PERFORMANCE MEASUREMENT AND OPTIMIZATION OF AN
IMPLEMENTATION OF STAGES IN B6700 ALGOL

BY M. NAGULESWARAN

A project report submitted in partial fulfillment for the
B Sc (Hons) degree in Computer Science

Supervisor: Dr R A Maclean

October 1977

ACKNOWLEDGEMENTS

The author wishes to acknowledge the assistance and guidance received from Dr R A Maclean, Senior Lecturer in Computer Science under whose supervision this work was carried out.

Thanks are also due to Dr A L Wilkinson, Senior Programmer/Analyst, Computer Center for the help rendered in connection with this work.
Chapter 1: Introduction 1

Chapter 2: STAGE2 Processor 2
2.1 The abstract Machine Concept 2
2.2 STAGE2 2

Chapter 3: Performance Measurements 4
3.1 The Tool used for Measurement 4
3.2 Tests and Results on ALGOL/STAGE2/V2 6
3.3 Discussion 6
3.4 Alternative Solutions 10

Chapter 4: Choice of Solution and Implementation - ALGOL/STAGE2/V3 14

Chapter 5: Measurements of Improvements 16

Chapter 6: Summary 17

References 18

Appendix A The FLUB Machine 19
Appendix B ALGOL/STAGE2/V2 Implementation details 20
Appendix C Word Formats for Alternative Implementations 21
Appendix D ALGOL/STAGE2/V3 Implementation details 23
Appendix E Order of Appearance of Labels in STAGE2 25
Appendix F Macro definitions used for ALGOL/STAGE2/V3 26
CHAPTER 1

INTRODUCTION

Program optimization is the process of changing the organization of a software system so as to optimize resource utilization. When the program to be optimized is an application program rather than a systems program, the factors to be optimized are very much simplified and usually involves, in priority order

i) CPU time

ii) I/O processor time

iii) Storage requirement

This project involved making performance measurements on a particular software, namely STAGE2 macroprocessor and on the strength of the information thus gained, change the organization of the software so as to produce an optimal version.

The work done involved

i) Obtaining statistics about the amount of CPU time spent in various sections in the program and measuring the frequency with which various sections of the program is executed.

ii) A comparative study of the alternative solutions and choosing and implementing one of them on the strength of the data obtained by measurements.

iii) Measurements of improvements in the new version.
CHAPTER 2

STAGE2 MACROPROCESSOR

2.1 The Abstract Machine Concept

STAGE2 was written by W K Waite (2) whose interest is in developing portable software. The philosophy of Waite basically consists of the following steps:

i) Decide on the basic machine instructions and word types required to write the particular software i.e. in effect design an abstract machine embodying these capabilities.

ii) Write the software in this abstract machine language.

iii) To implement it on a real machine a macroprocessor will be needed. A set of macros which will translate the abstract machine instructions into real machine instructions should be written. The macroprocessor will then be used to translate the software written in the abstract machine language into one in the real machine language.

iv) An I/O package will be written to interface the I/O done by the software and the I/O system of the real machine. The I/O package and the expanded software resulting from step-(ii) will form a complete executable program.

2.2 STAGE2

STAGE2 was written mainly to be used for many such implementations. STAGE2 itself is developed by this process. The abstract machine designed for writing STAGE2 is called PLUB. A brief description of PLUB is given in Appendix A.
Many useful text manipulating facilities are available in STAGE2 (4,5). Making precise statements about these facilities will involve lengthy descriptions and they are best obtained from the above mentioned references.

Mentioning the 'Flag line' is appropriate here as this is used in the new implementation. 'Flag line' is the first input record to STAGE2 and this defines a set of special characters that STAGE2 will recognize while processing the following records.

A running version of STAGE2 in B6700 ALGOL existed when this project was started. Details of this implementation which will be referred to as ALGOL/STAGE2/V2 is given in Appendix B.
3.1 The Tool used for Measurement.

The information that is required to design an optimal implementation of STAGE2 is the pattern of distribution of CPU time in the program. STAGE2 has 99 labels plus another 8 labels created for sub-routine return, all evenly distributed in the program. Thus B6700 \texttt{Algo}l compiler option \texttt{SET STATISTICS(LABELS)} is an appropriate method of collecting statistics. This compiler option will also give the frequency with which each block and label is accessed. While using this option following features of it must be borne in mind:

i) The statistics about CPU time is collected at each block or procedure in the 	exttt{Algo}l Program. If the \texttt{LABELS} option is also set at the start of a block or procedure then statistics is collected at label break-points as well as the block that contain the labels. The information obtained at labels is printed out, numbered 1 onwards in the order of appearance of the labels in the block or procedure.

ii) Where a call to procedure is made statistics collecting code is inserted before and after the calling instruction to ensure that the CPU time spent in the called procedure is not included in that of the calling block or its label.

iii) Part of the CPU time required to execute the statistics collecting code itself is included in the printout of statistics made by the compiler. An estimate of such times included in the printout at various levels is explained below. These can be used to obtain a more realistic estimation of the timings from the compiler printout.

Following is the two basic steps performed to obtain CPU time spent at each breakpoint.

Step (i) On entering a breakpoint, either block, procedure or label, the CPU time used by the program so far is obtained by calling the appropriate time intrinsic. If this is \( y \) units and if the CPU
time already spent for this breakpoint is $x$ units, then $(x - y)$ units is stored in the appropriate location for this breakpoint.

Step (ii) On exiting the block of code defined by a breakpoint either due to normal completion of execution of all code or calling another procedure, time intrinsic is called again. If this is $w$ units, then $(x - y) + w$ units is stored in the appropriate location for this breakpoint, which will now contain the new value of CPU time spent for this breakpoint.

An estimate of the times included in the printout are:

(a) The CPU time due to the above code that is included in the timings printout of a block of procedure is 150 - 170 micro-seconds, each time this block or procedure is entered and exited in the normal way.

