A register-transfer descriptive language and simulator for digital networks
by William Platt Crane

A thesis submitted in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in Electrical Engineering
Montana State University
© Copyright by William Platt Crane (1977)

Abstract:
A computer hardware descriptive language was developed to describe digital networks at the Register-Transfer level. This language was then implemented into a computer program to allow simulation of the network.

The description language defines a digital network in terms of the hardware components and the interconnections among the components. Bused and directly-connected transfers are available. A wide array of data operations are available. Control branching capability is provided. Very few restrictions are placed upon the design; such quantities as the sizes of components, their interconnections, and data types are left entirely up to the designer.

The simulation of a network consists of the step-by-step execution of each transfer and branch operation. Values of components may be displayed as often as desired. Real-time interrupts may be simulated.

STATEMENT OF PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at Montana State University, I agree that the Library shall make this thesis freely available for inspection. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permssion.

Signature: William P. Crane

Date: 27 Mar 77

A REGISTER-TRANSFER DESCRIPTIVE LANGUAGE AND

SIMULATOR FOR DIGITAL NETWORKS

by

WILLIAM PLATT CRANE II

A thesis submitted in partial fulfillment
of the requirements for the degree

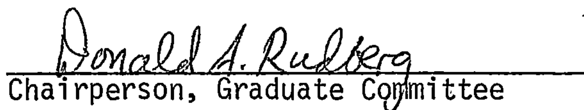
of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:



Chairperson, Graduate Committee

Head, Major Department

Graduate Dean

MONTANA STATE UNIVERSITY
Bozeman, Montana

June, 1977

Acknowledgments

I wish to thank Prof. Donald Rudberg who, as Chairman of my graduate committee, provided many helpful suggestions leading to the completion of this project.

I would also like to thank Jim Anderson for answering my endless stream of questions concerning the Sigma - 7 computer's operating system.

Many other people contributed ideas and criticism of this project as it evolved. A special thanks goes out to Bonnie Ellinghausen, Tim Estle, Dan Poole, Cheryl Schmidt, and John Campbell for their contributions.

TABLE OF CONTENTS

VITA ii

ACKNOWLEDGMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES vii

LIST OF TABLES viii

LIST OF PROGRAMS ix

ABSTRACT x

CHAPTER

I. INTRODUCTION 1

1.1. Current Status of Simulation Systems 1

1.2. Scope of Work 5

II. FEATURES OF THE SMALL DIGITAL SYSTEM SIMULATOR 6

III. A COMPLETE DESIGN EXAMPLE IN SDSS 11

IV. SUMMARY AND CONCLUSIONS 29

APPENDICES 33

A. SMALL DIGITAL SYSTEM SIMULATOR LANGUAGE REFERENCE MANUAL . 34

A1. Notation to be Used 36

A2. Character Set, Statement Format, and Operators . . . 36

A3. Symbolic Names 39

A4. Constants 39

A4.1. Unsigned Integer Constants 40

A4.2. Constants Formed by Alpha Generator 40

A4.3. Constants Formed by Omega Generator 41
A4.4. Constants Formed by Epsilon Generator 42
A4.5. Constants Formed by Encode Generator 43
A5. Types of SDSS Statements 46
A5.1. System Definition Statements 47
A5.1.1. Registers 48
A5.1.2. Scalars 49
A5.1.3. Panel Switches 51
A5.1.4. Panel Lights 52
A5.1.5. Random Acess Memory 54
A5.1.6. Read-Only Memory 56
A5.1.7. Logical Functions 57
A5.1.8. Data Paths 61
A5.1.8.1. Connect Statement 63
A5.1.8.2. Bus Statement 68
A5.2. Memory Initialization 73
A5.3. Control Sequence Statements 76
A5.3.1. Compression 77
A5.3.2. Bit Selection 79
A5.3.3. Reduction 80
A5.3.4. Transfer Statements 81
A5.3.4.1. Directly Connected Transfers 83

