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: [illegible]

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:

Donald A. Rudberg
Chairperson, Graduate Committee

[illegible]
Head, Major Department

Henry L. Parsons
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

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMS

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

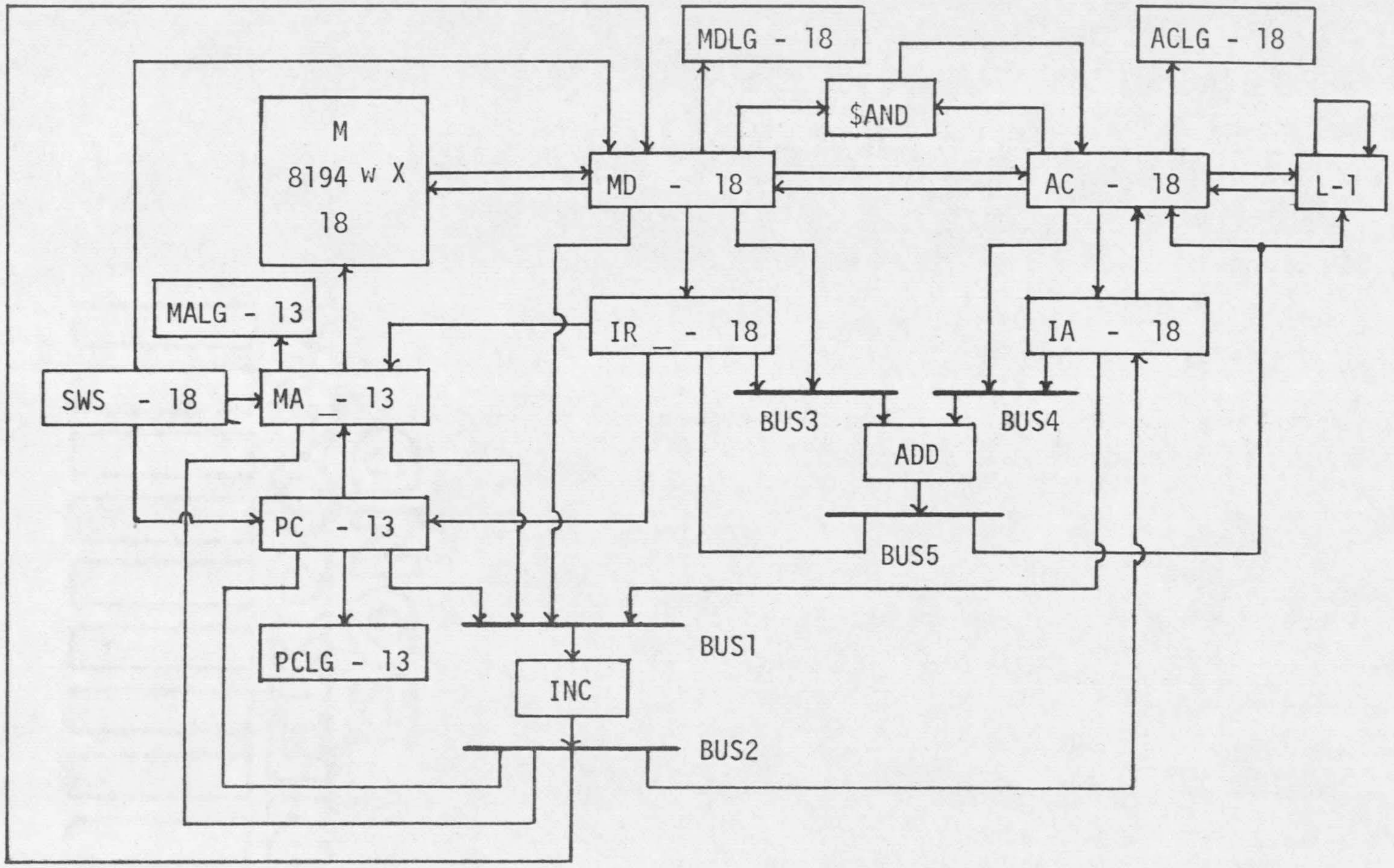


FIGURE 1

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.

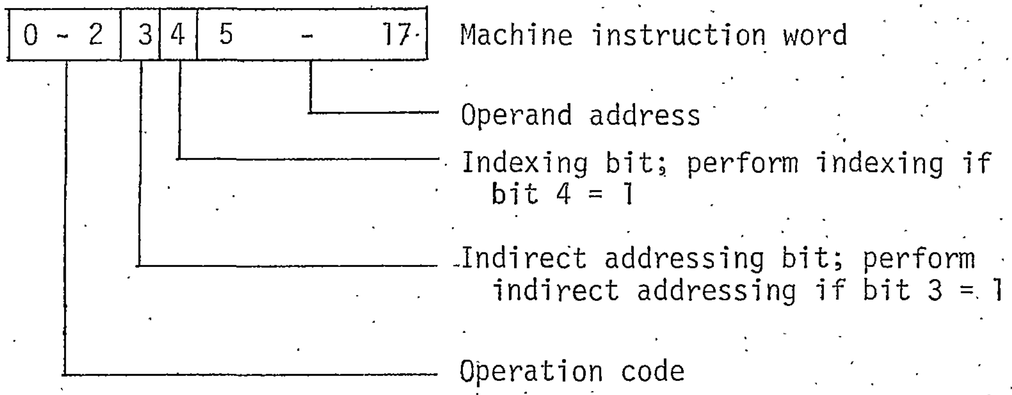
6) Single step through the next machine instruction.

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.


0 - 2
3
4
5 - 17
Machine instruction word
Operand address
Indexing bit; perform indexing if bit 4 = 1
Indirect addressing bit; perform indirect addressing if bit 3 = 1
Operation code

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
```

PROGRAM 1 (CONTINUED)

```
C     LOAD IA FROM AC

 225  IA < AC
      > 115

C     SET RUN FLAG = 0 -- MACHINE WILL ENTER CONSOLE
C  REQUEST  ROUTINE BEFORE NEXT INSTRUCTION FETCH.

 230  RFLAG < $ECD(0,1)
      > 115
C- - - - - - - - - - - - - - - - - - - - - - - - - -
C      INTERRUPT ROUTINE.  SET RFLAG = 0 AND FINISH
C    CURRENT INSTRUCTION THAT WAS INTERRUPTED.
C- - - - - - - - - - - - - - - - - - - - - - - - - -
      INTERRUPT 1
      RFLAG < $ECD(0,1)
      RETURN

C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
C      ROUTINE TO CONROL FRONT PANEL FUNCTIONS AND
C    DISPLAY REGISTER VALUES. REGISTERS ARE DISPLAYED
C    FOLLOWING EACH PANEL OPERATION.
C- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 500  PCLG < PC
      ACLG < AC
      MDLG < MD
      MALG < MA
      PRINT PCLG, MALG, MDLG, ACLG
```

PROGRAM 1 (CONTINUED)

```
C    REQUEST PANEL OPERATION AND SWITCH VALUES.

      OPER,SWS < FUN2( OPER, SWS )

C  OPER = 0 => ENTER WAIT STATE.
C         1 => LOAD PC FROM SWITCHES
C         2 => LOAD MA FROM SWITCHES
C         3 => STORE SWITCHES INTO MEMORY AT ADDRESS IN
C              MA AND INCREMENT VALUE IN MA BY 1
C         4 => DISPLAY MEMORY POINTED TO BY MA, AND
C              INCREMENT MA BY 1
C         5 => SET RUN FLAG = 1 AND RESUME EXECUTING
C              PROGRAM FROM THE ADDRESS IN PC
C         6 => SINGLE STEP NEXT MACHINE INSTRUCTION

      OPER : 1 > 545, 540, 505
 505  OPER : 3 > 535, 530, 510
 510  OPER : 5 > 525, 520, 515

C         SINGLE STEP THE MACHINE INSTRUCTION POINTED TO
C    BY  PROGRAM COUNTER.

 515  > 12

C         SET RUN FLAG = 1 AND RESUME PROCESSING WITH
C    INSTRUCTION  POINTED TO BY PROGRAM COUNTER.

 520  RFLAG < $ECD(1,1)
```

PROGRAM 1 (CONTINUED)

```
      > 10

C       DISPLAY MEMORY LOCATION AND INCREMENT MA

 525  MD < M $DCD MA
      BUS1 < MA; INC < BUS1; BUS2 < INC; MA < BUS2
      > 500

C       STORE SWS INTO MEMORY AND INCREMENT MA

 530  MD < SWS
      M $DCD MA < MD
      BUS1 < MA; INC < BUS1; BUS2 < INC; MA < BUS2
      > 500

C      LOAD MA FROM SWS

 535  MA < $W(13)/SWS
      > 500

C       LOAD PC FROM SWS

 540  PC < $W(13)/SWS
      > 500

C       PUT MACHINE INTO WAIT STATE; NEED TO INTERRUPT
C     TO  GET OUT.

545   RFLAG  < $ECD(0,1)
```

PROGRAM 1 (CONTINUED)

```
      OPER < WAIT( OPER )
      > 500
      END




     INTEGER FUNCTION FUN2( OPER,SWS )
     INTEGER OPER,SWS,FUN2
     DATA MASK / Z0003FFFF /
  20 OUTPUT 'ENTER OPERATION REQUEST'
     INPUT OPER
     IF ( OPER .LT. 0 .OR.
    1   OPER .GT. 6 ) OUTPUT 'INVALID REQUEST';
    2                 GO TO 20
     IF ( OPER .GE. 1 .AND.
    1   OPER .LE. 3     ) OUTPUT 'ENTER SWITCHES IN HEX';
    2                       READ ( 105,100 ) SWS
 100 FORMAT ( Z )
     SWS = IAND( SWS, MASK )
     FUN2 = IOR( ISL( OPER,18 ), SWS )
     RETURN
     END
```

PROGRAM 1 (CONTINUED)