(b) When a block or procedure is exited due to a call on another procedure step (ii) is performed to store up the time used so far and step (i) is performed on returning. 120 - 140 micro-seconds is included in the printout of calling block or procedure due to each such call and return.

(c) When statistics is collected at label breakpoints, 120 - 140 micro-seconds is included in the printout for a label each time the label is accessed.

(d) In a block or procedure for which statistics is collected at label breakpoints the CPU time spent inside the block or procedure increases linearly with the number of times labels are accessed. For each label access 250 - 275 micro-seconds is included in the timings for the block or procedure.

(e) If case (b) happens inside a block for which statistics is collected at label breakpoints, then statistics collecting code must store up the time used so far i.e. step (ii) must be performed for label and block before entering the call procedure. On return step (i) must be performed for label and block. 120 - 140 micro-seconds is included in the printout for label and 300 - 350 micro-seconds is included for the calling block or procedure for each such call.
Graphical Representation of Significant Sections of Table 1

Labels as printed by the compiler
3.2 Tests and results on ALGOL/STAGE2/V2

Tests on small test-data showed a very definite pattern of CPU time spent and frequency of access of various sections. A typical results printout is given in Table 1.

The test-data used is a typical input to a macro-processor. It is a set of macros written to expand STAGE2 together with a tenth of STAGE2 PLIB instructions. This consisted of an economical test-data that used most of the facilities provided by STAGE2 in a typical manner.

A considerable percentage of these timings is due to the time required to execute statistics collecting code itself as outlined in 3.1. The last column in Table 1 include an estimate of the time required by the actual code of the program, obtained by using information given in 3.1. Also given in Table 2 is the timings printout for the same test-data when the LABELS option is not set. A graphical representation of the estimated times is also given. Sections of the program where calls to procedures are made must be known to make this estimate closer to the real value. The values used for this estimation is the lowest values given in 3.1.

3.3 Discussion

Five important sections in STAGE2 program are located as a result of the above tests. These are

(i) Procedure IMUCH
(ii) LOC 58 - LOC 70 (Compiler number 71-80)
(iii) LOC 3
(iv) LOC 7
(v) LOC 57

These are presented in the above form because each of them constitutes a separate logical section. Other sections that are worth taking note of are

(vi) LOC 95 (Compiler number 110)
(vii) LOC 2
(viii) LOC 4
(ix) LOC 6
<table>
<thead>
<tr>
<th>BLOCK</th>
<th>TOTAL TIME</th>
<th>AVG TIME</th>
<th>STD TOT TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000</td>
<td>1.404228</td>
<td>1.404228</td>
<td>0.242922</td>
</tr>
<tr>
<td>1</td>
<td>0.638246</td>
<td>1.371</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.5303</td>
<td>0.9204</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.445486</td>
<td>1.5466</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.9404</td>
<td>1.9404</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.185509</td>
<td>1.4960</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.196322</td>
<td>4.0226</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.0403</td>
<td>3.0377</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3.5907</td>
<td>3.3536</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.350506</td>
<td>7.64</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.58633</td>
<td>4.4501</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0.192924</td>
<td>6.4308</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0.144714</td>
<td>2.1171</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>24.02714</td>
<td>24.827174</td>
<td>0.242922</td>
</tr>
<tr>
<td>14</td>
<td>71962</td>
<td>423</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>629738</td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>292977</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>235098</td>
<td>719</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>13104</td>
<td>410</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>131165</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>259052</td>
<td>296</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>285012</td>
<td>286</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>120805</td>
<td>449</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>137549</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>11408</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>11232</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>149952</td>
<td>274</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>132</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>132</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>382</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>288</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>10906</td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1260</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>41676</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>66612</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>5923</td>
<td>359</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>2442</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>19716</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>19680</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>13106</td>
<td>298</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>190995</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>278</td>
<td>289</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>32794</td>
<td>294</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>30242</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>9149</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>4682</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>7373</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>13966</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>7925</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>174971</td>
<td>405</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>72925</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>149630</td>
<td>509</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>23930</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>13497</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>105256</td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>0.620297</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>282398</td>
<td>249</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>297503</td>
<td>219</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>671214</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>0.149642</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>147938</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>175859</td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>46230</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>60576</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0.152568</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>0.620297</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>0.152568</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>0.152568</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>0.152568</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>0.152568</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>0.152568</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>15722</td>
<td>185</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>185423</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>22338</td>
<td>282</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>13970</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>13980</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>2178</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>25056</td>
<td>236</td>
<td></td>
</tr>
</tbody>
</table>
TABLE 1 Continued:

<table>
<thead>
<tr>
<th>BLOCK</th>
<th>FREQ</th>
<th>TOTAL TIME</th>
<th>AVG TIME</th>
<th>ESTD TOT TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
<td>178</td>
<td>23642</td>
<td>232</td>
<td>9968</td>
</tr>
<tr>
<td>94</td>
<td>16</td>
<td>4149</td>
<td>259</td>
<td>2224</td>
</tr>
<tr>
<td>95</td>
<td>105</td>
<td>17362</td>
<td>165</td>
<td>4725</td>
</tr>
<tr>
<td>96</td>
<td>105</td>
<td>15154</td>
<td>144</td>
<td>2520</td>
</tr>
<tr>
<td>97</td>
<td>89</td>
<td>14090</td>
<td>158</td>
<td>3582</td>
</tr>
<tr>
<td>98</td>
<td>1</td>
<td>154</td>
<td>154</td>
<td>34</td>
</tr>
<tr>
<td>102</td>
<td>1</td>
<td>197</td>
<td>197</td>
<td>67</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>194</td>
<td>194</td>
<td>64</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>170</td>
<td>170</td>
<td>4400</td>
</tr>
<tr>
<td>106</td>
<td>1</td>
<td>141</td>
<td>141</td>
<td>1864</td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>184</td>
<td>184</td>
<td>960</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>373</td>
<td>373</td>
<td>0.523457</td>
</tr>
<tr>
<td>111</td>
<td>2</td>
<td>6595</td>
<td>227</td>
<td>3103</td>
</tr>
<tr>
<td>112</td>
<td>1</td>
<td>329</td>
<td>329</td>
<td>329</td>
</tr>
<tr>
<td>113</td>
<td>1</td>
<td>127</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>114</td>
<td>1</td>
<td>22937</td>
<td>22937</td>
<td></td>
</tr>
</tbody>
</table>