A5.3.4.2. Bused Transfers 95

A5.3.5. Branch Statements 104

A5.3.5.1. Unconditional Branch . . 105

A5.3.5.2. Three-way Conditional Branch 106

A5.3.6. Halt Statement 107

A5.4. Housekeeping Statements 108

A5.4.1. Print Statement 109

A5.4.2. End Statement 112

A5.4.3. Interrupt Statement 112

A5.4.4. Return Statement 114

A6. Compilation and Simulation Procedures 115

B. RESULTS FROM A SIMULATION WITH THE COMPUTER OF CHAPTER III 118

C. SOME NOTES ON THE SDSS COMPILER 129

REFERENCES 136

LIST OF FIGURES

1. HARDWARE DIAGRAM OF EXAMPLE COMPUTER 12

2. MACHINE INSTRUCTION SET 15

A1. DIRECTLY-CONNECTED DATA PATHS 61

A2. BUS-CONNECTED DATA PATH 62

A3. VALID SOURCE SYNTAX FOR DIRECTLY-CONNECTED DATA TRANSFERS . 87

A4. HARDWARE DIAGRAM FOR BUSED TRANSFERS 103

B1. ASSEMBLY PROGRAM TO SUM THREE VALUES 119

LIST OF TABLES

A1. DEFINITION OF TERMS USED IN FIGURE A3. 89

LIST OF PROGRAMS

1. COMPLETE DESCRIPTION OF A COMPUTER IN SDSS 17

ABSTRACT

A computer hardware descriptive language was developed to describe digital networks at the Register-Transfer level. This language was then implemented into a computer program to allow simulation of the network.

The description language defines a digital network in terms of the hardware components and the interconnections among the components. Bused and directly-connected transfers are available. A wide array of data operations are available. Control branching capability is provided. Very few restrictions are placed upon the design; such quantities as the sizes of components, their interconnections, and data types are left entirely up to the designer.

The simulation of a network consists of the step-by-step execution of each transfer and branch operation. Values of components may be displayed as often as desired. Real-time interrupts may be simulated.

I. INTRODUCTION

1.1 Current Status of Simulation Systems

The use of simulation as a design aid in the construction of digital systems has seen a large increase in recent years. A major cause of this increase has been the decrease in the time and cost of developing digital systems when simulation is employed. Given an adequate description of the system, it is possible to perform tests on the simulated system to determine such quantities as timing estimates, behavior of the system under heavy or unusual conditions, and possible problems such as a bottleneck along a data bus.

The crux of the simulation problem is implied by the term: adequate description. Prior to performing a simulation, one must decide what information he wishes to gain from the simulation. The choice of computer hardware descriptive language (CHDL) to be used to describe the digital system depends upon this decision.

Bell and Newell (2) indicate four major levels at which digital systems may be described:

1) the electronics level
2) the logic level
3) the programming level
4) the processor-memory-switch level

At the electronics level, all hardware is described in terms of basic electronic components, such as resistors and capacitors. The result of a simulation at this level is a record of the voltages and currents of the circuit as a function of time. Any circuit may be described at this level; however, virtually all of the discrete nature of digital circuits is lost, as the voltages and currents vary continuously (although perhaps very rapidly). Another disadvantage is that a large number of components is necessary to even simple digital networks. A simulation at this level would produce a very large amount of information, most of which would be of little use. Clearly, such a level of description is useless for describing digital systems.

The logic level defines digital systems in terms of logic functions. A simulation at this level produces the results of the logical operations specified as a function of time. These results are descrete values corresponding to the final state of the system after an operation has been performed, and are not continuous functions of time.

This logic level is loosely divided into several sublevels. The lowest defines a network in terms of primitive logic gates and flip-flops. Moving upwards networks evolve from simple combinational circuits into synchronous and asynchronous circuits containing memory. The top level of the logic level is commonly called the Register-Transfer (RT) level. Here, networks are described in terms

of larger memory elements, such as registers and random access memories, and the data paths used to connect these elements and perform operations upon the data during transit. Individual flip-flops and logic gates are relatively rare at the RT level, although occasionally they are used.