```
      INTEGER FUNCTION WAIT (RFLAG )
      INTEGER RFLAG
      DATA J /Z0F001000 /
S10   CAL1,8    J
      RETURN
      END
```

IV. SUMMARY AND CONCLUSIONS

SDSS was developed to permit description and simulation of digital networks at a Register-Transfer level. The result of this development is a powerful and versatile system. The designer is free to choose virtually any hardware configuration. The hardware components that may be defined are registers, scalars, lights, switches, random and read-only memories, buses, and logical functions. Registers, lights, and switches may contain up to 32 bits, inclusive. The memories may have any wordsize up to 32 bits, inclusive, and contain any number of words. Buses may contain up to 64 bits, inclusive. Several logical functions to perform arithmetic operations are included within SDSS. For those operations not available, it is possible to incorporate standard Fortran FUNCTION subprograms to provide those operations. Any or all of the registers, switches, scalars, and memories may be initialized prior to initiation of the simulation to provide a starting point for the digital network.

Having decided upon a hardware configuration, the designer must specify all data paths and data operations which will be permitted. Operations available include the boolean operators (AND, NAND, OR, NOR, and XOR), shifting and rotation, concatenation, compression, and bit selection. All of these operations may be performed in directly connected transfers and in bused transfers.

A control sequence must be specified to define the particular transfers to be made, and the order in which they are to be made. Only those transfers which are along data paths which have been defined are allowed; that is, transfers must be made along existing hardware routes.

It is possible to describe a sequence of transfers which will interpret stored programs, or which will simply process data values. The former case is commonly called a computer, whereas the latter is representative of numerical algorithms and instrumentation systems.

Once the control sequence has been defined, SDSS will perform syntax checking on each statement. Valid statements will be compiled, while invalid statements will be flagged in error. Use of any hardware components that were not defined explicitly will cause an error, as will any transfers along non-existant data paths.

Having obtained a valid description, syntax-wise, the digital network may be simulated to verify or refute the logical description. Simulation consists of a step-by-step execution of each control sequence operation. If desired, the results following any operation may be displayed. Any number of test cases may be used to verify the design. Should the design be incorrect, it is a simple matter to track down the logic error(s).

Several other features are available in SDSS. Hardware interrupt routines may be defined to process random, exceptional conditions generated from outside of the digital system. Following the processing of the interrupt, the interrupted routine may be resumed. SDSS allows the contents of all hardware components to be displayed at any time; the values may be displayed in hexidecimal or decimal notation. The introduction of bit strings into the description is simplified by the several forms available in which bit strings may be defined.

SDSS will provide a useful instructional aid for those learning the concepts of sequential logic networks. The syntactical restrictions plus the one-to-one correspondence to hardware circuitry prohibits the design of networks that can not be realized. Such concepts as bused transfers, hardwired and micro-programmed control units, and machine organization can be explored easily. The ability to simulate a design provides perhaps the most efficient means of verifying concepts and designs utilizing these concepts.

A number of digital networks have been described and simulated under SDSS. From these simulations a number of recommendations can be made for future improvements in SDSS.

Some means of inputting data values directly into the network during simulation is necessary. Using a Fortran FUNCTION subprogram obscures the operation, and suffers from the inability to simulate a direct memory access operation, or set switches. (An 'INPUT' statement, suitable for inclusion in SDSS, is described in Appendix C.)

A statement similar to the Fortran 'CONTINUE' statement would be very handy to provide a null statement containing a label. Such a statement would be useful as a target for several branches all coming to the same point, such as a 'PRINT' statement. Currently, a transfer of the form:

$$A \leftarrow A$$

is necessary. Such a statement obscures the hardware somewhat.

A more versatile branch statement, analogous to the APL branch statement, would make branching of control easier to describe. Such a branch statement would perform boolean operations and reduction upon the data values to compute a single bit result. This single bit would control the branch. If no branch is taken, control would fall through to the next SDSS statement.

At the present, a processing unit (itself defined in SDSS) can not have a stack dedicated to its exclusive use; to utilize a stack, a portion of main memory must be used. By permitting multiple memories to be defined, it would be an easy matter to allow stacks, as well as consider memory interleaving concepts.

The most obvious deficiency in SDSS appears to be the lack of all timing information. Many operations do not take place exclusive of all others, but are initiated while other operations are concurrently taking place; results may not be available until several machine cycles later. Such transfers are impossible to simulate in SDSS.

APPENDICES

APPENDIX A

SMALL DIGITAL SYSTEM SIMULATOR

LANGUAGE REFERENCE MANUAL

THE SMALL DIGITAL SYSTEM SIMULATOR (SDSS) LANGUAGE AND COMPILER IS USED TO AID IN THE DESIGN OF SMALL DIGITAL SYSTEMS BY PROVIDING A CONCISE, EASILY-READ DESCRIPTION OF THE REGISTER-TRANSFERS WHICH TAKE PLACE WITHIN THE SYSTEM. THIS DESIGN MAY THEN BE TESTED BY SIMULATION TO DETERMINE IF THE DESIGN IS CORRECT.

SOME GENERAL FEATURES OF THE SDSS LANGUAGE AND COMPILER ARE:

1) ALL CONTROL SEQUENCE OPERATIONS SPECIFIED IN SDSS CORRESPOND TO A MICRO-OPERATION WITHIN AN ACTUAL DIGITAL SYTEM. EACH OPERATION IS WRITTEN AT THE REGISTER-TRANSFER LEVEL, PROVIDING ENOUGH DETAIL SO THAT IT IS EASY TO FOLLOW EACH OPERATION, YET NOT HAVING SO MUCH DETAIL THAT THE OVERALL OPERATION OF THE SYSTEM IS OBSCURED.

2) SEVERAL TYPES OF HARDWARE ELEMENTS WITH ARBITRARY SIZES MAY BE DEFINED.

3) DATA PATHS MUST BE DEFINED TO ALLOW THE TRANSFER OF DATA VALUES BETWEEN HARDWARE ELEMENTS.

4) OPERATIONS THAT ARE NOT AVAILABLE WITHIN SDSS MAY BE IMPLEMENTED BY MEANS OF A FORTRAN FUNCTION SUBPROGRAM.

5) REAL-TIME INTERRUPTS MAY BE SIMULATED.

6) AN EXTENSIVE ARRAY OF DATA TRANSFERS AND BRANCHING CAPABILITY IS AVAILABLE.

A1 NOTATION TO BE USED

IN THE FOLLOWING DESCRIPTION OF SDSS STATEMENT SYNTAX, THE NOTATION:

<NAME>

IS USED TO MEAN THAT THE QUANTITY "<NAME>" IS TO BE REPLACED BY A DESIGNER-SELECTED VALUE.

QUANTITIES WRITTEN WITHIN BRACKETS [ ] ARE OPTIONAL QUANTITIES: THEY MAY BE OMITTED IF DESIRED.

QUANTITIES WRITTEN WITHIN BRACES { } IMPLY THAT EXACTLY ONE OF THE ENCLOSED TERMS MUST BE CHOSEN.

BRACKETS AND BRACES MAY BE NESTED TO ANY LEVEL.

QUANTITES FOLLOWED BY AN ELIPSIS (...) MEAN THAT THE QUANTITY MAY BE REPEATED AS NECESSRY.

A2 CHARACTER SET, STATEMENT FORMAT, AND OPERATORS

THE CHARACTERS ALLOWED BY SDSS ARE THE FOLLOWING:

ALPHABETICS:	A - Z
NUMERICS:	0 - 9
SPECIAL CHARACTERS:	; : , = ( )
	' / * $ SP

THE ALPHANUMERIC CHARACTERS CONSIST OF THE ALPHABETICS PLUS THE NUMERICS.

AN SDSS STATEMENT FOLLOWS THE SAME GENERAL FORMAT AS ALLOWED IN FORTRAN:

A STANDARD 80-CHARACTER INPUT RECORD IS USED FOR SDSS STATEMENTS.

IF COLUMN 1 CONTAINS THE CHARACTER "C", THE CONTENTS OF THE RECORD ARE IGNORED EXCEPT FOR LISTING PURPOSES, AND MAY CONTAIN ANY DESIRED INFORMATION.

COLUMNS 1 - 5 ARE USED TO CONTAIN STATEMENT LABELS. A LABEL CONSISTS OF FROM 1 TO 5 NUMERICS; ALL SPACES WITHIN THE LABEL FIELD ARE IGNORED. A LABEL IS REQUIRED ON A STATEMENT ONLY IF THAT STATEMENT IS THE TARGET OF A BRANCH STATEMENT. LABELS MAY HAVE VALUES IN THE RANGE OF FROM 1 TO 99999, INCLUSIVE.

COLUMN 6 IS THE CONTINUATION COLUMN. IF COLUMN 6 CONTAINS ANY CHARACTER OTHER THAN A BLANK OR A ZERO, THE RECORD IS ASSUMED TO BE A CONTINUATION OF THE PREVIOUS RECORD. A TOTAL OF 10 RECORDS MAY BE USED TO CONTAIN AN SDSS STATEMENT. CONTINUATION RECORDS MAY NOT CONTAIN LABELS.

COLUMNS 7 - 72 ARE USED TO CONTAIN THE SDSS STATEMENT ITSELF.

COLUMNS 73 - 80 ARE IGNORED BY THE COMPILER, AND MAY BE USED FOR ANY DESIRED PURPOSE.

IN ADDITION, COMPLETELY BLANK LINES ARE PERMITTED IN SDSS. A BLANK LINE MAY NOT BE CONTINUED.

COMMENT LINES MAY NOT APPEAR WITHIN A CONTINUED STATEMENT.

BLANKS ARE IGNORED IN SDSS STATEMENTS EXCEPT WHEN THEY ARE CONTAINED WITHIN TEXT STRINGS. BLANKS MAY BE USED TO IMPROVE THE READABILITY OF THE SYSTEM DESCRIPTION.

A NUMBER OF OPERATORS ARE DEFINED IN THE SDSS LANGUAGE. SEVERAL CONSIST OF THE DOLLAR SIGN ("$") FOLLOWED BY A ONE TO THREE ALPHABETIC CHARACTER MNEMONIC. THESE OPERATORS ARE:

BOOLEAN OPERATORS:	$AND	: LOGICAL AND
	$NND	: LOGICAL NAND
	$OR	: LOGICAL OR
	$NOR	: LOGICAL NOR
	$XOR	: LOGICAL EXCLUSIVE-OR
	$NOT	: LOGICAL NEGATION
SHIFT OPERATOR:	$SL	: LEFT SHIFT
	$SR	: SHIFT RIGHT
ROTATE OPERATOR:	$RL	: ROTATE LEFT
	$RR	: ROTATE RIGHT
CONSTANT GENERATORS:	$A	: ALPHA CONSTANT GENERATOR
	$W	: OMEGA CONSTANT GENERATOR
	$E	: EPSILON CONSTANT GENERATOR
	$ECD	: ENCODE CONSTANT GENERATOR
MEMORY REFERENCE OPERATOR:	$DCD	

THE USAGE OF THE CONSTANT GENERATORS IS DETAILED IN SECTION A4, "CONSTANTS". THE USAGE OF THE OTHER "$" OPERATORS IS DESCRIBED IN SECTION A5.3.4, "TRANSFER

STATEMENTS', AND SECTION A5.3.5, 'BRANCH STATEMENTS'.

TWO OTHER OPERATORS ARE USED IN SDSS. THEY ARE THE COMPRESSION OPERATOR, DESCRIBED IN SECTION A5.3.1, AND THE REDUCTION OPERATOR, DESCRIBED IN SECTION A5.3.3.

A3 SYMBOLIC NAMES

A SYMBOLIC NAME CONSISTS OF ONE ALPHABETIC CHARACTER FOLLOWED BY ANY NUMBER, INCLUDING ZERO, OF ALPHANUMERIC CHARACTERS. HOWEVER, ONLY THE FIRST FOUR CHARACTERS OF A SYMBOLIC NAME ARE RETAINED BY THE COMPILER. THUS, EACH NAME SHOULD DIFFER IN THE FIRST FOUR POSITIONS.

A4 CONSTANTS

CONSTANTS ARE USED TO DENOTE A NUMERIC VALUE WHICH IS HARDWIRED INTO THE DIGITAL SYSTEM. EXCEPT FOR A FEW SPECIAL CASES WHICH WILL BE NOTED LATER, THE SDSS LANGUAGE DOES NOT RECOGONIZE NUMERIC VALUES AS BEING OTHER THAN SIMPLE BINARY BIT STRINGS. THUS, IT IS UP TO THE DESIGNER TO DETERMINE WHAT A BIT STRING REPRESENTS (SUCH AS A 2'S COMPLEMENT NUMBER). THIS FEATURE IS REFLECTED IN THE MANNER IN WHICH CONSTANTS ARE SPECIFIED.

SEVERAL FORMS OF CONSTANTS ARE ALLOWED. THEY ARE:

A4.1 UNSIGNED INTEGER CONSTANT

THIS FORM IS COMPOSED OF FROM 1 TO 10 DECIMAL DIGITS (NUMERICS); EMBEDDED BLANKS ARE IGNORED. VALUES OF FROM 0 TO 4294967295 (2**32 - 1) MAY BE REPRESENTED BY AN UNSIGNED INTEGER CONSTANT. UNLESS OTHERWISE STATED IN THIS MANUAL, ALL NUMERIC CONSTANTS WILL BE UNSIGNED INTEGERS.

A4.2 CONSTANTS FORMED BY ALPHA GENERATOR

THIS FORM CAUSES THE GENERATION OF A BIT STRING CONSISTING OF ONE OR MORE 1'S FOLLOWED BY ZERO OR MORE 0'S. TWO VARIATIONS OF THE ALPHA CONSTANT ARE AVAILABLE:

A) $A (<#ONES>, <#BITS>)

WHERE BOTH <#ONES> AND <#BITS> ARE UNSIGNED INTEGERS. THIS FORM SPECIFIED THAT THE LEFTMOST <#ONES> BITS OF A BIT STRING <#BITS> BITS LONG ARE TO BE SET TO 1'S. ANY REMAINING BITS ARE TO BE SET TO 0'S. THE VALUE OF <#BITS> MAY BE FROM 1 TO 32, INCLUSIVE. THE VALUE OF <#ONES> MAY BE FROM 1 TO <#BITS>, INCLUSIVE.

B) $A (<#ONES>)

WHERE <#ONES> IS AN UNSIGNED INTEGER. THIS ABBREVIATED FORM REQUIRES THAT THE LENGTH OF THE BIT STRING BE IMPLICITLY AVAILABLE FROM SOME OTHER PORTION OF THE SDSS STATEMENT. OF THIS LENGTH, THE LEFTMOST <#ONES> BITS ARE SET TO 1'S, AND ANY REMAINING BITS ARE SET TO 0'S. THE VALUE OF <#ONES> MAY NOT EXCEED THE IMPLICIT LENGTH. THE IMPLICIT LENGTH MAY NOT EXCEED 64 BITS.

A4.3 CONSTANTS FORMED BY OMEGA GENERATOR

THIS CONSTANT GENERATOR CREATES A BIT STRING CONSISTING OF ONE OR MORE 1'S PRECEEDED BY ZERO OR MORE 0'S. TWO FORMS ARE AVAILABLE:

A) $W (<#ONES>, <#BITS>)

WHERE <#ONES> AND <#BITS> ARE BOTH UNSIGNED INTEGER CONSTANTS. THIS FORM SPECIFIES THAT THE RIGHTMOST <#ONES> BITS OF A BIT STRING <#BITS> BITS LONG WILL BE SET TO 1'S, AND ANY

REMAINING BITS WILL BE SET TO 0'S. THE VALUE OF <#BITS> MAY BE FROM 1 TO 32, INCLUSIVE. THE VALUE OF <#ONES> MAY BE FROM 1 TO <#BITS>, INCLUSIVE.

B) $W (<#ONES>)

WHERE <#ONES> IS AN UNSIGNED INTEGER. THIS ABBREVIATED FORM REQUIRES THAT THE LENGTH OF THE BIT STRING BE IMPLICITLY ABBREVIATED FORM REQUIRES THAT THE LENGTH OF THE BIT STRING BE IMPLICITLY AVAILABLE FROM SOME OTHER PORTION OF THE SDSS STATEMENT. OF THIS LENGTH, THE RIGHTMOST <#ONES> BITS ARE SET TO 1'S, AND ANY REMAINING BITS ARE SET TO 0'S. THE VALUE OF <#ONES> MUST NOT EXCEED THE NUMBER OF BITS SPECIFIED BY THE IMPLICIT LENGTH. THE IMPLICIT LENGTH CAN NOT EXCEED 64 BITS.

A4.4 CONSTANTS FORMED BY EPSILON GENERATOR

THIS CONSTANT GENERATOR HAS TWO FORMS WHICH ARE INTERPRETED DIFFERENTLY. THEY ARE:

A) $E (<#ONES>)

WHERE <#ONES> IS AN UNSIGNED INTEGER. THIS FORM GENERATES A BIT STRING, KNOWN AS A FULL VECTOR, WHICH IS <#BITS> BITS LONG. EACH BIT OF THIS STRING IS SET TO A "1". THE VALUE OF <#ONES> MUST BE IN THE RANGE OF 1 TO 32, INCLUSIVE.

B) $E (<BIT>, <#BITS>)

WHERE BOTH <BIT> AND <#BITS> ARE UNSIGNED INTEGERS. THIS FORM GENERATES A BIT STRING WHICH IS <#BITS> LONG. ALL BITS IN THIS STRING ARE SET TO 0'S EXCEPT FOR THE <BIT>'TH BIT, WHICH IS SET TO A "1". NOTE THAT BIT-0 IS THE MOST SIGNIFICANT BIT IN THE STRING. THE VALUE OF <#BITS> MUST BE IN THE RANGE OF 1 TO 32, INCLUSIVE. THE VALUE OF <BIT> MUST BE IN THE RANGE OF 0 TO <#BITS>-1, INCLUSIVE. THIS FORM OF THE CONSTANT IS KNOWN AS A FULL VECTOR.

A4.5 CONSTANTS FORMED BY ENCODE GENERATOR

THIS CONSTANT GENERATOR ALLOWS ANY ARBITRARY BIT STRING TO BE SPECIFIED. THIS CONSTANT MAY BE GENERATED IN TWO FORMS:

A) $ECD (<VALUE>, <#BITS>)

WHERE <VALUE> AND <#BITS> ARE UNSIGNED INTEGERS. THIS FORM GENERATES A BIT STRING WHICH IS <#BITS> BITS LONG. <#BITS> MUST BE IN THE RANGE OF 1 TO 32, INCLUSIVE. THE CONSTANT GENERATED IS THE BINARY CODED VALUE OF <VALUE>. THE NUMERIC NUMERIC VALUE OF <VALUE> MUST BE IN THE RANGE OF 0 TO 4294967295 (2**32 - 1), INCLUSIVE. THE BINARY VALUE OF <VALUE> MUST OCCUPY NO MORE BITS THAN THOSE SPECIFIED BY <#BITS>.

B) $ECD (<VALUE>)

WHERE <VALUE> IS AN UNSIGNED INTEGER. THIS ABBREVIATED FORM REQUIRES THAT AN IMPLICIT LENGTH BE AVAILABLE FROM SOME OTHER PORTION OF THE SDSS STATEMENT. AS BEFORE, THE NUMBER OF BITS REQUIRED TO CONTAIN <VALUE> MUST NOT EXCEED THE NUMBER OF BITS GIVEN BY THE IMPLICIT LENGTH. <VALUE> MUST BE IN THE RANGE OF 0 TO 4294967295 (2**32-1).

EXAMPLES OF VALID CONSTANTS:

CONSTANT	BINARY VALUE
0	0
100	1100100
$A(5,10)	1111100000
$W(3,10)	0000000111
$E(4)	1111
$E(2,5)	00100
$ECD(1234,15)	000010011010010

A5 TYPES OF SDSS STATEMENTS

ALL STATEMENTS IN THE SDSS LANGUAGE CAN BE CLASSIFIED INTO FOUR GROUPS:

1) SYSTEM DEFINITION STATEMENTS. THESE STATEMENTS DEFINE THE HARDWARE ELEMENTS WHICH COMPOSE THE DIGITAL SYSTEM UNDER SIMULATION.

2) MEMORY INITIALIZATION STATEMENT. THIS STATEMENT ALLOWS MEMORIES TO BE INITIALIZED PRIOR TO THE SIMULATION OF THE SYSTEM IN ORDER TO SIMULATE AN INITIAL PROGRAM LOAD.

3) CONTROL SEQUENCE STATEMENTS. THESE STATEMENTS DEFINE THE SEQUENCE OF MICRO-OPERATIONS TO BE PERFORMED BY THE DIGITAL SYSTEM.

4) HOUSEKEEPING STATEMENTS. THESE STATEMENTS PERFORM SUCH OPERATIONS AS DISPLAY THE CONTENTS OF HARDWARE ELEMENTS, AND DEFINE INTERRUPT HANDLING ROUTINES.

EACH STATEMENT TYPE WILL BE DESCRIBED IN DETAIL DELOW.

A5.1 SYSTEM DEFINITION STATEMENTS

BEFORE ANY DIGITAL SYSTEM CAN BE SIMULATED, THE HARDWARE ELEMENTS WHICH COMPOSE THE SYSTEM MUST BE DEFINED. THE ELEMENTS WHICH MAY BE DEFINED IN SDSS ARE THE FOLLOWING:

REGISTERS
SCALARS
PANEL LIGHTS
PANEL SWITCHES
RANDOM ACCESS MEMORY
READ-ONLY MEMORY
LOGICAL FUNCTIONS
DATA PATHS

NO SYSTEM DEFINITION STATEMENTS MAY HAVE A LABEL. DEFINITION STATEMENTS MAY APPEAR IN ANY ORDER. HOWEVER, ALL SYSTEM DEFINITION STATEMENTS MUST PRECEED STATEMENTS OF ANY OTHER TYPE.

A5.1.1 REGISTERS

A REGISTER IS DEFINED BY MEANS OF THE 'REGISTER' STATEMENT. A REGISTER MAY CONTAIN FROM 1 TO 32 BITS, INCLUSIVE, AND MAY BE GIVEN AN INITIAL VALUE IF DESIRED. THE 'REGISTER' STATEMENT HAS THE FORM:

```
REGISTER <NAME> ( <SIZE> )[INITIAL ( <CONSTANT> )]
```

WHERE	<NAME>	IS THE UNIQUE NAME ASSIGNED TO THE REGISTER
	<SIZE>	IS THE SIZE, IN BITS, OF THE REGISTER. IT MAY HAVE A VALUE OF FROM 1 TO 32, INCLUSIVE.
	INITIAL	SPECIFIES THAT AN INITIAL VALUE IS TO BE ASSIGNED TO THE REGISTER.
	<CONSTANT>	IS ANY OF THE CONSTANTS DEFINED IN SECTION A4, AND GIVES THE THE INITIAL VALUE DESIRED.

IF THE LENGTH OF THE CONSTANT IS SPECIFIED, THEN THIS LENGTH MUST BE NO GREATER THAN THE NUMBER OF BITS IN THE REGISTER. IF NO LENGTH IS SPECIFIED IN THE CONSTANT, THEN THE LENGTH OF THE REGISTER WILL BE USED AS THE IMPLICIT LENGTH. IF A CONSTANT REQUIRES MORE BITS TO REPRESENT THE CONSTANT THAN ARE AVAILABLE IN

TWO OR MORE REGISTERS MAY BE DEFINED ON ONE 'REGISTER' STATEMENT BY SEPARATING EACH DEFINITION WITH COMMAS.

A MAXIMUM OF 50 REGISTERS MAY BE DEFINED. AS MANY 'REGISTER' STATEMENTS AS NECESSARY MAY BE USED TO DEFINE THESE REGISTERS.

EXAMPLES OF VALID REGISTER DEFINITIONS:

```
COLUMN
  678
   REGISTER REG1(10)
   REGISTER A123456(15) INITIAL (0)
   REGISTER A2(10), Y14(25) INITIAL
  1         ($A(5)), B(20)
```

A5.1.2 SCALARS

A SCALAR ELEMENT IS ONE WHICH STORES ONLY ONE BIT OF INFORMATION. AN OPTIONAL INITIAL VALUE MAY BE USED TO INITIALIZE THE SCALAR TO EITHER A "1" OR A "0". SCALAR ELEMENTS ARE DEFINED VIA THE 'SCALAR' STATEMENT, WHICH HAS THE FORM:

```
SCALAR <NAME> [INITIAL ( <CONSTANT> )]
```

WHERE <NAME>	IS THE UNIQUE NAME ASSIGNED TO THE SCALAR.
INITIAL	SPECIFIES THAT AN INITIAL VALUE IS TO BE ASSIGNED TO THE SCALAR.
<CONSTANT>	IS EITHER A "1" OR A "0". THIS VALUE IS USED AS THE INITIAL VALUE OF THE SCALAR.

MORE THAT ONE SCALAR MAY BE DEFINED BY A SINGLE "SCALAR" STATEMENT BY SEPARATING EACH DEFINITION WITH COMMAS.

A MAXIMUM OF 50 SCALARS MAY BE DEFINED. AS MANY "SCALAR" STATEMENTS AS NECESSARY MAY BE USED TO DEFINE THESE SCALARS.

EXAMPLES OF VALID SCALAR DEFINITIONS:

```
COLUMN
  678
   SCALAR A
   SCALAR J INITIAL (0)
   SCALAR K, L INITIAL (1), M
```

A5.1.3 PANEL SWITCHES

PANEL SWITCHES PERMIT THE SIMULATION OF MANUALLY ENTERING INFORMATION THROUGH THE FRONT PANEL INTO THE SYSTEM UNDER SIMULATION. A SET OF PANEL SWITCHES MAY BE THOUGHT OF AS A COLLECTION OF SINGLE SWITCHES, EACH CAPABLE OF HOLDING ONE BIT OF DATA INFORMATION. PANEL SWITCHES MAY BE GIVEN AN OPTIONAL INITIAL VALUE. SWITCHES ARE DEFINED VIA THE 'SWITCHES' STATEMENT, WHICH HAS THE FORM:

```
SWITCHES <NAME> ( <SIZE> )  [INITIAL ( <CONSTANT> )]
```

WHERE <NAME>	IS THE UNIQUE NAME ASSIGNED TO THE SET OF SWITCHES.
<SIZE>	IS THE NUMBER OF INDIVIDUAL SWITCHES MAKING UP THE SET. <SIZE> MAY HAVE A VALUE OF FROM 1 TO 32, INCLUSIVE.
INITIAL	INDICATES THAT AN INITIAL VALUE IS TO BE ASSIGNED TO THE SWITCHES.
<CONSTANT>	IS A VALID CONSTANT AS DEFINED IN SECTION A4, AND GIVES THE INITIAL VALUE DESIRED.

IF A LENGTH IS SPECIFIED IN THE CONSTANT, IT MUST NOT EXCEED THE SIZE OF THE SWITCHES. IF NO LENGTH IS SPECIFIED, THE LENGTH OF THE SWITCHES IS USED AS THE IMPLICIT LENGTH. IF THE CONSTANT REQUIRES MORE BITS FOR

ITS REPRESENTATION THAN ARE AVAILABLE IN THE SWITCHES, THE SWITCHES WILL NOT BE INITIALIZED.

TWO OR MORE SETS OF SWITCHES MAY BE DEFINED ON ONE "SWITCHES" STATEMENT BY SEPARATING EACH DEFINITION WITH COMMAS.

UP TO 5 SETS OF SWITCHES MAY BE DEFINED. AS MANY "SWITCHES" STATEMENTS AS NECESSARY MAY BE USED TO DEFINE THESE SWITCHES.

EXAMPLES OF VALID SWITCH DEFINITIONS:

```
COLUMN
  678
   SWITCHES A(10)
    SWITCHES B(15) INITIAL ( 0 )
   SWITCHES C(18)INITIAL($A(5)),
  1         SWS(16) INITIAL ($E(16))
```

A5.1.4 PANEL LIGHTS

PANEL LIGHTS PERMIT A VISUAL DISPLAY OF THE CONTENTS OF VARIOUS MEMORY ELEMENTS WITHIN THE SYSTEM UNDER SIMULATION. PANEL LIGHTS ARE DEFINED VIA THE "LIGHTS" STATEMENT, WHICH HAS THE FORM:

LIGHTS <NAME> (<SIZE>)

<SIZE> IS THE NUMBER OF BITS, OR INDIVIDUAL
BULBS IN THE SET OF LIGHTS. <SIZE>
MAY HAVE ANY VALUE FROM 1 TO 32,
INCLUSIVE.

UP TO FIVE SETS OF LIGHTS MAY BE DEFINED. AS MANY 'LIGHTS' STATEMENTS AS NECESSARY MAY BE USED.

TWO OR MORE SETS OF LIGHTS MAY BE DEFINED VIA ONE 'LIGHTS' STATEMENT BY SEPARATING EACH DEFINITION WITH COMMAS.

LIGHTS MAY NOT BE ASSIGNED AN INITIAL VALUE.

EXAMPLES OF VALID LIGHTS DEFINITIONS:

```
COLUMN
  678
   LIGHTS L1(18)
   LIGHTS L2(10), L21 ( 32 ),
  1       LGS(12)
```

A5.1.5 RANDOM ACCESS MEMORY

ONE RANDOM ACCESS MEMORY (RAM) MAY BE DEFINED FOR EACH DIGITAL SYSTEM. A RAM IS DEFINED BY MEANS OF THE "RAM" STATEMENT, WHICH HAS THE FORM:

```
RAM <NAME> ( <#BITS>, <#WORDS> ), MAR = <MARREG>,
                  MDR = <MDRREG>
```

WHERE <NAME>	IS THE NAME ASSIGNED TO THE MEMORY.
<#BITS>	IS THE NUMBER OF BITS PER WORD OF RAM, AND MUST BE IN THE RANGE OF 1 TO 32, INCLUSIVE.
<#WORDS>	IS THE TOTAL NUMBER OF WORDS IN THE MEMORY. <#WORDS> MUST BE GREATER THAN ZERO, AND IS LIMITED ONLY BY THE SIZE OF THE MEMORY OF THE HOST COMPUTER.
<MARREG>	IS THE NAME OF THE REGISTER TO BE USED AS THE MEMORY ADDRESS REGISTER. THIS REGISTER WILL CONTAIN THE ADDRESS OF THE LOCATION WITHIN THE RAM WHICH WILL BE ACCESSED IN A MEMORY REFERENCE OPERATION.
<MDRREG>	IS THE NAME OF THE MEMORY DATA REGISTER. THIS REGISTER WILL BE

USED TO SUPPLY DATA TO THE RAM, OR TO RECIEVE DATA FROM THE RAM, IN A MEMORY REFERENCE OPERATION.

THE REGISTERS <MARREG> AND <MDRREG> NEED NOT BE DEFINED AT THE TIME THE 'RAM' STATEMENT IS ENCOUNTERED. HOWEVER, THEY MUST BE DEFINED BY A 'REGISTER' STATEMENT PRIOR TO THE CONCLUSION OF OF THE SYSTEM DEFINITION STATEMENTS. THE <MARREG> REGISTER SHOULD CONTAIN ENOUGH BITS TO ADDRESS ALL WORDS IN THE RAM. THE <MDRREG> REGISTER MUST CONTAIN EXACTLY AS MANY BITS AS THE WORDSIZE OF THE RAM.

SHOULD THE <MARREG> REGISTER NOT CONTAIN ENOUGH BITS TO ACCESS ALL OF THE MEMORY, THAT PORTION OF THE MEMORY WITH ADDRESES IN ACCESS OF THE MAXIMUM ADDRESSABLE VALUE CAN NOT BE ACCESSED.

THE THREE OPERANDS IN A 'RAM' STATEMENT MAY APPEAR IN ANY ORDER.

EXAMPLES OF VALID RAM DEFINITIONS:

```
COLUMN
  678
   RAM MEM ( 18,1024 ),MAR=MA,MDR=MD
   RAM MAR=REG1, RAM(16,8192),
  *    MDR=REG2
```

NOTE THAT IF BOTH 'RAM' STATEMENTS APPEARED IN THE SAME SYSTEM DESCRIPTION, THE SECOND ONE WOULD BE IN ERROR; ONLY ONE RAM MAY BE DEFINED FOR EACH SYSTEM.

A 'RAM' STATEMENT IMPLICITLY DEFINES DIRECT DATA PATHS CONNECTING THE <MARREG> REGISTER WITH THE MEMORY, THE MEMORY WITH THE <MDRREG>, AND THE <MARREG> REGISTER WITH THE MEMORY. THE TWO REGISTERS <MDRREG> AND <MARREG> ARE THE ONLY ALLOWED MEANS OF COMMUNICATION WITH THE RAM.

A5.1.6 READ-ONLY MEMORY

ONE READ-ONLY (ROM) MEMORY MAY BE DEFINED FOR EACH DIGITAL SYSTEM. A ROM IS DEFINED BY MEANS OF THE 'ROM' STATMENT, WHICH HAS THE FORM:

```
ROM ( <#BITS>, <#WORDS> ), MAR = <MARREG>,
                     MDR = <MDRREG>
```

WHERE EACH OPERAND HAS THE SAME MEANING AS IT DOES IN A 'RAM' STATEMENT. ALL THE RULES OF DEFINITION OF A RAM APPLY TO A ROM. NOTE THAT IT IS POSSIBLE FOR BOTH A RAM AND A ROM TO HAVE THE SAME REGISTER(S) FOR THEIR MEMORY ADDRESSING AND MEMORY DATA REGISTERS.

A 'ROM' STATEMENT IMPLICITLY DEFINES DIRECT DATA PATHS CONNECTING THE ROM WITH THE <MDRREG> REGISTER, AND

THE <MARREG> REGISTER WITH THE ROM. THE TWO REGISTERS <MDRREG> AND <MARREG> ARE THE ONLY MEANS OF COMMUNICATING WITH THE ROM.

A5.1.7 LOGICAL FUNCTIONS

A LOGICAL FUNCTION, IN CONTRAST TO BOOLEAN OPERATORS, IS AN OPERATION THAT NORMALLY CAN NOT BE PERFORMED IN ONE MACHINE CYCLE TIME. SUCH OPERATIONS INCLUDE ADDITION, MULTIPLICATION, AND DIVISION. THESE OPERATIONS MUST BE PERFORMED BY A SUBSYSTEM OF THE MACHINE. THIS SUBSYSTEM MAY BE EITHER A HARDWIRED CIRCUIT ROUTINE, OR A SOFTWARE-CONTROLLED PROCESS. SDSS PROVIDES FOR THE INCLUSION OF LOGICAL FUNCTIONS TO DO SUCH OPERATIONS.

SDSS INCLUDES SEVERAL LOGICAL FUNCTIONS WHICH MAY BE REFERENCED DIRECTLY BY THE DESIGNER. THEY ARE: DIRECTLY BY THE DESIGNER. THEY ARE:

ADD -- TO ADD TWO VALUES TOGETHER USING 2'S COMPLEMENT ARITHMETIC.

INC -- TO INCREMENT A VALUE BY 1 USING 2'S COMPLEMENT ARITHMETIC.

DEC -- TO DECREMENT A VALUE BY 1 USING 2'S COMPLEMENT ARITHMETIC.

EACH OF THESE FUNCTION HAVE THREE OTHER NAMES BY WHICH THE SAME OPERATION MAY BE INVOKED. THE OTHER NAMES ARE:

FOR ADD: ADD1, ADD2, ADD3
FOR INC: INC1, INC2, INC3
FOR DEC: DEC1, DEC2, DEC3

THESE 12 FUNCTIONS ARE KNOWN AS THE BUILT-IN-FUNCTIONS (BIF'S).

THUS, IT IS POSSIBLE TO HAVE FOUR DIFFERENT HARDWARE UNITS IN THE SAME DIGITAL SYSTEM TO PERFORM THE SAME BASIC OPERATION, BUT HAVING NO INTERACTION AMONG THEM.

FOR THESE 12 FUNCTIONS, NO INDICATION OF OVERFLOW OR UNDERFLOW IS GIVEN. IT IS UP TO THE DESIGNER TO DETERMINE THE VALIDITY OF THE RESULTS.

SHOULD SOME OPERATION BE DESIRED THAT IS NOT AVALIABLE WHTHIN SDSS, THE DESIGNER CAN CREATE IT HIMSELF BY MEANS OF A STANDARD FORTRAN FUNCTION SUBPROGRAM, AND INCLUDE THIS SUBROUTINE AT PROGRAM LOAD TIME (SEE SECTION A6).

ALL FUNCTIONS, INCLUDING BIF'S, MUST BE DEFINED TO THE SDSS COMPILER BEFORE THEY MAY BE USED IN THE SYSTEM DESCRIPTION. TO DEFINE A FUNCTION, THE SDSS "FUNCTION" STATEMENT IS USED. IT HAS THE FORM:

```
FUNCTION <NAME> ( <#ARGS>, <#BITS> )
```

WHERE <#BITS>	IS THE NUMBER OF BITS IN THE RESULT RETURNED BY THE FUNCTION. <#BITS> MUST BE IN THE RANGE OF 1 TO 32, INCLUSIVE.
<#ARGS>	IS THE NUMMBER OF ARGUMENTS REQUIRED BY THE FUNCTION. ALL FUNCTIONS REQUIRE AT LEAST ONE ARGUMENT.

THE BUILT-IN-FUNCTIONS REQUIRE A FIXED NUMBER OF ARGUMENTS:

FUNCTION	NUMBER OF ARGUMENTS
ADD, ADD1, ADD2, ADD3	2
INC, INC1, INC2, INC3	1
DEC, DEC1, DEC2, DEC3	1

IF THE NAME OF A BIF IS USED AS SOME OTHER HARDWARE ELEMENT (SUCH AS A REGISTER) PRIOR TO BEING DEFINED AS A FUNCTION THEN THAT NAME AUTOMATICALLY CEASES TO BE A FUNCTION. SIMILARILY, IF A NAME HAS BEEN DEFINED AS A FUNCTION, NO OTHER ELEMENT MAY USE THAT NAME.

IT IS POSSIBLE TO REDEFINE THE NAME OF A BIF TO BE THE NAME OF A DESIGNER-SUPPLIED FUNCTION. TO DO THIS, SIMPLY PRECEED THE NAME OF THE BIF BY AN ASTERISK ("*"); THE NAME IS NO LONGER ASSOCIATED WITH THE BIF TO WHICH IT PREVIOUSLY REFERED.

THE USAGE OF A FUNCTION IN THE CONTROL SEQUENCE STATEMENTS IS DESCRIBED IN SECTION A5.3.4.

UP TO 12 EXTERNAL LOGICAL FUNCTIONS, IN ADDITION TO ANY BIF'S, MAY BE DEFINED. TWO OR MORE FUNCTIONS MAY BE DEFINED ON THE SAME "FUNCTION" STATEMENT BY SEPARATING EACH DEFINITION BY COMMAS.

VALID FUNCTION DEFINITIONS:

```
COLUMNS
  678
   FUNCTION ADD(2,19)
   FUNCTION INC(1,18), SUB(2,17)
   FUNCTION *DEC(3,10)
```

NOTE IN THE LAST EXAMPLE THE BIF 'DEC' IS REDEFINED TO BE AN EXTERNAL FUNCTION HAVING 3 ARGUMENTS AND RETURNING A VALUE 10 BITS LONG.

PROGRAMMING NOTE:

UNDER THE CURRENT IMPLEMENTATION OF SDSS, THERE IS NO PROVISION BY WHICH DATA VALUES MAY BE INPUT INTO THE SYSTEM DURING SIMULATION. ONE WAY TO OBTAIN DATA VALUES IS TO USE A FORTRAN FUNCTION SUBPROGRAM WHICH WILL REQUEST AND OBTAIN A DATA VALUE, AND RETURN IT TO THE SIMULATION AS ITS RESULT. SUCH A FORTRAN ROUTINE TO PERFORM THIS FUNCTION COULD BE:

```
FUNCTION INP1(I)
INPUT I
INP1 = I
RETURN
END
```

THIS FUNCTION COULD BE CALLED BY THE DIGITAL SYSTEM WITH THE TRANSFER:

```
A < INP1 (A)
```

SEE SECTION A5.3.4 FOR DETAILS ON THE TRANSFER STATEMENT.

A5.1.8 DATA PATHS

IN ORDER TO TRANSFER INFORMATION FROM ONE HARDWARE ELEMENT TO ANOTHER, A DATA PATH BETWEEN THE TWO ELEMENTS MUST EXIST. TWO TYPE OF DATA PATHS ARE AVAILABLE IN SDSS; THEY ARE THE DIRECTLY-CONNECTED DATA PATH, AND THE BUS-CONNECTED DATA PATH.

A DIRECTLY-CONNECTED DATA PATH IS ONE ON WHICH ONLY ONE UNIQUE HARDWARE ELEMENT MAY PLACE DATA, AND FROM WHICH ONLY ONE UNIQUE ELEMENT MAY EXTRACT DATA. SUCH A PATH IS DEPICTED IN FIGURE A1. THE ARROW INDICATES THE DATA PATH.

```
*****              *****
*   *              *   *
*   *              *   *        FIGURE A1
* A *===========>* B *
*   *              *   *     DIRECTLY CONNECTED
*   *              *   *
*****              *****         DATA PATH
```

EACH SUCH DATA PATH IS UNIDIRECTIONAL; DATA MAY BE TRANSFERED IN ONLY ONE DIRECTION. IN ORDER TO ALLOW TWO ELEMENTS TO 'TALK' WITH EACH OTHER, TWO DATA PATHS MUST BE DEFINED, ONE FOR EACH DIRECTION.

A BUS-CONNECTED DATA PATH ALLOWS ANY OF SEVERAL ELEMENTS (BUT ONLY ONE AT A TIME) TO PLACE DATA VALUES ON THE BUS, AND ONE OR MORE ELEMENTS (POSSIBLY SIMULTANEOUSLY) TO EXTRACT DATA VALUES FROM THE BUS. A BUS-CONNECTED DATA PATH MAY BE DEPICTED AS IN FIGURE A2. THE BUS IS NORMALLY AT LEAST AS WIDE (THAT IS, MAY CONTAIN AT LEAST AS MANY BITS) AS THE LARGEST ELEMENT THAT IS TO BE CONNECTED TO THE BUS.

```
*****        *        *****
*   *        *        *   *
* A *=======>*=======>* B *
*   *        *        *   *
*****        *        *****
             *
             *
*****        *
*   *=======>*
* C *        *
*   *<=======*
*****        *
             *
             *
*****        *        *****
*   *        *=======>*   *
* D *=======>*        * E *
*   *        *<=======*   *
*****        *        *****
            BUS
```

FIGURE A2

BUS-CONNECTED DATA PATH

AS BEFORE, EACH CONNECTION IS UNIDIRECTIONAL. NOTE THAT IN FIGURE A2 THAT ELEMENTS C AND E MAY BOTH SUPPLY DATA TO THE BUS AND EXTRACT DATA FROM THE BUS. ELEMENTS A AND D MAY ONLY SUPPLY DATA TO THE BUS, WHILE ELEMENT B MAY ONLY RECIEVE DATA FROM THE BUS. THE BUS IS REPRESENTED AS THE VERTICAL LINE IN FIGURE A2.

BOTH BUSED AND DIRECTLY CONNECTED DATA PATHS PROVIDE FOR THE CONCATENATION OF DATA SOURCES AND DESTINATIONS. TWO ITEMS MAY BE CONCATENATED TOGETHER TO FORM A SINGLE BIT STRING. THE RESULTING BIT STRING IS THEN TREATED AS A SINGLE BIT STRING IN ALL OPERATIONS.

TO DEFINE A DIRECTLY-CONNECTED DATA PATH, THE SDSS 'CONNECT' STATEMENT IS USED. TO DEFINE A BUS-CONNECTED DATA PATH, THE SDSS STATEMENT 'BUS' IS USED.

A5.1.8.1 CONNECT STATEMENT

THE 'CONNECT' STATEMENT IS USED TO DEFINE DIRECTLY CONNECTED DATA PATHS. THE STATEMENT HAS THE FORM:

```
CONNECT ( <PATH> ), ( <PATH> ), ...
```

WHERE <PATH> IS ONE OF NINE BASIC DATA PATH SPECIFICATIONS. EACH <PATH> DEFINES ONE UNIDIRECTIONAL DATA PATH. THE ALLOWED FORMS FOR <PATH> ARE THE FOLLOWING:

```
1) <ORG1>; <DEST1>
2) <ORG1>; <DEST1>, <DEST2>
3) <ORG1>, <ORG2>; <DEST1>
4) <ORG1>, <ORG2>; <DEST1>, <DEST2>
5) $<SR> ( <#S/R> ) <ORG1>; <DEST1>
6) $<SR> ( <#S/R> ) <ORG1>; <DEST1>, <DEST2>
7) $<SR> ( <#S/R> ) <ORG1>, <ORG2>; <DEST1>
8) $<SR> ( <#S/R> ) <ORG1>, <ORG2>; <DEST1>, <DEST2>
9) <ORG1> <OP> <ORG2>; <DEST1>
```

WHERE <SR>	IS A SHIFT OR ROTATE OPERATOR, AND MAY BE ONE OF: SL, SR, RL, OR RR.
<ORG1> & <ORG2>	SPECIFY THE DATA SOURCES TO BE USED FOR THIS DATA PATH. THE SOURCES MAY BE THE NAMES OF REGISTERS, SWITCHES, SCALARS, OR FUNCTIONS. A CONSTANT MAY ALSO BE USED AS A DATA SOURCE. IF A CONSTANT IS TO BE USED AS A DATA SOURCE, THEN THE TRANSFER PATH MUST ALLOW FOR THE CONSTANT. TO DO THIS, A ZERO ('0') IS USED FOR <ORG1> AND / OR <ORG2>. THE CONSTANT ITSELF IS NOT SPECIFIED UNTIL THE ACTUAL DATA TRANSFER STATEMENT IS ENCOUNTERED. ANY NUMBER OF DIFFERENT CONSTANTS MAY BE USED AS DATA SOURCES FOR THE TRANSFER ALONG A DATA PATH SO DEFINED.
<DEST1> & <DEST2>	ARE THE NAMES OF THE DATA DESTINATION ELEMENTS. DESTINATIONS MAY BE THE NAMES OF REGISTERS, SCALARS, FUNCTIONS, OR LIGHTS.

<#S/R> IS THE NUMBER OF SINGLE SHIFTS OR ROTATIONS DESIRED.

<OP> IS A BOOLEAN OPERATOR AND MUST BE ONE OF: $AND, $NND, $OR, $NOR, $XOR.

THE SEMICOLON SEPARATES THE DATA SOURCE FROM THE DESTINATION WITHIN EACH PATH FORM. COMMAS ARE USED TO INDICATE CONCATENATION OF ELEMENTS TO FORM SOURCES AND DESTINATIONS.

FORMS 1 THROUGH 4 ARE USED TO CONNECT ONE ELEMENT DIRECTLY TO ANOTHER. THE ONLY OPERATION UPON THE DATA ALONG THIS PATH IS NEGATION.

FORMS 5 THROUGH 8 ARE USED WHEN THE DATA SOURCE BITS ARE TO BE SHIFTED OR ROTATED BEFORE BEING STORED IN THE SPECIFIED DESTINATION. ANY SHIFT OR ROTATION OPERATOR MAY BE USED IN THESE FORMS.

FORM 9 IS USED WHEN A BOOLEAN OPERATION IS TO BE PERFORMED ON THE TWO DATA SOURCES SPECIFIED. THE RESULTING VALUE IS THEN STORED IN THE DESIGNATIED DESTINATION.

THE 'CONNECT' STATEMENT SAYS NOTHING ABOUT THE SIZES OF THE ELEMENTS THAT ARE CONNECTED. A DATA PATH BETWEEN TWO ELEMENTS IS ASSUMED TO BE CAPABLE OF TRANSFERING ANY OR ALL BITS OF THE SOURCE TO THE DESTINATION. ASSUMING THAT A AND B ARE REGISTERS OF 10 AND 20 BITS, RESPECTIVELY, THEN IT IS POSSIBLE TO TRANSFER THE ENTIRE CONTENTS OF A, OR A PORTION OF A, OR A SINGLE BIT OF A TO ANY EQUAL-SIZED PORTION OF B ALONG THE SINGLE DATA PATH GIVEN BY: 'CONNECT (A;B)'. DATA TRANSFERS ARE DESCRIBED IN SECTION A5.3.1.

THE FOLLOWING RESTRICTIONS MUST BE ADHERED TO WHEN USING THE 'CONNECT' STATEMENT:

1) DATA PATHS MUST CONNECT ALL ARGUMENTS TO ALL FUNCTIONS THAT UTILIZE THE ARGUMENT. DATA PATH FORM #1 MUST BE USED TO DO THIS. THE ARGUMENT IS SPECIFIED AS THE SOURCE, AND THE FUNCTION IS SPECIFIED AS THE DESTINATION. THE FUNCTION NAME MUST BE CONNECTED TO THE DESTINATION BY A DATA PATH OF FORM #1 OR #2.
2) A FUNCTION MAY NOT BE CONNECTED TO A FUNCTION. A FUNCTION NAME MAY NOT BE USED IN A CONCATENATED SOURCE OR DESTINATION SPECIFICATION, OR IN A SHIFT OR ROTATE DATA PATH.
3) LIGHTS MAY NOT BE USED AS DATA SOURCES.
4) SWITCHES MAY NOT BE USED AS A DATA DESTINATION.
5) THE NAMES OF BUSES AND MEMORIES MAY NOT BE USED IN A 'CONNECT' STATEMENT.
6) A HARDWARE ELEMENT MAY NOT BE CONCATENATED WITH ITSELF WHEN USED AS A DATA DESTINATION. IT MAY BE CONCATENATED WITH ITSELF WHEN USED AS A DATA SOURCE.

IT IS NOT NECESSRY TO HAVE DEFINED ALL HARDWARE ELEMENTS AT THE TIME THE 'CONNECT' STATEMENT IS ENCOUNTERED. HOWEVER, ALL ELEMENTS MUST BE DEFINED PRIOR TO THE CONCLUSION OF THE SYSTEM DEFINITION STATEMENTS.

APPROXIMATELY 200 DATA PATHS MAY BE DEFINED. THE EXACT NUMBER ALLOWED IS DEPENDENT UPON HOW MANY PATHS OF EACH TYPE ARE USED WITHIN THE SYSTEM DEFINITION SECTION.

VALID "CONNECT" STATEMENTS:

```
COLUMN
  678
   CONNECT (A;B), (A, B;C), (A,B;C,D)
   CONNECT ($SL(1)A;A), ($RR(2)B,D;B,D),
  1        (A $AND B; C )
   CONNECT ( 0;A ), ($SL(4)D,0; D )
```

THIS LAST "CONNECT" STATEMENT SPECIFIES THAT ONE OR MORE (AS YET UNSPECIFIED) CONSTANTS ARE TO BE TRANSFERED TO THE DESTINATION "A". THE SECOND PATH STATES THAT A CONSTANT IS TO BE CONCATENATED WITH "D", FORMING THE LOW ORDER BITS OF THE RESULTING STRING. THIS STRING IS THEN SHIFTED LEFT FOUR BITS. THIS TYPE OF PATH ALLOWS A SHIFTING OPERATION THAT SETS THE BITS SHIFTED INTO "D" TO BE SOMETHING OTHER THAN A STRING OF ALL 1'S OR 0'S.

A5.1.8.2 BUS STATEMENT

THE 'BUS' STATEMENT IS USED TO DEFINE BUS-CONNECTED DATA PATHS. THE 'BUS' STATEMENT HAS THE FOLLOWING TWO FORMS:

```
1) BUS <NAME> ( <SIZE> ), (IN= <IN>, <IN>, ... ),
        (OUT= <OUT>, <OUT>, ... )

2) BUS <NAME>, (IN= <IN>, <IN>, ... ),
        (OUT= <OUT>, <OUT>, ... )
```

WHERE <NAME>	IS THE UNIQUE NAME OF THE BUS.
<SIZE>	IS THE SIZE, IN BITS OF THE BUS. SIZE MUST BE IN THE RANGE OF 1 TO 64, INCLUSIVE.
<IN>	IS AN INPUT SPECIFICATION DEFINING A DATA SOURCE WHICH IS TO BE PLACED THE BUS.
<OUT>	IS AN OUTPUT SPECIFICATION DEFINING THE ELEMENTS WHICH MAY EXTRACT DATA FROM THE BUS.

THE SPECIFICATIONS <IN> AND <OUT> MAY HAVE ONE OF THE FOLLOWING FORMS:

