based on certain criteria, into a larger block or segment. This can easily be accomplished by making a segment table. Each row of the table will give the lower and upper bound of data elements belonging to the segment, as shown in Figure 8.2.

<table>
<thead>
<tr>
<th>Row #</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>100</td>
<td>117</td>
</tr>
<tr>
<td>2:</td>
<td>118</td>
<td>121</td>
</tr>
<tr>
<td>3:</td>
<td>156</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.2. Building segment table consecutive nodes.
If elements of segments are non-consecutive, then indirect accessing techniques may be used. This scheme is shown in Figure 8.3. Nodes 5, 16, 7, 18, 2, and 4 are in segment #1. Nodes 8, 12, and 68 are in segment #2.

<table>
<thead>
<tr>
<th>Row #</th>
<th>Low Seg Row</th>
<th>Up Seg Row</th>
<th>Row #</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>20</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>26</td>
<td>28</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td>23</td>
<td>18</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td>27</td>
<td>12</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td>28</td>
<td>68</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.3. Building segment table non-consecutive nodes.

For certain applications, there may be more than one criterion for making segments. One or more columns may be added to the segment index table to store information regarding these criteria.

As an example of segments, consider the module MULTISHIFT described on page 242 of reference [6]. Segments shown as destination group in Figure 7.16 of the text are based on clock enables. That is, all the bits of a register which are enabled by the same conditions are grouped together. Such segments can easily be made by examining enable
input of each element of a register and grouping the one with common enable signals into a segment. Notice that the book has assumed these bits to be consecutive. However, if the scheme of Figure 8.3 is used, then the assumption would not be necessary.

8.2.2. Changing node numbers

Node numbers of network elements were allocated without regard to their type or any other criteria. This was done because criteria suitable for one application may not be suitable for another.

Suppose that an already-existing local application program requires that all input/output nodes should be numbered from 1 to 50, control elements from 51 to 99, data flipflops from 100 to 200, and all the buses from 300 to 400. Undoubtedly, the restriction seems somewhat artificial, but such arbitrary restrictions are not at all uncommon. Most of the local application programs were developed several years ago when modern programming techniques were still in their infancy. This is not to belittle these programs; they are excellent and do their job well. Nevertheless, trying to modify them or even to change their front end will not be an easy task. It is the task of the Stage 3 designer to accommodate for all idiosyncrasies of the local software rather than trying to modify the software itself.

Node numbers can be changed easily. The first step is to make an array which maps old node numbers into new ones. Old numbers are stored in the array and its index gives the new node numbers. A simple Fortran program can build the node map shown in Figure 8.4. A partial listing of the program is given below:
OLD NODE NUMBERS

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
</tr>
</tbody>
</table>
| ... | ...
| 50 | 286 |
| 51 | 296 |
| ... | ...
| 90 | 356 |
| 100 | 357 |
| ... | ...
| 300 |   |
| 400 |   |
| 401 |   |

ALL I/O

ALL CONTROL ELEMENTS

ALL DATA FLIPFLOPS

ALL BUSES

OTHER ELEMENTS

Figure 8.4. Node map.

C Program to build the node map

NEWNOD = 1

DO 10 I = 1, GNEXT ! GNEXT gives the node number of the last node in the linked list

IF(GATTYP(I).NE.INPUTS.AND.
   GATTYP(I).NE.OUTPUTS) go to 10

NODMAP(NEWNOD) = 1

NEWNOD = NEWNOD + 1

10 CONTINUE

NEWNOD = 51

DO 20 I = 1, GNEXT

IF(GATTYP(I).NE.DFCS.AND.
   GATTYP(I).NE.ORCS) go to 20

NODMAP(NEWNOD) = 1

NEWNOD = NEWNOD + 1
Of course, more sophisticated algorithms may be developed to do the job in a single pass over the linked list. The old gate numbers may now be replaced by new numbers by a subroutine similar to REPLAC. However, a much quicker method would be to make another array with index corresponding to old gate numbers and entries corresponding to the new numbers. These two arrays can be used to translate old numbers into new ones and vice versa, without making any changes in the linked list itself.