The programming level marks a large change in the description of digital systems. Below this level, the description is based upon the existence of specific hardware elements, be they registers or resistors. At the programming level, the description is not concerned with the hardware necessary to perform an operation; that is, a result is desired, and the hardware necessary to compute that result is irrelevant.

The programming level is associated with computers; that is, machines that interpret stored programs. Many digital systems, such as instrumentation systems, do not operate under stored programs. Thus, they have a logic level, but no program level of description.

The programming level specifies operations on specific data types, such as integer or floating point values. Programs are able to define data structures as collections of values, and to manipulate these structures to produce other structures. The logic level does not have this capability; it is concerned with boolean operations (and perhaps simple arithmetic operations) upon bit strings. The interpretation of the bit strings is left up to the designer.

The processor-memory-switch (PMS) level looks at the interconnections of the major units of a computing system. These units include devices such as entire processing units (CPU's), mass storage devices, and input / output devices. These units are connected together by data links. Description at this level conveys how the data is to be transfered and manipulated at an information processing basis. Items such as transfer rates and band widths of data channels are considered. For a complete description of PMS, see Bell and Newell (2).

The majority of the CHDL's have been developed at the logic level. A brief description of a number of these languages, along with an extensive bibliography, was presented by Su (6). All of the languages described by Su have been developed into complete simulation systems. Several have been adapted to produce hardware diagrams from the CHDL. For examples, see Barbacci and Siewiorek (1), Knudson (5), and Gentry (3).

1.2 Scope of Work

At the time that this project was undertaken, there was no means available at Montana State University to simulate digital systems. As a large amount of digital design is done at Montana State, a simulator was felt to be highly desirable. This project consisted of the specification and implementation of a simulation system, called the 'Small Digital System Simulator'.

The Small Digital System Simulator language had to meet several goals. The language must be able to describe a wide class of digital systems in a reasonable concise manner. The description must have a direct correspondence with the hardware necessary to implement the design. The language must be as free as possible from such restrictions as data formats, hardware component sizes and configurations, and the sequence of operations. It must provide facilities to allow the designer to observe and control the behavior of the digital system during the actual simulation. Finally, the language had to be easily translatable into a form which would allow simulation on the host computer, a Honeywell Sigma 7.

II. FEATURES OF THE SMALL DIGITAL SYSTEM SIMULATOR

The Small Digital System Simulator (SDSS) is based upon the CHDL ' A Hardware Programming Language', developed by Hill and Peterson (5). SDSS is a Register-Transfer language. Its major components are memory elements, such as registers and random access memories, and the data paths along which transfers are made.

A complete description of SDSS is presented in Appendix A. It is recommended that the reader be familiar with the contents of Appendix A before proceeding. Only a brief description of the major features of SDSS will be given here.

SDSS requires that all hardware elements that are to be present in the digital system be explicitly defined. Symbolic names are assigned to each element, and are used to refer to the element thereafter. Each element has a specific size, given in bits. With the exception of scalar elements, which by their nature contain only one bit, elements may contain up to 32 bits, inclusive.

One random access memory and one read-only memory may be defined for each digital system. A memory definition specifies: (1) the word-size of the memory, (2) the number of words in the memory, (3) the name of the register which will contain the address of the desired location within the memory whenever a memory reference is made, and (4) the name of the register to or from which data will be transfered

when a memory reference operation is made. Of course, data can not be stored into a read-only memory.

Another class of hardware elements which finds much usage is the logical function. A logical function is essentially a logic network which performs some operation not easily handled by the primitive operations allowed within SDSS. Such operations are addition and multiplication. Since these operations are generally quite simple conceptually (and relatively easy to implement in hardware) it is reasonable to treat them as individual operations in the description. Each function must be defined to SDSS by specifying its symbolic name, the number of arguments supplied to the function, and the size of the result returned by the function, in bits. SDSS includes several commonly used logical functions as primitive operators, and allows the inclusion of Fortran function subprograms for arbitrary functions.

