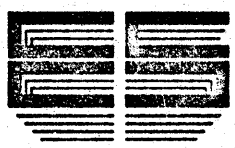
# EVANS & SUTHERLAND COMPUTER CORP.

# LINE DRAWING SYSTEM MODEL 1

# SYSTEM REFERENCE MANUAL



Evans & Sutherland Computer Corporation
Three Research Road
Salt Lake City, Utah 84112

November 1, 1970

U0800-1-1

For -101 Systems

# Preface

This manual has been developed by Evans & Sutherland Computer Corporation for use by experienced engineers and programmers who are interested in using and/or maintaining the LDS-1 Graphic Display System.

The first six chapters deal with the basic hardware of the LDS-1. The next three explain programming features. The remaining chapters deal with LDS-1 hardware options and how they relate to the rest of the system.

Any suggestions or corrections to this manual are invited, as it is our intent to continually improve it. Pages which may be used to suggest corrections are included at the end of the manual.

TABLE OF CONTENTS

# CHAPTER 1

# SYSTEM OVERVIEW

## 1.1 Function of the LDS-1

The Evans & Sutherland Line Drawing System Model 1 (LDS-1) is a high-speed display processing system which has the capability to process and display complex three-dimensional dynamic displays in real time as well as the capability to process and display two- and three-dimensional static displays. The LDS-1 is an interactive display system which will accept input from a variety of graphic input devices. The LDS-1 shares the memory of a host computer and is interfaced both directly to the memory port and to the I/O path of the host computer. In this environment the LDS-1 operates as an autonomous processor with its own instruction fetching and data accessing facilities and works off the memory of the host computer on a "cycle stealing" basis.

The LDS-1 is a fully programmable display processing system with very general and very powerful graphic processing capabilities and thus lends itself to a wide variety of problems which involve displays. Programs are written specifically for the LDS-1, assembled by the assembler of the host computer, stored in memory, and then interpreted and executed by the LDS-1. The LDS-1 can be started and stopped by the host computer's monitor via the I/O path.

The LDS-1 interprets a DRAWING definition, processes the DRAWING to generate the view the user wishes to see, and displays a PICTURE which represents the processed view of the drawing. For the purposes of this manual, these terms will be assumed to have the following definitions:

- DRAWING--the DRAWING is the definition stored in memory which consists of two- or three-dimensional coordinate data and a display program which determines how these coordinate values should be interpreted.

- PICTURE--the PICTURE is made up of the lines and dots that appear on the Display Scope.

The intent of this manual is to describe the principal features and explain the programming procedures for the LDS-1 when it is interfaced to a Digital Equipment Corporation PDP-10. The manual assumes the reader to have an introductory knowledge of the basic graphic functions performed by the LDS-1 and to be familiar with the DEC MACRO-10 Assembler.

## 1.2 System Configuration

The system configuration of the LDS-1 is shown in figure 1.1. Each of the major units operates asynchronously and

LDS-1 DISPLAY SYSTEM CONFIGURATION

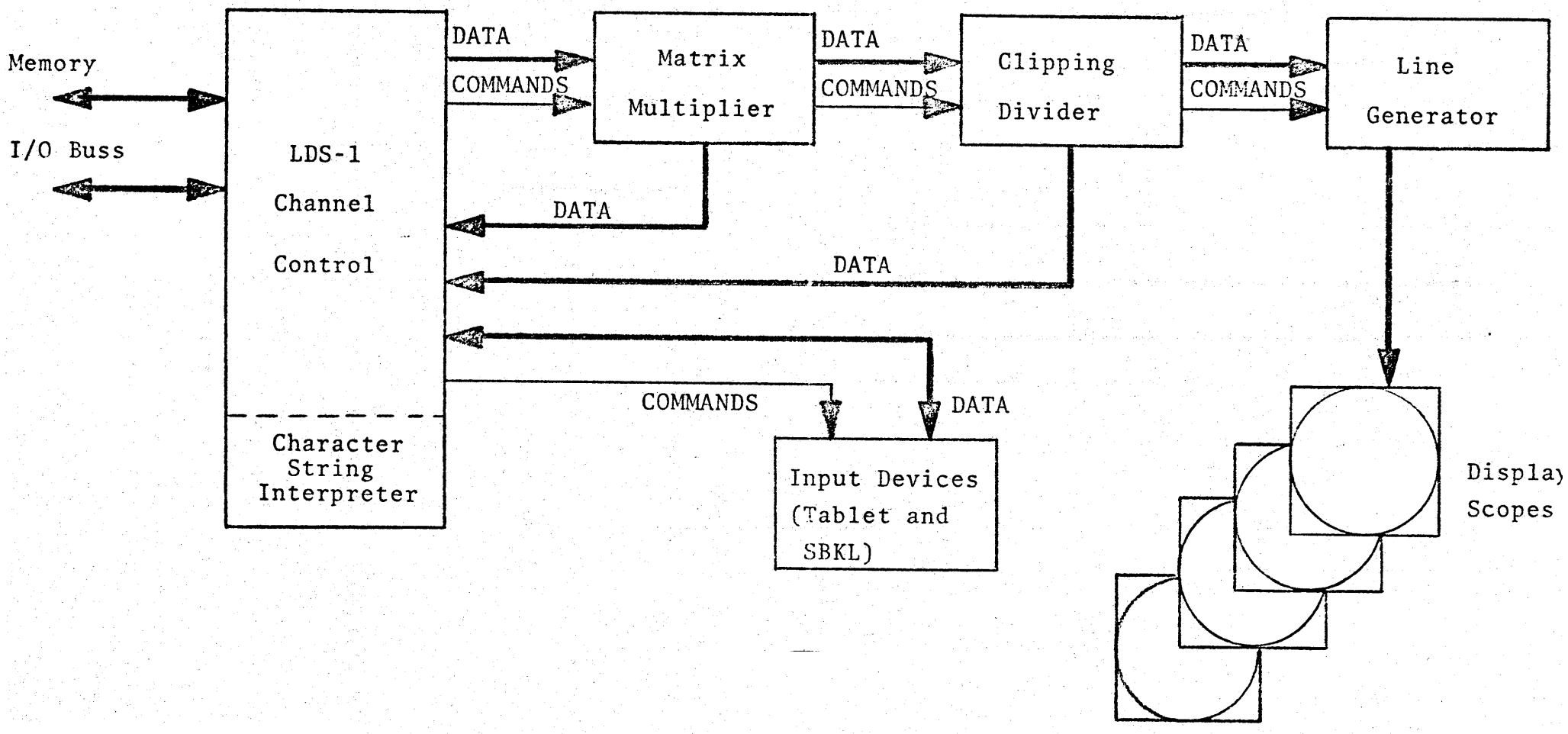



Figure 1.1

together they work as a processing "pipeline." The drawing information flows through the LDS-1 pipeline to produce a picture on the Display Scope. The function of each of the units is described briefly below:

- The Channel Control. The Channel Control interprets the display program and provides the other units of the LDS-1 with the appropriate data and control.

- The Character String Interpreter. The Character String Interpreter (sometimes referred to as the Character Bubble) can interpret a packed string of characters and call on either the hardware character generator (if any) or software character routines to draw the character.

- The Matrix Multiplier. The Matrix Multiplier can rotate, translate, and scale the drawing. The Matrix Multiplier also iterates the difference equations which define curves.

- The Clipping Divider. The Clipping Divider allows the user to specify the portion of the drawing he wishes to view. The Clipping Divider will automatically eliminate all portions of the drawing which lie outside the viewing area and then scale and position the picture on the Display Scope.

- The Line Generator and Display Scope. The Line Generator converts the digital specification of endpoints into analog sweep voltages to drive the deflection system of the Display Scope. The picture appears on the Display Scope.

- Input Devices. Various input devices are available with the LDS-1. These are described in later chapters. Generally, these devices pass input data through the Channel Control and into memory where it is processed by the LDS-1, the host computer, or both.

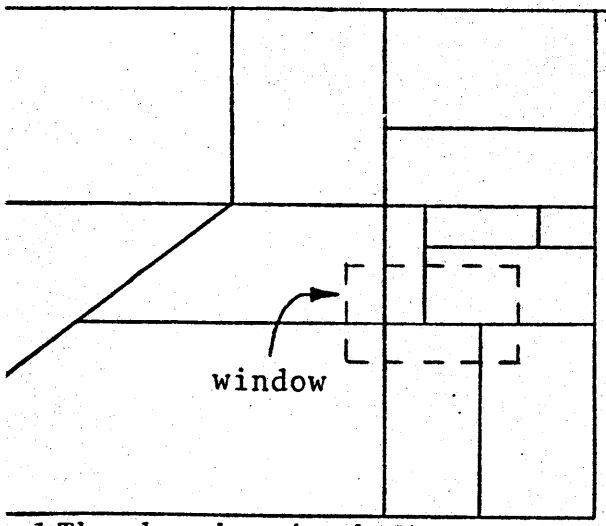
Figure 1.2 illustrates some of the processing functions performed by the LDS-1, in particular the major functions of Clipping Divider and Matrix Multiplier.

## 1.3 The Data Base

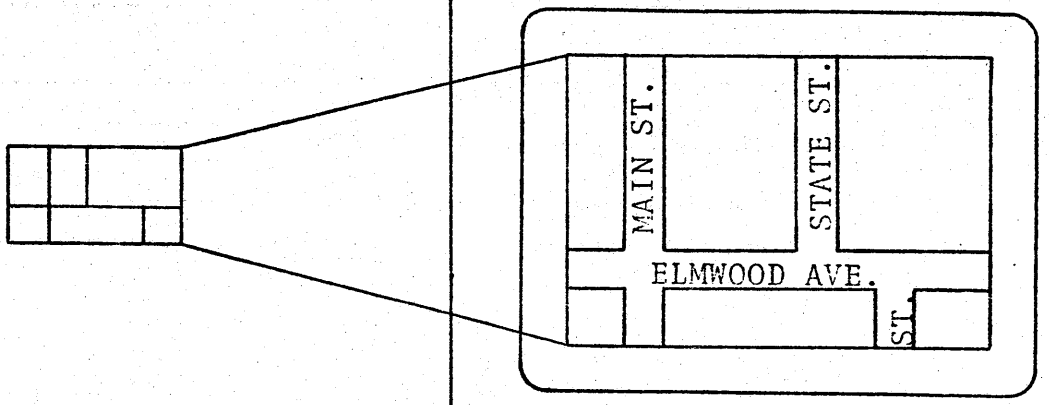
In seeking to understand and in programming the LDS-1, it is useful to think of three basic sets of information used by the LDS-1. These groups include:

- The Display Program. The display program contains the drawing instructions, addressing information for accessing data, and control information to control the processing performed by the LDS-1. The display program is interpreted by the Channel Control.

LDS-1 GRAPHIC PROCESSING



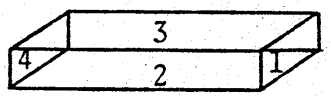

1.The drawing is defined in the user-chosen drawing space and a "window" is specified.



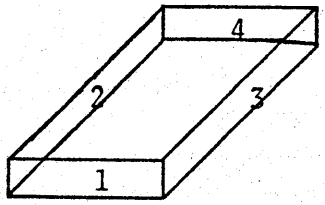

2. All parts of the drawing outside the "window" are eliminated by the Clipping Divider.

3. The clipped drawing is mapped onto the "viewport" on the Display Scope.

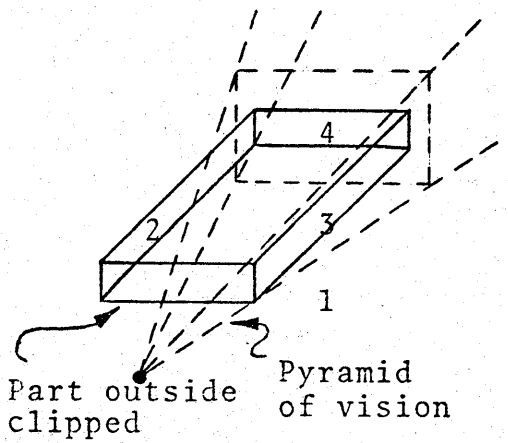
Two-dimensional windowing



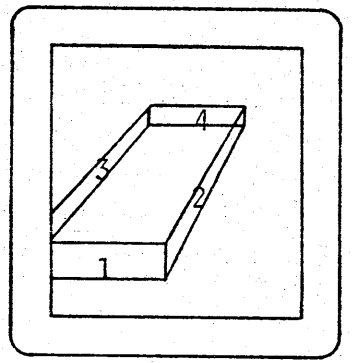

1.The drawing is defined in a three-dimensional drawing space.




2. The Matrix Multiplier rotates, translates, & scales the drawing.




3. The drawing is compared to a pyramid of vision by the Clipping Divider




4. The drawing is clipped, and put in perspective, then mapped onto the viewport of the Display Scope.

Three-dimensional processing

Figure 1.2

• The Coordinate Data. The coordinate data specify the Cartesian coordinate values for the position of dots or the endpoints of lines in the drawing. The coordinate values are processed by the arithmetic units in the pipeline.

• The Viewing Parameters. The viewing parameters are the numerical quantities which are used by the arithmetic units in processing the drawing to determine the position, orientation, and scale of the picture displayed.

The display program is generally stored separately from the coordinate data and viewing parameters. The Coordinate data and the viewing parameters are pure numerical quantities and can thus be manipulated directly by both the LDS-1 and the host computer. Thus the coordinate data and the viewing parameters can be considered part of a single data base which is common to the LDS-1 and to the host computer. This common data base makes direct interaction between the LDS-1 and the host computer possible as indicated in figure 1.3.

## 1.4 The Display Program

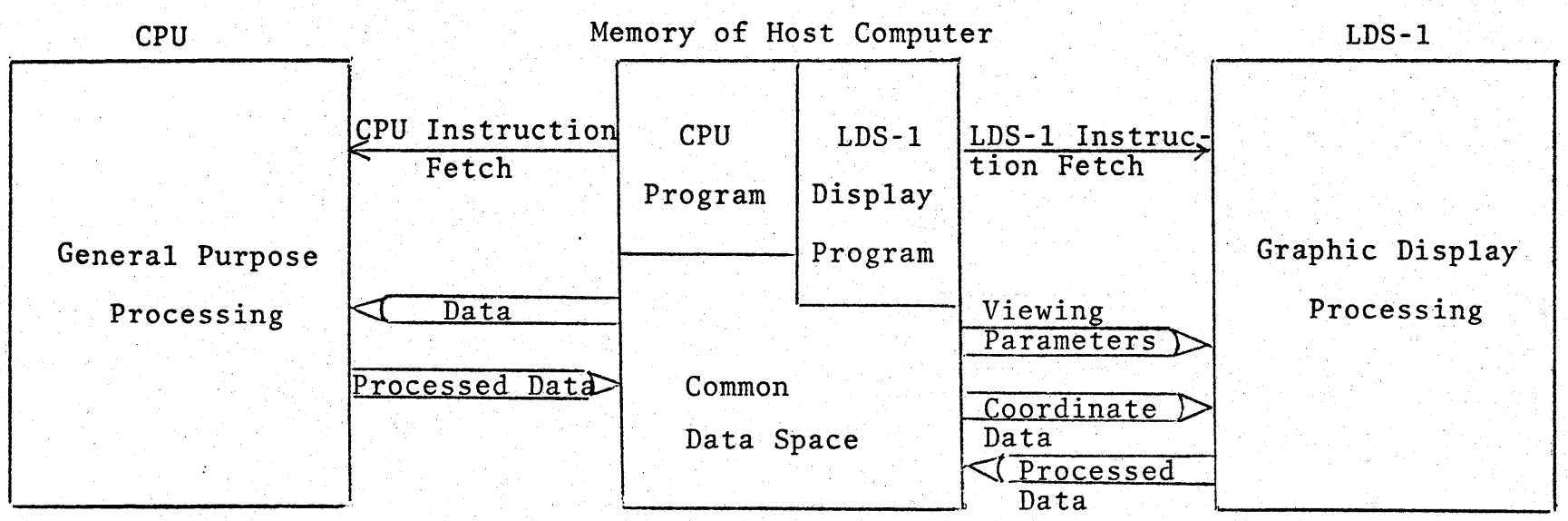
The display program contains (1) drawing instructions, which usually reference coordinate data, (2) instructions which manipulate the viewing parameters, (3) and instructions which control the addressing of data and the operation of the LDS-1. These instructions are described in detail in chapters 7 and 8 of this manual, but it is necessary to understand basically how the drawing instructions are used to define the drawing in order to comprehend the functions performed by the LDS-1.

### 1.4.1 Drawing Instructions

The drawing is interpreted, processed, and displayed one line (or dot) at a time. The LDS-1 maintains the coordinates of the "current point." The simple drawing instructions reference the coordinates of a "new point." In "set point" and "dot" operations, this new point is used for positioning and its value becomes the value of the current point. Lines may be drawn either to the new point from the current point ("draw to") or from the new point back to the current point ("draw from"). Examples of these operations are shown in figure 1.4.

More complex drawing instructions which result in the execution of a series of these basic drawing instructions are also included in the LDS-1. These instructions provide for drawing polygon-like figures, star-like figures, disconnected series of lines, and dots as shown in figure 1.4. The coordinate data for these instructions are stored in a continguous table to which the instruction makes reference.

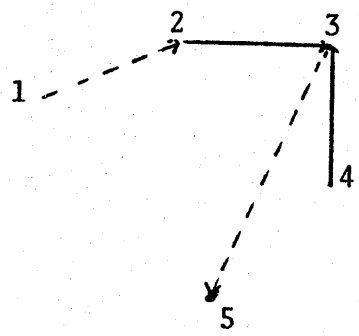
THE DATA BASE




The LDS-1 operates as an autonomous processor, similar to the CPU, but with specialized facilities for display processing tasks. The combination of the CPU and the LDS-1 gives a dual processor system where the CPU is used for general computing, and the LDS-1 is used to process and display pictures.

Figure 1.3

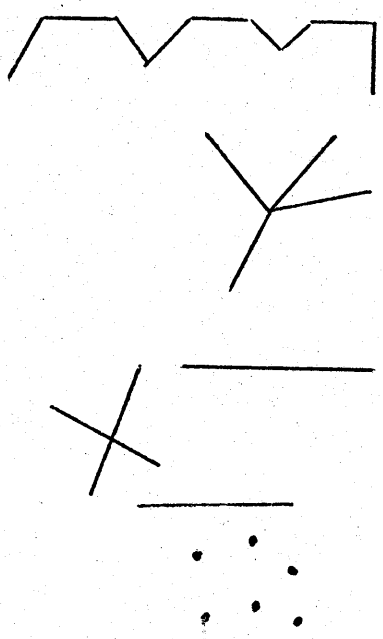
# DRAWING INSTRUCTIONS




## Basic Drawing Instructions

1 is current location.

| | |
|---|---|
| "Set point" to 2 | (2 becomes current location) |
| "Draw to" 3 | (3 becomes current location) |
| "Draw from" 4 | (3 remains current location) |
| "Dot" 5 | (5 becomes current location) |



## Complex Drawing Instructions

"Polygon" = Set point, draw to, draw to, draw to...

"Star" = Set point, draw from, draw from...

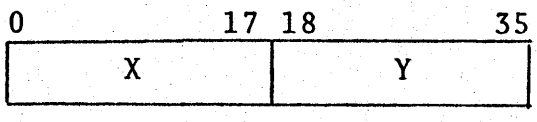
"Lines" = Set point, draw to, set point, draw to, set point...

"Dots" = dot, dot, dot.

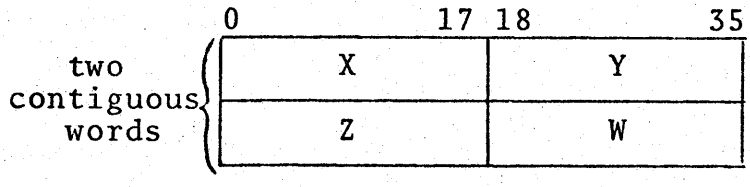
Figure 1.4

COORDINATE DATA FORMATS
FOR LDS-1/PDP-10

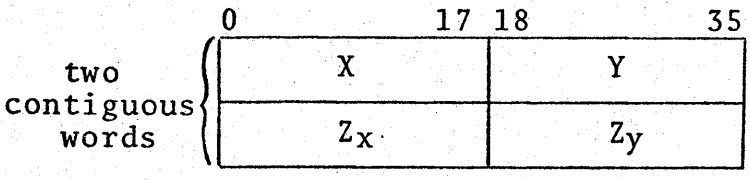
2-dimensional Data




3-dimensional Data (with Matrix Multiplier)



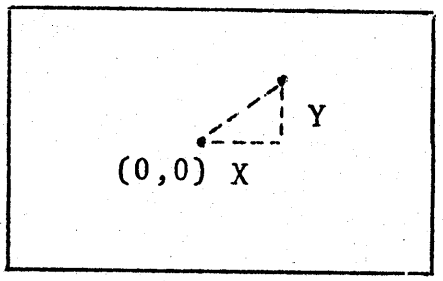

3-dimensional Data (without Matrix Multiplier or Matrix Multiplier inactive)



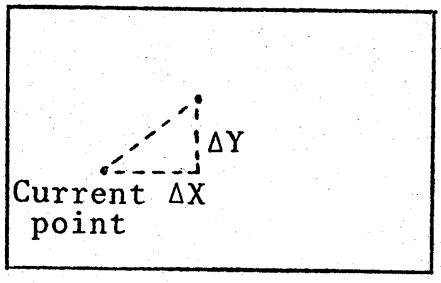

Note: $Z_X$ and $Z_Y$ are generally the same; making them different provides for different perspective divisors in X and Y, a feature which is useful in drawing certain types of curves.

Figure 1.5

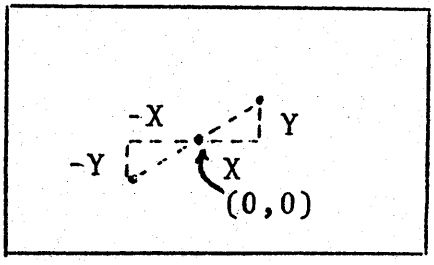
DATA FORM SPECIFICATIONS



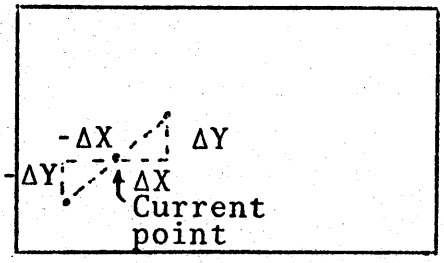

Absolute
(X,Y)




Relative
(ΔX, ΔY)




Size Absolute
(X,Y) (-X,-Y)




Size Relative
(ΔX, ΔY) (-ΔX, -ΔY)

Figure 1.6

## 1.5 Coordinate Data

The coordinate data specify the position of endpoints of lines or the position of dots. These points are specified by their Cartesian coordinate values. The coordinate system on which the drawing is defined is a two or three-dimensional virtual drawing space known as the PAGE. The PAGE is a fixed point two's complement space. When the LDS-1 is interfaced to a PDP-10, coordinate values are stored in 18-bit PDP-10 half-words and thus the page stretches from -400000 to +377777 (all addresses and coordinate data are expressed in octal) in each axis. It should be remembered that coordinate values are specified in page coordinates rather than in the coordinate system of the Display Scope.

Two-dimensional coordinate values are specified by X and Y values. These values are packed into PDP-10 half-words so that together they occupy a single PDP-10 word as shown in figure 1.5. Three-dimensional data should take one of two forms depending on whether the Matrix Multiplier is being used. Data to be processed by the Matrix Multiplier should be in "homogeneous coordinates". The format for homogeneous coordinates is (X, Y, Z, W). (Homogeneous coordinates are discussed in chapter 3 dealing with the Matrix Multiplier, and in Appendix III.) If the Matrix Multiplier is not used, coordinate values pass to the Clipping Divider and should be in the form (X, Y, $Z_X$, $Z_Y$) where $Z_X$ and $Z_Y$ are generally equal (see chapter 4). In either case, three-dimensional data occupy two full words of memory as shown in figure 1.5.

## 1.6 Data Form Specifications

The drawing instructions not only address the coordinate data for the new point, but also specify the way to interpret this data. Viewing parameter data can also be in one of several forms. See figure 1.6.

- Absolute. Data specified as "absolute" denote an absolute location on the drawing space.

- Relative. Data specified as "relative" are taken as a relative displacement from the current point. With the exception of matrix elements for the Matrix Multiplier, which are added to the old matrix values, this applies both to coordinate data and viewing parameters.

Two special data forms are used primarily for Clipping Divider viewing parameters, although they can also be used for coordinate data.

- Size Absolute. The "size absolute" specification results in the data being taken as both a positive and negative displacement from the origin of the drawing space.

- Size Relative. The "size relative" specification results in the data being taken as both a positive and a negative displacement from the current point.

## 1.7 Programming Language

This manual describes programming of the LDS-1 in PDP-10 MACRO-10 Assembly Language. The LDS-1 mnemonics are defined by MACRO-10 OPDEF's so that the MACRO-10 assembler will prepare LDS-1 programs. It is also possible to use other language translating facilities of the host computer to prepare LDS-1 programs.

# CHAPTER 2

# THE CHANNEL CONTROL

## 2.1 Function

The Channel Control is a stored program processor which operates off the memory of the host computer and functions as the programmable control unit of the LDS-1. The Channel Control interprets the display program and controls the operations of the arithmetic devices. In addition, the Channel Control accesses the necessary coordinate data and viewing parameters and passes this data to the processing pipeline.

A series of memory address registers within the Channel Control provide programmable addressing machanisms for instruction fetching and data accessing. A group of control registers are used to regulate the operation of the LDS-1 and to indicate the state of the system. These registers are described in detail in section 2.12 and reference should be made to this section during the following discussion of the operation of the Channel Control.

## 2.2 Instruction Set

The instruction set of the Channel Control is specifically oriented towards display processing tasks and the controlling of the processing performed by the LDS-1. The instruction set is divided into the following groups:

- Group 0 -- Load immediate data into the Channel Control registers. Push the contents of the Channel Control registers onto the stack. Change mode.
- Group 2 -- Conditional load immediate data into the Channel Control registers.
- Group 3 -- Manipulation of the parameter registers of other LDS-1 devices (load/store, sink/retrieve).
- Group 4 -- Drawing instructions with direct addressing.
- Group 5 -- Drawing instructions with indirect addressing.
- Group 6 -- Dataless drawing instructions (used by the Matrix Multiplier to generate curves and in several other cases where data is not relevant to the operation).
- Group 7 -- Character interpreting instructions.

These groups and the individual instructions are discussed in detail in chapters 7 and 8 of this manual, but it is useful to keep the basic groups in mind as one attempts to understand the operation of the Channel Control.

## 2.3 Operating Modes

The Channel Control has four operating modes which determine its method of instruction fetching:

- Program Mode -- For the normal processing of the display program, the Channel Control is in "program mode." The other three modes are for special situations encountered in the display program.

- Repeat Mode -- For drawing or character handling instructions which make repeated data accesses, the Channel Control should be put in "repeat mode" (see section 2.6).

- Peel Mode -- "Peel mode" is used to retrieve information from the stack (e. g. when returning from subroutines; see section 2.7).

- Execute Mode -- When a word in the data space is to be executed as an instruction, the Channel Control is put in "execute mode" (see section 2.10).

All of these modes are entered under program control. Many LDS-1 instructions have mode change fields. Since the new mode primarily affects the manner in which the next instruction is fetched, it is best to think of the mode change being made after the instruction has been executed. An exception is repeat mode, which is best thought of as being entered before the instruction has been executed (see section 2.6).

It is possible to specify mode changes to more than one mode at once. In such a case, the Channel Control first enters the mode with highest priority and then any modes with lower priority. A table of how to specify mode changes is shown in section 7.2. Figure 2.1 shows the operation of the Channel Control in each of its modes.

CHANNEL CONTROL MODES OF OPERATION

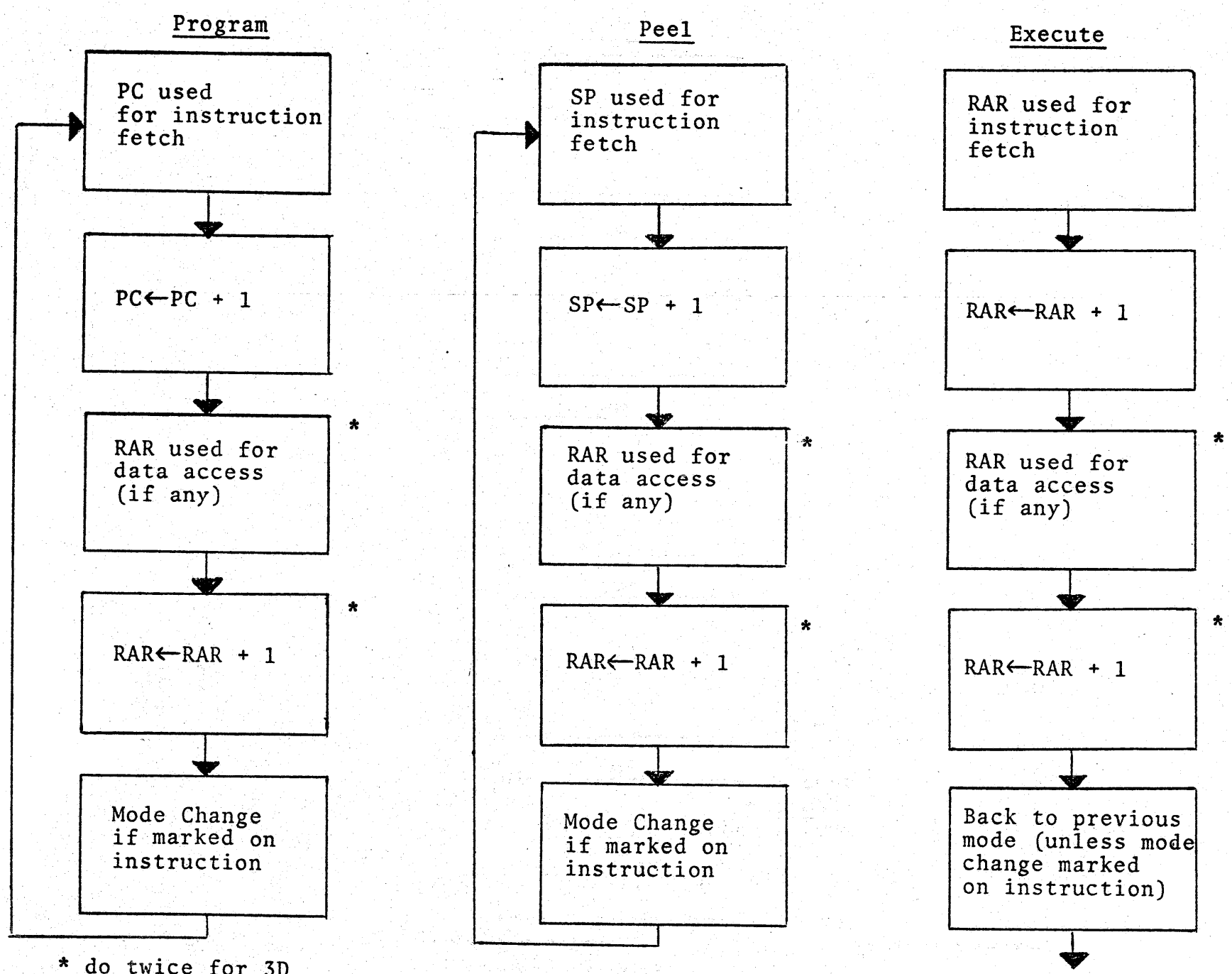
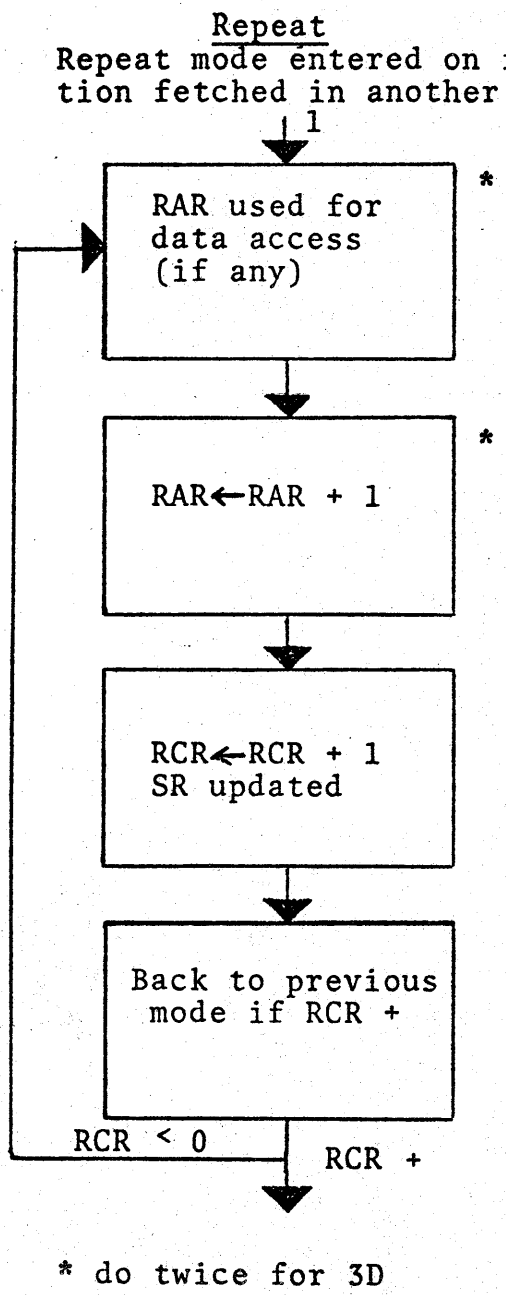
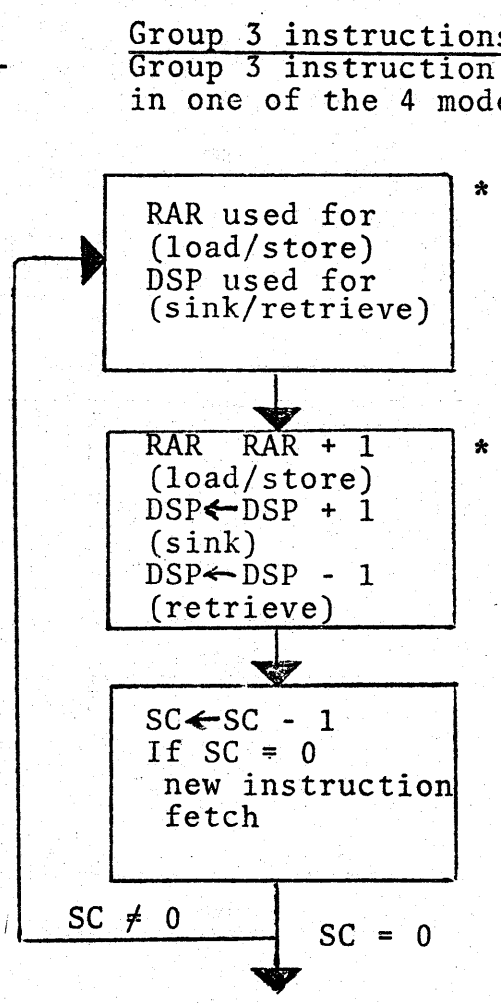



Figure 2.1

CHANNEL CONTROL MODES OF OPERATION




Mode Priority

1. Group 3[2]
2. Repeat
3. Execute
4. Program and Peel (mutually exclusive).

SC = Register count

[1]When using the Character String Interpreter to call software subroutines, an instruction is fetched in repeat mode. The Character String Interpreter provides the address which passes through, and is stored in register P1. The address in P1 is not, however, incremented after it is used.

[2]Group 3 instructions do not constitute a distinct mode.

Figure 2.1 (continued)

## 2.4 Instruction Fetching and Decoding

The address register used for the instruction fetch depends on the mode of the Channel Control. In program mode, the PROGRAM COUNTER (PC) is used for the instruction fetch. In peel mode, the STACK POINTER (SP) is used; and in execute mode the READ ADDRESS REGISTER (RAR), which is normally used for the data access, is used for the instruction fetch. In repeat mode no instruction fetch is made except when interpreting characters, in which case the address for the instruction fetch is provided by the Character String Interpreter and stored in the temporary register P1.

The left half of an instruction word is deposited in the instruction register where it is decoded. (Note: The instruction register is not otherwise accessible to the user.) The decoded instruction is used along with system mode control information stored in the DIRECTIVE register (DIR) to control the operation of the LDS-1. (See section 2.12 for details concerning the DIR and other registers mentioned here.)

In group 2 (conditional load) instructions the loading is not done unless the condition specified is met. The conditions which may be tested are stored in the STATUS REGISTER (SR). Some of these bits are program flags set by the program itself. Others indicate certain conditions detected by the arithmetic devices or external devices such as the tablet or Lorgnette.

## 2.5 Data Accessing

For instructions which require some action on the part of the processing units of the LDS-1 pipeline, the Channel Control generates the necessary control information and accesses the necessary data. The instruction may address the data either directly or indirectly. For direct addressing the READ ADDRESS REGISTER (RAR), which serves the Channel Control as a data pointer, is used to specify the address of the data. In two-dimensional operation, only one word is accessed; however, in three-dimensional operation, two contiguous words are accessed by the RAR. This is true regardless of the type of data being accessed (i. e. whether it is coordinate data, parameter data, or control information).

When data values are addressed indirectly, the RAR is used to fetch a single word which contains two half-word pointers which point to two separate coordinate specifications (e.g. for the two ends of a line). The pointers are deposited in registers P1 and P2 of the Channel Control. These registers are then used to access the actual coordinate data. As in the case of direct addressing, in three-dimensional operation two contiguous words are fetched by each pointer.

## 2.6 Repeat Drawing Instructions

The Channel Control can generate a repeated series of simple drawing instructions in order to draw more complex figures with a single instruction. When the Channel Control receives a "draw to", "draw from", "polygon", "star", "lines", or "dots" instruction and is put in repeat mode, it will automatically generate the appropriate series of basic drawing instructions. These complex drawing instructions make reference to a table of coordinate data rather than to a single data item. The RAR should point to the head of this table when the instruction sequence begins and will automatically step through the table as the instruction is being executed.

The iterations of this process are counted by the READ COUNT REGISTER (RCR) of the Channel Control. When using direct addressing the RCR should be loaded with the two's complement of the number of data elements (endpoints). Each time a data item is fetched (which is one word in 2D and two words in 3D), the RCR is incremented. For indirect addressing the RCR should be loaded with the two's complement of the number of pointer words and is incremented as each pointer word is accessed by the RAR. When the count in the RCR runs out, repeat mode is cleared and the Channel Control continues on with the program. The RCR can also be incremented by the display program (when the Channel Control is not in repeat mode) and tested for negative (or positive). It can thus be used as a counter for program loops or other functions.

For repeat mode drawing instructions, there is also an "absolute/relative" sequence generated by the Channel Control. This allows, for example, the first point to be taken as absolute, and the rest as relative. Many other sequences are possible, These sequences and the sequences for the drawing instruction are shown in figures 7.1 and 7.2.

## 2.7 The Stack

The stack is an area in memory of the host computer into which the information in the Channel Control registers may be pushed. The STACK POINTER (SP) of the Channel Control points to the last entry into the stack, When a "push" instruction is received the Channel Control generates a "load immediate" instruction which references the register from which the information is pushed. The information in the register is then placed in the immediate data field of the instruction and the entire instruction is written into the stack.

Information is retrieved from the stack by putting the Channel Control in peel mode. In peel mode the SP is used to supply the address for the instruction fetch so instructions are fetched from the stack. When these instructions

are executed they restore the information to the Channel Control registers.

The SP is decremented before it is used to write information onto the stack and incremented after it is used to retrieve information from the stack. The SP is thus incremented just as the PC and the stack can be processed as a program.

The Channel Control will stay in peel mode, executing instructions from the stack, until an instruction is encountered which indicates a mode change. Instructions on the stack which indicate a mode change are said to be "marked". The "push mark" instruction causes the code for a change to the current mode of the Channel Control to be included in the "load immediate" instruction which is pushed onto the stack. Marking is used, for example, for the first register of a particular group that is to be pushed onto the stack. When that group of registers is to be returned to their old values, the Channel Control is put into peel mode and the instructions on the Stack are executed until the marked instruction is encountered. The Channel Control will then return to its previous mode. (Note: The only way to push execute mode onto the stack is with an I/O instruction or with the Character String Interpreter.)

## 2.8 The Data Sink

It may also be necessary (e.g. when entering a subroutine) to save vital parameters from the other LDS-1 devices by saving the contents of the paramenter registers in the "data sink". The data sink is an area in memory which is addressed by the DATA SINK POINTER (DSP) of the Channel Control. The contents of the parameter registers of the arithmetic devices may be the "sinked" into or "retrieved" from the data sink. The sink and retrieve instructions can refer to a single register or sequential group of parameter registers within a single device. The DSP is incremented after it is used to sink a parameter and decremented before a parameter is retrieved.

## 2.9 Subroutines

The stack and the data sink provide convenient facilities for subroutine linkage and communication. Subroutines may be nested to any level, and they may be reentrant. Methods for inplementing subroutines are descussed in section 8.3.

## 2.10 Execute Mode

In execute mode, the RAR rather than the PC is used for the instruction fetch. As soon as the instruction is fetched, the Channel Control will drop out of execute mode. If, however, this instruction is marked with a mode change to execute mode, the Channel Control will again enter execute mode and again use the RAR for the instruction fetch. In this manner, execute instructions can be chained. If the state of PC is not changed by one of the instructions executed, as soon as the Channel Control returns to program mode, it fetches the instruction immediately following the first execute instruction in the chain.

It should be noted that repeat mode has priority over execute mode (see figure 2.1) so that if both modes are set, the Channel Control will first enter repeat mode and then when it drops out of repeat mode, it will enter execute mode and the RAR will be used for the next instruction fetch. It is, however, possible for the instruction fetched in execute mode to be a repeat instruction. In this case, execute mode has already been entered and cleared before the repeat mode instruction is processed so that when the Channel Control drops out of repeat mode, it will return to program mode. Examples of such cases are given in section 8.3 of this manual.