8.2.3. Assigning PIN Numbers to MSI Parts

MSI parts used in a circuit may be described as functional registers or CL units. Their input and output lines are given symbolic names. These names are used as formal arguments and are connected to actual arguments when the submodule is invoked. At the final stage in the design process, it is necessary to have a mapping between the formal arguments and the actual pin numbers of the MSI part. This can be accomplished by establishing a library of MSI parts which will give the correspondence between the formal arguments and the pin numbers. The pin numbers of each device can be stored in a one-dimensional array according to a mutually agreed-upon convention. Since there will be several devices in the library, some mechanism to access the proper array index must be developed. A simple method would be to use the device name to give the displacement into the array. For a library
with several hundred devices it will be useful to employ hash table techniques [27] to speed up the search process.

8.2.4. Minimizing the Number of Control Elements

The control unit is implemented using one control element for each control state. However, it is possible to use only $\log_2 n$ control elements for $n$ control states. At times such realization proves to be more economical. Furthermore, circuits which do not have reset lines must be realized using $\log_2 n$ control elements so that they assume one of the legal states after being turned on.

Suppose that a module has the following control sequence:

1. $1 \rightarrow \bar{a}/1$
2. $2 \rightarrow b/1$
3. $3 \rightarrow (\bar{c}, c)/(2, 4)$
4. $4 \rightarrow (1)$.

If four D-flipflops are used to implement the above control unit, then their excitation equation may be written as follows:

$$\begin{align*}
CSL1 &+ CSL1 \bar{a} + CSL2 b + CSL4 \\
CSL2 &+ CSL1 a + CSL3 \bar{c} \\
CSL3 &+ CSL2 \bar{c} \\
CSL4 &+ CSL3 c
\end{align*}$$

Left arrows are used instead of equal signs to symbolize the clock delay associated with flipflops. Above equations can be implemented by using four flipflops, six AND gates, 2 OR gates, and 3 Inverters.
It is possible to use only two flipflops to generate four control state levels (CSL). The above equations may be rewritten in terms of two flip CSM1 and CSM2.

\[
\begin{align*}
CSL_1 &= \overline{CSM1} \& CSM2 \\
CSL_2 &= \overline{CSM1} \& CSM2 \\
CSL_3 &= CSM1 \& \overline{CSM2} \\
CSL_4 &= CSM1 \& CSM2 \\
CSM1 &= CSL_2b + CSL_3e \\
CSM2 &= CSL_1a + CSL_3 \\
\end{align*}
\]

Above equations may be implemented by using two flipflops, 7 AND gates, two OR gates, and three inverters. No attempt was made to find the best state assignment.

An outline of a procedure to convert a control unit with one element per control state to the one with \( \log_2 n \) elements for \( n \) control state is given below:

1. Allocate \( \log_2 n \) flipflops where \( n \) is the number of control states excluding NODELAYS.

2. Using \( \log_2 n \) to \( n \) decoder or discrete gates, generate CSL1 through CSLN.

3. Input equations to the new flipflops will be OR of the input equation of properly selected old flipflops. For example, in the above illustration the input to CSM1 is OR of inputs to CSL3 and CSL4. Similarly the input to CSM2 is OR of inputs to CSL2 and CSL4.
4. Delete old flipflops.
5. Realize NODELAY steps.
6. Optimize the new control unit.

Subroutines SETGAT, ADDIN, SEEIN, DELGAT and CMPOPT may be used to accomplish the above task.

A note on the nature of AELL is in order. While at times it seems that the Stage 2 has produced the actual circuit, with very little room to make any modification, it is very easy to manipulate the linked list and come up with a different implementation. It is this flexibility, the author hopes, which would allow the AHPL compiler to work in different environments.