Data paths are those routes along which data may be transfered between hardware elements. Transfers are allowed only along defined paths. Any transfer may specify some operation upon the data, such as a boolean operation, a logical function, or rotation. A transfer may specify which bits of a memory element are to be used as a data source or data destination in a given transfer. Generally, any selection of bits from an element is valid.

Data buses may be defined in a digital system. A bus is defined by giving the symbolic name, the size (in bits) of the bus, and all

connections to and from the bus. Operations may be performed upon the data values either prior to their being placed upon the bus, or after the data has been picked off of the bus, or both. It is clear that a bus is a special case of a data path.

A set of control sequence statements is used to describe the sequence of operations to be performed by the system. These statements specify the individual transfers to be made, and the order in which they are to be made. From two to ten transfers may be specified as occuring simultaneously. In such a case, any or all of the transfers may specify a given element as a data source. Only one transfer may specify a given element as its data destination.

In every set of simultaneous transfers, the original values of all elements will not be modified until after the data sources for all of the transfers have been computed.

A bused transfer must be specified as a set of two or more simultaneous transfers. Each of these transfers must specify a bus as its data source or data destination, or both. The bus will retain any value placed upon it for the duration of the set of transfers; thus, two or more transfers may specify a given bus as their data source. A bus will not maintain its value beyond the transfer period.

The sequence of operations may be altered by branch statements. Branch statements allow both conditional and unconditional branching.

Conditional branching depends upon the current values of the elements of the digital system.

A number of pseudo-statements are available. These statements are used to specify actions that are not part of the control sequence, and to convey information to the SDSS compiler. These statements include defining interrupt-handling routines and requesting a display of the current value of hardware elements.

An interrupt routine is a hardware routine described by a set of control sequence statements. This routine will be entered upon reciept of a real-time interrupt from outside the digital system. Up to 256 interrupt routines may be defined.

SDSS does not assume any data types or formats. The designer is free to implement any data types he desired. The only exceptions to this rule are the logical functions defined within SDSS. These functions operate assuming their arguments are in 2's complement form. The usage of any other data types will require that arithmetic operations be done either with a series of control sequence statements, or by an external logical function.

SDSS does not maintain any timing information. It is not possible to specify how much time a particular operation will consume. All transfers are done asynchronously; each transfer is initiated immediately following completion of the previous statement. For the case of

several simultaneous transfers, the time required by the set of transfers will be the time required by the slowest transfer.

III. A COMPLETE DESIGN EXAMPLE IN SDSS

A complete design example is presented here. A single accumulator computer is described in SDSS. The hardware arrangement for this computer is shown in Figure 1. Note that both bused and directly-connected transfers are included. The data paths are shown as unidirectional paths. The data paths connecting the random access memory 'M' with its data register 'MD' and its memory addressing register 'MA' are not explicitly defined as data paths; they are inplicitly included by the 'RAM' statement.

Observe in Figure 1 there are no paths shown for either the assignment of constants to elements, or for the shifting and rotation hardware associated with the elements 'AC' and 'L'. These paths were omitted for clarity. There are no paths connecting the functions 'FUN2' and 'WAIT' with their arguments and destinations. These functions, strictly speaking, are not part of the hardware of the computer. Function 'FUN2' permits communication from the operator to the computer, and function 'WAIT' executes a call to the Sigma - 7 monitor to enter a wait-state. In a real computer, these functions would not be present or necessary.