## 2.11 Returning Processed Output to Memory

The processed output of the arithmetic devices may be returned to memory. The Channel Control regulates this process and controls the writing of data back into memory. When a unit has data ready to return to memory, it signals the Channel Control which stops its operation to record this data. The WRITE ADDRESS REGISTER (WAR) of the Channel Control is used to provide the memory address for recording the processed output. Since the WAR is incremented after each use, this data is recorded in a contiguous table. The length of this table may be limited by loading the WRITE COUNT REGISTER (WCR) of the Channel Control with the two's complement of the desired length of the table. The WCR is incremented each time a word is written back into memory of the host computer. When the count in the WCR runs out, an interrupt can be sent to the host computer (if the interrupt is enabled; see section 6.2). Note: Because the pipeline must clear before the LDS-1 is stopped (i.e. because all pending data must be processed and recorded), it is possible that that the length of the table will actually over-run the limit set by the count in the WCR. The overrun will in no case exceed 34 words. An overrun this large is, however, extremely unlikely because to have this happen, all units must be returning all possible outputs to memory (which results in a scrambled table in memory).

## 2.12 Channel Control Registers

In this section, the registers of the Channel Control are examined and the use of each register and the meaning of each bit is explained. Figure 2.2 shows a block diagram of the Channel Control registers. Registers 0-7 contain addresses. Their output goes to the Memory Address Register (MAR), register 15, where it is either incremented or decremented and then written back into the register. Registers 10-14 are control registers. Register 17 is the NEXT register of the Character String Interpreter.

The registers of the Channel Control are each 18 bits long (except the NEXT register which is only 8 bits). These bits are labeled 18-35 because their values are loaded from bits 18-35 of the instruction word. (Note: All register addresses are given in octal.)

# CHANNEL CONTROL REGISTER CONFIGURATION

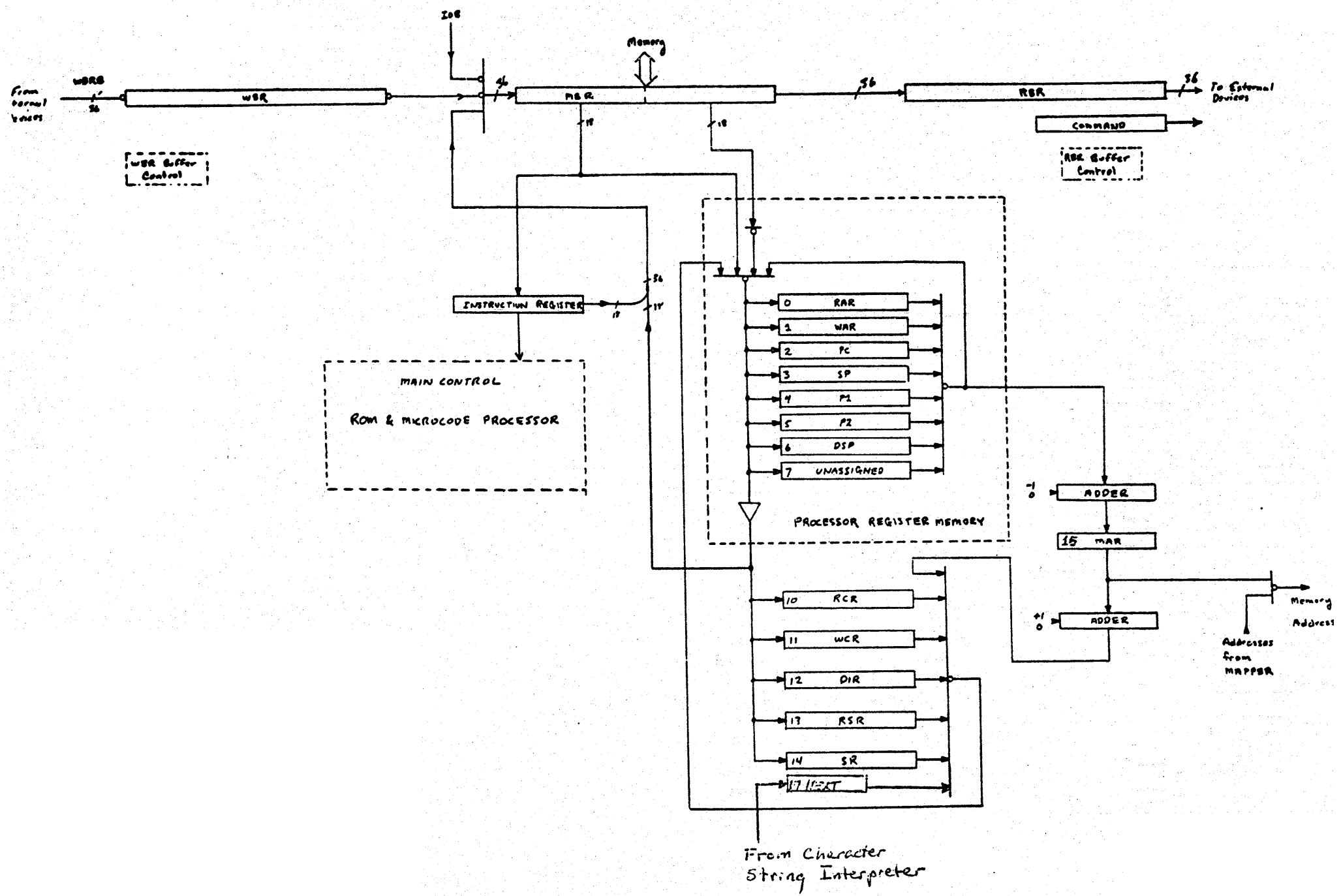
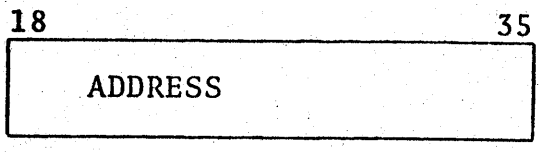



Figure 2.2

PROGRAM COUNTER (PC)

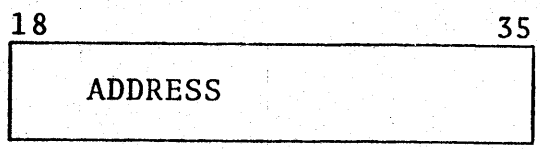



Binary Address
0010

USE: The PC is used for the instruction fetch when the Channel Control is in program mode.

OPERATIONS: PC ⟵ PC+1 (after each use)
PC ⟵ New Address (load immediate or jump)
PC ⟶ Stack (push PC)

READ ADDRESS REGISTER (RAR)



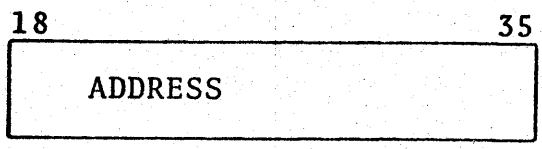

Binary Address
0000

USE: The RAR is used:

(1) to access data for direct addressing.
(2) to access a pointer word with two pointers for indirect addressing.
(3) to fetch the instruction in execute mode.
(4) to access data for load and store instructions.
(5) (software convention) to pass address of a parameter list to a subroutine.

OPERATIONS: RAR ⟵ RAR+1 (after each use)
RAR ⟵ New Address (load immediate)
RAR ⟶ Stack (push RAR)

P1

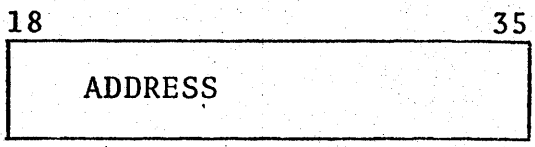



Binary Address
0100

USE: P1 is used as a temporary register to hold one pointer for indirect data addressing. The address provided by the Character String Interpreter (see section 2.12) also passes through, and is stored in register P1.

OPERATIONS:
P1 ⟵ New Address (load immediate)
P1 ⟵ Character Address (provided by Character String Interpreter)
P1 ⟶ Stack (push P1)
P1 ⟵ Left half at indirect address word.

P2



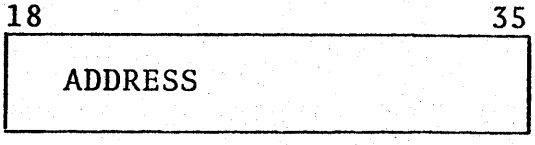

Binary Address
0101

USE: P2 is used as a temporary register to hold the other pointer for indirect addressing. P2 is also used during the push operation. Note: A "load immediate and push P2" results in pushing the new (rather than the old) data onto the stack.

OPERATIONS:

P2 ⟵ New Address (load immediate)
P2 ⟶ Stack (push P2)
NEW DATA ⟶ P2 ⟶ Stack (load immediate and push P2)
OLD DATA (from other CC register) ⟶ P2 ⟶ Stack (push other register)
P2 ⟵ Right half of indirect address word.

STACK POINTER (SP)



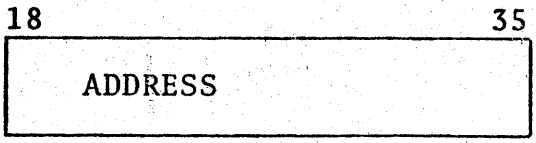

Binary Address
0011

USE: The SP is used to specify the address at which information from the Channel Control registers is pushed onto the stack. In peel mode, the SP is used for the instruction fetch.

OPERATIONS:

SP ⟵ SP-1 (before information is pushed on stack)
SP ⟵ SP+1 (after information is retrieved from the stack, peel mode)
SP ⟵ New Address (load immediate SP)
SP ⟶ Stack (push SP)

DATA SINK POINTER (DSP)



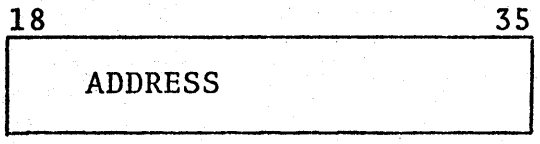

Binary Address
0110

USE: The DSP is used to specify the address at which information from the parameter registers is put into or taken from the data sink.

OPERATIONS:

DSP ⟵ DSP+1 (after sinking)
DSP ⟵ DSP-1 (before retrieving)
DSP ⟵ New Address (load immediate DSP)
DSP ⟶ Stack (push DSP)

WRITE ADDRESS REGISTER (WAR)

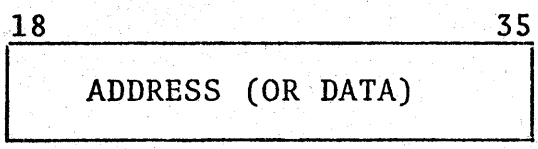



Binary Address
0001

USE: The WAR is used to specify the address for processed data written back into memory.

OPERATIONS:
WAR ⟵ WAR+1 (after each use)
WAR ⟵ New Address (load immediate WAR)
WAR ⟶ Stack (push WAR)

UNASSIGNED REGISTER (UR)

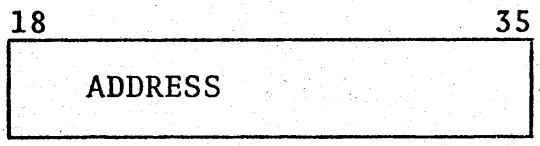



Binary Address
0111

USE: The UR is available for programmer use as desired.

OPERATIONS:
UR ⟵ New Address or Data (load immediate)
UR ⟶ Stack (push)

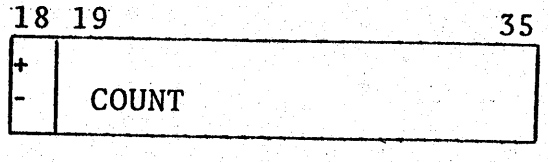
MEMORY ADDRESS REGISTER (MAR)




Binary Address
1101

USE: All memory addresses pass through the MAR. The MAR may not be loaded, but it may be pushed. However, SP-1 will be recorded, as the SP is used as the address for the push operation.

READ COUNT REGISTER (RCR)




Binary Address
1000

USE: The RCR is used to count the number of data items. It is loaded with the two's complement of the number of either:

(1) end points (draw direct)
(2) pointer words (draw indirect)
(3) characters (for use with Character String Interpreter)

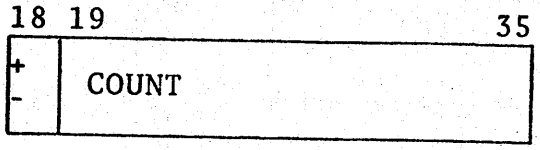
When the count in the RCR goes positive repeat mode is cleared. The RCR can also be used as a counter in program loops.

OPERATIONS:

RCR ⟵ RCR+1 (after each data item is processed)
RCR ⟵ Count (load immediate RCR)
RCR ⟶ Stack (push RCR)
RCR ⟵ RCR+1 (group 2 instruction if "J" RCRN)

BIT MEANINGS: The Contents of the RCR are taken as a signed count.

WRITE COUNT REGISTER (WCR)




Binary Address
1001

USE: The WCR is used to count the number of words (rather than data items) written back into memory. The WCR is incremented after each word is written back into memory. When the WCR goes positive, an interrupt can be sent to the host computer. If processed output is not being returned to memory, the WCR can also be used as a counter for program loops.

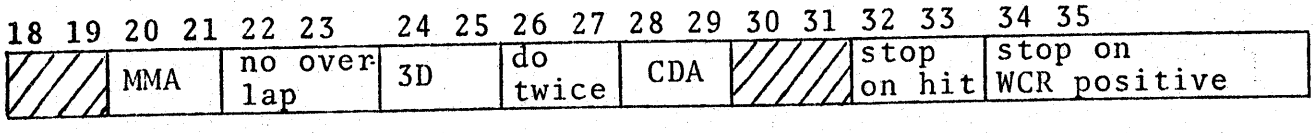
OPERATIONS:

WCR ⟵ WCR+1 (after each data item is written back into memory)
WCR ⟵ Count (load immediate WCR)
WCR ⟶ Stack (push WCR)
WCR ⟵ WCR+1 (group 2 instruction if "J" WCRN)

BIT MEANINGS: The contents of the WCR are taken as a signed count.

DIRECTIVE (DIR)

Binary Address
1010




USE: The DIR is used to specify global operating modes for the system. The bits are coded in a special way so that it is possible to manipulate functions independently.

OPERATIONS:
DIR ⟵ DATA (load immediate
DIR ⟶ Stack (push)

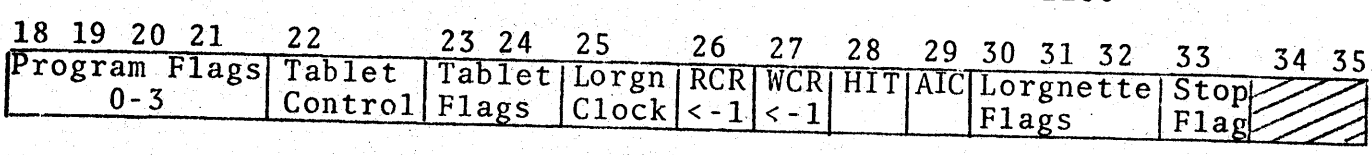
| "J" Set | "K" Clear | |
|---|---|---|
| 0 | 0 | no change |
| 0 | 1 | clear |
| 1 | 0 | set |
| 1 | 1 | complement |

| BITS | FUNCTION |
|---|---|
| 18 | Unused |
| 19 | Unused |
| 20 | J Matrix Multiplier Active |
| 21 | K Matrix Multiplier Active |
| 22 | J No Overlap. (If no overlap is set, the system will completely process one instruction (one line) before it starts to process the next. Although this slows the system down, it is useful in testing and "pointing" functions associated with the tablet). |
| 23 | K No Overlap |
| 24 | J 3D (Fetch two words of data) |
| 25 | K 3D |
| 26 | J Do twice (Do twice causes the two halves of the data word to be swapped and then used a second time. It is generally used with the SELF modes of the Clipping Divider, see section 4.10. Do twice is cleared if 3D is set.) |
| 27 | K Do twice |
| 28 | J Clipping Divider Active } Not yet implemented. These |
| 29 | K Clipping Divider Active } bits are there but do nothing. |
| 30-31 | Unused |
| 32 | J STOP ON HIT } Stop the Channel Control when |
| 33 | K STOP ON HIT } Clipping Divider sets the HIT bit. |
| 34 | J STOP ON WCR Positive |
| 35 | K STOP ON WCR Positive |

Note: The DIR does not hold the value loaded into it, but rather the status at the functions it controls. When the DIR is pushed and later restored these functions will be set to their previous state.

## STATUS REGISTER (SR)

Binary Address
1100




USE: The SR is used to hold the conditions which can be tested and changed by the conditional load instructions of group 2. Some of the bits are set and cleared only by the program, others are controlled both by the program and by the LDS-1, and some are controlled only by the LDS-1 and can be tested but not altered by the instructions.

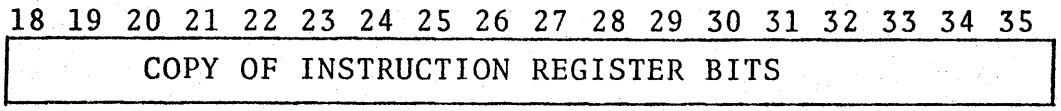
OPERATIONS:

SR ← DATA (load immediate)
SET CONDITION
CLEAR CONDITION
COMPLEMENT CONDITION
SR ⟶ Stack (push)
SR(bit) ⟵ condition from external equipment

| BIT | CONDITION NO. (Octal) | FUNCTION |
|---|---|---|
| 18-21 | 0-3 | Program Flags |
| 22 | 4 | Tablet Control (enables writing of tablet data into memory. The TABLET CONTROL bit is automatically cleared by "system clear" and by WCR positive. See chapter 12.) |
| 23-24 | 5-6 | Tablet flags (used to test Z values of tablet. see chapter 12). |
| 25 | 7 | Lorgnette Clock (see chapter 13). |
| 26 | 10 | RCR < -1 } Set automatically by RCR & WCR. |
| 27 | 11 | WCR < -1 } When these bits are "set" by a Group 2 instruction, the RCR or WCR is incremented; clearing has no effect. |
| 28 | 12 | Hit (set and cleared by Clipping Divider). |
| 29 | 13 | AIC (set and cleared by Clipping Divider). |
| 30-32 | 14-16 | Lorgnette color code (set by Lorgnette, cannot be set or cleared by group 2 instructions, see chapter 13). |
| 33 | 17 | Program Stop Flag |
| 34-35 | | Not used |

REPEAT STATUS REGISTER (RSR)

Binary Address
1011




USE: The RSR contains a copy of the bits in the Instruction Register. The RSR exists so that the Channel Control can be interrupted during a repeat mode instruction (or a group 3 instruction) and the present status of the Channel Control can be saved by pushing the RSR, then restored by peeling the value of the stack and returning it to the RSR.

OPERATIONS:

RSR ⟵ DATA (load immediate)
RSR ⟶ Stack (push)
RSR ⟵ Update for group 3 and repeat mode instructions (internal to Channel Control).

BIT MEANINGS:

| | |
|---|---|
| 18-20 | Instruction type (copy of IR 0-2) |
| *21-23 | Present state of the finite-state machine for setpoint/endpoint (copy of IR 3-5) |
| *24-26 | Present state of the finite-state machine for absolute/relative (copy of IR 6-8) |
| 27-30 | Register Address (copy of IR 9-12) |
| 31 | Copy of IR 13 |
| 32-35 | Short count for multiple load/store (copy of IR 14-17) |

*For group 3 instructions, bits 21-26 indicate:

| | |
|---|---|
| 21-22 | Load Store |
| 23-26 | Device and Manner |

NOTE: A LI RSR (RPTM) can be used to start up a repeat sequence.

## 2.13 The Character String Interpreter (Optional)

### 2.13.1 Function

The Character String Interpreter (sometimes referred to as the "Character Bubble") provides the LDS-1 with the capability to interpret character strings automatically. The Character String Interpreter unpacks the character code and sends the code to the hardware character generator (if this option is also included in the system). If there is no hardware character generator, if it is turned off, or if it does not recognize the character, the character code is used as an offset to fetch an instruction from a dispatch table, which can call the appropriate software character routine.

The Character String Interpreter is a much more powerful device than might appear at first glance. Because the subroutines called do not have to be merely character drawing routines, and because they themselves can use the Character String Interpreter, it is possible to treat the LDS-1 as a string processing machine with a syllable-by-syllable instruction code. The "character string" becomes the program with each operation code defined by an appropriate LDS-1 subroutine.

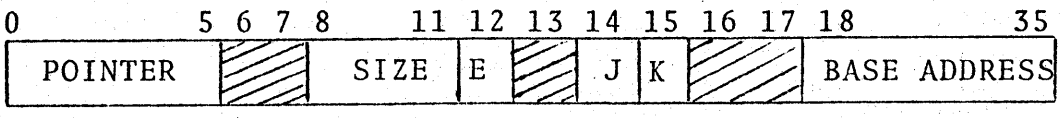
### 2.13.2 Registers

There are three registers associated with the Character String Interpreter. The FONT register specifies (1) the size of the character code (i.e. how many bits), (2) an internal pointer to the next character to be interpreted, (3) control signals for turning the hardware character generator (if any) on or off and (4) a base address for the character dispatch table. The format of this register is shown in figure 2.3. The CHAR register contains a copy of the memory word currently being interpreted. These two registers are treated as parameter registers of an external device by the Channel Control and are thus manipulated by group 3 instructions (see section 7.3.). The FONT register should be loaded prior to interpreting characters. The CHAR register is usually loaded automatically and need be manipulated directly only when the Character String Interpreter is "borrowed" by a character subroutine (see example, section 8.10).

In addition, the 8-bit (maximum) value of the next character to be interpreted may be read by the Channel Control register NEXT. The NEXT "register" is not implemented as a register but rather exist as the output of a "shift device" attached to the CHAR register. For this reason, it may be pushed onto the stack, but can not be loaded by a group 0 or group 2 instruction. Any attempt to load the NEXT register will change the state of the Character String Interpreter.

CHARACTER STRING INTERPRETER REGISTERS

FONT REGISTER — Binary Address 0001




where:

POINTER specifies the number of bits to the right of the end of the new Character to be interpreted.

SIZE specifies the bit length of each character code.

E specifies whether to load POINTER and SIZE, i. e. if $\overline{E}$ in the new font word is not one, POINTER and SIZE will not be loaded. E is always one when storing or sinking the font word.

JK turns the hardware character generator on or off according to the following table:

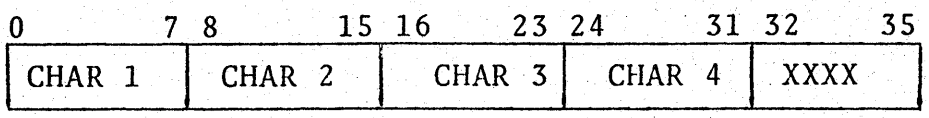
| J K | |
|---|---|
| 0 0 | remains unchanged |
| 0 1 | turns off |
| 1 0 | turns on |
| 1 1 | complements state |

BASE specifies the base address for the character dispatch table (BASE HAS NOTHING TO DO WITH THE LOCATION OF THE CHARACTER BEING INTERPRETED).

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

CHARacter REGISTER — Binary Address 0000

Example 8-bit characters - 36-bit word




- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

NEXT REGISTER — Binary Address 1111

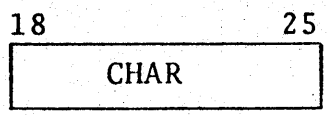



Figure 2.3

### 2.13.3 Operation

The Character String Interpreter will interpret characters up to 8 bits in length. Characters up to 15 bits may be designated and will be accessed from the data words according to the SIZE field, but only the least significant 8 bits are actually used in the addition to the base address. The size of the character codes is specified by the SIZE field of the FONT register. Characters should be packed into the memory word so that no character code overlaps word boundaries and the extra bits are on the right end of the word (see figure 2.3.)

The Character String Interpreter is turned on when the LDS-1 encounters a group 7 "do character" instruction which should also be marked to put the Channel Control in repeat mode. The word addressed by the "do character" instruction is loaded into the CHAR register. Prior to this, the RCR of the Channel Control should have been loaded with the (two's complement of the) number of characters to be interpreted (+1) and the FONT register should have been loaded with the appropriate information (see figure 2.3). The CHARacter register is loaded automatically with the word whose address is held in the READ ADDRESS REGISTER (RAR) of the Channel Control.

The Character String Interpreter begins by interpreting the first character. The number of bits specified by the SIZE field are peeled off and sent to the hardware character generator. If there is no hardware character generator, it is turned off, or if it does not recognize the character code, the code is added to the address specified by the BASE field of the font word. The resulting address is used to fetch a word from the character dispatch table. The address passes through, and is stored in register P1 of the Channel Control. Note that P1 is not incremented after this word is fetched. The word fetched by P1 is treated as an instruction. Normally the character dispatch table is made up of a series of "jump push" instructions as shown in section 8.10. These jump push instructions serve as subroutine calls. The character subroutines are discussed in more detail in section 8.10. After exiting from the subroutine, the Channel Control again returns to repeat mode and interprets the next character.

Each time a character is interpreted, the quantity in the SIZE field of the FONT register is subtracted from the quantity in the POINTER field, and the result is placed in the POINTER field. If this subtraction yields a negative result, a new word of characters is fetched, and the value in the pointer field is set to 36-SIZE. If, in fact, the POINTER value is larger than $47_8$, the POINTER field is reset to 36-SIZE. In this sense the Character String Interpreter will

"self-initialize" when the POINTER portion of the FONT register is loaded with $50_8$ or greater. When the first word of characters is fetched, the POINTER will be set to 36-SIZE. The Character String Interpreter will continue to interpret characters in this manner until the count in the RCR goes positive which will cause the Channel Control to drop out of repeat mode, or until a terminating character is encountered. Terminating characters are defined by placing a "no op" with a change to program mode in the character dispatch table.

# CHAPTER 3

# THE MATRIX MULTIPLIER

## 3.1 Function

The Matrix Multiplier is the first arithmetic device in the LDS-1 display processing pipeline. The Matrix Multiplier performs rotations, translations, and scalings of the drawing by multiplying the coordinate data by an internally stored transformation matrix. The Matrix Multiplier can also compute the product of two such transformation matrices to give a composite transformation for substructures within the drawing definition. The third function of the Matrix Multiplier involves iterating a set of difference equations for drawing two- or three-dimensional curves which are drawn as a series of short line segments. Families of such curves can also be generated to draw a cross-hatched surface patch.

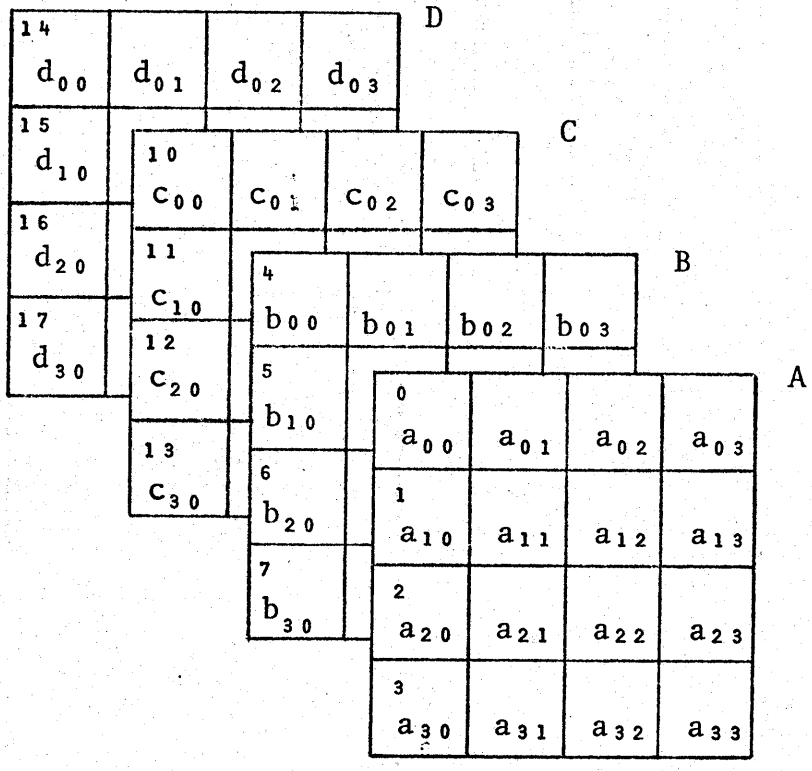
The basic configuration of the Matrix Multiplier and the addresses of the registers used for storing matrix elements are shown in figure 3.1. Four matrices A, B, C, and D each of dimension 4 x 4, are stored internally in a 4 x 4 x 4 matrix array of storage registers. The values in these registers may be manipulated by the "load", "store", "sink", and "retrieve" instructions. See section 7. The matrix multiplications are performed by a high-speed array multiplier. Input data for the Matrix Multiplier are passed from the Channel Control and the output is sent to the Clipping Divider, back to the memory of the host computer via the Channel Control, or both.

## 3.2 Three-dimensional Matrix Transformations

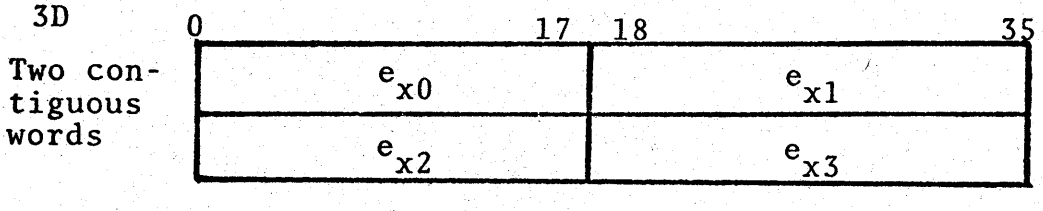
The Matrix Multiplier works on "homogeneous coordinates" (see appendix III.) In homogeneous coordinates three-dimensional coordinate data are represented by the four-component vector (X Y Z W), where X, Y, and Z are the normal orthogonal distances from the origin and W is used as a scale factor. The transformation matrix is the 4 x 4 matrix in position A. When the Matrix Multiplier is in three-dimensional operation and "active", all coordinate data values are multiplied by the matrix stored in position A (see figure 3.1). Note that this does not include parameter data for group 3 instructions. The form of the transformation and the equations which define this transformation are given in figure 3.2. In 3D entire rows of the matrices are affected by a "load", "store", "sink", or "retrieve" instruction, (i.e. four components are loaded at a time).

It should be noted that while the Matrix Multiplier expects input of the form (X Y Z W), The Clipping Divider expects ($X_\varepsilon$ $Y_\varepsilon$ Z W). The transform matrix can easily be structured so that it will make this change.

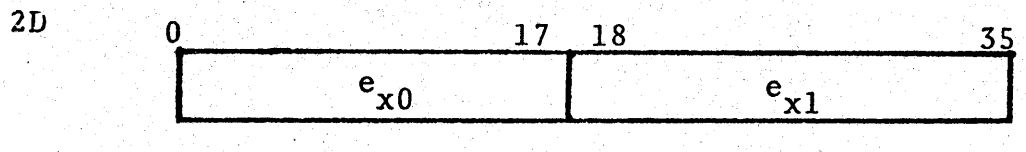
MATRIX MULTIPLIER REGISTERS




Matrix data is stored in memory in the format:




x = number of row as indicated above




Note: In 2D $e_{x2}$ and $e_{x3}$ are inaccessible

Figure 3.1

THREE-DIMENSIONAL MATRIX TRANSFORMATIONS

$$[X\ Y\ Z\ W] \begin{bmatrix} r_{00} & r_{01} & r_{02} & h_{03} \\ r_{10} & r_{11} & r_{12} & h_{13} \\ r_{20} & r_{21} & r_{22} & h_{23} \\ t_{30} & t_{31} & t_{32} & h_{33} \end{bmatrix} = [X'\ Y'\ Z'\ W']$$

Where

$$X' = r_{00}X + r_{10}Y + r_{20}Z + t_{30}W$$

$$Y' = r_{01}X + r_{11}Y + r_{21}Z + t_{31}W$$

$$Z' = r_{02}X + r_{12}Y + r_{22}Z + t_{32}W$$

$$W' = h_{03}X + h_{13}Y + h_{23}Z + h_{33}W$$

r = rotation terms

t = translation terms

h = homogenous terms

Figure 3.2

## 3.3 Two-dimensional Matrix Transformations

Two-dimensional coordinate data can also be transformed by the Matrix Multiplier. The "boxing" operation of the Clipping Divider (see section 4.5) is, however a more efficient way to effect two-dimensional transformations which do not involve rotations. For two-dimensional operation the input is made up simply of the X and Y coordinate values. These values are augmented (by the Matrix Multiplier) to take the form:

[X Y 1] for absolute data
[X Y 0] For relative data.

Figure 3.3 shows the structure of the two-dimensional transformation matrix, the equation for the transformations performed, and the Trigonometric values for the elements.

In 2D only the first two elements of each column in matrix A are loaded from a single word in memory. (See figure 3.1). The zeros and ones shown in the third column of the transformation matrix in figure 3.3 are not actually present but shown only for expository purposes.

## 3.4 Composite Transformations

When an object within the drawing is to be transformed with respect to the drawing and the drawing itself is also to be transformed, a composite transformation of the form

$$[X\ Y\ Z\ W]\ [T_1]\ [T_0] \longrightarrow [X'\ Y'\ Z'\ W']$$

is required. Instead of generating the intermediate result, $[X\ Y\ Z\ W]\ [T_1]$, and then multiplying it by $[T_0]$, the Matrix Multiplier can form the composite transformation $[T_1]\cdot[T_0]$. This is done by executing a "load product" instruction (see chapter 7). The load product instruction takes the matrix $[T_1]$ which is stored in memory, and multiplies it by $[T_0]$, which can be specified as either matrix B, C, or D (but not A). The resulting matrix is left in matrix A.

### 3.4.1 Nested Transformations

This method of forming composite transformations generalizes to any level. The "data sink" operated by the Channel Control (see section 2.8) serves as a pushdown stack for storing matrices in order to implement nested transformations. The sink and retrieve instructions for the Matrix Multiplier contain a "slide" option which allows matrix A and some other matrix (usually B) to be operated as the first two matrices in a pushdown stack. The slide option copies matrix A into another matrix (e.g. B) as that matrix is "sinked" into the data sink. Then when matrix B is retrieved from the data sink, the matrix in position B is copied back into A. The slide versions of the "sink" and "retrieve" instructions, together with the "product load", facilitate a recursive subroutine call with only a few instructions.

## TWO-DIMENSIONAL MATRIX TRANSFORMATIONS

$$[X\ Y\ (W)] \begin{bmatrix} r_{00} & r_{01} & 0 \\ r_{10} & r_{11} & 0 \\ t_{20} & t_{21} & 1 \end{bmatrix} = [X'\ Y'\ (W')]$$

Where

$$X' = r_{00}X + r_{10}Y + t_{20}(W)$$

$$Y' = r_{01}X + r_{11}Y + t_{21}(W)$$

W' is not computed

r = rotation terms

t = translation terms

w = is not provided by input, but rather augmented by the Matrix Multiplier

w = 1 for absolute

w = 0 for relative

Form of 2D Transformation Matrix

$$\begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ F_x & F_y & 1 \end{bmatrix}$$

Figure 3.3

### 3.4.2 Row-to Row Moves

Rows of matrix A may be copied into another matrix by the "push Matrix Multiplier" instruction, and similarly rows of one of the other matrices can be copied back into matrix A by the "pop Matrix Multiplier" instruction, thus allowing matrices B, C, and D to be used as pushdown storage. This feature can be used in special cases where subroutine depth is limited. The additional speed obtained in this manner by avoiding memory references is paid for by a loss of generality in the subroutine calls.

### 3.4.3 Matrix Normalization

Since the Clipping Divider performs perspective division yeilding $X/Z_X$ and $Y/Z_y$, homogenous transformation matrices may be scaled without effecting the transformation performed. It is customary to normalize the matrices used so that at least one element is between one-half and one in magnitude (taking matrix elements as signed fractions; see section 3.8). The multiplication of two such matrices may result in a matrix which is no longer normalized. Renormalization of this matrix before it is used in some subsequent concatenation will assure that maximum precision is maintained in the new transformation matrix. The "normalize" instruction (see section 7.3) is used to shift the elements of matrix A left until any element is greater than one-half in magnitude or until the "count" given in the normalize instruction runs out. The normalize instruction is disregarded in 2D.

## 3.5 Two-dimensional Curves

A two-dimensional curve is defined by the elements held in the first two columns of matrix A (see figure 3.4a). When a group 6 dataless drawing instruction (other than "box") is received, a coordinate value is calculated by an iteration of the matrix according to the equations shown in figure 3.4a and the output is sent to the Clipping Divider (or memory, or both). Usually, a complete curve is drawn with a "polygon" instruction with the Channel Control is repeat mode. In this case the RCR of the Channel Control should be loaded with the two's complement of the number of line segments that are to be in the curve (+1 for the initial setpoint). The class of curves that can be drawn includes all of the conic sections and a few other special curves such as circular and elliptical spirals.

## 3.6 Three-dimensional Curves

Three-dimensional curves are defined using all of matrix A as shown in figure 3.4b. The coordinate values for current location are held on the top row of matrix A. Dataless drawing instructions (other then "box") cause an iteration of the matrix to compute a new coordinate value and send it to the Clipping Divider. Following

2D CURVES

$$A = \begin{bmatrix} r_{00} & r_{10} \\ r_{10} & r_{11} \\ tx & ty \\ x & y \end{bmatrix}$$

Basic Representation

$[x, y] + [tx, ty] \longrightarrow$ Clipping Divider

Set Curve Operation

$[x, y]\ [R] + [tx, ty] \longrightarrow$ Clipping Divider

$[x, y]\ [R] \longrightarrow [x, y]$

Other Drawing Instructions

Figure 3.4a

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

top row specifies current absolute coordinate

Basic Representation

$$[a_{00}\ a_{01}\ a_{02}\ a_{03}] + Q[a_{10}\ a_{11}\ a_{12}\ a_{13}] \longrightarrow [a_{00}\ a_{01}\ a_{02}\ a_{03}]$$

$$[a_{10}\ a_{11}\ a_{12}\ a_{13}] + Q[a_{20}\ a_{21}\ a_{22}\ a_{23}] \longrightarrow [a_{10}\ a_{11}\ a_{12}\ a_{13}]$$

$$[a_{20}\ a_{21}\ a_{22}\ a_{23}] + Q[a_{30}\ a_{31}\ a_{32}\ a_{33}] \longrightarrow [a_{20}\ a_{21}\ a_{22}\ a_{23}]$$

$$[a_{30}\ a_{31}\ a_{32}\ a_{33}] + 0 \longrightarrow [a_{30}\ a_{31}\ a_{32}\ a_{33}]$$

$[a_{00}\ a_{01}\ a_{02}\ a_{03}] \longrightarrow$ Clipping Divider

Iteration

Note: Q is taken from the right half of the MDIR

Figure 3.4b

the perspective division performed by the Clipping Divider (see section 4.5), these cubic difference equations generate a very general class of curves called rational parametric cubics.

## 3.7 Surface Patches

Families of the curves generated in three-dimensional curve mode can be used to draw cross-hatched surface patches. The definition of the surface patch is stored in the matrix array as shown in figure 3.5. The "new curve" instruction is used to generate each new curve of the surface patch.

## 3.8 Arithmetic Conventions

The basic word length of the Matrix Multiplier is 18 bits, which corresponds to the half-word length of the PDP-10. The elements of input vectors and output vectors written into memory are all of this basic word length; however, the outputs to the Clipping Divider carry two extra bits of precision. The reason for this extra precision is that an output vector may be the result of up to 4 accumulating multiplications. The Clipping Divider can accomodate these extra bits in its Matrix Multiplier input. These extra bits are lost, however, if the output vector is written into memory or into the internal matrix storage (as happens when two transformations are concatenated, or during the iteration of curve-drawing difference equations).

All arithmetic operations are performed treating elements as 2's complement signed (fixed point) fractions. Since the word length is 18 bits, the algebraically largest number that can be represented is $1-2^{-17}$, and the algebraically smallest number that can be represented is -1. Data sent to the Clipping Divider range from $4-2^{-17}$ to -4. In binary notation (with the binary point separating the sign bit from the fraction):

0.111111... is the algebraically largest number

0.000000... is the unique representation for zero

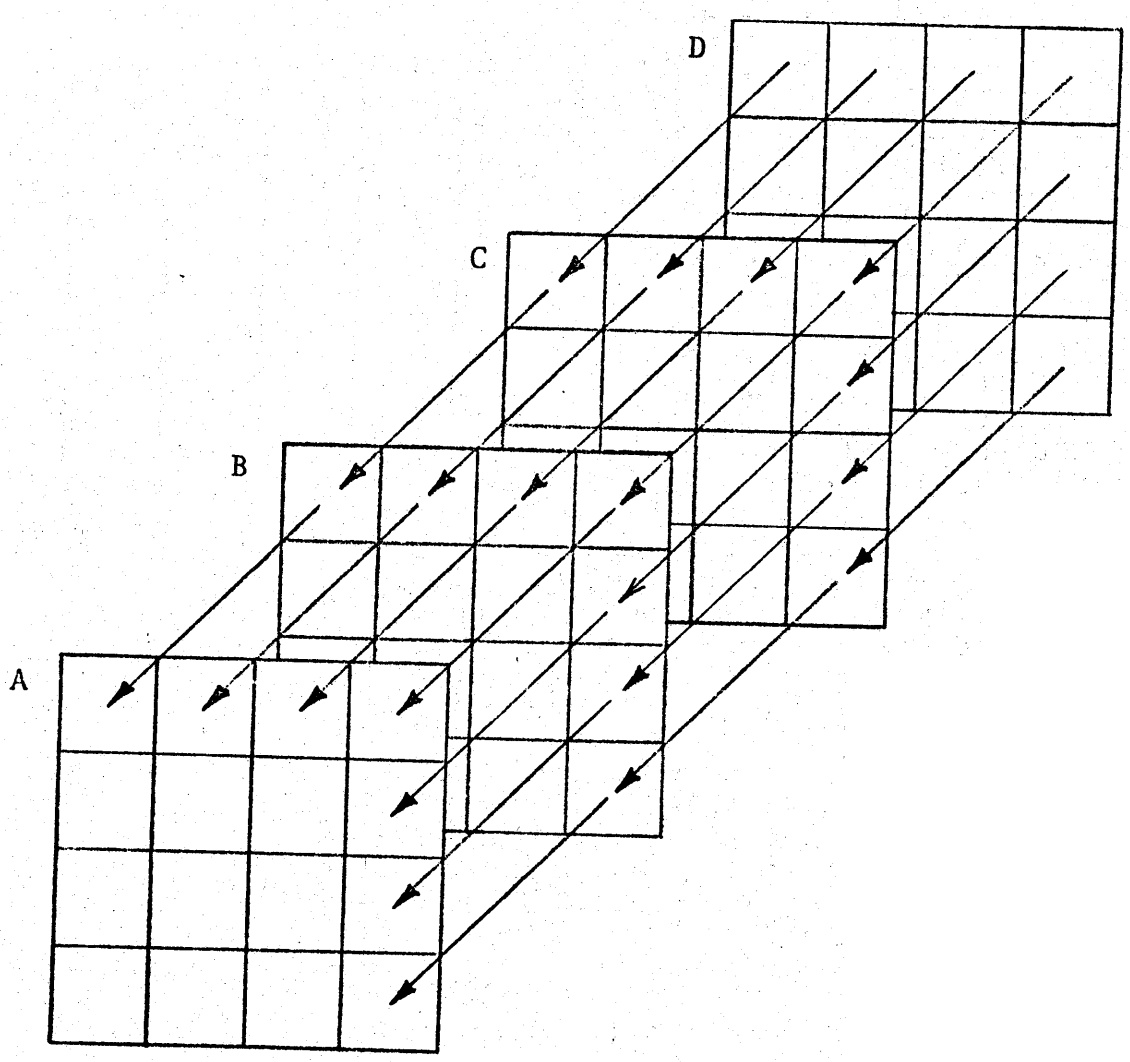
1.000000... is the algebraically smallest number (-1).