8.3. A Proposed Mask Generation System

A system to automatically generate masks from the functional description of a chip is proposed in this section. The purpose here is to use the system as a vehicle to illustrate several aspects of Stage 3 design and to demonstrate how Stage 2 output can be used by already-existing application programs. For the purpose of illustration, it is assumed that the available mask generation system accepts NAND representation of a circuit and generates the mask. The complex algorithm of routing and placement to make the best use of available chip area is considered to be outside of Stage 3. It is awkward to design a large circuit in terms of NAND gates. The system proposed here would allow the user to design the circuit using AHPL. A block diagram of the proposed system is shown in Figure 8.5.
8.3.1. Meeting FANIN FANOUT Requirements

FANIN and FANOUT limits are based on the technology. These limits may be entered as Stage 3 parameters. FANIN criteria may be met by simply breaking up a gate into smaller ones as shown below.

Suppose a gate Gx is to be broken up. If Gx is of type AND or OR, then the procedure is:

1. Using the subroutine SETGAT, allocate \( n \) gates of the same type as Gx, the one being broken up. Where \( n = \lfloor \frac{\text{# of INPUTS of } Gx}{\text{FANIN limit}} \rfloor \)

2. Connect \( \frac{\text{# of inputs}}{n} \) inputs to each new gate. By calling subroutine ADDIN.

3. Remove all the inputs from Gx, by using the subroutine DELIN.

4. Using the subroutine ADDIN, connect the newly generated gates to Gx as shown above.

Recursive application of the algorithm may be necessary if \( n \) is larger than the FANIN.

Similar algorithms for other types of gates may be developed. Notice that the newly generated gates are connected as input to Gx. Thus, the rest of the linked list need not be changed.

FANOUT limits can be met simply by using buffers, or parallel gates, as shown below.
Figure 8.5. The proposed mask generator system.
Figure 8.6. Meeting FANOUT requirements.

An algorithm using parallel gates is given below:

1. Allocate \( n \) gates of type \( G_x \) where \( n = \left\lceil \frac{\# \text{ of outputs of } G_x}{\text{FANOUT limit}} \right\rceil - 1 \)

2. Connect all the inputs of \( G_x \) to these gates also.

3. Divide the output equally, as shown above.

8.3.2. Converting into NAND Gates

AND/OR logic may be converted into NAND by changing all AND/OR operations into NAND operations and inverting signals at odd levels. A simple AND/OR logic and its NAND equivalent is shown in Figure 8.7.

In order to prove that the two circuits are equivalent, consider the boolean equation of the circuit on the right-hand side:

\[
\overline{a.b \cdot \overline{c.d} \cdot \overline{e.f} \cdot g \cdot \overline{h}}
\]

Simplifying:
\[ \begin{align*}
\overline{a+b} \cdot \overline{c+d} \cdot \overline{e+f} \cdot g \cdot \overline{h} \\
= \overline{a+b} \cdot (c \cdot d + e \cdot f) \cdot g \cdot \overline{h} \\
= \overline{a+b} \cdot ((c+d) \cdot (e+f) + g) \cdot \overline{h} \\
= ab + (cd + ef) \cdot g + h \text{ which is the same as the left-hand side.}
\end{align*} \]

Figure 8.7. A logic circuit and its NAND equivalents.

Dealing with equations and trying to find out whether a signal must be inverted or not is a difficult task. A much simpler approach which would yield the same result is to replace individual gates with their NAND equivalents.

\[ \begin{align*}
\overline{a \cdot b} = \overline{a} \cdot \overline{b} \\
\overline{a+b} = \overline{a} \cdot \overline{b} \\
\overline{a+b} = \overline{a} \cdot \overline{b} \\
\overline{a \oplus b} = a \cdot \overline{a} \cdot \overline{b} \cdot b
\end{align*} \]
Using these equations, a circuit may be easily be transformed into its NAND equivalent. The result of applying the first two equations on the circuit of Figure 8.7 is shown in Figure 8.8.