Note: For location numbers in STAGE2 corresponding to the numbers in the above table, see Appendix E

TABLE 2

<table>
<thead>
<tr>
<th>BLOCK</th>
<th>FREQ</th>
<th>TOTAL TIME</th>
<th>AVG TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.0000</td>
<td>1</td>
<td>1.138608</td>
<td>1.138608</td>
</tr>
<tr>
<td>INOP</td>
<td>622</td>
<td>0.408878</td>
<td>657</td>
</tr>
<tr>
<td>IREAD</td>
<td>405</td>
<td>0.431618</td>
<td>928</td>
</tr>
<tr>
<td>IWRIT</td>
<td>157</td>
<td>0.789989</td>
<td>5032</td>
</tr>
<tr>
<td>INPFN</td>
<td>3</td>
<td>0.210672</td>
<td>70224</td>
</tr>
<tr>
<td>IWRCH</td>
<td>4011</td>
<td>11.435494</td>
<td>2377</td>
</tr>
<tr>
<td>PROGR</td>
<td>1</td>
<td>7.614737</td>
<td>7.614737</td>
</tr>
</tbody>
</table>
Procedure IRCH:

This is a procedure which is part of the I/O package. This is called to output each character of a record that is to be written on one of the channels provided by STAGE2. The large amount of time spent inside this procedure could be attributed to the fact that it has a local array declaration. The solution in this case is quite straightforward and that is to modify the I/O package so that this set of code is inserted into the main STAGE2 procedure.

LOC 58 - 70

This section of the STAGE2 program does the tree matching process (5), which is used to recognize macro calls and setting up parameters of the call. Thus reviewing the way this section is implemented seems worthwhile.

LOC 3

This simply consists of six FLUB instructions. This code read in the remainder of an input record after an end of line flag, into FLUB memory. The purpose of this is to indicate the last record that was read, in the event of an error occurring.

LOC 7

This section is where the code body of a called macro is extracted from FLUB memory and actions taken accordingly. Thus this section involves frequent executions of the GET instruction of the FLUB language.

LOC 57

This section also consist of eight FLUB instructions. This section gets the characters of a record to be output from FLUB memory into the output buffer.

LOC 95

This section is same as LOC 57 where record in FLUB memory is transferred to output buffer. This section handles the writing out of error traceback.
LOC 2

This section precedes the section LOC 3 mentioned before, in the STAGES2 program. The input record up to end of line flag is transferred into FLUB memory in this section. The record is stored in FLUB memory as linked list of character string.

LOC 4

This section of the program does the parameter initialisation etc. before recognising a macro call.

LOC 6

This is where a check for skipping lines is made and if lines has to be skipped the end of line of each code body line is obtained and skipped appropriately. Again it appears the GAT instruction is executed many times.

3.4 Alternative Solutions

Many modifications aimed at improving these sections can be suggested. Modification of the I/O package eliminating the FIND procedure is an obvious solution. The number of times the code in LOC 3 is executed can be reduced in two ways. These are

(i) Since all that LOC 3 is doing is to read in that part of the record that is not really used by the program this code could be removed from the program. The purpose of this code can then be achieved as follows. The last record read in can be kept in the input buffer and printed out in the event of error occurring. Since there is a single section that outputs error tracebacks (LOC 95) extra code can be inserted in this section to print out the input buffer which will contain the last record that was read in.

(ii) This solution aims at reducing the number of times LOC 3 is executed rather than eliminating it. While STAGES2 extracts characters from input buffer it detects end of record by a special flag which will not be a valid character. If after reading in each record the I/O package inserts this special flag at the end of the comments that may follow the end of line flag then reading in the blank sections of the record will be
avoided, thus reducing the number of times LOC 3 is executed.

Implementing first solution would mean either adding an extra instruction to FLUB language to write out the last record read in from the input buffer and including this FLUB instruction to STAGE2 or changing the expanded STAGE2 by hand. Because of this difficulty it was decided to implement the second solution, which will be equally efficient if the amount of comments on the input records are small.

LOC 57 will be improved when the IMRCH procedure is eliminated as this section calls this procedure repeatedly to output characters to line buffer. If STAGE2 constructed its lines in successive FLUB memory locations then it would in fact be possible to output the constructed line directly to the device i.e. letting the system routines to build up the output buffer. But the way LOC 57 operates indicates that the constructed line is stored as a linked list which means it is necessary to execute the code in LOC 57. In fact even the input records can be read directly into memory avoiding the input buffer and FLUB register transfers. But again information is read in as a linked list which necessitates the above transfers. Therefore no specific improvements could be suggested for this section of code.

Next section that is of interest is from LOC 58 - 70, which constitutes the routine called scanner. The scanner uses and builds trees structured in a fashion (4,5), that do not always use all the three fields of FLUB word. It is easy to deduce from the algorithm that is used for tree matching that FLUB memory access will be made many times. Thus a more efficient implementation of the GET instruction is worth considering.