If these data were passed directly to the Clipping Divider, they would be leftward sign-extended (000.111111..., 000.000000..., and 111.000000..., respectively) by 2 bits.

The reader should note that the closest approximation to +1 is the fraction 0.111111..., which is close enough to +1 for practical cases.

Two's complement binary multiplication always invokes some questions. The Matrix Multiplier performs fractional multiplication, in which the 17 low-order bits of the product

SURFACE PATCH ITERATION




$$
\left.
\begin{aligned}
A + QB &\longrightarrow A \\
B + QC &\longrightarrow B \\
C + QD &\longrightarrow C \\
D + 0 &\longrightarrow D
\end{aligned}
\right\}
$$

For all 16 elements of each matrix

Note: Q is taken from the MDIR

Figure 3.5

are lost. These bits are used, however, for rounding. Multiplication of -1 by -1 (1.000000...x1.000000...) yields a product of +1 (001.000000..) with the extra precision, but a product of -1 (1.000000...) when truncated. Since the accumulation is carried out with the extra precision, the accumulated result of 4 such multiplications would be -4 (100.000000...), unfortunately, but the accumulation of 3 such multiplications is correct: +3 (011.000000...). It is usually best to avoid -1 altogether.

The practical consequence of using fractional arithmetic is that at least one of the two numbers involved in a multiplication must be a fraction, and the other number may be thought of as having the binary point located at the user's discretion. Figure 3.6 shows a good way to think of the structure of the input vector and the transformation matrix. The advantage of this structure is that both multiplication of the input vector by the transformation matrix and multiplication of one transformation matrix by another results in an integer times a fraction of a fraction times a fraction. In addition, multiplication of one matrix by another gives a matrix of the same form.

## 3.9 Mode Control

The mode of operation of the Matrix Multiplier is controlled both by the Channel Control Directive register (DIR), and by a directive register internal to the Matrix Multiplier (MDIR). In general, the DIR specifies global operating modes, which may apply to several of the operating units in the display system, while the MDIR specifies those modes which apply only to the Matrix Multiplier.

The following bits in the Channel Control DIR have a direct effect on the operations of the Matrix Multiplier:

MMA (Matrix Multiplier Active) -- When this bit is 0, the Matrix Multiplier is "transparent" -- that is, it simply passes its input data on to the Clipping Divider, and provides a level of data buffering in the computational pipeline. Matrix Multiplier load and store operations occur whether or not the MMA bit is set.

3D (3-dimensional operation) -- This bit determines whether the Channel Control fetches a 2-component (2D) or 4-component (3-D) input to the computational pipeline, and affects the subsequent processing accordingly. Please note that this applies both to drawing information and to load/store information.

(SWAP) is a hidden directive bit generated automatically in DO TWICE operation, and causes the Matrix Multiplier to swap the right and left halves of its input data word. Data are swapped whether or not the MMA bit is set.

## FRACTIONAL MULTIPLICATION

$$[X, Y, Z, W] = [I, I, I, F]$$

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & 0 \\ r_{10} & r_{11} & r_{12} & 0 \\ r_{20} & r_{21} & r_{22} & 0 \\ tx & ty & tz & s \end{bmatrix} = \begin{bmatrix} F & F & F & 0 \\ F & F & F & 0 \\ F & F & F & 0 \\ I & I & I & F \end{bmatrix}$$

Where F = Fractions

I = Integers

The coordinates (X, Y, Z) are usually best regarded as integers, while the homogenous term W is usually considered to be a fraction.

The elements of the 3 x 3 submatrix (R), the rotation matrix, are products of sines and cosines and are thus appropriately considered fractions. The translational elements (t) may be thought of as integers since W is a fraction. The "s" term is used to scale the matrix and is a fraction.

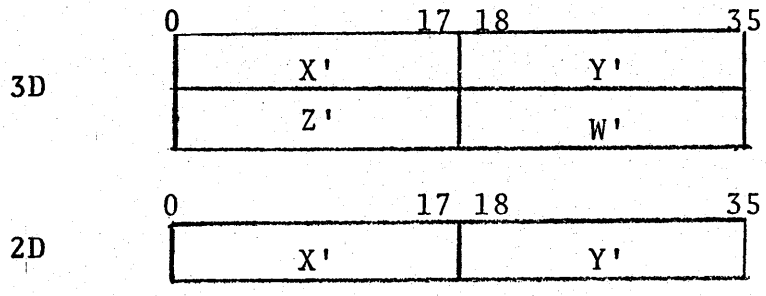
Figure 3.6

The directive information stored internally in the Matrix Multiplier MDIR register is the following:

MOC (Matrix Output to Clipper) -- causes the Matrix Multiplier to send its computational results to the Clipping Divider. This bit is ignored if MMA=0, in which case the Matrix Multiplier is "transparent" and always sends data to the Clipping Divider.

MOM (Matrix Output to Memory) -- causes the Matrix Multiplier to send its computational results to memory. This bit is ignored if MMA=0. The MOC and MOM bits are mutually independent, so it is possible to route the matrix output to the Clipping Divider, to memory, to both, or to neither.

Matrix Multiplier output to memory takes the following format:




CURVE (Curve Mode) -- causes the Matrix Multiplier to interpret drawing instructions as commands to iterate difference equations.

TR1, TR0 (Transpose Map) -- are interpreted as a 2-bit number which controls the addressing into the matrix scratchpad memory. They may be thought of as causing the array to be transposed about any one of its three diagonals. The matrix elements $a_{00}$, $b_{11}$, $c_{22}$, and $d_{33}$ remain in the same place, for any transposition, but the other elements are reflected in the following way:

| TR1 | TR0 | |
|---|---|---|
| 0 | 0 | -- no transposition |
| 0 | 1 | -- rows and columns are exchanged (i.e. matrices A, B, C, and D are each transposed). |
| 1 | 0 | -- columns and rods are exchanged. |
| 1 | 1 | -- rods and rows are exchanged. |

The planes about which the elements are reflected are shown in figure 3.7.

TRANSPOSITION PLANES

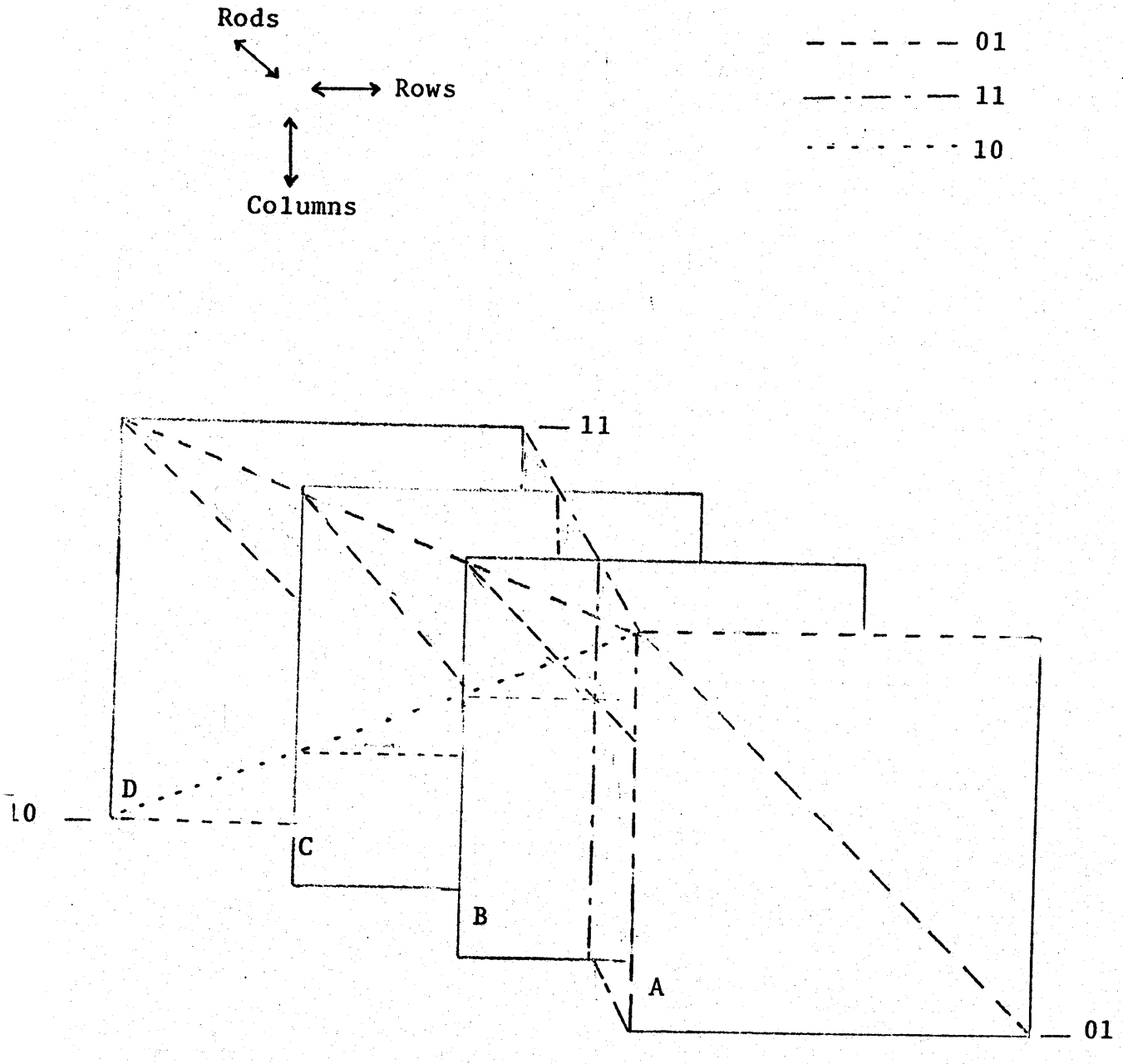
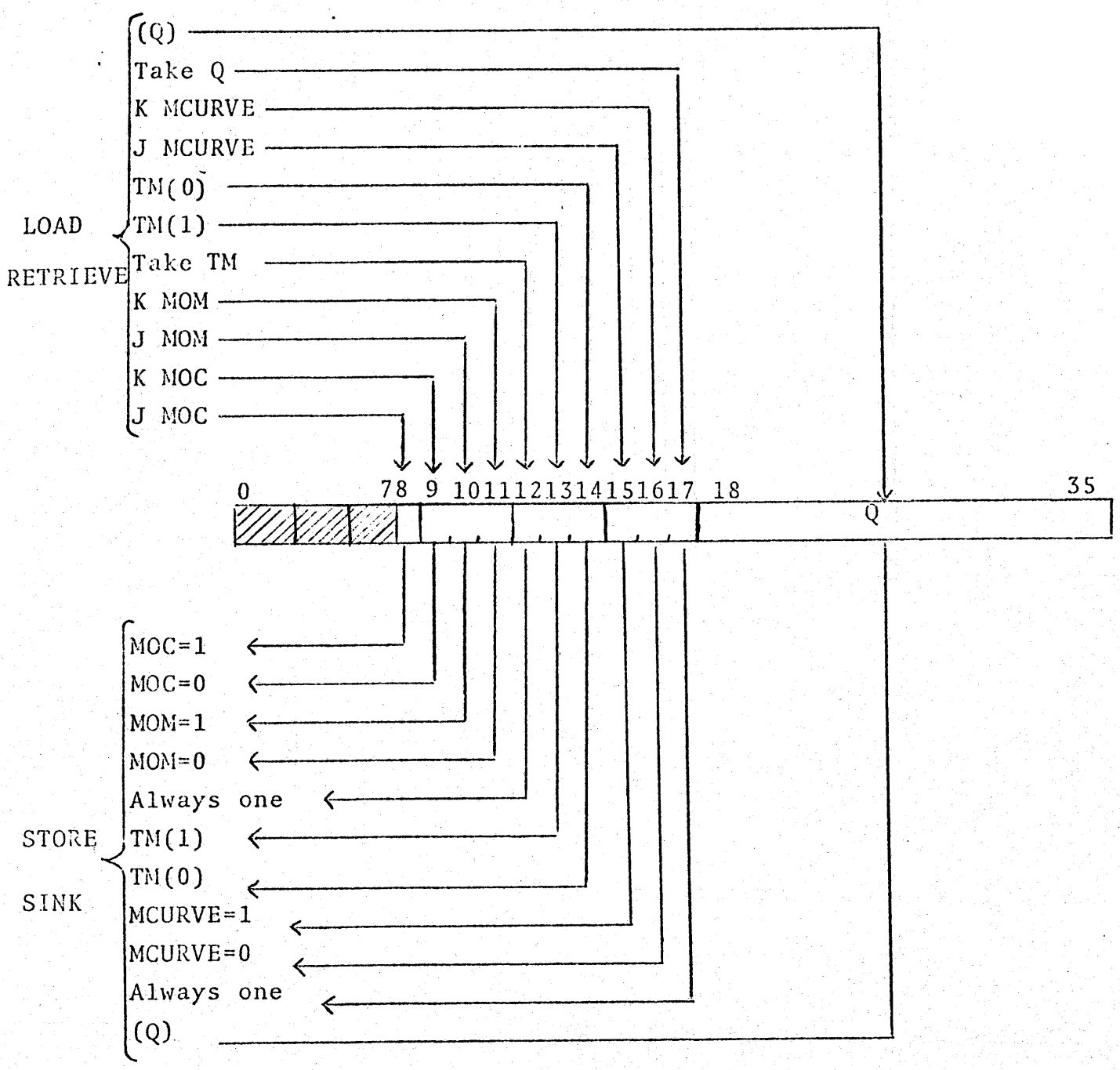



Figure 3.7

The MOC, MOM and CURVE bits and the transpose map are coded into the MDIR word in a special way, which permits the programmer to change one of them without knowing the values of the others. The right half of the MDIR is a numerical quantity, called Q, which is used in the 3D curve drawing operation. The left half of the MDIR register contains the actual directive coding, in the form shown in figure 3.8. Please note that if the MDIR register is stored (or sinked), and later is loaded (or retrieved) from data written, it will be restored to its original contents.

THE MDIR REGISTER




Note:

| J | K | Next |
|---|---|---|
| 0 | 0 | no change |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | complement |

Figure 3.8

# CHAPTER 4

# THE CLIPPING DIVIDER

## 4.1 Function

The Clipping Divider eliminates those portions of the drawing which lie outside the field of view and maps the remaining portion of the drawing into scope coordinates. Input data comes from the Matrix Multiplier (or The Channel Control if the Matrix Multiplier is not included in the system), and output goes to the Line Generator, back to memory via the Channel Control, or both.

## 4.2 The Current Point

The coordinates of the current point which are retained by the LDS-1 are stored in the SAVE register of the Clipping Divider. The Clipping Divider processes lines (dots being treated as lines of zero length). In most cases the current point serves as one end of the line and the new point, defined by the incoming data, serves as the other end of the line. The SAVE register is automatically updated by drawing instructions as explained in chapter 8. The address and structure of the SAVE register are shown in figure 4.1.

## 4.3 Relative Data

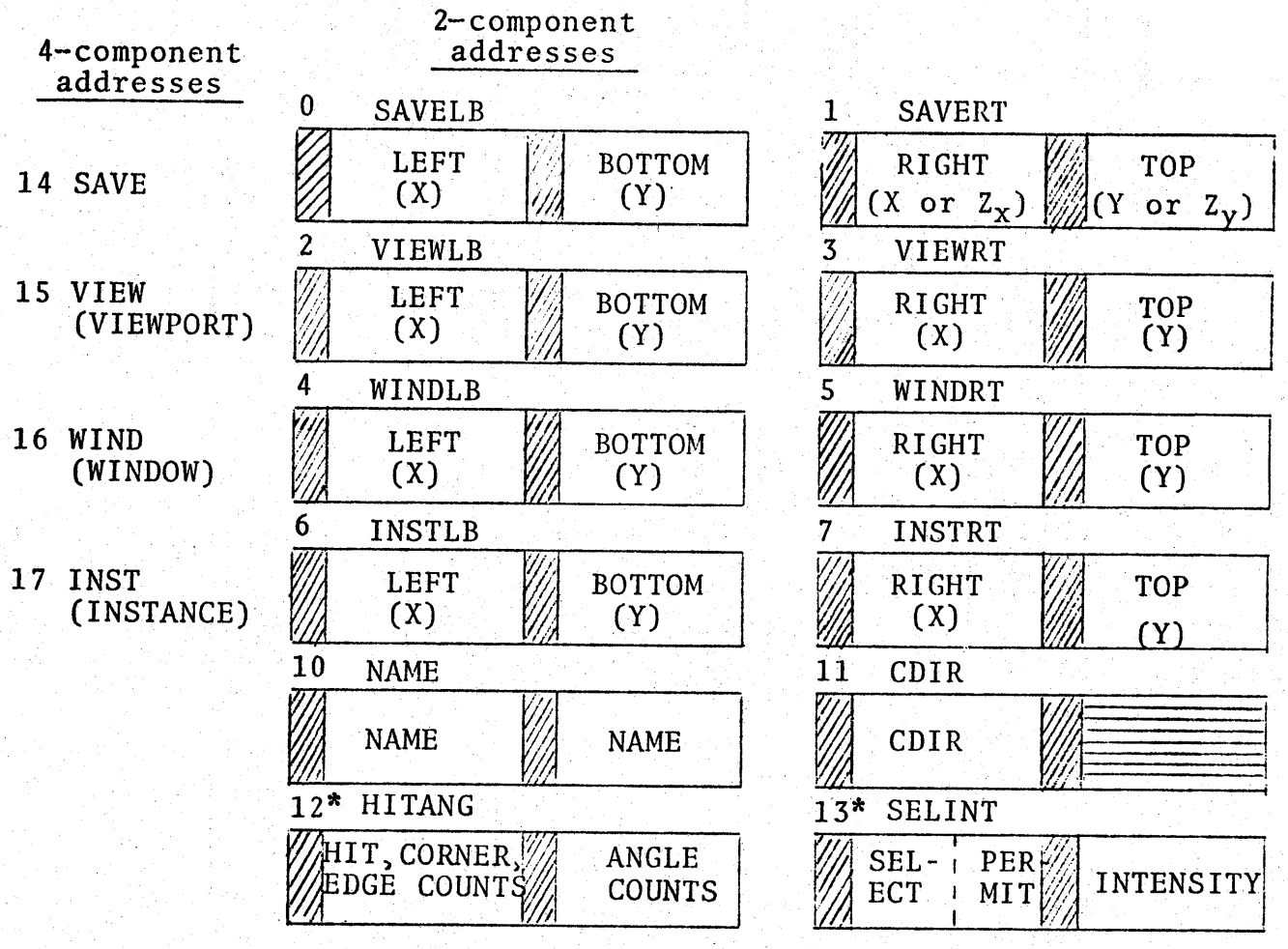
The current point also serves as a reference point for relative data specifications. For relative drawing instructions, the incoming data is first added to the contents of the SAVE register and the result is used as the endpoint. For relative parameter data (e.g. the window), data is also first added to the contents of the SAVE register and the result is used to load the parameter register.

The "size relative" specification, which is used mainly for parameter data, results in the data first being added to the SAVE register contents, for the second two components, and then subtracted from the contents of the SAVE register to give the first two components.

## 4.4 Two-dimensional Clipping and Division

In two-dimensional operation the Clipping Divider automatically eliminates portions of the drawing which lie outside a rectangular area of the drawing space, or "page". This area on the drawing space is known as the WINDOW. The user is able to specify what part of the drawing space he wishes to view by specifying a window in page coordinates which covers that area. The window is specified by giving the page coordinates for its left, bottom corner and its right, top corner. These values are loaded into the WINDOW register of the Clipping Divider.

## CLIPPING DIVIDER REGISTER CONFIGURATION




DATA FORMATS

2D

| 0 — 17 | 18 — 35 |
|---|---|
| LEFT or RIGHT (X) | BOTTOM or TOP (Y) |

3D

| | |
|---|---|
| LEFT (X) | BOTTOM (Y) |
| RIGHT ($X$ or $Z_x$) | TOP ($Y$ or $Z_y$) |

**Note:** The numbers associated with the registers are octal addresses. The names associated with the registers are LDS-1 mnemonics which have been defined in MACRO-10.

Figure 4.1

The user may specify the rectangular portion of the scope on which he wishes the picture to appear. This area on the scope is known as the viewport. The viewport is specified by loading the VIEWPORT register with the scope coordinates of its left, bottom corner and right, top corners. The scope coordinate system is centered about zero and stretches from -3777 to 3777 (i.e. 12 bits) but because the VIEWPORT register is a full 18-bit register (plus a two-bit sign extension) and because only the 12 least significant bits are used to drive the scope, each boundary of the viewport should be specified to be between -3777 and 3777. Specifying a larger viewport results in wraparound and specifying a smaller viewport results in the picture being drawn on less than the full viewing area on the scope.

The relation between the sizes of the window and viewport determines the scale of the drawing. A window specification of -377777, + 377777 (in each axis) and a viewport specification of -3777, + 3777 (each axis) will map the entire page onto the entire viewing area of the scope. If the window is only half as large (in each axis) and the viewport is the same size only ¼ of the drawing appears and the scale is twice as large.

The window and viewport need not be the same "shape". When they are different, the scale will be different in X and Y (to "stretch" the picture in one direction). Furthermore it is possible to create mirror images by specifying a "backward" viewport (i.e. where the value for the left edge is greater than the value for the right edge or the value for the bottom edge is greater than the value for the top edge). Specifying a backward window, however, results in none of the drawing being displayed.

## 4.5 Three-dimensional Clipping and Division

In three-dimensional operation the drawing is compared to a pyramid of vision rather than to the window. The pyramid of vision in defined for positive Z values by the planes $X = +Z$, $X = -Z$, $Y=+Z$, and $Y = -Z$, thus forming a right angle pyramid with its apex at an observation point about 5" from the face of the screen. Any portion of the drawing outside this pyramid of vision is eliminated. Thus only those lines or portions of lines where $|X| \leq Z_x$ and $|Y| \leq Z_y$ are displayed as shown in figure 4.3.

In three-dimensions, perspective division becomes part of the process of mapping the coordinate data into scope coordinates. This perspective division yield $X/Z_x$ and $Y/Z_y$. The viewport operates just as in two-dimensions, controlling the portion of the viewing area of the Display Scope onto which the picture is mapped.

TWO-DIMENSIONAL CLIPPING AND DIVISION

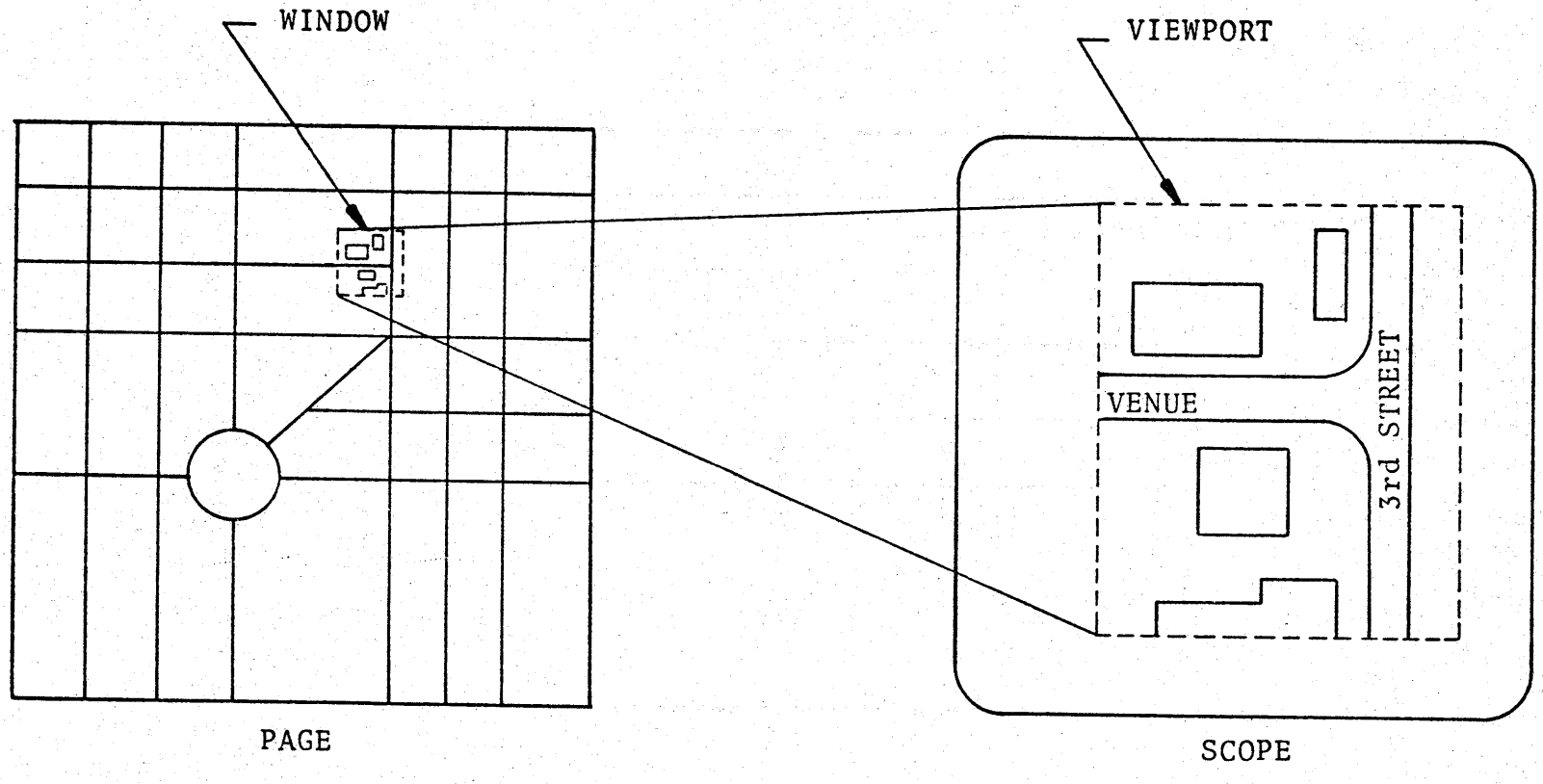
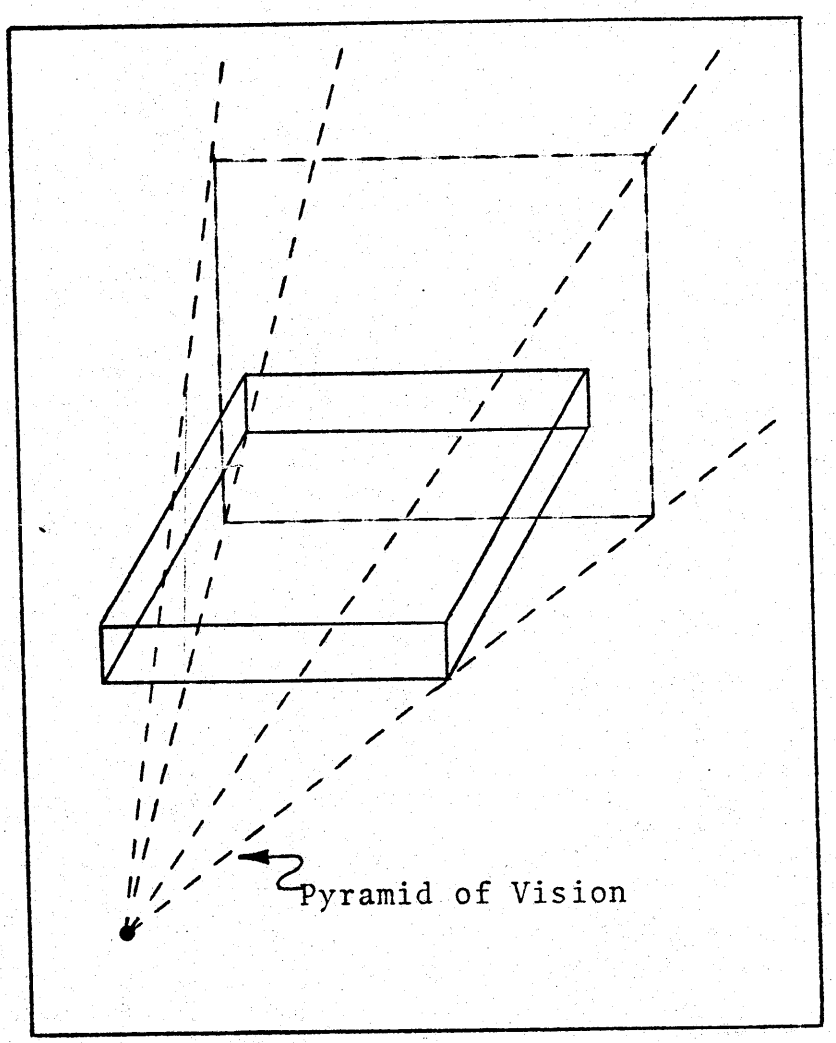
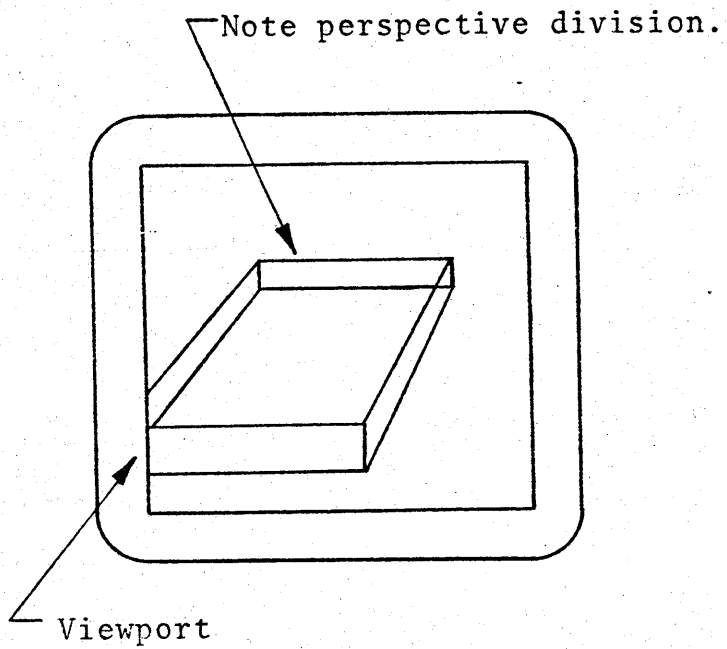



Figure 4.2

THREE-DIMENSIONAL CLIPPING AND DIVISION




Page




Scope

Figure 4.3

It should be noted that because the pyramid of vision is right-angled, the perspective looks strange unless viewed from very close to the scope face (about 5"). Other viewing angles can be implemented by using the transformation

$$Z = Z \tan(\alpha/2)$$

where $\alpha$ is the desired viewing angle.

## 4.6 Boxing

The boxing process is a special feature of the LDS-1 Clipping Divider which allows two-dimensional subpictures to be defined only once but appear in several different sizes and locations. In order to understand boxing it is useful to think of it conceptually as the concatenation of two mappings. The first mapping is from the subroutine definition space, a space similar to the page, onto the page. The second mapping is then the normal page to scope (window to viewport) mapping performed by the Clipping Divider. See figure 4.4.

The area on this subroutine definition space which is to be the domain in the first mapping is deliniated by the MASTER. The master specifies the rectangular portion of the subroutine definition space which is to be mapped onto the page. The area on the page onto which the MASTER is mapped is known as the INSTANCE. Once the subroutine has been mapped onto the page, the normal window-to-viewport mapping will eliminate any portion of the subroutine which lies outside the window and map the result onto the viewport, thus displaying the subroutine at the proper position and size.

The "box" instruction of the LDS-1 automatically sets up the window and viewport to perform a composite mapping. The subroutine is thus mapped directly from the subroutine definition space onto the scope. In order to compute these new parameters, the Clipping Divider must be provided with a master and an instance just as if two successive mappings were to be performed.

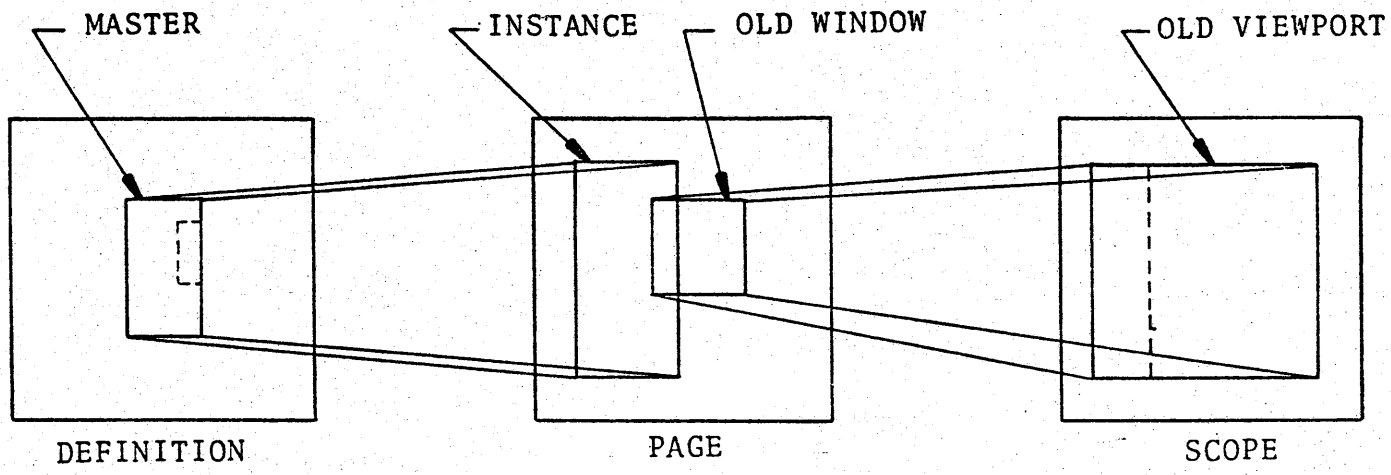
- The Master. The master is specified as a direct parameter of the box instruction (i.e. the data addressed by the box instruction is the master). The master should be specified by giving the left, bottom and right, top corners in the coordinate system of the subpicture to be drawn.

- The Instance. The instance should be loaded into the INSTANCE register of the Clipping Divider prior to executing the box instruction. The instance is specified by giving the page coordinates of its left, bottom and right, top corners.

The box operation results in defining a new window on the subroutine definition space and a new viewport on the scope. After the box instruction has been executed, the program can jump to the subroutine and draw the subpicture just as if it were executing a part of the main drawing routine. The

BOXING

The Two (Conceptual) Mappings




The Composite Mapping Set Up By Boxing

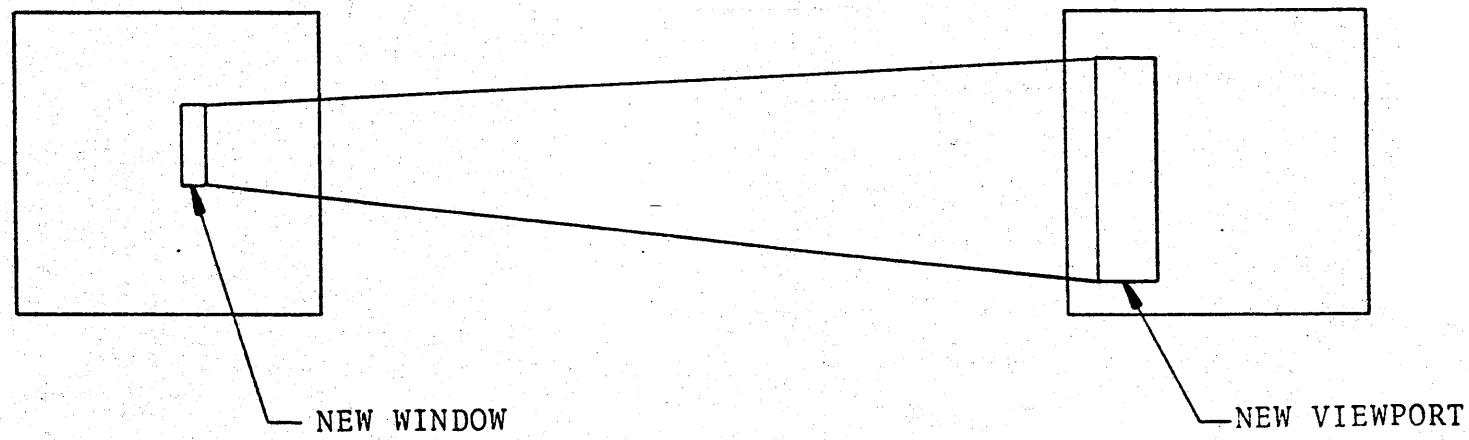



Figure 4.4

subpicture need not be in relative format. The relative size of the subpicture on the main drawing is determined by the ratio of the master to the instance and thus the subpicture can appear in any size. Finally, any part of the subpicture which lies outside the current window is clipped.

When the instance is loaded prior to boxing the Clipping Divider will check to see if there is any area in common between the current window and the instance. If not, there is no need to draw the subpicture and it can be skipped entirely. An "area in common" bit AIC is sent to the STATUS REGISTER of the Channel Control where it can be tested prior to boxing. Please note that for the AIC bit to operate properly the INSTANCE register must be the last register loaded with a 2D component load prior to the box instruction (i.e. no other register should be loaded between the loading of the INSTANCE and testing AIC) and the INSTANCE must be loaded with a 2D four-component load. See section 8.5. The AIC bit is cleared by a new 2D four-component load.

## 4.7 HIT and COUNT Functions

The HIT bit is generated by the Clipping Divider when some portion of the line being generated intersects the current window. This bit is sent to the STATUS REGISTER of the Channel Control where it can be tested. The HIT bit can also be enabled to stop the LDS-1 and cause an interrupt to the PDP-10. (See section 6.3). Once the HIT bit is set it remains on until cleared by a PDP-10 CONO instruction, a group 2 instruction which clears the HIT bit, or by a "load" immediate" of the STATUS REGISTER. The HIT bit thus gives the Clipping Divider the features of an automatic comparator which are very useful for "pointing" functions such as are associated with a tablet. (See chapter 9).

Several different counts that may be useful in examining the geometry of a drawing are maintained in the HITANG register. These counts are primarily useful for determining the relationship between polygons and the current window and thus will be explained assuming that a polygon is being drawn.

- EDGE COUNT. The EDGE COUNT is incremented whenever both ends of the line are outside the window and the line passes through the window.
- CORNER COUNT. The CORNER COUNT is incremented for each corner (i.e. endpoint connecting two lines) within the window.
- HIT COUNT. The HIT COUNT is incremented for each dot within the window or each line which intersects the window.

- ANGLE COUNTS (Q1-Q4). The four angle count registers may be used in conjunction with the other counts to determine how the polygon intersects the window. To understand the angle detection logic it is best to think of radials eminating from the corners of the window as shown in figure 4.5 (note that the radials do not include the edges of the window). Each time a polygon edge crosses the radial in a counter-clockwise direction the count is incremented and each time it crosses in a clockwise direction the count is decremented. The four angle counters are used to hold the accumulated counts for each quadrant (radial). Examples of the use of these registers are shown in figure 4.5

It should perhaps also be noted that in order to make intelligent use of these registers they must be zeroed before the polygon is processed. The HITANG register can be loaded stored, sinked, and retrieved by the instructions of group 3. (Note: These features are provided on a "best effort" basis and their proper functioning is not considered part of the acceptance criteria for the system.)

## 4.8 Scope Control

The SELINT register of the Clipping Divider contains scope selection and intensity information. The first 8 bits are used for scope selection. The next bit is used as a "take" bit for the select bits. If this bit is 0, the select bits are not loaded. It is thus possible to load the intensity bits without loading the select bits. The next 8 bits are used for the scope permit bits. These bits form a mask against which the scope selection bits are tested. If a violation occurs, a scope selection violation signal is generated which can be enabled to cause interrupt of the PDP-10 (see chapter 6). The permit bits can only be loaded with an I/O instruction and thus in a time-sharing environment are only accessible to the monitor.

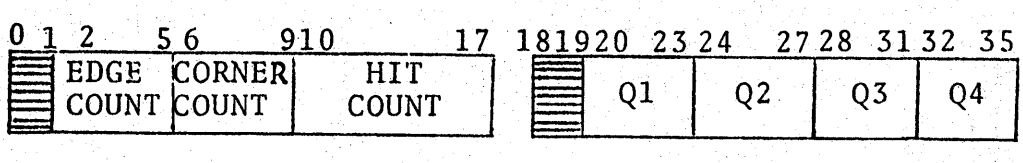
The last 18 bits of the SELINT register are used to specify the intensity. However, only the most significant 12 bits are actually used (see section 5.2.1). Zero specifies greatest intensity, 7777 specifies least intensity. The format for the SELINT register is shown in figure 4.5.

