A language to describe and simulate multiprocessor computer systems

Donald Earl Elliott
Iowa State University
INFORMATION TO USERS

This dissertation was produced from a microfilm copy of the original document. 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 original submitted.

The following explanation of techniques is provided to help you understand markings or patterns 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 thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

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

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
A Xerox Education Company
73-9443

ELLIOTT, Donald Earl, 1938-
A LANGUAGE TO DESCRIBE AND SIMULATE MULTIPROCESSOR
COMPUTER SYSTEMS.

Iowa State University, Ph.D., 1972
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan
A language to describe and simulate
multiprocessor computer systems

by

Donald Earl Elliott

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa
1972
PLEASE NOTE:

Some pages may have
indistinct print.
Filmed as received.

University Microfilms, A Xerox Education Company
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II</td>
<td>LITERATURE REVIEW</td>
<td>4</td>
</tr>
<tr>
<td>III</td>
<td>PROGRAMMING SYSTEM</td>
<td>8</td>
</tr>
<tr>
<td>IV</td>
<td>MULTIPROCESSOR DESCRIPTION</td>
<td>12</td>
</tr>
<tr>
<td>V</td>
<td>SIMULATION</td>
<td>36</td>
</tr>
<tr>
<td>VI</td>
<td>SINGLE PROCESSOR EXAMPLE</td>
<td>40</td>
</tr>
<tr>
<td>VII</td>
<td>MULTIPLE PROCESSOR EXAMPLE</td>
<td>47</td>
</tr>
<tr>
<td>VIII</td>
<td>LANGUAGE IMPLEMENTATION</td>
<td>54</td>
</tr>
<tr>
<td>IX</td>
<td>CONCLUSION</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>LITERATURE CITED</td>
<td>64</td>
</tr>
<tr>
<td>A</td>
<td>SYNTAX OF LANGUAGE</td>
<td>66</td>
</tr>
<tr>
<td>B</td>
<td>TRANSLATOR ROUTINES</td>
<td>69</td>
</tr>
<tr>
<td>C</td>
<td>SIMULATION ROUTINES</td>
<td>109</td>
</tr>
<tr>
<td>D</td>
<td>$RUN SOURCE</td>
<td>113</td>
</tr>
<tr>
<td>E</td>
<td>JOB CONTROL</td>
<td>116</td>
</tr>
<tr>
<td>F</td>
<td>SINGLE TEL_COMP DESCRIPTION</td>
<td>120</td>
</tr>
<tr>
<td>G</td>
<td>SINGLE TEL_COMP SIMULATION</td>
<td>124</td>
</tr>
<tr>
<td>H</td>
<td>MULTIPLE TEL_COMP DESCRIPTION</td>
<td>172</td>
</tr>
<tr>
<td>I</td>
<td>MULTIPLE TEL_COMP SIMULATION</td>
<td>177</td>
</tr>
</tbody>
</table>
CHAPTER I. INTRODUCTION

Currently there exists a need for a language capable of describing the structure and behavior of multiprocessor computer systems. This need is illustrated by the deficiency of a standard language to describe the construction and behavior of present day systems. The implementation of such a language would allow the computer designer to symbolically design a system, study its behavior and then use this symbolic design as the input to an automated design system to actually construct the system.

The requirements placed upon this language are many and varied. To insure that the resulting symbolic design could actually be the first step in an automated design system the language must reflect the structure or the logical building blocks within the system. This requirement, in and of itself, leads to other requirements. First, the language must accurately represent the interconnection of the iterated logic circuits found in digital computer systems. This representation must be concise and compact for ease of readability and yet reflect the behavior of these logic circuits.

Second, the language must allow the system to be partitioned into the various individual processors within the system. This feature allows the designer to add/delete processors to/from the system and study the effects on the behavior of the system due to these changes.

Third, the language must describe the interconnections between the various processors within the system. This enables the designer to accurately gauge the physical number of input and output lines required.
for each of the processors. Knowing this number may enable the designer to reconfigure his proposed design to reduce the number of these interconnections.

Fourth, the language must provide a convenient way for the designer to specify how and when the possible actions within each of the processors are to take place. This specification of the control signals should not have to be explicitly stated by the use of control signals and should allow for a wide variety of actions to occur in parallel, serial, or a combination of both.

Fifth, the language should provide a convenient way to indicate how the flow of control may be dependent upon the priority between signals. It should also explicitly show how the flow of control within one processor may be dependent not only upon its own internal conditions, but also upon external conditions.

In order for the resulting system to be capable of being effectively simulated other requirements must be placed upon the language. First, the simulation should be done at the highest level consistent with the requirements necessary for automated construction. This level is the register transfer level. There has been much work in the simulation of digital designs at the logic level and using the statistical characteristics of proposed systems, but very little work has been done to simulate multiprocessor systems at the register transfer level.

Second, the simulator should allow the designer to input test cases to check out the logical consistency of the system under consideration.
Other considerations besides the structure and behavior of the system come into play in the implementation of the language. One of these is that the language should be machine readable in order to allow the use of translation techniques already developed for high level languages. Also it allows a more systematic approach to changes to the language if the need arises.

And finally the language should reflect how the designer visualizes the flow of control within computer systems. General purpose languages tend to force the user, in this case the computer designer, into thinking not of how to solve his problem, but rather into how to get the general purpose language to do what he wants it to do.

The work reported on here does not attempt to solve the automated construction of digital systems, but instead focuses its attention on the development of a language to describe multiprocessor computer systems and the simulation of those systems at a high level. Its significance comes not only from the fact that the symbolic language is capable of describing the parallel and serial performance of events within a multiprocessor system, but from the simulation of those events.
CHAPTER II. LITERATURE REVIEW

Gorman and Anderson (12) recognized that computers could aid in the design of other computers. They wrote a program that read in the system structure of registers and transfer paths, and combined these with the hardware characteristics and the desired sequences of register transfers to obtain a new ordered sequences of legitimate register transfers. From these new sequences they formed the necessary Boolean equations to build the system. Case et al. (5) described an automated logic design programming system that documents the logic design for the SYSTEM/360 and allowed for partial simulation of that design.

Proctor (24) in his system LDT (Logic Design Translator) developed logic equations for a stored program digital computer from information contained in its system diagram and the instruction repertoire of the machine. He along with Bell and Newell (2,3), Pumplin (25) and Zucker (29) considered the special case of a stored program computer with its classical "fetch-execute" cycle.

Falkoff et al. (10) used APL (A Programming Language) by Iverson (14, 15) to describe the behavior of the SYSTEM/360. Unfortunately APL doesn't indicate the physical structure of the system because it contains no dimension statements and has operations within it that expand and compress the dimensions of their operands. Taking these discrepancies into account, Friedman (11) used a subset of APL to represent the logic design of a microprogrammed computer and generated its associated Boolean equations.
Schlaepi (26) in his language LOTIS (Logic, Timing, Sequencing) formally presented methods of describing register transfers, timing of those transfers, concurrent transfers and grouping of these transfers. He also suggested that such a language could describe both the structure and behavior of a computer.

Schorr (27) described an algorithm for converting a microprogram consisting of statements with a conditional part and a register transfer part into input equations for R-S flip flops.

Chu (6) suggested that the design language have the form of a high level language such as ALGOL (19). With such a form, the translation techniques developed for high level languages would be applicable to the translation of the computer design language. His language CDL (Computer Design Language) used conditional labels on groups of register transfer statements that were to be done in parallel. If the conditional label was true the statements were to be executed, otherwise they were not. Since he was describing only a synchronous computer, each of the conditional labels had a reference to one of the clock phases. This provided a convenient way of describing the flow of control within a synchronous machine. Pardo (20) demonstrated how CDL by Chu could be used for microprogram design and verification by writing a microprogram for code relocation in CDL. Interactive design and simulation for a language similar to CDL was done by Crall and Tracy (7).

Symbolic dimensioning of registers and memories was suggested by Metzc and Seshu (18), by using a FORTRAN type language as the basis for hardware independent system description language.
An experimental language describing the system as containing subsystems was presented by Parnas and Darringer (22). Each of the subsystems was described either structurally, i.e. their Boolean equations, or behaviorally, i.e. as an ALGOL program (21).

Duley and Dietmeyer (8,9) informally proposed a language for a system of finite automata each having private facilities (registers, memories, etc.) and sharing public facilities. In their descriptive language to specify these interconnected autonomous units, they described the flow of control within each unit as being composed of compatible sets of operations that would be executed simultaneously when the unit was in that state. They then described how the sequencing through these various states was done by a state sequencing register that could be changed by a statement within the language.

Two informal languages are used to describe a computer system by Bell and Newell (2). The interconnections between processor, memories, and associated switches are diagramed along with their gross hardware characteristics. This forms the system language. Each individual processor is then described in a register transfer language.

Stabler (28) used a register transfer language to define the possible microactions for a microprogrammed computer. Then he provided possible transformations on the microprograms which kept the system's behavior the same. Using these transformations the microprogrammer (i.e. the computer designer) can reduce the number of steps in the microprogram by parallelizing mutually exclusive microactions.

The description and simulation for only stored program computers was done by Pumplin (25) and Zucker (29). Simulation of flip flops and
logic elements was described by McClure (16) and Hays (13). McKay (17) suggested building a special purpose computer to do this simulation. Parnas (21) suggested extending ALGOL to simulate the "black box" picture of a system.

Potash et al. (23) translated his design description into a pseudo machine language that when executed on his pseudo machine would simulate the machine described. Whereas Pumplin (25) translated the design language into PL/1 statements which when compiled would act as the simulator.
CHAPTER III. PROGRAMMING SYSTEM

The programming system presented here consists of two parts. The first part is a language to symbolically describe the construction of multiprocessor computer systems. This language is described informally in Chapter IV.

The second primary part is the simulation of the symbolic design being considered. This also requires a language to describe the initial conditions of the processors and when the simulation is to be terminated. This language is described informally in Chapter V.

The implementation details of describing the implementation of these translators, the simulation of the resulting target code and their restrictions are discussed in Chapter VIII.

The examples of the description and simulation of processors are given in Chapter VI and VII. Chapter VI shows the description and simulation of a single processor while Chapter VII describes a multiprocessor system and its associated simulation.

One of the desirable qualities in a language to describe the construction of a multiprocessor system is that the syntax of the language be described formally. In order for this to occur, a meta language must be adopted. The meta language, or rather the language to describe another language, used here was developed by Brooker and Morris (4). It is a modification and extension of the meta language Backus-Naur used to describe ALGOL (19).

This meta language can best be explained by illustrating some of the basic primitives within the language currently to be described, i.e. the
language to describe multiprocessors (MPS). The following example is used to illustrate a simple definition using this meta language:

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

The leftmost item in each definition, in this case DIGIT, indicates the name of the syntactic class that is being defined. The characters "::=" following this name are used to separate the name of the class being defined and its definition. These characters "::=" can be literally read as "is defined to be". Characters belonging to the language being defined are enclosed in quotes (''). If the character quote (') is to be used in the language being defined, then two adjacent quotes ("'") must be used to represent the single quote. The meta character "|" is used to indicate alternates and can literally be read as "or". This statement can now be read as the "syntactic class DIGIT" "is defined to be" the character "0", "or" the character "1", "or" the character "2", etc.

The next example illustrates another symbol used in the meta language, the dollar sign ($). The dollar sign is used to indicate zero or more occurrences of the syntactic class or group following its use. For example the following definition of integer uses this symbol:

\[
\text{INTEGER} ::= \text{DIGIT} \$ \text{DIGIT}
\]

In this example the "syntactic class INTEGER" "is defined to be" one occurrence of the syntactic class DIGIT followed by zero or more occurrences of the syntactic class DIGIT which has already been defined. Some examples of valid integers are 0, 43, 689, etc. Integers, as implemented in MPS,
cannot have imbedded blanks. They are used only to represent values, such as, subscripts, number of lines in buses, and number of words in a memory. Integers are not used to represent constants. A constant, in MPS, is defined as follows:

\[
\text{CONSTANT} ::= 'B''BIT'
\]

\[
\text{BIT} ::= ('0'|'1') \$ ('0'|'1')
\]

This example also illustrates the use of parens in the meta language to indicate grouping of choices within the defined language. A more informal interpretation of the definition for a constant is that it consists of the letter "B" followed by a string of one or more 0's and 1's enclosed in quotes. Some valid constants in the language are B'l', B'0101', B'00001', etc. The length of constants is limited by the implementation to 32 bits.

Another primitive in the language is the name of an identifier. It is defined as follows:

\[
\text{LETTER} ::= 'A'|'B'|'C'| \ldots |'X'|'Y'|'Z'
\]

\[
\text{IDENTIFIER} ::= \text{LETTER} \$ (\text{LETTER}|\text{DIGIT})
\]

Informally the syntactic class identifier is defined to be a letter followed by zero or more letter's or digit's. Some examples of valid identifiers are C, IC, CORE, etc. Identifiers are used in the language to refer symbolically to buses, registers, states, processors, adders and memories.

Two other forms are used in the meta language to assist in the definition of the syntactic classes. One is the special syntactic class named
.EMPTY which is used to indicate that nothing has to be present for the resulting group to be valid. The other form is .OPT(items). In this latter form the items contained within the parens can be optionally present or not for the resulting syntactic class to be valid.

If names of syntactic classes have to be adjacent in the meta language due to the language being defined, these names can be separated by periods to avoid ambiguity.

Describing the language formally in this meta language allows the language to be machine readable and thus machine translatable to another language, i.e. a simulator. The complete formal syntax of the language described in this paper is given in Appendix A.

Comments are not formally allowed in the language, but source cards having an asterisk (*) in column one are only listed with the other source cards and their contents are ignored. This is very similar to most assembly languages.

In order to simplify the construction and implementation of the translator, the language MPS has several reserved words that cannot be used as names or identifiers by the user. These words are as follows: .SYSTEM., .END., BUS, REG, RAM, ROM, CAM, TERM, DEF, TIME, ADD, READ, WRITE, .SIMULATE., SET, ACTIVATE, and TRACE. Also identifiers, integers and keywords cannot have embedded blanks or be continued across card boundaries. Where a single blank can occur, many blanks can occur.
CHAPTER IV. MULTIPROCESSOR DESCRIPTION

The definition of a multiprocessor system as described in this paper consists of one or more processors inter-connected by buses. Each processor is in turn described as containing facilities and states. The facilities for each processor are registers, memories, and adders. The states within each processor are used to describe the control of the processor's facilities and are similar to the states in a Moore Sequential Machine. Each state specifies a set of actions to be performed simultaneously, serially, or a combination of both serial and simultaneous execution. The determination of the next state is based upon a priority scheme among selected conditions. A processor consists of an interconnected set of these states, with only one state active at any one time.

The symbolic description of the processors and the interconnecting buses are enclosed in the words "SYSTEM." and "END."

The BUS statement is used to describe the buses over which information can be communicated between the individual processors within the system. It consists of the word BUS followed by a list of the buses being declared separated by commas. If the bus is to consist of only one line it can be written for example as "A" or as "A<1>"; otherwise the integer enclosed in the brackets "<" and ">" indicates the number of lines within the bus. An example of the BUS statement is shown below:

BUS DATA<12>, ADDR<7>, REQ, ACK
in which there are four buses being declared. Their names are DATA, ADDR, REQ, and ACK. The bus DATA consists of 12 lines, the bus ADDR consists of 7 lines, and the buses REQ and ACK consist of one line each.

Describing how the processors are connected to the buses is best illustrated by an example as shown in figure 1. Figure 1a shows the block diagram of three processors whose names are A, B and C that are interconnected by the buses X, Y, Z and EXT; figure 1b shows how they are described in the language. Whether the buses are used for strictly input or output or a combination of both is not considered in this descriptive language. Whereas register, memory and adder names are local to each processor, the bus names are global to all of the processors within the system. The "wired-or" philosophy has to be done explicitly. The buses are treated in the same manner as registers.

The description of each of the processors consists of describing the processor's name, its interconnections to the other processors, its facilities such as registers, memories, and adders and the order of when the actions read, write, search, transfer and add are to take place.

**Processor's name and interconnections**

The declaration of the processor's name is best illustrated by means of an example as shown in figure 1b. The interconnections between the processors are indicated by writing their name and indicating the number of lines connected to the processor. Whereas X <3> in the BUS statement indicates three lines, when used to indicate that it is an interconnection
Figure 1a. An example of a multiprocessor Block Diagram
.SYSTEM.
    BUS  EXT<4>, X<3>, Y<2>, Z
A:PROCESSOR(X<1:3>, Y<1:2>, EXT<1:4>);
    .
    .
    .
    .
.END.
B:PROCESSOR(X<1:3>, Z<1>);
    .
    .
    .
C:PROCESSOR(Y<1:2>, Z<1>, X<3>);
    .
    .
    .
.END.
.END.

Figure 1b. Description of multiprocessor in MPS
between two processors it refers only to the single line \( X<3> \). Also referring to a bus by its name alone, such as, \( Y \) is the same as writing \( Y<1> \). This is different than the referencing of registers as will be seen later.

**Facilities**

Processor facilities are those registers, memories and adders that are local to (i.e. within) the processor being described. The names of these facilities are local to the processor being described and can be used within the description of other processors. The declaration of these facilities can be done in any order as long as they appear before the description of the states.

**Registers**

Describing the registers local to the processor is done by means of a REG statement. This statement is similar to the BUS statement. It consists of the word REG followed by a list of register names separated by means of commas. The number of bits in the register is determined by the integer number enclosed in brackets following the register name. If the integer is not present then the register is assumed to consist of only one bit. An example of a register statement is shown below:

\[
\text{REG \ ACC}<12>, \ IC<10>, \ AE, \ ON<1>
\]

In this example, the register whose name is "ACC" has 12 bits, the register whose name is "IC" has 10 bits and the registers whose names are "AE" and "ON" have one bit each.
Reference to an individual bit within a register is done by writing the number of the bit being referred to, i.e. the 9th bit of register ACC is referred to by writing ACC<9>. Reference to a group of adjacent bits within a register is done by writing the range of bits being referenced. For example, writing ACC<3:5> references the 3rd, 4th and the 5th bits of register ACC as a group. Reference to the entire register is done by writing the name of the register without any range. For example, writing ACC refers to all 12 bits of the register as a group.

All physical registers, other than those associated with memories, must be declared by means of the register statement.

Memories

There are three types of memories allowed in this language. They are Random Accessed Memories (RAM's), Read Only Memories (ROM's) and Content Addressable Memories (CAM's). The RAM, ROM, and CAM statements in the language provide the user with the ability to declare the type of memory wanted for a given processor. More than one memory can be declared within a processor.

RAM and ROM statements

Since the declaration of Random Accessed Memories and Read Only Memories are similar (they only differ in allowable usage) they will be described together.

The RAM statement consists of the word RAM followed by a list of RAM declarations separated by commas. Each RAM declaration consists of the name of the word (data) register (optionally followed by the number of bits in the word (data) register if more than one) followed by an equal sign (=).
Next the name of the memory is followed by the address register indicating the number of bits in the address register, enclosed in parens. Since there are only two possible forms for the RAM declaration, they are illustrated in the following RAM statement:

\[
\text{RAM DATA}\langle 4\rangle = \text{MEMORY(ADDR}\langle 8\rangle ) , \text{MWR}=\text{M(MAR}\langle 12\rangle )
\]

The block diagrams for these memories are shown in figures 2 and 3. In the first declaration the word register has the name DATA and consists of 4 bits, the name of the address register is ADDR and consists of 8 bits, and the memory's name is MEMORY and consists of 256 four bit words. The addresses of the words run from 0 to 255. The second example of a RAM declaration consists of a memory whose name is "M", having a one bit word register named "MWR" and a 12 bit address register named "MAR". This memory has addresses running from 0 to 4095.

It is assumed that all declared memories will consist of more than one word and thus the number of bits in the address register must be indicated. The address of the word being referenced in the memory is taken from the address register with the low order bit of the number being the bit within the address register having the highest subscript.

The ROM statement has the same form as the RAM declaration statement except the word RAM is replaced by the word ROM.

CAM statements

The CAM statement declares the Content Addressable Memories (CAM's) local within each processor. These CAM's are different from the RAM's and ROM's in that they are capable of being searched as well as written into
Figure 2. Block diagram of a multiple bit word RAM and its description

\[
\text{DATA}^{<4>} = \text{MEMORY(ADDR}^{<8>})
\]
Figure 3. Block diagram of a single bit word RAM and its description

\[ \text{MWR} = M(\text{MAR}^{<12>}) \]
and read from. The form of the CAM statement is the word "CAM" followed by a list of CAM declarations separated by commas.

To understand the CAM declaration it is best to look at what registers are associated with each memory. Besides the address register and the word register which are used for the read and write operations in the same manner as in the RAM's and ROM's, there are two other registers. They are the mask register and the match register. They are only used during the search operation. The mask register is used to select which bits in the word register are to be compared to the corresponding bits in each word of memory. The match register is used to indicate if a word in memory matches the word register. The mask register has the same number of bits as the number of words in the memory.

The two possible forms of the CAM declarations are shown in figures 4 and 5. The number of words in the memory and the number of bits in the match register are equal to $2^n$ where $n$ is the size of the address register.

Adder statements

The description of the number of adders and to what registers they are connected is described in the ADDER statement. This statement has the form of the word "ADDER" followed by a list of adders local to the processor being described separated by commas. Each adder description begins with the name of the adder then a description of the registers to be added enclosed in parens. The name of the adder will be used later as a reference to when the given addition is to take place. The description of the registers to be added can be described by means of examples.
Figure 4. Block diagram of a multiple bit word CAM and its description
Figure 5. Block diagram of a single bit word CAM and its description

\[ \text{MAT}(M(W) = T(A^{<2>})) \]
REG $C^{7}$, $A^{6}$, $B^{5}$

ADDER $SUM(C = A \cdot ADD \cdot B)$

In this example the adder whose name is "SUM" results in the addition of the registers $A$ and $B$ with the resulting sum in register $C$. It is necessary for the receiving register, in this case, $C$, to be one bit larger than the largest of either register $A$ or $B$.

REG $E^{3}$, $D^{2}$

ADDER $ADD(E = D \cdot ADD \cdot B'01')$

This example illustrates that a constant (B'01') is to be added to register $D$ and the result placed in register $E$. The constant can only be the right most operand in the adder declaration.

**Auxiliary functions**

Auxiliary functions while not implying actual physical facilities, such as, registers, memories, or adders provides a method for the designer to indicate Boolean equations that are continuously defined. They also enable the designer to symbolically refer to only parts of any physical register.

**Terminal statements**

Whereas register transfer actions are performed only with the application of a control signal, there is a need for declaring Boolean equations that are to be evaluated continuously. These Boolean equations are handled in this language by means of the terminal statement.

The terminal statement consists of the letters "TERM" followed by a list of Boolean equations separated by commas. The form and meaning applied
to Boolean equations will be explained later under Register Transfer Actions.

Two examples of the use of the terminal statement are shown in figures 6 and 7. The first example, figure 6, illustrates the fact that bus C is the "wire-or" of registers X and Y which are located in two different processors.

The next example, figure 7, illustrates that the bus is the output of register B located in processor A.