The machine wordsize is 18 bits. Six basic instructions are available in the computer, along with fourteen operate instructions. The operate instructions do not require a memory reference for their

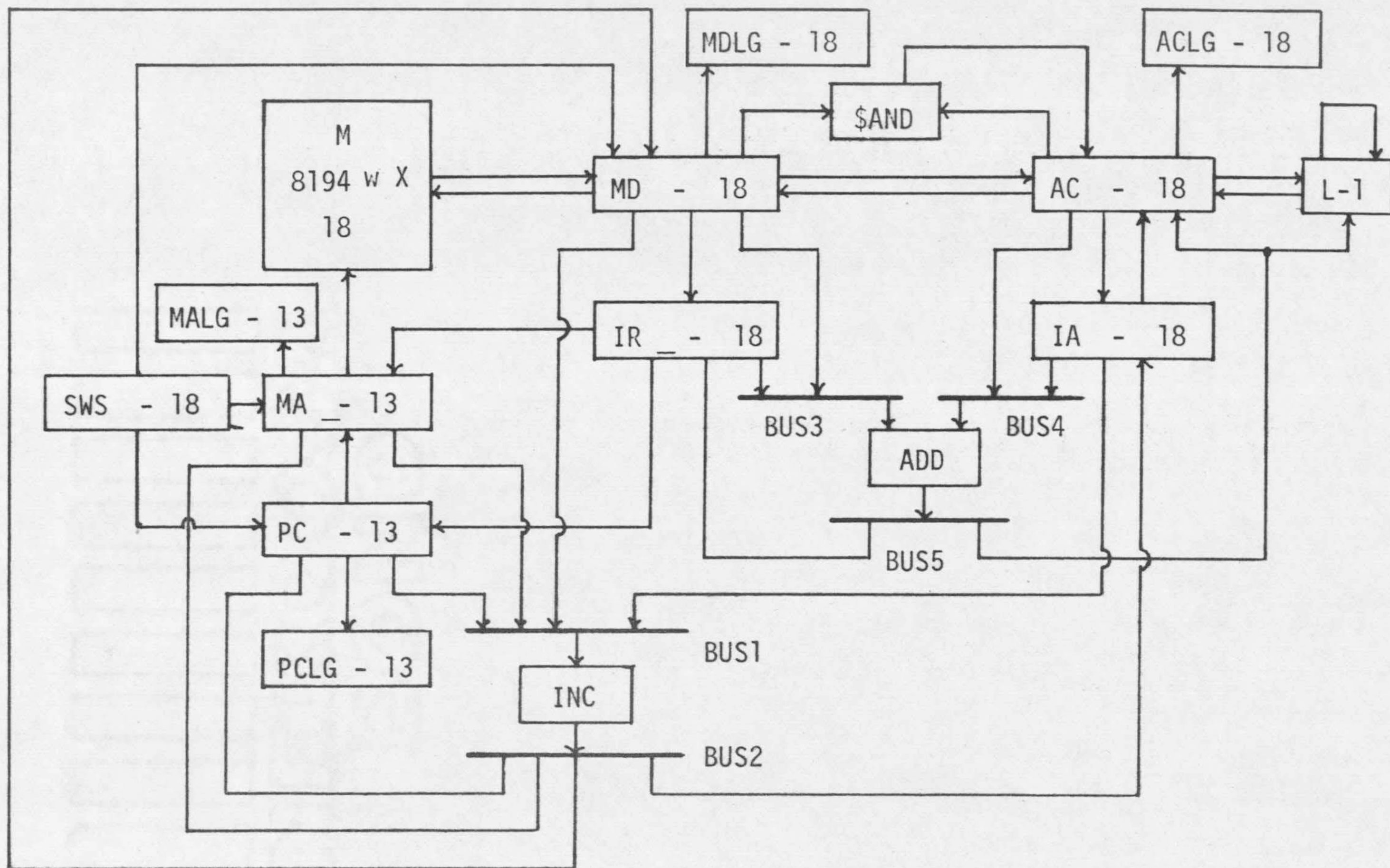


FIGURE 1

HARDWARE DIAGRAM OF EXAMPLE COMPUTER

(Numeric values give number of bits / component)

execution. The instruction set is defined in Figure 2.