![Figure 8.8. Replacing gates by their NAND equivalents.](image)

The only task remaining is to eliminate redundant gates. This can easily be done by a simple optimizing routine which would remove two inverters connected in series. It can be seen that gates 8 and 18, 9 and 11, 10 and 12, 14 and 15 would be eliminated by this scheme. Note that inverters in parallel must be used for fanning-OUT AND gates.

8.3.3. Converting Flipflops into NAND Gates

First a suitable NAND representation of a flipflop should be developed. The circuit given on page 536 of reference [26] is a valid candidate. The JK flipflop is modified into a D flipflop. In addition, provision is made for Enable, Set and Reset inputs. The modified circuit is shown in Figure 8.9.

A simple subroutine may be written which could convert all flipflops used in a circuit into their NAND equivalents. The subroutine is
given flipflop number, and the five inputs—D, C, E, S, and R. It builds the NAND equivalent. An outline of the algorithm is given below.

1. Allocate 10 NAND gates $G(1)$ through $G(10)$.
2. Add R, $G(8)$ and $G(9)$ as input to $G(10)$.
3. Add S, $G(7)$ and $G(10)$ as input to $G(9)$.
4. Add $G(6)$ and $G(2)$ to $G(8)$.

... 

10. Add D to $G1$.

And finally 11, Replace FFNO by $G(9)$.

All tasks described above can easily be performed by calls to appropriate AELL handling routines. Similarly, other circuit elements, for example, latches may also be replaced by their NAND equivalents.
8.3.4. Output Formatting

The final task of the Stage 3 is to output the circuit in a format acceptable to the front end of succeeding programs. Components of a design automation system may be picked up from different places. In Figure 8.5 it is shown that Stage 3 is connected to a mask generator system as well as to a test system. Thus, two output formatting routines must be written.

8.3.5. Conclusions

Section 8.3 has illustrated how AHPL can be used effectively to enhance the power of already-existing CAD systems. The proposed system allows the chip designer to concentrate on a functional level rather than going down to NAND gates. An additional advantage of the unified approach to design automation is that several types of available CAD systems may be attached. In the above example, since the mask generator and circuit testers are driven from the same description, one is sure that the circuit being tested is the same as the one being built.
CHAPTER 9

CONCLUSIONS AND CURRENT APPLICATIONS

9.1. Current Applications

Three applications of the multistage compilers will be discussed in this section. The author was closely associated with the first one, and had major design and implementation responsibility. He made some contributions towards the other two projects, but the major design responsibility rested with other researchers. These applications illustrate the possibility of designing a comprehensive automation system based on AHPL. Such a system can be expanded by adding pre-designed CAD software.

9.1.1. Device Modeling System for Test Engineers

In order to design test programs for a device it is necessary to prepare its model. The model is prepared from the documentation provided by the device manufacturers. Documentation for SSI and MSI usually include a detailed schematic. It is a simple task to prepare models from these schematics. A good model must be able to predict the behavior of the good device as well as the one with simulated internal failures. It is assumed that models derived directly from schematics would meet the criterion. This approach to modeling has been traditionally taken by automatic test equipment (ATE) manufacturers.
Teradyne, Inc., is an ATE manufacturer. One aspect of its activity is to aid in testing printed circuit boards. A circuit board may have several IC packages. Each IC package is individually modeled. The model is then processed by LASAR[29] software. The software includes a gate level simulator, a test sequence generator, and a diagnosis and analysis package. The test sequence generator automatically generates input patterns which can deduce failure within the device. It is a fairly sophisticated program and can attain a very high percentage of fault coverage for most MSI devices. These patterns generated for a model are used to check if the model is correct. The simulator gives the response of the model for the applied test sequence. The same sequence is also applied to an actual device. The two responses are compared to check if the model is correct. The whole board can then be tested by specifying interconnection of modeled devices and comparing the board's output with the simulator output for a well-designed test pattern. The system may be used for go/no-go test as well as for isolating faulty IC packages.