**Define statement**

Provision is made for X in the language to symbolically refer to only part of a physical register, i.e. one that has been declared in a REG statement. An example of this symbolic defining of part of a register is shown below:

\[\begin{align*}
\text{a)} & \quad \text{REG} \quad \text{INST} <12 > \\
\text{b)} & \quad \text{DEF} \quad \text{OP} <4 > := \text{INST}<1:4>, \\
& \quad \text{ADDR} <8> := \text{INST} <5:12>
\end{align*}\]

In statement a, the physical register "INST" is declared to consist of 12 flip flops. In statement b, the symbolic register OP consisting of 4 bits is defined to be the leftmost 4 bits (i.e. bits 1:4) of the register INST and the symbolic register ADDR consisting of 8 bits is defined to be bits 5 through 12 of register INST. These symbolic registers can now be used in boolean expressions in the same way as actual physical registers.

Another example

\[\begin{align*}
\text{REG} & \quad \text{LCR}<7> \\
\text{DEF} & \quad \text{LC}<6> := \text{LCR}<2:7> \\
\text{ADDER} & \quad \text{INCR} (\text{LCR} = \text{LC}.\text{ADD}.\text{B}'000001')
\end{align*}\]
A: PROCESSOR(C);
  REG X
  TERM C := C + X

B: PROCESSOR(C)
  REG Y
  TERM C := C + Y

Figure 6. Terminal statement used as "wired or"

A: PROCESSOR(X)
  REG B
  TERM X := B

Figure 7. Terminal statement describing bus as output from register
illustrates how the ADDER statement and the DEFINE statement are combined to describe an incrementing register or counter. In this case the sub-register LC is incremented by one each time the adder INCR is invoked. If overflow occurs, bit LCR <1> is a one, otherwise the subregister LC is modulo \(2^6\).

States

The method of describing the flow of control within each processor used in this language is by means of a state diagram with only one state active at a time. Each state contains a description of actions to be performed either serially, simultaneously, or a combination of serial/simultaneous actions followed by the determination of the next state based upon a priority structure.

The description of each state can be broken down into three parts: 1) its name, 2) its action-sequence and 3) the determination of the next state. The state-name is separated from the action-sequence by a colon (:) and the action-sequence is separated from the next-state-determination by the character "THEN." as shown below.

```
state-name : action-sequence .THEN. next-state-determination
```

State-name

The state-name can consist of from 1 to 8 alphanumeric characters, the first of which must be alphabetic. Each state-name within a given processor must be unique.
**Action-sequence**

As stated, the flow of control within each state is either done serially or in parallel (simultaneously). If the actions or group of actions are separated by commas the actions are all started at the same time and continue in parallel. If $a_i$ represents the $i$th action then writing "$a_1, a_2, a_3$" implies that all three action $a_1$, $a_2$, and $a_3$ are done in parallel as illustrated below:

\[ a_1, a_2, a_3 \]

\[ \overrightarrow{a_1} \quad \overrightarrow{a_2} \quad \overrightarrow{a_3} \]

If the actions or groups of actions are separated by semicolons the action are done in sequence. In other words, writing "$a_4; a_5; a_6$" implies action $a_4$ is done, followed by action $a_5$, after which action $a_6$ is done as illustrated below:

\[ a_4; a_5; a_6 \]

\[ \overrightarrow{a_4} \quad \overrightarrow{a_5} \quad \overrightarrow{a_6} \]

The following example illustrates the combining of the previous two examples with a comma and shows action $a_3$ occurring in parallel with the serial action $a_4$, $a_5$, and $a_6$.

\[ a_1, a_2, a_3, a_4; a_5; a_6 \]

\[ \overrightarrow{a_1} \quad \overrightarrow{a_2} \quad \overrightarrow{a_3} \quad \overrightarrow{a_4} \quad \overrightarrow{a_5} \quad \overrightarrow{a_6} \]

Whereas when these same two examples are combined by a semicolon the actions $a_4$, $a_5$ and $a_6$ start after completion of $a_3$, as shown below:
Actions can also be grouped by use of parens. This allows the user to specify the order of execution within each state according to the dependencies of the actions with respect to each other without considering explicitly the timing problem. Figure 8 shows some possible ways of grouping actions by use of parens and the order of how the actions would be performed based upon the groupings.

**Next state**

The next-state is determined after the completion of all the actions within the state. This determination can be unconditional or based upon the priority between existing conditions at that time i.e. Boolean equations. This behavior will be illustrated by the following examples where \( c_1 \) and \( c_2 \) are Boolean expressions and \( s_0, s_1 \) and \( s_2 \) are names of states:

a) \( s_0: \) action-sequence .THEN. \((c_1) s_1 > (c_2) s_2 \)

b) \( s_0: \) action-sequence .THEN. .GOTO. \( s_1 \)

In the first example the condition \( c_1 \) has a higher priority than the condition \( c_2 \). If both conditions are true, i.e. equal to B'1', then the next state would be \( s_1 \). If only one condition was true then the next state would be based upon which condition was true. If neither of the conditions were true then the processor would wait until at least one of the conditions became true. At that time it would make the determination of the next state.
Figure 8. Examples of grouping actions and how execution would be performed.
It should be noted that if the conditions are mutually exclusive, this determination would behave the same as a decoder.

The second example b, simply illustrates an unconditional transfer to the state $s_1$ without consideration of the conditions within the processor.

**Actions**

There are three basic actions allowed in this language. They are register transfer, memory and adder actions. The time required for each action to be performed is specified by writing an integer after a dollar sign ($) following the action. For example the following list of actions in the language

$$A = B \; \$ \; 10 \; ; \; \text{READ(CORE)} \; \$ \; 100 \; ; \; \text{ADD(SEQ)} \; \$ \; 50$$

indicates that the register transfer $A=B$ will take 10 units of time, the memory action READ(CORE) will take 100 units of time and the adder action ADD(SEQ) will take 50 units of time. If the time is not specified after the action, the action is assumed to take one (1) unit of time. Fractional units of time are not allowed.

**Register transfer**

The register transfer action consists of applying logical, relational and concatenation operations between registers and transferring this result to another, or possibly the same, register.

The logical operations allowed are complementation, inclusive or and the logical "and" operation. The complementation operation is an unary operation that is specified by the characters .NOT., followed by the argument enclosed in parens. The result of this operation is a bit by bit
complementation of the argument with all of the 1's originally in the argument changed to 0's and all the 0's originally in the argument changed to 1's. The argument can be either a simple register or an expression.

Both the inclusive or (+) and the "and" (*) operations are binary and when used both of the operands must be the same length. The results of these operations are the bit by bit inclusive "or" or the "and" of the operands.

The relational operations are binary and result in only a single value depending upon the comparison between the operands. The comparison considers both of the operands as unsigned integers. If the comparison is true the resulting value of the operation is B'1', otherwise it is B'0'. Both operands must have the same number of bits. The permitted relationals that can be used in this language are "greater than" (.GT.), "greater than or equal" (.GE.), "equal" (.EQ.), "not equal" (.NE.), "less than or equal" (.LE.), and "less than" (.LT.).

The catcatenation operation (|) allows the designer to symbolically combine registers temporarily into a single entity during the evaluation of the Boolean expression.

For example, the following register transfer action indicates that the contents of register B is shifted one place to the left and that the rightmost bit is then set equal to a zero.

\[
\begin{align*}
\text{REG} & \quad B<10> \\
B<1:10> & = B<2:10> | B'0'
\end{align*}
\]
The basic form of the register transfer action is a destination register being assigned the value of a Boolean expression as in the following:

\[
\text{destination-register} = \text{Boolean-expression}
\]

The Boolean-expression, consisting of register operations, is evaluated and the result of this evaluation is transferred to the destination-register. The number of bits in the final result of the Boolean-expression must be the same as the number of bits in the destination-register, otherwise the register transfer action is invalid and erroneous results will occur.

The order of the evaluation of the Boolean-expression is based upon the priority of the register operations. These register operation priorities are as follows:

- Highest: Concatination, Complementation (.NOT.( )
- Logical "and": *
- Logical "or": +
- Lowest: Relational:.LT. .LE. .EQ. .NE. .GE. .GT.

The operation with the highest priority is performed first followed by the other operations in the order of their priority. If two operations of the same priority can be performed, the leftmost one is performed first then the other. This order of evaluation can be modified by the use of parens, with the parens overriding the priorities between the operations as is done in present day high level programming languages.
Memory actions

A memory action involves reading, writing, or searching of a declared memory. The syntax for invoking these memory actions is as follows:

```
READ(memory-name)
WRITE(memory-name)
SEARCH(memory-name)
```

The read memory action uses the contents of the declared memory address register as an address to a particular word in memory and places the contents of that word into the memory word register. The read memory action is valid for all three types of memories.

The write memory action is opposite of the read memory action in that it puts the contents of the word register into the word specified by the address register. The write memory action is only valid for Random Accessed Memories and Content Addressable Memories.

The search memory action is used only in conjunction with Content Addressable Memories. It searches each location in memory to see if each bit position within the memory is the same as the corresponding bit position in the word register. The mask register is used to indicate which bit positions within the word register are to be ignored and not compared. A one in the mask register indicates that this bit is to be compared and a zero indicates that this bit position within the word register is to be ignored. For every location in the memory that matches the "masked" word register a one is placed in its corresponding position in the match register. If the word in memory does not match the "masked" word register then a zero is placed in the match register.
Adder actions

The adder action consists of adding the two declared registers and placing the result of that addition in a third register. The syntax to invoke this action is as follows:

ADD(adder-name)
CHAPTER V. SIMULATION

In order to exercise or to simulate the multiprocessor system described by the language there must be some way of indicating the initial conditions of the system, what items are to be monitored and when the simulation is completed. The statements SET, ACTIVATE and TRACE are used to perform these functions. They are explained informally in this section.

The simulation to see if the system is logically correct is divorced from the actual construction of the system to allow for possible use as input to an automated construction system. Thus, the instructions to perform this simulation are separated from the system description and are enclosed by the words "SIMULATE." and "END."

The initialization of the system is performed by two statements SET and ACTIVATE. The SET statement is used to initialize the various physical registers and memories, whereas the ACTIVATE statement is used to specify each processor's beginning state.

The form of the SET statement is the word "SET" followed by a list of register or memory initializations separated by commas.

There are three forms of the register initialization as shown in the following examples using register ACM that has been declared as REG ACM<6>:

a) ACM = B'111101'
b) ACM<3> = B'0'
c) ACM<2:5> = B'0011'

In form a, the entire register ACM is initialized to the constant B'111101'. In form b, only the single bit ACM<3> is initialized. The third form c,
as shown in c, illustrates initialization of a range of bits within a given register, in this case, bits 2 through 5 are initialized to the constant B'0011'.

There is only one form of memory initialization and it initializes an entire word or memory at a time. Some examples of this memory initialization form using the declared memory

\[ \text{RAM WORD}^{<4>} = \text{CORE(ADDR}^{<8>}) \]

are as follows:

\[ \text{CORE(20)} = \text{B'1101'}, \text{CORE(254)} = \text{B'0011'} \]

In the first case the 20th word of the memory CORE is initialized to the value B'1101' and in the second case the 254th word is initialized to the value B'0011'.

**Activate**

The ACTIVATE statement is used to tell which is to be the beginning state for each of the processors within the system that is to be simulated. Only one beginning state should be specified for a given processor.

The form of the ACTIVATE statement is the word "ACTIVATE" followed by a list of states to be activated. Each state in the activation list has the following form

\[ \text{processor-name . state-name} \]

In this form the processor-name is separated from the state-name by a period. The processor-name specifies the name of the process to be
activated and the state-name specifies the name of the beginning state for that processor.

**Tracing**

Controlling the tracing of the simulation is done by the TRACE statement. This statement indicates which items are to be printed and when the simulation is complete.

The form of the TRACE statement is as follows:

```
TRACE item-list.UNTIL.(timing-condition)
```

The item-list consists of a list of items to be traces separated by commas. There are four possible items to be traced. Three of these are to trace the contents of registers or parts of registers and the other is to trace the contents of a word in memory.

The item to trace the contents of a word in memory has the following form:

```
processor-name . memory-name (word-number)
```

In this form, as in the others, the period and parens are used to separate the various parts of the item. The processor-name indicates the name of the processor where the memory is located, the memory-name indicates the name of the memory and the word-number is an integer indicating which word is to be traced.

The three forms to allow tracing of contents of the registers are illustrated as follows:
The first form a, illustrates tracing the contents of the entire register whereas the other two forms trace only parts of the register. In the second form b, only the bit referred to by the integer will be traced. The third form c, illustrates the tracing of adjacent bits within a given register with the two integers defining the group of bits to be traced.

When the simulation is complete it is determined by the timing-condition enclosed in the parens after the word "UNTIL". This timing-condition can have only the following forms:

a) \text{TIME} \geq \text{integer}

b) \text{TIME} > \text{integer}

c) \text{TIME} = \text{integer}

The meaning of these forms is self-evident.
CHAPTER VI. SINGLE PROCESSOR EXAMPLE

This is an example of how the description language can describe and simulate the Tel_comp computer. The Tel_comp computer is a simple hypothetical computer used as a classroom aid to illustrate the basic concepts of computer construction and principles.

It is a one address stored program computer with each instruction consisting of 9 bits or a word. The leftmost 3 bits of the instruction indicates the operation code, i.e. it has eight possible instructions. The rightmost 6 bits of the instruction indicates the direct address of the simple operand. The description of the Tel_comp computer is given in Appendix F and will be referred to by the line numbers on its left hand side.

Its working storage consists of 4 physical registers LCR, ACR, IR and ACL and a main memory M as shown in figure 9.

The register IR is a 3 bit register used as the instruction register and holds the current instruction being executed. The register ACL is a 9 bit register and is used as the accumulator. It also is one input to the adder ADDIT. The register LCR is a 7 bit register and is further divided into two subregisters, LCOV and LC. This division is necessary to allow for the incrementing of the subregister LC modulo $2^6$ by the adder INCR. This subregister LC is used as the location counter and contains the address of the next instruction to be executed.

The register ACR is a 10 bit register that is divided into two sub-registers, OV and AC2. Subregister OV is defined to be the leftmost bit in register ACR and could be used to indicate overflow of the addition of
Figure 9. Block diagram of Tel_comp's working storage
registers XR and AC1, but in this computer it is not used. The subregister AC2 is the rightmost 9 bits of register ACR and is used to temporarily hold the results of the binary addition of the registers XR (the memory word register) and AC1 (the accumulator).

The Tel_comp computer also has a main memory that is a Random Accessed Memory named M. It consists of 32 words with each word containing 9 bits. Its memory address register, MAR, consists of 6 bits and its memory word register, XR, consists of 9 bits. The memory word register, XR, is subdivided into two subregisters OP and ADDR. The subregister OP is defined to be the leftmost 3 bits of the register XR and the subregister ADDR is defined to be the remaining 6 bits or the rightmost 6 bits of the register XR.

There are two adders in the Tel_comp computer and they are declared in its description as ADDIT and INCR. The adder ADDIT is used to add the contents of register XR to the contents of register AC1 (the accumulator) and put the result in the temporary register ACR. The other adder, INCR, is used to increment the location counter, register LC, by one.

Illustrating the flow of control, or when the primitive actions (register transfers, memory and adder actions) for the Tel_comp computer are to take place is done by means of a diagram. The diagram for the Tel_comp computer is shown in figure 10. The circles in this diagram represent a sequence of serial or parallel primitive actions and the arrows indicate the order of when these actions are to take place. The unlabeled arrows in the figure indicate control going from one circle to another unconditionally. The labeled arrows indicate the conditions necessary for control to follow those arrows.
Figure 10. State Diagram of Tel_comp computer
This diagram is a one-to-one graphical representation of the written description of the Tel-comp computer. The detailed descriptions of the individual action groups are illustrated beginning with the word FETCH the Tel_comp description.

The group named FETCH illustrates two features of the language. The first is the specification of serial execution of the actions MAR=LC and READ(M) followed by the parallel execution of the actions IR=OP, MAR=ADDR and ADD(INCR). The second feature is the specification of an instruction decoder. The reason this next-state-determination acts as a decoder is that all of the next-state-conditions are mutually exclusive.

The primitive action-sequences necessary for the execution of each instruction are described in the other groups. They are ADD (ADDition), SUB (SUBtraction), SRO (Shift Right One), TRU (TRansfer Unconditionally), TRN and TRNL (TRansfer if accumulator is Negative), STA (STore Accumulator), CLA (CLeaR Accumulator), and STP (SToP).

The STP group illustrates the use of the next-state-determination to cause the Tel_comp computer to wait in the STP group until certain conditions are met. The depression of the start switch or SR=B'1' allows the computer to continue fetching and executing instructions. Until the start switch is depressed the computer will stay in the STP group.

Tel_comp computer simulator

Once the Tel_comp computer has been described, then actual programs written in its machine language can be simulated.

In the example presented here, the program to be simulated consists strictly of the single execution of each possible instruction of the
Tel_comp computer. This was to insure that each instruction in the computer functioned properly as expected.

The symbolic description or assembly language version program is illustrated by the comment cards after the word "SIMULATE." The program is fairly straightforward, in that if the computer has been described correctly, the order of execution of the instructions should be according to their addresses 0, 1, 2, 3, 4, 7, 5, 6.

The initialization of the start switch (SR), location counter (LC), accumulator (AC1), temporary result (AC2), memory address register (MAR), exchange register (XR), and the instruction register (IR) is done in the SET statement. Also this SET statement initializes the memory to contain the given program to be simulated.

The ACTIVATE statement indicates that the starting group of the processor is to be FETCH.

The first TRACE statement indicates that the contents of the location counter (LC), accumulator (AC1) and temporary register (AC2) of the processor named TEL are to be traced.

The processor is then to be initialized again and the same program executed only this time tracing the exchange register (XR), the instruction register (IR), and the memory address register (MAR).

The trace of the Tel_comp computer simulation is given in Appendix G. The contents of registers or memory location indicated in the TRACE statement are printed anytime any action takes place within the multiprocessor system being simulated.

The time is printed in the left hand column and if any actions take place, the indicated contents are printed to the right of the time. The
actions in this simulation are assumed to take place, then the time required for that action is considered.

The following figure illustrates the timing of the first few units of time for this simulation:

Thus at time = 0 the content of register MAR is changed, at time = 1 the READ(M) action changes register XR and at time = 52 the actions MAR=ADDR, IR=OP, and ADD(INCR) cause the registers LC, MAR and IR to be changed.

Every time the simulation enters a new state the name of the state being entered is printed.
CHAPTER VII. MULTIPLE PROCESSOR EXAMPLE

The previous section considered a single Tel_comp computer using a single memory. The example in this section will compare this single processor vs. two processors sharing a common memory.

Interconnections

The block diagram of the two Tel_comp computers and the common memory is shown in figure 11.

The line REQ is used by each processor to request a memory action. Which action the processor wants done is indicated by the RW line. When line RW equals B'0', the processor wants a read action and when line RW equals B'1', a write action. The bus ADDR is used to pass the address of the location to be read or written. The bus WORD is used to pass the data between the processor and memory.

The line ACK is used by the memory to acknowledge the request from the processor and to indicate the action has been done.

Sharing conventions

In order for the two processors to share the common memory certain conventions must be followed. These conventions and the actions taken by the processor and memory are graphically illustrated in figure 12.

When a processor wants a memory action the processor sets its REQ line equal to B'1'. If the memory is free, the action is performed and the processor is notified of completion by the memory control setting the processor's ACK line equal to B'1'. The processor then in turn can notify
Figure 11. Block diagram of Multiple Tel_comp computer
Figure 12. Signals between Tel_comp and Memory Control
the memory control that it has used the data made available by setting its
REQ line equal to B'0'. Now the memory control can set the appropriate
ACK line equal to B'0' and wait for another request.

State diagrams

The state diagram for each processor is shown in figure 13. It should
be noted that the original states FETCH, ADD, SUB and STA had to be broken
into two states. This is because of the memory actions, Read and Write,
required in these original states. Now with shared memory, each processor
must make a request for memory and then wait until that request is acknowl-
edged.

The state diagram for the MEMORY is shown in figure 14. The MEMORY
starts in the WAIT State by waiting for a request from either processor.
When a request arrives it honors that request and performs the necessary
actions. If both processors make a request at the same time, the priority
between the requests determines which processor is served first.

Simulation

The multiprocessor description is given in Appendix H and its simula-
tion is given in Appendix I. The multiprocessor simulation has each proc-
essor executing the same program as described in the simple Tel_comp
computer example. In this case processor TELA executes the program starting
in location zero and TELB starts in location 12.
Figure 13. State Diagram for processor TELA or TELB
Figure 14. State Diagram for processor MEMORY
Whereas the single Tel_comp computer took 750 units of time to complete the program, the two Tel-comp computers sharing a common memory takes 1349 units of time. This illustrates the contention between the two computers for the memory. If there had been no contention for the memory, the time should be 750 units or the same as the single processor.

These examples also show that by adding hardware needed for the two processors that instead of taking 1500 units of time to execute the two programs it only takes 1349 units of time. This means there is a savings of about 150 units of time or about a 10 percent increase in speed over a single processor. The designer can now make a more informed decision either to have a single processor, dual processors or consider a different way of implementing the system design.

Comparisons between the simulation of these two examples are shown in table 1.

Table 1. Comparisons between the simulation of the two examples

<table>
<thead>
<tr>
<th></th>
<th>One Processor</th>
<th>Two Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/1 statements generated</td>
<td>404</td>
<td>1333</td>
</tr>
<tr>
<td>Simulated time to perform the program twice 2*750 =</td>
<td>1500</td>
<td>1349</td>
</tr>
<tr>
<td>Computer time (seconds)</td>
<td>55</td>
<td>125</td>
</tr>
<tr>
<td>Core required to execute the generated PL/1 program</td>
<td>76K</td>
<td>126K</td>
</tr>
</tbody>
</table>
The overall picture of this programming system is that the translator converts the multiprocessor system description and simulation source language into PL/1 statements. These PL/1 statements when compiled and executed behave as the multiprocessor system being described. The intent of this section is to give some insight into the implementation of this language simulator. Figures 15, 16 and 17 gives a pictorial view of the programs and files necessary to implement the language.

As just stated, the translator converts the source language statements, i.e. the description of the multiprocessor system, into PL/1 statements as shown in figure 15. One question that can be raised is "Why use PL/1 as the target language of the translator?". This question can be answered very simply. Just as FORTRAN is more machine independent than machine language so is PL/1. This Machine Independence allows this system to be more mobile than machine language. Unfortunately, standard FORTRAN does not provide for character bit and label data types. These data types are available in PL/1 and are needed in writing both the translator and the simulation run-time routines.

The translator is a top-down translator written in PL/1 that doesn't require backup. Normally a top-down translator requires back-up i.e. has to look at the source language more than once. This is not the case in this translator, because the syntax or grammar of the source language was chosen to eliminate the necessity for back up by looking ahead one word.

The PL/1 statements generated by the translator are divided into six groups based upon where they are to be placed in the PL/1 program $RUN.
Figure 15. Translator Step in the Simulation
Figure 16. Macro library Generation for PL/1 compiler
Figure 17. Execution step in the Simulation
When the program $\text{RUN}$ is compiled and executed it behaves as the multi-
processor being described. These six groups are called INITS, DECLS, TERMS, 
CONDS, STATS, and SIMUL.