A memory reference instruction may specify either indirect addressing or indexing, or both, to form the address of its operand. If both are requested, the indirect addressing is resolved first.

The machine will test the value of 'RFLAG' (Run-Flag) prior to each instruction fetch. If the value of 'RFLAG' is 1, the next machine instruction will be fetched from memory and executed. If the value of 'RFLAG' is 0, control will branch to the routine to request operator intervention via the front panel.

'RFLAG' may be reset to 0 by executing a 'HALT' instruction, or by recieving an external interrupt. An external interrupt indicates that the operator wishes to communicate with the computer through the front panel.

The front panel contains the following controls:

1) Run / halt switch
2) Load program counter from switches
3) Load memory address from switches
4) Store the contents of the switches into the memory location given by the contents of the memory address register, and increment the memory address register
5) Display the contents of the memory location specified by

the contents of the memory address register, and increment the contents of the memory address register.

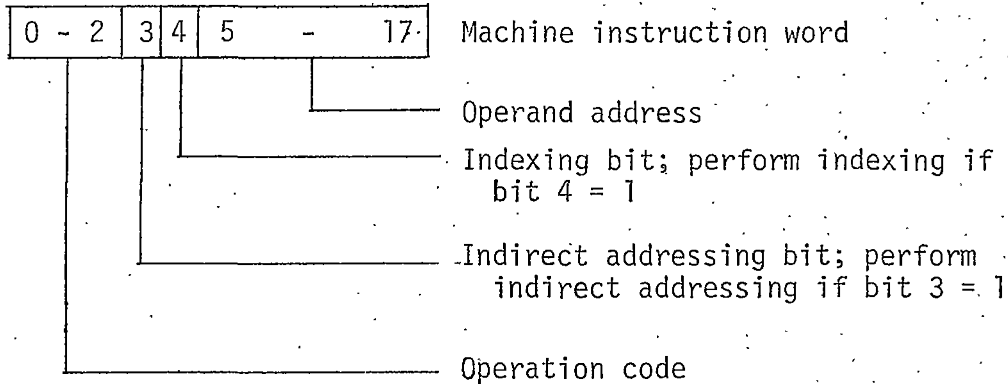
6) Single step through the next machine instruction.

The front panel also contains lights to display the contents of the memory address register, the memory data register, the program counter, and the accumulator. The contents of all four registers will be displayed following each front panel operation.

The SDSS description of this computer is given in Program 1.

An example of the results produced by a simulation of this computer is given in Appendix B. The simulation consists of entering a short program into the computer through the front panel and then executing the program.

Observe that the front panel could have been implemented by defining separate interrupt routines for each panel control. While this method may present more realism in terms of the hardware of the computer, it causes a lack of realism in terms of interaction with the computer. With several interrupt routines, it would be necessary to cause an interrupt, request the particular interrupt routine, and then enter the value of the switches (if necessary) to request one panel function. It appears to be a toss-up as to which method is more realistic.



Op code	Instruction
000	ISZ - Increment memory operand and skip the following instruction if the result is zero.
001	LAC - Load accumulator from memory.
010	AND - And the accumulator with the memory operand. Put result into the accumulator.
011	TAD - Two's complement addition of the contents of the accumulator with the memory operand. Result is placed in the accumulator.
100	JMS - Jump to subroutine. Increment the program counter and store value in memory location. Increment memory location value, and fetch next instruction from this location.
101	DAC - Deposit accumulator into memory location.
110	JMP - Fetch next instruction from memory location.