The most basic building block with which a LASAR model can be prepared is the NAND gate. NAND models for several hundred devices are available in the LASAR library. Models of other building blocks, for example, flipflops, AND gates, OR gates, etc. are also available. A new model may be described in terms of NAND gates or in terms of available library models.

LASAR provides a simple and direct way to build a model from the device schematics. The use of library facilitates hierarchical modeling.
Models of most MSI devices can be prepared with a few days of effort. However, it is not so easy to model an LSI or a VLSI device. The documentation available from the manufacturer for these devices does not include a detailed schematic. The modeling, therefore, becomes a process of re-designing the device from the functional descriptions. LASAR does not help in the design process. The whole design must be broken up into flipflops, gates, and other available library elements. Each element must be assigned a unique number and the whole model must be entered using the format [30]:

[element number] [element type] / [input list]//

To model a device of the complexity of a microprocessor using the above format is not very practical. It is tedious, time consuming, and prone to errors. Modeling a device like M6800 would require several man-months of effort. Thus the need of a more sophisticated method to enter device model became evident. A method which would not only simplify the coding but would also help in the design process.

Design languages provide a formal mechanism to express complex digital devices. It was deduced that with a design language as its front end, the LASAR system can indeed become a powerful modeling tool. Among the several design languages studied, AHPL was found to be the most suitable language for the purpose [30]. AHPL used by Teradyne is slightly different than the one discussed in Chapter 3. However, the basic characteristics are the same. With the new system, the model is
described in AHPL. The linked list output of AHPL compiler is converted into a format suitable for LASAR system. The program to convert linked list output into LASAR format is very simple and needed only a few days of programming effort. This clearly indicates that the AHPL compiler can be used on existing CAD software to meet the challenge of LSI and VLSI technology. Some advantages of the new system, as perceived by Teradyne are:

1. It helps in design organization. Logical ideas can be expressed with ease and flexibility.
2. Actual coding of models becomes much simpler because of a high level of abstraction.
3. Debugging is easier. Logical errors are accurately pinpointed and correction can be made easily.
4. Maintenance of models, revisions and extensions are simpler.
5. High level of abstraction improves communication among test engineers and models can easily be exchanged.

It is debatable whether the model prepared from the functional description is as good as the one prepared from the detailed schematics. INPUT/OUTPUT behavior of both the models will be the same at functional level. But, what about non-functional behavior? Or behavior of a device with simulated faults? How valid is the percentage fault coverage analysis given by the diagnosis program? These questions need a careful study and may be a topic of an interesting and useful research. One must bear in mind, though, that a detailed schematic is usually not
available. And even if it was available it would be a tedious and time-consuming job to develop an LSI model from a detailed schematic. In the future, when LSI and VLSI manufacturers would themselves switch to design language based system, the discrepancy between the actual device and its model will no longer exist.

9.1.2. Test Sequence Generator

A device is said to be functioning properly if it produces the desired output sequence for a given input sequence for all acceptable input sequences. A sequence of $2^n$ patterns is sufficient to test a purely combinational n-input device. Considerably fewer patterns will be needed if special techniques are employed [32]. However, since the response of a sequential device depends on all previous inputs to the device, a very long test sequence may be necessary. Patterns generated manually are usually not adequate to test a highly sequential device. For this reason, attempts have been made to automatically generate these patterns.

A fault is said to be observable if it can be propagated to the device output. Because most practical devices have some amount of redundancy, not all faults are observable. The purpose of a test sequence, therefore, is to propagate observable faults to the output. Effectiveness of a test sequence is measured in terms of percentage fault coverage:

\[
\text{Percent fault coverage} = \frac{\text{Number of faults propagated to the output}}{\text{Total Number of Observable Faults}} \times 100.
\]
A device testing program usually consists of three parts: a test pattern generator, a simulator, and a diagnosis program. The test pattern generator produces a test sequence, the simulator gives the response of the device to the test pattern, and the diagnosis program analyzes which faults have been propagated to the output. It also checks what percentage of fault has been covered and whether more test patterns are needed. For most large sequential circuits, it is not practical to achieve a 100% fault coverage. It must also be noted that most diagnosis program cannot make adjustments for the redundancy in the circuit and therefore overestimate the number of observable faults. In such cases it is not possible to achieve a 100% fault coverage.