The group INITS contains a procedure for each processor in the system. 
The only statements within each of these procedures are the necessary 
Boolean assignment statements for the Boolean variables used to control the 
parallel execution of action groups within the processor.

The group DECLS contains all of the declaration statements of the PL/1 
variables needed by the simulator program. It also contains PL/1 procedures 
used by the memory actions READ, WRITE and SEARCH as well as the PL/1 
procedure generated for each declared adder.

The group TERMS contains Boolean assignment statements for the declared 
terminals in the system. These assignment statements are executed every 
time unit, thus behaving as if they were continuously being evaluated.

The group CONDS contains Boolean assignment statements for all of the 
conditions or Boolean expressions found in the next state determination. 
They are evaluated every time unit along with the terminal assignment 
statements. The Boolean equation found in the next state determination is 
assigned to a dummy Boolean variable generated by the translator. This variable is then used in actual next state determination.

The group STATS contains the PL/1 statements that when executed per-
form the actions, calls to the scheduler routine $\text{CH}$, and the conditional 
statements to perform the next state determination.

The group SIMUL contains the PL/1 statements generated for controlling 
the simulation of the system. The SET statements simply generate Boolean 
assignment statements. The ACTIVATE statements generate calls to the
scheduler to schedule the indicated states in the first time slot. The Trace statement generates an ON condition block that is executed each time any action within the system takes place.

Macro library generation

The program LEBGENER as shown in figure 16 is a utility program that is required by the PL/1 compiler. PL/1 requires that to use the PL/1 macro statement ^INCLUDE file (name). This file, in this case MPSLIB, must be a partitioned data set. This requirement also forces that the characters "/*END*/" be at the end of each file.

Execution step

In figure 17 the macro compilation of the PL/1 source program $RUN places the groups of PL/1 proper statements generated by the translator into their position. This PL/1 program is then compiled and its resulting object deck combined with the previously compiled scheduling routines to form the multiprocessor simulator. The execution of this simulator program generates a printed trace of how the multiprocessor system would behave based upon its initial conditions.

Run-time routines

The subroutine $ETUP determines the size of the run-time arrays and allocates storage for those arrays.

The subroutine $TART initializes the run-time arrays used in simulation of the multiprocessor system. It is called each time the TRACE statement is executed.
The subroutine $INCR is called to increment the current time after all actions scheduled have been performed.

The subroutine $CH is called to schedule an action to be performed. It has two arguments, a label and an integer. The label is the label on the action to be scheduled and the integer is the time the action is to be performed relative to the current time. This value must be greater than or equal to zero. The use of the value zero is used to schedule parallel actions.
CHAPTER IX. CONCLUSION

The language presented in this paper allows not only for the description of stored program computer as done in Bell and Newell (2,3) and Pumplin (25), but also for describing one or more processors interconnected by buses. This allows the system designer a greater freedom and variety in the systems he can describe. Separating the system into interconnected processors allows the designer to view the partitioning of the system. Although Duley and Dietmeyer (8) consider autonomous units processors they do not explicitly force the designer to specify the number of interconnections between these processors. The language presented here goes beyond them and forces the designer into considering the problem of interconnections.

Each processor is in turn described as a group of states. Each of these states within a processor consists of a series of serial, parallel and a combination of serial and parallel events. Whereas designers have used this notation to describe the flow of control within a processor and are familiar with it, it has never been used in a language to design computers.

The language's provision for describing the serial, parallel and a combination of serial and parallel events allows the designer a greater degree of freedom than previously offered. Most other languages (6,8,21,25) allow for only parallel execution of actions and one allows for a limited form of parallel execution of serial events (2,3). This feature, in and of itself, allows the designer to describe when the multitude of events are to occur within the system more accurately.
The simulation of the descriptive language allows the designer the ability to test not only the logical consistency of his proposed system design, but also it allows the designer to compare his design vs. other designs. Without being able to simulate the descriptive language, the descriptive language becomes almost worthless except as input to an automated construction system. By being able to simulate the descriptive language the proposed system can be checked.

One addition that would enhance the language is the inclusion of some way to be able to define more and different types of facilities for each processor. With this feature, as more and more of the basic computer functions become available in hardware packages, the designer would have the ability to include these new definitions.

The inclusion of a input statement to assist in controlling the simulation would also aid in making the language more flexible and less expensive to use. With an input statement the generated PL/1 program could be compiled only once and then different test cases run using this object deck. At present, each time the language is used, the generated PL/1 program has to be compiled and this compilation time far outweighs the actual execution of the generated program.

Other languages additions and improvements could be as follows:

a) incorporation of a first-in-first-out (FIFO) queuing into the next state determination.

b) provision to define multiple processors of the same type as would be required in describing the ILLIAC IV.

c) incorporation of a way to determine the time necessary to perform given actions, such as register transfer.
In conclusion, the language presented thus provides a simple yet powerful language to describe and simulate multiprocessor computer systems. Its use will provide the systems designer with an aid in designing more complicated digital systems.

Using this language the systems designer can describe and simulate a proposed system. If the proposed system does not meet his design criteria he can go back and make changes to the system and simulate it again until he is satisfied with the proposed system characteristics. He can then use the description of the system as input to an automated construction system and build the system in hardware.
LITERATURE CITED


APPENDIX A:

SYNTAX OF LANGUAGE
PROGRAM ::= SYSTEM . SIMULATE

SYSTEM ::= \'.SYSTEM.' SYSTEM_DCL . PROC $ PROC
SYSTEM_DCL ::= 'BUS' BUSDCL $( ', ' BUSDCL )
BUSDCL ::= ID . OPT( '<' INT '>' )

PROC ::= PROC_DCL $( FAC_DCL ';' ) STATE $( ';' . OPT( STATE ) ) '.END.' . OPT( ';' )
PROC_DCL ::= ID ':=' 'PROCESSOR' '(' BUSLIST ')' ';
BUSLIST ::= BUSREF $( ', ' BUSREF )
BUSREF ::= ID . OPT( '<' INT . OPT( ':' INT ) '>' )

FAC_DCL ::= REGSTMT | RAMSTMT | ROMSTMT | CAMSTMT | ADDSTMT | TERMSTMT | DEFSTMT
REGSTMT ::= 'REG' REGDCL $( ', ' REGDCL )
REGDCL ::= ID . OPT( '<' INT '>' )
RAMSTMT ::= 'RAM' RAMDCL $( ', ' RAMDCL )
RAMDCL ::= ID . OPT( '<' INT '>' ) ' = ' ID '(' ID '<' INT '>' ')'
ROMSTMT ::= 'ROM' ROMDCL $( ', ' ROMDCL )
ROMDCL ::= ID . OPT( '<' INT '>' ) ' = ' ID '(' ID '<' INT '>' ')'
CAMSTMT ::= 'CAM' CAMDCL $( ', ' CAMDCL )
CAMDCL ::= ID '(' ID '(' ID '<' INT '>' ') ' = ' ID '(' ID '<' INT '>' ')'
ADDSTMT ::= 'ADDER' ADDDCL $( ', ' ADDDCL )
ADDDCL ::= ID '(' ID '=' ID '.ADD.' CONSTANT )
TERMSTMT ::= 'TERM' TRANSFER $( ', ' TRANSFER )
DEFSTMT ::= 'DEF' DEFDCL $( ', ' DEFDCL )
DEFDCL ::= ID ' = ' ID . OPT( '<' INT '>' )

STATE ::= ID ':=' . OPT( ACTIONLIST ) '.THEN.' ( CONDLIST | GOTO )
ACTIONLIST ::= ACTION $( ', ' ACTION )
ACTION ::= '(' ACTIONLIST ')' . ACT
ACT ::= ( MEMORY | ARITH | TRANSFER ) . OPT( '$' INT )
CONDLIST ::= COND $( '>' COND )
COND ::= '(' EXPR ')' ID
GOTO ::= '.GOTO.' ID
MEMORY ::= 'READ' '(' ID ')' | 'WRITE' '(' ID ')' | 'SEARCH' '(' ID ')' | 'ADD' '(' ID ')' | 'TRANSFER' = 'EXPR'

EXPR ::= TERM . OPT( RELOP . TERM )
RELOP ::= '.LT.' | '.LE.' | '.EQ.' | '.NE.' | '.GE.' | '.GT.'
TERM ::= FACTOR $( '+' FACTOR )
FACTOR ::= SEC $( '*' SEC )
SEC ::= '.NOT.' | '(' GROUP ')' | GROUP
GROUP ::= PRIM $( '(' PRIM )
PRIM ::= CONSTANT | REGREF | '(' TERM ')' | 'B' ' ' BIT $ BIT ''
REGREF ::= ID . OPT( '<' INT . OPT( ':' INT ) '>' )
These are the basic syntax rules

BIT ::= '0'|'1'
DIGIT ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
INT ::= DIGIT $ DIGIT
ALPHA """"= 'A'|'B'|'C'| ... |'X'|'Y'|'Z'
ID ::= ALPHA $(ALPHA|DIGIT)
APPENDIX B:

TRANSLATOR ROUTINES
mps: procedure options(main) recursive;

/* file numbers */
1 states */
2 terminals */
3 conditions */
4 control */
5 declares */
6 simulate */

DCL card char(80) external, /* input card */
c(80) char(1) def card, /* input card */
print file print, /* output */
line char(80) external, /* output */
l fixed bin external, /* index on line */
i fixed bin external, /* index on card */
stmt fixed bin external, /* stmt number */
file fixed bin external, /* file */
true bit(1) static init('1'8), /* indicates item found */
false bit(1) static init('0'8), /* indicates item not found */
program entry returns( bit(1) ), /* program entry */
system entry returns( bit(1) ), /* system entry */
simulate entry returns( bit(1) ), /* simulate entry */
dummy fixed bin;

on condition( eoc ) begin; la:read file(sysin) into(card);
call nl; /* debug */
if c(1)='*' then do; put file(sysprint) edit(card)(col(7),a);
put file(print) edit(card)(skip,col(7),a);
go to la;end; /* skip comment cards */
put file(sysprint) edit(stmt,' ',card)
(skip,f(5),a,a);
PUT FILE(PRINT)EDIT(STMT,' \*CARD)(SKIP,F(5),A,A);
      I=1; END;

ON ENDFILE(SYSIN) GO TO DONE;

/* PUT OUT HEADING */

PUT FILE(SYSPRINT) EDIT(' M P S')(PAGE,A)
('*** SOURCE CODE ***')(SKIP(3),COL(7),A);
PUT FILE(PRINT) EDIT(' M P S')(PAGE,A)
('*** SOURCE CODE ***')(SKIP(3),COL(7),A);

/* INITIALIZE */

FILE =1;
STMT=1; L=8; LINE=(80)'
SIGNAL CONDITION(EOC);

IF PROGRAM THEN PUT FILE(SYSPRINT)EDIT('SUCCES')(SKIP,A);
ELSE PUT FILE(SYSPRINT)EDIT('UNSUCESSFUL')(SKIP,A);

CALL NL; FILE=1; CALL OUT(' */END*/');
CALL NL; FILE=2; CALL OUT(' */END*/');
CALL NL; FILE=3; CALL OUT(' */END*/');
CALL NL; FILE=4; CALL OUT(' */END*/');
CALL NL; FILE=5; CALL OUT(' */END*/');
CALL NL; FILE=6; CALL OUT(' */END*/');
CALL NL;
GO TO ERREXIT;

ERR: ENTRY( X ); DCL X CHAR(*);
PUT FILE(SYSPRINT) EDIT('$ \,*x)(SKIP,COL(I+6),A,A);
PUT FILE(PRINT) EDIT('$ \,*x)(SKIP,COL(I+6),A,A);
STOP; /* DEBUG */

/* PROGRAM ::= SYSTEM .SIMULATE; */

PROGRAM:PROCEDURE RETURNS( BIT(1) );
   IF ~SYSTEM THEN RETURN( FALSE );
   IF ~SIMULATE THEN CALL ERR(' SIMULATE NOT FOUND');
RETURN( TRUE ); END PROGRAM;

DONE: /* EXIT WHEN ALL CARDS HAVE BEEN READ */
   PUT FILE(SYSPRINT)EDIT('NO MORE CARDS')(SKIP(3),A);

ERREXIT: /* EXIT WHEN SYNTAX ERROR HAS BEEN FOUND */
END MPS;
(SUBRG, STRG):
SYSTEM: PROCEDURE RETURNS( BIT(1) );

/*
SYSTEM ::= '.SYSTEM.' SYSTEM_DCL .PROC $ Proc
*/

DCL LIT ENTRY( CHAR(*) ) RETURNS( BIT(1) ),
TRUE BIT(1) STATIC INIT('1'B),
FALSE BIT(1) STATIC INIT('0'B),
ERR ENTRY( CHAR(*) ),
SYSTEM ENTRY RETURNS( BIT(1) ),
SYSTEM_DCL ENTRY RETURNS( BIT(1) ),
BUSDCL ENTRY RETURNS( BIT(1) ),
SYSTIME ENTRY RETURNS( BIT(1) ),
SYSOP ENTRY RETURNS( BIT(1) ),
PROC ENTRY RETURNS( BIT(1) ),
OUT ENTRY( CHAR(*) VARYING ),
ID ENTRY RETURNS( BIT(1) ),
STAR CHAR(32) VARYING EXTERNAL,
FILE FIXED BIN EXTERNAL,
STMT FIXED BIN EXTERNAL,
INT ENTRY RETURNS( BIT(1) );

FILE =5;
IF ~LIT('SYSTEM.') THEN RETURN( FALSE );
IF ~SYSTEM_DCL THEN CALL ERR('SYSTEM_DCL NOT FOUND');
CALL OUT('**');
STMT=STMT+1;
L5: IF ~LIT('**') THEN CALL ERR('MISSING **');
IF ~SYSTEM_DCL THEN GO TO L6;
CALL OUT('**');
STMT=STMT+1;
GO TO L5;
L6: IF ~PROC THEN CALL ERR('PROCESSOR NOT FOUND');
L1: IF ~PROC THEN RETURN( TRUE );
GO TO L1;

/*
SYSTEM_DCL ::= 'BUS' BUSDCL $(',' BUSDCL ) |
/*
'*TIME' SYSTIME $(',' SYSTIME ) ; */
SYSTEM_DCL:PROCEDURE RETURNS( BIT(1) );
  IF ~LIT("BUS") THEN GO TO L4;
  CALL OUT("DCL");
  IF ~BUSDCL THEN CALL ERR("BUSDCL NOT FOUND");
L2: IF ~LIT(",") THEN RETURN( TRUE );
  CALL OUT(","); CALL NL;
  IF ~BUSDCL THEN CALL ERR("BUSDCL2 NOT FOUND");
  GO TO L2;
L4: IF ~LIT("TIME") THEN RETURN( FALSE );
  IF ~SYSTIME THEN CALL ERR("MISSING SYSTEM TIME");
L3: IF ~LIT("") THEN RETURN( TRUE );
  IF ~SYSTIME THEN CALL ERR("MISSING SYSTEM TIME");
  GO TO L3; END SYSTEM_DCL;

/*         BUSDCL ::= ID *OPT( '<' INT '>' ) ; */
BUSDCL:PROCEDURE RETURNS( BIT(1) );
  IF ~ID THEN RETURN( FALSE );
  CALL OUT(STAR);
  IF ~LIT("<") THEN DO; CALL OUT(" BIT(1)"); RETURN( TRUE ); END;
  IF ~INT THEN CALL ERR("MISSING UPPER SCRIPT ON DCL");
  CALL OUT(" BIT("|STAR("|"))");
  IF ~LIT(">") THEN CALL ERR("MISSING RIGHT BRACKET >");
  RETURN( TRUE ); END BUSDCL;

/*         SYSTIME ::= SYSOP '$' INT ; */
SYSTIME:PROCEDURE RETURNS( BIT(1) );
  IF ~SYSOP THEN RETURN( FALSE );
  IF ~LIT("$") THEN CALL ERR("MISSING ");
  IF ~INT THEN CALL ERR("MISSING INTEGER");
  RETURN( TRUE ); END SYSTIME;

/*         SYSOP ::= 'READ' | 'WRITE' | 'SEARCH' | '->' | '+' | '*' | */
/*         | 'ADD' | 'SUB' ; */
SYSOP:PROCEDURE RETURNS( BIT(1) );
  IF LIT("READ") THEN RETURN( TRUE );
  IF LIT("WRITE") THEN RETURN( TRUE );
  IF LIT("SEARCH") THEN RETURN( TRUE );
IF LIT('−') THEN RETURN( TRUE );
IF LIT('+) THEN RETURN( TRUE );
IF LIT('∗') THEN RETURN( TRUE );
IF LIT('ADD.') THEN RETURN( TRUE );
IF LIT('SUB.') THEN RETURN( TRUE );
RETURN( FALSE );   END SYSOP;
END SYSTEM;
(SUBRG,STRG):
PROC:PROCEDURE RETURNS( BIT(1) );
    PROC ::= PROCDCCL $( FACDCL ';' )
    /* PROC ::= PROCDCL $( FACDCL ';' ) */
    STATE $( ';'; STATE ) '.END.' ';' ; /* STATE $( ';'; STATE ) '.END.' ';' ; */

DCL LIT ENTRY( CHAR(*) ) RETURNS( BIT(1) ),
    TRUE BIT(1) STATIC INIT('1'B),
    FALSE BIT(1) STATIC INIT('0'B),
    FILE FIXED BIN EXTERNAL,
    OUT ENTRY( CHAR(*) VARYING ),
    ERR ENTRY( CHAR(*) ),
    PROC ENTRY RETURNS( BIT(1) ),
    PROCDCCL ENTRY RETURNS( BIT(1) ),
    BUSLIST ENTRY RETURNS( BIT(1) ),
    BUSREF ENTRY RETURNS( BIT(1) ),
    STATE ENTRY RETURNS( BIT(1) ),
    FACDCL ENTRY RETURNS( BIT(1) ),
    STMT FIXED BIN EXTERNAL,
    ID ENTRY RETURNS( BIT(1) ),
    PNAME CHAR(8) VARYING EXTERNAL,
    STAR CHAR(32) VARYING EXTERNAL,
    NL ENTRY,
    INT ENTRY RETURNS( BIT(1) );

    IF ¬ PROCDCCL THEN RETURN( FALSE );
L1: IF ¬ FACDCL THEN GO TO L2;
    IF ¬ LIT(';' ') THEN CALL ERR('MISSING ;');
    CALL OUT( '"' );
    GO TO L1;
L2: IF ¬ STATE THEN CALL ERR('NOSTATE IN PROCESSOR');
L3: IF ¬ LIT(';' ') THEN GO TO L4;
    IF ¬ STATE THEN GO TO L4;
    GO TO L3;
L4: IF ¬ LIT('.END.' ') THEN CALL ERR('MISSING END STMT');
    CALL NL; FILE=4; CALL OUT( '"END' ); CALL NL; FILE=5;
    IF ¬ LIT(';' ') THEN
    RETURN( TRUE );
PROCDECL ::= ID '::' 'PROCESSOR' '(' 'BUSLIST' ')' ';' ;
PROCDECL:PROCEDURE RETURNS( BIT(1) );
  IF ~ID THEN RETURN( FALSE );
  CALL NL; FILE=4; CALL OUT(''INIT_''|STAR|''::PROCEDURE'');
  PNAME=STAR; CALL NL; FILE=5; CALL OUT(''DCL'');
  IF ~LIT('::') THEN CALL ERR('MISSING ::');
  IF ~LIT('PROCESSOR') THEN CALL ERR('MISSING PROCESSOR');
  IF ~LIT('(') THEN CALL ERR('MISSING LEFT PAREN');
  IF ~BUSLIST THEN CALL ERR('MISSING BUS LIST');
  IF ~LIT(')') THEN CALL ERR('MISSING RIGHT PAREN');
  IF ~LIT(')') THEN CALL ERR('MISSING ;');
  CALL OUT(';;'); STMT=STMT+1;
  RETURN( TRUE ); END PROCDECL;

BUSLIST ::= BUSREF $( 'BUSREF' ) ;
BUSLIST:PROCEDURE RETURNS( BIT(1) );
  IF ~BUSREF THEN RETURN( FALSE );
  L5: IF ~LIT(')') THEN RETURN( TRUE );
  CALL OUT('); CALL NL;
  IF ~BUSREF THEN CALL ERR('MISSING BUS REF');
  GO TO L5; END BUSLIST;

BUSREF ::= ID OPT( '<' INT OPT( '::' INT '>::' INT ) '>'; ) ;
BUSREF:PROCEDURE RETURNS( BIT(1) );
DCL INT1 CHAR(5) VARYING, ID1 CHAR(8) VARYING;
  IF ~ID THEN RETURN( FALSE );
  ID1=STAR; CALL OUT(PNAME||''_''|STAR);
  IF ~LIT('<') THEN DO; CALL OUT(' BIT(1) DEFINED '||STAR);
      RETURN( TRUE ); END;
  IF ~INT THEN CALL ERR('MISSING LOWER SUBSCRIPT');
  INT1=STAR;
  IF ~LIT('::') THEN DO; CALL OUT(' BIT(1) DEFINED '||ID1||
      ' POSITION('||STAR||''));
      GO TO L6; END;
  IF ~INT THEN CALL ERR(' MISSING UPPER SUBSCRIPT');
CALL OUT(' BIT('||STAR||'-'||INT1||'*+1) DEFINED '||ID1||'
    POSITION('||INT1||'*'));
L6: IF -LIT('>') THEN CALL ERR('MISSING RIGHT BRACKET >');
    RETURN( TRUE );    END BUSREF;
END PROC;
{SUBRG, STRG}:

**FACDCL**: procedure returns (bit(1));

/* FACDCL ::= REGSTMT | RAMSTMT | ROMSTMT | CAMSTMT | */

/* TERMSTMT | DEFSTMT | TIMESTMT | ADDSTMT; */

DCL id1 char(8) varying, id2 char(8) varying, id3 char(8) varying,
   id4 char(8) varying, id5 char(8) varying, int1 char(5) varying,
   int2 char(5) varying;

DCL stmt fixed bin external,
   true bit(1) static init('1'b),
   false bit(1) static init('0'b),
   lit entry (char(*) ) returns (bit(1)),
   out entry (char(*) varying ),
   err entry (char(*) ),
   nl entry,
   id entry returns (bit(1)),
   int entry returns (bit(1)),
   star char(32) varying external,
   pname char(8) varying external,
   facdcl entry returns (bit(1)),
   registmt entry returns (bit(1)),
   regdcl entry returns (bit(1)),
   ramstmt entry returns (bit(1)),
   ramdcl entry returns (bit(1)),
   romstmt entry returns (bit(1)),
   romdcl entry returns (bit(1)),
   camstmt entry returns (bit(1)),
   camdcl entry returns (bit(1)),
   termstmt entry returns (bit(1)),
   defstmt entry returns (bit(1)),
   defdcl entry returns (bit(1)),
   timestmt entry returns (bit(1)),
   proctime entry returns (bit(1)),
   proco entry returns (bit(1)),
   transfer entry returns (bit(1)),
   addstmt entry returns (bit(1)),