FIGURE 2

MACHINE INSTRUCTION SET

111 OPR - operate instruction.

Operate instructions utilize bits 0-6 for their operation code. Bits 0-2 are always 1's. No memory reference is necessary. Bits 7-17 are ignored. The operate instructions perform the following actions:

Op code	Instruction
1110000	Halt
1110001	IA <= AC
1110010	IA <= INC(IA)
1110011	AC <= IA
1110100	L,AC <= $SL(1) L,AC
1110101	L,AC <= $sr(1) L,AC
1110110	L,AC <= $RL(1) L,AC
1110111	L,AC <= $RR(1) L,AC
1111000	L <= 0
1111001	L <= 1
1111010	L <= $NOT L
1111011	AC <= 0
1111100	AC <= 1
1111101	AC <= $NOT AC
1111110	NOP
1111111	NOP

FIGURE 2 (Continued)

PROGRAM 1

COMPLETE DESCRIPTION OF A COMPUTER IN SDSS

```
C------------------------------------------------------------
C       DEFINE THE HARDWARE ELEMENTS OF THE MACHINE AND
C       DATA TRANSFER PATHS.
C------------------------------------------------------------

        REGISTER IA(18), MD(18),  AC(18),  IR(18),  MA(13),
        1       PC(13),  OPER(3)
        SCALAR L, RFLAG INITIAL (0)
        RAM M(18,8192),MAR = MA, MDR = MD
        FUNCTION INC (1,18), WAIT(1,3), FUN2(2,21),
        1       ADD (2,19)
        BUS BUS1(18),(IN=MA,PC,MD,IA),(OUT=INC)
        BUS BUS2(18),(IN=INC),(OUT=PC,MD,IA,MA)
        BUS BUS3(18),(IN=IA,MD),(OUT=ADD)
        BUS BUS4(18),(IN=AC,IR),(OUT=ADD)
        BUS BUS5(19),(IN=ADD),(OUT=(L,AC),IR)
        CONNECT (MD;AC),(AC;MD),(IA;AC),(AC;IA),(MD;IR),
        1      (IR;PC),(PC;MA),(SWS;MD),(OPER;WAIT),
        2      (WAIT;OPER),(OPER;FUN2),(SWS;FUN2),
        3      (MD $AND AC;AC), (PC;PCLG),(MA;MALG),
        4      (MD;MDLG),(0;RFLAG),(0;AC),(AC;AC),(L;L),
        5      ($SL(1)L,AC;L,AC),($SR(1)L,AC;L,AC),
        6      ($RL(1)L,AC;L,AC),(0;L),(SWS;MA),(SWS;PC),
        7      (IR;MA),(AC;ACLG),(FUN2;OPER,SWS),
```

PROGRAM 1 (CONTINUED)

```
        8       (#RR(1)L,AC;L,AC),(OPER;OPER)

C   CURRENT IMPLEMENTATION OF SDSS DOES NOT ALLOW
C SETTING THE SWITCHES BY THE PROGRAM OR BY AN OPERATOR.
C TO GET AROUND THIS, DEFINE THE SWITCHES AS AN ORDINARY
C REGISTER.

      REGISTER SWS(18)
      LIGHTS PCLG(13), ACLG(18), MDLG(18), MALG(13)
C- - - - - - - - - - - - - - - - - - - - - - - - - - -
C        CONTROL SEQUENCE STEPS FOLLOW.
C- - - - - - - - - - - - - - - - - - - - - - - - - - -
C        IF RFLAG = 1, THEN FETCH AND EXECUTE THE
C  INSTRUCTION POINTED TO BY THE PROGRAM COUNTER. ELSE,


 10   RFLAG:0 > 12, 500, 12
 12   MA < PC
      MD < M $DCD MA
      IR < MD
      PRINT PC,MA,MD,AC,IR,L,IA,RFLAG

C   LOOK FOR OPERATE INSTRUCTION

      $A(3)/IR:7 > 15, 125, 15

C   LOOK FOR 1 LEVEL OF INDIRECT ADDRESSING

 15   IR(3) : 1 > 25, 20, 25
```