In ALGOL/STAGE2/V2 GET instruction obtains each field of a word into a B6700 word - see Appendix B. Since there are sections of STAGE2 which do not always use all the three fields of the FLUB register it seemed that a better way of implementing FLUB register - memory id to have a single B6700 word for each register and extract the fields as required for doing operations on it. The sections of STAGE2 that do not use all
three fields of the register all the time includes tree storage, code body storage and character strings storage.

But such an implementation will increase the time required to execute some other MULB instructions. To estimate the effects this implementation will have on various MULB instructions a comparative study of the instruction timings for alternative implementations was undertaken. The alternatives include

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Number of B6700 words</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a  b  c  d</td>
</tr>
<tr>
<td>one MULB word</td>
<td>3  1  1  1</td>
</tr>
<tr>
<td>one MULB register</td>
<td>3  1  3  2</td>
</tr>
</tbody>
</table>

Word formats for these implementations are given in Appendix C. Each column represents a possible alternative implementation. Implementation (a) and (b) are the two extremes and other possibilities which is in between these two can be devised. The table below gives the result of the study for the 4 possible implementation described in Table 3. The ALGOL/STAGE2/V2 is same as implementation (c) in Table 3.

<table>
<thead>
<tr>
<th>Table 4</th>
<th>Time in microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULB instructions</td>
<td>a  b  c  d</td>
</tr>
<tr>
<td>GET $\beta = \beta$</td>
<td>23  10  35  19.5</td>
</tr>
<tr>
<td>STO $\beta = \beta$</td>
<td>21.5  6.5  28.5  11.5</td>
</tr>
<tr>
<td>CONDITIONAL BRANCH instruction</td>
<td>6  7.5  6.0  7.5</td>
</tr>
<tr>
<td>PTR $\beta = \beta + \beta$</td>
<td>6.8  10  6.8  6.8</td>
</tr>
<tr>
<td>VAL $\beta = \beta + \beta$</td>
<td>6.8  13  6.8  13</td>
</tr>
<tr>
<td>PTR $\beta = \beta + 0$</td>
<td>4.8  6.5  4.8  4.8</td>
</tr>
<tr>
<td>VAL $\beta = \beta + 0$</td>
<td>4.8  6.5  4.8  6.5</td>
</tr>
<tr>
<td>PTR * / divide and multiply by 10</td>
<td>15  20  15  15</td>
</tr>
</tbody>
</table>
Since STAGE2 predominantly uses only a subset of the total PLUB instructions, these were borne in mind when making the above study. For example most of the PLUB instructions used in STAGE2 involves one fixed value and one or two variable values. Thus if these fixed values are taken into account while expanding STAGE2 then some of the instruction timings can be reduced. PTR \( \div \) instruction almost always uses a fixed value as the third operand. PTR multiplication and division instructions except for arithmetic expression handling is used only for character to integer and integer to character conversions where the third operand is always 10.
CHAPTER 4

CHOICE OF SOLUTION AND IMPLEMENTATION - ALGOL/STAGE2/V3

The solution that was implemented with the aim of optimizing is alternative (b) in Table 3. Following is a list of reasons that lead to this choice.

(i) As described in 3.3 the template tree matching process is the major section that consumes CPU time within STAGE2 program itself. As this process involves many PLUR memory references before a successful match it seemed appropriate to optimize the GET instruction.

(ii) LOC 7 consists of code which extracts code body elements of a macro from PLUR memory and takes appropriate action. The way code body elements of a macro are stored (4,5) indicate in this situation only VAL field of the register is accessed mostly. Therefore the choice of implementation (b) is further supported.

(iii) In addition to improving GET and STO instructions it is in fact possible to have efficient PTR addition and subtraction operation as explained below.

Let \( \text{PTR } A \rightarrow A + 1 \) be a PLUR instruction. In a general expansion this will expand to,

\[
\text{RA:=} (\text{RA}&[38:11:12]+1)&\text{RA}[38:38:12];
\]

If \( \text{PTR } A \rightarrow 1 \) then RA:=*+1; will be a legal expansion.

Because under this condition the PLG and VAL fields will not be affected. Though most of the PTR operations in STAGE2 are of the above form they must be expanded for a general case. If necessary heavily used sections can then be hard coded whenever the above conditions are satisfied. It is in fact easy to ascertain that this is the case whenever the PTR field is used as a pointer into memory. However these hard coding changes have not been incorporated in ALGOL/STAGE2/V3 as the improvements obtained
by changing I/O package seems to overshadow the improvements that will be obtained by this.

(iv) Though many other MLUB instructions timings are higher for this implementation it must be remembered that fields of a memory word that is accessed by GET instruction is tested once and actions taken accordingly. Therefore not all of the increased instruction timings are real inefficiency of the implementation.

More details of this implementation which has the modified I/O package attached to it is given in Appendix D. A listing of the macro definitions used for this implementation is also given in Appendix E.
CHAPTER 5
MEASUREMENTS OF IMPROVEMENTS

A comparative study of improvements for three test data are presented for the following implementations.

(i) ALGOL/STAGE2/V2
(ii) Implementation (b) of STAGE2 with old I/O package
(iii) ALGOL/STAGE2/V3 (implementation (b) with new I/O package)

Test data I: Expanding STAGE2
Test data II: Test data which is mainly I/O bound
Test data III: Test data which is mainly processor bound

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Test data I</th>
<th>II</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU time for implementation (i)</td>
<td>95.5</td>
<td>15.3</td>
<td>5.91</td>
</tr>
<tr>
<td>do</td>
<td>(ii)</td>
<td>87.4</td>
<td>14.8</td>
</tr>
<tr>
<td>do</td>
<td>(iii)</td>
<td>41.6</td>
<td>3.5</td>
</tr>
</tbody>
</table>

From Table 2 it can be deduced that 50% of the total time is spent inside IRNCH procedure, 35% inside STAGE2 program and the rest of the time in other I/O routines. Thus for Test data II which is mainly I/O bound no improvements is noticeable until the new I/O package is attached. The eight seconds improvement between implementation (i) and (ii) for Test data I is due to STAGE2 program, since 35% of the time is spent in STAGE2 program, eight seconds improvements is equivalent to

\[
\frac{8 \times 100}{35 \times 95.5} = 25\% \text{ improvement of STAGE2 program.}
\]

This is further supported by the results of Test data III which is mainly processor bound. Therefore no improvements is obtained in this Test data by attaching the new I/O package.
CHAPTER 6

SUMMARY

The performance measurement and improvements done on STAGE2 program clearly indicated two major disadvantages of FLUB machine,

(i) The linked list storage used for character strings causes inefficiencies.