## 4.9 The NAME Register

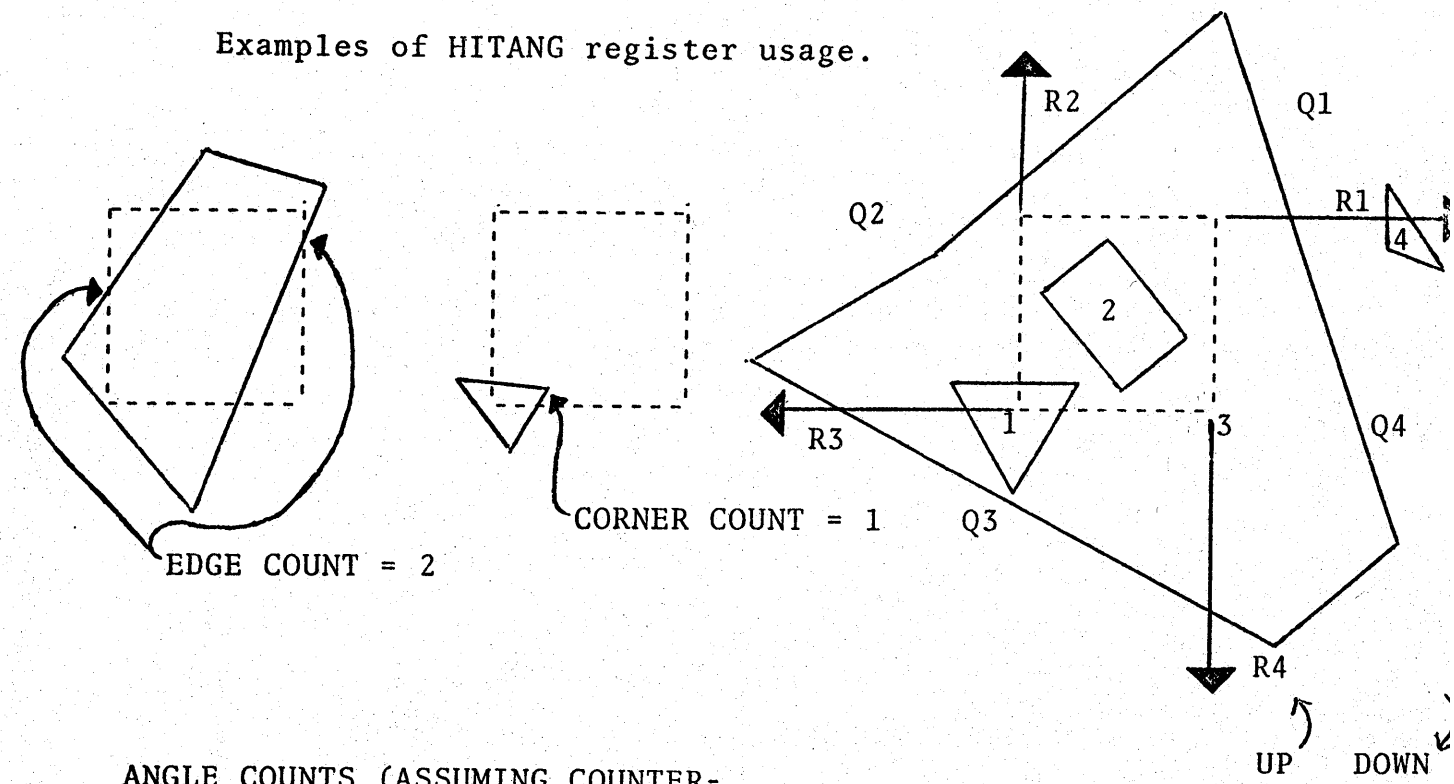
The NAME register of the Clipping Divider is an unassigned register which can be used by the programmer as a storage register. The NAME register can be loaded, stored, sinked, or retrieved.

## HITANG and SELINT REGISTERS

### HITANG REGISTER




Examples of HITANG register usage.




ANGLE COUNTS (ASSUMING COUNTER-CLOCKWISE TRACE)

| | | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|---|
| 1. Intersects the window | 1. | 0 | 0 | 1 | 0 |
| 2. Entirely within the window | 2. | 0 | 0 | 0 | 0 |
| 3. Entirely surrounds the window | 3. | 1 | 1 | 1 | 1 |
| 4. Outside the window | 4. | 0 | 0 | 0 | 0 |

---

### SELINT REGISTER

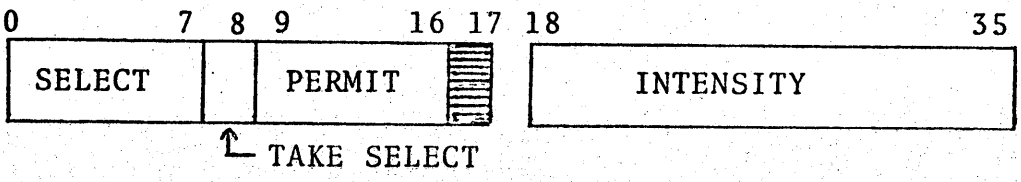



Figure 4.5

## 4.10 Graph Mode

The Clipping Divider can be put into "graph mode" by specifying "self X" or "self Y" in the Clipping Divider directive register (see next section). In this mode either the X or the Y values in the SAVE register (or both) are incremented by the corresponding X or Y value in the INSTANCE register to form the new point, and the X or Y part of the incoming data is ignored. For more efficient storage of data in this mode, the DO TWICE bit of the Channel Control DIRECTIVE register should be set. In all self modes all drawing instructions should be relative. Also, both the X components and both the Y components of the INSTANCE registers should be loaded with $\Delta X$ or $\Delta Y$.

If SELFY and DO TWICE are set, the data should appear as follows:

$X_1$, $X_2$
$X_3$, $X_4$
etc.

If SELFX and DO TWICE are set, the data must appear as:

$Y_2$, $Y_1$
$Y_4$, $Y_3$
etc.

The reason for this is that data is always taken normally the first time and reversed the second time.

## 4.11 Mode Control

The 3D bit of the Channel Control DIRECTIVE (DIR) register determines whether the Clipping Divider is in 2D or 3D mode. The rest of the mode control information is stored in the Clipping Divider directive register (CDIR).

The bits of this register are as follows:

| | | |
|---|---|---|
| 0-1 | Unused | |
| 2 | STOS | Scaled output to scope. |
| 3 | STOM | Scaled output to memory (see figure 4.6 for format). |
| 4 | ZTOS | Z sent to scope to control intensity. (Otherwise the intensity bits of the SELINT register control intensity). |
| 5 | PTOM | Clipped page coordinates (before division) to memory (see figure 4.6 for format). |
| 6 | NTOM | NAME register contents to memory (see figure 4.6 for format). |
| 7 | Take bits 2-6 | If not set bits 2-6 are not loaded. |
| 8 | J (Set) CURVE | If CURVE mode is set in 3D, the Clipping Divider calculates the part of the drawing within the negative Z pyramid as well as the positive Z pyramid. The result is that the drawing behind the observer is also projected onto the scope. This feature is useful in displaying certain types of curves. CURVE for the Clipping Divider should not be confused with MCURVE for the Matrix Multiplier. |
| 9 | K (Clear) CURVE | |
| 10 | J (Set) MEF | Minimum Effort Mode is a special mode where the Clipping Divider merely computes the X, Y and Z coordinates for some point which is visible on the specified line. (PTOM should be set to get these values into memory). |
| 11 | K (Clear) MEF | |
| 12 | J (Set) Dashed Line | Causes the line drawn on the scope to be dashed rather than solid. |

| | | |
|---|---|---|
| 13 | K (Clear) Dashed Line | |
| 14 | Unused | |
| 15 | SELF X | Use INSTANCE register for ΔX displacement. |
| 16 | SELF Y | Use INSTANCE register for ΔY displacement. |
| 17 | Take SELF | If not set SELF bits are not loaded. |

FORMAT FOR CLIPPING DIVIDER OUTPUT TO MEMORY

PTOM (Clipped page coordinates)

| | 0 — 17 | 18 — 35 | |
|---|---|---|---|
| Previous Point | X | Y | |
| | Z | Z | * |
| New Point | X | Y | |
| | Z | Z | * |

NTOM (Name Register)

| NAME L | NAME R |
|---|---|

STOM (Scaled scope coordinates)

| | | |
|---|---|---|
| Previous Point | X | Y |
| New Point | X | Y |

IF all three are set, data are deposited on the order shown.

Dots are recorded only <u>once</u> (i.e. New Point)

*Omitted if 2D set.

Figure 4.6

# CHAPTER 5

# THE LINE GENERATOR AND DISPLAY SCOPE

## 5.1 Function

The last units in the LDS-1 processing pipeline are the Line Generator and Display Scope. The Line Generator accepts digital input from the Clipping Divider, converts these to analog signals and generates the sweep voltages required to drive the deflection system of the Display Scope. Input includes 12 bits of X, 12 bits of Y, and 12 bits of Z intensity, as well as scope selection data, MOVE/DRAW commands, and the DASHED LINE command.

## 5.2 Control

The programmable control for the Line Generator and Display Scope is contained in the Clipping Divider.

### 5.2.1 Intensity

The intensity modulation of the line drawn on the Display Scope is under program control in one of two ways. First, if the ZTOS (Z to scope) bit of the Clipping Divider directive register (CDIR) is set, the Z value of the line is used to modulate intensity. This "depth cueing" makes the intensity of any point on the line a function of the Z coordinate of that point. Thus lines that extend very far from the observation point will grow dim at the far end.

If ZTOS is not set, the most significant 12 bits of the value stored in the INTENSITY register (i.e. the right 18 bits of the SELINT register) of the Clipping Divider are used to determine intensity.

### 5.2.2 Scope Selection

The Line Generator can drive up to four scopes. The selection for these scopes is determined by the Select register (bits 0-7 of SELINT) of the Clipping Divider. These bits are masked against the bits in the Permit register (bits 9-16 of SELINT) and in the case of violation, a scope select violation bit is sent to the Channel Control. This bit can be enabled so that it will cause an interrupt (see section 6.2). The permit bits can be set only via an I/O instruction and are thus protected. For the format of the SELINT register see figure 4.5. A line can be displayed on any combination of the available display scopes.

### 5.2.3 Beam Control

The Clipping Divider controls the movement of the beam on the Display Scope. The "set point" and drawing instructions received by the Clipping Divider are used to control the MOVE/DRAW function of the Line Generator. The clipping process insures that the Line Generator will not be fed values which are off the edge of the viewing area of the Display Scope.

The Display Scope can be made to draw a dashed line (instead of a solid one) by setting the DASHED LINE bit of the Clipping Divider directive register.

# CHAPTER 6

# LDS-1/PDP-10 INTERFACE

## 6.1 General

The LDS-1 is interfaced directly to the memory port of the PDP-10 (DMA) or to a memory part multiplexor. In addition, the LDS-1 is interfaced to the I/O path of the PDP-10 in order to allow I/O communications. Minor modifications to the PDP-10's monitor and extensions to MACRO-10 through the use of OPDEF's provide the necessary adjustments so that the system software of the PDP-10 can accommodate the LDS-1.

## 6.2 Hardware Interfaces

The LDS-1 operates off the memory of the PDP-10. Instructions and data are fetched by the LDS-1 on a "cycle stealing" basis. The LDS-1 provides a memory address and makes a read or write memory cycle request. The memory address provided is an 18-bit unsigned number. If the memory protection and relocation option is included in the system, this address is first mapped; otherwise, the address is taken as the effective address. See Chapter 10 for a description of the memory protection and relocation functions. When the memory cycle is granted, the information passes either to or from the LDS-1 through the DMA of the PDP-10. All data is buffered by the LDS-1.

If the LDS-1 attempts to address non-existent memory, the PDP-10 will not respond with a memory cycle and the LDS-1 will hang. The NXM indicator light on the Channel Control control panel will go on. The LDS-1 also checks for parity errors and lights the PARITY ALARM light if a parity error occurs. Both of these conditions may be detected by the CONI instruction.

I/O communications take place through the I/O interface. Several conditions within the LDS-1 can be enabled by the PDP-10 CONO instruction word to cause a PDP-10 interrupt. System clear and the priority interrupt assignment are also implemented through the CONO instruction word (see Figure 6.1). When an interrupt occurs or when it is necessary to test some condition of the LDS-1, a PDP-10 CONI instruction word is used (see Figure 6.2). The CONI word can be tested by CONSO or CONSZ as explained in the PDP-10 Reference Manual.

Considerable care has been exercised to insure that the display system will operate gracefully in a multiple-user or time-shared environment. The display may be interrupted by the CPU at the end of any instruction, and during the execution of a repeat mode sequence or multiple LOAD, STORE,

SINK, or RETRIEVE. Instructions can be terminated and later resumed, since the state of the instruction execution is saved in the repeat status register of the Channel Control. The CPU may inject an "I/O stop" request by means of a CONO instruction, and may then give commands to the display processor with DATAO instructions. The full I/O word transferred by a DATAO is interpreted as a standard LDS-1 instruction. "Push" and "store" instructions issued in this way can be used to save the entire state of the LDS-1 in memory so that after processing another user's material, it is possible to resume the interrupted process. A program to interrupt and restore a user is shown in the programming examples.

The display processor includes provision in its memory and I/O bus interfaces for memory protection and relocation. The memory protection and relocation option is described in Chapter 10.

## 6.3 System Software Interface

In order to properly handle the LDS-1, minor modifications must be made to the PDP-10's monitor. These modifications are described in a separate document.[1]

Programs for the LDS-1 are prepared by the MACRO-10 assembler. LDS-1 instructions are defined in OPDEF's which are used to assemble LDS-1 code from the LDS-1 mnemonics. The mnemonics commonly used for LDS-1 instructions are given in Chapters 7 and 8 of this manual. It should be realized, however, that not all possible instructions have mnemonics defined for them and that there is, in a sense, nothing sacred about the mnemonics which have been defined; that is, they may be freely changed to fit the needs of a particular system.

It should also be remembered that, once initialized, the LDS-1 acts as an autonomous processor, interpreting its own program. The PDP-10 can also be running a program. Both programs may access the same data, but it is, however, important that the LDS-1 does not try to execute PDP-10 code and vice-versa, as disaster may result.

---

[1]See LDS-1/PDP-10 monitor modifications.

CONO BITS

18 - System clear. This has the same effect as the console I/O reset switch and the clear switches on the Clipping Divider and Channel Control. The clipper and processor are cleared and the processor is sent to the STOP state while the clipper is sent to the INPUT WAIT state. Two successive clears are necessary if the clipper has been operating in the 3D mode. The clipper will not finish the line it is working on, nor will the processor complete its instruction. No information is lost in the processor.

19 - Allow Memory Alarm Interrupt. This allows non-existent memory and parity errors to cause the host computer to interrupt on the selected channel.

20 - Disallow Memory Alarm Interrupt.

21 - Enable Memory Protection and Relocation.

22 - Allow Map/Protect Interrupt. This bit is used in connection with memory protection.

23 - Disallow Map/Protect Interrupt.

24 - Set I/O Stop.

25 - Allow Stop Interrupt.

26 - Disallow Stop Interrupt.

27 - Clear I/O Stop.

28 - Clear Program Stop.

29 - Clear Hit.

30 - Step. If the Channel Control is stopped (due to an I/O Stop) a CONO instruction with this bit set causes the Channel Control to execute one instruction and then return to the STOP state.

31 - Unused.

32 - Allow Priority Interrupt Assignment.

33, 34, 35 - Priority Interrupt Assignment.

Figure 6.1

CONI BITS

0-3 - Unused

4-17 - Unused

18 - Parity Alarm

19 - NXM Alarm (non-existent memory)

20 - Alarm Interrupt On

21 - Map/Protect Violation

22 - Map/Protect Interrupt On

23 - Unused

24 - Stopped and Ready (DATAO may be given only if this bit is on, otherwise the DATAO is ignored)

25 - Stop Interrupt On

26 - Memory To Memory Stop

27 - I/O Stop

28 - Program Stop

29 - Hit Stop

30 - Scope Select Violation Stop

31 - Unused

32 - LDS-1 Caused Interrupt (i.e. Interrupt has actually occurred)

33, 34, 35 - Priority Interrupt Assignment

Figure 6.2

# CHAPTER 7

# THE INSTRUCTION SET - STRUCTURAL BREAKDOWN

## 7.1 General

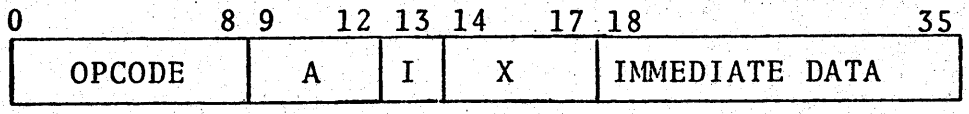
The instruction set of the LDS-1 can be easily broken down by the codes in bits 0-2. The contents of this three-bit field designate the group number of the instructions. This is a meaningful division because group number determines how the rest of the instruction word is to be split into fields and what those fields are to mean. Furthermore, in a very formal sense, group number indicates the general function of the instruction. These functions are:

| Group | Function |
|---|---|
| 0 | Channel Control Register Transmission |
| 2 | Conditional Loading and Condition Modification |
| 3 | External Device Register Transmission |
| 4 | Direct Address Drawing Instructions |
| 5 | Indirect Address Drawing Instructions |
| 6 | Matrix Multiplier Curve Mode Drawing Instructions |
| 7 | Character String Interpreter Drawing Instruction |

Note that Group 1 has been reserved for later system expansion.

LDS-1 instructions conform to the format of PDP-10 assembly language instructions and are decoded by the same assembler.

Instructions are held in 36-bit words with the following format:

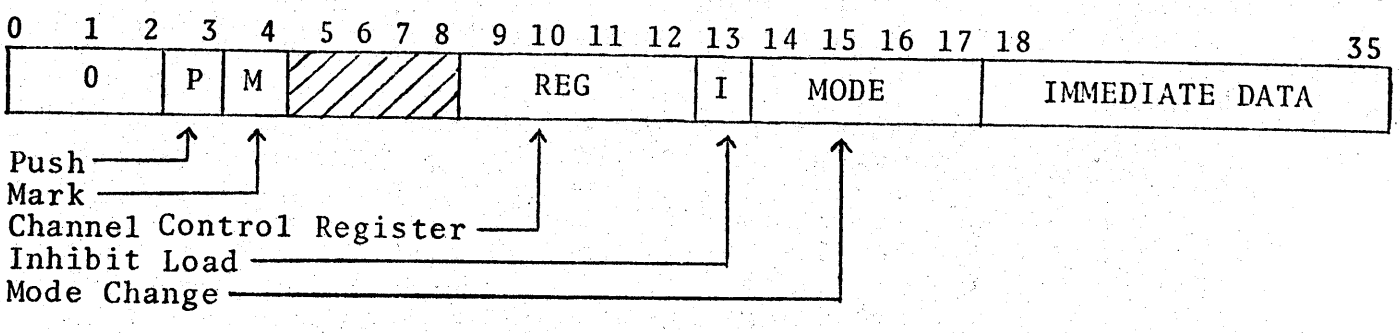



The assembly language fields are as follows:

```
LABEL:  OPCODE  A,@DATA(X); COMMENT
```

The @ is present if the "I" bit is to be set and absent otherwise. Spaces are ignored except that at least one space must follow the OPCODE. Missing fields are compiled as zeroes. If the A field is absent the comma may be omitted.

## 7.2 Group 0




The sequence of operations performed is shown below.

1. The contents of the Channel Control register addressed by the REG field are placed in the temporary Channel Control Register P2.

2. If the I bit is zero, the immediate data are placed into the Channel Control register addressed by the REG field.

3. If the P bit is one and the M bit is zero, the Channel Control register is pushed onto the stack; i.e., the SP (STACK POINTER) is decremented and the following instruction is written into the location it addresses:

LI A, OLDATA

where A is a copy of the REG field and OLDATA is the previous contents of that register, held in P2.

4. If the P bit is one and the M bit is also one, action is taken as in 3, but the instruction pushed is:

LI A, OLDATA(OLDMOD)

where OLDMOD is the current mode of the Channel Control. This constitutes marking the pushed instruction.

5. If the MODE field is non-zero, it is used to determine the mode of the next instruction fetch. The MODE field is decoded according to the following table:

| BITS 14-17 | RESULTS |
|---|---|
| 0 0 0 0 | Do nothing |
| 0 0 0 1 | Go to PROG mode, clear EXEQ and REPT |
| 0 0 1 0 | Go to PEEL mode, clear EXEQ and REPT |
| 0 0 1 1 | Clear EXEQ and REPT |
| 0 1 X X | Go to REPT mode, clear EXEQ (XX indicates PROG and PEEL modes as above) |
| 1 0 X X | Go to EXEQ mode, clear REPT |
| 1 1 X X | Go to EXEQ and REPT |

The modes have the following priority:

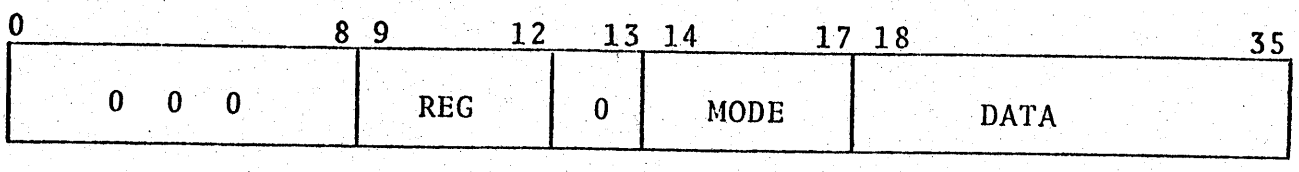
Repeat
Execute (self-clearing, applies only once)
Program } mutually exclusive
Peel }

### 7.2.1 Instructions in Group 0

Mnemonic: LI (Load Immediate)

Assembler definition: [ 0 ]

Structure:




Function: Load immediate data into Channel Control register; change mode (optional).

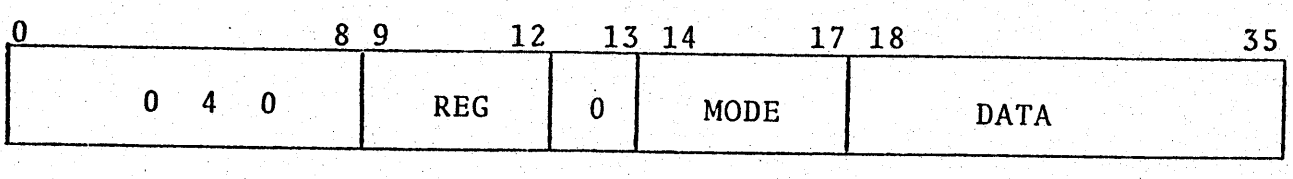
Format: LI REG, DATA(MODE)

where: REG is the register to be loaded
DATA is the immediate data
MODE is the new mode (optional)

Mnemonic: LIPSH (Load Immediate and PuSH old value)

Assembler definition: [040000000000]

Structure:




Function: Load immediate data; push old value; change mode (optional).

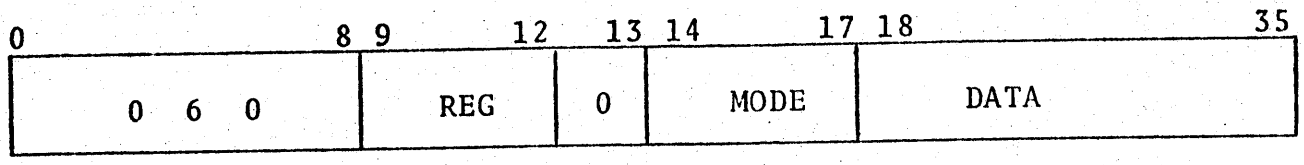
Format: LIPSH REG, DATA(MODE)

where: REG is the register
DATA is the immediate data
MODE is the new mode (optional)

Mnemonic: LIPSHM (load Immediate, PuSH old value, Mark)

Assembler definition: [060000000000]

Structure:




Function: Load immediate data; push old value marked; change mode (optional).

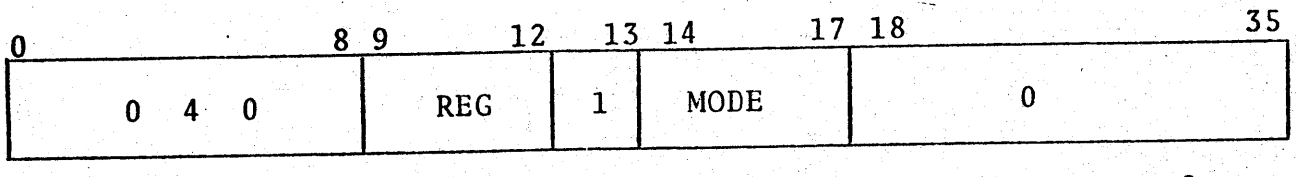
Format: LIPSHM REG, DATA(MODE)

where: REG is the register
DATA is the immediate data
MODE is the new mode (optional)

Mnemonic: PSH (PuSH)

Assembler definition: [LIPSH ,@0]

Structure:




Function: Push Channel Control register leaving the contents of the register unchanged; change mode (optional).

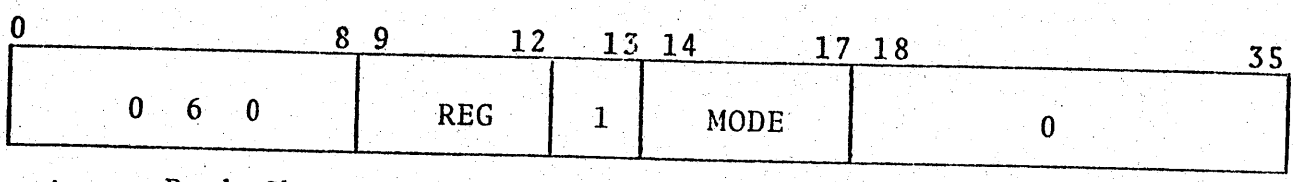
Format: PSH REG, (MODE)

where: REG is the register
MODE is the new mode (optional)

Mnemonic: PSHM (PuSH and Mark)

Assembler definition: [LIPSHM ,@0]

Structure:




Function: Push Channel Control register marked leaving the contents of that register unchanged; change mode (optional).

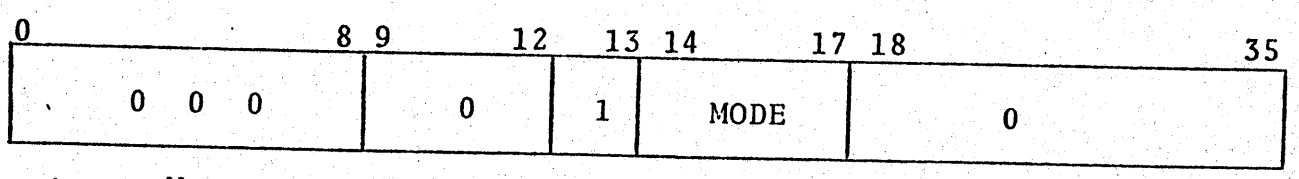
Format: PSHM REG, (MODE)

where: REG is the register
MODE is the new mode (optional)

Mnemonic: NOP (No-OPeration)

Assembler definition: [LI ,@0]

Structure:




Function: No-op; change mode (optional).

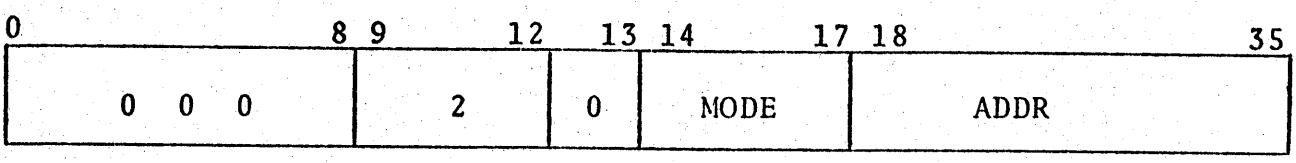
Format: NOP (MODE)

where: MODE is the new mode (optional)

Mnemonic: JMP (JuMP)

Assembler definition: [LI PC, 0]

Structure:




Function: Jump to specified address; change mode (optional).

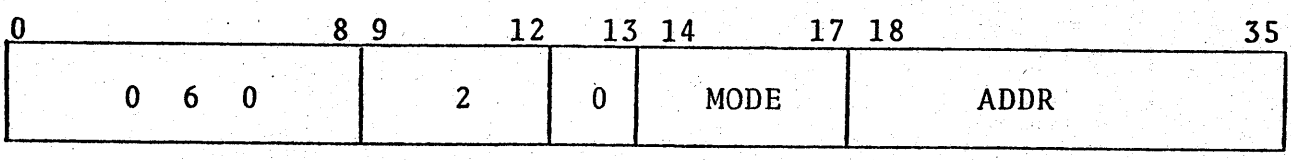
Format: JMP ADDR(MODE)

where: ADDR is the address
MODE is the new mode (optional)

Mnemonic: JMPPSH (JuMP and PuSH old value)

Assembler definition: [LIPSHM PC, 0]

Structure:




Function: Jump to specified address; push old program counter; change mode (optional).

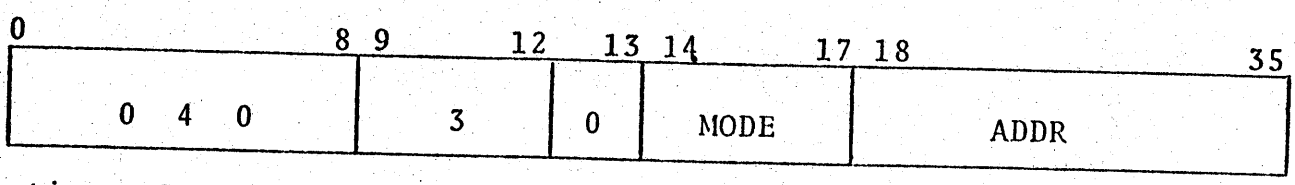
Format: JMPPSH ADDR(MODE)

where: ADDR is the address
MODE is the new mode (optional)

Mnemonic: NWSTK (NeW STacK)

Assembler definition: [LIPSH SP, 0]

Structure:




Function: Create a new stack at specified address; push old stack pointer onto new stack; change mode (optional).

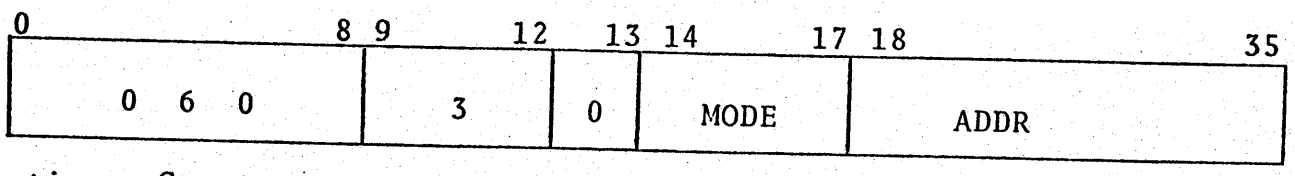
Format: NWSTK ADDR(MODE)

where: ADDR is the address
MODE is the new mode (optional)

Mnemonic: NWSTKM (NeW STacK, Mark)

Assembler definition: [LIPSHM SP, 0]

Structure:




Function: Create a new stack at specified address; push old stack pointer marked onto new stack; change mode (optional).

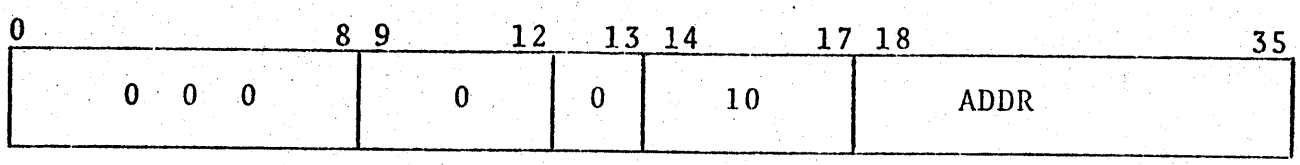
Format: NWSTKM ADDR(MODE)

where: ADDR is the address
MODE is the new mode (optional)

Mnemonic: XQTA (execute at address)

Assembler definition: [LI RAR,0(XQTM)]

Structure:




Function: Execute the instruction at the specified address.

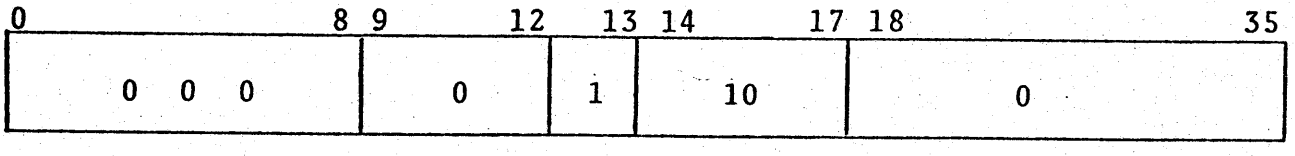
Format: XQTA ADDR

where: ADDR is the address

Mnemonic: XQT (enter XQTM)

Assembler definition: [NOP ,0(XQTM)]

Structure:



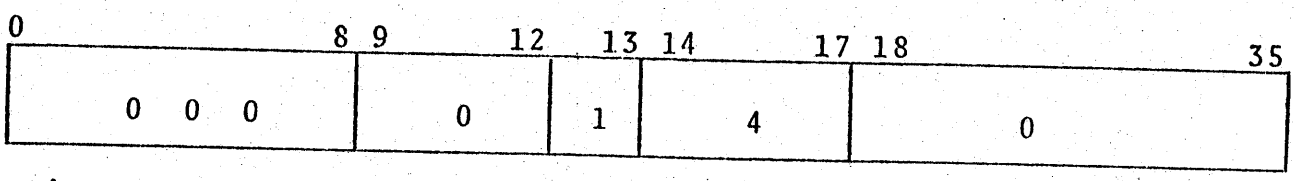

Function: Enter EXECUTE mode.

Format: XQT

Mnemonic: RPT (enter RPTM)

Assembler definition: [NOP ,0(RPTM)]

Structure:



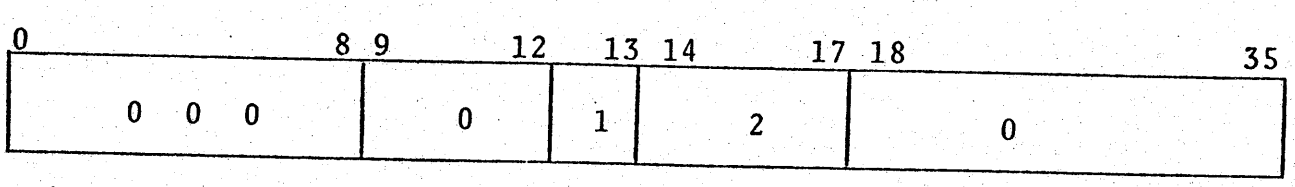

Function: Enter REPEAT mode.

Format: RPT

Mnemonic: PEEL (enter PEELM)

Assembler definition: [NOP ,0(PEELM)]

Structure:



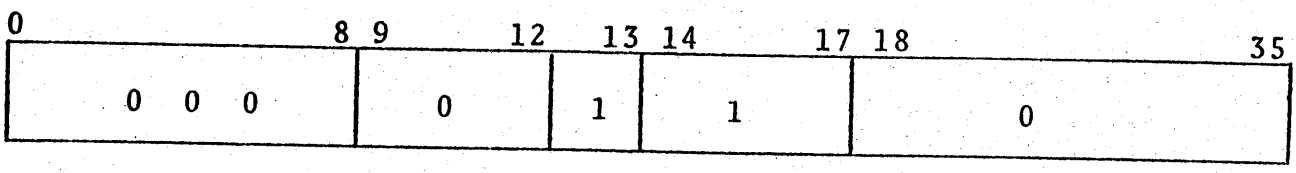

Function: Enter PEEL mode.

Format: PEEL

Mnemonic: PROG (enter PROGM)

Assembler definition: [NOP ,0(PROGM)]

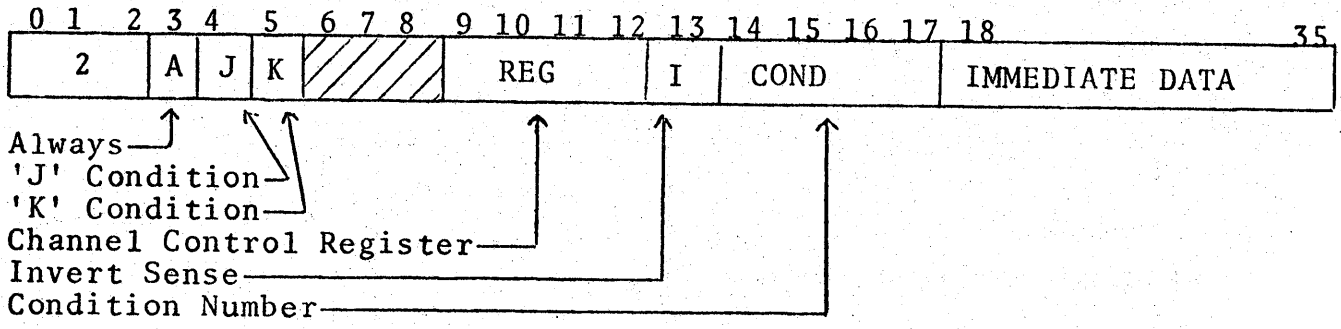
Structure:




Function: Enter PROG mode.

Format: PROG

## 7.3 Group 2




Group 2 instructions perform two operations: the bits A and I and the value of condition COND are examined to determine whether the Immediate Data should be placed in the specified Channel Control register; and the condition COND is modified according to the bits J and K.

The first operation is summarized in the following table:

| A | I | COND | RESULT | COMMENT |
|---|---|---|---|---|
| 1 | 0 | – | load | load always |
| 1 | 1 | – | do not load | load never |
| 0 | 0 | 1 | load | load if condition holds |
| 0 | 0 | 0 | do not load | |
| 0 | 1 | 0 | load | load if condition does not hold |
| 0 | 1 | 1 | do not load | |

The second operation, which is performed only after the first is completed, is summarized in the following table:

| J | K | RESULT |
|---|---|---|
| 0 | 0 | no change |
| 0 | 1 | clear condition |
| 1 | 0 | set condition |
| 1 | 1 | complement condition |

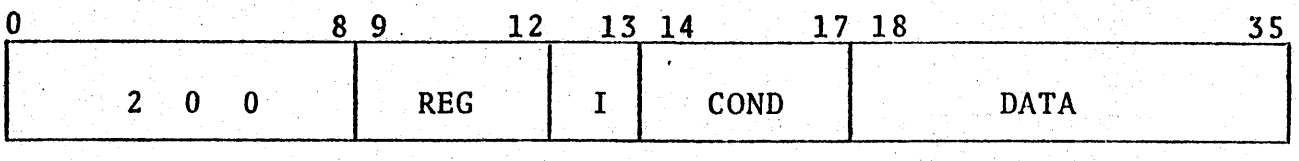
Group 2 instructions are not executed until the pipeline is clear. This is because the condition to be tested may not be determined until the pipeline has finished its processing. Thus LAL differs from LI and should, for example, be used to load the WAR.

### 7.3.1 Instruction in Group 2

Mnemonic: LIF (Load immediate IF)

Assembler definition: [200000000000]

Structure:




Function: Load immediate data into Channel Control register if condition holds and I is zero or if condition does not hold and I is one.

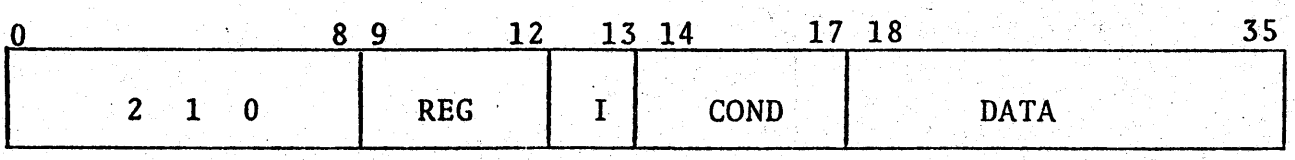
Format: LIF REG, DATA(COND); or
LIF REG, @DATA(COND)

where: REG is the register
DATA is the immediate data
COND is the condition
@ sets the I bit

Mnemonic: LIFCL (Load immediate IF, CLear condition)

Assembler definition: [210000000000]

Structure:




Function: Load register if condition holds and I is zero or if condition does not hold and I is one; then clear condition.

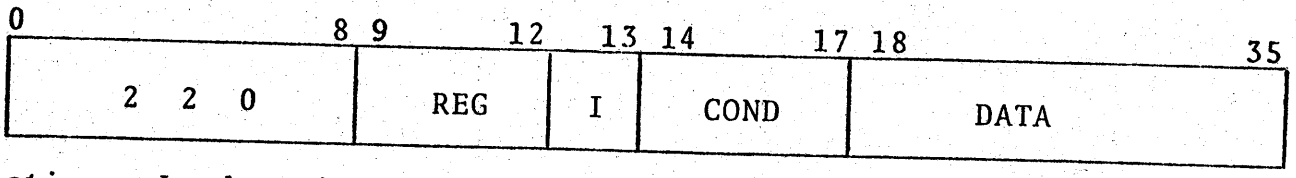
Format: LIFCL REG, DATA(COND); or
LIFCL REG, @DATA(COND)

where: REG is the register
DATA is the immediate data
COND is the condition
@ sets the I bit

Mnemonic: LIFST (Load immediate IF, SeT condition)

Assembler definition: [220000000000]

Structure:




Function: Load register if condition holds and I is zero or if condition does not hold and I is one; then set condition.

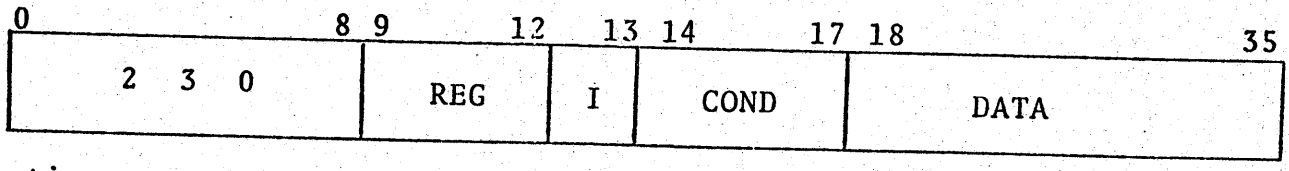
Format: LIFST REG, DATA(COND); or
LIFST REG @DATA(COND)

where: REG is the register
DATA is the immediate data
COND is the condition
@ sets the I bit

Mnemonic: LIFCM (Load immediate IF, CoMplement condition)

Assembler definition: [230000000000]

Structure:




Function: Load register if condition holds and I is zero or if condition does not hold and I is one; then complement condition.