PROGRAM 1 (CONTINUED)

```
 20   MA < $W(13)/IR
      MD < M $DCD MA
      $W(13)/IR < $W(13)/MD

C  LOOK FOR INDEXING

 25   IR(4) : 1 > 35, 30, 35
 30.  BUS3 < IA; BUS4 < IR; ADD < BUS3; ADD < BUS4;
    1      BUS5 < ADD;  $W(13)/IR < BUS5

C     SEPARATE REMAINING INSTRUCTIONS

 35   IR(0) : 0 > 40, 45, 40
 40   IR(1) : 0 > 120, 45, 120
 45   MA < $W(13)/IR
      IR(0) : 1 > 50, 90, 50
 50   MD < M $DCD MA
      $A(3)/IR : 0 > 55, 80, 55
 55   $A(3)/IR : 1 > 60, 75, 60
 60   $A(3)/IR : 2 > 65, 70, 65

C     TAD INSTRUCTION

 65.  BUS3 < MD; BUS4 < AC; ADD < BUS3; ADD < BUS4;
    1      BUS5 < ADD;  L,AC < BUS5
      > 115

C    AND INSTRUCTION
```

PROGRAM 1 (CONTINUED)

```
 70   AC < MD $AND AC
      > 115

C    LAC INSTRUCTION

 75   AC < MD
      > 115

C     ISZ INSTRUCTION

 80   BUS1 < MD;
     1 INC < BUS1;
     2 BUS2 < INC;
     3 MD < BUS2
      M $DCD MA < MD
      $OR/MD : 0 > 115, 85, 115
 85   BUS1 < PC; INC < BUS1; BUS2 < INC; PC < BUS2
      > 115

C       SEPARATE JMS FROM DAC

 90   IR(2) : 0 > 95, 100, 95

C     DAC INSTRUCTION

 95   MD < AC
      > 105

C     JMS INSTRUCTION
```

PROGRAM 1 (CONTINUED)

```
100   BUS1 < PC; INC < BUS1; BUS2 < INC; MD < BUS2
105   M $DCD MA < MD
      IR(2) : 0 > 115, 110, 115
110   PC < $W(13)/IR
115   BUS1 < PC; INC < BUS1; BUS2 < INC; PC < BUS2
      > 10

C     JMP INSTRUCTION

120   PC < $W(13)/IR
      > 10


C   OPERATE INSTRUCTIONS : DECODE OP CODE AND BRANCH

125   $W(4)/$A(7)/IR : 1 > 230, 225, 130
130   $W(4)/$A(7)/IR : 3 > 220, 215, 135
135   $W(4)/$A(7)/IR : 5 > 210, 205, 140
140   $W(4)/$A(7)/IR : 7 > 200, 195, 145
145   $W(4)/$A(7)/IR : 9 > 190, 185, 150
150   $W(4)/$A(7)/IR : 11> 180, 175, 155
155   $W(4)/$A(7)/IR : 13> 170, 165, 160

C     NO OPS

160   > 115

C     COMPLEMENT AC
```

PROGRAM 1 (CONTINUED)

```
 165   AC < $NOT AC
       > 155

C      SET AC TO 1'S

 170   AC < $E(18)
       > 155

C      SET AC TO 0

 175   AC < $ECD(0,18)
       > 115

C      COMPLEMENT L

 180   L < $NOT L
       > 115

C      SET L TO 1

 185   L < $ECD(1,1)
       > 115

C      SET L TO 0

 190   L < $ECD(0,1)
       > 115
```

PROGRAM 1 (CONTINUED)

```
C      ROTATE (L,AC) RIGHT 1 BIT

 195   L,AC < $RR(1) L,AC
       > 115

C      ROTATE (L,AC) LEFT 1 BIT

 200   L,AC < $RL(1) L,AC
       > 115

C      SHIFT (L,AC) RIGHT 1 BIT

 205   L,AC < $SL(1,0) L,AC
       > 115

C      SHIFT (L,AC) LEFT 1 BIT

 210   L,AC < $SL(1,0) L,AC
       > 115

C      LOAD IA INTO AC

 215   AC < IA
       > 115

C      INCREMENT IA

 220   BUS1 < IA; INC < BUS1; BUS2 < INC; IA < BUS2
       > 115
```