(ii) The I/O interface with the real machine was still taking 40% of the total macroprocessing time.

These disadvantages were pointed out by Waite (3). Thus it appears an abstract machine with three word types namely character, integer, tree node could have better suited a macroprocessor like STAGE2. This can be seen quite clearly in regards to the R6700 implementation. A character word type would have meant eliminating the transferring of input and output records character by character to and from a link list. Records can be read in and written out directly from the STAGE2 memory, effectively reducing the I/O time spent. Also a separate integer word type would improve the timing for all integer arithmetic operations and at the same time eliminating the frequent execution of the time consuming GNT and STO instruction.
REFERENCES


(4) Waite W N, 'Implementing Software for Non-numeric Applications' PRENTICE-HALL, pp 319-450

APPENDIX A

The FLUB Machine

(a) Data transfer operations

\begin{align*}
\text{GET} \; \& = \; \\
\text{STO} \; \& = \;
\end{align*}

\begin{align*}
\text{FLG} \; \& = \; \\
\text{VAL} \; \& = \; \text{PTR} \; \\
\text{PTR} \; \& = \; \text{VAL} \\
\end{align*}

(b) Integer arithmetic operations

\begin{align*}
\text{VAL} \; \& = \; + \; \\
\text{VAL} \; \& = \; - \\
\text{PTR} \; \& = \; + \\
\text{PTR} \; \& = \; - \\
\text{PTR} \; \& = \; \times \\
\text{PTR} \; \& = \; \\
\end{align*}

(c) Control operations

\begin{align*}
\text{TO} \; \&. \\
\text{STOP}. \\
\text{TO} \; \& \text{IF FLG} \; \& = \; \\
\text{TO} \; \& \text{IF FLG} \; \& \text{NE} \\
\text{TO} \; \& \text{IF VAL} \; \& = \; \\
\text{TO} \; \& \text{IF VAL} \; \& \text{NE} \\
\end{align*}

(d) I/O operations

\begin{align*}
\text{VAL} \; \& = \; \text{CHAR} \\
\text{CHAR} \; = \; \text{VAL} \\
\text{READ} \; \&. \\
\text{WRITE} \; \&. \\
\text{REWIND} \; \\
\end{align*}

(e) Pseudo operations

\begin{align*}
\text{LOC} \; \\
\end{align*}

END PROGRAM.

Fig. 9.10. FLUB operations.

FLUB Register and Word Format

<table>
<thead>
<tr>
<th>FLG</th>
<th>VAL</th>
<th>PTR</th>
</tr>
</thead>
</table>

FLG field must be at least 2 bits

VAL field must be at least 8 bits. It must hold a character or length of a character string.

PTR field must be able to address memory and capable of integer arithmetic.

GET and STO instructions loads and stores a register respectively, with the memory word addressed by the appropriate PTR field.

FLUB machine has 36 registers 0-9, A-Z and as much memory as required.
APPENDIX B

ALGOL/STAGE2/V2 Implementation Details

Each field of the FLUB register is a B6700 integer word.

Each FLUB memory word is a word in B6700 ALPHA array.

Format of this array word

<table>
<thead>
<tr>
<th>Bits</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:2</td>
<td>PLG field</td>
</tr>
<tr>
<td>45:16</td>
<td>(VAL + 1)</td>
</tr>
<tr>
<td>35:1</td>
<td>Sign of PTR field</td>
</tr>
<tr>
<td>34:35</td>
<td>PTR field</td>
</tr>
</tbody>
</table>

I/O Package

Read, Write and Rewind operations of the FLUB language is implemented by calling the IOP procedure with appropriate parameters.

This procedure has local procedures IREAD, IWRIT and IOPEN to implement the appropriate operation.

A single ALPHA array is used as input and output buffers.

Reading off the buffer (VAL ≠ CHAR) is implemented as a Replace Statement.

Filling the buffer for output (CHAR = VAL) is implemented by procedure IWRCH which again uses pointer Replace statement.

STAGE2 procedure itself is implemented as procedure PROGR.
APPENDIX C

Word Formats for Alternative Implementations for which Timing Measurements are shown in Table 4

Implementation (a)
Each field in PLUB register is a B6700 integer word.
Each field in PLUB memory is a word in a B6700 ALPHA array. This means each PLUB memory word is contained in three words of three different arrays, - one array for each field.

Implementation (b)
This is same as that given in Appendix D.
Each PLUB register is a B6700 integer word.
Each PLUB memory word is a word in a B6700 INTEGER array.
The fields are packed as follows for PLUB register and memory:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>46:1</td>
<td>sign of the PTR field</td>
</tr>
<tr>
<td>38:2</td>
<td>FLG field</td>
</tr>
<tr>
<td>36:16</td>
<td>(VAL + 1)</td>
</tr>
<tr>
<td>26:27</td>
<td>PTR field</td>
</tr>
</tbody>
</table>

Implementation (c)
This is same as that given in Appendix B
Each field of the PLUB register is a B6700 integer word
Each PLUB memory word is a word in B6700 ALPHA array

Format of this array word

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:2</td>
<td>FLG field</td>
</tr>
<tr>
<td>45:10</td>
<td>(VAL + 1)</td>
</tr>
<tr>
<td>35:1</td>
<td>sign of PTR field</td>
</tr>
<tr>
<td>34:35</td>
<td>PTR field</td>
</tr>
</tbody>
</table>
Implementation (d)

Each FLUB register is contained in two D6700 integer words: one word for PTR field and one word for VAL and PLG field.