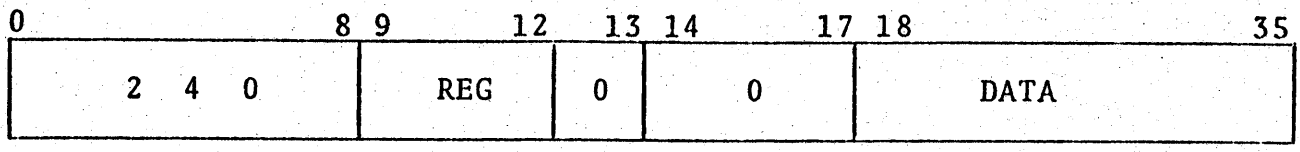
Format: LIFCM REG, DATA(COND); or
LIFCM REG, @DATA(COND)

where: REG is the register
DATA is the immediate data
COND is the condition
@ sets the I bit

Mnemonic: LAL (Load ALways)

Assembler definition: [240000000000]

Structure:




Function: Load immediate data in Channel Control register.

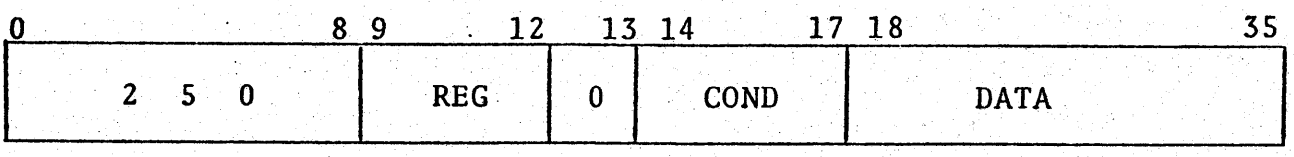
Format: LAL REG, DATA

where: REG is the register
DATA is the immediate data

Mnemonic: LALCL (Load ALways, CLear condition)

Assembler definition: [250000000000]

Structure:




Function: Load register; then clear condition.

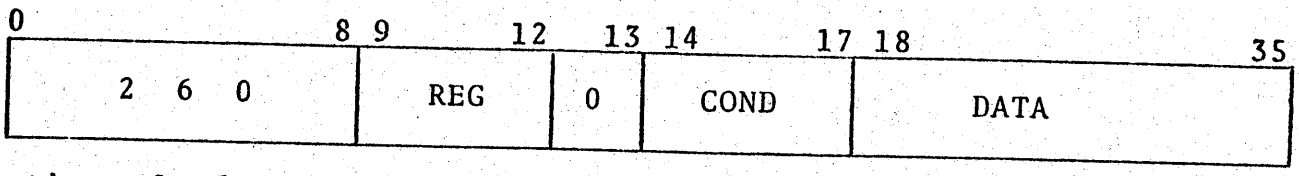
Format: LALCL REG, DATA(COND)

where: REG is the register
DATA is the immediate data
COND is the condition

Mnemonic: LALST (Load ALways, SeT condition)

Assembler definition: [260000000000]

Structure:




Function: Load register; then set condition.

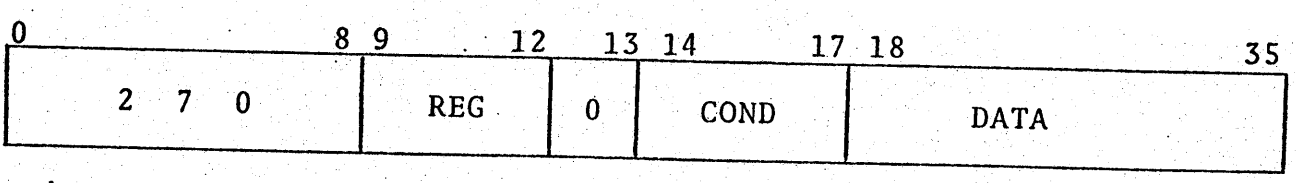
Format: LALST REG, DATA(COND)

where: REG is the register
DATA is the immediate data
COND is the condition

Mnemonic: LALCM (Load ALways, CoMplement condition)

Assembler definition: [270000000000]

Structure:




Function: Load register; then complement condition.

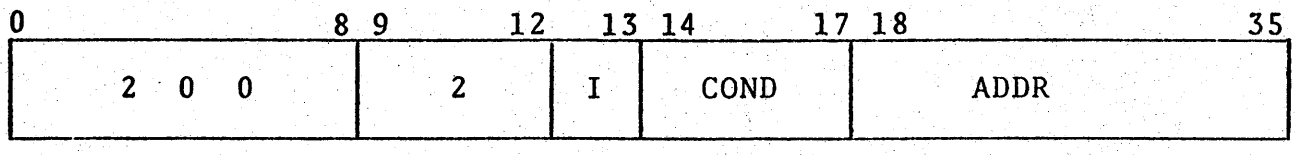
Format: LALCM REG, DATA(COND)

where: REG is the register
DATA is the immediate data
COND is the condition

Mnemonic: JIF (Jump IF)

Assembler definition: [LIF PC,0]

Structure:




Function: Jump to specified address if condition holds and I is zero or if condition does not hold and I is one.

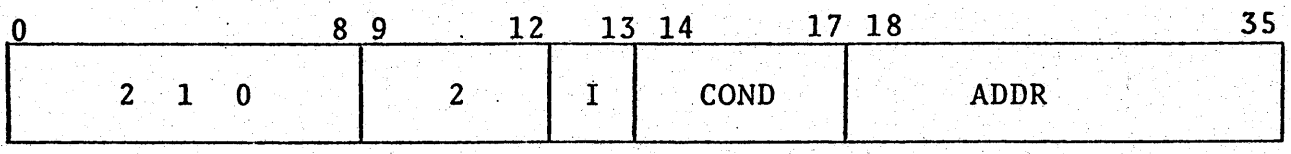
Format: JIF ADDR(COND); or
JIF @ADDR(COND)

where: ADDR is the address
COND is the condition
@ sets the I bit

Mnemonic: JIFCL (Jump IF CLear condition)

Assembler definition: [LIFCL PC,0]

Structure:




Function: Jump to specified address if condition holds and I is zero or if condition does not hold and I is one; then clear condition.

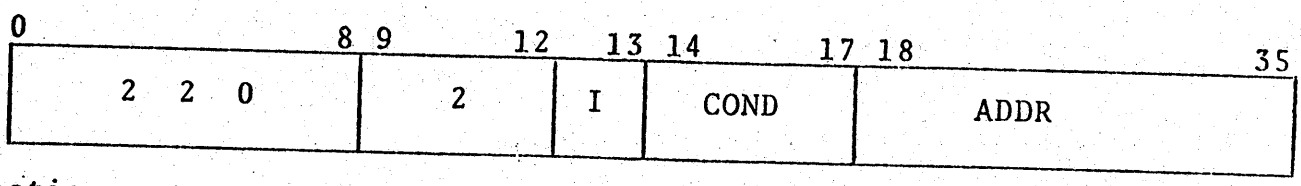
Format: JIFCL ADDR(COND); or
JIFCL @ADDR(COND)

where: ADDR is the address
COND is the condition
@ sets the I bit

Mnemonic: JIFST (Jump IF, SeT condition)

Assembler definition: [LIFST PC,0]

Structure:




Function: Jump to specified address if condition holds and I is zero or if condition does not hold and I is one; then set condition.

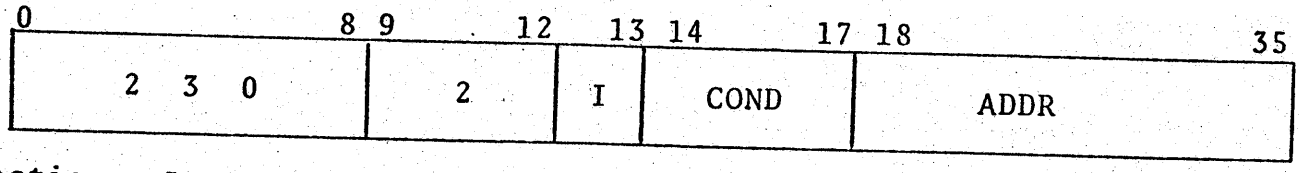
Format: JIFST ADDR(COND); or
JIFST @ADDR(COND)

where: ADDR is the address
COND is the condition
@ sets the I bit

Mnemonic: JIFCM (Jump IF, CoMplement condition)

Assembler definition: [LIFCM PC,0]

Structure:




Function: Jump to specified address if condition holds and I is zero or if condition does not hold and I is one; then complement condition.

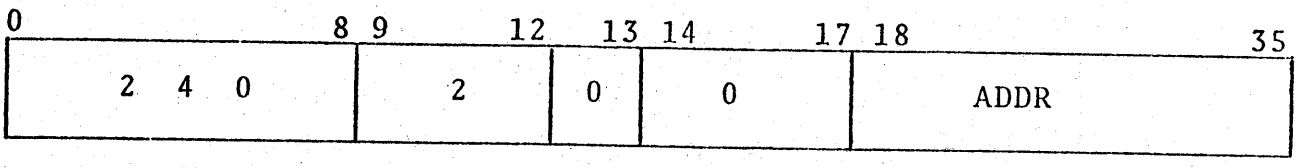
Format: JIFCM ADDR(COND); or
JIFCM @ADDR(COND)

where: ADDR is the address
COND is the condition
@ sets the I bit

Mnemonic: JAL (Jump ALways)

Assembler definition: [LAL PC,0]

Structure:




Function: Jump to specified address.

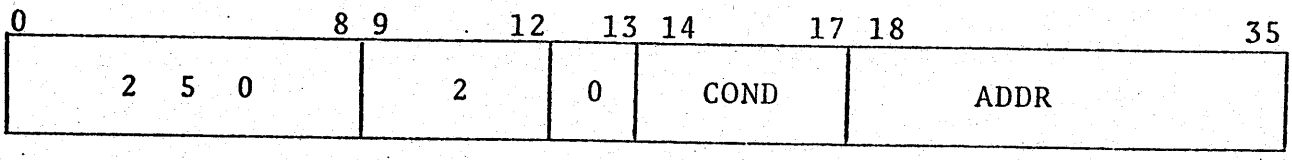
Format: JAL ADDR

where: ADDR is the address

Mnemonic: JALCL (Jump ALways, CLear condition)

Assembler definition: [LALCL PC,0]

Structure:




Function: Jump to specified address; then clear condition.

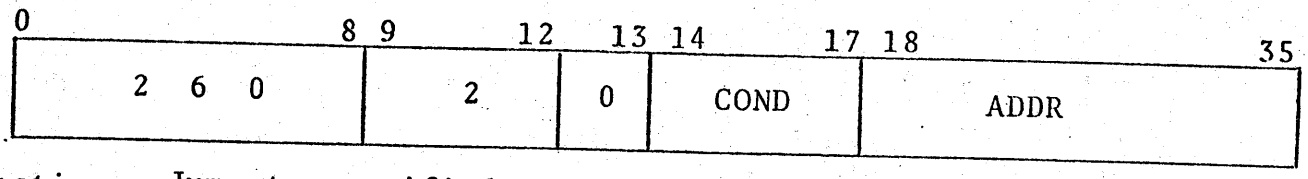
Format: JALCL ADDR(COND)

where: ADDR is the address
COND is the condition

Mnemonic: JALST (Jump ALways, SeT condition)

Assembler definition: [LALST PC,0]

Structure:




Function: Jump to specified address; then set condition.

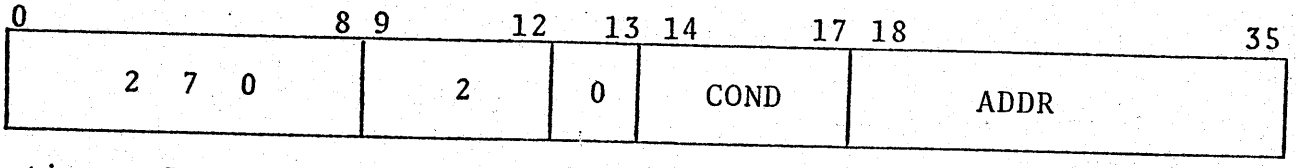
Format: JALST ADDR(COND)

where: ADDR is the address
COND is the condition

Mnemonic: JALCM (Jump ALways, CoMplement condition)

Assembler definition: [LALCM PC,0]

Structure:




Function: Jump to specified address; then complement condition.

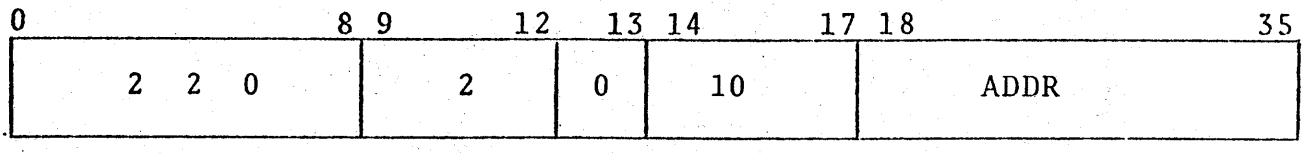
Format: JALCM ADDR(COND)

where: ADDR is the address
COND is the condition

Mnemonic: IJNRCR (Increment and Jump if Negative RCR)

Assembler definition: [JIFST ,0(RCRN)]

Structure:




Function: Jump to specified address if RCR<-1; then increment RCR. (Note: this is equivalent to incrementing the RCR first and then jumping if it is still negative.

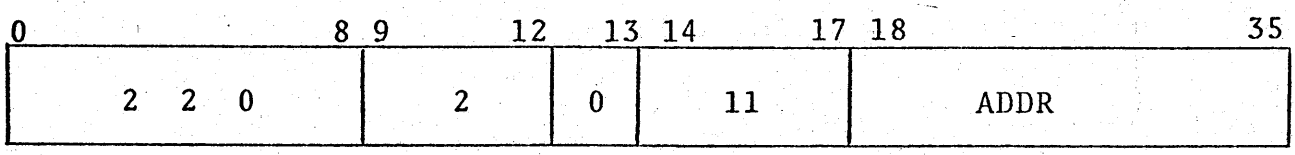
Format: IJNRCR ADDR

where: ADDR is the address

Mnemonic: IJNWCR (Increment and Jump if Negative WCR)

Assembler definition: [JIFST ,0(WCRN)]

Structure:




Function: Jump to specified address if WCR<-1; then increment WCR. (Note: this is equivalent to incrementing the WCR first and jumping if it is still negative.)

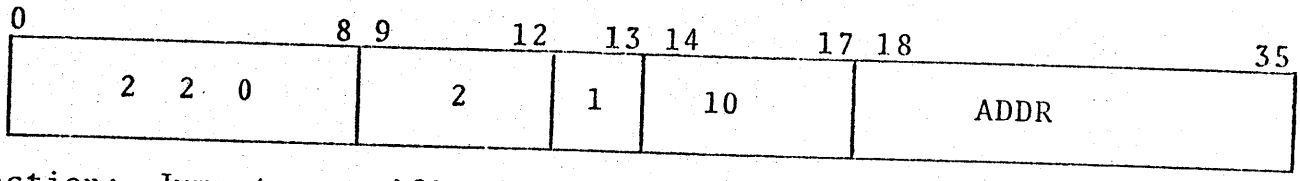
Format: IJNWCR ADDR

where: ADDR is the address

Mnemonic: IJPRCR (Increment and Jump if Positive RCR)

Assembler definition: [IJNRCR @0]

Structure:




Function: Jump to specified address if RCR≥-1; then increment RCR. (Note: this is equivalent to incrementing the RCR and jumping if it is positive or 0.

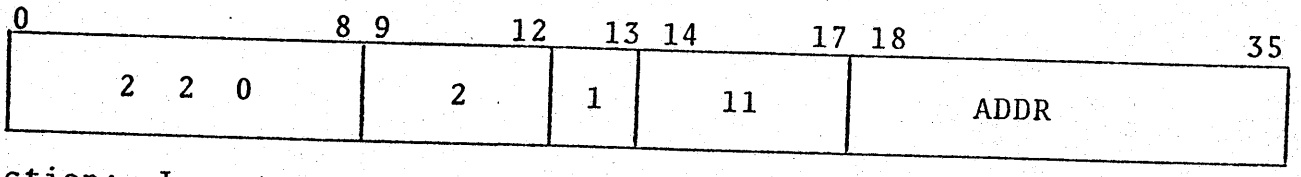
Format: IJPRCR ADDR

where: ADDR is the address

Mnemonic: IJPWCR (Increment and Jump if Positive WCR)

Assembler definition: [IJNWCR @0]

Structure:




Function: Jump to specified address if WCR≥-1; then increment WCR. (Note: this is equivalent to incrementing the WCR and jumping if it is positive or 0.)

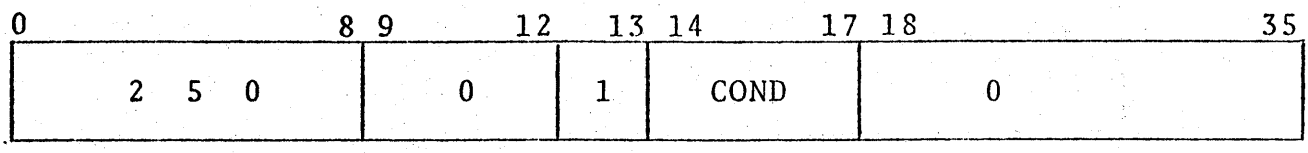
Format: IJPWCR ADDR

where: ADDR is the address

Mnemonic: CL (CLear condition)

Assembler definition: [LALCL @0]

Structure:




Function: Clear specified condition.

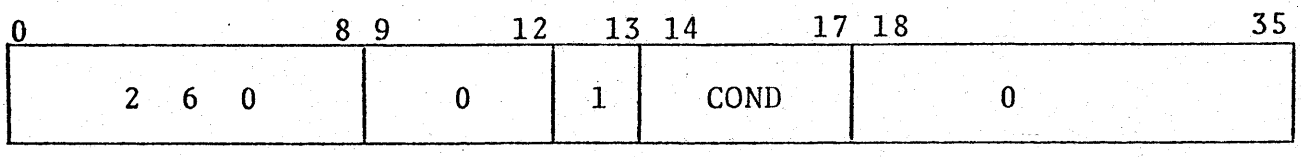
Format: CL (COND)

where: COND is the condition

Mnemonic: ST (SeT condition)

Assembler definition: [LALST @0]

Structure:




Function: Set specified condition.

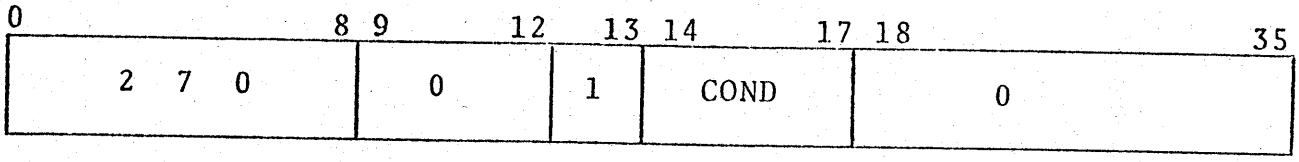
Format: ST (COND)

where: COND is the condition

Mnemonic: CM (CoMplement condition)

Assembler definition: [LALCM @0]

Structure:




Function: Complement specified condition.

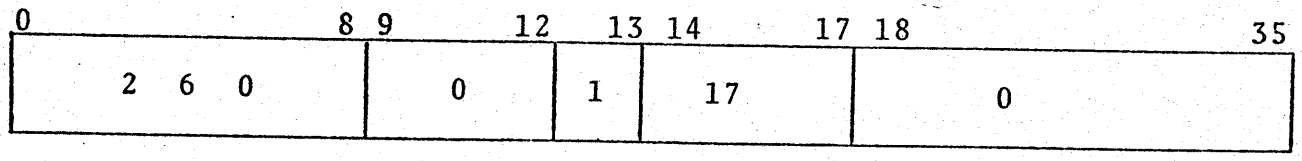
Format: CM (COND)

where: COND is the condition

Mnemonic: STOP (STOP)

Assembler definition: [ST (STOPF)]

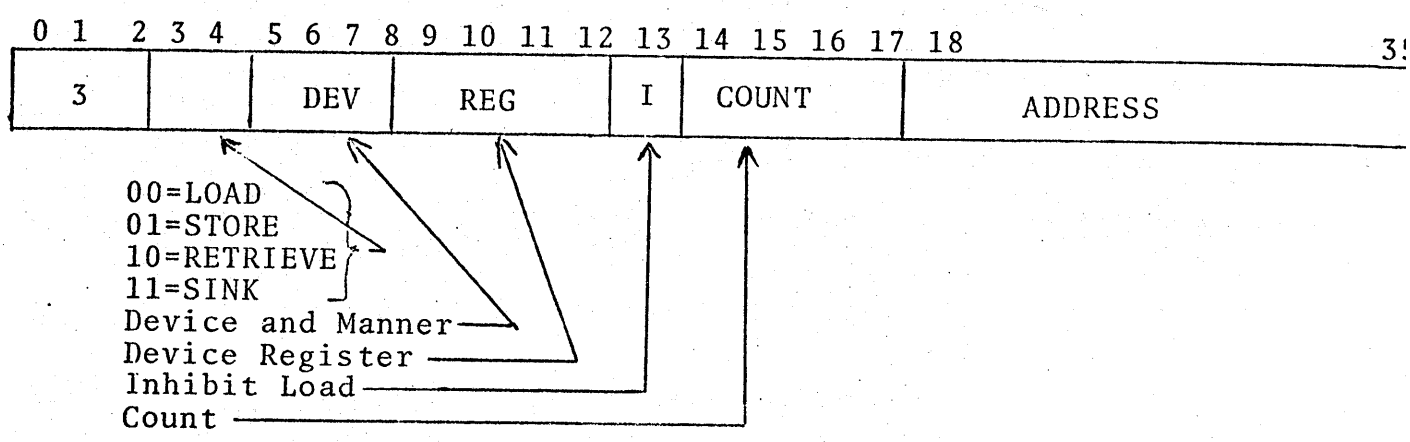
Structure:




Function: Stop the LDS-1.

Format: STOP

## 7.4 Group 3




This instruction group is used to transmit external device registers in and out of core. It is also used for miscellaneous Matrix Multiplier operations. These instructions are not executed until the pipeline is settled. Four types of transmission are provided, as determined by bits 3 and 4:

LOAD loads registers from the core location held in the RAR.

STORE stores registers in the core location held in the RAR.

RETRIEVE loads registers from the core location held in the DSP (i. e. it retrieves from the data sink).

SINK stores registers in the core location held in the DSP (i. e. it pushes onto the data sink).

The first thing these instruction do, if the I bit is zero, is to load the ADDRESS field into the RAR (for load and store) or the DSP (for sink and retrieve). No loading is done if the I bit is one. The I bit is generally one for sink and retrieve instructions (and is in fact defined as one for the mnemonics) so that the DSP is not disturbed from keeping track of the current location in the data sink.

The registers are then transmitted. The first register to be transmitted is specified by the REG field, and the number of registers to be transmitted is specified by the COUNT field, where a COUNT of zero means sixteen. After each register is transmitted, the following actions are taken:

1. The REG field is incremented (load, store, and sink) or decremented (retrieve).

2. The RAR is incremented (load and store), the DSP is incremented (sink), or the DSP is decremented (retrieve).

3. The COUNT field is decremented and tested for zero; if non-zero, another register is transmitted.

The external device in which the registers are located is specified by the DEV field, as well as the manner of transmission. The store and sink variants have only one manner-absolute. Load and retrieve have four manners, however. Listed below are the meanings of these manners:

Clipping Divider Registers

| | |
|---|---|
| Absolute: | Core values are copied into registers; |
| Relative: | Core values are added to the contents of the SAVE register and placed in the registers; |
| Size Absolute: | Core values are copied into the second two components of the registers, and the negative of the core values are copied into the first two components of the registers (2D only, registers 14-17 only); |
| Size Relative: | Core values are added to the contents of the SAVE register and placed in the second half of the registers; core values are subtracted from the contents of the SAVE register and placed in the first half of the registers (2D only, registers 14-17 only). |

Matrix Multiplier Registers

| | |
|---|---|
| Absolute: | Core values are copied into registers. |
| Relative: | Core values are added to the contents of the respective register and the sum is used to fill the register. |
| Product: | Core values are multiplied by contents of a matrix (not A) and the result is placed in matrix A. |

Size Absolute and Size Relative are not used for the Matrix Multiplier.

A summary of the devices and manners representable by the DEV field of load instructions follows:

| DEV | Device and Manner | Abbreviation |
|---|---|---|
| 0 | Clipper-Absolute | CLA |
| 1 | Clipper-Relative | CLR |
| 2 | Clipper-Size Absolute | CLSA |
| 3 | Clipper-Size Relative | CLSR |
| 4 | Matrix Multiplier-Absolute | MM |
| 5 | Matrix Multiplier-Relative | MMR |

| DEV | Device and Manner | Abbreviation |
|---|---|---|
| 6 | Matrix Multiplier-Product | MMP |
| 7 | Matrix Multiplier Directive-Absolute | MDIR |
| 14 | Character Bubble-Absolute | CB |
| 16 | Switches, Buttons, Knobs, Lights-Absolute | SBKL |
| 17 | Additional SBKL Registers-Absolute | |

In store instructions, only DEV fields of 0, 4, 7, 14, 16, and 17 are legal. However, a DEV field of 5 in a store instruction has a special meaning for the Matrix Multiplier: "normalize." Also a DEV field of 6 means "push Matrix Multiplier." See section 8.7

In sink instructions, the same DEV fields are legal: 0, 4, 7, 14, 16, and 17. Here a DEV field of 5 has the meaning sink and slide.

All the DEV fields legal for load are also legal for retrieve instructions; but 5 means retrieve and slide and 6 means "pop Matrix Multiplier" as described in section 8.7 as well.

### 7.4.1 2D and 3D Register Transmission

In 2D, two-component registers are generally loaded with a single memory word whereas in 3D the registers loaded are four-component registers, and two data words are used. Each of the four-component registers is made up of two separate two-component registers.

2D loads for the Matrix Multiplier address rows of the matrices just as 3D loads, the difference being that in 2D only the first two elements of the row are loaded whereas in 3D the whole row is loaded.

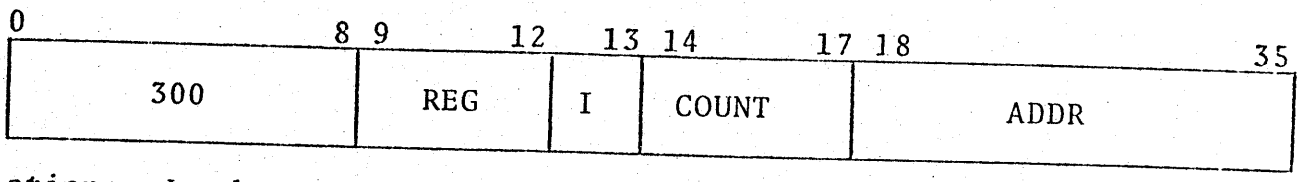
For the Clipping Divider, however, 2D loads use different addresses than 3D loads (see Figure 4.1). Registers 0-$13_8$ are two-component registers, and registers 14-$17_8$ the corresponding four-component registers. A four-component register may, however, be specified in 2D. If the manner is size absolute or size relative, the load works as described previously. If the manner is simply absolute or relative, the contents of the first two components of the SAVE register are copied into the first two components of the register, while the data word is copied into the remaining two components of the register (absolute) or first added to the contents of SAVE and then copied into the remaining two components (relative). 2D, four-component loads are used preliminary to boxing (see section 8.5).

## 7.4.2 Instructions in Group 3

Mnemonic: LOCLA (LOad CLipping divider Absolute)

Assembler definition: [300000000000]

Structure:




Function: Load COUNT Clipping Divider registers directly from core beginning at ADDR (if I=0) or at the current value of the RAR (if I=1).

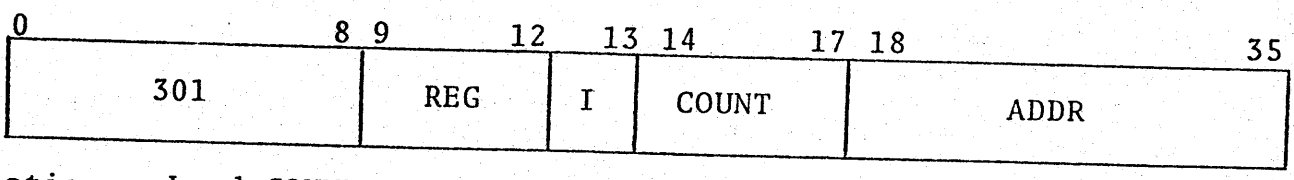
Format:
```
LOCLA  REG,ADDR(COUNT)  ;  or
LOCLA  REG,@(COUNT)
```

where: REG is the first register to be loaded.
ADDR is the address.
COUNT is the number of registers to be loaded.
@ sets the I bit.

Mnemonic: LOCLR (LOad CLipping divider Relative)

Assembler definition: [301000000000]

Structure:




Function: Load COUNT Clipper registers with the sum of the SAVE register contents plus values taken from core beginning at ADDR (if I=0) or the RAR (if I=1).

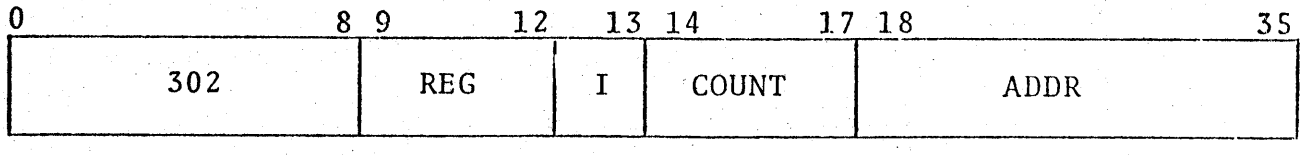
Format:
```
LOCLR  REG,ADDR(COUNT)  ;  or
LOCLR  REG,@(COUNT)
```

where: REG is the first register.
ADDR is the address.
COUNT is the number of registers.
@ sets the I bit.

Mnemonic: LOCLSA (LOad CLipping divider Size Absolute)

Assembler definition: [302000000000]

Structure:




Function: Load COUNT four-component Clipper registers using values beginning at ADDR (if I=0) or the RAR (if I=1). Place the values in the second half of the register and the negative of those values in the first half.

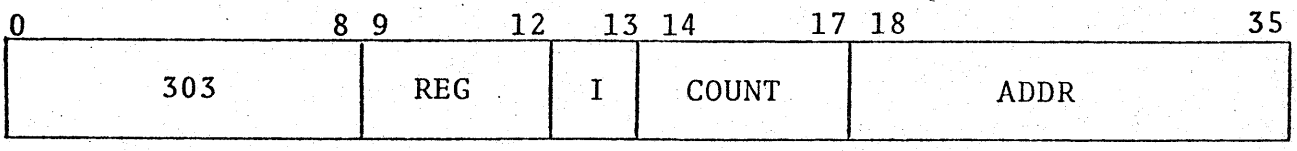
Format:
```
LOCLSA  REG,ADDR(COUNT)  ; or
LOCLSA  REG,@(COUNT)
```

where: REG is the first register.
ADDR is the address.
COUNT is the number of registers.
@ sets the I bit

Mnemonic: LOCLSR (LOad CLipping divider Size Relative)

Assembler definition: [303000000000]

Structure:




Function: Load COUNT four-component Clipper registers using values beginning at ADDR (if I=0) or the RAR (if I=1). Obtain the second half of the registers by adding the values found to the SAVE register; obtain the first half by subtracting those values from the SAVE register.

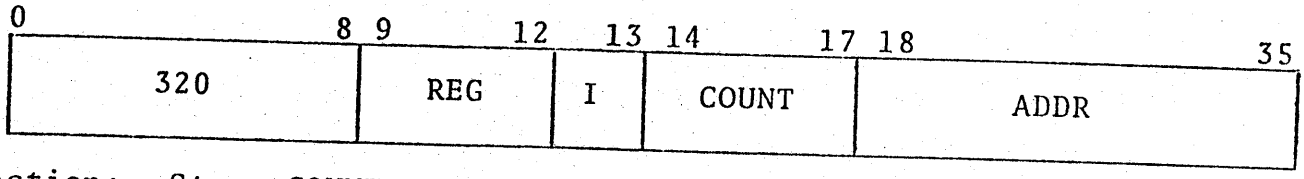
Format:
```
LOCLSR  REG,ADDR(COUNT)  ; or
LOCLSR  REG,@(COUNT)
```

where: REG is the first register (REG = 14, 15, 16, or 17)
ADDR is the address.
COUNT is the number of registers.
@ sets the I bit.

Mnemonic: STCL (STore CLipping divider)

Assembler definition: [320000000000]

Structure:




Function: Store COUNT Clipper registers into the core locations addressed by ADDR (if I=0) or the RAR (if I=1).

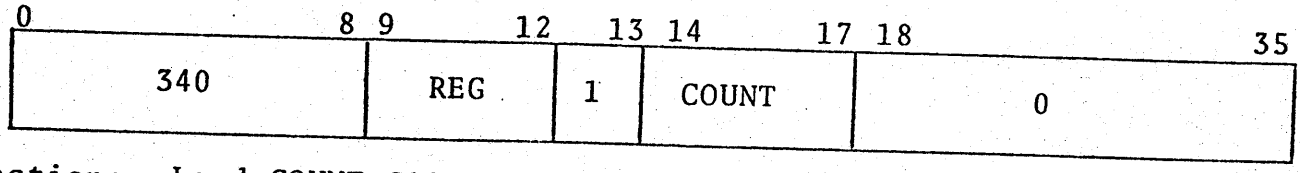
Format:
```
STCL  REG,ADDR(COUNT)  ; or
STCL  REG,@(COUNT)
```

where: REG is the first register to be stored.
ADDR is the address.
COUNT is the number of registers.
@ sets the I bit.

Mnemonic: RTCLA (ReTrieve CLipping divider Absolute)

Assembler definition: [340 , @0]

Structure:




Function: Load COUNT Clipper registers, beginning at REG and counting backwards, from the current value of the DSP (which is decremented after each register is retrieved).

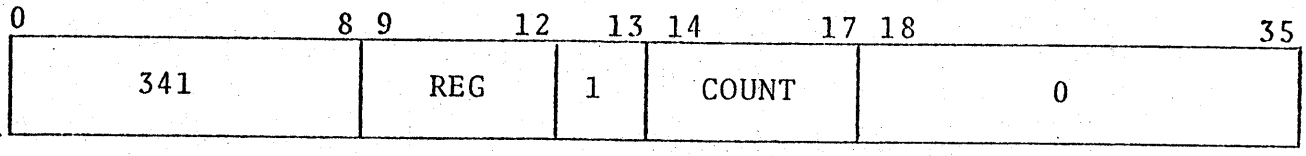
Format:
```
RTCLA  REG,(COUNT)
```

where: REG is the first register to be retrieved.
COUNT is the number of registers.

Mnemonic: RTCLR (ReTrieve CLipping divider Relative)

Assembler definition: [341 ,@0]

Structure:




Function: Load COUNT Clipper registers, counting backward from REG, by adding the contents of the SAVE register to the values addressed by the DSP (which is decremented after each use).

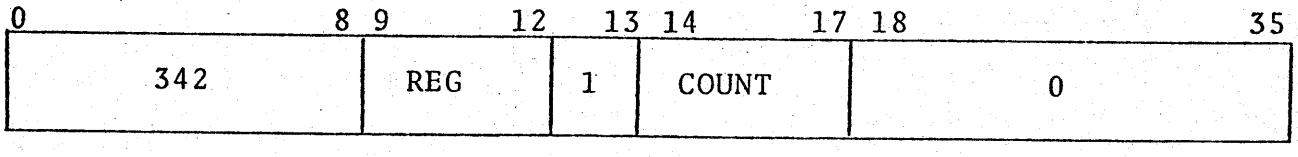
Format: RTCLR REG,(COUNT)

where: REG is the first register.
COUNT is the number of registers.

Mnemonic: RTCLSA (ReTrieve CLipping divider Size Absolute)

Assembler definition: [342 ,@0]

Structure:




Function: Load COUNT four-component Clipper registers counting backwards from REG, using values addressed by the decrementing DSP. Load the values into the second half of the registers and the negative of those values into the first half.

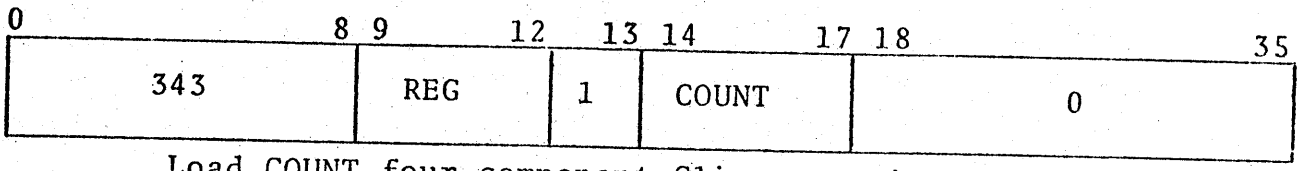
Format: RTCLSA REG,(COUNT)

where: REG is the first register (REG = 14, 15, 16, or 17)
COUNT is the number of registers.

Mnemonic: RTCLSR (ReTrieve CLipping divider Size Relative)

Assembler definition: [343 ,@0]

Structure:




Function: Load COUNT four-component Clipper registers, counting backward from REG, using values addressed by the decrementing DSP. Obtain the second half of the registers by adding the values addressed to the contents of the SAVE register; obtain the first half by subtracting those values from the contents of the SAVE register.

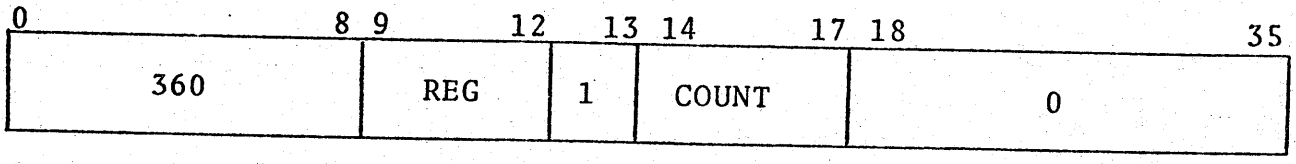
Format: RTCLSR REG,(COUNT)

where: REG is the first register (REG = 14, 15, 16, or 17)
COUNT is the number of registers.

Mnemonic: SKCL (SinK CLipping divider)

Assembler definition: [360 ,@0]

Structure:




Function: Store COUNT Clipper registers into core, beginning at the location addressed by the DSP.

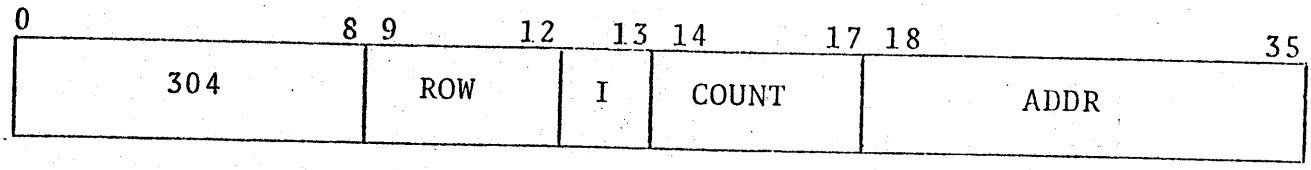
Format: SKCL REG,(COUNT)

where: REG is the first register.
COUNT is the number of registers.

Mnemonic: LOMM (LOad Matrix Multiplier, absolute implied)

Assembler definition: [304000000000]

Structure:




Function: Load COUNT Matrix Multiplier rows, beginning at ROW, from core locations beginning at ADDR (if I=0) or the RAR (if I=1).

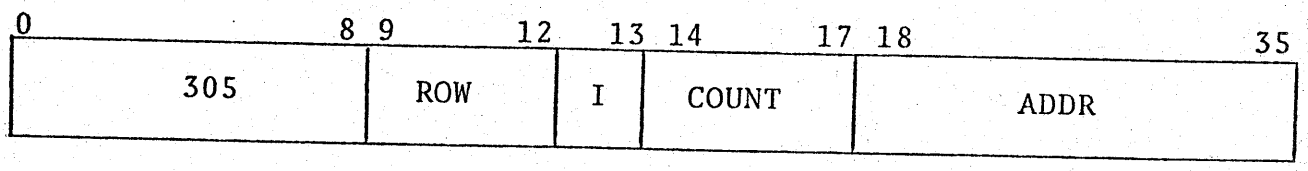
Format:
```
LOMM  ROW,ADDR(COUNT)  ; or
LOMM  ROW,@(COUNT)
```

where: ROW is the first row to be loaded.
ADDR is the address.
COUNT is the number of rows.
@ sets the I bit.

Mnemonic: LOMMR (LOad Matrix Multiplier Relative)

Assembler definition: [305000000000]

Structure:




Function: Add the values addressed by ADDR (if I=0) or the RAR (if I=1) element-by-element to each of COUNT Matrix Multiplier rows, beginning at ROW.

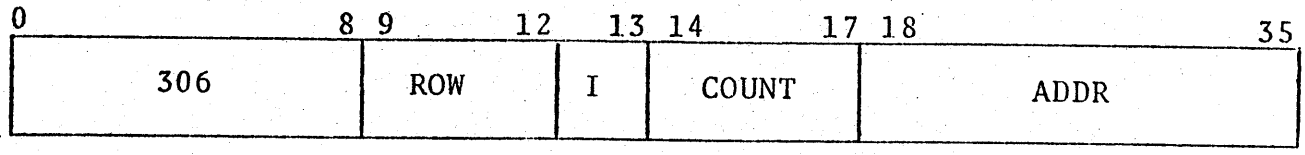
Format:
```
LOMMR  ROW,ADDR(COUNT)  ; or
LOMMR  ROW,@(COUNT)
```

where: ROW is the first row to be modified.
ADDR is the address.
COUNT is the number of rows.
@ sets the I bit.

Mnemonic: LOMMP (LOad Matrix Multiplier Product)

Assembler definition: [306000000000]

Structure:




Function: Multiply the values addressed by ADDR (if I=0) or the RAR (if I=1) by the entire matrix in which ROW is located, and place the result in the corresponding row of matrix A. If COUNT is greater than one, do the same for ROW+1, etc.