79
ADDCL ENTRY RETURNS( BIT(1) ),
CONSTANT ENTRY RETURNS( BIT(1) ),
FILE FIXED BIN EXTERNAL;

FILE=5;
IF REGSTMT THEN GO TO L1;
IF RAMSTMT THEN GO TO L1;
IF ROMSTMT THEN GO TO L1;
IF CAMSTMT THEN GO TO L1;
IF TERMSTMT THEN GO TO L1;
IF DEFDSTMT THEN GO TO L1;
IF TIMESTMT THEN GO TO L1;
IF ADDSTMT THEN GO TO L1;
RETURN( FALSE );
L1: STMT=STMT+1; RETURN( TRUE );

/* REGSTMT ::= 'REG' REGDCL $( ',', REGDCL ) ; */
REGSTMT:PROCEDURE RETURNS( BIT(1) );
  IF -LIT('REG') THEN RETURN( FALSE );
  CALL OUT('DCL ');
  IF -REGDCL THEN CALL ERR('MISSING REGDCL');
L2: IF -LIT(')') THEN RETURN( TRUE );
  CALL OUT(')'); CALL NL;
  IF -REGDCL THEN CALL ERR('MISSING REGDCL2');
  GO TO L2; END REGSTMT;

/* REGDCL ::= ID .OPT! '<' INT •>' ) ; */
REGDCL:PROCEDURE RETURNS( BIT(1) );
  IF -ID THEN RETURN( FALSE );
  CALL OUT(PNAME||'.'||STAR);
  IF -LIT('<') THEN DO; CALL OUT(' BIT(1)'); RETURN( TRUE ); END;
  IF -INT THEN CALL ERR('MISSING UPPER SCRIPT ON DCL');
  IF -LIT('>') THEN CALL ERR('MISSING RIGHT BRACKET > '); CALL OUT(' BIT'||STAR||''));
RETURN( TRUE ); END REGDCL;

/* RAMSTMT ::= 'RAM' RAMDCL $( ',', RAMDCL ) ; */
RAMSTMT: PROCEUDRE RETURNS ( BIT(1) );
    IF ~LIT('RAM') THEN RETURN ( FALSE );
    IF ~RAMDCL THEN CALL ERR('MISSING RAM DCL1');
L3:  IF ~LIT(',',') THEN RETURN ( TRUE );
    IF ~RAMDCL THEN CALL ERR('MISSING RAM DCL2');
    GO TO L3; END RAMSTMT;

    /* RAMDCL ::= ID*OPT('<'INT'>')='ID'('ID'<'INT'>')'; */
RAMDCL: PROCEDURE RETURNS ( BIT(1) );
    IF ~ID THEN RETURN ( FALSE );
    CALL NL; INT1='1';
    ID1=STAR; CALL OUT('DCL | |PNAME||_' | |STAR); 
    IF ~LIT('<') THEN DO; CALL OUT(' BIT(1)','); GO TO L4; END;
    IF ~INT THEN CALL ERR('MISSING SUBSCRIPT');
    INT1=STAR; CALL OUT(' BIT(1)| |STAR| |)',');
    IF ~LIT('>') THEN CALL ERR('MISSING >');
L4:  IF ~LIT('=') THEN CALL ERR('MISSING = IN RAM DCL ');
    IF ~ID THEN CALL ERR('MISSING MEMORY NAME IN RAM DCL');
    ID2=STAR;
    IF ~LIT('*') THEN CALL ERR('MISSING *');
    IF ~ID THEN CALL ERR('MISSING ADDR REG');
    ID3=STAR; CALL NL; CALL OUT(PNAME||_' | |STAR); 
    IF ~LIT('<') THEN CALL ERR('MISSING <');
    IF ~INT THEN CALL ERR('MISSING SCRIPT');
    INT2=STAR; CALL OUT(' BIT(1)| |STAR| |)',');
    IF ~LIT('>') THEN CALL ERR('MISSING >');
    IF ~LIT('**') THEN CALL ERR('MISSING RIGHT PAREN IN RAM DCL');
    CALL NL;
    CALL CUT(PNAME||_' | |ID2|| '(0:2**| |INT2|| '-1)BIT(1||INT1||)');
    CALL LAB('READ_ | |PNAME||_' | |ID2); CALL OUT('PROCEDURE;');
    CALL CUT(PNAME||_' | |ID1|| '=' | |PNAME||_' | |ID2|| '(' | |PNAME||_' | |ID3|| ')'); END;
    CALL LAB('WRITE_ | |PNAME||_' | |ID2); CALL OUT('PROCEDURE;');
    CALL OUT(PNAME||_' | |ID2|| '(' | |PNAME||_' | |ID3|| ')=' | |PNAME||_' | |ID1|| ' '); END;
    RETURN ( TRUE ); END RAMDCL;
/* ROMSTMT ::= 'RAM' ROMDCL $(',', ROMDCL ) ; */
ROMSTMT:PROCEDURE RETURNS( BIT(1) );
  IF -LIT('ROM') THEN RETURN( FALSE );
  IF -ROMDCL THEN CALL ERR('MISSING ROM DCL 1');
L5: IF -LIT(',') THEN RETURN( TRUE );
  IF -ROMDCL THEN CALL ERR('MISSING ROM DCL2');
  GO TO L5; END ROMSTMT;

/* ROMDCL ::= ID.OPT('<'INTID'>')='ID'('ID'<INTID'>')' ; */
ROMDCL:PROCEDURE RETURNS( BIT(1) );
  IF -ID THEN RETURN( FALSE );
  CALL NL; INT1='1';
  ID1=STAR; CALL OUT('DCL '||PNAME||'_'||STAR);
  IF -LIT('<') THEN DO; CALL OUT(' BIT(1)'); GO TO L6; END;
  IF -INT THEN CALL ERR('MISSING SUBSCRIPT');
  INT1=STAR; CALL OUT(' BIT('||STAR||')',');  
  IF -LIT('>') THEN CALL ERR('MISSING >');
L6: IF -LIT('=') THEN CALL ERR('MISSING = IN RAM DCL');
  IF -ID THEN CALL ERR('MISSING MEMORY NAME IN RAM DCL');
  ID2=STAR;
  IF -LIT('(') THEN CALL ERR('MISSING (');
  IF -ID THEN CALL ERR('MISSING ADDR REG');
  ID3=STAR; CALL NL; CALL OUT(PNAME||'_'||STAR);
  IF -LIT('<') THEN CALL ERR('MISSING <');
  IF -INT THEN CALL ERR('MISSING SCRIPT');
  INT2=STAR; CALL OUT(' BIT('||STAR||')',');  
  IF -LIT('>') THEN CALL ERR('MISSING >');
  IF -LIT(')') THEN CALL ERR('MISSING RIGHT PAREN IN RAM DCL');
  CALL NL;
  CALL OUT(PNAME||'_'||ID2||(0:2**||INT2||'-1')BIT(||INT1||')');
  CALL LAB('READ_'||PNAME||'_'||ID2); CALL OUT('PROCEDURE;');
  CALL OUT(PNAME||'_'||ID1||=''||PNAME||'_'||ID2||(''||PNAME||'_'||ID3|')');END;);
  RETURN( TRUE ); END ROMDCL;

/* CAMSTMT ::= 'CAM' CAMDCL $(',', CAMDCL ) ; */
CAMSTMT:PROCEDURE RETURNS( BIT(1) );
IF not LIT('CAM') THEN RETURN( FALSE );
IF not CAMDCL THEN CALL ERR('MISSING CAM DCL1');
L7: IF not LIT(',') THEN RETURN( TRUE );
IF not CAMDCL THEN CALL ERR('MISSING CAM DCL2');
GO TO L7; END CAMSTMT:

/*
CAMDCL ::= ID('ID'( 'ID'('ID'< 'INT'>')')')' = ' ID('ID'< 'INT'>')';
CAMDCL:PROCEDURE RETURNS( BIT(1) );
IF not ID THEN RETURN( FALSE );
ID1=STAR; INT1='1';
IF not LIT(')') THEN CALL ERR('MISSING LEFT PAREN');
IF not ID THEN CALL ERR('MISSING ID');
ID2=STAR;
IF not LIT(')') THEN CALL ERR('MISSING LEFT PAREN');
IF not ID THEN CALL ERR('MISSING REG');
ID3=STAR;
IF not LIT('<') THEN GO TO L8;
IF not INT THEN CALL ERR('MISSING SUBSCRIPT');
INT1=STAR;
IF not LIT(>') THEN CALL ERR('MISSING >');
L8: IF not LIT(')') THEN CALL ERR('MISSING (');
IF not LIT(')') THEN CALL ERR('MISSING )');
IF not ID THEN CALL ERR('MISSING ID');
ID4=STAR;
IF not LIT(')') THEN CALL ERR('MISSING LEFT PAREN');
IF not ID THEN CALL ERR('MISSING ADDR REG');
ID5=STAR;
IF not LIT('<') THEN CALL ERR('MISSING <');
IF not INT THEN CALL ERR('MISSING SUBSCRIPT');
INT2=STAR;
IF not LIT(>') THEN CALL ERR('MISSING >');
IF not LIT(')') THEN CALL ERR('MISSING RIGHT PAREN');
IF not LIT(')') THEN CALL ERR('MISSING RIGHT PAREN');
CALL NL; CALL OUTCDCL ID5| ID2| ID3| INT2| INT1| NLPCAM| ID4| INT1| INT2| INT5| BIT(1| INT2| INT1| NLPCAM);
CALL NL;
CALL OUT(PNAME || ' ' || ID1 || ' BIT(2**I || INT2 || ')', ');
CALL NL;
CALL OUT(PNAME || ' ' || ID4 || ' (0:2**I || INT2 || '-1) BIT(' || INT1 || ')', ');
CALL LAB('READ_' || PNAME || ' ' || ID4);
CALL OUT('PROCEDURE; ');
CALL OUT(PNAME || ' ' || ID4 || '='' || PNAME || ' ' || ID4 || '({)');</
CALL OUT(PNAME || ' ' || ID5 || '); END;');</
CALL LAB('WRITE_' || PNAME || ' ' || ID4);
CALL OUT('PROCEDURE;');</
CALL OUT(PNAME || ' ' || ID4 || '='' || PNAME || ' ' || ID4 || '({)');</
CALL OUT('PROCEDURE; DCL I FIXED BIN;');</
CALL NL;
CALL OUT('DO I =1 TO 2**I || INT2 || ');
CALL NL;
CALL OUT('SUBSTR(' || PNAME || ' ' || ID1 || ' ', I, 1) = (');</
CALL OUT(PNAME || ' ' || ID2 || ' ' || PNAME || ' ' || ID3);
CALL OUT('(' || PNAME || ' ' || ID2 || ' ' || PNAME || ' ' || ID4);
CALL OUT(') = (' || PNAME || ' ' || ID2 || ' ' || PNAME || ' ' || ID4);
CALL OUT('(' || ID-1 || ') End; End;');</
RETURN( TRUE ); END CAMDCL;

/* TERMSTMT ::= 'TERM' TRANSFER $( TRANSFER ) ; */
TERMSTMT: PROCEDURE RETURNS BIT(1); IF NOT LIT('TERM') THEN RETURN( FALSE ); CALL NL; FILE=2;
IF NOT TRANSFER THEN CALL ERR('MISSING TRANSFER STMT');
L9: IF NOT LIT(',' ) THEN DO; CALL NL; FILE=5; RETURN( TRUE ); END;
IF NOT TRANSFER THEN CALL ERR('MISSING TRANSFER2');
GO TO L9; END TERMSTMT;

/* DEFSSTMT ::= 'DEF' DEFDCL $( ',' DEFDCL ) ; */
DEFSSTMT: PROCEDURE RETURNS BIT(1); IF NOT LIT('DEF') THEN RETURN( FALSE ); CALL OUT('DCL ');
IF NOT DEFDCL THEN CALL ERR('MISSING DEFDCL');
LA: IF ¬LIT(';',') THEN RETURN( TRUE );
    CALL OUT(';',')'; CALL NL;
    IF ¬DEFDCL THEN CALL ERR('MISSING DEFDCL2');
    GO TO LA; END DEFSTMT;

/*
   DEFDCL ::= ID(OPT('<INT>')| '<INT>')::='ID'<INT';
   PROCEDURE RETURNS! BIT(1) );
*/

DEFDCL:PROCEDURE RETURNS( BIT(1) );
    IF ¬ID THEN RETURN( FALSE );
    CALL OUT(PNAME||'_'||STAR);
    IF ¬LIT('::='') THEN GO TO LB;
    CALL OUT(' BIT(1) DEF ');
    IF ¬ID THEN CALL ERR('MISSING IDENTIFIER');
    CALL OUT(PNAME||'_'||STAR);
    IF ¬LIT('<') THEN RETURN( TRUE );
    IF ¬INT THEN CALL ERR('MISSING INTEGER');
    CALL OUT(' POSITION('||STAR|'|'));
    IF ¬LIT('>') THEN CALL ERR('MISSING >');
    RETURN( TRUE );

LB: IF ¬LIT('<') THEN CALL ERR('MISSING <');
    IF ¬INT THEN CALL ERR('MISSING SUBSCRIPT');
    CALL OUT(' BIT('||STAR|'|') DEF ');
    IF ¬LIT('>') THEN CALL ERR('MISSING >');
    IF ¬LIT('::='') THEN CALL ERR('MISSING ::=');
    IF ¬ID THEN CALL ERR('MISSING IDENTIFIER');
    CALL OUT(PNAME||'_'||STAR);
    IF ¬LIT('<') THEN CALL ERR('MISSING <');
    IF ¬INT THEN CALL ERR('MISSING SUBSCRIPT');
    CALL OUT(' POSITION('||STAR|'|'));
    IF ¬LIT('>') THEN CALL ERR('MISSING >');
    RETURN( TRUE ); END DEFDCL;

/*
   TIMESTMT ::= 'TIME' PROCTIME $(',',' PROCTIME ) ;
*/
TIMESTMT:PROCEDURE RETURNS( BIT(1) );
    IF ¬LIT('TIME') THEN RETURN( FALSE );
IF \( \neg \text{PROCTIME} \) THEN CALL ERR("MISSING TIME DCL");

LC: IF \( \neg \text{LIT}('*,') \) THEN RETURN( TRUE );
IF \( \neg \text{PROCTIME} \) THEN CALL ERR("MISSING TIME DCL");
GO TO LC; END TIMESTMT;

/*
   \text{PROCTIME} ::= \text{PROCP} \ '*$' \text{INT} ;
*/

\text{PROCTIME}:\text{PROCEDURE} \text{RETURNS( BIT(1) )};
IF \( \neg \text{PROCP} \) THEN RETURN( FALSE );
IF \( \neg \text{LIT}('*$') \) THEN CALL ERR("MISSING $'");
IF \( \neg \text{INT} \) THEN CALL ERR("MISSING INTEGER");
RETURN( TRUE ); \text{END PROCTIME};

/*
   \text{PROCOP} ::= 'READ'|'WRITE'|'SEARCH'|'\neg'|'+|'\|'**'
   'ADD.'|'\text{SUB.}';
*/

\text{PROCOP}:\text{PROCEDURE} \text{RETURNS( BIT(1) )};
IF \( \neg \text{LIT}('READ') \) THEN RETURN( TRUE );
IF \( \neg \text{LIT}('WRITE') \) THEN RETURN( TRUE );
IF \( \neg \text{LIT}('SEARCH') \) THEN RETURN( TRUE );
IF \( \neg \text{LIT}('\neg') \) THEN RETURN( TRUE );
IF \( \neg \text{LIT}('+') \) THEN RETURN( TRUE );
IF \( \neg \text{LIT}('**') \) THEN RETURN( TRUE );
IF \( \neg \text{LIT}('ADD') \) THEN RETURN( TRUE );
IF \( \neg \text{LIT('\text{SUB.}') \) THEN RETURN( TRUE );
RETURN( FALSE ); \text{END PROCOP};

/*
   \text{ADDSTMT} ::= 'ADDER' \text{ADDCL} \{'','\text{ADDCL}'\} ;
*/

\text{ADDSTMT}:\text{PROCEDURE} \text{RETURNS( BIT(1) )};
IF \( \neg \text{LIT}('ADDER') \) THEN RETURN( FALSE );
IF \( \neg \text{ADDCL} \) THEN CALL ERR("MISSING ADDCL");
LD: IF \( \neg \text{LIT}(','') \) THEN RETURN( TRUE );
IF \( \neg \text{ADDCL} \) THEN CALL ERR("MISSING ADDCL2");
GO TO LD; END ADDSTMT;

/*
   \text{ADDCL} ::= \text{ID1}('ID2'='ID3',\text{ADD.}'(\text{CONSTANT}|\text{ID}')\}'
*/

\text{ADDCL}:\text{PROCEDURE} \text{RETURNS( BIT(1) )};
IF \( \neg \text{ID} \) THEN RETURN( FALSE );
\text{ID1} \equiv \text{STAR};
IF ¬LIT(') THEN CALL ERR('MISSING(');
IF ¬ID THEN CALL ERR('MISSING ID');
ID2=STAR;
IF ¬LIT('=',) THEN CALL ERR('MISSING =');
IF ¬ID THEN CALL ERR('MISSING ID');
ID3=STAR;
IF ¬LIT('.ADD.',) THEN CALL ERR('MISSING .ADD.');
CALL LAB('ADD_'||PNAME||'_'||ID1);
CALL OUT('PROCEDURE;');
CALL OUT(PNAME||'_'||ID2||'=SUBSTR(UNSPEC(');
CALL OUT(PNAME||'_'||ID3||'+');
IF CONSTANT THEN GO TO LE;
IF ¬ID THEN CALL ERR('MISSING CONSTANT OR ID');
CALL OUT(PNAME||'_'||STAR);
LE: CALL OUT('),33-LENGTH('||PNAME||'_'||ID2);
CALL OUT('),LENGTH('||PNAME||'_'||ID2');END;');
IF ¬LIT(')') THEN CALL ERR('MISSING )');
RETURN( TRUE ); END ADDCL;
END FACDCCL;
{SUBRSG,STRG}:
STATE::PROCEDURE RETURNS( BIT(1) );

/*
#A LABEL CN ACTIONS
*/

/*
#B CONDITION VARIABLES
*/

/*
#C COMMA LABELS
*/

/*
#D PARALLEL SCHEDULING LABELS
*/

/*
#N CONDITION LIST LABELS
*/

/*
#P PARALLEL CONTROL LABELS
*/

/*
#T SIMULATE LABELS
*/

/*
#S PARALLEL CONTROL VARIABLES
*/

/*
STATE ::= ID*:'OPT!ACTIONLIST«.THEN.MCONDLIST1 GOTO) ;
*/

DCL ID ENTRY RETURNS( BIT(1) ),
    TRUE BIT(1) STATIC INIT('1'B),
    FALSE BIT(1) STATIC INIT('0'B),
    STAR CHAR(32) VARYING EXTERNAL,
    PNAME CHAR(8) VARYING EXTERNAL,
    SNAME CHAR(8) VARYING EXTERNAL,
    LAB ENTRY( CHAR(*) VARYING),
    OUT ENTRY( CHAR(*) VARYING),
    ERR ENTRY( CHAR(*) ),
    LIT ENTRY( CHAR(*) ) RETURNS( BIT(1) ),
    NL ENTRY,
    ACTIONLIST ENTRY RETURNS( BIT(1) ),
    STATE ENTRY RETURNS( BIT(1) ),
    ACTION ENTRY RETURNS( BIT(1) ),
    ACT ENTRY RETURNS( BIT(1) ),
    CONDLIST ENTRY RETURNS( BIT(1) ),
    COND ENTRY RETURNS( BIT(1) ),
    GOTO ENTRY RETURNS( BIT(1) ),
    TRANSFER ENTRY RETURNS( BIT(1) ),
    MEMORY ENTRY RETURNS( BIT(1) ),
    ARITH ENTRY RETURNS( BIT(1) ),
    EXPR ENTRY RETURNS( BIT(1) ),
    INT ENTRY RETURNS( BIT(1) ),
    STMT FIXED BIN EXTERNAL,
FILE FIXED BIN EXTERNAL,
REGREF ENTRY RETURNS( BIT(1) );

DCL AN FIXED BIN STATIC, /* ACTION NUMBER WITHIN STATE */
ANC CHAR(2) STATIC, /* ACTION NUMBER CHARACTER */
PN FIXED BIN STATIC, /* PAREN NUMBER WITHIN STATE */
PNC CHAR(2) STATIC, /* PAREN NUMBER CHARACTER */
CN FIXED BIN STATIC, /* COMMA NUMBER WITHIN PARENS */
CNC CHAR(2) STATIC, /* COMMA NUMBER CHARACTER */
CSTK(1:9) FIXED BIN STATIC, /* COMMA STACK */
PSTK(1:9) FIXED BIN STATIC, /* PAREN STACK */
MP FIXED BIN STATIC, /* MAX NO. PARENS */
LN FIXED BIN STATIC, /* INDEX TO CSTK AND PSTK */
BN FIXED BIN STATIC, /* COND NUMBER */
BNC CHAR(2) STATIC, /* COND NUMBER CHARACTER */
K FIXED BIN STATIC,
KC CHAR(2) STATIC,
T FIXED BIN STATIC, /* TIME FOR ACTION */
TC CHAR(5) STATIC; /* TIME FOR ACTION CHARACTERS */