Each FLUB memory word is a word in a D6700 INTEGER array.

The fields are packed in the same way as for Implementation (b).
APPENDIX D

ALGOL/STAGE2/V3 Implementation Details

Each FLUB register is a B6700 integer word

Each FLUB word is a word in a B6700 INTEGER array

FLUB word and register format:

<table>
<thead>
<tr>
<th>Bits</th>
<th>46:1</th>
<th>sign of PTR field (This is also sign of the integer word)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>38:2</td>
<td>MLG field</td>
</tr>
<tr>
<td>Bits</td>
<td>36:10</td>
<td>(VAL +1)</td>
</tr>
<tr>
<td>Bits</td>
<td>26:27</td>
<td>PTR field</td>
</tr>
</tbody>
</table>

GET $A = B$ instruction expands as follows:

IF GET A = B is the FLUB instruction then index to the array is in B. 26:27

\[ A_i = \text{Array}[B.[26:27]] \]

VAL. Field operations that involve only VAL. fields are done in the domain (VAL +1). Thus all VAL field fixed values such as 0 - 9 etc. are initialised to contain 1 - 10. A subtraction of one and addition of one is required in the case of addition and subtraction operations respectively for VAL fields.

PTR. fields arithmetic operations are expanded so that the MLG and VAL fields of the registers are zeroed before operations but most of this which involve only positive PTR fields could be changed so that only one of the operands' MLG and VAL fields need to be zero.

In STAGE2 PTR field of register 8 and 9 is used as the travelling pointers to keep track of the boundaries of STAGE2 memory. Thus these two fields are kept in separate variables because they are accessed very frequently.
All 'Flag line' characters are kept in VAL field of registers, and these are assigned to separate B6700 variables and recognised when the rest of the macros are expanded. Same is true for fixed values 0 - 9.

I/O Package:

Records are read into and output to a buffer which is an INTEGER array. Thus VAL \( \hat{Y} \) = CHAR and CHAR = VAL \( \hat{Y} \) involve updating an integer pointer and assigning values to an integer array or vice versa. After reading in a record the blank sections of the record is eliminated by inserting carriage return flag after the last non-blank character.

STAGE2 program itself is implemented as procedure PROGR.
The Order of Appearance of Labels in STAGE2 for use with Compiler Print Out of Statistics

<table>
<thead>
<tr>
<th>CH</th>
<th>LH</th>
<th>CH</th>
<th>LH</th>
<th>CH</th>
<th>LH</th>
<th>CH</th>
<th>LH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100,1</td>
<td>26</td>
<td>21</td>
<td>51</td>
<td>42</td>
<td>76</td>
<td>63</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>27</td>
<td>22</td>
<td>52</td>
<td>111</td>
<td>77</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>28</td>
<td>23</td>
<td>53</td>
<td>43</td>
<td>78</td>
<td>65</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>29</td>
<td>107</td>
<td>54</td>
<td>112</td>
<td>79</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>30</td>
<td>32</td>
<td>55</td>
<td>44</td>
<td>50</td>
<td>67</td>
</tr>
<tr>
<td>6</td>
<td>102,5</td>
<td>31</td>
<td>24</td>
<td>56</td>
<td>45</td>
<td>81</td>
<td>68</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>32</td>
<td>25</td>
<td>57</td>
<td>46</td>
<td>82</td>
<td>69</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>33</td>
<td>26</td>
<td>58</td>
<td>47</td>
<td>83</td>
<td>70</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>34</td>
<td>27</td>
<td>59</td>
<td>48</td>
<td>84</td>
<td>71</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>35</td>
<td>28</td>
<td>60</td>
<td>113</td>
<td>85</td>
<td>72</td>
</tr>
<tr>
<td>11</td>
<td>103</td>
<td>36</td>
<td>31</td>
<td>61</td>
<td>99</td>
<td>86</td>
<td>73</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>37</td>
<td>29</td>
<td>62</td>
<td>49</td>
<td>87</td>
<td>74</td>
</tr>
<tr>
<td>13</td>
<td>104</td>
<td>38</td>
<td>30</td>
<td>63</td>
<td>50</td>
<td>88</td>
<td>75</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>39</td>
<td>33</td>
<td>64</td>
<td>51</td>
<td>89</td>
<td>76</td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>40</td>
<td>108</td>
<td>65</td>
<td>52</td>
<td>90</td>
<td>77</td>
</tr>
<tr>
<td>16</td>
<td>105</td>
<td>41</td>
<td>34</td>
<td>66</td>
<td>53</td>
<td>91</td>
<td>78</td>
</tr>
<tr>
<td>17</td>
<td>13</td>
<td>42</td>
<td>35</td>
<td>67</td>
<td>54</td>
<td>92</td>
<td>79</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>43</td>
<td>36</td>
<td>68</td>
<td>55</td>
<td>93</td>
<td>80</td>
</tr>
<tr>
<td>19</td>
<td>15</td>
<td>44</td>
<td>37</td>
<td>69</td>
<td>56</td>
<td>94</td>
<td>90</td>
</tr>
<tr>
<td>20</td>
<td>106</td>
<td>45</td>
<td>38</td>
<td>70</td>
<td>57</td>
<td>95</td>
<td>81</td>
</tr>
<tr>
<td>21</td>
<td>16</td>
<td>46</td>
<td>39</td>
<td>71</td>
<td>58</td>
<td>96</td>
<td>82</td>
</tr>
<tr>
<td>22</td>
<td>17</td>
<td>47</td>
<td>109</td>
<td>72</td>
<td>59</td>
<td>97</td>
<td>83</td>
</tr>
<tr>
<td>23</td>
<td>18</td>
<td>48</td>
<td>110</td>
<td>73</td>
<td>60</td>
<td>98</td>
<td>84</td>
</tr>
<tr>
<td>24</td>
<td>19</td>
<td>49</td>
<td>40</td>
<td>74</td>
<td>61</td>
<td>99</td>
<td>85</td>
</tr>
<tr>
<td>25</td>
<td>20</td>
<td>50</td>
<td>41</td>
<td>75</td>
<td>62</td>
<td>100</td>
<td>86</td>
</tr>
</tbody>
</table>
APPENDIX F.