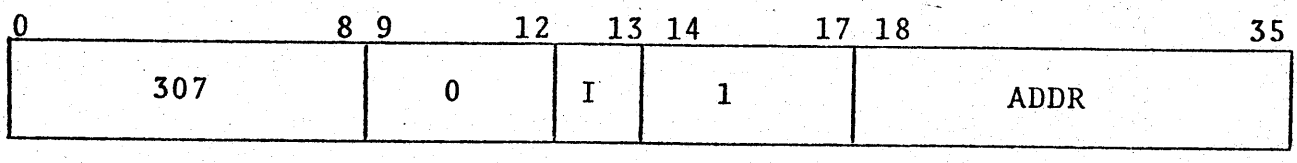
Format:
```
LOMMP  ROW,ADDR(COUNT)  ; or
LOMMP  ROW,@(COUNT)
```

where: ROW specifies the matrix (high two bits) and row for result (low two bits).
ADDR is the address.
COUNT is the number of vector products. COUNT should be $\leq 4$.
@ sets the I bit.

Mnemonic: LOMDIR (LOad Matrix Multiplier DIRective register)

Assembler definition: [307 ,0(1)]

Structure:




Function: Load the Matrix Multiplier Directive register with the word addressed by ADDR (if I=0) or the RAR (if I=1).

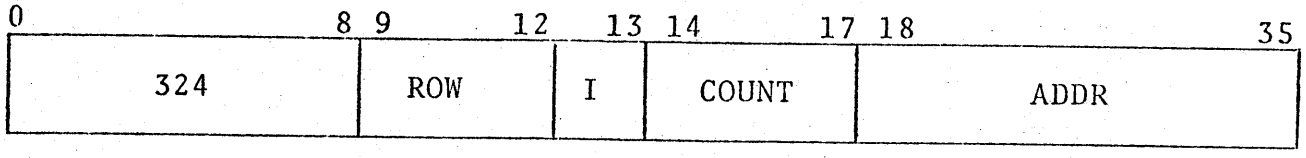
Format:
```
LOMDIR  ADDR  ; or
LOMDIR  @
```

where: ADDR is the address.
@ sets the I bit.

Mnemonic: STMM (STore Matrix Multiplier, absolute implied)

Assembler definition: [324000000000]

Structure:




Function: Store COUNT Matrix Multiplier rows beginning at ROW into core beginning at ADDR (if I=0) or the RAR (if I=1).

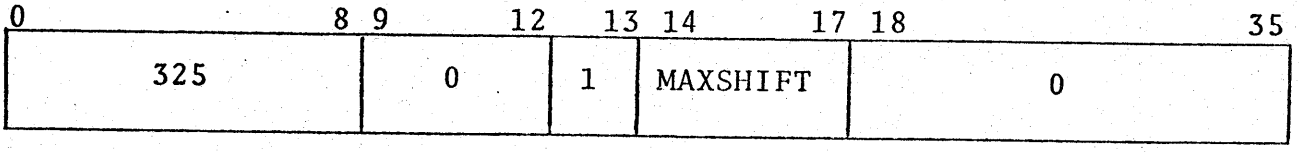
Format:
```
STMM  ROW,ADDR(COUNT)  ;  or
STMM  ROW,@(COUNT)
```

where: ROW is the first row to be stored.
ADDR is the address.
COUNT is the number of rows.

Mnemonic: NOMM (NOrmalize Matrix Multiplier)

Assembler definition: [325 ,@0]

Structure:




Function: Normalize matrix A by shifting each of its elements left one bit until either MAXSHIFT shifts have taken place or some element of A is between one-half and one in magnitude.

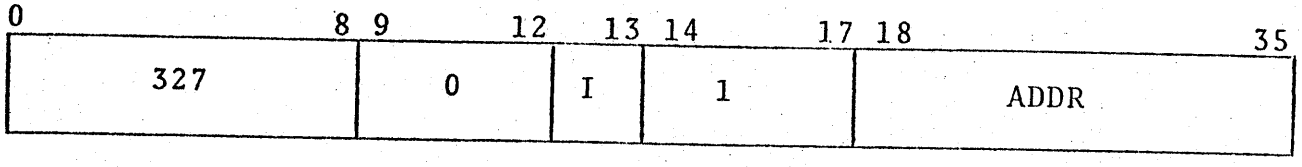
Format: NOMM (MAXSHIFT)

where: MAXSHIFT is the maximum number of shifts.

Mnemonic: STMDIR (STore Matrix multiplier DIRective register)

Assembler definition: [327 ,0(1)]

Structure:




Function: Store the Matrix Multiplier Directive register into the core location addressed by ADDR (if I=0) or the RAR (if I=1).

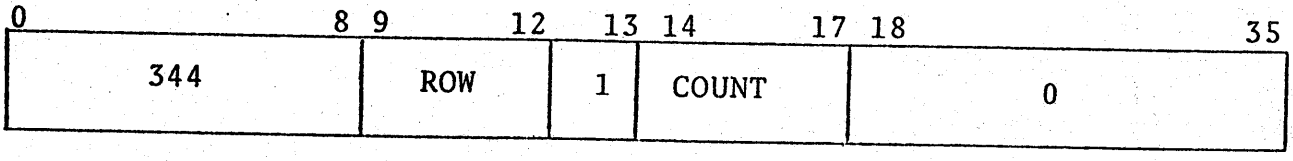
Format: STMDIR ADDR ; or
STMDIR @

where: ADDR is the address.
@ sets the I bit.

Mnemonic: RTMM (ReTrieve Matrix Multiplier, absolute implied)

Assembler definition: [344 ,@0]

Structure:




Function: Load COUNT Matrix Multiplier rows beginning at ROW and counting backwards, with values addressed by the DSP, which is decremented after each use.

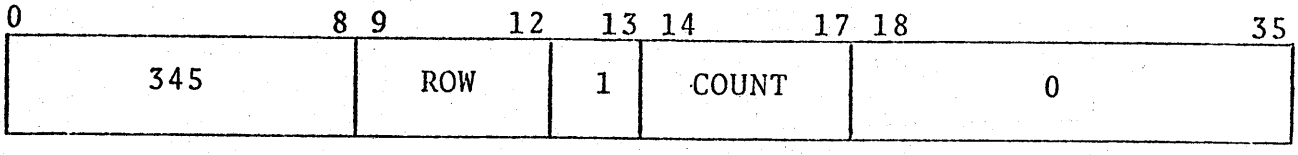
Format: RTMM ROW,(COUNT)

where: ROW is the first row to be retrieved.
COUNT is the number of rows.

Mnemonic: RTMMS (ReTrieve Matrix Multiplier and Slide)

Assembler definition: [345 ,@0]

Structure:




Function: Load COUNT Matrix Multiplier rows, counting backward from ROW, from the decrementing DSP; but before each load, copy the old value of the row into the corresponding row of matrix A.

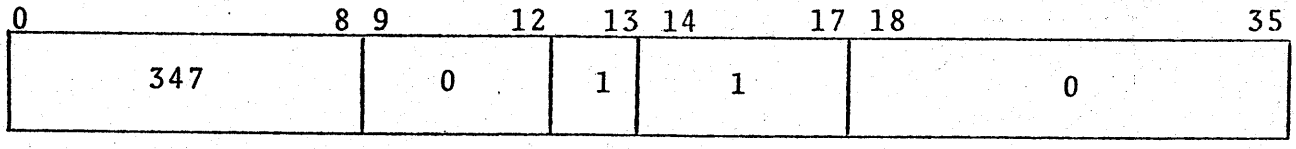
Format: RTMMS ROW,(COUNT)

where: ROW is the first row to be retrieved.
COUNT is the number of rows.

Mnemonic: RTMDIR (ReTrieve Matrix multiplier DIRective register)

Assembler definition: [347 ,@0(1)]

Structure:



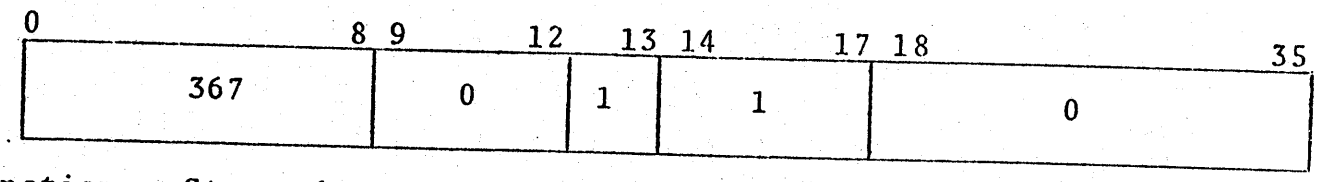

Function: Retrieve the value of the Matrix Multiplier Directive register from the DSP.

Format: RTMDIR

Mnemonic: SKMDIR (SinK Matrix multiplier DIRective register)

Assembler definition: [367 ,@0(1)]

Structure:



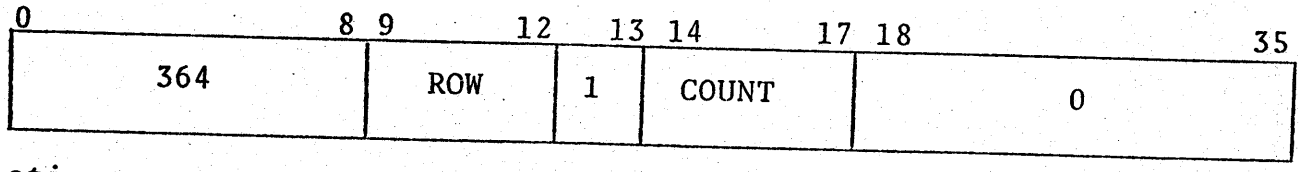

Function: Store the Matrix Multiplier Directive register at the location addressed by the DSP.

Format: SKMDIR

Mnemonic: SKMM (SinK Matrix Multiplier)

Assembler definition: [364 ,@0]

Structure:




Function: Store COUNT Matrix Multiplier registers beginning at ROW into core beginning at the DSP.

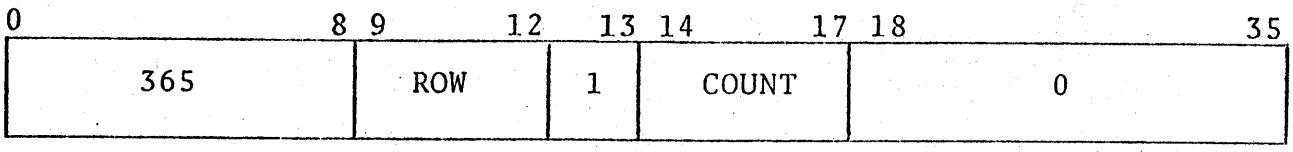
Format: SKMM ROW,(COUNT)

where: ROW is the first row to be sinked.
COUNT is the number of rows.

Mnemonic: SKMMS (SinK Matrix Multiplier and Slide)

Assembler definition: [365 ,@0]

Structure:




Function: Store COUNT Matrix Multiplier registers beginning at ROW into core beginning at the DSP, and after each store replace the sinked row with the corresponding row of matrix A.

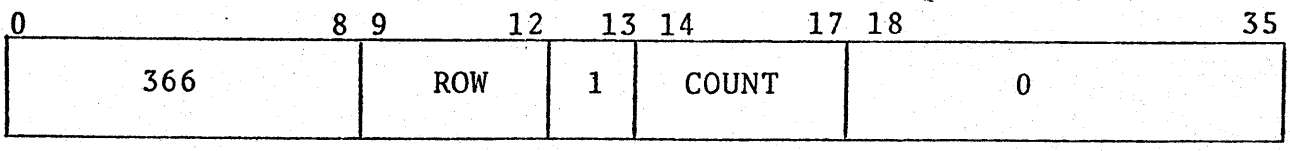
Format: SKMMS ROW,(COUNT)

where: ROW is the first row to be sinked and replaced.
COUNT is the number of rows.

Mnemonic: PUSHMM (PUSH Matrix Multiplier)

Assembler definition: [326 ,@0]

Structure:




Function: Load each of COUNT Matrix Multiplier rows starting with ROW with its corresponding row from matrix A.

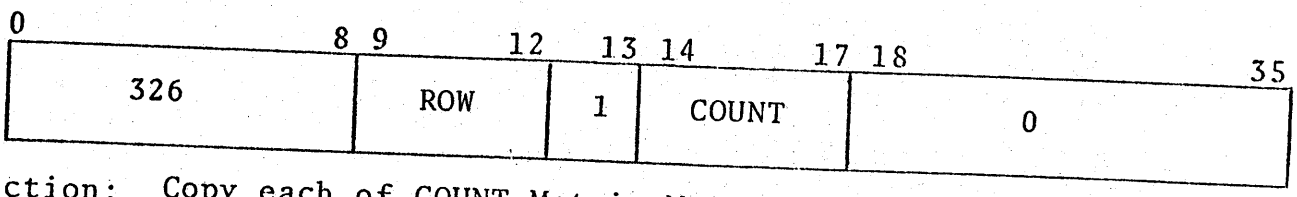
Format: PUSHMM ROW,(COUNT)

where: ROW is the first row to be loaded.
COUNT is the number of rows.

Mnemonic: POPMM (POP Matrix Multiplier)

Assembler definition: [366 ,@0]

Structure:




Function: Copy each of COUNT Matrix Multiplier rows starting with ROW into its corresponding row in matrix A.

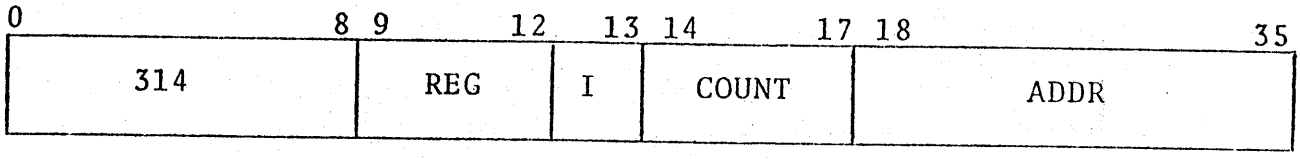
Format: POPMM ROW,(COUNT)

where: ROW is the first row to be copied.
COUNT is the number of rows.

Mnemonic: LOCB (LOad Character Bubble absolute)

Assembler definition: [314000000000]

Structure:




Function: Load COUNT registers of the Character Bubble with data addressed by ADDR (if I=0) or the RAR (if I=1).

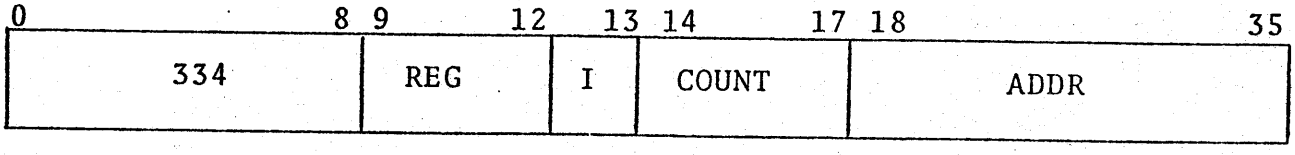
Format:
```
LOCB  REG,ADDR(COUNT)  ; or
LOCB  REG,@(COUNT)
```

where: REG is the first register to be loaded.
ADDR is the address.
COUNT is the number of registers.
@ sets the I bit.

Mnemonic: STCB (STore Character Bubble)

Assembler definition: [334000000000]

Structure:




Function: Store COUNT Character Bubble registers into core locations beginning at ADDR (if I=0) or the RAR (if I=1).

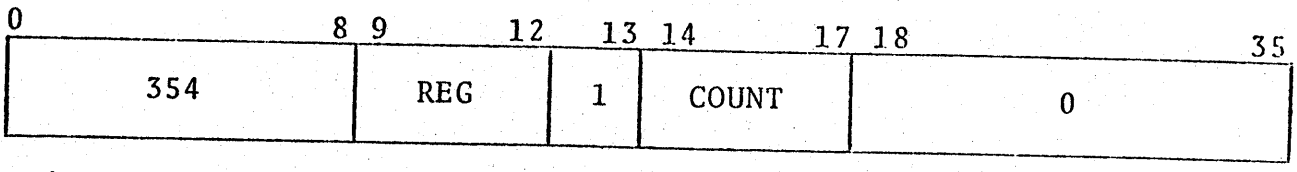
Format:
```
STCB  REG,ADDR(COUNT)  ; or
STCB  REG,@(COUNT)
```

where: REG is the first register to be stored.
ADDR is the address.
COUNT is the number of registers.
@ sets the I bit.

Mnemonic: RTCB (ReTrieve Character Bubble absolute)

Assembler definition: [354 ,@0]

Structure:




Function: Load COUNT Character Bubble registers, starting at REG and counting backwards, from the location held in the DSP (which is decremented after each retrieve).

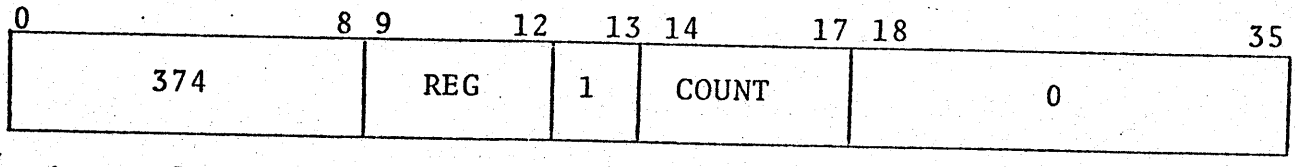
Format: RTCB REG,(COUNT)

where: REG is the first register to be retrieved.
COUNT is the number of registers.

Mnemonic: SKCB (SinK Character Bubble)

Assembler definition: [374 ,@0]

Structure:




Function: Store COUNT Character Bubble registers starting with REG into core beginning at the DSP.

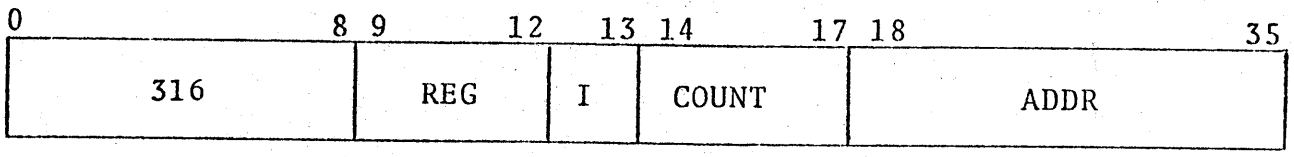
Format: SKCB REG,(COUNT)

where: REG is the first register to be sinked.
COUNT is the number of registers.

Mnemonic: LOSBKL (LOad SBKL)

Assembler definition: [316000000000]

Structure:




Function: Load COUNT SBKL registers beginning at REG with values addressed by ADDR (if I=0) or the RAR (if I=1).

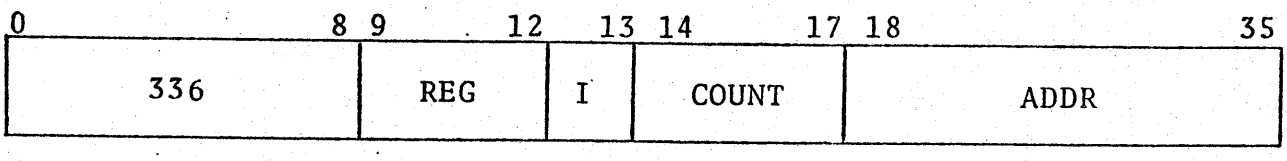
Format:
```
LOSBKL  REG,ADDR(COUNT)  ; or
LOSBKL  REG,@(COUNT)
```

where: REG is the first register to be loaded.
ADDR is the address.
COUNT is the number of registers.
@ sets the I bit.

Mnemonic: STSBKL (STore SBKL)

Assembler definition: [336000000000]

Structure:




Function: Store COUNT SBKL registers starting with REG into core starting at ADDR (if I=0) or the RAR (if I=1).

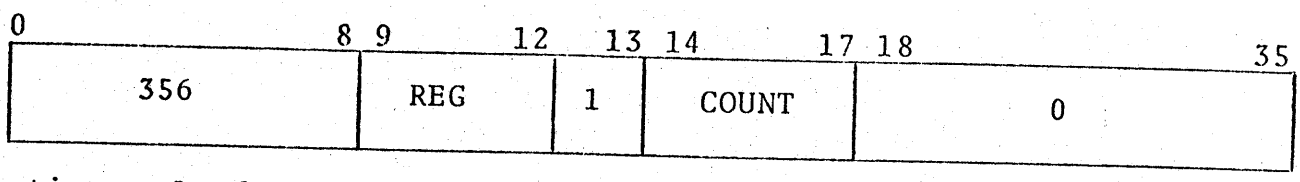
Format:
```
STSBKL  REG,ADDR(COUNT)  ; or
STSBKL  REG,@(COUNT)
```

where: REG is the first register to be stored.
ADDR is the address.
COUNT is the number of registers.
@ sets the I bit.

Mnemonic: RTSBKL (ReTrieve SBKL)

Assembler definition: [356 ,@0]

Structure:




Function: Load COUNT SBKL registers beginning at REG and counting backwards, from the location held in the DSP, which is decremented after each use.

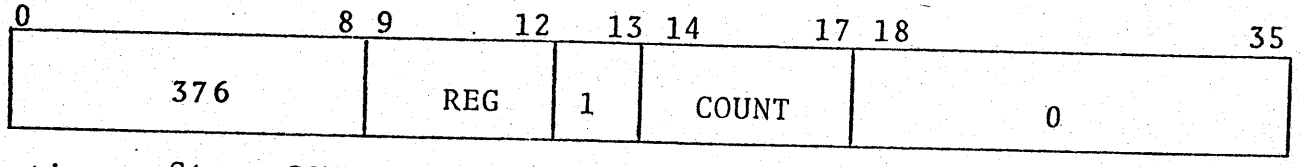
Format: RTSBKL REG,(COUNT)

where: REG is the first register to be retrieved,
COUNT is the number of registers.

Mnemonic: SKSBKL (SinK SBKL)

Assembler definition: [376 ,@0]

Structure:




Function: Store COUNT SBKL registers starting with REG into core beginning at the DSP.

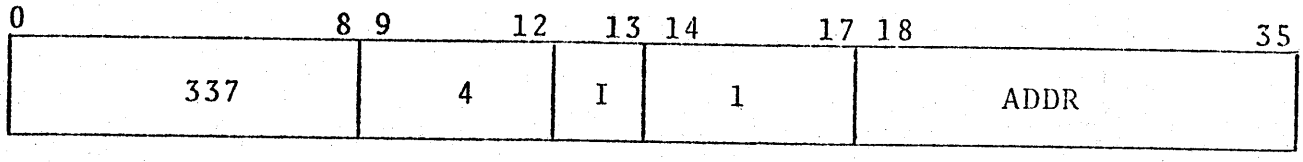
Format: SKSBKL REG,(COUNT)

where: REG is the first register to be sinked,
COUNT is the number of registers.

Mnemonic: STSWCH (STore SWitCHes)

Assembler definition: [STSBKL 4,0(1)]

Structure:




Function: Store the Swtich settings at ADDR (if I=0) or the RAR (if I=1).

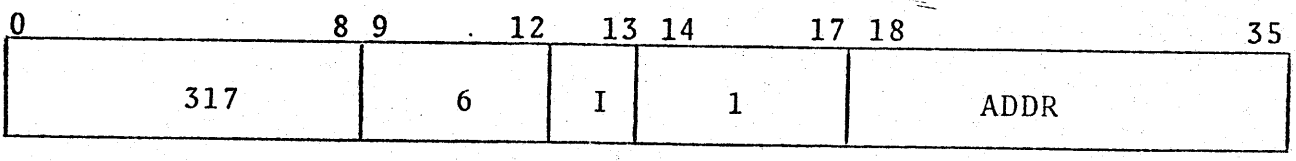
Format:
```
STSWCH  ADDR  ; or
STSWCH  @
```

where: ADDR is the address.
@ sets the I bit.

Mnemonic: LOLITS (LOad LIghTS)

Assembler definition: [LOSBKL 6,0(1)]

Structure:




Function: Load the register driving the lights with the word addressed by ADDR (if I=0) or the RAR (if I=1).

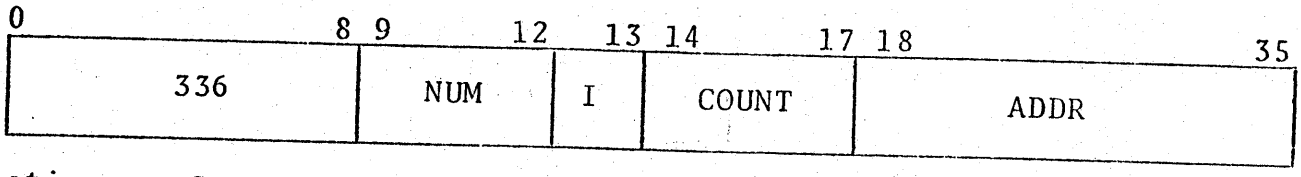
Format:
```
LOLITS  ADDR  ; or
LOLITS  @
```

where: ADDR is the address.
@ sets the I bit.

Mnemonic: STKNOB (STore KNOB)

Assembler definition: [STSBKL, 0]

Structure:

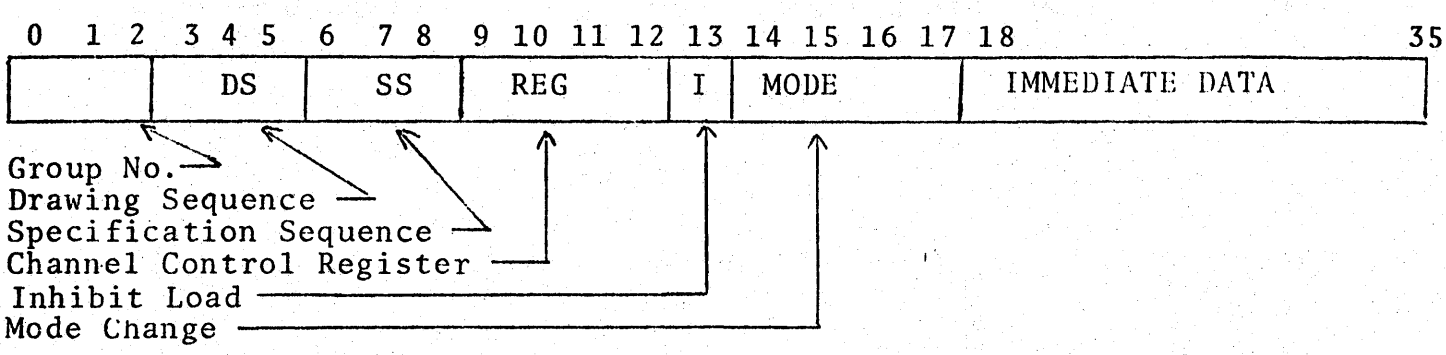



Function: Store COUNT knob pair readings, starting with pair number NUM, into core beginning at ADDR (if I=0) or the RAR (if I=1).

Format:
```
STKNOB NUM,ADDR(COUNT)
STKNOB NUM,@(COUNT)
```

where:
NUM is the first pair number
ADDR is the address
COUNT is the number of pairs (NUM + COUNT should not exceed 4)
@ sets the I bit

## 7.5 Group 4, 5, 6, and 7




Group 4-7 instructions are the only instructions that paint on the scope. Each group has its own function, but they are all discussed together because they have the same fields and the same sequence of operation.

These instructions begin by placing the immediate data into the specified Channel Control register, unless the I bit is one.

Then bits 1-8 of the instruction are copied into the RSR (Repeat Status Register).

At this point the groups diverge into their own functions:

Group 4 instructions fetch data (one word in 2D, two words in 3D) from the location held in the RAR; interpret the data according to the SS field; and draw according to the DS field. The RAR is incremented as it is used (once in 2D, twice in 3D).

Group 5 instructions fetch the word of data addressed by the RAR; put the left-half into P1 and the right half into P2; fetch data (one word in 2D, two words in 3D) first from P1 and then from P2; interpret the data according to the SS field; and draw according to the DS field. The RAR, P1 and P2 are incremented after each time they are used.

Group 6 instructions fetch no data and do not use the SS field, but rather draw using data stored internally in the Matrix Multiplier (if its CURVE bit is set) according to the DS field.

Group 7 has only one instruction, DOCHAR, which places the word addressed by the RAR into the Character String Interpreter's CHAR register; then either: 1) has the Character Generator draw the character, if possible, or 2) fetches the word whose address is the sum of the character code and the FONT register's BASE field and then executes that word as an instruction; in either case, it increments the position pointer in the FONT register and increments the RCR. If REPEAT mode is specified it continues this sequence until all the characters held in its CHAR register have been drawn or

the RCR runs out. If the CHAR register is exhausted before the RCR runs out, the RAR is incremented, the word it now addresses is placed in the CHAR register, and the procedure is resumed.

Before groups 4-7 are executed they check the MODE field of the instruction for a possible mode change. If the mode change is to REPEAT mode, the RCR is incremented.

The sequences representable by the DS and SS fields are useful mainly to update the command in REPEAT mode.

The DS field represents any of eight sequences whose basic elements are chosen from these drawing types:

set - move to a new point without drawing, make this the current point
draw to - draw to a new point, make this the current point
draw from - draw to a new point, leave the current point unchanged
dot - move to a new point, place a dot there, make this the current point
box - perform a box operation

The sequence available for the DS field are defined as follows:

LS = 60 = (set, draw to, set, draw to, etc.)
LT = 70 = (draw to, set, draw to, set, etc.)
PO = 30 = (set, draw to, draw to, etc.)
TO = 20 = (draw to, draw to, etc.)
SS = 40 = (set, draw from, draw from, etc.)
FR = 50 = (draw from, draw from, etc.)
DT = 10 = (dot, dot, etc.)
BX = 0 = (box, box, etc.)

See figure 7.1.

The SS field represents any of eight sequences of data interpretation types whose basic elements are chosen from:

absolute - accept data as is
relative - add data to current point
size absolute - form two points, the first being the negative of the data, the second being the data as is
size relative - form two points, the first being the current point minus the data, the second being the current point plus the data

SPECIFICATION OF SET/DRAW AND DOT AND BOX

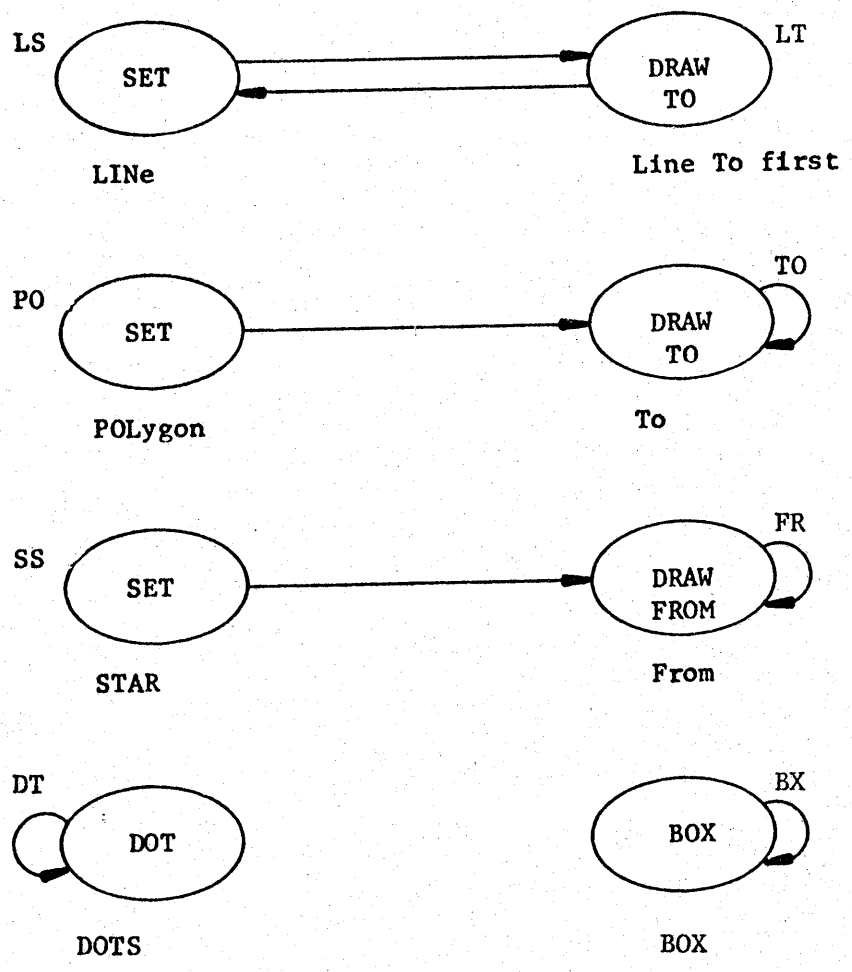



Figure 7.1

The sequences available for the SS field are defined as follows:

```
RX = 7 =  (relative, absolute, relative, absolute, etc.)
AX = 6 =  (absolute, relative, absolute, relative, etc.)
RA = 3 =  (relative, absolute, absolute, etc.)
AB = 2 =  (absolute, absolute, etc.)
AR = 4 =  (absolute, relative, relative, etc.)
RE = 5 =  (relative, relative, etc.)
SL = 1 =  (size relative, size relative, etc.)
SA = 0 =  (size absolute, size absolute, etc.)
```

See figure 7.2.

Size absolute and size relative specifications differ from absolute and relative in two important respects. First, the current point is never altered. Secondly, instead of causing a draw (or move) between the current point and the new point, they cause a draw (or move) between the two new points.

Furthermore, the groups themselves have abbreviations:

```
DD = 400 =  draw direct (Group 4)
DI = 500 =  draw indirect (Group 5)
DN = 600 =  draw internal (Group 6)
```

Group 7 has no abbreviation since it consists of only one instruction.

Mnemonics may be defined (theoretically) for any combination of one group name, one DS field, and one SS field. This can be done in a program by the statement:

```
OPDEF MNEM[GN + DS + SS]
```

where GN is a group name, DS is a drawing sequence name, and SS is a specification sequence name.

The most common and useful combinations have already been given mnemonics, which are listed on the following pages with their assembler definition, structure, and function.

Group 6 instructions have the format:

```
MNEM  (MODE)
```

and groups 4, 5, and 7 instructions have the format:

```
MNEM  REG, DATA(MODE); or
MNEM  @(MODE)
```

where: REG is the register to be loaded
DATA is the data to be placed in the register
MODE is the new mode (optional)
@ sets the I bit, inhibiting the load

Most commonly REG is the RAR, in which case it may be omitted (since RAR=0). In this case DATA is the address of the coordinates to be fetched. Since this is by far the most common usage, REG will be assumed to be the RAR on the following pages, and hence bits 9-12 will be all zeros (since RAR = 0). Also, the name ADDR will be used in bits 18-35, since in this usage that field always represents an address.

Also, bits 14-17 usually contain zeros (for no mode change) or 4 (for REPEAT mode) but may contain any mode changes. The mode field is decoded as shown on section 7.2. The customary use of that field will be stated for each instruction.

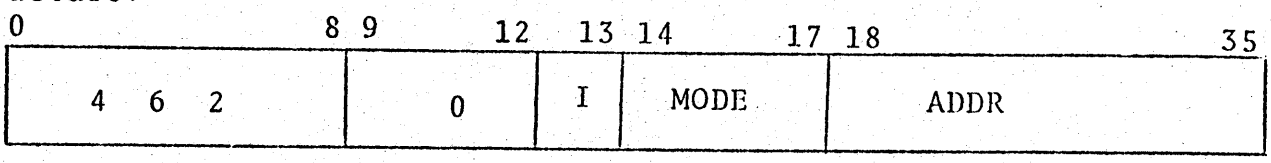
It is well to remember however that any Channel Control register may be loaded in these instructions (if the RAR is already set to correct location) and that any mode may be entered after their execution, even if REPEAT mode is also specified.

7.3.1 Instructions in Group 4

Mnemonic: SETPTA (SETPoinT Absolute)

Assembler definition: [DD + LS + AB ,0]

Structure:

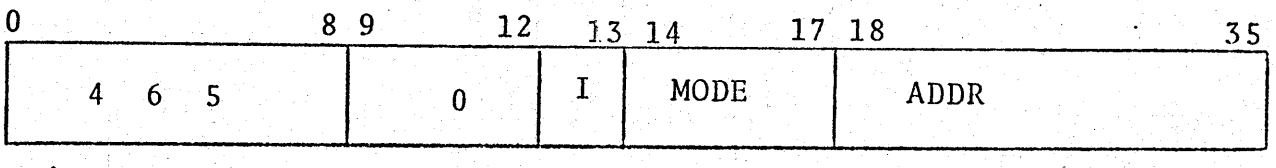



Function: Set point without drawing to indicated position; call this the current point.
REPEAT mode not generally used.

Mnemonic: SETPTR (SETPoinT Relative)

Assembler definition: [DD + LS + RE ,0]

Structure:

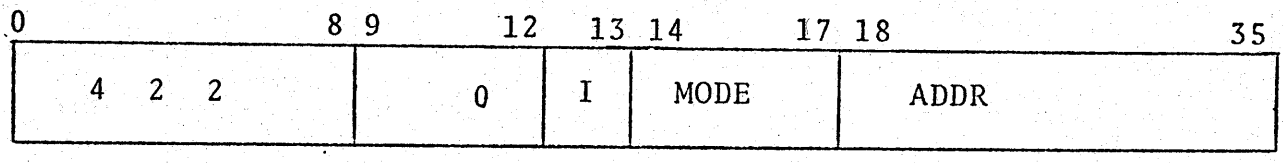



Function: Set point without drawing by indicated displacement; call new point the current point.
REPEAT mode not generally used.

Mnemonic: DRAWTA (DRAW To Absolute)

Assembler definition: [DD + TO + AB ,0]

Structure:

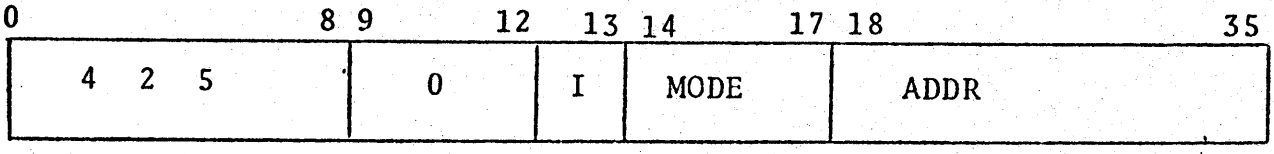



Function: Draw to indicated position; call this the current point.
REPEAT mode optional.

Mnemonic: DRAWTR (DRAW To Relative)

Assembler definition: [DD + TO + RE ,0]

Structure:

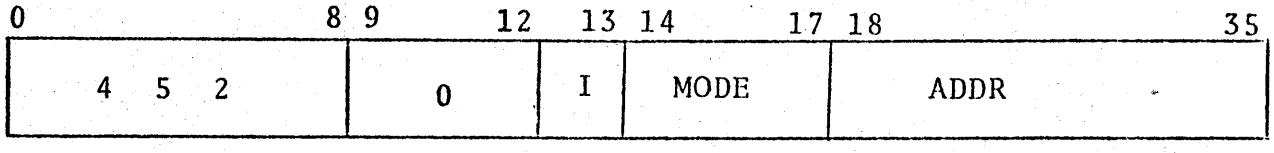



Function: Draw from current point by indicated displacement; call new point the current point.
REPEAT mode optional

Mnemonic: DRAWFA (DRAW From Absolute)

Assembler definition: [DD + FR + AB ,0]

Structure:

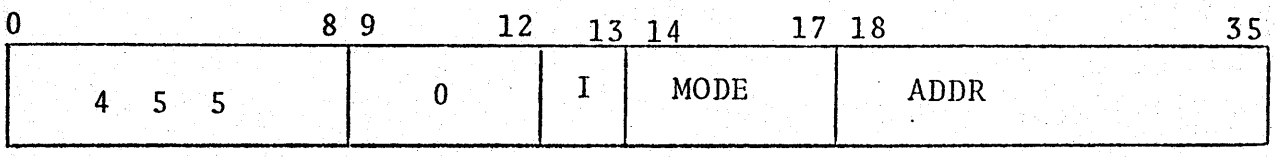



Function: Draw to indicated position; leave current point unchanged.
REPEAT mode optional

Mnemonic: DRAWFR (DRAW From Relative)

Assembler definition: [DD + FR + RE ,0]

Structure:

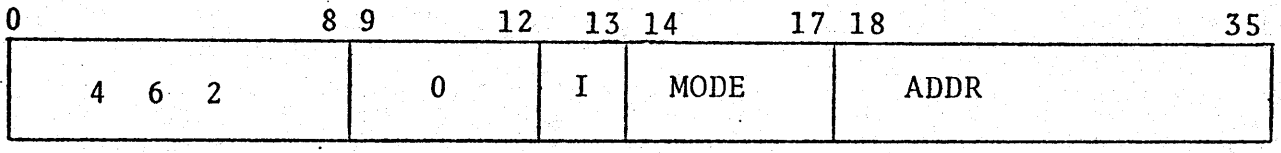



Function: Draw from current point by indicated displacement;
leave current point unchanged.
REPEAT mode optional

Mnemonic: LINAA (LINes Absolute Absolute)

Assembler definition: [DD + LS + AB ,0]

Structure:

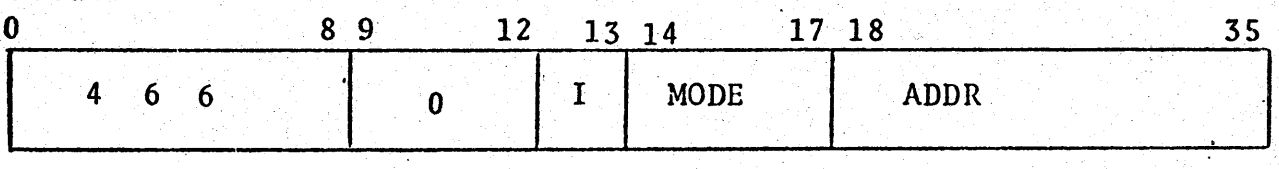



Function: Set point absolute, draw to absolute, set point absolute,
draw to absolute, etc.
REPEAT mode generally used.

Mnemonic: LINAR (LINes Absolute Relative)

Assembler definition: [DD + LS + AX ,0]

Structure:

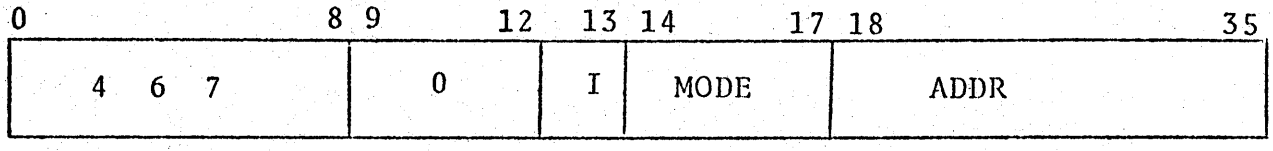



Function: Set point absolute, draw to relative, set point absolute,
draw to relative, etc.
REPEAT mode generally used.