Three criteria are used to measure the performance of a test sequence generator:

1. Percentage fault coverage.
2. Length of the test sequence.
3. CPU time spent in generating the sequence.

Several methods have traditionally been used to test sequential circuits. These include: converting the sequential circuit into an iterative combinational logic network [33]; comparing the state table of unit under test with that of a good circuit [34, 35]; and simultaneous simulation of all single fault versions of the circuit for user-supplied input. The hardware compiler can be used to support any of the above-mentioned methods. However, the above methods are not adequate for highly sequential LSI devices with very few output pins. This
observation led to the search for a new method to generate test sequences. This method is based on the design language AHPL.

AHPL partitions a circuit into control sections and a data section. State table of the control section may be searched exhaustively, while only a small portion of the much larger data section needs to be exercised. Furthermore, the compact design language description would aid in guiding the heuristic test sequence generation operation. This is the approach taken by SCRITSS (Sequential Circuit Test Search System) designers [37-39]. Reference [31] gives an overview of this approach and details advantages of this approach over the others.

Figure 9.1 shows a block diagram of the SCIRITSS system. The sensitization search (block 4) finds an input sequence which will drive a particular fault into a memory element or to some circuit output. The propagation search (block 6) finds an input sequence to propagate a stored fault to one of the output. Concatenation of these sequences gives a test pattern. Heuristics are employed for both sensitization and propagation searches to reduce the search cost.

The multistage AHPL compiler can be interfaced with SCRITSS software. Stage 1 output tables may be used to aid the sensitization and propagation searches. The output of Stage 2 may be used by the element simulator, by the D-algorithm program, and by the analysis program. Figure 9.2 shows the block diagram of the proposed system.

9.1.3. SLA Implementation of VLSI

Very large-scale integration technology of 1980s would make it possible to manufacture devices with more than a million elements [41]
Figure 9.1. SCIRTS flow diagram.
Figure 9.2. AHPL compiler interfaced with SCIRTSS.
on a single chip. This means that highly complex special-purpose de­
vices and super microcomputers may be fabricated on a single VLSI pack­
age. Current ad-hoc methods employed by logic designers will not be 
adequate to handle devices of such complex structure. Also, testing 
of highly sequential devices designed by ad-hoc method will not be 
easy. In order to make VLSI realization of special-purpose low-volume 
devices feasible, it is necessary to automate the design process. A 
VLSI design system based on AHPL is under development.

The Universal AHPL supports MOS technology. The Pass trans­
sistor has been implemented as a primitive function. Multiphase clocks 
are allowed. This helps in reducing the power consumption [40]. A 
Stage 3 can be written to convert the linked list output of Stage 2 into 
physical hardware to be implemented as a VLSI chip. Such a program must 
perform the task of automatic routing and placement to make most effi­
cient use of the available chip area. To write a good placement algo­
rithm for a complex device is difficult. However, if some constraints 
are put on the circuit layout then the placement algorithm will be rela­
tively simple. The storage logic array (SLA) [42] constraints the layout 
by distributing logic between rows and columns of an array of memory 
elements. The constraints imposed by SLA simplifies the task of automa­
tic layout with a relatively small penalty in chip area utilization 
[24]. Figure 9.3 shows the block diagram of the process of converting 
AHPL description into SLA layout.
9.2. Evaluation of Accomplishment

Applications discussed in the previous section have demonstrated that a design language-based hardware design automation system can meet the challenge of ever-increasing complexity of digital devices. The three systems developed separately can be combined together into a powerful design automation tool. Additional subsystems may be developed and attached later. Thus the combined system will be able to support all phases of the design activity including testing, from a single AHPL description of the device. Such a system will surely make the task of digital design engineers much simpler and more rewarding.