MACRO DEFINITIONS USED FOR ALGOL/STAGE2/V3

FLAG LINE CHARACTERS ARE: F2 F3 F4 F5

SWITCH

5 'F2$
$
AT STORE
$
'F3$
$
IF IS SKIP
$
'F50$
$
SKIP
$
'F4$

GET =

P '20'36$
$'31'46$

IF '40 IS SKIP 2$
$'40'26$

SKIP 1$

R '20+27'26$
R '+10+='20+1'F1$

STO =

P '10'36$
$'31'46$

IF '40 IS SKIP 2$
$'40'16$

SKIP 1$

R '10+27'16$
L '10+='20+1'F1$

VAL = PTR

P '20'46$
$'41'46$

IF '50 IS SKIP 2$
$'50'26$

SKIP 1$

R '20+10+26$
R '10+101=20+1'F1$

PTR = VAL

R '20+46$
$'41'46$

IF '50 IS SKIP 2$
$'50'26$

SKIP 1$

R '20+27'26$
R '+10+='20+1'F1$

VAL =

P '40'56$
$'41'46$

IF '50 IS SKIP 2$
$'50'46$

SKIP 1$

R '20+10'26$
R '10+101=20+1'F1$

VAL =

P '40'56$
$'41'46$

IF '50 IS SKIP 2$
$'50'46$

SKIP 1$

R '20+36'26$
R '10+101=20+1'F1$

VAL =

P '20'36$
$'31'46$

IF '40 IS SKIP 2$
$'40'26$

SKIP 1$

R '20+36'26$

```
R'10(36:10) = '20(31)F1$

$ TU ' IF FLG ' ' ' ' ' ' ' 
P'40'56$
  '51'66$
  IF '50 IS SKIP 2 $
  '50'46$
  SKIP 1 $
  R'40.0[36:10]46$
  IF '20.[36:12]31' THEN GU TO L'10';F1$

$ TU ' IF VAL ' ' ' ' ' ' ' 
V'40'56$
  '51'66$
  IF '50 IS SKIP 2 $
  '50'46$
  SKIP 1 $
  R'40.[36:10]46$
  IF '20.[36:10]31' THEN GU TO L'10';F1$

$ TU ' IF PTR ' ' ' ' ' ' ' 
P'20'56$
  '51'66$
  IF '50 IS SKIP 2 $
  '50'26$
  SKIP 1 $
  R'20&0[38:11:12]26$
  P'40'56$
  '51'66$
  IF '50 IS SKIP 2 $
  '50'46$
  SKIP 1 $
  R'40&0[38:11:12]46$
  IF '20.'31'40 THEN GU TO L'10';F1$

$ PTR ' ' 
P'40'56$
  '51'66$
  IF '60 IS SKIP 2 $
  '60'46$
  SKIP 1 $
  R'40&0&38:11:12]46$
  R'10+R'20&0[36:11:12]31'40) & R'10[38:38:12]F1$

$ PTR 8 ' ' 
P'30'46$
  '41'56$
  IF '50 IS SKIP 2 $
  '50'36$
  SKIP 1 $
  R'30&0[38:11:12]36$
  IF '20 IS 9 SKIP 2 $
  P81=R'10&0[38:11:12]21 '30;F1$

$ PTR 9 ' ' 
P'30'46$
  '41'56$
  IF '50 IS SKIP 2 $
  '50'36$
  SKIP 1 $
  R'30&0[38:11:12]36$
  IF '20 IS 9 SKIP 2 $
  P91=R'10&0[38:11:12]21 '30;F1$

$ PTR 10 ' ' 
P'30'46$
  '41'56$
  IF '10 IS '20 SKIP 1 $
  R'10+R'20&0[38:11:12]F1$

$ PTR 8 ' ' 
P'10'46$
  '21'57$
  IF '10 IS 8 SKIP 1 $
  P81=R'10&0[38:11:12]F1$

$ PTR 9 ' ' 
P'10'46$
  '21'57$
  IF '10 IS 9 SKIP 1 $
```
P91=R'10&0[38h11112]H'F1$
S
PTR' = ' + '.
P$30.465$
'41.565$
IF'50 IS SKIP 2$
'50.368$
SKIP 1$
R'30&0[38h11112]H'36$
R'101=(R'20&0[38h11112]=H'30)&R'101[38h38h12]H'F1$
S
PTR' = ' + '.
P$30.465$
'41.565$
IF'50 IS SKIP 2$
'50.368$
SKIP 1$
R'30&0[38h11112]H'36$
R'101=(R'20&0[38h11112]=H'30)&R'101[38h38h12]H'F1$
S
PTR' = 8 + '.
P$20.465$
'41.565$
IF'50 IS SKIP 2$
'50.266$
SKIP 1$
R'20&0[38h11112]H'26$
R'101=(P9+H'20)&R'101[38h38h12]H'F1$
S
PTR' = 8 - '.
P$20.465$
'41.565$
IF'50 IS SKIP 2$
'50.266$
SKIP 1$
R'20&0[38h11112]H'26$
R'101=(P9+H'20)&R'101[38h38h12]H'F1$
S
PTR' = 9 + '.
P$20.465$
'41.565$
IF'50 IS SKIP 2$
'50.266$
SKIP 1$
R'20&0[38h11112]H'26$
R'101=(P9+H'20)&R'101[38h38h12]H'F1$
S
PTR' = 9 - '.
P$20.465$
'41.565$
IF'50 IS SKIP 2$
'50.266$
SKIP 1$
R'20&0[38h11112]H'26$
R'101=(P9+H'20)&R'101[38h38h12]H'F1$
S
PTR' = ' + 0.
IF'20 IS 0 SKIP 6$
P$20.365$
'31.465$
IF'40 IS SKIP 2$
'40.265$
SKIP 1$
R'20.26$
R'101=(P9+H'20)&R'101[38h38h12]H'F1$
S
PTR' = ' + 8.
IF'20 IS 0 SKIP 11$
P$20.465$
'41.565$
IF'50 IS SKIP 2$