Mnemonic: LINRA (LINes Relative Absolute)

Assembler definition: [DD + LS + RX ,0]

Structure:

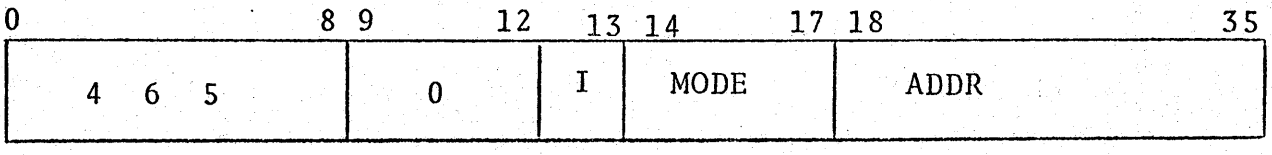



Function: Set point relative, draw to absolute, set point relative, draw to absolute, etc. REPEAT mode generally used.

Mnemonic: LINRR (LINes Relative Relative)

Assembler definition: [DD + LS + RE ,0]

Structure:

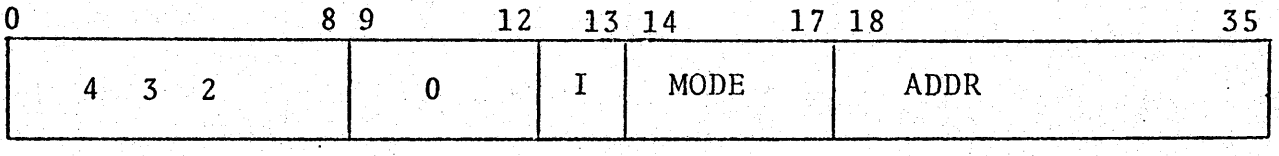



Function: Set point relative, draw to relative, set point relative, draw to relative, etc.
REPEAT mode generally used.

Mnemonic: POLAA (POLygon Absolute Absolute)

Assembler definition: [DD + PO + AB ,0]

Structure:

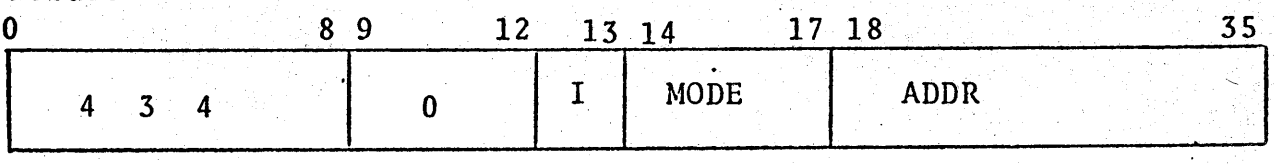



Function: Set point absolute, draw to absolute, draw to absolute, etc.
REPEAT mode generally used.

Mnemonic: POLAR (POLygon Absolute Relative)

Assembler definition: [DD + PO + AR ,0]

Structure:

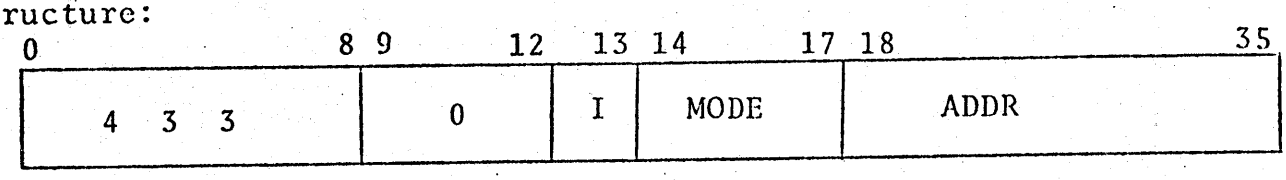



Function: Set point absolute, draw to relative, draw to relative, etc
REPEAT mode generally used.

Mnemonic: POLRA (POLygon Relative Absolute)

Assembler definition: [DD + PO + RA ,0]

Structure:

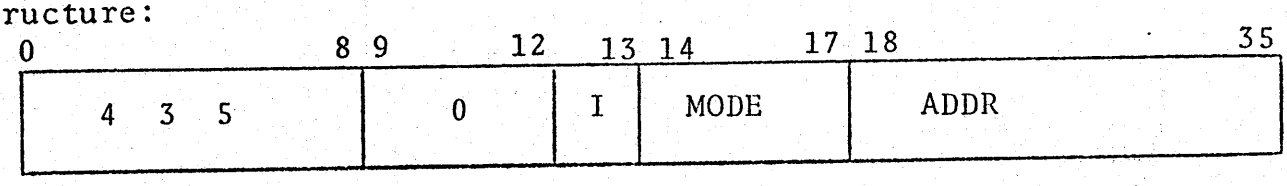



Function: Set point relative, draw to absolute, draw to absolute, etc.
REPEAT mode generally used.

Mnemonic: POLRR (POLygon Relative Relative)

Assembler definition: [DD + PO + RE ,0]

Structure:

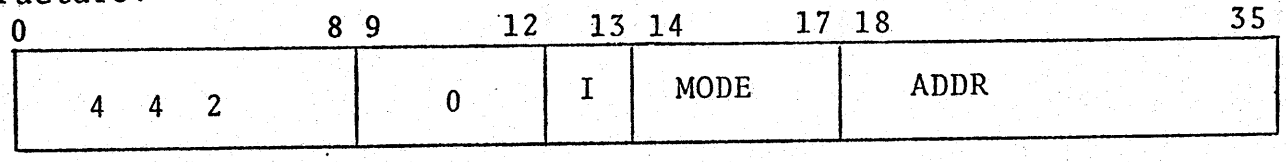



Function: Set point relative, draw to relative, draw to relative, etc.
REPEAT mode generally used.

Mnemonic: STARAA (STAR Absolute Absolute)

Assembler definition: [DD + SS + AB ,0]

Structure:

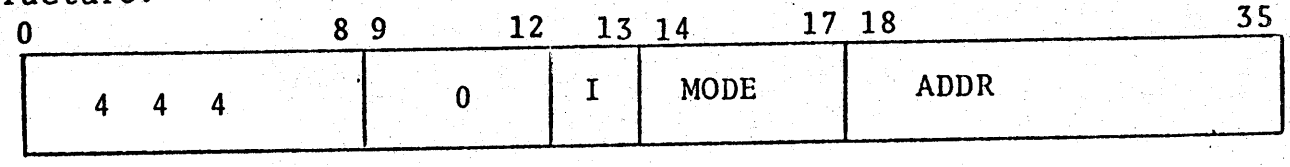



Function: Set point absolute, draw from absolute, draw from absolute, etc.
REPEAT mode generally used.

Mnemonic: STARAR (STAR Absolute Relative)

Assembler definition: [DD + SS + AR ,0]

Structure:

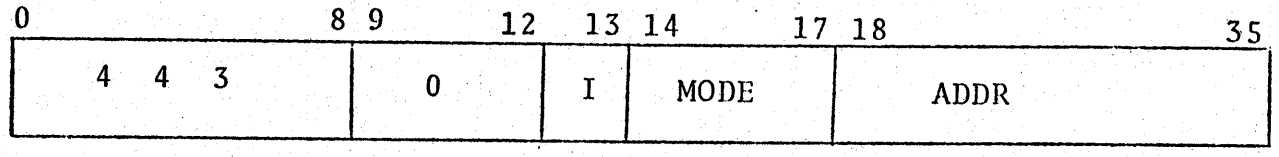



Function: Set point absolute, draw from relative, draw from relative, etc.
REPEAT mode generally used.

Mnemonic: STARRA (STAR Relative Absolute)

Assembler definition: [DD + SS + RA ,0]

Structure:

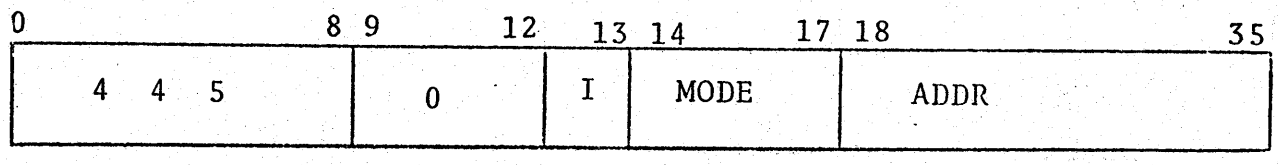



Function: Set point relative, draw from absolute, draw from relative, etc.
REPEAT mode generally used.

Mnemonic: STARRR (STAR Relative Relative)

Assembler definition: [DD + SS + RE ,0]

Structure:

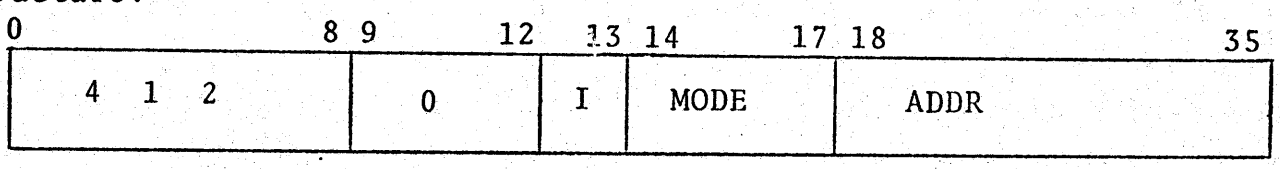



Function: Set point relative, draw from relative, draw from relative, etc.
REPEAT mode generally used.

Mnemonic: DOTSAA (DOTS Absolute Absolute)

Assembler definition: [DD + DT + AB ,0]

Structure:

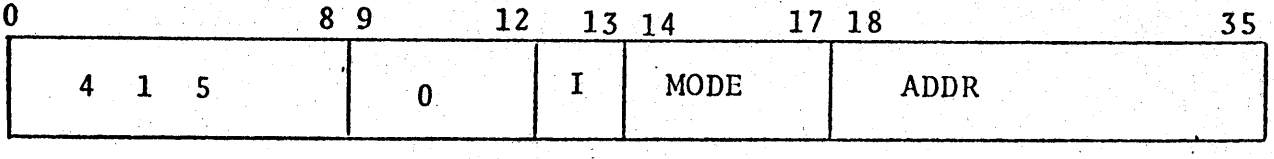



Function: Draw a dot at specified position; call this the current point.
REPEAT mode optional.

Mnemonic: DOTSRR (DOTS Relative Relative)

Assembler definition: [DD + DT + RE ,0]

Structure:

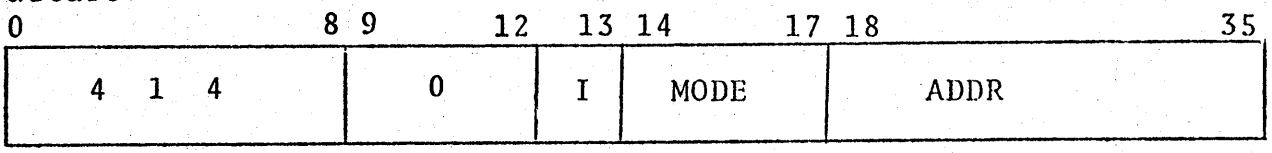



Function: Draw a dot at current point offset by indicated displacement; call this new point the current point.
REPEAT mode optional.

Mnemonic: DOTSAR (DOTS Absolute Relative)

Assembler definition: [DD + DT + AR]

Structure:

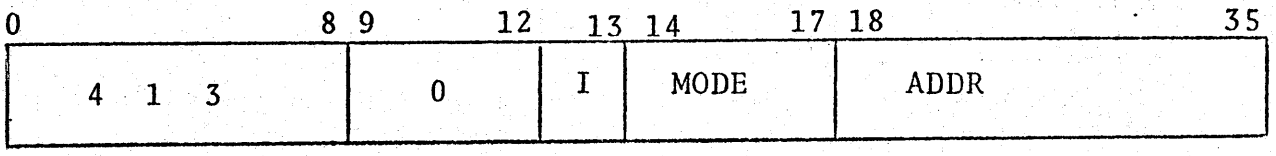



Function: Dot absolute, dot relative, dot relative, etc.
REPEAT mode generally used.

Mnemonic: DOTSRA (DOTS Relative Absolute)

Assembler definition: [DD + DT + RA ,0]

Structure:

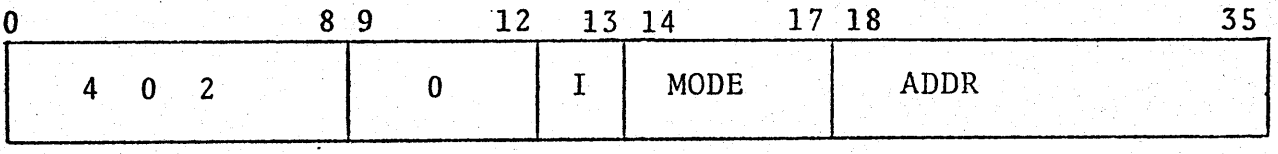



Function: Dot relative, dot absolute, dot absolute, etc.
REPEAT mode generally used.

Mnemonic: BOXA (BOX Absolute)

Assembler definition: [DO + BX + AB ,0]

Structure:

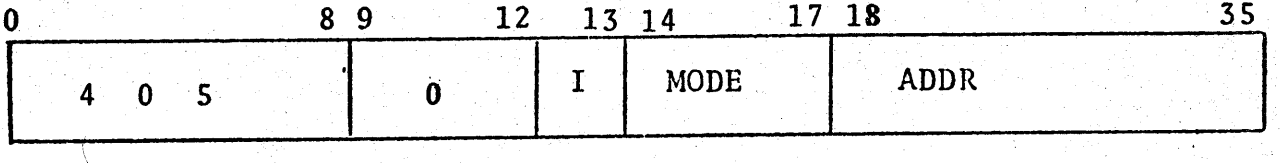



Function: Perform a box operation, creating a master rectangle whose left-bottom is the current point and whose right-top is at the indicated position.
REPEAT mode not generally used.

Mnemonic: BOXR (BOX Relative)

Assembler definition: [DD + BX + RE ,0]

Structure:

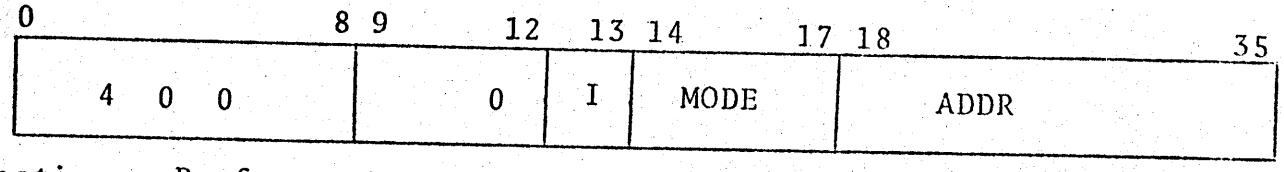



Function: Perform a box operation, creating a master rectangle whose left-bottom is the current point and whose right-top is offset by the indicated displacement.
REPEAT mode not generally used.

Mnemonic: BOXSA (BOX Size Absolute)

Assembler definition: [DD + BX + SA ,0]

Structure:

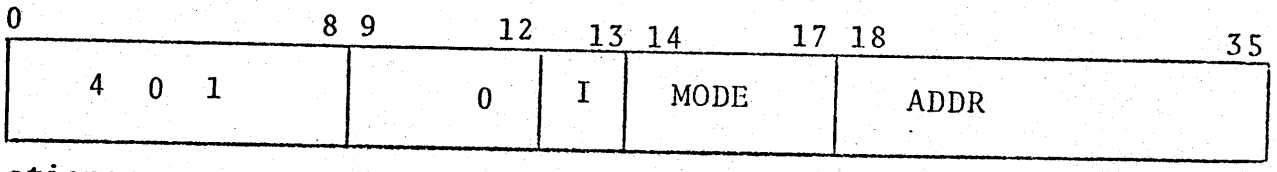



Function: Perform a box operation, creating a master whose right-top is at the indicated position and whose left-bottom is given by the negative of the coordinates of that position.

Mnemonic: BOXSR (BOX Size Relative)

Assembler definition: [DD + BX + SL ,0]

Structure:



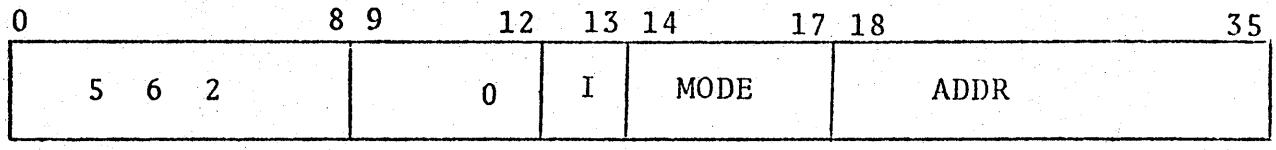

Function: Perform a box operation, creating a master whose right-top is offset from the current point by the indicated displacement, and whose left-bottom is offset from the current point by the negative of the components represented by that displacement.

### 7.5.2 Instructions in Group 5

Mnemonic: LINIAA (LINes Indirect Absolute Absolute)

Assembler definition: [DI + LS + AB ,0]

Structure:

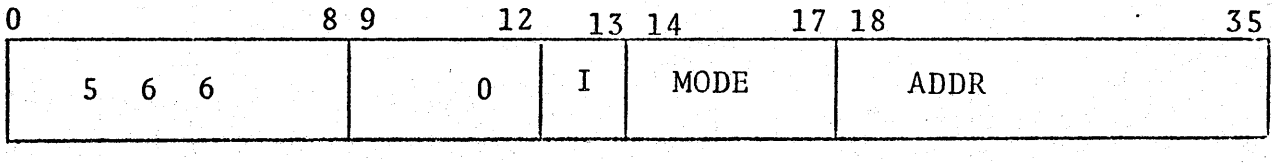



Function: Set point absolute and draw to absolute (indirect) for each pointer.
REPEAT mode generally used.

Mnemonic: LINIAR (LINes Indirect Absolute Relative)

Assembler definition: [DI + LS + AX ,0]

Structure:

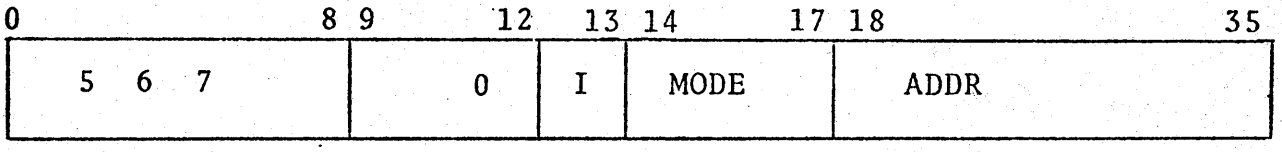



Function: Set point absolute and draw to relative (indirect) for each pointer.
REPEAT mode generally used.

Mnemonic: LINIRA (LINes Indirect Relative Absolute)

Assembler definition: [DI + LS + RX ,0]

Structure:

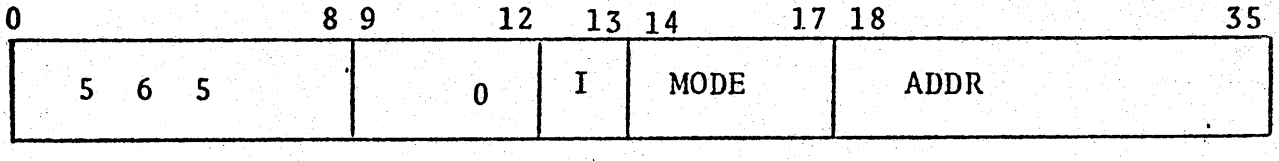



Function: Set point relative and draw to absolute (indirect) for each pointer.
REPEAT mode generally used.

Mnemonic: LINIRR (LINes Indirect Relative Relative)

Assembler definition: [DI + LS + RE ,0]

Structure:

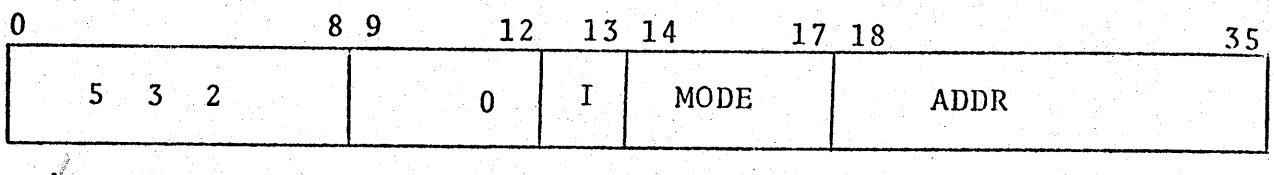



Function: Set point relative and draw to relative (indirect) for each pointer.
REPEAT mode generally used.

Mnemonic: POLIAA (POLygon Indirect Absolute Absolute)

Assembler definition: [DI + PO + AB ,0]

Structure:

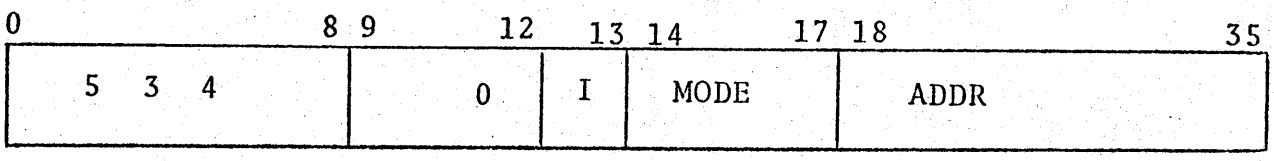



Function: Set point absolute, draw to absolute, draw to absolute, draw to absolute,....(indirect).
REPEAT mode generally used.

Mnemonic: POLIAR (POLygon Indirect Absolute Relative)

Assembler definition: [DI + PO + AR ,0]

Structure:

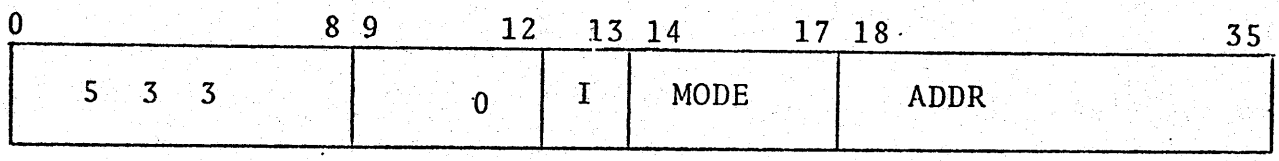



Function: Set point absolute, draw to relative, draw to relative, draw to relative,....(indirect)
REPEAT mode generally used.

Mnemonic: POLIRA (POLygon Indirect Relative Absolute)

Assembler definition: [DI + PO + RA ,0]

Structure:

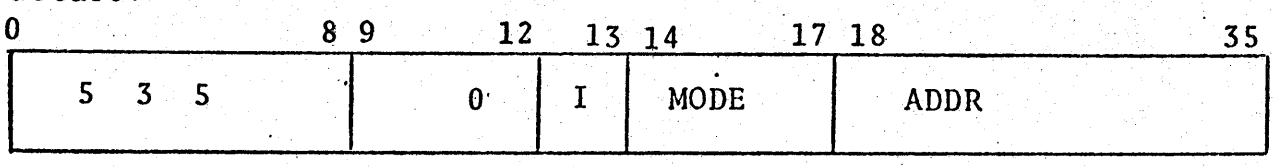



Function: Setpoint relative, draw to absolute, draw to absolute, draw to absolute,....(indirect).
REPEAT mode generally used.

Mnemonic: POLIRR (POLygon Indirect Relative Relative)

Assembler definition: [DI + PO + RE ,0]

Structure:



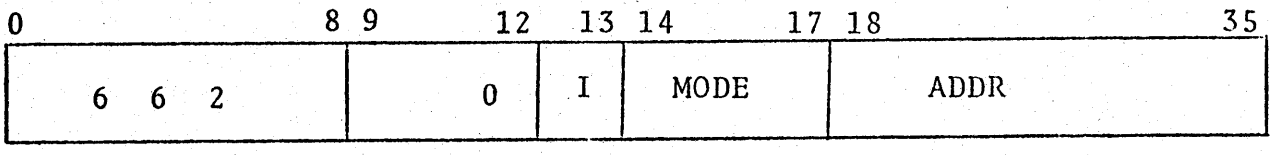

Function: Setpoint relative, draw to relative, draw to relative, draw to relative,....(indirect).
REPEAT mode generally used.

### 7.5.3 Instructions in Group 6

Mnemonic: SETCRV (SET CuRVe)

Assembler definition: [DN + LS + AB ,@0]

Structure:

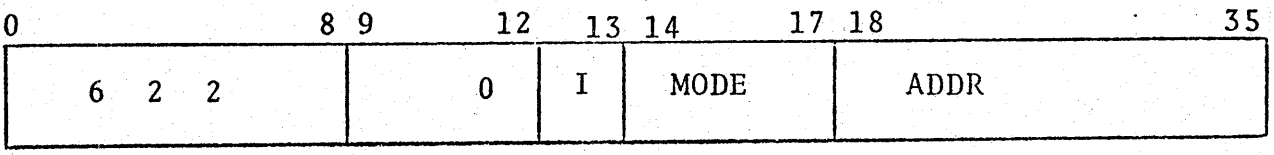



Function: Set point to the initial position held in the Matrix Multiplier's A matrix.
REPEAT mode not generally used.

Mnemonic: DRACRV (DRAw CuRVe)

Assembler definition: [DN + TO + AB ,@0]

Structure:

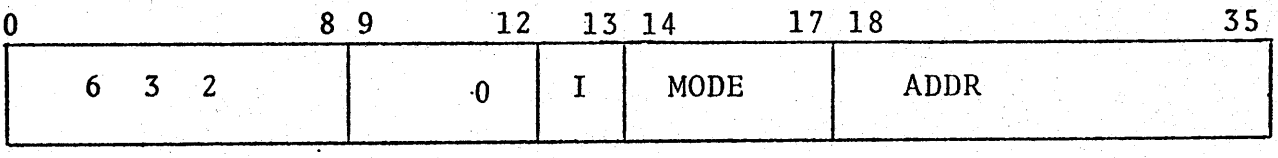



Function: Iterate the difference equation held in the A matrix and draw to the point calculated.
REPEAT mode generally used.

Mnemonic: POLCRV (POLygon CuRVe)

Assembler definition: [DN + PO + AB ,@0]

Structure:

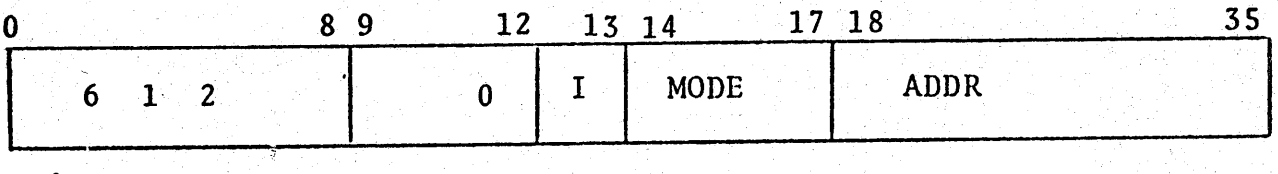



Function: Using the A matrix, set point to the initial position, and repeatedly iterate and draw to.
REPEAT mode generally used.

Mnemonic: DOTCRV (DOT CuRVe)

Assembler definition: [DN + DT + AB ,@0]

Structure:

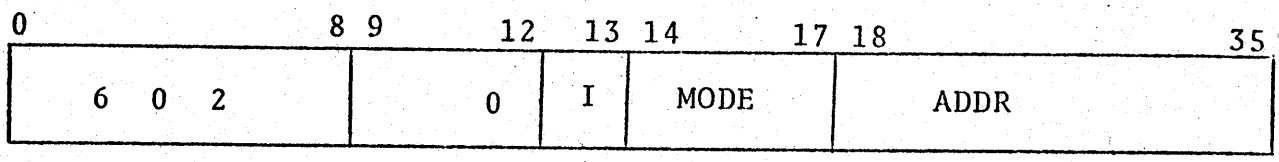



Function: Iterate the difference equation held in the A matrix and draw a dot at the point calculated.
REPEAT mode generally used.

Mnemonic: NEWCRV (NEW CuRVe)

Assembler definition: [DN + BX + AB ,@0]

Structure:



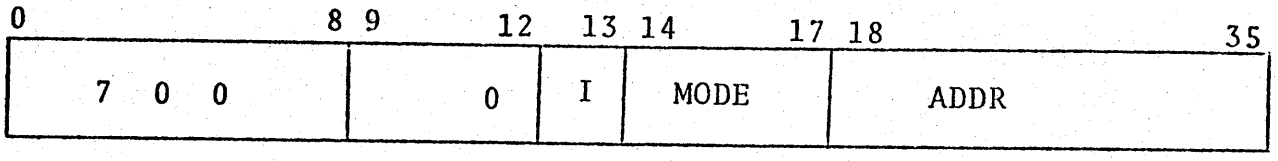

Function: Perform a single iteration along the rods of the Matrix Multiplier.
REPEAT mode not generally used. (Used to draw surface patches)

## The Group 7 Instruction

Mnemonic: DOCHAR (DO CHARacters)

Assembler definition: [7000000000000]

Structure:




Function: Draw the characters represented by the indicated words. REPEAT mode generally used.

# CHAPTER 8

# THE INSTRUCTION SET - FUNCTIONAL BREAKDOWN

The instructions of the LDS-1 may be usefully classified by group number, as described previously, but from a programming viewpoint it is probably more convenient to classify them by function. This section will list the functional areas into which they fall and describe each in such a context.

## 8.1 Channel Control Register Loading

The registers of the Channel Control are the keys to the operation of the LDS-1. They contain the pointers, control words, counts, intermediaries, addresses, and statuses around which the machine revolves. They are actually loaded far more often than the programmer realizes, sometimes as a consequence of other instructions, sometimes directly but in a hidden fashion. The following are instructions which can be used by a programmer to load Channel Control registers.

### LI, LAL

The following instruction allows loading of a register with an immediate value:

```
LI  REG, DATA(NEWMOD)
```

where REG is the register to be loaded, DATA is the immediate data to be placed into REG, and NEWMOD is an optional field for mode change. Mode changes will be discussed later. For example,

```
LI  RCR, -10
```

would fill the RCR register with -10.

Another instruction exists which is identical in function to LI, but is slower. It has the form:

```
LAL  REG, DATA
```

but it is generally pointless not to use LI. Note that LAL has no mode change field.

### LIF

It is also possible to load a Channel Control register only if a specific condition holds, by the instruction LIF:

```
LIF  REG, DATA(CNDTN)
```

where register REG is to be loaded with the immediate data

DATA only if condition CNDTN is "set" (or "1" or "on").
See section 8.4.
For example,

```
LIF  WAR, ADDR(PF2)
```

puts ADDR into the WAR if PF2=1.

To invert the sense of the test, i.e. to load the register if the condition does not hold, an "@" should be included in the instruction. E.g.,

```
LIF  SR,@0(AICF)
```

will zero the SR if AICF is not set.

## 8.2 Push-Down Stack Manipulation

The push-down stack is the mechanism for storing (saving) the values of the Channel Control registers. The stack is merely an area of core set aside for this purpose. There is a Channel Control register called the stack pointer (SP) which always contains the address of the last-filled location in the stack and is used by the stack instructions. The SP is automatically decremented every time a register is pushed onto the stack and incremented whenever one is peeled off it. It can be initially loaded using an LI instruction containing the location of (the highest address of) an area to use for the stack.

A brief discussion is in order on exactly what it means to push a register. Pushing amounts to writing an instruction in the stack area. The instruction is created by the LDS-1. The instruction written is a load immediate (LI) of the register with its value at the time of pushing. The register may then be modified and later its old value may be restored by peeling from the stack, which amounts to executing the LI instruction written there. This is accomplished by the LDS-1 by using the SP rather than the program counter (PC) for instruction fetching.

### PSH

The basic instruction for pushing is:

```
PSH  REG,(NEWMOD)
```

where REG is the register to be pushed and NEWMOD is the optional mode change field. As an example, suppose the RCR contains -50 and the instruction:

```
PSH  RCR,
```

is executed. The SP would be decremented and at the location it then addresses would be written:

```
LI  RCR,-50
```

### PEEL, PEELM

To restore the RCR to -50, it is possible to enter PEEL mode by executing the instruction:

```
PEEL
```

It can also be restored by appending (PEELM) to any group 0 or group 4-7 instruction, as for example:

```
LI  DIR,002000(PEELM)
```

Entering PEEL mode causes the SP register rather than PC to serve as program counter. In this mode, the SP acts just as the PC normally does, pointing to the instruction to be fetched and being incremented after use. Thus pushing takes one instruction per register, but many registers may be peeled at once.

PSHM

The Channel Control stays in PEEL mode until encountering an instruction indicating a return to PROG mode, and returning the instruction fetching job to the PC. The last instruction to be peeled must be marked with a return to PROG mode. This is done at the time of pushing registers by using, instead of PSH, the instruction:

```
PSHM  REG,(NEWMOD)
```

for "push and mark", where REG is the register and the optional NEWMOD is for mode change. For example, if the SR contains 0, the instruction:

```
PSHM  SR,
```

would write the following at the SP location:

```
LI  SR,0(PROGM)
```

and a subsequent PEEL would zero the SR and return to PROG mode. Since execution in PEEL mode proceeds in the reverse order of stacking, the marked register load instruction should be the first one stacked in a series of register loads to be peeled.

LIPSH, LIPSHM

The instructions PSH and PSHM may be combined with LI for more compact and efficient code with the instructions:

```
LIPSH  REG, DATA(NEWMOD)
```

and

```
LIPSHM  REG, DATA(NEWMOD)
```

where REG is the register to be pushed, DATA is the immediate data to be loaded into REG after the old data is pushed, and NEWMOD is optional for mode change. Thus,

```
LIPSH  RCR,-100
```

is generally equivalent to

```
PSH  RCR,

LI  RCR,-100
```

(except when the register is SP, in which case the new SP value loaded is used as the push pointer).

Incidentally, the register P2 is used as temporary storage in the push operation. Therefore the instruction:

```
LIPSH  P2,DATA
```

pushes not the old value of P2, but rather the new one, DATA, onto the stack. This peculiarity may be used to push data onto the stack with a single instruction.

NWSTK

A new stack may be created as a continuation of an old stack, complete with a pointer back to the old stack, by the instruction:

```
NWSTK  ADDR
```

where ADDR is the address where the new stack is to start. NWSTK is a LIPSH of the SP.

NWSTKM

To mark the pointer back to the old stack, the instruction:

```
NWSTKM  ADDR
```

should be used in lieu of NWSTK.

## 8.3 Program Control

Normally, LDS-1 instructions are executed in sequence, as the program counter is incremented following each instruction fetch. However, program flow can be altered in a variety of ways.

### JMP, JAL

Unconditional jump is accomplished by the instruction:

```
JMP  ADDR(NEWMOD)
```

where ADDR is the location to which to jump and NEWMOD is an optional field for mode change. There is another instruction which performs the same function:

```
JAL ADDR
```

but JAL is slower than JMP. No mode change may appear in a JAL.

### JIF

A conditional jump is accomplished by:

```
JIF ADDR(CNDTN)
```

where transfer to ADDR will be effected if condition CNDTN is set. See the section 8.4. Otherwise, the next instruction in sequence will be executed. It is possible to jump if and only if a condition is not set, by inserting an "@" in the instruction. For example,

```
JIF @X(HITF)
```

will cause a jump to X if HITF is not set.

### JMPPSH

A very useful instruction is one that transfers but saves the current program counter, as a subroutine call. In the LDS-1, write:

```
JMPPSH SUBR(NEWMOD)
```

where SUBR is the address of the beginning of the subroutine, and can contain any normal instruction. NEWMOD is an optional mode change field. JMPPSH is actually a LIPSHM of the PC, thus pushing and marking the PC and resetting it to the subroutine beginning. Return from the subroutine is accomplished by a simple PEEL which restores the PC and returns to PROG mode.

Upon entering a subroutine, it is necessary to save any Channel Control registers which will be changed by the subroutine. A convenient way to enter and exit from subroutines is shown below:

```
        JMPPSH SUBR
            :
            :
SUBR:LIPSH  REG1, DATA1;    Save REG1
     LIPSH  REG2, DATA2;    Save REG2
            :
            :
     ;  Subroutine instructions (may call another sub-
            :       routine)
            :
            :
            :
     PEEL;                  Exit
```

The JMPPSH pushes the subroutine return location onto the stack marked with a return to PROG mode. Thus when peel mode is entered all of the information on the stack will be restored to the appropriate registers, and finally the PC will be restored to the address of the instruction after the JMPPSH. Since this instruction on the stack is marked, the Channel Control will leave peel mode and continue on with the program.

XQTA

The LDS-1 has an execute instruction:

```
XQTA  ADDR
```

which executes the instruction whose address is ADDR. What this instruction does is load the Read Address Register (RAR) with ADDR, change to execute mode to execute the instruction addressed by the RAR, and change back to normal (PROG) mode. Normal execution is then continued at the instruction after the XQTA unless the instruction in ADDR is also marked with mode change (like another execute!).

XQT, XQTM

If the RAR is already pointing to the instruction to be executed, it is necessary only to enter EXECUTE mode. This can be done by appending (XQTM) to any group 0 or group 4-7 instruction, e.g.:

```
LIPSH  DIR, 100010(XQTM)
```

which causes EXECUTE mode to be entered after pushing and loading the DIR register, or by the instruction:

```
XQT
```

which is a no-op causing EXECUTE mode to be entered.

If the instruction executed is a repeat mode instruction, it is convenient to store the data for that instruction in a table immediately following the instruction since the RAR is pointing at the instruction. For example, the sequence:

```
        XQTA POL1
             :
             :
             :
POL1:POLAA  RCR,-4(RPTM)
DATA:P1
     P2
     P3
     P4
```

will execute the polygon instruction at POL1. Since the RAR is already pointing to the DATA (remember that the RAR is incremented each time it is used), the RCR can be loaded as part of the polygon instruction. When the polygon instruction is finished, the instruction in the location immediately following the XQTA will be fetched and the Channel Control will continue with the program.

It is possible to combine execute and repeat mode in another way. The instruction sequence:

```
        LI     RAR,  DATA
        POLAA  RCR, -10(RPTM+XQTM)
DATA:P1
     P2
     :
     :
     :
     P10
     POLAA  RCR, -4(RPTM+XQTM)
DAT2:P11
     P12
      :
     Etc.
     JMP   NXTJOB
```

forms a chain of polygon instructions. The polygon instruction is executed first in repeat mode. Execute mode is then entered and the next polygon instruction (which is being pointed to by the RAR) is executed, causing repeat mode to be entered, etc. This sequence can be continued as far as desired. The final instruction in the chain should be a JMP to reset the PC to the next instruction.

IJNRCR

There is a class of instructions facilitating program loops. (Also see condition manipulation.) The instruction:

```
        IJNRCR  ADDR
```

increments the RCR and jumps to ADDR if the RCR is negative.

IJNWCR

The instruction:

```
IJNWCR  ADDR
```

does the same thing but for the WCR.

IJPRCR

The instruction:

```
IJPRCR  ADDR
```

increments the RCR as above, but jumps to ADDR if the result is positive.

IJPWCR

The instruction:

```
IJPWCR  ADDR
```

is the WCR counterpart.

These statements are used at the end of a program loop when the RCR or WCR has been initialized before the loop to the (negative of the) number of iterations desired.

While it is generally not of consequence to the programmer, it is noteworthy that these instructions test for "less than -1" and not really "negative." This is because the actual course of events is to test, then increment, then possibly jump.

RPT, RPTM

One of the above instructions (generally IJNRCR is used) is necessary for a program loop consisting of more than one statement. However, if a single instruction is to be executed several times, a "built-in" IJNRCR may be facilitated by causing that instruction to enter REPEAT mode. This is done for group 0 or group 4-7 instructions by appending (RPTM) to them, after the RCR has been properly initialized. For example:

```
LI   RCR,-5

POLAA  ADDR(RPTM)
```

REPEAT mode says to execute the instruction addressed by the PC until the RCR runs out. REPEAT mode may be entered directly by the instruction:

```
RPT
```

which is a no-op causing REPEAT mode to be entered, but this is of dubious utility since the instruction to be repeatedly executed is a no-op.

PROG, PROGM

Another instruction of infrequent use is the call to enter PROG mode, which is the normal mode of fetching instructions sequentially from the PC. This is done by appending (PROGM) to a group 0 or group 4-7 instruction:

```
LI   RCR,-2(PROGM)
```

as is done automatically in the stack for a marked push, or directly by:

```
PROG
```

which is a no-op causing PROG mode to be entered. These instructions are not often used since the LDS-1 is generally in PROG mode when statements the programmer writes are encountered. An exception is programming for the Character String Interpreter.

## 8.4 Condition Manipulation

The LDS-1 contains several independently testable conditions. Each is one bit long and can have the value "1" (or "set" or "on") or the value "0" (or "clear" or "off"). Some can be set by the programmer and can be used as desired for program flags; other serve as signals generated by the LDS-1 or its external devices. These conditions are listed below:

| | |
|---|---|
| PF0, PF1, PF2, PF3 | Program Flags - May be used as desired |
| PF4 | Tablet Read Signal |
| PF5, PF6 | Tablet 'Z' Values |
| PF7 | Lorgnette Clock |
| RCRN- | Sign of the RCR |
| WCRN- | Sign of the WCR |
| HITF- | Hit Flag |
| AICF- | Area in Common Flag |
| PF14, PF15, PF16 | Lorgnette Values |
| STOPF | Stop Flag |

ST, CL, CM

Normally, only program flags 0 through 3 are manipulated in a program. They may be set (to a value of one):

```
ST (GNDTN)
```

cleared (to a value of zero):

```
CL (CNDTN)
```

or complemented:

```
CM (CNDTN)
```

In each case, CNDTN is the condition to be modified. For instance:

```
CM (PF3)
```

would complement program flag 3 (set it if it was cleared, or clear it if it was set).

These flags may be tested by the LIF or JIF instructions discussed earlier.

Program flags 4 through 6 (PF4, PF5, and PF6) are discussed in the tablet section.

PF7 and PF14, PF15, and PF16 discussed in the Lorgnette section.