IF →ID THEN RETURN( FALSE );
SNAME=STAR; LN=1; MP=1;
BN=0; CALL NL; FILE=1;
AN=1; ANC='01'; CN=1; Cnc='01'; PN=1; Pnc='01'; T=0; TC='00000';
CALL LAB('#'||PNAME||'.||SNAME);
CALL OUT('CALL INIT_||PNAME||.||');
CALL OUT('PUT FILE(SYSPRINT)');
CALL OUT('EDIT("IN STATE ");
CALL OUT(PNAME||'.||||STAR);
CALL OUT('||||SKlP,A||');
CALL OUT('CALL $CH( #A'||PNAME||'.||SNAME||_01,0):');
CALL OUT('GO TO #C'||PNAME||'.||SNAME||_01_01:');
IF ←LIT(':'') THEN CALL ERR('MISSING : IN STATE');
IF ←ACTIONLIST THEN ;
IF ←LIT('THEN.') THEN CALL ERR('MISSING THEN.);
CALL OUT('CALL $CH(#D1'||PNAME||'.||SNAME||.||ANC||','||TC||');');
CALL OUT('GO TO $CONTROL;');
CALL LAB('#D1'|PNAME||'_'||SNAME||'_'||ANC);
CALL NL; /* DEBUG */
CALL OUT('#S_||PNAME||'_'||SNAME||'_'||PNC||'('||CNC||')='"1"B;');
CALL NL; FILE=4;
CALL OUT('#S_'|IPNAME|'|SNAME|'|PNC|'|'('|CNC|'|')='"0"B;');
CALL NL; FILE=1;
CALL OUT('GO TO $CONTROL;');
CALL LAB('#P_||PNAME||'_'||SNAME||'_'||PNC);
CALL OUT('IF ~();
   DO K=1 TO CN-1; PUT STRING(KC)EDIT(K)(P\'99'); CALL NL;
   CALL OUT('#S_'|IPNAME|'|SNAME|'|PNC|'|'('|KC|'|')&');
   END;
   CALL NL;
   CALL OUT('#S_'|IPNAME|'|SNAME|'|PNC|'|M|'|CNC|'|');
   CALL OUT(' THEN GO TO $CONTROL;');
   CALL NL; FILE=5;
   CALL OUT('DCL #S_||PNAME||'_'||SNAME||'_'||PNC|'|'('|CNC|'|')BIT(1);');
   CALL NL; FILE=1;
   IF CONDLIST THEN RETURN( TRUE );
   IF GOTO THEN RETURN( TRUE );
   CALL ERR('MISSING COND LIST OR GO TO');
/* ACTIONLIST ::= ACTION $( ACTION I ACTION ) ; */
ACTIONLIST:PROCEDURE RETURNS( BIT(1) ) RECURSIVE;
   IF ~ACTION THEN RETURN( FALSE );
L1: IF ~LIT(',*') THEN GO TO L2;
   CALL OUT('CALL $CH(#D2'||PNAME||'_'||SNAME||'_'||ANC||',',|TC|');');
   CALL OUT('GO TO $CONTROL;');
   CALL LAB('#D2'||PNAME||'_'||SNAME||'_'||ANC);
   CALL NL; /* DEBUG */
   CALL OUT('#S_||PNAME||'_'||SNAME||'_'||PNC|'|'('|CNC|'|')='"1"B;');
   CALL NL; FILE=4;
   CALL OUT('#S_||PNAME||'_'||SNAME||'_'||PNC|'|'('|CNC|'|')='"0"B;');
   CALL NL; FILE=1;
CALL OUT('GO TO #P_||PNAME||'_'||SNAME||'_'||PNC||'');
CALL LAB('#C_||PNAME||'_'||SNAME||'_'||PNC||'');
CALL OUT('CALL $CH(#A_||PNAME||'_'||SNAME||'_'||ANC); AN=AN+1; PUT STRING(ANC)E0IT(AN)(P'99');
CALL OUT('GO TO #C_||PNAME||'_'||SNAME||'_'||PNC||'');
    IF -ACTION THEN CALL ERR('MISSING ACTION');
    GO TO L1;
L2: IF -LIT('') THEN RETURN( TRUE );
    CALL NL; /* DEBUG */
CALL OUT('CALL $CH(#A_||PNAME||'_'||SNAME||'_'||ANC); AN=AN+1; PUT STRING(ANC)E0IT(AN)(P'99');
CALL OUT('GO TO $CONTROL;');
    IF -ACTION THEN CALL ERR('MISSING ACTION');
    GO TO L1; END ACTIONLIST;

/* ACTION ::= '(' ACTIONLIST ')' | ACT */
ACTION:PROCEDURE RETURNS( BIT(1) ) RECURSIVE;
    IF -LIT('(') THEN GO TO L3;
    CALL NL; /* DEBUG */
CALL LAB('#A_||PNAME||'_'||SNAME||'_'||ANC);
    CALL NL; /* DEBUG */
CALL OUT('CALL $CH(#A_||PNAME||'_'||SNAME||'_'||ANC); AN=AN+1; PUT STRING(ANC)E0IT(AN)(P'99');
CALL OUT('GO TO #C_||PNAME||'_'||SNAME||'_'||PNC||'');
    IF -ACTION THEN CALL ERR('MISSING ACTION LIST');
    IF -LIT(')') THEN CALL ERR('MISSING CLOSING PAREN');
CALL OUT('CALL $CH(#D3||PNAME||'_'||SNAME||'_'||ANC); AN=AN+1; PUT STRING(ANC)E0IT(AN)(P'99');
CALL OUT('GO TO $CONTROL;');
    CALL LAB('#D3||PNAME||'_'||SNAME||'_'||ANC);
    CALL NL; /* DEBUG */
CALL OUT('#S_||PNAME||'_'||SNAME||'_'||PNC||'');
    CALL NL; FILE=4;
CALL OUT('#S_||PNAME||'_'||SNAME||'_'||PNC||'');
    CALL NL; FILE=1;
CALL OUT('GO TO #P_||PNAME||'_'||SNAME||'_'||PNC||'');
CALL LAB('#C_||PNAME||'_'||SNAME||'_'||PNC||'');
CALL OUT('GO TO $CONTROL;');
CALL LAB('#P_'||PNAME||'_'||SNAME||'_'||PNC);
CALL OUT('IF NOT(');
    DO K=1 TO CN-1; PUT STRING(KC)EDIT(K)(P'99'); CALL NL;
            CALL OUT('#S_'||PNAME||'_'||SNAME||'_'||PNC||'('||KC||')&');
    END;
CALL NL;  TC=00000;
CALL OUT('#S_'||PNAME||'_||SNAME||'_||PNC||'('||CNC||')
CALL OUT(' THEN GO TO $CONTROL;');
CALL NL;  FILE=5;
CALL OUT('DCL #S_'||PNAME||'_||SNAME||'_||PNC||'('||CNC||')BIT(1);');
CALL NL;  FILE=1;
LN=LN-1; CN=CSTK(LN); PN=PSTK(LN);
PUT STRING(CNC)EDIT(CN)(P'99'); PUT STRING(PNC)EDIT(PN)(P'99');
RETURN( TRUE );
L3: CALL LAB('#A_'||PNAME||'_*||SNAME||'_||ANC);
    IF NOT(ACT) THEN RETURN( FALSE );
    CALL OUT('SIGNAL CONDITION($TRACE); M;
    STMT=STMT+1; AN=AN+1; PUT STRING(ANC)EDIT(AN)(P'99');
    RETURN( TRUE ); END ACTION;
/* CONDLIST ::= COND $( •>• COND ) ; */
CONDLIST:PROCEDURE RETURNS( BIT(1) );
    CALL LAB('#N_'||PNAME||'_||SNAME);
    IF NOT(COND) THEN RETURN( FALSE );
L4:IF NOT(LIT('>') THEN DO; CALL OUT('CALL $CH(#N_'||PNAME||'_||SNAME||'
L4:IF NOT(LIT('>') THEN DO; CALL OUT('CALL $CH(#N_'||PNAME||'_||SNAME||'
L4:IF NOT(LIT('>') THEN DO; CALL OUT('CALL $CH(#N_'||PNAME||'_||SNAME||'
/* COND ::= '(' EXPR ')• ID ; */
COND:PROCEDURE RETURNS( BIT(1) );
    IF NOT(LIT('')) THEN RETURN( FALSE );
    BN=BN+1; PUT STRING(BNC)EDIT(BN)(P'99');
CALL NL;  FILE=5;
CALL OUT(*DCL #B_'||PNAME||'_'||SNAME||'_'||BNC||' BIT(1);');
CALL NL; FILE=3;
CALL OUT(*B_'||PNAME||'_'||SNAME||'_'||BNC||'=');
IF -EXPR THEN CALL ERR('MISSING EXPR IN CONDITION');
CALL OUT(''); CALL NL; FILE=1;
IF -LIT('') THEN CALL ERR('MISSING )');
IF -ID THEN CALL ERR('MISSING NEXT STATE');
CALL NL; CALL OUT(' IF #B_'||PNAME||'_'||SNAME||'_'||BNC); CALL OUT(' THEN GO TO #'||PNAME||'_'||STAR||';'); RETURN( TRUE ); END COND;

/* GO TO ::= 'GOTO.' ID ; */
GOTO:PROCEDURE RETURNS( BIT(1) );
IF -LIT('GOTO.') THEN RETURN( FALSE );
IF -ID THEN CALL ERR('MISSING NEXT STATE');
CALL NL; CALL OUT('GO TO #'||PNAME||'_'||STAR||';'); RETURN( TRUE ); END GOTO;

/* ACT ::= MEMORY *OPT( '$' INT ) | */
/* ARITH *OPT( '$' INT ) | */
/* TRANSFER *OPT( '$' INT ) ; */
ACT:PROCEDURE RETURNS( BIT(1) ) RECURSIVE;
TC='00000';
IF -MEMORY THEN GO TO L5;
IF -LIT('$') THEN DO; TC='00001'; RETURN( TRUE ); END;
IF -INT THEN CALL ERR('MISSING INTEGER');
K=LENGTH(STAR); SUBSTR(TC,6-K,K)=STAR;
RETURN( TRUE );

L5: IF -ARITH THEN GO TO L8;
IF -LIT('$') THEN DO; TC='00001'; RETURN( TRUE ); END;
IF -INT THEN CALL ERR('MISSING INTEGER');
K=LENGTH(STAR); SUBSTR(TC,6-K,K)=STAR;
RETURN( TRUE );

L8: IF -TRANSFER THEN RETURN( FALSE );
IF -LIT('$') THEN DO; TC='00001'; RETURN( TRUE ); END;
IF -INT THEN CALL ERR('MISSING INTEGER');
K=LENGTH(STAR); SUBSTR(TC,6-K,K)=STAR;
RETURN( TRUE ); END ACT;

/* MEMORY ::= 'READ' '(' ID ')' | 'WRITE' '(' ID ')' | */
/* 'SEARCH' '(' ID ')' */
MEMORY: PROCEDURE RETURNS( BIT(1) );
    IF -LIT('READ') THEN GO TO L6;
    IF -LIT('WRITE') THEN CALL ERR('MISSING ()');
    IF -LIT('READ') THEN ERR('MISSING NAME OF MEMORY');
    IF -LIT('WRITE') THEN ERR('MISSING ) ');
    CALL NL; CALL OUT('CALL READ_'||PNAME||'_'||STAR||');
    RETURN( TRUE );
L6: IF -LIT('WRITE') THEN GO TO L7;
    IF -LIT('READ') THEN CALL ERR('MISSING ()');
    IF -LIT('WRITE') THEN ERR('MISSING MEMORY NAME');
    IF -LIT('READ') THEN ERR('MISSING ) ');
    CALL NL; CALL OUT('CALL WRITE_'||PNAME||'_'||STAR||');
    RETURN( TRUE );
L7: IF -LIT('SEARCH') THEN RETURN( FALSE );
    IF -LIT('READ') THEN CALL ERR('MISSING ()');
    IF -LIT('WRITE') THEN ERR('MISSING NAME OF MEMORY');
    IF -LIT('READ') THEN ERR('MISSING ) ');
    CALL NL; CALL OUT('CALL SEARCH_'||PNAME||'_'||STAR||');
    RETURN( TRUE ); END MEMORY;

/* ARITH ::= 'ADD' '(' ID ')' */
ARITH: PROCEDURE RETURNS( BIT(1) );
    IF -LIT('ADD') THEN RETURN( FALSE );
    IF -LIT('ADD') THEN CALL ERR('MISSING (');
    IF -LIT('ADD') THEN ERR('MISSING ID');
    CALL OUT('CALL ADD_'||PNAME||'_'||STAR||');
    IF -LIT('ADD') THEN CALL ERR('MISSING ) ');
    RETURN( TRUE ); END ARITH;
END STATE;
TRANSFER:PROCEDURE RETURNS( BIT(1) );

/*
TRANSFER ::= REGREF = EXPR ;
*/

DCL REGREF ENTRY RETURNS( BIT(1) ),
TRUE BIT(1) STATIC INIT('1'B),
FALSE BIT(1) STATIC INIT('0'B),
LIT ENTRY( CHAR(*) ) RETURNS( BIT(1) ),
ERR ENTRY( CHAR(*) ),
OUT ENTRY( CHAR(*) VARYING ),
EXPR ENTRY RETURNS( BIT(1) );

IF ¬REGREF THEN RETURN( FALSE );
IF ¬LIT('=') THEN CALL ERR('MISSING = IN TRANSFER STMT');
CALL OUT('=');
IF ¬EXPR THEN CALL ERR('MISSING EXPR IN TRANSFER');
CALL OUT('!');
RETURN( TRUE );
END TRANSFER;
(SUBRGTSTRGL:
SIMULATE:PROCEDURE RETURNS( BIT(1) );

/*
SIMULATE ::= 'SIMULATE.'SIMSTMT$(";"OPT(SIMSTMT))' END'; */

DCL FALSE BIT(1) STATIC INIT('0'B),
TRUE BIT(1) STATIC INIT('1'B),
STMT FIXED BIN EXTERNAL,
FILE FIXED BIN EXTERNAL,
TN FIXED BIN STATIC,
TNC CHAR(2) STATIC,
ID1 CHAR(8) VARYING STATIC,
ID2 CHAR(8) VARYING STATIC,
INT ENTRY RETURNS( BIT(1) ),
INT1 CHAR(5) VARYING STATIC,
INT2 CHAR(5) VARYING STATIC,
ID ENTRY RETURNS( BIT(1) ),
ERR ENTRY( CHAR(*) ),
OUT ENTRY( CHAR(*) VARYING ),
LAB ENTRY( CHAR(*) VARYING ),
NL ENTRY,
LIT ENTRY( CHAR(*) ) RETURNS( BIT(1) ),
CONSTANT ENTRY RETURNS( BIT(1) ),
SIMSTMT ENTRY RETURNS( BIT(1) ),
SETSTMT ENTRY RETURNS( BIT(1) ),
SETITEM ENTRY RETURNS( BIT(1) ),
ACTSTMT ENTRY RETURNS( BIT(1) ),
TRACESTMT ENTRY RETURNS( BIT(1) ),
TITEM ENTRY RETURNS( BIT(1) ),
TIMING ENTRY RETURNS( BIT(1) ),
STAR CHAR(32) VARYING EXTERNAL;

FILE=6;
IF ~LIT('SIMULATE.*) THEN RETURN( FALSE );
TN=1; TNC='01';
IF ~SIMSTMT THEN CALL ERR('MISSING SIM STMT');
STMT=STMT+1;
L2: IF ~LIT(';;') THEN GO TO L1;
IF ~SIMSTMT THEN GO TO L1;
STMT=STMT+1;
GO TO L2;
L1: IF ~LIT('"END"') THEN CALL ERR('MISSING END');
RETURN( TRUE );

/* SIMSTMT ::= SETSTMT | ACTSTMT | TRACESTMT ; */
SIMSTMT:PROCEDURE RETURNS( BIT(1) );
   IF SETSTMT THEN RETURN( TRUE );
   IF ACTSTMT THEN RETURN( TRUE );
   IF TRACESTMT THEN RETURN( TRUE );
   RETURN( FALSE ); END SIMSTMT;

/* SETSTMT ::= 'SET' SETITEM $( ',', SETITEM ) ; */
SETSTMT:PROCEDURE RETURNS( BIT(1) );
   IF ~LIT('SET') THEN RETURN( FALSE );
   IF ~SETITEM THEN CALL ERR('MISSING SET ITEM');
L3: IF ~LIT(',') THEN RETURN( TRUE );
   IF ~SETITEM THEN CALL ERR('MISSING SETITEM');
   GO TO L3; END SETSTMT;

/* SETITEM ::= ID '.' ID.CPT('"INT"') | '<' INT.OPT('"INT"') >' */
/* =CONSTANT ; */
SETITEM:PROCEDURE RETURNS( BIT(1) );
   IF ~ID THEN RETURN( FALSE );
   ID1=STAR;
   IF ~LIT('.') THEN CALL ERR('MISSING .');
   IF ~ID THEN CALL ERR('MISSING IDEN');
   ID2=STAR;
   IF ~LIT('(') THEN GO TO L4;
   IF ~INT THEN CALL ERR('MISSING SUBSCRIPT');
   IF ~LIT(')') THEN CALL ERR('MISSING );');
   CALL OUT( ID1 || '.' || ID2 || '(' || STAR || ') ');
   GO TO L5;
L4: IF ~LIT('<') THEN DO; CALL OUT( ID1 || '_' || ID2 ); GO TO L5; END;
   IF ~INT THEN CALL ERR('MISSING SUBSCRIPT');
   CALL OUT( 'SUBSTR("" || ID1 || '_' || ID2 || '"' || STAR || '"' );
IF -LIT('::') THEN DO; CALL OUT('1'); GO TO L6; END;
CALL OUT('1-||STAR||');
IF -INT THEN CALL ERR('MISSING UPPER SUBSCRIPT');
CALL OUT('*+||STAR||*');
L6: IF -LIT('>') THEN CALL ERR('MISSING >');
L5: IF -LIT('==') THEN CALL ERR('MISSING =');
CALL OUT('==');
IF -CONSTANT THEN CALL ERR('MISSING CONSTANT');
CALL OUT('::*');
RETURN( TRUE ); END SETITEM;

/* ACTSTMT ::= 'activate' id '.' id $( id id ) ; */
ACTSTMT:PROCEDURE returns! bit(l) );
IF -LIT('ACTIVATE') THEN RETURN( FALSE );
CALL OUT('CALL $TART;');
IF -ID THEN CALL ERR('MISSING IDEN');
ID1=STAR;
IF -LIT('.') THEN CALL ERR('MISSING .');
IF -ID THEN CALL ERR('MISSING IDEN');
ID2=STAR;
CALL OUT('CALL $CH(#||ID1||=_||ID2||',0);');
L7: IF -LIT(',',') THEN RETURN( TRUE );
IF -ID THEN CALL ERR('MISSING IDEN');
ID1=STAR;
IF -LIT(',') THEN CALL ERR('MISSING ,');
IF -ID THEN CALL ERR('MISSING IDEN');
ID2=STAR;
CALL OUT('CALL $CH(#||ID1||=_||ID2||',0);');
GO TO L7; END ACTSTMT;

/* TRACESTMT ::= 'trace'titem $( titem ) */
TRACESTMT:PROCEDURE returns! bit!1);*
IF -LIT('trace') then return! false );
CALL NL;
CALL OUT('ON CONDITION($TRAC{E}')
CALL OUT('PUT FILE( SYSPRINT )EDIT');
/* tracestmt ::= 'trace'titem $! ',' titem ) */
/* .until. '(' timing ')' ; */
TRACESTMT:PROCEDURE returns! bit!1);*
IF -LIT('trace') then return! false );
CALL NL;
CALL OUT('ON CONDITION($TRAC{E}')
CALL OUT('PUT FILE( SYSPRINT )EDIT');
CALL OUT('("-"')(COL(13),A)');
IF ¬TITEM THEN CALL ERR('MISSING TRACE ITEM');
L8: IF ¬LIT('"') THEN GO TO L9;
CALL NL;
IF ¬TITEM THEN CALL ERR('MISSING TRACE ITEM');
GO TO L8;
L9: IF ¬LIT('UNTIL.') THEN CALL ERR('MISSING UNTIL');
CALL OUT(''); CALL NL;
CALL OUT('PUT FILE(SYSPRINT)');
CALL OUT('EDIT("INITIAL O-"')(SKIP,A)');
CALL OUT('SIGNAL CONDITION($TRACE)');
CALL LAB('#T'||TNC);
CALL OUT('IF ');
IF ¬LIT('(') THEN CALL ERR('MISSING (');
IF ¬TIMING THEN CALL ERR('MISSING TIMING INFO');
IF ¬LIT(')') THEN CALL ERR('MISSING )');
CALL OUT(' THEN DO;$BACK=#T'||TNC);
CALL OUT(';$DONE=#TD'||TNC'||GO TO $I;END;');
CALL LAB('#TD'||TNC);
TN=TN+1; PUT STRING(TNC)EDIT(TN)(P'99');
RETURN( TRUE ); END TRACESTMT;

/ * TITEM ::= ID*ID.OPT(('INT')*'|'<INT.OPT(':')INT')* ') * /
TITEM:PROCEDURE RETURNS( BIT(1) );
IF ¬ID THEN RETURN( FALSE );
ID1=STAR;
IF ¬LIT('.') THEN CALL ERR('MISSING .');
IF ¬ID THEN CALL ERR('MISSING IDEN');
ID2=STAR;
IF ¬LIT('(') THEN GO TO LB;
IF ¬INT THEN CALL ERR('MISSING SUBSCRIPT');
IF ¬LIT(')') THEN CALL ERR('MISSING )');
CALL OUT('""||ID1'||'_'||ID2'||'("'||STAR'||')='''', '''
ID1'||'_'||ID2'||'("'||STAR'||'),'''')(A,B,A)');
RETURN( TRUE );
LB: IF ¬LIT('<') THEN DO; CALL OUT('""||ID1'||'_'||ID2'||'"', ''')(A,B,A)');
RETURN( TRUE ); END;
IF ¬INT THEN CALL ERR('MISSING SUBSCRIPT ');
INT1=STAR;
IF ¬LIT('>') THEN DO; CALL OUT('**|ID1||_||ID2||<||INT1|
'>&='<',SUBSTR('**|ID1||_||ID2||','||
INT1||',,1),''):(A,B,A'); GO TO LC; END;
IF ¬INT THEN CALL ERR('MISSING UPPER SUBSCRIPT');
INT2=STAR;
CALL OUT('**|ID1||_||ID2||<||INT1||''||INT2|
'>&='<',SUBSTR('**|ID1||_||ID2||','||INT1||',1-''
INT1||'':''|INT2||'),''):(A,B,A');
LC: IF ¬LIT('>') THEN CALL ERR('MISSING >');
RETURN( TRUE ); END TITEM;

/* TIMING ::= 'TIME' ( '>=', '>' | '=' ) INT ; */
TIMING:PROCEDURE RETURNS( BIT(1) );
IF ¬LIT('TIME') THEN RETURN( FALSE );
CALL OUT('TIME');
IF LIT('>=') THEN DO; CALL OUT('<'); GO TO LD; END;
IF LIT('>') THEN DO; CALL OUT('<='); GO TO LD; END;
IF ¬LIT('='!) THEN CALL ERR('MISSING RELATIONAL');
CALL OUT('='!);
LD: IF ¬INT THEN CALL ERR('MISSING INTEGER');
CALL OUT(STAR);
RETURN( TRUE ); END TIMING;

END SIMULATE;
(SUBRG, STRG):
EXPR:PROCEDURE RETURNS( BIT(1) );
/*
   EXPR ::= TERM OPT( RELOP . TERM ) ; 
*/

DCL ERR ENTRY( CHAR(*) ),
   TRUE BIT(1) STATIC INIT('1'B),
   FALSE BIT(1) STATIC INIT('0'B),
   LIT ENTRY( CHAR(*) ) RETURNS( BIT(1) ),
   OUT ENTRY( CHAR(*) VARYING ),
   EXPR ENTRY RETURNS( BIT(1) ),
   RELOP ENTRY RETURNS( BIT(1) ),
   TERM ENTRY RETURNS( BIT(1) ),
   FACTOR ENTRY RETURNS( BIT(1) ),
   SEC ENTRY RETURNS( BIT(1) ),
   GROUP ENTRY RETURNS( BIT(1) ),
   PRIM ENTRY RETURNS( BIT(1) ),
   CONSTANT ENTRY RETURNS( BIT(1) ),
   REGREF ENTRY RETURNS( BIT(1) ),
   BIT ENTRY RETURNS( BIT(1) ),
   ID ENTRY RETURNS( BIT(1) ),
   STAR CHAR(32) VARYING EXTERNAL,
   PNAME CHAR(8) VARYING EXTERNAL,
   INT ENTRY RETURNS( BIT(1) );