'50'26$  
SKIP 1 $  
R'20[0][38+11\{12]+'26$  
IF '10 IS 9 SKIP 2 $  
P9'i10'60[38+11\{12]+'20J'F1$  
SKIP 1 $  
P8'i'20J'F1$  
$  
PTR 9 = '1' - '1'.  
P'20'46$  
'41'56$  
IF '50 IS SKIP 2 $  
'5026$  
SKIP 1 $  
R'20[0][38+11\{12]+'26$  
IF '10 IS 9 SKIP 2 $  
P9'i10'60[38+11\{12]+'20J'F1$  
SKIP 1 $  
P9'i'20J'F1$  
$  
PTR 9 = '1' + '1'.  
IF '20 IS 0 SKIP 11 $  
P'20'46$  
'41'56$  
IF '50 IS SKIP 2 $  
'5026$  
SKIP 1 $  
R'20[0][38+11\{12]+'26$  
IF '10 IS 9 SKIP 2 $  
P9'i10'60[38+11\{12]+'20J'F1$  
SKIP 1 $  
P9'i'20J'F1$  
$  
TO '1'.  
GU TO L'10J'F1$  
$  
STOP.  
GU F11'F1$  
$  
TO '1' BY '1'.  
'00+99'96$  
R'20[0][26+27]+'00J'F1$  
GU TO L'10J'94'F1$  
$  
RETURN BY '1'.  
GU TO SIR'10[4\{15]+'F1$  
$  
VAL ' = CHAR.  
R'10[36+10]=LBI[INDX]+1; IF LBI[INDX]=1 THEN INDX=1 ELSE INDX=++1'F1$  
IF '10 IS R SKIP 1 $  
SKIP 14 $  
PE'i0&REF[26126[127]+'F1$  
VE'i0&REF[9136110]+'F1$  
VA'i0&REF[9136110]+'F1$  
VD'i0&REF[9136110]+'F1$  
VC'i0&REF[9136110]+'F1$  
VD'i0&REF[9136110]+'F1$  
VE'i0&REF[9136110]+'F1$  
VF'i0&REF[9136110]+'F1$  
VM'i0&REF[9136110]+'F1$
VHI = 0 & RN[9:36:10] 1 F1$
VOL = 0 & RT[9:36:10] 1 F1$
VP1 = 0 & RP[9:36:10] 1 F1$
VOL = 0 & RQ[9:36:10] 1 F1$
VLO = 0 & RQ[9:36:10] 1 F1$

CHAR = VAL 1
IF R10.x[36:10] LSS 1 OR INDX GTR 80 THEN BEGIN F1$
BEGIN LBL[INDEXI]; LBLM = INDEXI; R10.x[36:2] = INDEXI = 1 END; F1$
ELSE BEGIN LBL[INDEXI] = R10.x[36:10] - 1; F1$
INDEXI := INDEXI + 1; R10.x[38:21] = 0 END; F1$

READ NEXT 1
CH = R10.x[36:10] - 1; F1$
R10.x[38:2] = 100P(1, CH, LB, LBLM) 1 INDXI = 1; F1$

WRITE NEXT 1
CH = R10.x[36:10] - 1; F1$
R10.x[38:2] = 100P(1, CH, LB, LBLM) 1 INDXI = 1; F1$

REWIND 1
CH = R10.x[36:10] - 1; F1$
R10.x[38:2] = 100P(0, CH, LB, LBLM) 1 F1$

LOC 1
L = 101; F1$

MESSAGE 1 TO 1
A4 = JR4.x[38:2] = 100P(1, CH, 10, 20) 1 F1$

END PROGRAM 1
FINI END; F1$

AT V0 STORE 1$
AT V1 STORE 2$
AT V2 STORE 3$
AT V3 STORE 4$
AT V4 STORE 5$
AT V5 STORE 6$
AT V6 STORE 7$
AT V7 STORE 8$
AT V8 STORE 9$
AT V9 STORE 10$
AT VA STORE VA$
AT VB STORE VB$
AT VC STORE VC$
AT VD STORE VD$
AT VE STORE VE$
AT VF STORE VF$
AT VL STORE VL$
AT VM STORE VM$
AT VN STORE VN$
AT V0 STORE V0$
AT VP STORE VP$
AT VQ STORE VQ$
AT VR STORE VR$
AT PE STORE PE$
AT PO STORE PO$
AT PE STORE PE$
AT PS STORE PS$
AT P5 STORE P5$
AT PH STORE PH$
AT P7 STORE P7$
AT P1 STORE P1$
AT P2 STORE P2$
AT P3 STORE P3$
AT P4 STORE P4$
AT P5 STORE P5$
AT P6 STORE P6$
AT P7 STORE P7$
AT + STORE +$
AT - STORE -$
AT = STORE =$
AT NE STORE NE0.