```
 ┌ 0                                                            ┐
 │ <NAME>                                                       │
 │ $<SR> ( <#SR> ) <NAME>                                       │
 │                       ┌ <NAME> ┐  ┌ <NAME> ┐                 │
 │ $<SR> ( <#SR> ) (  {           } ,{          } )            }
 │                       └   0    ┘  └   0    ┘                 │
{  $<OP>                                                        │
 │  ┌ <NAME> ┐  ┌ <NAME> ┐                                      │
 │ ({          } ,{          } )                                │
 │  └   0    ┘  └   0    ┘                                      │
 └                                                              ┘
```

WHERE <NAME> IS THE NAME OF SOME HARDWARE ELEMENT.

<SR>	IS A SHIFT OR ROTATION OPERATOR, AND MAY BE ONE OF: SL, SR, RL, RR.
<#SR>	IS THE NUMBER OF SHIFTS OR ROTATES TO BE PERFORMED.
0	SPECIFIES THAT A CONSTANT WILL BE USED AS THE DATA SOURCE FOR THIS INPUT. THE CONSTANT ITSELF IS NOT GIVEN AT THIS TIME, BUT IS SPECIFIED AT THE TIME THE DATA TRANSFER IS ACTUALLY PERFORMED. ONCE THE BUS HAS "0" SPECIFIED AS AN INPUT, ANY NUMBER OF CONSTANTS MAY BE USED AS INPUT TO THAT BUS.
<OP>	IS A BOOLEAN OPERATOR, AND MAY BE ONE OF: AND, NND, NOR, OR, XOR.

NAMES OR CONSTANT SPECIFICATIONS (THE "0") THAT ARE ENCLOSED WITHIN PARENTHESIS INDICATE THAT THE TWO ELEMENTS

ARE TO BE CONCATENATED TOGETHER; THE LEFTMOST NAME OR CONSTANT BECOMES THE MORE SIGNIFICANT PORTION OF THE RESULTING BIT STRING.

FORM 1 OF THE 'BUS' STATEMENT IS THE COMPLETE DEFINITION FOR A BUS. ALL NECESSARY INFORMATION TO DEFINE THE BUS IS PROVIDED. FORM 2 IS USED SHOULD MORE SOURCES AND / OR DESTINATIONS BE REQUIRED THAN WILL FIT IN ONE SDSS STATEMENT. IF THIS FORM IS USED, THEN IT IS PERMISSIBLE TO OMIT ONE OF THE OPERANDS '(IN=.....)' OR '(OUT=.....)' IF THAT OPERAND IS NOT NEEDED.

A 'BUS' STATEMENT OF FORM 1 MUST ALWAYS APPEAR IF A BUS IS TO BE DEFINED. IF FORM 2 IS USED TO EXTEND THE NUMBER OF CONNECTIONS, IT MUST USE THE SAME NAME AS USED IN A FORM 1 'BUS' STATEMENT. THE FORM 2 STATEMENT MAY APPEAR PRIOR TO THE FORM 1 STATEMENT.

THE FOLLOWING RESTRICTIONS APPLY TO THE 'BUS' STATEMENT:

1) THE NAME OF A MEMORY MUST NOT APPEAR IN A BUS STATEMENT.
2) A CONSTANT ('0') MAY NOT BE GIVEN AS AN OUTPUT SPECIFICATION.
3) A HARDWARE ELEMENT MAY NOT BE CONCATENATED WITH ITSELF WHEN IT IS USED AS AN <OUT> SPECIFICATION.
4) SWITCHES MAY NOT BE USED AS AN <OUT> SPECIFICATION.
5) LIGHTS MAY NOT BE USED AS AN <IN> SPECIFICATION.
6) A BUS MAY BE CONNECTED TO ANOTHER BUS. HOWEVER, A BUS MAY NOT BE USED AS A CONNECTION TO ITSELF.

THE PHYSICAL ARRANGEMENT OF THE HARDWARE THAT IS CONNECTED TO A BUS MAY BE VISUALIZED AS FOLLOWS: IF SOMETHING IS CONNECTED AS AN INPUT TO THE BUS, THEN ALL OPERATIONS (IF ANY) WILL BE PERFORMED PRIOR TO THE DATA BEING PLACED ONTO THE BUS. SIMILARILY, FOR AN OUTPUT FROM THE BUS, DATA IS TAKEN FROM THE BUS, AND THEN OPERATED UPON BEFORE BEING PLACED IN THE DESIRED DESTINATION ELEMENT.

ALL DATA VALUES PLACED UPON THE BUS ARE POSITIONED IN THE LEAST SIGNIFICANT PORTION OF THE BUS. LIKEWISE, WHEN DATA IS BEING TAKEN FROM THE BUS, THE LEAST SIGNIFICANT BITS OF THE BUS ARE USED TO SUPPLY THE DATA BITS. THE BUS SHOULD BE AT LEAST AS WIDE AS THE LARGEST BIT STRING WHICH IS TO BE PLACED ON THE BUS.

GIVEN A BUS-TO-BUS CONNECTION, IT IS NOT NECESSARY TO DUPLICATE THE CONNECTION SPECIFICATION IN BOTH BUSES INVOLVED. FOR EXAMPLE, IN THE FOLLOWING "BUS" STATEMENTS:

```
BUS A(10), (OUT= B, ... ), (IN= ...)
BUS B(10), (IN = B, ... ), (OUT= ...)
```

THE PATHS SHOWN ARE IDENTICAL; EITHER ONE OF THEM MAY BE OMITTED. DUPLICATE CONNECTIONS (EITHER BUSED OR DIRECTLY CONNECTED) ARE ACCEPTED BY SDSS.

IF IT IS DESIRED TO PLACE THE CONTENTS OF A SINGLE BUS ONTO THE CONCATENATED COMBINATION OF TWO BUSES, THE PATH:

```
(OUT=(ABUS,BBUS), ... )
```

IS REQUIRED IN THE 'BUS' STATEMENT DEFINING THE SOURCE BUS. SIMILARILY, TO CONCATENATE TWO BUSES TO SUPPLY DATA TO A THIRD, THE PATH:

```
          (IN=(ABUS,BBUS), ... )
```

IS REQUIRED IN THE 'BUS' STATEMENT DEFINING THE THE RECIEVING BUS.

EXAMPLES OF VALID 'BUS' STATEMENTS:

```
COLUMN
 678
  BUS ABUS(18), (IN= A, B, (D,C), 0 ),
 *          (OUT= F,E,(H,I))
  BUS BBUS,(OUT= $SL(1),A $RR(2)(B,C), J ),
 *     (IN=0)
  BUS BBUS(20), (IN=TEST),(OUT=DUM)
  BUS CBUS(32), (IN= $AND, $NND, $XOR, $OR),
 *          (OUT=ACCUM)
```

THIS LAST CASE COULD BE SIMPLY A COLLECTION POINT FOR THE OUTPUT OF FROM ALL THE BOOLEAN OPERATORS, AND HAVE ONLY ONE ELEMENT (PERHAPS THE ACCUMULATOR) AS THE OUTPUT OF THE BUS.

A5.2 MEMORY INITIALIZATION

SDSS PROVIDES THE CAPABILITY OF INITIALIZING BOTH RANDOM ACCESS AND READ-ONLY MEMORIES PRIOR TO THE SIMULATION OF THE SYSTEM. THE 'FILL' STATEMENT IS USED FOR THIS PURPOSE. WHEN USED FOR A READ-ONLY MEMORY, THE 'FILL' IS THE ONLY MEANS BY WHICH THE MEMORY MAY HAVE ITS CONTENTS SPECIFIED. FOR A RANDOM-ACCESS MEMORY, THE 'FILL' STATEMENT MAY SIMULATE AN INITIAL PROGRAM LOAD PROCEDURE FOR THE DIGITAL SYSTEM.

A 'FILL' STATEMENT, IF PRESENT, MUST FOLLOW ALL SYSTEM DEFINITION STATEMENTS, AND PRECEED ALL CONTROL SEQUENCE AND HOUSEKEEPING STATEMENTS.

THE 'FILL' STATEMENT HAS THE FORM:

```
FILL <MEMORY> (<LO>,<HI>), (<LO>,<HI>), ...
```

WHERE <MEMORY>	IS THE NAME OF THE MEMORY BEING INITIALIZED.
<LO> & <HI>	SPECIFY THE LOW AND HIGH ADDRESSES OF A SECTION OF THE MEMORY WHICH IS TO BE INITIALIZED. BOTH <LO> AND <HI> MUST SPECIFY LOCATIONS WITHIN THE MEMORY. THE VALUE OF <HI> MUST BE NO SMALLER THAN THAT OF <LO>.

ALL WORDS WITHIN THE RANGE OF <LO> TO <HI>, INCLUSIVE, WILL BE INITIALIZED. AS MANY SECTIONS OF MEMORY AS DESIRED MAY BE INITIALIZED BY ONE 'FILL' STATEMENT. THE SECTIONS OF MEMORY TO BE INITIALIZED MAY BE IN ANY ORDER, AND MAY OVERLAP EACH OTHER.

IF IT IS DESIRED TO INITIALIZE ONLY ONE WORD OF MEMORY, THEN <LO> = <HI>.

THERE IS NO LIMIT TO THE NUMBER OF 'FILL' STATEMENTS.

THE FILL OPERATION IS INITIATED AT THE TIME THE SIMULATION IS REQUESTED, BUT PRIOR TO ANY OPERATIONS SPECIFIED BY THE HOUSEKEEPING OR CONTROL SEQUENCE STATEMENTS.

THE DATA VALUES WHICH ARE USED TO INITIALIZE THE MEMORY ARE READ FROM THE M:INF DATA CONTROL BLOCK (DCB). IT IS THE RESPONSIBILITY OF THE DESIGNER TO ENSURE THAT M:INF IS PROPERLY ASSIGNED VIA A SYSTEM ASSIGNMENT CONTROL STATEMENT (SEE SECTION A6).

THE DATA RECORDS READ THROUGH THE M:INF DCB ARE QUITE FREE-FORMAT. A STANDARD 80-CHARACTER RECORD IS READ; ANY EXCESS CHARACTERS ARE IGNORED. DATA VALUES MAY BE WRITTEN IN EITHER HEXIDECIMAL OR DECIMAL FORMAT.

TO SPECIFY THE FORMAT TO BE USED, A SINGLE LETTER IS PLACED IN COLUMN 1 OF THE DATA RECORD. TO SPECIFY A HEXIDECIMAL INPUT, THE CHARACTER 'X' IS USED; TO SPECIFY A DECIMAL INPUT, THE CHARACTER 'T' IS USED. ('T' IMPLIES BASE TEN. 'D' FOR DECIMAL IS NOT USED SINCE THE HEXIDECIMAL SYSTEM USES 'D' AS A VALID DIGIT.) ONCE A DATA FORMAT HAS BEEN SPECIFIED, IT THEN APPLIES TO ALL DATA VALUES ON THE CURRENT RECORD. THE FORMAT WILL REMAIN IN EFFECT OVER SUBSEQUENT RECORDS UNTIL EXPLICITLY CHANGED.

IF NO FORMAT IS SPECIFIED ON THE FIRST DATA RECORD, THE HEXIDECIMAL FORMAT IS ASSUMED BY DEFAULT.

ALL DATA VALUES ARE TERMINATED WHEN EITHER A SPACE OR A BLANK IS ENCOUNTERED, OR WHEN THE END OF THE RECORD IS REACHED. ALL BLANKS PRECEEDING A DATA VALUE ARE IGNORED.

A COMMA IMMEDIATELY FOLLOWING THE LAST DATA VALUE ON A RECORD INDICATES THAT ANOTHER VALUE IS TO BE OBTAINED FROM THAT RECORD. SINCE NO VALUE IS EXPLICITLY GIVEN, A ZERO WILL BE ASSUMED. FOR EXAMPLE, THE DATA RECORD:

```
        1,     2,3,4,
```

CONTAINS THE 5 DATA VALUES: 1,2,3,4, AND 0, IN THAT ORDER.

CONSECUTIVE COMMAS, WITH OR WITHOUT INTERVENING BLANKS, RESULT IN THE GENERATION OF ZEROS. FOR EXAMPLE, THE RECORD:

```
            1,2,   ,,, ,   3,
```

CONTAINS THE DATA VALUES: 1,2,0,0,0,0,3, AND 0, IN THIS ORDER.

IF AN END-OF-FILE IS ENCOUNTERED ON THE M:INF DCB PRIOR TO COMPLETION OF THE INITIALIZATION, ZEROS ARE GENERATED AS THE INITIAL VALUES UNTIL ALL REMAINING MEMORY LOCATIONS ARE FILLED. THIS PROVIDES A CONVENIENT MEANS TO SET LARGE BLOCKS OF MEMORY TO ZERO.

ANY INITIAL VALUE WHICH IS TOO LARGE TO BE CONTAINED WITHIN THE MEMORY WORDSIZE (AS DEFINED IN A 'RAM' OR 'ROM' STATEMENT) WILL HAVE HIGH ORDER BITS TRUNCATED SO THAT THE RESULTING VALUE WILL FIT WITHIN THE MEMORY WORDSIZE.

EXAMPLES OF VALID "FILL" STATEMENTS:

```
COLUMN
 678
  FILL MEM1 ( 0,100), ( 109, 109 )
  FILL MEM2 (100,120), (110, 115 )
```

A5.3 CONTROL SEQUENCE STATEMENTS

CONTROL SEQUENCE STATEMENTS ARE USED TO DESCRIBE THE INDIVIDUAL MICRO-OPERATIONS WHICH ARE INVOLVED DURING THE OPERATION OF A DIGITAL SYSTEM. CONTROL SEQUENCE STATEMENTS MAY BE GROUPED AS FOLLOWS:

1) TRANSFER STATEMENTS. THESE STATEMENTS SPECIFY HOW DATA IS TO BE MANIPULATED AND TRANSFERED FROM ONE HARDWARE ELEMENT TO ANOTHER. TRANSFER STATEMENTS ARE DESCRIBED IN SECTION A5.3.4.

2) BRANCH STATEMENTS. THESE STATEMENTS ARE USED TO MODIFY THE ORDER IN WHICH THE TRANSFER STATEMENTS ARE EXECUTED. BRANCH STATEMENTS ARE DESCRIBED IN SECTION A5.3.5.

3) "HALT" STATEMENT. THE "HALT" STATEMENT IS USED TO TERMINATE OPERATION OF A DIGITAL SYSTEM. THE "HALT" STATEMENT IS DESCRIBED IN SECTION A5.3.6.

ANY CONTROL SEQUENCE STATEMENT MAY HAVE A LABEL; ONLY THOSE STATEMENTS WHICH ARE TARGETS OF BRANCH STATEMENTS ARE REQUIRED TO HAVE A LABEL.

THE OPERATIONS OF COMPRESSION AND REDUCTION, AND THE BIT SELECTION NOTATION PROVIDE CONVENIENT MEANS OF SPECIFYING USEFUL OPERATIONS. THESE OPERATIONS AND NOTATION WILL NOW BE DEFINED.

A5.3.1 COMPRESSION

THE APPLICATION OF COMPRESSION PROVIDES A MEANS OF SELECTING ONLY CERTAIN BITS LOCATIONS FROM A MULTIPLE-BIT ELEMENT. THESE LOCATIONS WILL THEN BE USED AS DATA SOURCES OR DESTINATIONS IN A TRANSFER STATEMENT. (SEE SECTION A5.3.4 FOR USAGE OF COMPRESSION IN A TRANFER STATEMENT.)

A COMPRESSION IS REQUESTED BY THE FOLLOWING NOTATION:

<CONSTANT> / <NAME>

WHERE <CONSTANT>	IS ANY CONSTANTT GENERATED BY A CONSTANT GENERATOR ($A, $W, $E, OR $ECD). A LENGTH MAY BE SPECIFIED IN <CONSTANT>. IF SO, THEN THE LENGTH MUST SPECIFY EXACTLY THE SAME NUMBER OF BITS AS THERE ARE IN THE ELEMENT BEING COMPRESSED. IF

NO LENGTH IS SPECIFIED IN <CONSTANT>, THEN THE LENGTH OF THE ELEMENT WILL BE USED AS THE IMPLICIT LENGTH.

<NAME>	IS THE NAME OF THE MULTIPLE-BIT ELEMENT BEING COMPRESSED.

THE COMPRESSION OPERATION IS SIMPLE. IF <CONSTANT> IS WRITTEN AS A BINARY STRING, THEN, FOR EVERY "1" IN THE STRING, THE CORRESPONDING BIT LOCATION IN <NAME> IS SELECTED. THOSE BIT POSITIONS IN <NAME> WHICH CORRESPOND TO 0'S IN THE CONSTANT ARE IGNORED. WHAT IS DONE TO THE BIT POSITIONS THUS SELECTED DEPENDS ON THE USAGE OF THE COMPRESSION OPERATOR IN A TRANSFER STATEMENT.

AS AN EXAMPLE OF COMPRESSION, ASSUME "A" TO BE A 10-BIT REGISTER. THEN THE COMPRESSION

```
$W(5) / A
```

SELECTS THE LAST 5 BIT POSITIONS OF "A". THE COMPRESSION

```
$A(3) / A
```

SELECTS THE FIRST 3 BIT POSITIONS OF "A". THE COMPRESSION

```
$ECD( 682,10 ) / A
```

SELECTS EVERY OTHER BIT POSITION OF "A", BEGINNING WITH THE MOST SIGNIFICANT BIT OF "A".

MULTIPLE COMPRESSIONS ARE LEGAL. THEY HAVE THE FORM:

<CONSTANT> / <CONSTANT> / ... / <NAME>

IN SUCH A CASE, COMPRESSION PROCEEDS FROM RIGHT TO LEFT. EACH CONSTANT MUST SPECIFY NO MORE BITS THAN REMAIN AFTER THE COMPRESSION TO ITS RIGHT HAS TAKEN PLACE. FOR EXAMPLE, IF 'A' IS A 10-BIT REGISTER, THEN

```
$E(3) / $ECD( 248,10 ) / A
```

SPECIFIES BITS 4, 5, AND 6 OR 'A'.

SPECIAL CONSIDERATIONS MUST BE GIVEN TO THE COMPRESSION OF A BUS. IF THE BUS CONTAINS NO MORE THAN 32 BITS, THEN ANY OF THE FOUR CONSTANTS GENERATORS MAY BE USED TO COMPRESS THE BUS. IF THE BUS CONTAINS MORE THAN 32 BITS, THEN ONLY THE $A AND $W CONSTANTS GENERATORS MAY BE USED TO COMPRESS THE BUS, AND AN EXPLICIT LENGTH MUST NOT BE GIVEN IN THE CONSTANT. IF A LENGTH IS GIVEN IN SUCH A CASE, AN ERROR WILL RESULT.

A5.3.2 BIT SELECTION

BIT SELECTION IS A NOTATION USED TO SELECT A SINGLE BIT POSITION FROM A MULTIPLE-BIT HARDWARE ELEMENT. THE NOTATION USED IS:

<NAME> (<BIT>)

WHERE <NAME> IS THE NAME OF THE ELEMENT.

<BIT> SPECIFIES THE BIT POSITION TO BE SELECTED. <BIT> MAY HAVE A VALUE OF FROM 0 TO N-1, WHERE N IS THE NUMBER OF BITS IN <NAME>.

WHAT IS DONE WITH THIS BIT POSITION DEPENDS UPON ITS USAGE IN A TRANSFER OR BRANCH STATEMENT.

FOR EXAMPLES OF VALID BIT SELECTIONS, ASSUME THAT "B" IS A 10-BIT ELEMENT. THEN:

B(0) SELECTS THE MOST SIGNIFICANT BIT OF "B".
B(9) SELECTS THE LEAST SIGNIFICANT BIT OF "B".
B(8) SELECTS THE NEXT TO LEAST SIGNIFICANT BIT OF "B".

A5.3.3 REDUCTION

THE REDUCTION OPERATOR GENERATES A SINGLE BIT RESULT FROM A MULTIPLE BIT HARDWARE ELEMENT. THE REDUCTION OPERATOR IS INVOLKED AS SHOWN BELOW:

```
<OP> / <NAME>
```

WHERE <OP> IS ONE OF THE BOOLEAN OPERATORS: $AND, $OR, $NND, $NOR, $XOR.

<NAME> IS THE NAME OF A MULTIPLE-BIT ELEMENT.

THE REDUCTION OPERATION IS A SHORTHAND NOTATION FOR THE EXPRESSION:

```
(H(0)<OP>(...(H(N-3)<OP>(H(N-2)<OP>H(N-1) ) ... )
```

WHERE H IS A MULTIPLE BIT ELEMENT THAT IS N BITS LONG.

THE OPERATOR <OP> IS ALWAYS APPLIED IN A RIGHT-TO-LEFT MANNER ACROSS ALL BITS OF THE ELEMENT. REDUCTION MAY BE USED ONLY WITHIN BRANCH STATEMENTS.

<u>A5.3.4</u> <u>TRANSFER STATEMENTS</u>

A TRANSFER STATEMENT SPECIFIES DATA MOVEMENT (WITH POSSIBLE DATA MANIPULATION) ALONG A DATA PATH. THE DATA PATH MUST HAVE BEEN DEFINED PREVIOUSLY VIA A "CONNECT" OR 'BUS' STATEMENT. EVERY TRANSFER SPECIFIES A DATA SOURCE AND A DATA DESTINATION. EACH TRANSFER MAY BE WRITTEN AS:

```
<DESTINATION> < <SOURCE>
```

WHERE <DESTINATION>	DENOTES THE HARDWARE ELEMENTS WHICH ARE TO RECIEVE THE DATA VALUE SPECIFIED BY <SOURCE>.
<SOURCE>	DESIGNATES THE DATA ORIGIN, AND MAY INCLUDE OPERATIONS ON THAT DATA.

< IS THE TRANSFER OPERATOR.

TWO OR MORE TRANSFERS MAY BE EXECUTED "SIMULTANEOUSLY" BY WRITING THEM ON THE SAME SDSS SOURCE RECORD; EACH TRANSFER MUST BE SEPARATED FROM THE OTHERS BY SEMICOLONS. SUCH A SET OF TRANSFERS IS CALL A COMPOUND TRANSFER. A COMPOUND TRANSFER MAY EXTEND OVER CONTINUATION LINES, IF NECESSARY.

A TRANSFER MUST TAKE PLACE ALONG A DATA PATH. IF THE PATH IS A DIRECTLY-CONNECTED PATH (DEFINED VIA A "CONNECT" STATEMENT) THEN THE TRANSFER STATEMENT MUST EXPLICITLY SPECIFY SOURCE ELEMENT(S), POSSIBLE DATA OPERATIONS, AND THE DESTINATION ELEMENTS INTO WHICH THE DATA VALUE IS TO BE PLACED. EACH DIRECTLY-CONNECTED TRANSFER DENOTES A COMPLETE TRANSFER.

IF THE TRANSFER IS ALONG A BUSED DATA PATH, THEN A MINIMUM OF TWO TRANSFERS ARE NECESSARY TO SPECIFY THE TOTAL TRANSFER. (FOR EXAMPLE, ONE TRANSFER LOADS A BUS FROM A REGISTER; THE OTHER TAKES DATA FROM THAT BUS, AND DEPOSITS IT INTO A REGISTER.) THUS, A COMPOUND TRANSFER IS ALWAYS REQUIRED WHEN DEALING WITH BUSED TRANSFERS.

IT IS LEGAL TO COMBINE BUSED TRANSFERS WITH NON-BUSED TRANSFERS IN A SINGLE COMPOUND TRANSFER STATEMENT, PROVIDED THERE IS NO CONFLICT OF HARDWARE RESOURCES.

THERE ARE NO TIMING CONSIDERATIONS WITHIN SDSS; EACH COMPOUND TRANSFER WILL CONSUME AS MUCH "TIME" AS NECESSARY TO COMPLETE THE ENTIRE SET OF TRANSFERS. DURING THIS TIME, ALL BUSES WILL MAINTAIN THEIR ASSIGNED VALUES. NOTE THAT THE BUS WILL NOT RETAIN ITS ASSIGNED VALUE AFTER ALL THE TRANSFERS HAVE BEEN COMPLETED.

THE 'TIME' REQUIRED TO COMPLETE EACH COMPOUND TRANFSER STATEMENT IS THE TIME REQUIRED BY THE 'SLOWEST' SINGLE TRANSFER WITHIN THE COMPOUND TRANSFER. THE FOLLOWING CONTROL SEQUENCE STATEMENT WILL NOT BE INITIATED UNTIL THE CURRENT STATEMENT IS COMPLETED. IT IS NOT POSSIBLE TO INITIATE A TRANSFER (FOR EXAMPLE, AN EXTENDED PRECISION FLOATING POINT DIVISION) AND PICK UP THE RESULTS AT SOME LATER TIME.

THERE IS NO CONFLICT BETWEEN TWO OR MORE TRANSFERS WITHIN A SINGLE COMPOUND TRANSFER WHEN THEY ALL REFER TO THE SAME ELEMENT AS THEIR DATA SOURCE, AND ONE TRANSFER REFERS TO THE SAME ELEMENT AS ITS DATA DESTINATION. EACH TRANSFER WILL USE THE VALUE FOUND IN THE ELEMENT AT THE INITIATION OF THE COMPOUND TRANSFER. THE ELEMENT WILL NOT HAVE ITS VALUE CHANGED UNTIL THE RESULTS HAVE BEEN COMPUTED FOR ALL THE OTHER TRANSFERS.

THE FORMS ALLOWED FOR DIRECTLY CONNECTED AND BUSED TRANSFERS DIFFER SOMEWHAT. EACH WILL BE DESCRIBED BELOW.

A5.3.4.1 DIRECTLY CONNECTED TRANSFERS

A TRANSFER ALONG A DIRECTLY CONNECTED DATA PATH HAS THE GENERAL FORM:

```
<DESTINATION> < <SOURCE>
```

WHERE <DESTINATION> SPECIFIES THOSE HARDWARE ELEMENTS

	WHICH ARE TO RECIEVE THE BIT STRING GENERATED BY <SOURCE>.
<SOURCE>	SPECIFIES THE HARDWARE ELEMENTS WHICH CONTAIN THE DATA VALUES TO BE TRANSFERED, AND ANY OPERATIONS TO BE PERFORMED UPON THOSE VALUES.

THE QUANTITY <DESTINATION> MAY HAVE ANY OF THE FOLLOWING FORMS:

$$\left\{\begin{array}{c} \left[\text{<COMPRESSION>}\right]\left\{\begin{array}{c}\text{<REGISTER>}\\ \text{<LIGHTS>}\end{array}\right\} \\ \text{<SCALAR>} \\ \left\{\begin{array}{c}\text{<REGISTER>}\\ \text{<LIGHTS>}\end{array}\right\}(\ \text{<BIT>}\ ) \end{array}\right\} \left[,\ \left\{\begin{array}{c} \left[\text{<COMPRESSION>}\right]\left\{\begin{array}{c}\text{<REGISTER>}\\ \text{<LIGHTS>}\end{array}\right\} \\ \text{<SCALAR>} \\ \left\{\begin{array}{c}\text{<REGISTER>}\\ \text{<LIGHTS>}\end{array}\right\}(\ \text{<BIT>}\ ) \end{array}\right\}\right]$$

WHERE	<REGISTER>	IS THE NAME OF A REGISTER.
	<LIGHTS>	IS THE NAME OF A SET OF LIGHTS.
	<COMPRESSION>	IS A VALID COMPRESSION OPERATION ON A REGISTER OR LIGHTS. WHEN A COMPRESSION IS USED IN A DESTINATION, IT MERELY SPECIFIES THOSE BITS WHICH ARE TO ACCEPT NEW DATA VALUES. ANY REMAINING BITS IN THE DESTINATION

ARE NOT MODIFIED.

NOTE: CURRENT IMPLEMENTATION OF SDSS PERMITS ONLY SINGLE COMPRESSIONS IN DATA DESTINATIONS. MULTIPLE COMPRESSIONS ARE NOT VALID.

<BIT>	IS A BIT SELECTION ON THE LIGHTS OR REGISTER.
<SCALAR>	IS THE NAME OF A SCALAR ELEMENT.

THE COMMA (,) ABOVE INDICATES CONCATENATION. FOR EXAMPLE, TWO REGISTERS MAY BE CONCATENATED TOGETHER TO FORM A SINGLE DESTINATION. EACH COMPONENT IN A CONCATENATED DESTINATION IS TREATED INDEPENDENTLY OF THE OTHER. THAT IS, ANY ELEMENT ON THE LEFT-HAND SIDE OF THE SPECIFICATIONS ABOVE MAY BE CONCATENATED WITH ANY ELEMENT ON THE RIGHT-HAND SIDE.

GIVEN THE FOLLOWING SYSTEM DEFINITION STATEMENTS:

```
REGISTER A(10), B(5)
SCALAR L
LIGHTS LGS (3)
```

THEN THE FOLLOWING ARE VALID DESTINATIONS:

A	SPECIFIES 10 BIT DESTINATION
L	1
A,B	15
$W(3)/A,B	8
L,$W(3,5)/B	4
LGS,L	4

A DIRECTLY CONNECTED TRANSFER MAY HAVE A DATA SOURCE GIVEN BY ONE OF THE FOUR FORMS GIVEN IN FIGURE A3. NOTE THAT TERM DEFINITIONS APPLY TO ALL FOUR FORMS.

Form 1

$$\left[\text{\$NOT}\right]\left\{\begin{array}{c} \left[\langle\text{COMPRESSION}\rangle\right]\left\{\begin{array}{l}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\end{array}\right\} \\ \langle\text{SCALAR}\rangle \\ \langle\text{CONSTANT}\rangle \\ \left\{\begin{array}{l}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\end{array}\right\}\left(\langle\text{BIT}\rangle\right)\end{array}\right\} \left[\;,\left[\text{\$NOT}\right]\left\{\begin{array}{c} \left[\langle\text{COMPRESSION}\rangle\right]\left\{\begin{array}{l}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\end{array}\right\} \\ \langle\text{SCALAR}\rangle \\ \langle\text{CONSTANT}\rangle \\ \left\{\begin{array}{l}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\end{array}\right\}\left(\langle\text{BIT}\rangle\right)\end{array}\right\}\right]$$

Form 2

$$\text{\$}\left\{\begin{array}{l} \left\{\begin{array}{l}\text{SL}\\ \text{SR}\end{array}\right\}\left(\langle\#\text{S}\rangle\;,\langle\text{BIT-IN}\rangle\right) \\ \left\{\begin{array}{l}\text{RL}\\ \text{RR}\end{array}\right\}\left(\langle\#\text{R}\rangle\right)\end{array}\right\} \left\{\begin{array}{c}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\\ \langle\text{CONSTANT}\rangle\\ \langle\text{SCALAR}\rangle\end{array}\right\}\;,\;\left\{\begin{array}{c}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\\ \langle\text{CONSTANT}\rangle\\ \langle\text{SCALAR}\rangle\end{array}\right\}$$

FIGURE A3

VALID SOURCE SYNTAX FOR DIRECTLY-CONNECTED DATA TRANSFERS

Form 3

$$\left[\text{\$NOT}\right]\left\{\begin{array}{c}\left[\langle\text{COMPRESSION}\rangle\right]\left\{\begin{array}{c}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\end{array}\right\}\\ \langle\text{SCALAR}\rangle\\ \langle\text{CONSTANT}\rangle\\ \left\{\begin{array}{c}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\end{array}\right\}(\langle\text{BIT}\rangle)\end{array}\right\}\left\{\begin{array}{c}\text{\$AND}\\ \text{\$NND}\\ \text{\$OR}\\ \text{\$NOR}\\ \text{\$XOR}\end{array}\right\}\left[\text{\$NOT}\right]\left\{\begin{array}{c}\left[\langle\text{COMPRESSION}\rangle\right]\left\{\begin{array}{c}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\end{array}\right\}\\ \langle\text{SCALAR}\rangle\\ \langle\text{CONSTANT}\rangle\\ \left\{\begin{array}{c}\langle\text{REGISTER}\rangle\\ \langle\text{SWITCHES}\rangle\end{array}\right\}(\langle\text{BIT}\rangle)\end{array}\right\}$$

Form 4

$$\left[\text{\$NOT}\right]\ \left[\langle\text{COMPRESSION}\rangle\right]\ \langle\text{FUNCTION}\rangle\ (\ \langle\text{ARG}\rangle\ \left[,\ \langle\text{ARG}\rangle\ ,\ \ldots\right]\ )$$

FIGURE A3 (Continued)

TABLE A1

DEFINITION OF TERMS USED IN FIGURE A3

$NOT	SPECIFIES A BIT-BY-BIT LOGICAL NEGATION OF THE SOURCE BITS AFTER ANY BIT SELECTION OR COMPRESSION HAS TAKEN PLACE.
<COMPRESSION>	IS A VALID COMPRESSION OF A REGISTER OR SET OF SWITCHES.
<REGISTER>	IS THE NAME OF A REGISTER.
<SCALAR>	IS THE NAME OF A SCALAR.
<CONSTANT>	IS A CONSTANT GENERATED BY A CONSTANT GENERATOR.
<BIT>	SPECIFIES A DESIRED BIT IN A REGISTER OR SWITCHES.
<#S>	SPECIFIES THE NUMBER OF SINGLE BIT SHIFTS TO BE PERFORMED. THE DIRECTION OF THE SHIFT IS DETERMINED BY THE OPERATOR $SL (FOR LEFT SHIFT) OR $SR (FORR RIGHT SHIFT).
<BIT-IN>	SPECIFIES THE VALUE OF THE BIT WHICH WILL BE FED INTO THE VACATED POSITION FOLLOWING A SHIFT. THE VALUE OF <BIT-N> MAY BE EITHER 0 OR 1.

TABLE A1 (CONTINUED)

<#R>	SPECIFIES THE NUMBER OF SINGLE BIT ROTATIONS TO BE PERFORMED UPON THE DATA SOURCE. THE DIRECTION OF THE ROTATION IS GIVEN BY #RL (FOR ROTATE LEFT) OR #RR (FOR ROTATE RIGHT).
<FUNCTION>	IS THE NAME OF A FUNCTION.
<ARG>	IS THE NAME OF AN ARGUMENT TO THE FUNCTION. IT MAY BE THE NAME OF A REGISTER, SWITCHES, OR SCALAR. AT LEAST ONE ARGUMENT MUST BE GIVEN.

FORM 1 IS USED TO MOVE DATA FROM ONE HARDWARE ELEMENT TO ANOTHER WITH NO OPERATIONS OTHER THAN NEGATIONOR COMPRESSION. EACH OPERAND IN A CONCATENATED SOURCE IS TREATED INDEPENDENTLY OF THE OTHER IN FORMING THE SOURCE BIT STRING.

FORM 2 IS USED TO PROVIDE SHIFTING AND ROTATION OF A DATA STRING. IF A REGISTER, SWITCHES, OR CONSTANT IS SPECIFIED, THEN ALL BITS OF THE ELEMENT ARE USED IN THE SHFITING OR ROTATION.

FORM 3 APPLIES A BOOLEAN OPERATOR TO THE TWO BIT STRINGS SPECIFIED. THE TWO OPERANDS MUST HAVE EQUAL LENGTHS EXCEPT FOR THE SPECIAL CASE IN WHICH ONE OPERAND IS EXACTLY ONE BIT LONG. IN THIS CASE, THE SINGLE BIT IS EXPANDED TO THE SIZE OF THE MULTIPLE-BIT ELEMENT PRIOR TO THE BOOLEAN OPERATOR BEING APPLIED. NEGATION, IF SPECIFIED, IS APPLIED PRIOR TO THE BOOLEAN OPERATION.

FORM 4 IS USED TO REFERENCE A LOGICAL FUNCTION. AS MANY ARGUMENTS AS NEEDED MAY BE SUPPLIED IN A FUNCTION.

IT IS IMPOSSIBLE TO MODIFY THE ARGUMENTS OF A FUNCTION WITHIN AN EXTERNAL FUNCTION SUBPROGRAM. ONLY THE DESTINATION ELEMENT(S) WILL BE MODIFIED BY A FUNCTION REFERNCE.

EACH DIRECT TRANSFER MUST BE MADE ALONG A DATA PATH DEFINED BY A 'CONNECT' STATEMENT. IN ORDER TO DETERMINE IF A TRANSFER CAN BE MADE, THE FOLLOWING PROCEDURE MAY BE FOLLOWED:

1) IF THE TRANSFER CONTAINS A FUNCTION REFERENCE, THEN EACH ARGUMENT MUST BE CONNECTED TO THE FUNCTION, AND THE FUNCTION NAME MUST BE CONNECTED TO THE DESTINATION(S) ELEMENTS.

2) FOR ANY OTHER TRANSFERS, REMOVE ALL COMPRESSIONS, BIT SELECTIONS, NEGATIONS, AND BIT-IN SPECIFICATIONS; REPLACE ALL CONSTANTS BY THE CHARACTER "0". THE REMAINING DATA SOURCE AND DESTINATION SPECIFICATIONS MUST APPEAR TOGETHER IN A "CONNECT" DATA PATH IN ORDER FOR THE TRANSFER TO BE VALID.

FOR EXAMPLE, GIVEN THE TRANSFER:

```
$W(3,5)/B,C(10) < $A(10)/D $AND $NOT B
```

APPLYING STEPS 1 AND 2 ABOVE RESULTS IN THE REDUCTION OF THIS TRANSFER TO ONE OF THE FORM:

```
B,C < D $AND B
```

THUS, THE ORIGINAL TRANSFER REQUIRES THE "CONNECT" DATA PATH:

```
CONNECT ( D $AND B;B,C )
```

RECALL THAT THE DATA SOURCE IN A "CONNECT" DATA PATH IS ON THE LEFT OF THE ";", AND THE DESTINATION IS TO THE RIGHT.

THERE ARE TWO SPECIAL DIRECTLY CONNECTED TRANSFERS WHICH ARE USED TO REFERENCE A MEMORY. THEY ARE:

```
<MEMORY> $DCD <MARREG> < <MORREG>
<MORREG> < <MEMORY> $DCD <MARREG>
```

THE FORMER TRANSFER DEPOSITS THE QUANTITY IN THE <MDRREG> REGISTER INTO THE MEMORY AT THE LOCATION GIVEN BY THE CONTENTS OF THE <MARREG> REGISTER. THE LATTER TRANSFER FETCHES THE CONTENTS OF MEMORY LOCATION GIVEN BY THE CONTENTS OF THE <MARREG> REGISTER AND DEPOSITS IT INTO THE <MDRREG> REGISTER. OF COURSE, A MEMORY DEPOSIT IS ILLEGAL FOR A READ-ONLY MEMORY.

THESE TRANSFERS ARE ALONG A DIRECTLY-CONNECTED DATA PATH WHICH WAS IMPLICITLY DEFINED BY A "RAM" OR "ROM" STATEMENT. THE NAMES OF <MARREG> AND <MDRREG> MUST BE THE SAME AS WERE ORIGINALLY DESIGNATED ON THE "RAM"/"ROM" STATEMENT. USE OF ANY OTHER REGISTER, OR ANY OTHER ELEMENT NAME IS ILLEGAL.

THESE TWO TRANSFERS ARE THE ONLY CONTROL SEQUENCE STATEMENTS IN WHICH THE NAME OF A MEMORY MAY APPEAR, AND THE ONLY STATEMENTS IN WHICH THE MEMORY REFERENCE OPERATOR "$DCD" MAY BE USED.

EVERY TRANSFER STATEMENT ALONG A DIRECTLY CONNECTED DATA PATH MUST SPECIFY EXACTLY THE SAME NUMBER OF BITS IN THE DATA DESTINATION AS IT DOES IN THE DATA SOURCE. ALL COMPRESSIONS, CONCATENATIONS, AND OPERATIONS ON THE DATA ARE PERFORMED PRIOR TO THE COMPARISON OF THE LENGTHS. NOTE THAT THE COMPRESSION OPERATION MAY BE USED TO MATCH THE SIZES OF THE SOURCE AND DESTINATION.

THE CONTENTS OF HARDWARE ELEMENTS USED AS DATA SOURCES ARE NOT MODIFIED UNLESS THAT ELEMENT IS ALSO USED AS THE DATA DESTINATION. IF AN ELEMENT USED AS A DESTINATION HAS ONLY A PORTION OF ITS BITS SELECTED, THEN ONLY THOSE BITS WILL BE MODIFIED.

ANY ELEMENT MAY BE NEGATED IN A TRANSFER WITHOUT HAVING TO SPECIFY THE NEGATION HARDWARE IN A DATA PATH DEFINITION. THIS IS BECAUSE THE NEGATION OF A BIT STRING IS ALMOST ALWAYS AVAILABLE FOLLOWING MOST DIGITAL OPERATIONS. THUS, IT IS NOT EXPLICITLY DECLARED.

A SINGLE HARDWARE ELEMENT MAY BE CONCATENATED WITH ITSELF WHEN USED AS A DATA SOURCE. IT MAY NOT BE CONCATENNATED WITH ITSELF WHEN USED AS A DATA DESTINATION.

THE FOLLOWING ARE EXAMPLES OF VALID DIRECTLY CONNECTED TRANSFERS. NOTE THAT ALL HARDWAE ELEMENTS AND DATA PATHS ARE DEFINED IN THIS EXAMPLE.