IF ~TERM THEN RETURN( FALSE );
IF ~RELOP THEN RETURN( TRUE );
IF ~TERM THEN CALL ERR('MISSING SECOND TERM');
RETURN( TRUE );
/*
   RELOP ::= '.LT.'|'.LE.'|'.EQ.'|'.NE.'|'.GE.'|'.GT.' 
*/
RELOP:ENTRY RETURNS( BIT(1) );
IF LIT('.LT.') THEN DO; CALL OUT('<'); RETURN( TRUE ); END;
IF LIT('.LE.') THEN DO; CALL OUT('<='); RETURN( TRUE ); END;
IF LIT('.EQ.') THEN DO; CALL OUT('='); RETURN( TRUE ); END;
IF LIT('.NE.') THEN DO; CALL OUT('!='); RETURN( TRUE ); END;
IF LIT('.GE.') THEN DO; CALL OUT('>='); RETURN( TRUE ); END;
IF LIT('.GT.') THEN DO; CALL OUT('>' ); RETURN( TRUE ); END;
RETURN( FALSE );

/*
 * TERM ::= FACTOR $( '+' FACTOR ) ;
 * TERM:ENTRY RETURNS( BIT(1) ) RECURSIVE;
 * CALL OUT(">"); # L1: IF -LIT("+")) THEN DO; CALL OUT("'"); RETURN( TRUE ); END;
 * IF -FACTOR THEN RETURN( FALSE );
 *
 * L1: IF -LIT("+")) THEN DO; CALL OUT("'"); RETURN( TRUE ); END;
 * CALL OUT("'"); # L1: IF -FACTOR THEN CALL ERR("MISSING FACTOR");
 * GO TO L1;
 *
 * FACTOR ::= SEC $( '*' SEC ) ;
 * FACTOR:ENTRY RETURNS( BIT(1) ) RECURSIVE;
 * CALL OUT("*"); # L2: IF -LIT("*") THEN DO; CALL OUT("'"); RETURN( TRUE ); END;
 * IF -SEC THEN RETURN( FALSE );
 *
 * L2: IF -LIT("*") THEN DO; CALL OUT("'"); RETURN( TRUE ); END;
 * CALL OUT("*"); # L2: IF -SEC THEN CALL ERR("MISSING SECONDARY");
 * GO TO L2;
 *
 * SEC ::= '.NOT.' 'I' GROUP ')' | GROUP ;
 * SEC:ENTRY RETURNS( BIT(1) ) RECURSIVE;
 * IF -LIT(".NOT.")) THEN GO TO L3;
 * IF -LIT("'")) THEN CALL ERR("MISSING ("));
 * CALL OUT("-"); # L6: IF -LIT("'") THEN CALL ERR("MISSING GROUP");
 * IF -GROUP THEN CALL ERR("MISSING GROUP");
 * IF -LIT("'")) THEN CALL ERR("MISSING");
 * CALL OUT("'")); // L6: IF -LIT("'") THEN CALL OUT("'")); RETURN( TRUE );
 * RETURN( TRUE );
 *
 * L3: IF - GROUP THEN RETURN( FALSE );
 * RETURN( TRUE );
 *
 * GROUP ::= PRIM $( '|' PRIM ) ;
 * GROUP:ENTRY RETURNS( BIT(1) ) RECURSIVE;
 * CALL OUT("|"); # L6: IF -LIT("'") THEN CALL OUT("'")); RETURN( TRUE );
 * IF -PRIM THEN RETURN( FALSE );
 *
 * L6: IF -LIT("'") THEN DO; CALL OUT("'")); RETURN( TRUE ); END;
 */
CALL OUT('|||');
IF ¬PRIM THEN CALL ERR('MISSING PRIMARY');
GO TO L6;

/* PRIM ::= CONSTANT | REGREF | ('TERM') */
PRIM:ENTRY RETURNS(Bit(1)) RECURSIVE;
  IF CONSTANT THEN RETURN(TRUE);
  IF REGREF THEN RETURN(TRUE);
  IF ¬LIT('()') THEN RETURN(FALSE);
  CALL OUT('()');
  IF ¬TERM THEN CALL ERR('MISSING TERM WITHIN PARENS');
  IF ¬LIT(')') THEN CALL ERR('MISSING RIGHT PAREN FOLLOWING TERM');
  CALL OUT(')');
RETURN(TRUE);

/* CONSTANT ::= 'B' BIT ' ' */
CONSTANT:ENTRY RETURNS(Bit(1));
  IF ¬LIT('B') THEN RETURN(FALSE);
  IF ¬BIT THEN CALL ERR('MISSING BIT STRING');
  IF ¬LIT(' ') THEN CALL ERR('MISSING SPACE');
  CALL OUT('B');
RETURN(TRUE);

/* REGREF ::= ID .OPT('<' INT .OPT(':' INT )'>' ) */
REGREF:ENTRY RETURNS(Bit(1));
  IF ¬ID THEN RETURN(FALSE);
  IF ¬LIT('<') THEN DO; CALL OUT(PNAME||'_'||STAR);
          RETURN(FALSE); END;
  CALL OUT('SUBSTR'||PNAME||'_'||STAR||',');
  IF ¬INT THEN CALL ERR('MISSING LOWER SUBSCRIPT');
  CALL OUT(STAR||');
  IF ¬LIT('>') THEN DO; CALL OUT('1'); GO TO L7; END;
  CALL OUT('1-||STAR||');
  IF ¬INT THEN CALL ERR('MISSING UPPER SUBSCRIPT');
  CALL OUT(STAR);
L7: IF ¬LIT('>') THEN CALL ERR('MISSING RIGHT BRACKET >');
CALL OUT(')');
RETURN( TRUE );
ENDExpr;
{SUBRG, STRG}:
LIT: PROCEDURE (STR) RETURNS (BIT (1)) ;
   DCL CARD CHAR (80) EXTERNAL,
       C (80) CHAR (1 ) DEF CARD,  
       I FIXED BIN EXTERNAL,
       STR CHAR (*),
       STAR CHAR (32) VARYING EXTERNAL,
   LIT ENTRY (CHAR(*)) RETURNS (BIT (1)),
   ID ENTRY RETURNS (BIT (1)),
   INT ENTRY RETURNS (BIT (1)),
   BIT ENTRY RETURNS (BIT (1)),
     TRUE BIT (1) STATIC INIT ('1'B),
     FALSE BIT (1) STATIC INIT ('0'B),
     J FIXED BIN STATIC;
     J = LENGTH (STR);
     IF I > 80 THEN SIGNAL CONDITION (EOC);
     DO WHILE ( C (I) = ' ' ); I = I + 1;
       IF I > 80 THEN SIGNAL CONDITION (EOC); END;
       IF I + J > 81 THEN RETURN ( FALSE );
       IF SUBSTR ( CARD, I, J ) = STR THEN DO; I = I + J; RETURN ( TRUE ); END;
       RETURN ( FALSE );
     ID: ENTRY RETURNS (BIT (1));
     IF I > 80 THEN SIGNAL CONDITION (EOC);
     DO WHILE ( C (I) = ' ' ); I = I + 1;
       IF I > 80 THEN SIGNAL CONDITION (EOC); END;
       IF ( C (I) >= ' A ' ) AND ( C (I) <= ' Z ' ) THEN J = I;
         ELSE RETURN ( FALSE );
     DO WHILE ( C (I) >= ' A ' ); I = I + 1; END;
     STAR = SUBSTR ( CARD, J, I - J ); RETURN ( TRUE );
     INT: ENTRY RETURNS (BIT (1));
     IF I > 80 THEN SIGNAL CONDITION (EOC);
     DO WHILE ( C (I) = ' ' ); I = I + 1;
       IF I > 80 THEN SIGNAL CONDITION (EOC); END;
       IF C (I) >= '0' THEN J = I;
ELSE RETURN( FALSE );
DO WHILE( C(I)>'0' ); I=I+1; END;
STAR=SUBSTR(CARD,J,I-J); RETURN( TRUE );

BIT:ENTRY RETURNS( BIT(I) );
IF I>80 THEN SIGNAL CONDITION(EOC);
DO WHILE( C(I)=' ' ); I=I+1;
   IF I>80 THEN SIGNAL CONDITION(EOC); END;
IF( C(I)='0' || (C(I)='1' ) THEN J=I;
   ELSE RETURN( FALSE );
   DO WHILE( (C(I)='0' )|| (C(I)='1' ) ); I=I+1; END;
   STAR=SUBSTR(CARD,J,I-J); RETURN( TRUE ); END BIT;
PROCEDURE(STR);

/* OUTPUT TO BUFFER */

/* FILE NUMBERS */
1 STATES */
2 TERMINALS */
3 CONDITIONS */
4 CONTROL */
5 DECLARES */
6 SIMULATE */

DCL STR CHAR(*) VARYING,
LA(6) LABEL, LB(6) LABEL, LC(6) LABEL,
LINE CHAR(80) EXTERNAL,
L FIXED BIN EXTERNAL,
FILE FIXED BIN EXTERNAL,
OUT ENTRY(CHAR(*) VARYING),
LAB ENTRY(CHAR(*) VARYING),
NL ENTRY,
PRINT FILE PRINT,
J FIXED BIN STATIC;

J=LENGTH(STR);
IF L+J>72 THEN DO;
  PUT FILE( PRINT)EDIT(LINE)(SKIP,COL(40),A);
    GO TO LA(FILE);
  LA(1):WRITE FILE(STATS) FROM(LINE); GO TO LOUT;
  LA(2):WRITE FILE(TEMS) FROM(LINE); GO TO LOUT;
  LA(3):WRITE FILE(CONDS) FROM(LINE); GO TO LOUT;
  LA(4):WRITE FILE(INITS) FROM(LINE); GO TO LOUT;
  LA(5):WRITE FILE(DECLS) FROM(LINE); GO TO LOUT;
  LA(6):WRITE FILE(SIMUL) FROM(LINE);
LOUT:
  L=8; LINE=(80)' '; END;
  SUBSTR(LINE,L,J)=STR; L=L+J; RETURN;

LAB:ENTRY(STR); /* OUTPUT LABEL */
J=LENGTH(STR);
  PUT FILE( PRINT)EDIT(LINE)(SKIP,COL(40),A);
  GO TO LB(FILE);
LB(1): WRITE FILE(STATS) FROM(LINE); GO TO LLAB;
LB(2): WRITE FILE(TERMS) FROM(LINE); GO TO LLAB;
LB(3): WRITE FILE(CONDS) FROM(LINE); GO TO LLAB;
LB(4): WRITE FILE(INITS) FROM(LINE); GO TO LLAB;
LB(5): WRITE FILE(DECLS) FROM(LINE); GO TO LLAB;
LB(6): WRITE FILE(SIMUL) FROM(LINE);
LLAB:  L=2; LINE=(80)' ';
       SUBSTR(LINE,L,J)=STR; L=L+J+1; SUBSTR(LINE,L-1,1)=':'; RETURN;

NL:ENTRY; /* OUTPUT NEW LINE */
       PUT FILE( PRINT)EDIT(LINE)(SKIP,COL(40),A);
       GO TO LC(FILE);
LC(1): WRITE FILE(STATS) FROM(LINE); GO TO LNL;
LC(2): WRITE FILE(TERMS) FROM(LINE); GO TO LNL;
LC(3): WRITE FILE(CONDS) FROM(LINE); GO TO LNL;
LC(4): WRITE FILE(INITS) FROM(LINE); GO TO LNL;
LC(5): WRITE FILE(DECLS) FROM(LINE); GO TO LNL;
LC(6): WRITE FILE(SIMUL) FROM(LINE);
LNL:  L=8; LINE=(80)' ' ; END OUT;
APPENDIX C:

SIMULATION ROUTINES
(SUBRG, STRG):
$CH:PROCEDURE(L,T);

DCL L LABEL, T FIXED BIN;
DCL $LOT(0:$MS) FIXED BIN EXTERNAL CONTROLLED,
     $A($MA) LABEL EXTERNAL CONTROLLED,
     $NA($MA) FIXED BIN EXTERNAL CONTROLLED,
     $MS FIXED BIN EXTERNAL,
     $FA FIXED BIN EXTERNAL,
     TIME FIXED BIN EXTERNAL,
     $RT FIXED BIN EXTERNAL,
     $NS FIXED BIN EXTERNAL,
     $MA FIXED BIN EXTERNAL,
     $E FIXED BIN EXTERNAL,
     $FE FIXED BIN EXTERNAL,
     $ME FIXED BIN EXTERNAL,
     $LAT($ME) LABEL EXTERNAL CONTROLLED,
     $AT($ME) FIXED BIN EXTERNAL CONTROLLED,
     $C FIXED BIN EXTERNAL,
     $NAT($ME) FIXED BIN EXTERNAL CONTROLLED;

DCL ( I,J,K,M ) FIXED BIN STATIC;

IF T>= $NS THEN DO; /* ADD TO EXTRA LIST */
   IF $FE=0 THEN DO; PUT FILE(SYSPRINT) EDIT('EXTRA LIST FULL')(SKIP,A);
                STOP; END;
   I=$FE; $FE=$NAT(I); /* REMOVE FROM FREE LIST */
   $NAT(I)=$E; $E=I; /* ADD TO EXTRA LIST */
   $LAT($E)=L; $AT($E)=TIME+T; /* PUT IN DATA */
   RETURN; END;

/* ADD TO TIME SLOT */
I=MOD(TIME+T,$NS); /* FIND WHICH SLOT */
IF $FA=0 THEN DO; PUT FILE(SYSPRINT)EDIT('TIME LIST FULL') (SKIP,A); STOP; END;
J=$FA; $FA=$NA($FA); /* UPDATE FREE LIST */
$A(J)=L; \$NA(J)=$LOT(I); \$LOT(I)=J; /* PUT I */
$C=$C+1;
RETURN;

$INCR:ENTRY;
TIME=TIME+1; \$RT=$RT+1; /* INCREMENT TIME */
IF \$RT>$MS THEN DO; /* LOOK IN EXTRA LIST FOR ITEMS TO BE ADDED */
I=$E; \$E=0; /* I NOW POINTS TO EXTRA LIST */
NEXT:
IF I>0 THEN DO; J=I; I=$NAT(J); \$AT(J)=\$AT(J)-\$NS;
/* J POINTS TO ITEM BEING REMOVED FROM LIST */
/* I POINTS TO REMAINDER OF LIST */
/* ITEM VALUE IS LESS TIME TO GO THROUGH SLOTS */
IF \$AT(J)<=\$MS THEN DO; /* PUT IN TIME SLOT */
IF \$FA=0 THEN DO; PUT FILE(SYSPRINT)
EDIT('NO TIME SLOTS AVAL')
(SKIP,A); STOP; END;
K=$FA; \$FA=$NA(K); /* FREE ITEM FOR TIME */
$A(K)=$LAT(J); /* PUT LABEL IN IT */
M=$AT(J); \$NA(K)=$LOT(M); \$LOT(M)=K;
/* FREE ITEM THAT WAS IN EXTRA LIST */
$NAT(J)=\$FE; \$FE=J; END;
ELSE DO; /* PUT BACK INTO ACTIVE */
$NAT(J)=\$E; \$E=J; END;
GO TO NEXT; /* GO BACK AND SEE IF MORE IN LIST */
END;
$RT=0; END;
PUT FILE(SYSPRINT)EDIT(TIME,‵−−′)(SKIP,F(10),A);
RETURN;
END $CH;
(SUBRG, STRG): $ETUP: PROCEDURE;

DCL $LOT(0:$MS) FIXED BIN EXTERNAL CONTROLLED,
    $A($MA) LABEL EXTERNAL CONTROLLED,
    $NA($MA) FIXED BIN EXTERNAL CONTROLLED,
    $MS FIXED BIN EXTERNAL,
    $FA FIXED BIN EXTERNAL,
    TIME FIXED BIN EXTERNAL,
    $RT FIXED BIN EXTERNAL,
    $NS FIXED BIN EXTERNAL,
    $MA FIXED BIN EXTERNAL,
    $E FIXED BIN EXTERNAL,
    $FE FIXED BIN EXTERNAL,
    $ME FIXED BIN EXTERNAL,
    $LAT($ME) LABEL EXTERNAL CONTROLLED,
    $AT($ME) FIXED BIN EXTERNAL CONTROLLED,
    $C FIXED BIN EXTERNAL,
    $NAT($ME) FIXED BIN EXTERNAL CONTROLLED;

/* SET UP VALUES OF ARRAYS */
$MS=99; $NS=$MS+1; $MA=25; $ME=20;
ALLOCATE $LOT, $A, $NA, $LAT, $AT, $NAT;
RETURN;

$START: ENTRY;

/* INITIALIZE THESE ARRAYS */
$LOT=0; $E, TIME, $RT=0; $FA, $FE=1; $C=0;
DO I=1 TO $MA-1; $NA(I)=I+1; END; $NA($MA)=0;
DO I=1 TO $ME-1; $NAT(I)=I+1; END; $NAT($ME)=0;
RETURN;
END $ETUP;
APPENDIX D:

$RUN SOURCE
(SUBRG, STRG):

$RUN:PROCEDURE OPTIONS(MAIN);

DCL $LOTQ:$MS) FIXED BIN EXTERNAL CONTROLLED,
   $A($MA) LABEL EXTERNAL CONTROLLED,
   $NA($MA) FIXED BIN EXTERNAL CONTROLLED,
   $MS FIXED BIN EXTERNAL,
   $FA FIXED BIN EXTERNAL,
   TIME FIXED BIN EXTERNAL,
   $RT FIXED BIN EXTERNAL,
   $NS FIXED BIN EXTERNAL,
   $MA FIXED BIN EXTERNAL,
   $E FIXED BIN EXTERNAL,
   $FE FIXED BIN EXTERNAL,
   $ME FIXED BIN EXTERNAL,
   $LAT($ME) LABEL EXTERNAL CONTROLLED,
   $AT($ME) FIXED BIN EXTERNAL CONTROLLED,
   $NAT($ME) FIXED BIN EXTERNAL CONTROLLED,
   $C FIXED BIN EXTERNAL,
   $DONE LABEL,
   $BACK LABEL,
   $ETUP ENTRY,
   $START ENTRY,
   $CH ENTRY( LABEL, FIXED BIN ),
   $INCR ENTRY,
   $T FIXED BIN STATIC,
   $N FIXED BIN STATIC,
   $L LABEL;

CALL $ETUP;
%INCLUDE MPSLIB(INITS);
%INCLUDE MPSLIB(DECLS);

GO TO $BEGIN;
$\text{SIM:}$

$\text{CONTROL: }$ $N = \text{LCT}(RT);$ 
\hspace{1em} \text{IF } N > 0 \text{ THEN DO}; $L = A(N);$ $\text{LOT}(RT) = \text{NA}(N);$ 
\hspace{1em} $C = C - 1;$ 
\hspace{1em} $\text{NA}(N) = FA;$ $FA = N;$ \text{GO TO } L; \text{ END;}
\hspace{1em}$
\hspace{1em} $\%\text{INCLUDE MPSLIB(TERMS);}$
\hspace{1em} $\%\text{INCLUDE MPSLIB(CONDS);}$

$\text{CALL } $\text{INCR;}$
\hspace{1em} $\text{IF } C > 0 \text{ THEN GO TO } $\text{BACK;} \text{ ELSE GO TO } $\text{DONE;}$
\hspace{1em}$
\hspace{1em} $\%\text{INCLUDE MPSLIB(STATS);}$

$\text{BEGIN:}$
\hspace{1em} $\%\text{INCLUDE MPSLIB(SIMUL);}$

$\text{END } $\text{RUN;
APPENDIX E:

JOB CONTROL
//E205ELL JOB 'I4600,TIME=9,SIZE=160K,CARDS=2000', ELLIOTT
//STEP1 EXEC P11F,TIME=P11L=1,REGION=P11L=128K,
//      PARM=LKED='LIST,LET,OVLY',
//      TIME.GO=1,REGION.GO=160K,LINES=9,
//      COND.GO=((7,LT,LKED),(9,LT,P11L))
//P11L.SYSIN DD *

*MPS SOURCE *

//LKED.SYSLMOD DD DSN=&GOSET(GO),UNIT=SPOOL,DISP=(MOD,PASS),
//      SPACE=(TRK,(25,5,1))
//LKED.SYSIN DD *

* OBJECT DECKS OF TRANSLATOR *

INSERT MPS,**MPSA
INSERT CARD,LINE,EOC,L,I,STMT,FILE,STAR,PNAME,SNAME
INSERT LIT,**LITA
INSERT OUT,**OUTA
INSERT EXPR,**EXPRA
OVERLAY ALPHA
INSERT SIMULATE,**SIMULATEA
OVERLAY ALPHA
INSERT SYSTEM,**SYSTEMA
INSERT PROC,**PROCA
INSERT TRANSFER,**TRANSFERA
OVERLAY BETA
INSERT FACDCL,**FACDCLA
OVERLAY BETA
INSERT STATE,**STATEA

//GO.PRINT DD SYSOUT=A,SPACE=(CYL,(4)),
//DCB=(RECFM=VBA,LRECL=137,BLKSIZE=3292)
//GO.INITS DD DCB=(LRECL=80,RECFM=FB,BLKSIZE=400),DISP=(NEW,PASS),
//UNIT=SPOOL,SPACE=(TRK,(2,1)),DSNAME=&INITS
//GO.DECLS DD DCB=(LRECL=80,RECFM=FB,BLKSIZE=400),DISP=(NEW,PASS),
//UNIT=SPOOL,SPACE=(TRK,(2,1)),DSNAME=&DECLS
//GO.TERMS DD DCB=(LRECL=80,RECFM=FB,BLKSIZE=400),DISP=(NEW,PASS),
//GO. CONDS DD DCB=(LRECL=80,RECFM=FB,BLKSIZ=400),DISP=(NEW,PASS), X
//UNIT=SPOOL,SPACE=(TRK,(2,1)),DSNAME=&amp;TERMS
//GO. STATS DD DCB=(LRECL=80,RECFM=FB,BLKSIZ=400),DISP=(NEW,PASS), X
//UNIT=SPOOL,SPACE=(TRK,(2,1)),DSNAME=&amp;CONS
//GO. SIMUL DD DCB=(LRECL=80,RECFM=FB,BLKSIZ=400),DISP=(NEW,PASS), X
//UNIT=SPOOL,SPACE=(TRK,(2,1)),DSNAME=&amp;SIMUL
//GO. SYSIN DD *

* MULTIPROCESSOR DESCRIPTION *

//STEP2 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSUT2 DD DSNAME=&&MPSLIB,DCB=(RECFM=FB,LRECL=80,BLKSIZ=400), X
//UNIT=SPOOL,DISP=(NEW,PASS),SPACE=(TRK,(10,5,2))
//SYSIN DD *

GENERATE MAXNAME=6,MAXG=6
MEMBER NAME=INITS
RECORD IDENT=(7,'/*END*/',8)
MEMBER NAME=DECLS
RECORD IDENT=(7,'/*END*/',8)
MEMBER NAME=TERMS
RECORD IDENT=(7,'/*END*/',8)
MEMBER NAME=CONDS
RECORD IDENT=(7,'/*END*/',8)
MEMBER NAME=STATS
RECORD IDENT=(7,'/*END*/',8)
MEMBER NAME=SIMUL
RECORD IDENT=(7,'/*END*/',8)