In designing the system the first task was to select a proper design language. For reasons explained in Chapter 1, AHPL II was chosen as a starting point. Requirements of a wide spectrum of design environments was studied. AHPL II was updated to the Universal AHPL to meet these requirements. The syntax of the new language was rigorously defined using BNF notation. This assured that there was no syntactic ambiguity in the language. The BNF description also helped to make use of an automatic parser generator program. Care was taken to keep the Universal AHPL compatible with AHPL II to the extent possible. The
concept of hardware parameter was introduced. This feature allows specifications of technological details enclosed in curly brackets without affecting the rest of the language. Primitive functions are included to allow the user to express detailed logic in a compact way. At present there are six primitive functions. New functions can be included in the future with minimum programming effort and without affecting the language syntax. Many improvements have been made in the AHPL procedure mechanism. A distinction between local name and generic name allows the user to create several submodules from the same generic description. Variable dimensions are allowed in the submodule description. This allows the user to create submodules of different dimensions from the same generic description. Several other features have been added. A summary of these features is given in Section 2.3.

The language is implemented in a well-structured and modular way. Confusing GOTOs have been avoided. Simplicity and clarity have been maintained throughout the program, even at the expense of additional coding. Compilation is done in stages, with each stage performing a well-defined task.

Stage 1 consists of two modules, a syntax analysis module and a semantic module. Syntax analysis is done by a parser automatically generated from the BNF description of the language. This assures that there is no ambiguity in the syntax. Also, since the parser was automatically generated it was easy to fix errors in the BNF. Future syntactic extension can easily be made by simply changing the BNF and recompiling the parser. The semantic module is also well-organized.
The main semantic routine does the simple semantic tasks itself and calls other subroutines for more elaborate tasks. New productions can easily be included. Tables generated by the Stage 1 compiler can easily be processed by subsequent programs.

The Stage 2 processor produces linked list representation of the circuit from Stage 1 tables. The linked list data structure proved to be very useful. The method to remove redundant gates, discussed in Chapter 5, is unique and has greatly reduced the cost associated with network optimization. As suggested by several examples discussed in Chapter 8, the linked list structure is easy to manipulate and will simplify the task of Stage 3 designers.

Several applications within a short span of time indicate that the methodology followed was a useful one, and the system developed here will be able to meet the challenge of future technology.

9.3. Future Research

The present compiler produces two data bases from an AHPL description of a device. Output of the Stage 1 compiler is a tabular representation of the functional behavior of the device. Output of the Stage 2 processor is a linked list representation of the device in terms of memory elements and logic gates. It will be useful to go one step further and produce another data base representing the circuit in terms of transistors, resistors and other such elements. Obviously this data base will be technology-dependent and the processor to produce it will be called a Stage 3 processor. Several such processors
may be written depending on the technology in which the circuit is to be implemented. Using these three data bases, a very comprehensive hardware design automation system, similar to the one proposed in reference [43], can be developed. Figure 9.4 shows a block diagram of the proposed system.

9.4. Available AHPL Software

Available AHPL software include a function level simulator—HPSIM [10], a compiler—HPCOM [25], and a CLU compiler [36]. HPSIM and HPCOM were written for AHPL11. The CLU compiler is for Universal AHPL. The CLU compiler has been incorporated in the main Universal AHPL Compiler [17] presented in this report. HPSIM is written in FORTRAN and is available on CDC, IBM and DEC10. HPCOM is written in standard SNOBOL4. The Universal AHPL compiler is written in FORTRAN. The compiler is portable, only slight changes in the scanner will be necessary to transport the compiler to other computers.
Figure 9.4. A comprehensive digital design automation system.
LIST OF REFERENCES