```
REGISTER A(10), B(5), C(10)
SCALAR L
CONNECT ( A;B ), ( A,B;C ), ( $SL(2)A;A ),
*  (A $AND B; A ), (A $OR L; B )
```

THE FOLLOWING TRANSFERS ARE VALID:

```
B < $W(5) / A
B(4) < A(0)
C < $A(5) / A,B
$W(5)/C < $A(5) / A $AND $NOT B;   A < $SL(2) A
```

THE FOLLOWING TRANSFERS ARE ILLEGAL.

```
A < B
A < $SL(1) A
B(1) < L
```

IN THESE THREE CASES, THERE IS NO DATA PATH ALONG WHICH THE TRANSFER CAN BE MADE.

A5.3.4.2 BUSED TRANSFERS

A BUSED TRANSFER IS A TRANSFER WHICH UTILIZES A BUS ALONG AT LEAST ONE PORTION OF ITS DATA PATH. A BUSED TRANSFER MAY BE THOUGHT OF AS CONSISTING OF A SERIES OF MICRO-TRANFERS, ALL OF WHICH TAKE PLACE SIMULTANEOUSLY.

SDSS REQUIRES THE COMPLETELY-BUSED STRUCTURE FOR ALL BUSED TANSFERS; THAT IS, EACH MICRO-TRANSFER MUST SPECIFY A BUS (OR A CONCATENATED PAIR OF BUSES) FOR EITHER ITS DATA SOURCE, OR DATA DESTINATION, OR BOTH. THUS, EACH MICRO-TRANSFER EITHER PLACES DATA ONTO A BUS, OR EXTRACTS DATA FROM A BUS.

THE GENERAL FORM OF A BUSED MICRO-TRANSFER IS THE SAME AS FOR A DIRECTLY-CONNECTED TRANSFER:

```
<DESTINATION> < <SOURCE>
```

THE QUANTITIES ALLOWED FOR <DESTINATION> ARE IDENTICAL TO THOSE ALLOWED FOR THE DATA DESTINATION IN A DIRECTLY CONNECTED TRANSFER. IN ADDITION, <DESTINATION> MAY ALSO BE:

$$\left\{\begin{array}{c}\text{<BUS>}\\ \text{<OP>}\\ \text{<OP> (<ID>)}\\ \text{<BUS>, <BUS>}\\ \text{<FUNCTION>}\end{array}\right\}$$

WHERE <FUNCTION>	IS THE NAME OF A LOGICAL FUNCTION. THIS IMPLIES THAT SOME ARGUMENT, LOCATED ON A BUS, IS BEING SUPPLIED TO THE FUNCTION BY THIS TRANSFER.
<OP>	IS A BOOLEAN OPERATOR: $AND, $NND, $OR, $NOR, $XOR. THIS IMPLIES THAT SOME OPERAND, LOCATED ON A BUS, IS BEING SUPPLIED TO THE OPERATOR AS INPUT DATA.
<BUS>	IS THE NAME OF A DATA BUS. THIS SIMPLY MEANS THAT DATA IS TO BE PLACED ONTO THE BUS.
<ID>	IS AN INTEGER IN THE RANGE OF 1 TO 255, INCLUSIVE. SHOULD MORE THAT ONE BOOLEAN OPERATION OF A GIVEN TYPE (FOR EXAMPLE, TWO 'AND'S) BE REQUIRED IN A SINGLE COMPOUND TRANSFER STATEMENT, THIS <ID> VALUE IS USED TO DISTINGUISH ONE OPERATOR FROM THE OTHER. FOR EXAMPLE, GIVEN THE TRANSFERS:

```
$AND(1) < A; $AND(2) < B; $AND(1) < C;
             $AND(2) < D
```

THEN THE RESULTS OBTAINED WILL BE:

A $AND C   B $AND D

AN OPERATOR NAME APPEARING WITHOUT THE <ID> TERM IS DIFFERENT FROM ALL OPERATORS WITH THE <ID> TERM. THERE IS NO POSSIBILITY OF CONFLICT BETWEEN BUSED BOOLEAN OPERATORS AND DIRECTLY CONNECTED OPERATORS.

IN A MICRO-TRANSFER THAT PLACED DATA ONTO A BUS, THE SOURCE BIT STRING IS ALIGNED SO THAT ITS LEAST SIGNIFICANT BIT IS PLACED INTO THE LEAST SIGNIFICANT BIT OF THE BUS. ANY EXCESS BITS IN THE SOURCE STRING THAT CAN NOT FIT ONTO THE BUS ARE LOST. ANY EXCESS BITS ON THE BUS THAT ARE NOT EXPLICITLY SET BY THE DATA STRING ARE AUTOMATICALLY SET TO ZEROS. IT IS NOT POSSIBLE TO COMPRESS A BUS TO SELECT WHCIH BIT POSITIONS ARE TO RECIEVE DATA VALUES. ALL BIT POSITIONS AR THE RECIEVING BUS ARE USED IN THE TRANSFER.

THE QUANTITY <SOURCE> FOR A MICRO-TRANSFER MAY BE ANY OF THE <SOURCE> SPECIFICATIONS GIVEN BY FORM 1 AND FORM 2 (FIGURE A3) OF A DIRECTLY CONNECTED TRANSFER. IN ADDITION, THE FOLLOWING ARE ALLOWED:

$$\left\{\begin{array}{l} \left[\text{\$NOT}\right] \left\{\begin{array}{l} \left[\text{<COMPRESSION>}\right] \left\{\begin{array}{c}\text{<BUS>}\\ \text{<FUNCTION>}\end{array}\right\} \\ \text{<BUS> (<BIT>)} \\ \text{<OP>} \left[\text{(<ID>)}\right] \end{array}\right\} \\ \text{\$} \left\{\begin{array}{l} \left\{\begin{array}{c}\text{SL}\\ \text{SR}\end{array}\right\} \text{(<\#S>} \left[\text{, <BIT-IN>)}\right] \\ \left\{\begin{array}{c}\text{RL}\\ \text{RR}\end{array}\right\} \text{(<\#R>)} \end{array}\right\} \text{<BUS>} \left[\text{, <BUS>}\right] \end{array}\right\}$$

WHERE <COMPRESSION>	IS A VALID COMPRESSION OF THE BUS. IF THE BUS CONTAINS MORE THAN 32 BITS, THEN ONLY "$A" AND "$W" CONSTANTS MAY BE USED IN A COMPRESSION OF THAT BUS. (SEE SECTION A5.3.1)
<FUNCTION>	IS THE NAME OF A LOGICAL FUNCTION. THIS IMPLIES THAT THE OUTPUT OF THE FUNCTION HAS BEEN COMPUTED, AND IS NOW AVAILABLE TO BE TRANSFERED TO SOME DESTINATION.
<BUS>	IS THE NAME OF A BUS.
<BIT>	IS A BIT SELECTION FROM THE BUS.
<#S>	IS THE NUMBER OF SINGLE BIT SHIFTS TO BE PERFORMED.
<BIT-IN>	IS THE BIT TO BE FED INTO THE POSITION(S)

	VACATED BY A SHIFT.
<#R>	IS THE NUMBER OF SINGLE BIT ROTATIONS TO BE PERFORMED.
<OP>	IS THE NAME OF A BOOLEAN OPERATOR, AND IS ONE OF: $AND, $NND, $OR, $NOR, $XOR. THE USE OF A BOOLEAN OPERATOR AS A DATA SOURCE MEANS THAT THE OPERATOR IS TO BE APPLIED TO THE TWO ARGUMENTS ALREADY SUPPLIED TO THE OPERATOR BY PREVIOUS TRANSFERS WITHIN THIS COMPOUND TRANSFER STATEMENT. THE RESULT OF THE OPERATOR IS TO BE USED AS THE DATA SOURCE FOR THE TRANSFER.
<ID>	IS AN IDENTIFICATION NUMBER IN THE RANGE OF 1 TO 255, INCLUSIVE, WHICH SPECIFIES WHICH BOOLEAN OPERATOR IS TO BE USED. SHOULD TWO OR MORE BOOLEAN OPERATORS OF THE SAME TYPE (FOR EXAMPLE, TWO $OR'S) BE REQUIRED IN ONE COMPOUND TRANSFER STATEMENT, THIS VALUE DISTINGUISHES ONE FROM ONE ANOTHER. SEE THE EXAMPLE IN THE DISCUSSION OF DESTINATIONS FOR BUSED TRANSFERS ABOVE.

IF A SHIFT OR ROTATION OF A BUS IS REQUESTED, THE ENTIRE BUS PARTICIPATES IN THE SHIFT OR ROTATION. BUSES AND THE RESULTS FROM LOGICAL FUNCTIONS MAY BE COMPRESSED, IF DESIRED, TO SELECT CERTAIN BIT POSITIONS FROM THE BUS OR THE FUNCTION RESULT.

TO PERFORM A BOOLEAN OPERATION, EACH OF 2 OPERANDS MUST BE PLACED ON SEPARATE BUSES. THE CONTENTS OF EACH

BUS ARE THEN TRANSFERED TO THE BOOLEAN OPERATOR. THE RESULT OF THE BOOLEAN OPERATION (DENOTED BY THE NAME OF THE OPERATOR) IS THEN TRANSFERED TO A THIRD BUS. THIS THIRD BUS MAY THEN BE TRANSFERED TO THE FINAL DESTINATION.

TO INVOKE A LOGICAL FUNCTION, EACH ARGUMENT MUST BE PLACED ONTO A SEPARATE BUS. THE CONTENTS OF EACH OF THESE BUSES MUST THEN BE TRANSFERED TO THE FUNCTION. THE RESULT OF THE FUNCTION (DENOTED BY THE NAME OF THE FUNCTION ITSELF) MUST NOW BE TRANSFERED TO AN OUTPUT BUS. THE CONTENTS OF THIS OUTPUT BUS MAY NOW BE TRANSFERED TO THE FINAL DESTINATION

SINCE EACH MICRO-TRANSFER IS ONLY A SINGLE COMPONENT OF AN OVERALL MACRO-TRANSFER, AT LEAST TWO MICRO-TRANSFERS ARE NECESSARY TO PERFORM A MACRO-TRANSFER. EACH BUSED TRANSFER MUST BE EXPRESSED AS A COMPOUND TRANSFER. ALL MICRO-TRANSFERS IN ONE COMPOUND TRANSFER STATEMENT ARE ASSUMED TO OCCUR SIMULTANEOUSLY.

THE FOLLOWING RESTRICTIONS MUST BE ADHERED TO WHEN USING MICRO-TRANSFERS:

1) EVERY BUSED MICRO-TRANSFER MUST SPECIY A BUS AS ITS DATA SOURCE, ITS DATA DESTINATION, OR BOTH.
2) THE SET OF MICRO-TRANSFERS WHICH COMPOSE A MACRO-TRANSFER MUST BE WRITTEN IN THE COMPOUND TRANSFER SUCH THAT, AS THE TRANSFERS ARE SCANNED FROM LEFT TO RIGHT, EVERY BUS THAT IS USED AS A DATA SOURCE HAS ALREADY BEEN ASSIGNED A VALUE BY A PREVIOUS TRANSFER.
3) ANY ELEMENT MAY BE USED AS A DESTINATION IN ONLY ONE MICRO-TRANSFER IN A COMPOUND STATEMENT. A BUS MAY BE USED AS A DATA SOURCE ANY NUMBER OF TIMES ONCE IT HAS BEEN ASSIGNED A VALUE.

4) A BUS WILL RETAIN WHATEVER VALUE IS PLACED UPON THAT BUS FOR THE DURATION OF THE COMPOUND TRANSFER. THAT VALUE IS LOST AT THE COMPLETION OF THE COMPOUND TRANSFER; IT CAN NOT BE RETAINED PAST THE SINGLE COMPOUND STATEMENT.

5) IF TWO BUSES ARE CONCATENNATED, THEIR TOTAL LENGTH CAN NOT EXCEED 64 BITS. A BUS MAY BE CONCATENATED WITH ITSELF WHEN USED AS A DATA SOURCE; IT MAY NOT BE CONCATENATED WITH ITSELF WHEN IT IS USED AS A DATA DESTINATION.

6) THE DATA SOURCE AND DESTINATION IN A SINGLE MICRO-TRANSFER MAY NOT BOTH BE CONCATENATED BUSES. ONLY THE SURCE, OR THE DESTINATION MAY BE CONCATENATED. THE DEFINITION OF CONNECTIONS TO AND FROM BUSES PROHIBIT SUCH CONCATENATIONS.

IT IS LEGAL TO COMBINE MICRO-TRANSFERS AND DIRECTLY-CONECTED TRANSFERS IN THE SAME COMPOUND TRANSFER STATEMENT.

THE FOLLOWING SEGMENT OF AN SDSS DESCRIPTION IS GIVEN TO ILLUSTRATE VARIOUS BUSED TRANSFERS. THE DESCRIPTION IS BASED UPON THE HARDWARE DIAGRAM SHOWN IN FIGURE A4.

```
      REGISTER A(10), B(10)
      SCALAR L
      BUS ABUS(10),(IN=A,L),(OUT=CBUS,$AND,ADD,
     *                          $SL(1)CBUS )
      BUS BBUS(10),(IN=B),(OUT=CBUS,$AND,ADD)
      FUNCTION ADD(2,11)
      BUS CBUS(11),(OUT=(L,A),B),(IN=BBUS,ABUS,
     *                                 $AND, ADD )
C
C        TO MOVE THE CONTENTS OF A TO B....
C
      ABUS < A; CBUS < ABUS; B < CBUS
C
C        TO ADD A AND B TO GIVE RESULT INTO (L,A)....
C
      ABUS < A;  BBUS < B;  ADD < ABUS;  ADD < BBUS;
     *CBUS < ADD;  L,A < CBUS
C
C        TO SHIFT A LEFT 1 BIT, FEED IN A '1', AND PUT
C     RESULT INTO A....
C
      ABUS < A;  CBUS < $SL(1,1) ABUS;  A < CBUS
C
C        THE FOLLOWING IS AN INVALID SEQUENCE OF
C     MICRO-TRANSFERS ATTEMPTING TO 'AND' A AND NOT-B
C     TOGETHER, AND PLACE THE RESULT INTO A.  THE BUS
C     'ABUS' IS REFERENCED PRIOR TO BEING ASSIGNED A
C     VALUE.  IF THE 2ND AND 3RD TRANSFERS WERE
C     EXCHANGED, THE SEQUENCE WOULD BE CORRECT.
C
      BBUS < $NOT B;  $AND < ABUS;  ABUS < A;
      *$AND < BBUS;  CBUS < $AND;  A < CBUS;
C
      END
```

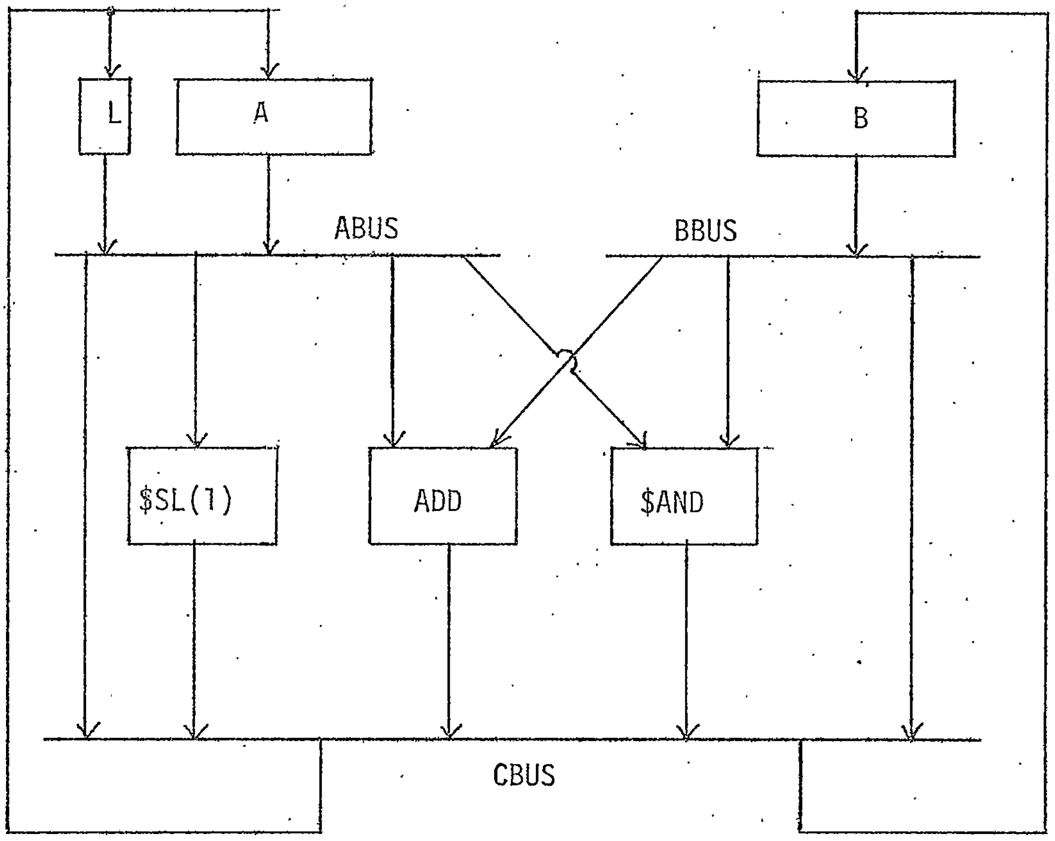


FIGURE A4

HARDWARE DIAGRAM FOR BUSED TRANSFERS

EVERY BUSED MICRO-TRANSFER MAY SPECIFY AN OPERATION ON THE DATA. THE OPERATION MUST HAVE BEEN SPECIFIED AS LEGAL IN A 'BUS' STATEMENT. (SEE SECTION A5.1.8.2 AND THE EXAMPLE ABOVE.)

EVERY BUSED MICRO-TRANSFER MUST TAKE PLACE ALONG A DATA PATH THAT WAS DEFINED VIA A 'BUS' STATEMENT. THE FOLLOWING PROCEDURE MAY BE USED TO DETERMINE IF A MICRO-TRANSFER HAS A VALID DATA PATH FOR THE TRANSFER:

1) REMOVE ALL COMPRESSIONS, BIT SELECTIONS, NEGATIONS, AND BIT-IN SPECIFICATIONS. REPLACE ANY CONSTANTS WITH THE CHARACTER '0'.
2) IF THE DATA DESTINATION IS A BUS, THEN THE DATA SOURCE MUST BE SPECIFIED IN AN 'IN=' CONNECTION FOR THAT BUS.
3) IF THE DATA SOURCE IS A BUS, THEN THE DATA DESTINATION MUST BE SPECIFIED AS AN 'OUT=' CONNECTION FOR THAT BUS.

A5.3.5 BRANCH STATEMENTS

BRANCH STATEMENTS ARE USED TO MODIFY THE SEQUENCE OF MACHINE OPERATIONS DURING THE SIMULATION OF A DIGITAL SYSTEM. A BRANCH STATEMENT MAY BE UNCONDITIONAL, MEANING THE BRANCH WILL BE TAKEN, OR CONDITIONAL, MEANING THE BRANCH IS DETERMINED BY VALUES WITHIN THE SYSTEM AT THE TIME THE BRANCH STATEMENT IS ENCOUNTERED.

A BRANCH STATEMENT MAY HAVE A LABEL. ONLY THOSE BRANCH STATEMENTS WHICH ARE A TARGET OF ANOTHER BRANCH STATEMENT ARE REQUIRED TO BE LABELED. IT IS NOT POSSIBLE FOR A BRANCH STATEMENT TO CAUSE A BRANCH TO ITSELF.

THERE ARE TWO FORMS OF BRANCH STATEMENTS AVAILABLE IN SDSS. THEY WILL BE DESCRIBED BELOW.

A5.3.5.1 UNCONDITIONAL BRANCH

THE UNCONDITIONAL BRANCH STATEMENT HAS THE FORM:

```
> <LABEL>
```

WHERE <LABEL> IS A VALID STATEMENT LABEL.

THE CONTROL SEQUENCE OPERATION WHICH IS PERFORMED FOLLOWING THIS BRANCH STATEMENT IS THE ONE LABELED BY <LABEL>.

IN ORDER TO BE ABLE TO EXECUTE THE SDSS STATEMENT FOLLOWING THE UNCONDITIONAL BRANCH, THE NEXT STATEMENT MUST BE LABELED OR BE AN 'INTERRUPT' STATEMENT.

EXAMPLE OF A VALID UNCONDITIONAL BRANCH:

```
> 10
```

A5.3.5.2 THREE-WAY CONDITIONAL BRANCH

THE THREE-WAY CONDITIONAL BRANCH ALLOWS TRANSFER OF CONTROL TO ONE OF THREE POSSIBLE STATEMENTS. THE FORM OF THIS BRANCH STATEMENT IS AS FOLLOWS:

```
 (                        ( <REG> )  )
 | [<RED>] [<COMP>]  <            >  |
 |                        ( <SWS> )  |
<          <SCALAR>                   >  :  <CONST> > <L1>,<L2>,<L3>
 |   ( <REG> )                       |
 |  <         >   ( <BIT> )          |
 (   ( <SWS> )                       )
```

WHERE	<REG>	IS THE NAME OF A REGISTER.
	<SWS>	IS THE NAME OF A SET OF SWITCHES.
	<SCALAR>	IS THE NAME OF A SCALAR.
	<COMP>	IS A SINGLE OR MULTIPLE COMPRESSION OF THE REGISTER OR SWITCHES.
	<RED>	IS A REDUCTION DONE ON EITHER A REGISTER, SWITCHES, OR ON THE COMPRESSION OF A REGISTER OR SWITCHES.
	<BIT>	SPECIFIES A BIT SELECTION FROM THE REGISTER OR SWITCHES.
	<CONST>	IS ANY VALID CONSTANT AS DEFINED IN SECTION A4. ALL CONSTANTS GENERATED BY THE CONSTANTS GENERATORS MUST HAVE A LENGTH SPECIFIED.

<L1>
<L2>
<L3>   SPECIFY VALID STATEMENT LABELS.

THE EXECUTION OF THE THREE-WAY BRANCH IS AS FOLLOWS: THE VALUE OF THE EXPRESSION TO THE LEFT OF THE COLON IS EVALUATED. THE RESULTING VALUE IS THEN COMPARED TO THE <CONST>. CONTROL THEN PASSES TO THE STATEMENT LABELED <L1>, <L2>, OR <L3>, IF THE EXPRESSION RESULT IS LESS THAN, EQUAL TO, OR GREATER THAN THE <CONST>.

THE CONTROL SEQUENCE STATEMENT FOLLOWING THE THREE-WAY BRANCH MUST HAVE A LABEL, OR BE AN 'INTERRUPT' STATEMENT IN ORDER THAT THAT STATEMENT BE ACCESSABLE.

EXAMPLES OF VALID THREE-WAY BRANCH STATEMENTS:

```
COLUMN
 678
  A : 1000 > 1, 2, 2
  $AND / B : 1 > 10, 11, 10
  L : 0 > 1, 2, 1
  $W(5)/C : $W(5,5) > 100, 104, 107
  A(6) : 0 > 20, 30, 34
```

A5.3.6 HALT STATEMENT

THE "HALT" STATEMENT IS USED TO TERMINATE OPERATION OF THE SYSTEM UNDER SIMULATION. THE "HALT" STATEMENT HAS THE FORM:

```
HALT [<TEXT>]
```

WHERE <TEXT> IS ANY CHARACTER STRING. WHEN THE "HALT" STATEMENT IS ENCOUNTERED IN THE COURSE OF THE SIMULATION, THE MESSAGE:

```
HALT AT LINE NNNN
```

IS PRINTED VIA THE M:LO DCB. NNNN IS THE LINE NUMBER IN WHICH THE "HALT" STATEMENT WAS ENCOUNTERED. IF <TEXT> IS PRESENT, IT IS PRINTED FOLLOWING THE ABOVE MESSAGE.

THE "HALT" STATEMENT MAY HAVE A LABEL.

A5.4 HOUSEKEEPING STATEMENTS

HOUSEKEEPING STATEMENTS ARE STATEMENTS WHICH COMMUNICATE WITH THE COMPILER DURING COMPILATION, OR SPECIFY ACTIONS THAT ARE TO TAKE PLACE. THESE ACTIONS ARE, GENERALLY, NOT PART OF THE CONTROL SEQUENCE OR SYSTEM DEFINITION, AND ARE NOT TO BE CONSIDERED AS SUCH.

HOUSEKEEPING STATEMENTS MAY BE PLACED AT ANY LOCATION FOLLOWING ALL SYSTEM DEFINITION STATEMENTS AND MEMORY INITIALIZATION ("FILL") STATEMENTS. A HOUSEKEEPING

STATEMENT WILL TERMINATE THE SYSTEM DEFINITION SECTION AND THE MEMORY INITIALIZATION SECTION, IF EITHER IS CURRENTLY IN PROGRESS.

HOUSEKEEPING STATEMENTS CONSIST OF THE FOLLOWING STATEMENT TYPES:

PRINT	- TO DISPLAY THE CONTENTS OF HARDWARE ELEMENTS.
INTERRUPT	- TO DEFINE THE BEGINNING OF AN INTERRUPT ROUTINE.
RETURN	- TO RETURN FROM AN INTERRUPT ROUTINE TO THE INTERRUPTED ROUTINE.
END	- TO INDICATE TO SDSS THAT THERE ARE NO MORE SYSTEM DESCRIPTION STATEMENTS TO READ.

EACH STATEMENT TYPE WILL BE DESCRIBED BELOW.

A5.4.1 PRINT STATEMENT

THE 'PRINT' STATEMENT IS USED TO DISPLAY THE CURRENT CONTENTS OF A REGISTER, SCALAR, SET OF LIGHTS, SET OF SWITCHES, OR PORTIONS OF A MEMORY. VALUES MAY BE DISPLAYED IN EITHER HEXIDECIMAL NOTATION OR DECIMAL (BASE-10) NOTATION. IF HEXIDECIMAL NOTATION IS CHOSEN, EACH VALUE WILL BE PRINTED AS A 32-BIT VALUE; ANY EXCESS HIGH-ORDER BITS NOT NEEDED BY THE VALUE WILL BE SET TO

ZEROS. IF DECIMAL NOTATION IS USED, THE VALUE WILL BE DISPLAYED AS A 10-DIGIT POSITIVE INTEGER. ANY HIGH-ORDER ZEROS WILL BE SUPPRESSED.

THE DESIGNER HAS NO CONTROL OVER THE OUTPUT FORMAT, EXCEPT FOR THE CHOICE OF NOTATION. THE OUTPUT PRODUCED BY THE "PRINT" STATEMENT IS FORMATED TO FIT A STANDARD 72 CHARACTER WIDE TERMINAL.

EACH "PRINT" STATEMENT WILL CAUSE THE MESSAGE:

```
VALUES AT LINE: NNNN
```

TO BE PRINTED PRIOR TO THE VALUES OF THE ELEMENTS DESIRED. NNNN IS THE LINE NUMBER OF THE "PRINT" STATEMENT ITSELF.

ALL OUTPUT PRODUCED BY THE "PRINT" STATEMENT IS WRITTEN THROUGH THE M:LO DCB.

THE "PRINT" STATEMENT HAS THE FORM:

```
        ┌   D   ┐
PRINT   │ (   ) │ <ELEMENT>, <ELEMENT>, ...
        │   X   │
        └       ┘
```

WHERE (D)	INDICATES THAT DECIMAL NOTATION IS TO BE USED.
(X)	INDICATES THAT HEXIDECIMAL NOTATION IS TO BE USED.
<ELEMENT>	IS THE NAME OF AN ELEMENT WHOSE VALUE IS TO BE DISPLAYED. <ELEMENT> MAY BE THE NAME OF A REGISTER, SWITCHES, SCALAR, LIGHTS, OR A MEMORY. IF A MEMORY IS SPECIFIED, THEN THE NAME MUST BE FOLLOWED BY A RANGE OF LOCATIONS WHICH ARE TO BE DISPLAYED. THIS RANGE HAS THE FORM:

<MEMORY> (<LO>, <HI>)

WHERE <LO> AND <HI> SPECIFY THE LOW AND HIGH ADDRESSES OF THE MEMORY SEGMENT TO BE DISPLAYED. BOTH MUST SPECIFY LOCATIONS WITHIN THE MEMORY, AND THE VALUE OF <HI> MUST NOT BE LESS THAN THAT OF <LO>.

IF NEITHER (D) OR (X) ARE SPECIFIED, HEXIDECIMAL NOTATION IS ASSUMED.

ANY NUMBER OF ELEMENTS AND MEMORY SEGMENTS MAY BE DISPLAYED BY ONE 'PRINT' STATEMENT. IF MORE THAN ONE ELEMENT IS TO BE DISPLAYED, THEN ALL WILL BE DISPLAYED IN THE SAME NUMBER BASE. IF NO ELEMENTS ARE SPECIFIED, THE STATEMENT WILL BE IGNORED.

EXAMPLES OF VALID 'PRINT' STATEMENTS:

```
COLUMN
  678
   PRINT A
   PRINT (D) A, B, MEMORY(0,100),  C
```

A5.4.2 END STATEMENT

THE 'END' STATEMENT IS USED TO INDICATE THE END OF ALL SOURCE STATEMENTS DESCRIBING THE SYSTEM. EVERY DESCRIPTION MUST CONTAIN 1, AND ONLY 1, 'END' STATEMENT. THE 'END' STATEMENT MAY NOT HAVE A LABEL. IF THE 'END' STATEMENT IS ENCOUNTERED DURING A SIMULATION , IT IS TREATED AS IF IT WERE A 'HALT' STATEMENT WITH NO TEXT STRING FOLLOWING.

THE 'END' STATEMENT HAS THE FOLLOWING FORM:

```
END [<LABEL>]
```

WHERE <LABEL> IS THE LABEL OF A STATEMENT. IF PRESENT, <LABEL> SPECIFIES THE FIRST STATEMENT WHICH IS TO BE EXECUTED WHEN SIMULATION BEGINS. IF <LABEL> IS OMITTED, OR IS GIVEN AS 0, THEN THE FIRST STATEMENT FOLLOWING ALL SYSTEM DEFINITION AND FILL STATEMENTS WILL BE EXECUTED FIRST.

A5.4.3 INTERRUPT STATEMENT

THE 'INTERRUPT' STATEMENT IS USED TO DEFINE THE BEGINNING OF AN INTERRUPT ROUTINE; THAT IS, A HARDWARE

ROUTINE WHICH WILL BE ENTERED UPON RECIEPT OF AN INTERRUPT FROM OUTSIDE THE DIGITAL SYSTEM.

THE 'INTERRUPT' STATEMENT MUST FOLLOW ALL SYSTEM DEFINITION STATEMENTS AND MEMORY INITIALIZATION (FILL) STATEMENTS.

THE INTERRUPT STATEMENT HAS THE FORM:

```
INTERRUPT <NUMBER>
```

WHERE <NUMBER> SPECIFIES A PARTICULAR INTERRUPT ROUTINE.

<NUMBER> MUST BE IN THE RANGE OF 1 TO 255, INCLUSIVE. EACH INTERRUPT NUMBER MUST BE SPECIFIED ONLY ONCE PER DESIGN. THIS NUMBER DISTINGUISHES EACH INTERRUPT ROUTINE FROM ALL OTHER INTERRUPT ROUTINES.

AN 'INTERRUPT' STATEMENT DEFINES THE BEGINNING OF A HARDWARE INTERRUPT-HANDLING ROUTINE. THIS ROUTINE CONSISTS OF ONE OR MORE CONTROL SEQUENCE STATEMENTS DEFINING THE ACTION TO BE TAKEN UPON RECIEPT OF THE INTERRUPT. THIS ACTION MAY BE AS SIMPLE AS SETTING A FLAG TO INDICATE THAT THE INTERRUPT HAS OCCURED, OR AS COMPLEX AS A ROUTINE TO HANDLE A POWER- FAILURE CONDITION.

IF THE CONTROL SEQUENCE STATEMENT JUST PRIOR TO THE 'INTERRUPT' STATEMENT DOES NOT CAUSE A BRANCH TO SOME OTHER PORTION OF THE CONTROL SEQUENCE, THEN CONTROL WILL FALL THROUGH INTO THE INTRRUPT HANDLING ROUTINE.

WHEN AN INTERRUPT IS REQUESTED (SEE SECTION A6), THE CONTROL SEQUENCE STATEMENT IMMEDIATELY FOLLOWING THE INTERRUPT STATEMENT WILL BE EXECUTED NEXT. THE SDSS STATEMENT WHICH WAS IN PROGRESS WHEN THE INTERRUPT WAS

RECIEVED WILL BE COMPLETED PRIOR TO THE INTERRUPT ROUTINE BEING ENTERED.

EXAMPLES OF VALID 'INTERRUPT' STATEMENTS:

```
COLUMN
  678
   INTERRUPT 0
   INTERRUPT 100
```

A5.4.4 RETURN STATEMENT

THE 'RETURN' STATEMENT IS USED TO RETURN TO THE CONTROL SEQUENCE STATEMENT THAT WOULD HAVE BEEN EXECUTED NEXT IF AN INTERRUPT HAD NOT BEEN RECIEVED. THAT IS, IT ALLOWS A RETURN TO THE MOST RECENTLY INTERRUPTED ROUTINE.

THE 'RETURN' STATEMENT MUST FOLLOW ALL SYSTEM DEFINITION STATEMENTS AND MEMORY INITIALIZATION STATEMENTS.

THE 'RETURN' STATEMENT HAS TWO FORMS:

```
FORM 1:  RETURN
FORM 2:  RETURN I
```

FORM 1 IS USED IF A RETURN TO THE MOST RECENTLY INTERRUPTED ROUTINE IS DESIRED. THE STATEMENT TO WHICH CONTROL IS RETURNED IS THE ONE FOLLOWING THE STATEMENT IN WHICH THE INTERRUPT WAS DETECTED.

EACH INTERRUPT CAUSES A RETURN LOCATION TO BE STORED INTO AN INTERNAL STACK. A MAXIMUM OF 20 INTERRUPTS MAY BE STACKED UP HERE. SHOULD IT NOT BE DESIRED TO RETURN TO THE MOST RECENTLY INTERRUPTED ROUTINE, BUT MERELY REMOVE ITS LOCATION FROM THE STACK, FORM 2 IS USED. CONTROL WILL THEN PROCEED WITH THE STATEMENT FOLLOWING THE 'RETURN' STATEMENT.

A6 COMPILATION AND SIMULATION PROCEDURES

THE COMPILER EXISTS AS A LOAD MODULE CALL 'SDSS' UNDER ACCOUNT 197. THERE IS NO PASSWORD. THE CURRENT IMPLEMENTATION SUPPORTS ONLINE OPERATION ONLY; ANY ATTEMPT TO COMPILE IN BATCH MODE WILL TERMINATE COMPILER OPERATION.

THE COMPILER ACCEPTS ALL SOURCE STATEMENTS THROUGH THE M:SI DCB. ALL SOURCE LISTINGS ARE WRITTEN THROUGH THE M:LO DCB. ERROR MESSAGES ARE WRITTEN THROUGH THE M:DO DCB. IF THE M:LO DCB IS ASSIGNED TO A FILE OR TO A DEVICE OTHER THAN THE TERMINAL, ANY ERROR MESSAGES WILL BE WRITTEN TO BOTH THE TERMINAL AND THE LO DEVICE/FILE. THE COMPILER OBJECT OUTPUT IS WRITTEN THROUGH THE M:GO DCB.

TO CALL THE COMPILER, THE STANDARD CP-V TEL COMMAND TO INITIATE ANY LOAD MODULE IS USED:

```
SDSS.197 <SOURCE-FILE> OVER <GO-FILE>  , <LISTING-OUTPUT>
```

THE SDSS COMPILER WILL NOW ASK FOR OPTIONS. THE LEGAL OPTIONS ARE:

LS - LIST SOURCE STATEMENTS
NS - DO NOT LIST SOURCE STATEMENTS
LD - LIST SYSTEM DEFINITION SYMMARY
ND - DO NOT LIST SYSTEM DEFINITION SYMMARY

THE DEFAULT OPTIONS ARE NS AND ND. IF NO OPTIONS ARE NECESSARY, SIMPLY TYPE A CARRAIGE RETURN.

IF THE DESIGNER SPECIFIES 'ME' FOR <SOURCE-FILE>, THE COMPILER WILL NOW PROMPT WITH A COLON AND WAIT FOR A SOURCE LINE.

SOURCE STATEMENTS WILL BE ACCEPTED UNTIL AN 'END' STATEMENT OR AN END-OF-FILE IS ENCOUNTERED. ANY FOLLOWING RECORDS WILL BE IGNORED. IF <SOURCE-FILE> WAS ASSIGNED TO 'ME', THEN IT IS NECESSARY TO TYPE AN END-OF-FILE ON THE TERMINAL FOLLOWING THE 'END' STATEMENT. THE END-OF-FILE CHARACTER IS AN ESCAPE-F COMBINATION.

IF NO ERRORS WERE DETECTED DURING COMPILATION, THE DESIGN MAY NOW BE TESTED BY SIMULATION.

TO PERFORM THE SIMULATION, THE OBJECT CODE PRODUCED BY THE COMPILER (AND PLACED INTO THE 'GO' FILE) MUST NOW BE LOADED WITH THE SDSS LIBRARY. THE LIBRARY IS CALLED '#LIB' AND IS ON ACCOUNT 197. THERE IS NO PASSWORD. IF ANY EXTERNAL FUNCTION SUBPROGRAMS TO PERFORM LOGICAL FUNCTIONS ARE NEEDED, THEY MUST BE INCLUDED AT THIS TIME. ANY OF THE HONEYWELL ROUTINES TO INITIATE PROGRAM EXECUTION MAY BE USED FOR THIS PURPOSE.

THE SIMULATION WILL MAY BE DONE EITHER ON-LINE OR IN BATCH MODE. HOWEVER, IF ANY INTERRUPTS WERE REQUESTED, THEN THE SIMULATION MUST BE RUN ON-LINE.

ALL 'PRINT' STATEMENTS WILL DISPLAY DATA VIA THE M:LO DCB. ALL 'FILL' STATEMENTS WILL REQUEST DATA FROM THE M:INF DCB. THESE DCB'S MAY BE ALLOWED TO DEFAULT TO THE TERMINAL, OR THEY MAY BE SET TO A FILE. (IN THE CASE OF M:LO, IT WILL DEFAULT TO THE LINE PRINTER IF THE SIMULATION IS RUN IN BATCH MODE.)

TO SIMULATE A REAL-TIME INTERRUPT, FIRST INITIATE EXECUTIONS OF THE GO-FILE. WHENEVER AN INTERRUPT IS DESIRED, DEPRESS THE 'BREAK' KEY ON THE TERMINAL. THE PROGRAM WILL RESPOND BY REQUESTING AN INTERRUPT NUMBER. THE INTERRUPT NUMBER IS THEN ENTERED ON THE TERMINAL. THIS NUMBER MUST CORRESPOND WITH THE NUMBER DEFINED BY AN 'INTERRUPT' STATEMENT IN THE SYSTEM DESCRIPTION. IF THE NUMBER IS VALID, CONTROL WILL THEN BRANCH TO THE FIRST STATEMENT WITHIN THE INTERRUPT ROUTINE.

APPENDIX B

RESULTS FROM A SIMULATION WITH THE COMPUTER OF CHAPTER III

A short program to sum three values in a list was written based upon the machine language of the computer described in Chapter III. This program, written in standard assembler format, is shown in Figure B1.

```
 LOC   CODE  LINE
_____________________________________________________________________
0000  08009    1. BEGIN  LAC   KNT         PUT INDEX VALUE INTO INDEX
0001  38800    2.        LIA                   REGISTER IA
0002  3D800    3.        CLA               SET AC = 0
0003  1A00D    4. LOOP   TAD   TABLE+3,I   ADD TABLE ELEMENT USING INDEX
0004  39000    5.        INA               INCREMENT IA BY 1
0005  00009    6.        ISZ   KNT         ADD 1; IF = 0, ARE DONE
0006  30003    7.        JMP   LOOP        GO BACK FOR MORE
0007  2800D    8.        DAC   RESULT      STORE SUM IN MEMORY
0008  38000    9.        HALT              ALL DONE.....
0009  3FFFD   10. KNT    DC    -3          LOOP COUNTER
000A  00005   11. TABLE  DC    5,3,-6      VALUES TO BE SUMED
000B  00003
000C  3FFFA
000D          12. RESULT DS    1           RESERVE 1 WORD FOR SUM
              13.        END   BEGIN
```

FIGURE B1

ASSEMBLY PROGRAM TO SUM THREE VALUES

This program uses indexing, in line 4, to select a particular value from TABLE. The assembler notation is somewhat arbitrary, as there is in fact no assembler for this machine.

The program was manually entered into the computer through the front panel as part of the simulation. (The program could have been loaded by a 'FILL' statement.) The 'PRINT' statements in the original description in Chapter III request a display of all register contents following each instruction fetch, as well as after each operation on the front panel.

The results of the simulation are given below. Note that the last three operations on the front panel (at the end of the simulation output) displays the memory location containing the sum of the three values, and causes the computer to enter a 'WAIT' state. The only exit from this 'WAIT' state is by an external interrupt.