//SYSUT1 DD DCB=(LRECL=80,RECFM=FB,BLKSIZ=400),DISP=(OLD,DELETE), X
//UNIT=SPOOL,DSNAME=&amp;INITS
//DD DCB=(LRECL=80,RECFM=FB,BLKSIZ=400),DISP=(OLD,DELETE), X
//UNIT=SPOOL,DSNAME=&amp;DECLS
//DD DCB=(LRECL=80,RECFM=FB,BLKSIZ=400),DISP=(OLD,DELETE), X
//UNIT=SPOOL,DSNAME=&amp;TERMS
//DD DCB=(LRECL=80,RECFM=FB,BLKSIZ=400),DISP=(OLD,DELETE), X
//UNIT=SPOOL,DSNAME=&amp;CONDS
// DD DCB=(LRECL=80,RECFM=FB,BLKSIZE=400),DISP=(OLD,DELETE), UNIT=SPOOL,DSNAME=&&STATS
// DD DCB=(LRECL=80,RECFM=FB,BLKSIZE=400),DISP=(OLD,DELETE), UNIT=SPOOL,DSNAME=&&SIMUL
//STEP3 EXEC PLIL,PARM.PLIL='LOAD,NODECK,MACRO',TIME.PLIL=3, REGION.PLIL=128K,REGION.GO=128K,LINES=6
//PLIL.MPSLIB DD DSNAME=&&MPSLIB,DCB=(LRECL=80,BLKSIZE=400,RECFM=FB), UNIT=SPOOL,DISP=(OLD,DELETE)
//PLIL.SYSIN DD *

* $RUN SOURCE *

//LKED.SYSLMOD DD DSN=GOSET(TS),UNIT=SPOOL,DISP=(MOD,PASS), SPACE=(TRK,(99,9,1))
//LKED.SYSIN DD *

* OBJECT DECKS OF RUN TIME ROUTINES *
APPENDIX F:

SINGLE TEL_COMP DESCRIPTION
*SYSTEM.
* 
* BUS SR;
* 
* TELCOMP HYPO COMPUTER
* 
* TEL PROCESS CR(SR);
* 
REG LCR<7>, ACR<10>, AC1<9>, IR<3>;
DEF OP<3>:=XR<1:3>, ADDR<6>:=XR<4:9>,
    LCOV:=LCR<1>, LC<6>:=LCR<2:7>,
    OV:=ACR<1>, AC2<9>:=ACR<2:10>;
ADDER ADDIT(ACR=XR•ADD•AC1), INCR( LCR=LC•ADD•B'0000001' );
RAM XR<9>=M( MAR<6> );
*
FETCH: MAR=LC$l; READ(M)$50 ; ( IR=OP$1•MAR=ADDR$1•ADD• INCR )$10 ; THEN.
   (IR.EQ. B'000')ADD > (IR.EQ. B'001')SUB > (IR.EQ. B'010')SRO >
   (IR.EQ. B'011')TRU > (IR.EQ. B'100')TRN > (IR.EQ. B'101')STA >
   (IR.EQ. B'110')CLA > (IR.EQ. B'111')STP ;
*
ADD:READ(M)$50 ; ADD( ADDIT )$10 ; AC1=AC2$1. THEN. goto. FETCH;
*
SUB:READ(M)$50 ; XR=.NOT.(XR)$1; ADD( ADDIT )$10;
   ( XR=B'000000001'$1,AC1=AC2$1 ); ADD( ADDIT )$10; AC1=AC2$1
   .THEN. goto. FETCH;
*
SRO: XR=B'000000000'$1; ADD( ADDIT )$10; AC1=B'0'|AC2<1:8>$2
   .THEN. goto. FETCH;
*
TRU: LC=MAR$1 . THEN. goto. FETCH;
*
TRN: .THEN. (AC1<1>)TRN1 > (.NOT.(AC1<1>))FETCH ;
TRN1: LC=MAR$1 . THEN. goto. FETCH;
*
STA:XR=AC1$1; WRITE(M)$50 . THEN. goto. FETCH;
*
CLA: AC1=B'000000000' $1 .THEN. .GOTO. ADD ;
*
STP: SR=B'0'$1 .THEN. ( SR ) FETCH
*
*END.*
*
*SIMULATE.*
*
*ADDR OP ADDR CODE
* 0 CLA 8 110 001000
* 1 SRO 0 010 000000
* 2 ADD 9 000 001001
* 3 SUB 10 001 001010
* 4 TRN 7 100 000111
* 5 STA 11 101 001011
* 6 STP 0 111 000000
* 7 TRU 5 011 000101
* 8 DC 8 000 001000
* 9 DC 4 000 000100
* 10 DC 9 000 001001
* 11 DS 0 *** *******
*
SET TEL.SR=B'1', TEL.LC=B'000000',
TEL.AC1=B'111111111', TEL.AC2=B'111111111', TEL.MAR=B'1111111',
TEL.XR=B'1111111111', TEL.IR=B'111',
TEL.M(0)=B'110001000', TEL.M(1)=B'010000000', TEL.M(2)=B'0000001001',
TEL.M(3)=B'0001001010', TEL.M(4)=B'1000001111', TEL.M(5)=B'101001011',
TEL.M(6)=B'1110000000', TEL.M(7)=B'0110001011', TEL.M(8)=B'000001000',
TEL.M(9)=B'0000000100', TEL.M(10)=B'000001001', TEL.M(11)=B'1111111111';
*
ACTIVATE TEL.FETCH;
*
TRACE TEL.LC, TEL.AC1, TEL.AC2
 .UNTIL.( TIME > 800 );
* * *

SET TEL.SR=B'1', TEL.LC=B'000000',
    TEL.AC1=B'11111111', TEL.AC2=B'11111111', TEL.MAR=B'111111',
    TEL.XR=B'11111111', TEL.IR=B'111',
    TEL.M(0)=B'11000100', TEL.M(1)=B'01000000', TEL.M(2)=B'000001001',
    TEL.M(3)=B'00100100', TEL.M(4)=B'10000111', TEL.M(5)=B'10100101',
    TEL.M(6)=B'111100000', TEL.M(7)=B'011000101', TEL.M(8)=B'000000100',
    TEL.M(9)=B'000000100', TEL.M(10)=B'000000100', TEL.M(11)=B'11111111';

* ACTIVATE TEL.FETCH;
* TRACE TEL.XR, TEL.IR, TEL.MAR
    .UNTIL ( TIME > 800 );
*
* END.
APPENDIX G:

SINGLE TEL_COMP SIMULATION
INITIAL
0---TEL_LC=<000000>, TEL_AC1=<11111111>, TEL_AC2=<11111111>,
IN STATE TEL_FETCH
---TEL_LC=<000000>, TEL_AC1=<11111111>, TEL_AC2=<11111111>,
1---TEL_LC=<000000>, TEL_AC1=<11111111>, TEL_AC2=<11111111>,
2---
3---
4---
5---
6---
7---
8---
9---
10---
11---
12---
13---
14---
15---
16---
17---
18---
19---
20---
21---
22---
23---
24---
25---
26---
27---
28---
29---
30---
31---
32---
33---
34---
TEL_LC = <000001>, TEL_AC1=<111111111>, TEL_AC2=<111111111>

IN STATE TEL_CLA
-TEL_LC=<000001>, TEL_AC1=<000000000>, TEL_AC2=<111111111>

IN STATE TEL_ADD
-TEL_LC=<000001>, TEL_AC1=<000000000>, TEL_AC2=<111111111>
103--
104--
105--
106--
107--
108--
109--
110--
111--
112---TEL_LC=<000001>,TEL_AC1=<00000000>,TEL_AC2=<000001000>,
113--
114--
115--
116--
117--
118--
119--
120--
121--
122---TEL_LC=<000001>,TEL_AC1=<00001000>,TEL_AC2=<000001000>,
123--
IN STATE TEL_FETCH
   -TEL_LC=<000001>,TEL_AC1=<000001000>,TEL_AC2=<000001000>,
124---TEL_LC=<000001>,TEL_AC1=<00001000>,TEL_AC2=<000001000>,
125--
126--
127--
128--
129--
130--
131--
132--
133--
134--
135--
136--
137--
IN STATE TEL_SRO

-TEL_LC=<000010>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
-TEL_LC=<000010>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
175--
176--
177--
178--
179--
180--
181--
182--
183--
184--
185--
186--
187--
188--
189--
190--
191--
192--
193--
194--
195--
196--
197--

IN STATE TEL_FETCH

-TEL_LC=<000010>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
198--TEL_LC=<000010>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
199--
200--
201--
202--
203--
204--
205--
IN STATE TEL_ADD

-TEL_LC=<000011>, TEL_AC1=<000000100>, TEL_AC2=<000001000>,
-TEL_LC=<000011>, TEL_AC1=<000000100>, TEL_AC2=<000001000>,
-TEL_LC=<000011>, TEL_AC1=<000000100>, TEL_AC2=<000001000>,

TEL_LC=<000011>, TEL_AC1=<000000100>, TEL_AC2=<000001000>,
IN STATE TEL_FETCH
   TEL_LC=<000011>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,

   TEL_LC=<000011>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>

IN STATE TEL_SUB
   -TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,

TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
   -TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
   -TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,

TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
   -TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,

TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
   -TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,

TEL_LC=<000100>, TEL_AC1=<000001000>, TEL_AC2=<000001000>,
IN STATE TEL_FETCH
-TEL_LC=<000100>, TEL_AC1=<11111111>, TEL_AC2=<11111111>,
454-- TEL_LC=<000100>, TEL_AC1=<11111111>, TEL_AC2=<11111111>,
455--
456--
457--
458--
459--
460--
461--
462--
463--
464--
465--
466--
467--
468--
469--
470--
471--
472--
473--
474--
475--
476--
477--
478--
479--
480--
481--
482--
483--
484--
485--
486--
487--
488--
TEL_LC = <000101>, TEL_AC1 = <111111111>, TEL_AC2 = <111111111>,
-TEL_LC = <000101>, TEL_AC1 = <111111111>, TEL_AC2 = <111111111>,
-TEL_LC = <000101>, TEL_AC1 = <111111111>, TEL_AC2 = <111111111>,

IN STATE TEL_TRN
IN STATE TEL_TRN1
-TEL_LC = <000111>, TEL_AC1 = <111111111>, TEL_AC2 = <111111111>,

IN STATE TEL_FETCH
-TEL_LC = <000111>, TEL_AC1 = <111111111>, TEL_AC2 = <111111111>,
-TEL_LC = <000111>, TEL_AC1 = <111111111>, TEL_AC2 = <111111111>,
-TEL_LC = <000111>, TEL_AC1 = <111111111>, TEL_AC2 = <111111111>,

517--
518--
IN STATE TEL_TRU
  -TEL_LC=<001000>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
  -TEL_LC=<001000>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
  -TEL_LC=<001000>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

IN STATE TEL_FETCH
  -TEL_LC=<000101>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
  -TEL_LC=<000101>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
  -TEL_LC=<000101>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
  -TEL_LC=<000101>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

141
TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
-TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
-TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

IN STATE TEL_STA
-TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
-TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
-TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

IN STATE TEL_FETCH

-TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
-TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,
-TEL_LC=<000110>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

IN STATE TEL_STP

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>,

-TEL_LC=<000111>, TEL_AC1=<111111111>, TEL_AC2=<111111111>. 
INITIAL 0 --- TEL_XR = <111111111>, TEL_IR = <111>, TEL_MAR = <111111>,
IN STATE TEL_FETCH
   - TEL_XR = <111111111>, TEL_IR = <111>, TEL_MAR = <000000>,
   1 --- TEL_XR = <110001000>, TEL_IR = <111>, TEL_MAR = <000000>,
   2 ---
   3 ---
   4 ---
   5 ---
   6 ---
   7 ---
   8 ---
   9 ---
  10 ---
  11 ---
  12 ---
  13 ---
  14 ---
  15 ---
  16 ---
  17 ---
  18 ---
  19 ---
  20 ---
  21 ---
  22 ---
  23 ---
  24 ---
  25 ---
  26 ---
  27 ---
  28 ---
  29 ---
  30 ---
  31 ---
IN STATE TEL_CLA
  -TEL_XR=<110001000>,TEL_IR=<110>,TEL_MAR=<001000>,

IN STATE TEL_ADD
  -TEL_XR=<000001000>,TEL_IR=<110>,TEL_MAR=<001000>,

-TEL_XR=<110001000>,TEL_IR=<111>,TEL_MAR=<000000>,
  -TEL_XR=<110001000>,TEL_IR=<111>,TEL_MAR=<001000>,
  -TEL_XR=<110001000>,TEL_IR=<110>,TEL_MAR=<001000>,

149
STATE TEL.FETCH

IN STATE TEL.FETCH

-TEL_XR=<000001000>, TEL_IR=<110>, TEL_MAR=<001000>,

124---TEL_XR=<000001000>, TEL_IR=<110>, TEL_MAR=<001000>,

124---TEL_XR=<010000000>, TEL_IR=<110>, TEL_MAR=<000001>,

125--
126--
127--
128--
129--
130--
131--
132--
133--
134--
IN STATE TEL_SRO
  -TEL_XR=<000000000>, TEL_IR=<010>, TEL_MAR=<000000>,

IN STATE TEL_FETCH
  -TEL_XR=<000000000>, TEL_IR=<010>, TEL_MAR=<000010>,

172--
173--
174-- TEL_XR=<010000000>, TEL_IR=<110>, TEL_MAR=<000001>,
     -TEL_XR=<010000000>, TEL_IR=<110>, TEL_MAR=<000000>,
     -TEL_XR=<010000000>, TEL_IR=<010>, TEL_MAR=<000000>,

175--
176--
177--
178--
179--
180--
181--
182--
183--
184--

185-- TEL_XR=<000000000>, TEL_IR=<010>, TEL_MAR=<000000>,

186--
187--
188--
189--
190--
191--
192--
193--
194--
195-- TEL_XR=<000000000>, TEL_IR=<010>, TEL_MAR=<000000>,

196--
197--
IN STATE TEL_ADD
  TEL_XR=<000000100>, TEL_IR=<000>, TEL_MAR=<001001>,
  TEL_XR=<000000100>, TEL_IR=<000>, TEL_MAR=<001001>,

STATE TEL. Add
  TEL_XR=<000000100>, TEL_IR=<000>, TEL_MAR=<001001>,
  TEL_XR=<000000100>, TEL_IR=<000>, TEL_MAR=<001001>,
  TEL_XR=<000000100>, TEL_IR=<000>, TEL_MAR=<001001>,

TEL_XR=<000001001>, TEL_IR=<010>, TEL_MAR=<000010>,
  TEL_XR=<000001001>, TEL_IR=<010>, TEL_MAR=<000010>,
  TEL_XR=<000001001>, TEL_IR=<000>, TEL_MAR=<001001>,
IN STATE TEL_FETCH
    TEL_XR=<000000100>, TEL_IR=<000>, TEL_MAR=<001001>,
319--
    TEL_XR=<001001010>, TEL_IR=<000>,
320--
    TEL_XR=<000000100>, TEL_IR=<000>, TEL_MAR=<000011>,
321--
    TEL_XR=<001001010>, TEL_IR=<000>, TEL_MAR=<000011>,
322--

345---
346---
347---
348---
349---
350---
351---
352---
353---
354---
355---
356---
357---
358---
359---
360---
361---
362---
363---
364---
365---
366---
367---
368---
369---
370---TEL_XR=<001001010>, TEL_IR=<000>, TEL_MAR=<000011>,
       -TEL_XR=<001001010>, TEL_IR=<000>, TEL_MAR=<001010>,
       -TEL_XR=<001001010>, TEL_IR=<000>, TEL_MAR=<001010>,
371---
372---
373---
374---
375---
376---
377---
378---
379---
IN STATE TEL_SUB
-TEL_XR=<00001001>, TEL_IR=<001>, TEL_MAR=<001010>,
TEL_XR=11110110, TEL_IR=001, TEL_MAR=001010,
TEL_XR=11110110, TEL_IR=001, TEL_MAR=001010,
IN STATE TEL_FETCH

- TEL_XR=<000000001>, TEL_IR=<001>, TEL_MAR=<000100>,
- TEL_XR=<100000111>, TEL_IR=<001>, TEL_MAR=<000100>,
- TEL_XR=<000000001>, TEL_IR=<001>, TEL_MAR=<001010>,

TEL_XR=<100000111>, TEL_IR=<001>, TEL_MAR=<000100>,
-TEL_XR=<100000111>, TEL_IR=<001>, TEL_MAR=<000111>,
-TEL_XR=<100000111>, TEL_IR=<100>, TEL_MAR=<000111>,

IN STATE TEL_TRN
IN STATE TEL_TRN1
-TEL_XR=<100000111>, TEL_IR=<100>, TEL_MAR=<000111>,

IN STATE TEL_FETCH
-TEL_XR=<100000111>, TEL_IR=<100>, TEL_MAR=<000111>,

IN STATE TEL_FETCH
516---TEL_XR=<011000101>, TEL_IR=<100>, TEL_MAR=<000111>,
517--
518--
519--
520--
521--
522--
523--
524--
525--
526--
527--
528--
529--
530--
531--
532--
533--
534--
535--
536--
537--
538--
539--
540--
541--
542--
543--
544--
545--
546--
547--
548--
549--
550--
551--
552--
TEL_XR=<011000101>, TEL_IR=<100>, TEL_MAR=<000111>,
-TEL_XR=<011000101>, TEL_IR=<100>, TEL_MAR=<000101>,
-TEL_XR=<011000101>, TEL_IR=<011>, TEL_MAR=<000101>,

IN STATE TEL_TRU
-TEL_XR=<011000101>, TEL_IR=<011>, TEL_MAR=<000101>,

IN STATE TEL_FETCH
-TEL_XR=<011000101>, TEL_IR=<011>, TEL_MAR=<000101>,
TEL_XR=<101001011>, TEL_IR=<011>, TEL_MAR=<000101>,
TEL_XR=<101001011>, TEL_IR=<011>, TEL_MAR=<000101>,
TEL_XR=<101001011>, TEL_IR=<011>, TEL_MAR=<000101>,
-TEL_XR=<101001011>, TEL_IR=<011>, TEL_MAR=<010111>,
-TEL_XR=<101001011>, TEL_IR=<101>, TEL_MAR=<001011>,

TEL_XR = <111111111>, TEL_IR=<101>, TEL_MAR=<001011>,

IN STATE TEL_STA
-TEL_XR=<111111111>, TEL_IR=<101>, TEL_MAR=<001011>,
-TEL_XR=<111111111>, TEL_IR=<101>, TEL_MAR=<001011>,

166
726--
727--
728--
729--
730--
731--
732--
733--
734--
735--
736--
737--
738--
739--
740----TEL_XR=<111000000>, TEL_IR=<101>, TEL_MAR=<000110>,
       -TEL_XR=<111000000>, TEL_IR=<101>, TEL_MAR=<000000>,
       -TEL_XR=<111000000>, TEL_IR=<111>, TEL_MAR=<000000>,
741--
742--
743--
744--
745--
746--
747--
748--
749--
750--

IN STATE TEL_STP
   -TEL_XR=<111000000>, TEL_IR=<111>, TEL_MAR=<000000>,
751--
752--
753--
754--
755--
756--
757--
758--
APPENDIX H:

MULTIPLE TEL_COMP DESCRIPTION
*SYSTEM*

* BUS ASR, ARW, AREQ, AACK, AWORD<9>, AADDR<6>;
 BUS BSR, BRW, BREQ, BACK, BWORD<9>, BADDR<6>;
*

* MEMORY: PROCESSOR( ASR, ARW, AREQ, AACK, AWORD<1:9>, AADDR<1:6>,
 BSR, BRW, BREQ, BACK, BWORD<1:9>, BADDR<1:6> );
 RAM MWR<9> = M( MAR<6> );
*

* WAIT: .THEN. ( AREQ*ARW ) WRITEA > ( AREQ*.NOT.(ARW) ) READO >
   ( BREQ*BRW ) WRITEB > ( BREQ*.NOT.(BRW) ) READB ;
 WRITEA: ( MAR=AAADDR$1 , MWR=AWORD$1 ) ; WRITE( M )$50 ; AACK=B'*$1
   .THEN. ( AREQ.EQ.B'O' ) ADONE;
 READO: MAR=AAADDR$1 ; READ( M )$50 ; AWORD=MWR$1 ; AACK=B'*$1
   .THEN. ( AREQ.EQ.B'O' ) ADONE ;
 ADONE: AACK=B'*$1 .THEN. .GOTO. WAIT ;
*
 WRITEB: ( MAR=BADDR$1 , MWR=BWORD$1 ) ; WRITE( M )$50 ; BACK=B'*$1
   .THEN. ( BREQ.EQ.B'O' ) BDONE;
 READB: MAR=BADDR$1 ; READ( M )$50 ; BWORD=MWR$1 ; BACK=B'*$1
   .THEN. ( BREQ.EQ.B'O' ) BDONE ;
 BDONE: BACK=B'*$1 .THEN. .GOTO. WAIT ;
*
* TELA: PROCESSOR( ASR, ARW, AREQ, AACK, AWORD<1:9>, AADDR<1:6> );
*
 REG LCR<7>, ACR<10>, AC1<9>, IR<3>, XR<9>, MAR<6>;
 DEF OP<3>:=XR<1:3>, AADDR<6>:=XR<4:9>,
   LCOV:=LCR<1>, LC<6>:=LCR<2:7>,
   OV:=ACR<1>, AC2<9>:=ACR<2:10>;
 ADDER ADDIT( ACR=XR, ADD=AC1 ) , INCR( LCR=LC, ADD=B'000001' );
*
 FETCH: ( AADDR=LC$1 , ARW=B'0'$1 ) ; AREQ=B'*$1 .THEN. ( AACK ) FETCH1 ;
 FETCH1: ( IR=AWORD<1:3>$1 , MAR=AWORD<4:9>$1 ) ; AREQ=B'0'$1 , ADD( INCN )$10
.THEN. (IR.EQ.B'000') ADD > (IR.EQ.B'001') SUB > (IR.EQ.B'010') SRO > (IR.EQ.B'011') TRU > (IR.EQ.B'100') TRN > (IR.EQ.B'101') STA > (IR.EQ.B'110') CLA > (IR.EQ.B'111') STP;

ADD: (ADDR=MAR$1, ARW=B'0'$1); AREQ=B'1'$1.
.THEN. (AACK) ADD1;
ADD1: XR=AWORD$1; (AREQ=B'0'$1, ADD(ADDIT)$10); AC1=AC2$1.
.THEN. GOTO FETCH;

SUB: (ADDR=MAR$1, ARW=B'0'$1); AREQ=B'1'$1.
.THEN. (AACK) SUB1;
SUB1: XR=AWORD$1; (AREQ=B'0'$1, XR=NOT(XR$1); ADD(ADDIT)$10;
AC1=AC2$1.
.THEN. GOTO FETCH;

SRO: XR=B'0000000001'$1; ADD(ADDIT)$10; AC1=B'0'|AC2<1:8'>$2
.THEN. GOTO FETCH;

TRU: LC=MAR$1.
.THEN. GOTO FETCH;

TRN: (AC1<1>) TRN1 > (NOT(AC1<1>) IFETCH;

TRN1: LC=MAR$1.
.THEN. GOTO FETCH;

STA: (ADDR=ADDR$1, AWORD=AC1$1, ARW=B'1'$1); AREQ=B'1'$1.
.THEN. (AACK) STA1;
STA1: AREQ=B'0'$1.
.THEN. GOTO FETCH;

CLA: AC1=B'0000000001'$1.
.THEN. GOTO ADD;

STP: ASR=B'0'$1.
.THEN. (ASR) FETCH;

*END.*

*TELB:PROCESSOR( BSR, BRW, BREQ, BACK, BWORD<1:9>, BADDR<1:6> );

*REG LCR<7>, ACR<10>, AC1<9>, IR<3>, XR<9>, MAR<6>;

DEF OP<3>::=XR<1:3>; ADDR<6>::=XR<4:9>;
LCOV:=LCR<1>, LC<6>::=LCR<2:7>
OV:=ACR<1>, AC2<9>::=ACR<2:10>;
ADDER ADDIT(ACR=XR, ADD, AC1); INCR(LCR=LC. ADD. B'000001');

*FETCH: (BADDR=LC$1, BRW=B'0'$1); BREQ=B'1'$1.
.THEN. (BACK) FETCH1;
FETCH1: (IR=BWORD<1:3>$1, MAR=BWORD<4:9>$1); BREQ=B'0'$1.
ADD(INCR)$10.
.THEN. (IR.EQ.B'000') ADD > (IR.EQ.B'001') SUB > (IR.EQ.B'010') SRO > (IR.EQ.B'011') TRU > (IR.EQ.B'100') TRN > (IR.EQ.B'101') STA > (IR.EQ.B'110') CLA > (IR.EQ.B'111') STP;

ADD: (BADDR=MAR$1, BRW=B'0'$1); BREQ=B'1'$1.
.THEN. (BACK) ADD1;
ADD1: XR=BWORD$1; (BREQ=B'0'$1, ADD(ADDIT)$10); AC1=AC2$1.
.THEN. GOTO FETCH;
SUB: (BADDR=MAR$1, BRW=B'0'$1); BREQ=B'1'$1.
.THEN. (BACK) SUB1;
SUB1: XR=BWORD$1 ; ( BREQ=B'0'$1 , XR=.NOT.(XR)$1 ) ; ADD( ADDIT )$10 ;
    AC1=AC2$1 .THEN. .GOTO. FETCH ;
SRO: XR=B'000000001'$1 ; ADD( ADDIT )$10 ; AC1=B'0'|AC2<1:8>$2
    .THEN. .GOTO. FETCH ;
TRU: LC=MARS1 .THEN. .GOTO. FETCH ;
TRN: .THEN. ( AC1<1> )TRN1 > ( .NOT.(AC1<1>) )FETCH ;
TRN1: LC=MAR$1 .THEN. .GOTO. FETCH ;
STA: ( BADDR=ADDR$1 , BWORD=AC1$1 , BRW=B'1'$1 ) ; BREQ=B'1'$1 .THEN. (BACK)STA1;
    STA1: BREQ=B'0'$1 .THEN. .GOTO. FETCH ;
CLA: AC1=B'000000000'$1 .THEN. .GOTO. ADD ;
STP: BSR=B'0'$1 .THEN. ( ASR ) FETCH ;
.END.

* * * * *

* * * * *

*SIMULATE.*

* *

* LOC OP ADDR CODE LOC OP ADDR CODE
  * 0 CLA 8 110 001000 12 CLA 20 110 010100
  * 1 SRO 0 010 000000 13 SRO 0 010 000000
  * 2 ADD 9 000 001001 14 ADD 21 000 010101
  * 3 SUB 10 001 001010 15 SUB 22 001 010110
  * 4 TRN 7 100 000111 16 TRN 19 100 010011
  * 5 STA 11 101 001011 17 STA 23 101 010111
  * 6 STP 0 111 000000 18 STP 12 111 001100
  * 7 TRU 5 011 000101 19 TRU 17 011 010001
  * 8 DC 8 000 001000 20 DC 8 000 001000
  * 9 DC 4 000 000100 21 DC 4 000 000100
  * 10 DC 9 000 001001 22 DC 9 000 001001
  * 11 DS 0 *** *** 23 DS 0 *** ***

SET TELA.AADDR=B'1111111', TELB.BADDR=B'1111111', TELA.AWORD=B'1111111111', TELB.BWORD=B'1111111111', TELA.AREQ=B'0', TELB.BREQ=B'0',
TELA. ASR=B'1', TELB. BSR=B'1',
MEMORY. AACK=B'0', MEMORY. BACK=B'0',
TELA. LC=B'000000', TELB. LC=B'001100',
TELA. AC1=B'111111111', TELB. AC1=B'111111111',
TELA. AC2=B'111111111', TELB. AC2=B'111111111',
TELA. MAR=B'111111', TELB. MAR=B'111111',
TELA. XR=B'111111111', TELB. XR=B'111111111',
TELA. IR=B'111', TELB. IR=B'111',
MEMORY. M(0)=B'110001000', MEMORY. M(1)=B'010000000', MEMORY. M(2)=B'000001001',
MEMORY. M(3)=B'000001010', MEMORY. M(4)=B'100001111', MEMORY. M(5)=B'101001011',
MEMORY. M(6)=B'011000000', MEMORY. M(7)=B'001100001', MEMORY. M(8)=B'000001000',
MEMORY. M(9)=B'000000100', MEMORY. M(10)=B'000000100', MEMORY. M(11)=B'100101010',
MEMORY. M(12)=B'110010100', MEMORY. M(13)=B'010000101', MEMORY. M(14)=B'100000111',
MEMORY. M(18)=B'111000000', MEMORY. M(19)=B'011010001', MEMORY. M(20)=B'000001000',
MEMORY. M(21)=B'000000100', MEMORY. M(22)=B'000001000', MEMORY. M(23)=B'111111111';
* ACTIVATE MEMORY. WAIT, TELA. FETCH, TELB. FETCH;
TRACE TELA. AREQ, TELA. AACK, TELB. BREQ, TELB. BACK .UNTIL.(TIME>1450);
* *
* END.
APPENDIX I:

MULTIPLE TEL_COMP SIMULATION
INITIAL
0---TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,

IN STATE TELB_FETCH
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,

IN STATE TELA_FETCH
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,

IN STATE MEMORY_WAIT
1---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,

2--

IN STATE MEMORY_READA
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
3---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
4--
5--
6--
7--
8--
9--
10--
11--
12--
13--
14--
15--
16--
17--
18--
19--
20--
21--
22--
23--
24--
25--
26--
IN STATE "TELA_FETCH1"
-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
56----TELA_AREQ=<0>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
57----

IN STATE "MEMORY_ADCONE"
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
55----TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
54----TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
53----TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,

IN STATE MEMORY_WAIT

IN STATE MEMORY_READB
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,

IN STATE TELA_CLA
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,

IN STATE TELA_ADD
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,

IN STATE TELA_ADD
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
  - TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
IN STATE TELB_FETCH1
-TELA_AREQ<1>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<1>,
-TELA_AREQ<1>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<1>,
-TELA_AREQ<1>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<1>,
112----TELA_AREQ<1>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<1>,
113---
IN STATE MEMORY_BDONE
-TELA_AREQ<1>, TELA_AACK<0>, TELB_BREQ<0>, TELB_BACK<0>,
114---
IN STATE MEMORY_WAIT
IN STATE MEMORY_READA
IN STATE TELB_CLA
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,

IN STATE TELB_ADD
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
IN STATE TELA_ADD1
  -TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
  168---TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
      -TELA_AREQ=<0>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,

IN STATE MEMORY_ADONE
  -TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,

IN STATE MEMORY_WAIT

IN STATE MEMORY_READB
176--
177--
178----TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
179--

IN STATE TELA_FETCH
---TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
---TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
180----TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
181--
182--
183--
184--
185--
186--
187--
188--
189--
190--
191--
192--
193--
194--
195--
196--
197--
198--
199--
200--
201--
202--
203--
204--
205--
206--
207--
208--
209--
IN STATE TELB_ADD1
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
224---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<1>,
225--

IN STATE MEMORY_BDONE
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
226--

IN STATE MEMORY_WAIT

IN STATE MEMORY_READA

IN STATE TELB_FETCH
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,

TELA_AREQ<1>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
273—
274—
275—
276—
277—-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
278—-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
279—
280—TELA_AREQ=<0>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
281—
282—
283—
284—
285—
286—
287—
288—
289—
290—-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
291—
292—
293—
294—
295—
296—
297—
298—

IN STATE TELA_FETCH1
-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,

IN STATE MEMORY_ADONE
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,

IN STATE MEMORY_WAIT

IN STATE MEMORY_READB

IN STATE TELA_SRO
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,

299---
300---TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
301---
302---

IN STATE TELA_FETCH
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
303---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
304---
305---
306---
307---
308---
309---
310---
311---
312---
313---
314---
315---
316---
317---
318---
319---
320---
321---
322---
323---
324---
325---
326---
327---
328---
329---
330---
331---
332---
333--TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
334--TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
335--
IN STATE TELB_FETCH1
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
336--TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<1>,
337--
IN STATE MEMORY_BDONE
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
338--
IN STATE MEMORY_WAIT
IN STATE MEMORY_REAODA
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
339--TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
340--
341--
342--
343--
344--
345--
IN STATE TELB_SRO
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
346--TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
347--
348--
349--
350--
351--
352--
353--
354--
355--
356--TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
357--
358--
IN STATE TELB_FETCH

- TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
- TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
359--- TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
360---
361---
362---
363---
364---
365---
366---
367---
368---
369---
370---
371---
372---
373---
374---
375---
376---
377---
378---
379---
380---
381---
382---
383---
384---
385---
386---
387---
388---
389--- TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
390--- TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
391---

IN STATE TELA_FETCH1
IN STATE MEMORY_ADONE
-TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,

IN STATE MEMORY_WAIT

IN STATE MEMORY_READB
-TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,

IN STATE TELA_ADD
-TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
-TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
-TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,

-TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
IN STATE TELB_FETCH1
  -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
  -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
  -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
  -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,

IN STATE MEMORY_BDONE
IN STATE TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <0>, TELB_BACK = <0>,

450--

IN STATE MEMORY_WAIT

IN STATE MEMORY_READA

- TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <0>, TELB_BACK = <0>,

451-- TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <0>, TELB_BACK = <0>,

452--

453--

454--

455--

456--

457--

IN STATE TELB_ADD

- TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <0>, TELB_BACK = <0>,

- TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <0>, TELB_BACK = <0>,

458-- TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <1>, TELB_BACK = <0>,

459--

460--

461--

462--

463--

464--

465--

466--

467--

468--

469--

470--

471--

472--

473--

474--

475--

476--

477--

478--

479--
480—
481—
482—
483—
484—
485—
486—
487—
488—
489—
490—
491—
492—
493—
494—
495—
496—
497—
498—
499—
500—
501——TEL_AREQ=<1>, TEL_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
502——TEL_AREQ=<1>, TEL_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
503——
IN STATE TELA_ADD1
---TEL_AREQ=<1>, TEL_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
504---TEL_AREQ=<1>, TEL_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
---TEL_AREQ=<0>, TEL_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
505---
IN STATE MEMORY_ADONE
---TEL_AREQ=<0>, TEL_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
506---
IN STATE MEMORY_WAIT
IN STATE MEMORY_READB
509--
510--
511--
512--
513--
514---TEL_AREQ=<0>, TEL_AACK=<0>, TEL_BREQ=<1>, TEL_BACK=<0>,
515--
516---IN STATE TELA_FETCH
-TEL_AREQ=<0>, TEL_AACK=<0>, TEL_BREQ=<1>, TEL_BACK=<0>,
-TEL_AREQ=<0>, TEL_AACK=<0>, TEL_BREQ=<1>, TEL_BACK=<0>,
516---TEL_AREQ=<1>, TEL_AACK=<0>, TEL_BREQ=<1>, TEL_BACK=<0>,
517--
518--
519--
520--
521--
522--
523--
524--
525--
526--
527--
528--
529--
530--
531--
532--
533--
534--
535--
536--
537--
538--
539--
540--
541--
542--
IN STATE TELB_ADD1

- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,

IN STATE MEMORY_BOONE

- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,

IN STATE MEMORY_WAIT

IN STATE MEMORY_READA

- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,
In state TELB_FETCH

- TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <0>, TELB_BACK = <0>,
- TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <0>, TELB_BACK = <0>,

572—- TELA_AREQ = <1>, TELA_AACK = <0>, TELB_BREQ = <1>, TELB_BACK = <0>,
573--
574---
575---
576---
577---
578---
579---
580---
581---
582---
583---
584---
585---
586---
587---
588---
589---
590---
591---
592---
593---
594---
595---
596---
597---
598---
599---
600---
601---
602---
603---
604---
605---
606--
607--
608--
609--
610--
611--
612--
613----TEL_A_REQ=<1>, TEL_A_ACK=<0>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
614----TEL_A_REQ=<1>, TEL_A_ACK=<1>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
615--
IN STATE
   TELA_FETCH
   -TEL_A_REQ=<1>, TEL_A_ACK=<1>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
   -TEL_A_REQ=<1>, TEL_A_ACK=<1>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
   -TEL_A_REQ=<1>, TEL_A_ACK=<1>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
616----TEL_A_REQ=<0>, TEL_A_ACK=<1>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
617--
IN STATE
   MEMORY_ADONE
   -TEL_A_REQ=<0>, TEL_A_ACK=<0>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
618--
IN STATE
   MEMORY_WAIT
IN STATE
   MEMORY_READ
   -TEL_A_REQ=<0>, TEL_A_ACK=<0>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
619----TEL_A_REQ=<0>, TEL_A_ACK=<0>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
620--
621--
622--
623--
624--
625--
IN STATE TELA_SUB
   -TEL_A_REQ=<0>, TEL_A_ACK=<0>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
   -TEL_A_REQ=<0>, TEL_A_ACK=<0>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
626----TEL_A_REQ=<1>, TEL_A_ACK=<0>, TEL_B_REQ=<1>, TEL_B_BACK=<0>,
627--
628--
629--
630--
668--
669- - - TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
670- - - TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=1,
671--

IN STATE TELB_FETCH1
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=1,
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=1,
672- - - TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=1,
673--

IN STATE MEMORY_BOONE
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,
674--

IN STATE MEMORY_WAIT
IN STATE MEMORY_READA
IN STATE TELB_SUB
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,
675- - - TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,
676--
677--
678--
679--
680--
681--

IN STATE TELB_SUB
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=0, TELB_BACK=0,
682- - - TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
683--
684--
685--
686--
687--
688--
689--
690--
691--
692--
TEL_AREQ=1, TEL_AACK=0, TEL_BREQ=1, TEL_BACK=0,

TEL_AREQ=1, TEL_AACK=1, TEL_BREQ=1, TEL_BACK=0,

IN STATE TELA_SUB1
  TEL_AREQ=1, TEL_AACK=1, TEL_BREQ=1, TEL_BACK=0,
728-- TELA_AREQ=\langle 1 \rangle, TELA_AACK=\langle 1 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,
- TELA_AREQ=\langle 0 \rangle, TELA_AACK=\langle 1 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,

729-- TELA_AREQ=\langle 0 \rangle, TELA_AACK=\langle 1 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,

IN STATE MEMORY_DONE
- TELA_AREQ=\langle 0 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,

730--

IN STATE MEMORY_WAIT

IN STATE MEMORY_READB

731-- TELA_AREQ=\langle 0 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,

732--

733--

734--

735--

736--

737--

738--

739-- TELA_AREQ=\langle 0 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,

740--

IN STATE TELA_FETCH

- TELA_AREQ=\langle 0 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,
- TELA_AREQ=\langle 0 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,

741-- TELA_AREQ=\langle 1 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,

742--

743--

744--

745--

746--

747--

748--

749--

750--

751--

752--

753--

754--

755--
IN STATE TELB_SUB1
   -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
   -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
   -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
   -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,

IN STATE MEMORY_BDONE
   -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,

IN STATE MEMORY_WAIT
IN STATE MEMORY_READA
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
  787---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
  788--
  789--
  790--
  791--
  792--
  793--
  794--
  795---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
  796--

IN STATE TELB_FETCH
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
  -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
  -TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
  797---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
  798--
  799--
  800--
  801--
  802--
  803--
  804--
  805--
  806--
  807--
  808--
  809--
  810--
  811--
  812--
  813--
  814--
  815--
  816--
  817--
  818--
819—
820—
821—
822—
823—
824—
825—
826—
827—
828—
829—
830—
831—
832—
833—
834—
835—
836—
837—TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
838—TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
839—

IN STATE TELA_FETCH1
     -TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
     -TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
     -TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
     -TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
840—TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
841—

IN STATE MEMORY_ADONE
     -TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
842—

IN STATE MEMORY_WAIT

IN STATE MEMORY_READB
     -TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
843—TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
844—
845—
846—
IN STATE TELA_TRN
IN STATE TELA_TRN1
  -TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,

IN STATE TELA_FETCH
  -TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
  -TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
  -TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,
  -TELA_AREQ<0>, TELA_AACK<0>, TELB_BREQ<1>, TELB_BACK<0>,

857--
858--
859--
860--
861--
862--
863--
864--
865--
866--
867--
868--
869--
870--
871--
872--
873--
874--
875--
876--
877--
IN STATE TELB_FETCH1
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
896---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
897---

IN STATE MEMORY_DONE
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
898--

IN STATE MEMORY_WAIT
IN STATE MEMORY_READA
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
899---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
900--
901--
902--
903--
904--
905--
IN STATE TELB_TRN
IN STATE TELB_TRN1
    -TELA_AREQ=<1>,TELA_AACK=<0>,TELB_BREQ=<0>,TELB_BACK=<0>,
906---
IN STATE TELB_FETCH
    -TELA_AREQ=<1>,TELA_AACK=<0>,TELB_BREQ=<0>,TELB_BACK=<0>,
    -TELA_AREQ=<1>,TELA_AACK=<0>,TELB_BREQ=<0>,TELB_BACK=<0>,
907---TELA_AREQ=<1>,TELA_AACK=<0>,TELB_BREQ=<1>,TELB_BACK=<0>,
908---
909---
910---
911---
912---
913---
914---
915---
916---
917---
918---
919---
920---
921---
922---
923---
924---
925---
926---
927---
928---
929---
930---
931---
932---
933---
934---
935---
936---
937—
938—
939—
940—
941—
942—
943—
944—
945—
946—
947—
948—
949---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
950---TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
951---

IN STATE TELA_FETCH1
-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
952---TELA_AREQ=<0>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
953---

IN STATE MEMORY_ADONE
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
954---

IN STATE MEMORY_WAIT
IN STATE MEMORY_READB
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
955---TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
956---
957---
958---
959---
960---
961---

IN STATE TELA_TRU
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
962---
IN STATE TELA_FETCH

- TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
- TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
- TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
IN STATE TELB_FETCH1
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
1008----TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<1>,
1009---IN STATE MEMORY_BDONE
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1010---IN STATE MEMORY_WAIT
IN STATE MEMORY_READA
IN STATE TELB_TRU
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1018---IN STATE TELB_FETCH
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1019----TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,

IN STATE MEMORY_READA
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1011----TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1012--
1013--
1014--
1015--
1016--
1017--
TEL_AREQ=<1>, TEL_AACK=<0>, TEL_BREQ=<1>, TEL_BACK=<0>,

1118---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
1119---
IN STATE TELB_FETCH1
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
1120---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<1>,
1121---
IN STATE MEMORY_BDONE
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1122---
IN STATE MEMORY_WAIT
IN STATE MEMORY_WRITEA
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1123---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1124---
1125---
1126---
1127---
1128---
1129---
IN STATE TELB_STA
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<0>, TELB_BACK=<0>,
1130---TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
1131---
1132---
1133---
1134---
1135---
1136---
1137---
1138---
1139---
1140---
TEL_AREQ=<1>, TEL_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
IN_STATE TELA_STA1
- TELA_AREQ=<0>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,

IN STATE TELA_FETCH
- TELA_AREQ=0, TELA_AACK=1, TELB_BREQ=1, TELB_BACK=0,
- TELA_AREQ=0, TELA_AACK=1, TELB_BREQ=1, TELB_BACK=0,
- TELA_AREQ=0, TELA_AACK=1, TELB_BREQ=1, TELB_BACK=0,
- TELA_AREQ=0, TELA_AACK=1, TELB_BREQ=1, TELB_BACK=0,

IN STATE MEMORY_ADONE
- TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,

IN STATE MEMORY_WAIT

IN STATE MEMORY_WRITEB
- TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
- TELA_AREQ=0, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,
- TELA_AREQ=1, TELA_AACK=0, TELB_BREQ=1, TELB_BACK=0,

1176--
1177---
1178--
1179--
1180--
1181--
1182--
1183--
1184--
1185--
1186--
1187--
1188--
1189--
1190--
1191--
1192--
1193--
1194--
1195--
1196--
1197--
1198--
1199--
1200--
1201--
1202--
IN STATE TELB_STA1

IN STATE TELB_FETCH

IN STATE MEMORY_BDONE

IN STATE MEMORY_WAIT

IN STATE MEMORY_READA
-TELA_AREQ=\langle 1 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 0 \rangle, TELB_BACK=\langle 0 \rangle,
-TELA_AREQ=\langle 1 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,
1231---TELA_AREQ=\langle 1 \rangle, TELA_AACK=\langle 0 \rangle, TELB_BREQ=\langle 1 \rangle, TELB_BACK=\langle 0 \rangle,
IN STATE TELA_FETCH1
-TELA_AREQ=<1>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
-TELA_AREQ=<1>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
1284---TELA_AREQ=<0>, TELA_AACK=<1>, TELB_BREQ=<1>, TELB_BACK=<0>,
1285--
IN STATE MEMORY_ADONE
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
1286--
IN STATE MEMORY_WAIT
IN STATE MEMORY_READB
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
1287---TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
1288--
IN STATE TELA_STP
-TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,

1294---
1295---
1296---
1297---
1298---
1299---
1300---
1301---
1302---
1303---
1304---
1305---
1306---
1307---
1308---
1309---
1310---
1311---
1312---
1313---
1314---
1315---
1316---
1317---
1318---
1319---
1320---
1321---
1322---
1323---
1324---
1325---
1326---
1327---
1328---
1329--
1330--
1331--
1332--
1333--
1334--
1335--
1336--
1337--- TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<0>,
1338--- TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
1339--
IN STATE TELB_FETCH1
--- TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
--- TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
--- TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
1340--- TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
1341--
IN STATE MEMORY BDONE
--- TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
1342--
IN STATE MEMORY_WAIT
1343--
1344--
1345--
1346--
1347--
1348--
1349--
IN STATE TELB STP
--- TELA_AREQ=<0>, TELA_AACK=<0>, TELB_BREQ=<1>, TELB_BACK=<1>,
1350--
1351--
1352--
1353--
1354--
1355--
1356--