```
!SDSS.197 MACHINE OVER ROMS,FILE
SDSS V0-LO 13:35 MAR 23,'77
OPTIONS:LS,LD
NO ERRORS

!LINK ROMS,FUNCTIONSR,#LIB.197 OVER LMN
     LINKING  ROMS
     LINKING  FUNCTIONSR
     LINKING  #LIB
'P1' ASSOCIATED.
     LINKING  SYSTEM LIB

!R

!LMN.
VALUES AT LINE  225


PCLG = 00001FC0          MALG = 00001FC0     MDLG = 0003BFC0
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
?2
 ENTER SWITCHES IN HEX
?0


VALUES AT LINE  225


PCLG = 00001FC0          MALG = 00000000     MDLG = 0003BFC0
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
?3
 ENTER SWITCHES IN HEX
?08009


VALUES AT LINE  225


PCLG = 00001FC0          MALG = 00000001     MDLG = 00008009
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
?3
 ENTER SWITCHES IN HEX
?38800
```

```
VALUES AT LINE  225


PCLG = 00001FC0          MALG = 00000002       MDLG = 00038800
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 3D800


VALUES AT LINE  225


PCLG = 00001FC0          MALG = 00000003       MDLG = 0003D800
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 1A00D


VALUES AT LINE  225


PCLG = 00001FC0          MALG = 00000004       MDLG = 0001A00D
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 39000


VALUES AT LINE  225


PCLG = 00001FC0          MALG = 00000005       MDLG = 00039000
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 00009
```

```
VALUES AT LINE   225


PCLG = 00001FC0          MALG = 00000006      MDLG = 00000009
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 30003


VALUES AT LINE   225


PCLG = 00001FC0          MALG = 00000007      MDLG = 00030003
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 2800D


VALUES AT LINE   225


PCLG = 00001FC0          MALG = 00000008      MDLG = 0002800D
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 38000


VALUES AT LINE   225


PCLG = 00001FC0          MALG = 00000009      MDLG = 00038000
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 3FFFD
```

```
VALUES AT LINE  225

PCLG = 00001FC0          MALG = 0000000A     MDLG = 0003FFFD
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 00005

VALUES AT LINE  225

PCLG = 00001FC0          MALG = 0000000B     MDLG = 00000005
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 00003

VALUES AT LINE  225

PCLG = 00001FC0          MALG = 0000000C     MDLG = 00000003
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 3
 ENTER SWITCHES IN HEX
? 3FFFA

VALUES AT LINE  225

PCLG = 00001FC0          MALG = 0000000D     MDLG = 0003FFFA
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 1
 ENTER SWITCHES IN HEX
? 0
```

```
VALUES AT LINE   225

PCLG = 00000000          MALG = 0000000D     MDLG = 0003FFFA
ACLG = 0000BFC0
 ENTER OPERATION REQUEST
? 5

VALUES AT LINE   42

PC    = 00000000         MA   = 00000000     MD   = 00008009
AC    = 0000BFC0         IR   = 00008009     L    = 00000001
IA    = 0002BFC0         RFLA = 00000001

VALUES AT LINE   42

PC    = 00000001         MA   = 00000001     MD   = 00038800
AC    = 0003FFFD         IR   = 00038800     L    = 00000001
IA    = 0002BFC0         RFLA = 00000001

VALUES AT LINE   42

PC    = 00000002         MA   = 00000002     MD   = 0003D800
AC    = 0003FFFD         IR   = 0003D800     L    = 00000001
IA    = 0003FFFD         RFLA = 00000001

VALUES AT LINE   42

PC    = 00000003         MA   = 00000003     MD   = 0001A00D
AC    = 00000000         IR   = 0001A00D     L    = 00000001
IA    = 0003FFFD         RFLA = 00000001

VALUES AT LINE   42

PC    = 00000004         MA   = 00000004     MD   = 00039000
AC    = 00000005         IR   = 00039000     L    = 00000000
IA    = 0003FFFD         RFLA = 00000001
```

```
VALUES AT LINE    42

PC    = 00000005        MA   = 00000005    MD    = 00000009
AC    = 00000005        IR   = 00000009    L     = 00000000
IA    = 0003FFFE        RFLA = 00000001

VALUES AT LINE    42

PC    = 00000006        MA   = 00000006    MD    = 00030003
AC    = 00000005        IR   = 00030003    L     = 00000000
IA    = 0003FFFE        RFLA = 00000001

VALUES AT LINE    42

PC    = 00000003        MA   = 00000003    MD    = 0001A00D
AC    = 00000005        IR   = 0001A00D    L     = 00000000
IA    = 0003FFFE        RFLA = 00000001

VALUES AT LINE    42

PC    = 00000004        MA   = 00000004    MD    = 00039000
AC    = 00000008        IR   = 00039000    L     = 00000000
IA    = 0003FFFE        RFLA = 00000001

VALUES AT LINE    42

PC    = 00000005        MA   = 00000005    MD    = 00000009
AC    = 00000008        IR   = 00000009    L     = 00000000
IA    = 0003FFFF        RFLA = 00000001

VALUES AT LINE    42

PC    = 00000006        MA   = 00000006    MD    = 00030003
AC    = 00000008        IR   = 00030003    L     = 00000000
IA    = 0003FFFF        RFLA = 00000001
```

```
VALUES AT LINE    42

PC     = 00000003        MA    = 00000003     MD    = 0001A00D
AC     = 00000008        IR    = 0001A00D     L     = 00000000
IA     = 0003FFFF        RFLA  = 00000001

VALUES AT LINE    42

PC     = 00000004        MA    = 00000004     MD    = 00039000
AC     = 00000002        IR    = 00039000     L     = 00000001
IA     = 0003FFFF        RFLA  = 00000001

VALUES AT LINE    42

PC     = 00000005        MA    = 00000005     MD    = 00000009
AC     = 00000002        IR    = 00000009     L     = 00000001
IA     = 00000000        RFLA  = 00000001

VALUES AT LINE    42

PC     = 00000007        MA    = 00000007     MD    = 0002800D
AC     = 00000002        IR    = 0002800D     L     = 00000001
IA     = 00000000        RFLA  = 00000001

VALUES AT LINE    42

PC     = 00000008        MA    = 00000008     MD    = 00038000
AC     = 00000002        IR    = 00038000     L     = 00000001
IA     = 00000000        RFLA  = 00000001

VALUES AT LINE   225

PCLG = 00000009          MALG = 00000008      MDLC = 00038000
ACLG = 00000002
```

```
 ENTER OPERATION REQUEST
? 2
 ENTER SWITCHES IN HEX
? 000D


VALUES AT LINE   225


PCLG = 00000009          MALG = 0000000D     MDLG = 00038000
ACLG = 00000002
 ENTER OPERATION REQUEST
? 4


VALUES AT LINE   225


PCLG = 00000009          MALG = 0000000E     MDLG = 00000002
ACLG = 00000002
 ENTER OPERATION REQUEST
? 0
```

APPENDIX C

SOME NOTES ON THE SDSS COMPILER

THIS APPENDIX CONTAINS SOME USEFUL INFORMATION CONCERNING THE SDSS COMPILER ITSELF. IT IS WRITTEN FOR THOSE PERSONS WHO WILL BE MODIFYING AND IMPROVING SDSS IN THE FUTURE. A BASIC UNDERSTANDING OF THE SIGMA - 7 AND THE SDSS SOURCE CODE IS ASSUMED.

SDSS ITSELF IS A COMPILER; IT ACCEPTS PROGRAMS WRITTEN IN SDSS AS SOURCE DATA, AND PRODUCES SIGMA - 7 OBJECT CODE AS OUTPUT.

SDSS IS A 1-PASS COMPILER. AS SUCH, SDSS PERFORMS NO GLOBAL OPTIMIZATION OF OBJECT CODE OVER ADJACENT STATEMENTS. THIS RESULTS IN A CONSIDERABLE AMOUNT OF INEFFICIENCY IN THE GENERATED CODE, BUT THE COMPILER WAS MUCH EASIER TO WRITE THAN WOULD HAVE BEEN A MULTI-PASS COMPILER.

WITH TWO EXCEPTIONS, SDSS IS WRITTEN ENTIRELY IN FORTRAN. THE TWO EXCEPTIONS ARE A SERIES OF ASSEMBLY ROUTINES WHICH PERFORM FUNCTIONS DIFFICULT OR IMPOSSIBLE TO PERFORM ENTIRELY IN FORTRAN (SUCH AS I/O ROUTINES AND DISK FILE MANIPULATION), CALLED 'SDSS', AND THE CONVERSON FROM CHARACTER INTEGER TO BINARY INTEGER IN ROUTINE 'DECMAL'. ROUTINE 'DECIMAL' CONTAINS SEVERAL LINES OF IN-LINE ASSEMBLY CODE TO ENABLE THE CONVERSION TO PROCESS VALUES UP TO 2**32 -1, WHICH OCCUPY A 32-BIT INTEGER.

SDSS IS CONSTRUCTED IN A MODULAR FASHION. MOST MAJOR FUNCTIONS ARE PERFORMED IN A SEPARATE SUBROUTINE, WHICH MAY CALL OTHER SUBROUTINES TO HELP IT OUT. APPROXIMATELY ONE-HALF OF SDSS IS WRITTEN IN A STRUCTURED FORM; THAT IS, IT IS WRITTEN, IN FORTRAN, IN A FORM ANALOGOUS TO PL/1'S 'DO-WHILE' AND EXTENDED 'IF-THEN-ELSE' STATEMENTS. FORTRAN 'GO-TO' STATEMENTS ARE LIMITED TO THOSE NECESSARY TO IMPLEMENT THE ABOVE STATEMENT FORMS. USE OF THIS

STRUCTURED FORM GREATLY SIMPLIFIES THE EFFORT OF WRITTING AND MODIFYING THE SOURCE CODE. IT IS HIGHLY RECOMMENDED THAT ALL ADDITIONS AND MODIFICATIONS TO THE COMPILER BE WRITTEN IN A STRUCTURED FORM. (THE OTHER HALF OF THE COMPILER WAS WRITTEN IN A VERY UNSTRUCTURED FORM, AND IS CORRESPONDINGLY MORE DIFFICULT TO UNDERSTAND AND MODIFY.)

THE USE OF VARIOUS DISK FILES FACILITATE THE COMPILATION PROCESS. FOR EXAMPLE, ALL ERROR CONDITIONS CAUSE DATA TO BE WRITTEN TO A KEYED DISK FILE. THE KEY IS COMPOSED OF THE LINE NUMBER OF THE RECORD IN WHICH THE ERROR WAS DETECTED, AND A VALUE INDICATING WHICH ERROR THIS IN IN THE PARTICULAR RECORD (1, 2, ETC.). WHEN THE END OF THE STATEMENT HAS BEEN REACHED, A CHECK IS ALWAYS MADE TO SEE IF THE STATEMENT SHOULD BE WRITTEN OUT. ANY ERRORS ALWAYS CAUSES THE STATEMENT TO BE WRITTEN. AT THIS POINT, ANY ERROR MESSAGES CAN BE WRITTEN IMMEDIATELY FOLLOWING THE STATEMENT IN WHICH THEY WERE DETECTED, AND IN THE SAME ORDER IN WHICH THEY WERE DETECTED.

ANOTHER USE OF THE DISK FILES IS IN BUILDING A TABLE OF ALL CONSTANTS GENERATED BY SDSS DURING COMPILATION. IN ORDER TO PREVENT FORTRAN FROM FILLING UP AN ARRAY WITH THESE CONSTANTS, AND CAUSING THE COMPILER TO QUIT, THE ARRAY IS WRITTEN TO DISK SHOULD IT EVER BECOME FILLED. THE ARRAY IS NOW EMPTY, AND CAN BE FILLED AGAIN. IF IT IS NECESSARY TO LOOK UP THE LOCATION OF A CONSTANT, THE VALUES IN THE ARRAY ARE CHECKED FIRST, FOLLOWED BY A SEARCH OF THE COPIES OF THE ARRAY ON DISK. THIS ENTIRE OPERATION IS CONTROLLED BY SUBROUTINE 'CONST'.

THIS SAME METHOD COULD HAVE BEEN EXTENDED TO ALL THE TABLES CONTAINING NECESSARY VALUES, SUCH AS THE TABLE OF REGISTERS. HOWEVER, THIS WAS NOT DONE DUE TO THE

COMPLEXITY OF HAVING AS MANY SETS OF THIS ROUTINE AS WOULD BE NECESSARY FOR ALL THE TABLES. THUS THERE ARE LIMITS ON THE MAXIMUM NUMBER OF MOST ELEMENTS.

THE OBJECT CODE THAT IS GENERATED BY THE COMPILER IS QUITE WELL DOCUMENTED WITHIN THE SOURCE PROGRAM AT THE POINT WHERE IT IS GENERATED. EACH LOADER ITEM IS DEFINED BY A MNEMONIC CODE ALONG WITH ITS OPERANDS, AND IS ACCOMPANIED BY A HEXIDECIMAL VERSION OF WHAT SHOULD BE GENERATED.

THERE IS ONE COMPILER OPTION WHICH WAS NOT DESCRIBED IN THE LANGUAGE REFERENCE MANUAL. THIS THE THE 'LO' OPTION. USE OF THIS OPTION CAUSES THE COMPILER TO GENERATE INTERNAL SYMBOL TABLES OF ALL THE ELEMENTS DEFINED BY THE DESIGN, AND ALL TEMPORARY LOCATIONS DEFINED BY THE COMPILER. THIS FEATURE IS OF GREAT USE IN DEBUGGING THE COMPILER AND THE GENERATED CODE UNDER THE 'DELTA' PROCESSOR ON THE SIGMA - 7.

USE OF THE 'LO' OPTIONS ALSO FORCES THE 'LS' OPTION. USE OF THE 'NS' OPTION PROHIBITS THE GENERTION OF INTERNAL SYMBOL TABLES.

SDSS ALSO CONTAINS A BUILD-IN DEBUGGING OPTION. THE SDSS STATEMENT

```
*TRACE ON
```

MAY BE PLACED INTO THE SYSTEM DESCRIPTION. USE OF THIS STATEMENT CAUSES THE COMPILER TO GENERATE A TRACE OF EVERY SUBROUTINE ENTRY AND EXIT, ALONG WITH THE VALUES OF SEVERAL VARIABLES AT THE ENTRY AND EXIT POINTS. USE OF THIS STATEMENT CAN CAUSE THE PRODUCTION OF LARGE AMOUNTS OF INFORMATION. THUS, IT IS TO BE USED CAUTIOUSLY.

TO TERMINATE THE TRACING OPTION AT ANY TIME, THE STATEMENT

```
*TRACE OFF
```

IS USED.

SDSS IS CURRENTLY SET UP TO ALLOW THE INCLUSION OF AN 'INPUT' STATEMENT. SUCH A STATEMENT WOULD BE INTERPRETED AS AN 'INPUT' STATEMENT, AND A SUBROUTINE, CALLED 'INPUT', WOULD BE CALLED. THIS SUBROUTINE SIMPLY PRINTS OUT A MESSAGE STATING THAT 'INPUT' STATEMENTS ARE NOT ACCEPTED BY SDSS, AND THEN IGNORES THE STATEMENT. TO IMPLEMENT AN 'INPUT' STATEMENT, ALL THAT WOULD HAVE TO BE DONE IS TO REPLACE THIS ONE SUBROUTINE WITH ONE WHICH WOULD GENERATE OBJECT CODE.

AN 'INPUT' STATEMENT MIGHT HAVE THE FORM:

$$\text{INPUT} \left\{ \begin{array}{c} \text{<NAME>} \\ \\ \text{<MEMORY> (<LO>, <HI>)} \end{array} \right\} , \ldots$$

WHERE	<NAME>	IS THE NAME OF A REGISTER, SCALAR, OR SWITCHES.
	<MEMORY>	IS THENAME OF A RANDOM-ACCESS MEMORY.
	<LO>	IS THE FIRST LOCATION IN THE MEMORY TO BE READ INTO.
	<HI>	IS THE LAST LOCATION IN THE MEMORY TO BE READ INTO.

IT IS ENVISIONED THAT THIS STATEMENT WOULD CAUSE THE GENERATION OF AN ARGUMENT LIST AND A BRANCH TO A LIBRARY

SUBROUTINE WHICH WOULD DO THE ACTUAL DATA INPUT OPERATION. IT MIGHT BE NICE FOR THE SUBROUTINE TO REQUEST DATA FROM THE TERMINAL BY NAME (SO AS TO PREVENT CONFUSION BY THE DESIGNER AS TO WHICH VALUE HE WAS TYPING IN). THE SUBROUTINE WOULD ALSO HAVE TO MASK OFF ANY HIGH ORDER BITS THAT EXCEED THE SIZE OF THE ELEMENT WHICH IS TO RECIEVE THE VALUE. SUCH A SUBROUTINE WOULD NOT BE DIFFICULT TO IMPLEMENT. IT MIGHT EVEN BE POSSIBLE TO UTILIZE ROUITNE '#FILLMEM' FROM THE LIBRARY TO PERFORM MOST OF THE NECESSRY OPERATIONS.

THE OBJECT CODE GENERATED BY SDSS MUST BE LINKED WITH THE SDSS LIBRARY IN ORDER TO PRODUCE THE SIMULATION LOAD MODULE. THE LIBRARY ROUTINES CONSIST OF ASSEMBLY ROUTINES TO PERFORM SUCH FUNCTIONS AS INITIALIZATION OF VALUES, HANDLING INTERRUPTS, AND PERFORMING I/O OPERATIONS AS DICTATED BY 'PRINT' STATEMENTS.

THE COMPILER GENERATES TWO SEPARATE ROMS AS OUTPUT. THE FIRST CONSISTS OF THE MACHINE CODE NECESSARY TO PERFORM THE SIMULATION, AND CONTAINS ALL CONSTANTS DEFINED BY THE COMPILER. THE PHYSICALLY FIRST MACHINE INSTRUCTION GENERATED IS LABELED BY THE EXTERNAL NAME '#MAINPGM'. THE FIRST INSTRUCTION TO BE EXECUTED WHEN THE SIMULATION IS INITIATED IS LABELED '#START'. #START IS ALWAYS LOCATED AT A HIGHER CORE ADDRESS THAN IS #MAINPGM. THE SEQUENCE OF CODE FOLLOWING #START CALL ANY INITIALIZATION ROUTINES, AND THEN BRANCHES TO THE FIRST INSTRUCTION OF THE SIMULATION CODE. THIS FIRST INSTRUCTION MAY NOT BE AT #MAINPGM SINCE IT IS POSSIBLE TO SPECIFY ANOTHER STATEMENT AS THE FIRST TO BE EXECUTED ON THE 'END' STATEMENT.

THE SECOND ROM IS DEFINED BY THE EXTERNAL NAME '#DATA'. THIS ROM CONTAINS ALL THE DATA REGIONS NECESSARY

FOR THE SIMULATION. ALL HARDWARE ELEMENTS ARE DEFINED AT THE BEGINNING OF THIS MODULE, AND ANY TEMPORARY STORAGE LOCATIONS ARE DEFINED FOLLOWING THE ELEMENT LOCATIONS. FOR THE LOCATION OF THE ELEMENTS WITHIN THIS MODULE, SIMPLY REQUEST A SYSTEM DEFINITION SUMMARY WITH THE COMPILER OPTION 'LD'.

REFERENCES

1. Barbacci, M. B., and D. P. Siewiorek, 'Automated Exploration of the Design Space for Register Transfer (RT) Systems', Proceedings of the First Annual Symposium of Computer Architecture, Gainsville, Florida, December, 1973.

2. Bell, C. G., and A. Newell, Computer Structures: Readings and Examples. New York: McGraw-Hill, 1971.

3. Gentry, M., 'ACompiler for AHPL Control Sequences', PhD dissertation, University of Arizona, June, 1971.

4. Hill, F. J., and G. R. Peterson, Digital Systems: Hardware Organization and Design. New York: John Wiley and Sons, 1973.

5. Knudson, M., 'PMSL - An Interactive Language for High Level Description and Analysis of Computer Systems', Technical Report, Computer Science Department, Carnige-Mellon University, 1974.

6. Su, Stephen Y. H., 'A Survey of Computer Hardware Descriptive Languages in the USA', Computer, Vol 7, #12, December 1974, pp. 45-51.

[redacted]

N378
C851
cop.2

Crane, William P
A register-transfer descriptive language and simulator for digital networks

DATE	ISSUED TO
JAN 2	[redacted]ey HAYDEN 1102 S 6th st 587-9627 Bozeman
[illegible]	1104 S. Montana Yavar Seadati 6-8841
OCT 14	Al Yousif 763 4558 P.O. Box 3526

[redacted]

N378
C851
Cop 2