The RCRN flag is a special condition bit, set by the Channel Control to indicate the status of the RCR. Whenever the RCR is altered, RCRN is set on if the RCR contains a value less than -1, and is set off if the RCR contains a value equal to or greater than - 1. An instruction to clear the RCRN flag has no effect on the RCRN (an effective no-op), but setting or complementing it has the unique effect of incrementing the RCR. The RCRN value is then set as an indication of the status of the RCR. RCRN may be tested by a LIF of JIF. The instruction:

```
IJNRCR  ADDR
```

is actually defined to be

```
JIFST  ADDR(RCRN)
```

for: jump to ADDR if the RCRN flag is set and then set the RCRN (i.e., increment the RCR).

WCRN is a flag identical to the RCRN except that it refers to the WCR.

HITF is a flag set automatically by the Clipping Divider whenever a line, dot, or setpoint falls within the window. It is cleared by a:

```
CL (HITF)
```

AICF is a flag set by the Clipping Divider to indicate whether there is any area in common between the window and instance. It will be discussed in detail in the next section.

STOP

STOPF is the flag which when set stops execution of the LDS-1. It may be set by:

```
ST (STOPF)
```

or its equivalent:

```
STOP
```

The stop flag may also be set by certain conditions in the LDS-1 (see Chapter 7).

These conditions are all bits of the Status Register (SR) of the Channel Control unit. As such, their values

may be set by loading the SR and saved by pushing the SR. The SR bits which are functions of LDS-1 units are set by these units. They may be loaded, but unless they are tested before the unit changes their value, the load is basically ignored.

The CONO word contains bits for clearing HITF and STOPF. See the section on communications via the I/O buss.

LALST, LALCL, LALCM, LIFST, LIFCL, LIFCM, JALST, JALCL, JALCM, JIFST, JIFCL, JIFCM

The instructions ST, CL, and CM may be combined with LAL, LIF, JAL, or JIF instructions to produce more compact and efficient code. In each case, the condition is tested before it is modified. For instance, the instruction:

```
LIFCL WAR, ADDR(PF0)
```

would first test PF0, then if set would place ADDR into the WAR, then clear PF0.

There are a total of twelve instructions that may be formed by such concatenation:

LALST - Load always and set
LALCL - Load always and clear
LALCM - Load always and complement
LIFST - Load if and set
LIFCL - Load if and clear
LIFCM - Load if and complement
JALST - Jump always and set
JALCL - Jump always and clear
JALCM - Jump always and complement
JIFST - Jump if and set
JIFCL - Jump if and clear
JIFCM - Jump if and complement

It is noteworthy that while LI is faster than LAL, LALST is faster than a ST followed by a LI. The same holds for JAL vs. JMP.

## 8.5 Drawing Instructions

This section deals with the instructions used to actually draw pictures on the scope. The format of these instructions is

```
COMMAND  REG,DATA
```

In each case, the named register is first loaded and then the drawing function is performed. Any Channel Control register can be loaded as part of these instructions.

However, the REG field may be omitted and commonly is. When it is omitted, the RAR is assumed to be the register and the format of the instruction becomes

```
COMMAND  ADDR
```

If the drawing instruction does not require the loading of the RAR (i.e. the RAR already contains the required address), the instruction format reduces further to

```
COMMAND  @
```

which bypasses the loading of the RAR. In this case only the drawing function of the instruction is performed. Use of the "@" permits a single instruction to address different locations each time it is encountered. The "@" may be used in all of the line drawing instructions with the same meaning.

By appending "(RPTM)" to the end of the drawing instruction, the drawing function is repeated under the control of the RCR (see discussion of RPTM). Thus format can become

```
COMMAND  REG,DATA(RPTM)

COMMAND  ADDR(RPTM)

COMMAND  @(RPTM)
```

In all drawing instructions, the RAR is used to obtain the coordinates of the point to be "set to", "drawn to", or whatever. These instructions fetch one word per point in 2D and two words per point in 3D. Recall that the RAR is incremented every time it is used so that it may step through a table of data in the repeat mode.

In this section on drawing instructions, all formats except the

```
COMMAND  REG,DATA
```

form are shown. Use of this format is only required to load a Channel Control register other than the RAR, and generally has no direct bearing on the drawing performed.

SETPTA

The instruction:

```
SETPTA  ADDR
```

(for SET PoinT Absolute) draws nothing, but sets the current point (in the SAVE register of the Clipper) to the point addressed by ADDR (which address is first placed in the RAR). This may be thought of as comparable to moving to a new point on a plotter with the pen up.

An alternative form to this instruction is:

```
SETPTA  @
```

which set points to the point currently addressed by the RAR, without reloading it.

The instruction:

```
SETPTA  ADDR
```

is equivalent to:

```
SETPTA  RAR,ADDR
```

However, if the RAR is already pointing to the right place, any other Channel Control register may be loaded during the same instruction, e.g.:

```
SETPTA  DIR, 004000
```

This is equivalent to:

```
LI DIR, 004000
SETPTA  @
```

SETPTR

An instruction similar to SETPTA is:

```
SETPTR  ADDR
```

or

```
SETPTR  @
```

(for SET PoinT Relative), which also draws nothing but adds the coordinates of the current point to the coordinates of the point addressed by ADDR and moves the result back into

the current point. As an example, suppose the program is in 3D and the current point is (50,20,100). Now suppose the instruction:

```
SETPTR  X
```

is executed, where X is defined:

```
X:  XWD  5,2
    XWD  10,10
```

Then the current point would be redefined to be (55,22,110).

DRAWTA

The instruction:

```
DRAWTA  ADDR
```

or

```
DRAWTA  @
```

(for DRAW To Absolute) draws a line on the scope from the current point to the point addressed by ADDR (or by the RAR, in the second form). It also resets the current point to the point drawn to.

Another form of this instruction is:

```
DRAWTA  ADDR(RPTM)
```

or

```
DRAWTA  @(RPTM)
```

which causes the "draw to" to be executed repeatedly, until the RCR runs out. Refer back to the discussion of REPEAT mode. If the RCR is initialized to -N, then N lines will be drawn: from the current point to the point addressed by ADDR (or by the RAR), from that point to the next point, etc.

DRAWTR

A similar instruction (in all its forms) is:

```
DRAWTR  ADDR
DRAWTR  @
DRAWTR  ADDR(RPTM)
DRAWTR  @(RPTM)
```

(for DRAW To Relative) which draws lines from the current point to the point whose displacement from the current point is addressed by ADDR (or the RAR). As with DRAWTA, the

current point is reset each time to the last point drawn to.

DRAWFA

Another instruction whose forms are:

```
DRAWFA  ADDR
DRAWFA  @
DRAWFA  ADDR(RPTM)
DRAWFA  @(RPTM)
```

(for DRAW From Absolute) does the same thing as DRAWTA except that the current point is not changed after a draw. Thus, lines are drawn from the current point to the points referred to.

DRAWFR

Similarly, the instructions:

```
DRAWFR  ADDR
DRAWFR  @
DRAWFR  ADDR(RPTM)
DRAWFR  @(RPTM)
```

(for DRAW From Relative) draw lines from the current point to points obtained by adding the coordinates of the current point to the coordinates of the points addressed, never changing the current point.

POLAA

There are several instructions, useful only in REPEAT mode, which are slightly more complex. One is:

```
POLAA  ADDR(RPTM)
POLAA  @(RPTM)
```

(for POLygon, Absolute setpoint, Absolute drawing points) which perform a SETPTA followed by repeated DRAWTAs. Thus this one instruction does two different types of things: it moves to a new point and then draws. When initializing the RCR, the SETPTA should be counted as one of the repeats.

POLAR

A variation to this instruction is:

```
POLAR  ADDR(RPTM)
POLAR  @(RPTM)
```

(for POLygon, Absolute setpoint, Relative drawing points) which does SETPTA followed by repeated DRAWTRs.

POLRA

The instruction:

```
POLRA  ADDR(RPTM)
POLRA  @(RPTM)
```

(for POLygon, Relative setpoint, Absolute drawing points) does a SETPTR followed by repeated DRAWTAs.

POLRR

Finally, the instruction:

```
POLRR  ADDR(RPTM)
POLRR  @(RPTM)
```

(for POLygon, Relative setpoint, Relative drawing points) does a SETPTR followed by repeated DRAWTRs.

STARAA

Along the same vein as the POLygon type instructions are the STAR type instructions.

```
STARAA  ADDR(RPTM)
STARAA  @(RPTM)
```

(for STAR, Absolute setpoint, Absolute drawing points) performs a SETPTA followed by repeated DRAWFAs. Thus the figure this instruction draws is a set of lines emanating from a single point.

STARAR

The instruction:

```
STARAR  ADDR(RPTM)
STARAR  @(RPTM)
```

(for STAR, Absolute setpoint, Relative drawing points) performs a SETPTA followed by repeated DRAWFRs.

STARRA

The instruction:

```
STARRA  ADDR(RPTM)
STARRA  @(RPTM)
```

(for STAR, Relative setpoint, Absolute drawing points) performs a SETPTR followed by repeated DRAWFAs.

STARRR

And the instruction:

```
STARRR  ADDR(RPTM)
STARRR  @(RPTM)
```

(for STAR, Relative setpoint, Relative drawing points) performs a SETPTR followed by repeated DRAWFRs.

LINAA

Another class of drawing instructions generates unconnected lines:

```
LINAA  ADDR(RPTM)
LINAA  @(RPTM)
```

(for LINes, Absolute setpoints, Absolute drawing points). LINAA executes a SETPTA followed by a DRAWTA followed by a SETPTA followed by a DRAWTA, etc. The RCR must be initialized for the total number of SETPTAs plus the total number of DRAWTAs. For example, to draw three complete lines, the RCR must be initialized to -6.

LINAR

Another instruction in this family is:

```
LINAR  ADDR(RPTM)
LINAR  @(RPTM)
```

(for LINes, Absolute setpoints, Relative drawing points) which alternately executes SETPTAs and DRAWTRs. Thus in this instruction not only does the type of motion alternate but also the absolute-relative specification.

LINRA

A third instruction in this class is:

```
LINRA  ADDR(RPTM)
LINRA  @(RPTM)
```

(for LINes, Relative setpoints, Absolute drawing points) which alternately executes SETPTRs and DRAWTAs.

LINRR

Finally, the instruction:

```
LINRR  ADDR(RPTM)
LINRR  @(RPTM)
```

(for LINes, Relative setpoints, Relative drawing points) alternately executes SETPTRs and DRAWTRs.

DOTSAA

The LDS-1 is also capable of drawing individual dots on the scope, either one at a time or in REPEAT mode. Thus this instruction has four forms:

```
DOTSAA  ADDR
DOTSAA  @
DOTSAA  ADDR(RPTM)
DOTSAA  @(RPTM)
```

(for DOTS, Absolute first dot, Absolute subsequent dots). If REPEAT mode is not specified, a dot is drawn at the point addressed by ADDR (or the RAR). If REPEAT mode is specified, the RCR should be initialized for the desired number of dots, and ADDR (or the RAR) should point to the head of the table. Each time a dot is drawn, the current point is reset to the coordaintes of that dot.

DOTSAR

This too has variants:

```
DOTSAR  ADDR
DOTSAR  @
DOTSAR  ADDR(RPTM)
DOTSAR  @(RPTM)
```

(for DOTS, Absolute first dot, Relative subsequent dots) which draws a dot at the first point referenced and at subsequent points obtained by adding the displacement referenced to the current point, updating the current point each time.

DOTSRA

Another variant:

```
DOTSRA  ADDR
DOTSRA  @
DOTSRA  ADDR(RPTM)
DOTSRA  @(RPTM)
```

(for DOTS, Relative first dot, Absolute subsequent dots) draws a dot at the current point offset by the first displacement referenced, and then at each coordinate referenced.

DOTSRR

Finally:

```
DOTSRR  ADDR
DOTSRR  @
DOTSRR  ADDR(RPTM)
DOTSRR  @(RPTM)
```

(for DOTS, Relative first dot, Relative subsequent dots) obtains locations for drawing dots by adding displacements referenced to the current point, updating the current point each time.

The LDS-1 also has a facility for indirect addressing of points in drawing instructions. Such instructions do not address the coordinates directly, but rather a pointer word in which each half-word addresses the actual coordinates for a point (thus giving two points).

These instructions may use an "@" to inhibit loading of the RAR (which is not disturbed in the second level data fetch, incidentally) and may use REPEAT mode as before. The RCR should be initialized to the number of pointer words, not the number of points to be referenced.

POLIAA

Indirection is generally used with POLYGON and LINES instructions, and each of the direct drawing versions of these two types has an indirect version. For one:

```
POLIAA  ADDR(RPTM)
POLIAA  @(RPTM)
```

(for POLygon, Indirect, Absolute setpoint, Absolute drawing points) breaks up the word addressed by ADDR (or the RAR) into two words, then SETPTAs using the point at the first address, then DRAWTAs using the point at the second address. It then breaks up the word in ADDR + 1 (or the incremented RAR) and DRAWTAs using the point addressed by the first half, the DRAWTAs using the point addressed by the second half, etc.

For example, suppose that this code is executed in

```
LI      RCR, -2
POLIAA  TRIANG(RPTM)
```

where TRIANG is defined:

```
TRIANG:  XWD PT1,PT2
         XWD PT3,PT1
```

and the points are defined:

```
PT1:     XWD  10,20
PT2:     XWD  50,50
PT3:     XWD  50,0
```

A triangle would then be drawn with vertices at PT1, PT2, and PT3. In 3D the only change is that PT1, PT2, and PT3 would address two words each, e.g.:

```
PT1:    XWD  10,20
        XWD  30,30
PT2:    XWD  50,50
        XWD  70,70
PT3:    XWD  50,0
        XWD  100,100
```

The points PT1, PT2, and PT3 can appear in any order and need not be contiguous, but TRIANG and its subsequent pointer words must appear as shown.

POLIAR

The indirect version of POLYGON also has its variants, e.g.:

```
POLIAR  ADDR(RPTM)
POLIAR  @(RPTM)
```

(for POLygon, Indirect, Absolute setpoint, Relative drawing points). This instruction performs like POLIAA except that the coordinates addressed by the right half of ADDR (or the RAR) and by both halves of the rest of the pointer words represent displacements from the current point, which is updated each time.

POLIRA

The instruction:

```
POLIRA  ADDR(RPTM)
POLIRA  @(RPTM)
```

(for POLygon, Indirect, Relative setpoint, Absolute drawing points) performs as expected, the left half of ADDR (or the RAR) pointing to a displacement from the current point, and the right half of that word and both halves of succeeding words pointing to coordinate points to be drawn to.

POLIRR

```
POLIRR  ADDR(RPTM)
POLIRR  @(RPTM)
```

(for POLygon, Indirect, Relative setpoint, Relative drawing points) breaks up each pointer word into addresses of displacements from the current point, which is updated after the set point and each draw.

LINIAA

The other type of drawing instruction which uses indirection is LINES. It is particularly handy here since each pointer word addresses the two endpoints of a line to be drawn. For one:

```
LINIAA  ADDR(RPTM)
LINIAA  @(RPTM)
```

(for LINes, Indirect, Absolute setpoints, Absolute drawing points) first breaks up the word addressed by ADDR (or the RAR) into two pointer words; then SETPTAs to the point addressed by the first half; and then DRAWTAs the point addressed by the second. It then does the same for each word after ADDR (or the RAR) until the RCR runs out.

LINIAR

Another version:

```
LINIAR  ADDR(RPTM)
LINIAR  @(RPTM)
```

(for LINes, Indirect, Absolute setpoints, Relative drawing points) does the same thing but interprets the second half of each pointer word as pointing to a displacement from the point addressed by the first half.

LINIRA

```
LINIRA  ADDR(RPTM)
LINIRA  @(RPTM)
```

(for LINes, Indirect, Relative setpoints, Absolute drawing points) interprets the word addressed by the left half of each pointer word as a displacement and the word addressed by the second half of each as a point to be drawn to.

LINIRR

```
LINIRR  ADDR(RPTM)
LINIRR  @(RPTM)
```

(for LINes, Indirect, Relative setpoints, Relative drawing points) interprets all pointer-word halves as addressing displacements from the current point, which is updated each time.

The preceding is a complete list of the drawing instructions defined in the LDS-1 adjunct to the PDP-10 assembler mnemonics. It is not however a complete list of the drawing instructions available. For instance, indirection is provided in the Assembler with the LINES and POLYGON sequences but not with the STAR sequence. Also not provided in the Assembler are size absolute and size relative specifications in drawing instructions. While these unusual combinations are generally not needed, instructions calling for them may easily be defined and used by the programmer. See section 7.5 for the method of doing this.

## 8.6 Clipping Divider Instructions

### 8.6.1 Clipping Divider Register Transmission

The registers of the Clipping Divider (Clipper) may be loaded from memory, stored into memory, placed in a data sink, and retrieved from the data sink. These operations constitute the topic of this section.

The Clipping Divider registers are referred to as two-component or four-component registers. Two-component registers are 40 bits long and generally contain two distinct 20-bit pieces of information. Four-component registers are 80-bits long consisting of a pair of related two-component (40-bit) registers.

The distinction is important because the method of loading, storing, etc. depends on whether the LDS-1 is in 2D or 3D because this in turn determines how many words should be referred to in a data fetch or store.

The registers are arranged in the following configuration and can be referred to by name or number (octal):

| | NAME | NUMBER | DESCRIPTION |
|---|---|---|---|
| two-component | SAVELB | 0 | SAVE left and bottom |
| | SAVERT | 1 | SAVE right and top |
| | VIEWLB | 2 | VIEWPORT left and bottom |
| | VIEWRT | 3 | VIEWPORT right and top |
| | WINDLB | 4 | WINDOW left and bottom |
| | WINDRT | 5 | WINDOW right and top |
| | INSTLB | 6 | INSTANCE left and bottom |
| | INSTRT | 7 | INSTANCE right and top |
| | NAME | 10 | NAME register |
| | CDIR | 11 | Clipper Directive Register |
| | HITANG | 12 | Hit Count and Angle Count |
| | SELINT | 13 | Scope Select and Intensity |
| four-component | SAVE | 14 | SAVELB and SAVERT |
| | VIEW | 15 | VIEWLB and VIEWRT |
| | WIND | 16 | WINDLB and WINDRT |
| | INST | 17 | INSTLB and INSTRT |

There are four instructions for loading the Clipper registers representing four types of loads: absolute, relative, size absolute, and size relative. In all these instructions the specified address may be replaced by an "@". In this case the address used is the current value of the RAR.

**LOCLA**

The absolute specification:

```
LOCLA REG,ADDR(COUNT)
```

loads the number of registers specified by COUNT, beginning at

register REG, from memory beginning at ADDR. In 2D, one 36-bit memory word is placed in each 40-bit register, placing 18 bits with sign extension in each 20-bit half of the register. In 3D, two contiguous words are put in each register. Thus, only four-component registers should normally be loaded in 3D. If a four-component register is specified in 2D, the current point is placed in the left-bottom and the addressed data in the right top.

If only one register is to be loaded, COUNT should be 1. Specifying a COUNT of 0, or leaving out the (COUNT) field, calls for all sixteen registers to be loaded, something that is seldom desired.

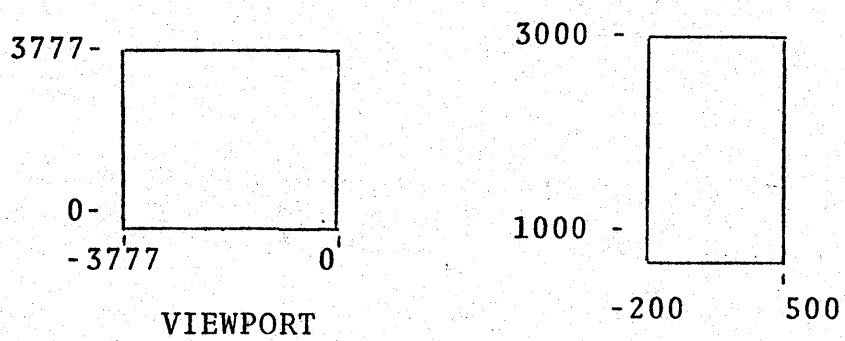
Suppose that the LDS-1 is in 2D and the statement:

```
LOCLA  VIEWLB, V(4)
```

is encountered, where V is defined in PDP-10 assembly code as four contiguous words:

```
V:   XWD -3777,0    ; VIEWPORT LEFT,BOTTOM
     XWD  0,3777    ; VIEWPORT RIGHT,TOP
     XWD -200,1000  ; WINDOW LEFT,BOTTOM
     XWD  500,3000  ; WINDOW RIGHT,TOP
```

Then VIEWLB would be set to -3777,0; VIEWRT would be 0,3777; WINDLB would be -200,1000; and WINDRT would be 500,3000. This would define the viewport and window to be:




VIEWPORT
(Second quadrant of the scope) WINDOW

Suppose now that the machine is in 3D and the same window and viewport are to be defined. Then the proper instruction is:

```
LOCLA  VIEW, V(2)
```

where V is as before.

The absolute specification is the only way Clipping Divider registers may be loaded in 3D. The following discussion of other specifications applies only to the 2D case.

LOCLR

For a relative load, use the instruction:

```
LOCLR  REG,ADDR(COUNT)
```

where REG and COUNT are as before but the data in ADDR represent a displacement from the current point, held in the SAVE register. In 2D, the instruction:

```
LOCLR  INSTLB,LOC(2)
```

would add the left half of LOC to $SAVE_x$ and the right half of LOC to $SAVE_y$ and put the result in INSTLB; then add the left half of LOC+1 to $SAVE_x$ and the right half of LOC+1 to $SAVE_y$ and put the result in INSTRT.

LOCLSA

In addition to absolute and relative loads, there is the size absolute specification used almost exclusively to define the window, viewport, or instance:

```
LOCLSA  REG,ADDR(COUNT)
```

Here, REG is a four-component register and COUNT is the number of such four-component registers to be loaded. The register will be loaded with the coordinates in ADDR in its right-top and the negative of those coordinates in its left-bottom. Thus, the rectangle it refers to will be centered at (0,0) and have its half-length in X and Y specified in ADDR.

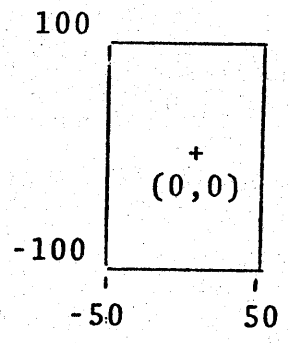
For example, the instruction:

```
LOCLSA  WIND,W(1)
```

where W is defined as:

```
W:  50,100
```

would define the window:




LOCLSR

The fourth specification, size relative, is like size

absolute except that the current point (held in the SAVE register) becomes the center of the rectangle. Thus in:

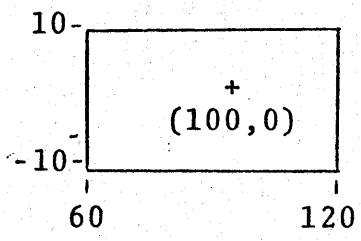
```
LOCLSR  REG,ADDR(COUNT)
```

the fields are the same as for LOCLSA.

For instance, if SAVE contained (100,0) and V is defined as (20,10), the instruction:

```
LOCLSR  VIEW,V(1)
```

would define this viewport:




STCL

Storing the Clipper registers is a much simpler process, since there is only one specification - absolute. The basic instruction is:

```
STCL  REG,ADDR(COUNT)
```

which means that COUNT registers, beginning at REG, are to be written into memory locations beginning at ADDR.

In 2D, one memory location is filled for each register stored, and only two-component registers should be referenced. For example:

```
STCL  CDIR,LOC(3)
```

would write the Clipper Directive into LOC, the HITANG into LOC+1, and SELINT into LOC+2.

In 3D, two contiguous memory locations are filled for each register referenced, which should be only four-component registers. For instance:

```
STCL  SAVE,LOC(2)
```

would write the SAVELB into LOC, the SAVERT into LOC+1, the VIEWLB into LOC+2, and the VIEWRT into LOC+3.

SKCL

The data sink is an area of core set aside as a push-down stack for Clipping Divider registers. The Channel Control register DSP (Data Sink Pointer) keeps track of the next location to be filled in the sink.

Registers may be pushed onto the sink by the instruction:

```
SKCL  REG,(COUNT)
```

where the number of registers specified by COUNT and beginning at REG are written into the data sink.

As with STCL, one location is filled per register in 2D and two locations per register are filled in 3D, and consequently, only two-component registers should be referenced in 2D and only four-component registers in 3D.

For instance, in 2D the instruction:

```
SKCL  INSTLB,(3)
```

would write the values of the INSTLB, INSTRT, and NAME onto the data sink.

In 3D,

```
SKCL  SAVE,(1)
```

would sink SAVELB and SAVERT.

RTCLA

A wider variation of methods exists for retrieving Clipper registers from the sink after they have been placed there. In fact, retrieving is exactly analogous to loading in the methods available and the restrictions on each. A unique feature of the retrieve instructions is that they count backwards through the register configuration. This is so that registers may be retrieved in the inverse order that they were sinked, in order that each receives its proper former value.

Registers may be retrieved exactly as they sit in the data sink by the instruction:

```
RTCLA  REG,(COUNT)
```

specifying that COUNT registers going backward from REG are to be retrieved. As always, one location is referenced per register in 2D and two locations per register are referenced in 3D.

As a 2D example, suppose that the sink contains values of the INSTLB, INSTRT, and NAME, as the 2D example for SKCL put them there. Then the instruction:

```
RTCLA  NAME,(3)
```

would retrieve them correctly.

In 3D, the instruction:

```
RTCLA  SAVE,(1)
```

would restore the value of the SAVE register had the 3D example for SKCL been executed previously.

RTCLR

The specifications relative, size absolute, and size relative are available for the retrieve instructions also, but as with loading, they can be used only in 2D.

The instruction:

```
RTCLR  REG,(COUNT)
```

is a relative retrieve. The coordinates retrieved from the sink are added to the coordinates of the current point as held in the SAVE register. Suppose that the instruction:

```
SKCL  WINDLB,(2)
```

has been executed and that the SAVE register now contains (-100,0).

Then the instruction:

```
RTCLR  WINDRT,(2)
```

would create a window of the same size and shape as the previous one but 100 units to the left.

RTCLSA,RTCLSR

Of dubious utility are the instructions:

```
RTCLSA  REG,(COUNT)
```

and

```
RTCLSR  REG,(COUNT)
```

for retrieving size absolute and size relative, respectively. They operate as expected, RTCLSA loading the right-top of the specified register with the last word sinked and the left-bottom with the negative of the coordinates in that word; and RTCLSR doing the same but offsetting the result by the contents of the SAVE register.

If these instructions are used, only four-component registers should be specified.

There is no direct way to transmit two-component registers in 3D. This is no problem when the two-component register is part of a four-component register, but it does come up with the others. The best solution is to switch to 2D, do the loading or whatever, and then switch back to 3D.

### 8.6.2 Boxing

The boxing capability of the LDS-1 provides a very convenient method for placing a 2D object (i.e. symbol) at several places in a drawing, varying its size and scale if desired. A less obvious but important related capability is that of discarding the object by a simple test when it is entirely outside the window. This saves time from being wasted and permits more complex pictures to be displayed flicker-free.

The boxing instruction itself draws nothing on the scope. Rather, it modifies the window and viewport to permit drawing to take place in the desired coordinate system and still show up at the right area of the scope and picture. Prior to boxing, the Clipping Divider INSTANCE register (INST) should be loaded with the area on the current window where the object is to appear. Then the window and instance should be checked for area in common; if there is no area in common, the rest of this procedure should be skipped. The Area-In-Common bit is sent to the Channel Control STATUS Register and may be tested by an instruction like

```
JIF @NEXT(AICF)
```

Next, if the current window and viewport are to be used later, they should be sinked (or stored), because these two registers will be changed by the box instruction. The box instruction does two things: First, the viewport is redefined to be that section of the current viewport corresponding to the area in common between the instance and the old window; secondly, it redefines the window in the following fashion: the box instruction contains directions for forming a "master" rectangle on the new subpicture coordinate system, optionally independent of the old window's coordinate system; the new window is then defined to be that section of the "master" corresponding to the area in common between the instance and the old window. See figure 4.4. The subpicture may now be drawn by regular LDS-1 instructions. Finally, the old window and viewport can be retrieved (or loaded) and normal processing can continue.

There are four forms to the box instruction, corresponding to the four ways to load a Clipper register. In each case the specified address may be replaced by an "@", indicating that the current value of the RAR should be used for the address.

BOXA

The absolute form:

BOXA ADDR

defines a "master" whose lower left-hand corner is the current point (held in the SAVE register) and whose upper right-hand corner is in location ADDR.

BOXR

The relative form:

BOXR ADDR

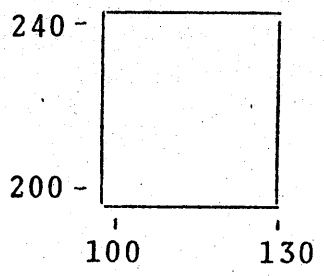
makes the lower left-hand corner of the "master" to be the current point and the upper right-hand corner to be the current point plus the contents of ADDR. As an example, suppose the SAVE register contains the point ($100_8$,$200_8$) and the instruction:

BOXR DISP

is executed, where DISP is defined to be:

DISP: XWD 30,40

Then this master would be created:




BOXSA

The size absolute specification can be used:

BOXSA ADDR

to specify a "master" whose center is at (0,0) and whose upper right-hand corner is given by the contents of ADDR, and the lower left-hand corner is given by the negative of the contents of ADDR.

Finally, the size relative specification:

```
BOXSR  ADDR
```

specifies the "master" exactly as BOXSA does, except that it is centered on the current point.

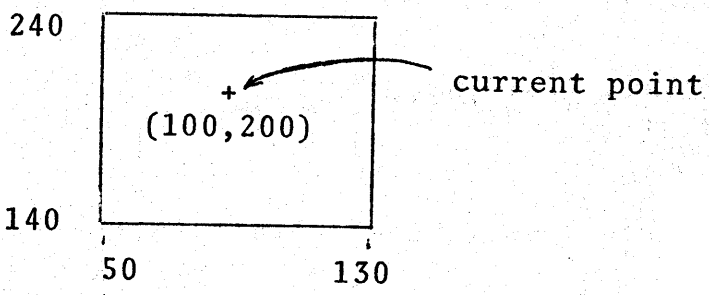
Suppose, for example, that the SAVE register contains ($100_8$,$200_8$) as before and a:

```
BOXSR  DISP
```

is executed, where DISP is also defined as before:

```
DISP:  XWD  30,40
```

Then the "master" created would be:




The important things to remember about these instructions are that they all are useful only in 2D; that they update the viewport to be the image of the area in common between the instance and the old window under the existing window-to-viewport mapping; that they differ only in how they define the "master" and therefore redefine the window; and that they throw the LDS-1 into no special "box mode" or anything, i. e. normal instructions may follow just as if the window and viewport were redefined by LOCL's instead of BOX's.

Checking for area in common is not a completely straightforward process. First of all, AICF does not necessarily test the window vs. the instance, but rather the window vs. the last register loaded. This precludes loading the instance before the window, for then AICF would check for area in common between the window and itself. Secondly, AICF is updated only on a four-component register load. Usually the program is in 2D at the time the instance is to be loaded, in which case the following procedure does work:

```
SETPTA  LB
LOCLA   INST, RT(1)
```

where LB addresses the coordinates of the left-bottom of the instance to be defined, and RT addresses its right-top coordinates.

If RT addresses the displacement necessary to obtain the right-top from the left-bottom, the code:

```
SETPTA  LB
LOCLR  INST, RT(1)
```

would work. These procedures work because the SETPTA loads the SAVE register, which is used by the LOCLA and LOCLR in a four-component 2D load to specify the first two components.

The instructions:

```
LOCLSA  INST,ADDR(1)
```

and

```
LOCLSR  INST,ADDR(1)
```

work as expected, update the AICF, and need not be preceded by a SETPTA.

The following is a complete example of all the steps involved in boxing:

```
      SETPTA  LB            ;PREPARE THE SAVE
      LOCLA  INST, RT(1)    ;LOAD INSTANCE
      JIF @NOBOX(AICF)      ;GO AWAY IF NO A.I.C.
      SKCL  VIEWLB, (4)     ;SINK VIEWPORT, WINDOW
      SETPTA MASTER         ;PREPARE TO BOX ABSOLUTE
      BOXA MASTER+1         ;PERFORM THE BOX OPERATION
            .
            .               ;DRAW THE OBJECT
            .
      RTCLA  WINDRT, (4)    ;RESTORE WINDOW, VIEWPORT
NOBOX:
            .
            .
            .               ;CONTINUE
LB:     XWD  40,40          ;INSTANCE LEFT,BOTTOM
RT:     XWD 200,140         ;INSTANCE RIGHT,TOP
MASTER: XWD   0,            ;MASTER LEFT,BOTTOM
        XWD 1000,1000       ;MASTER RIGHT,TOP
```

## 8.7 Matrix Multiplier Instructions

### 8.7.1 Matrix Manipulations

The matrix multiplier has a 36-bit directive register and a set of 6 18-bit numerical registers arranged in a 4 x 4 x 4 array.

LOMDIR,STMDIR:

The directive register is handled individually. It may be loaded from memory:

```
LOMDIR  ADDR
```

or stored into memory:

```
STMDIR  ADDR
```

where ADDR is the address of the memory location to be loaded from or stored into.

SKMDIR,RTMDIR:

The directive may also be placed into the same data sink as for the clipper registers:

```
SKMDIR
```

and retrieved from the data sink:

```
RTMDIR
```

The best procedure is always to perform these instructions in 2D, since in 3D they call for double word data fetches and stores. If the program is in 3D, it should be switched to 2D for the desired operation and then back to 3D.

The 64 element array is addressed by row, rows 0-3 forming the 4 x 4 matrix A, rows 4-7 forming B, rows 10-13 forming C, and rows 14-17 forming D. Thus rows are loaded, stored, etc., as a unit. The Matrix Multiplier instruction set is the same for 2D and 3D, and the row is the basic unit of transmission in each. However, as always, 2D data fetches and stores are one word each and in 3D they are two words each. In 3D, the two 36-bit words correspond to the four 18-bit elements of one register or row. In 2D, the one 36-bit word corresponds to the first two 18-bit elements, and the other two elements are ignored.

LOMM

Matrix rows may be loaded directly from memory by the instruction:

```
LOMM  REG,  ADDR(COUNT)
```

where COUNT registers beginning at REG are to be loaded from core locations beginning at DATA. In 2D, COUNT memory locations are used, one per register (row). In 3D, twice as many memory words are read, two per register.

An example is in order. The instruction:

```
LOMM  0,MATRIX(4)
```

loads matrix A (the four registers beginning at 0) from the eight locations beginning at MATRIX in 3D or from the four locations beginning at MATRIX in 2D.

LOMMR

Relative loads are facilitated by the instruction:

```
LOMMR  REG,  DATA(COUNT)
```

where REG, DATA, and COUNT are as before but the incoming vector is added element by element to the specified registers.

LOMMP

Matrices may be multiplied using the instruction:

```
LOMMP  REG,  ADDR(COUNT)
```

which multiplies COUNT input vectors starting at ADDR (one word each in 2D, two words each in 3D) by the entire matrix in which REG is located and stores each resultant vector in its corresponding row in matrix A. (One can think of the two high-order bits of REG as specifying the matrix to be multiplied, and the two-low order bits as specifying the row in A where the results are to begin being placed.)

For example, the instruction:

```
LOMMP 4, INMAT(4)
```

multiplies the matrix stored at INMAT by matrix B (where register 4 is located) and leaves the result in matrix A. To multiply by matrix C, use a REG of 10, and to multiply by matrix D, use a REG of 14. Clearly multiplying by matrix A would not give a desired result since A is changed after each row multiplication.

STMM

There is one matrix multiplier store instruction:

```
STMM REG, ADDR(COUNT)
```

which stores COUNT registers starting at REG in core locations beginning at ADDR. COUNT locations are filled in 2D, and twice as many are filled in 3D.

SKMM

Matrix Multiplier registers may be sinked into and retrieved from the data sink like Clipper registers. The basic sink instruction is:

SKMM REG, (COUNT)

specifying that COUNT registers starting at REG are to be sinked onto the sink. Each register takes up one sink word in 2D and two words in 3D.

RTMM

Retrieving Matrix Multiplier registers from the data sink is analogous to retrieving Clipper registers, except that relative, size absolute, and size relative retrieves are not permitted. The basic instruction is:

RTMM REG, (COUNT)

specifying that COUNT registers counting backward from REG are to be peeled off the data sink. The inverse order is necessary so that each register receives its proper former value.

For example:

RTMM 17, (0)

retrieves all 16 registers (as always, a COUNT of 0 calls for 16 registers).

SKMMS

The sink and retrieve instructions have useful slide variants for matrix multiplier registers. The instruction:

SKMMS REG,(COUNT)

sinks COUNT registers starting at REG and replaces each with the corresponding row from matrix A.

When entire matrices are manipulated in this fashion (COUNT=4), the data sink, appended first by matrix B, C, or D and then by matrix A, may be considered as a push-down stack of matrices.

RTMMS

The inverse of SKMMS is provided by:

```
RTMMS  REG, (COUNT)
```

which copies COUNT registers counting backward from REG into the corresponding rows in matrix A and replaces each from the data sink.

Thus if the instruction:

```
SKMMS  4, (4)
```

has been used to sink B and replace B with A, former status could be restored with:

```
RTMMS  7, (4)
```

PUSHMM

The facility exists to copy rows of matrix A into corresponding rows of the other matrices without doing any sinking.

To do this, use the instruction:

```
PUSHMM  REG, (COUNT)
```

to specify that COUNT rows starting with REG are to be filled with the corresponding rows of matrix A.

POPMM

The reverse procedure can be accomplished by the instruction:

```
POPMM  REG, (COUNT)
```

which specifies that each of COUNT rows starting with REG are to be copied into its corresponding row in matrix A.

For example, if the instruction:

```
PUSHMM  14, (4)
```

had been used to copy matrix A into matrix D, A could be later restored with:

```
POPMM  14, (4)
```

It is noteworthy that this procedure does not restore matrix D to its former status, as opposed to the SKMMS and RTMMS procedure.

NOMM

Matrices must be normalized to maintain maximal precision. While input matrices are generally normalized by the programmer, multiplying matrices together may leave an un-normalized result in matrix A. This situation may be rectified in 3D without affecting the transformation represented except to make its result more precise, by the instruction:

NOMM (MSHIFT)

This instruction shifts each element of matrix A to the left until either MSHIFT shifts have taken place, or some element of A is between one-half and one in magnitude (i.e. A is normalized). The reason why the transformation is unaffected is that factors thus introduced are cancelled out later by the perspective division.

Normally it is pointless to specify a maximum number of shifts, so that the instruction:

NOMM

(i.e. MSHIFT=0, implying a maximum shift of 16) is guaranteed to normalize A. However, normalizing takes some time, and it may be desired to limit the amount of shifting with the first option.

The NOMM instruction is ignored in 2D, because the element $a_{22}$ is always considered to be 1. See Chapter 3, Matrix Multiplier.

NOTE: Normalization is pointless unless the scale factor ($a_{33}$) is unequal to +1. Normalizing is important before using it in a concantenation.

### 8.7.2 Curves And Surface Patches

The Matrix Multiplier can be put into a special mode in which it draws 2D or 3D curves or 3D surface patches, all from data stored internally by the Matrix Multiplier. See sections 3.5 to 3.7 for a description of the representation of these curves in the Matrix Multiplier registers.

SETCRV

A setpoint (loading of the Clipper Divider SAVE register) to the beginning of a curve may be accomplished by the instruction:

```
SETCRV
```

where the initial point has been previously placed in the Matrix Multiplier's Matrix A.

DRACRV

After the RCR has been initialized to the (negative of) the number of points on a curve, the curve may be drawn by the instruction:

```
DRACRV  (RPTM)
```

POLCRV

The two operations above may be combined in the same way that a set point and draw to's (absolute) are combined in a POLYGON type instruction by using:

```
POLCRV  (RPTM)
```

where the RCR has been initialized to (the negative of) one greater than the number of lines in the curve.

For example the 2D matrix

$$A = \begin{bmatrix} \cos(2\pi/200_8) & \sin(2\pi/200_8) \\ -\sin(2\pi/200_8) & \cos(2\pi/200_8) \\ 40000_8 & 20000_8 \\ 100000_8 & 0 \end{bmatrix}$$

in curve mode with the instruction:

```
LOMM    0,A(4)
POLCRV  RCR,-201(RPTM)
```

will draw a complete circle centered at octal coordinates (40000,20000) with a radius of $100000_8$, and composed of 20 line segments.

DOTCRV

For a curve consisting of dots instead of lines, the instruction:

```
DOTCRV  (RPTM)
```

should be used instead of DRACRV, where the RCR has been initialized to the number of dots per curve.

Note that DOTCRV does not draw a dot at the initial point. To place a dot there, use the sequence:

```
SETCRV
DOTSRR [0]
LI  RCR,-N
DOTCRV  (RPTM)
```

NEWCRV

A surface patch in 3D consists of a number of related curves. The iteration of points along a curve is done by the preceding instructions, but the iteration to get from one curve to another is done by:

```
NEWCRV
```

## 8.8 Character String Interpreter Instructions

The Character String Interpreter uses directions supplied by its FONT register to decode words packed with character codes and to draw the desired characters using the hardware character generator, if possible, or software subroutines.

The Character String Interpreter contains two 36-bit registers: FONT, whose address is 1 , and CHAR, whose address is 0. CHAR is seldom directly accessed in a program.

The following instructions should be used only in 2D.

NOTE: The Character String Interpreter is also sometimes referred to as the "Character Bubble." Therefore, the following instructions use "CB" as part of the mnemonics.

### LOCB

The registers can be loaded by the instruction:

```
LOCB  REG,  ADDR (COUNT)
```

specifying that COUNT registers starting with REG are to be loaded with data beginning at ADDR. COUNT is normally 1, but it can be 2 (to load both registers). In this usage, if REG is FONT, ADDR should contain data for FONT, and ADDR+1 should contain data for CHAR. If REG is CHAR, ADDR should contain data for CHAR, and ADDR+1 should contain data for FONT .

### STCB

Storing the registers is accomplished by the instruction:

```
STCB  REG,  ADDR(COUNT)
```

specifying that COUNT registers starting with REG are to be stored at ADDR.

### SKCB

The registers may be pushed onto the data sink by using:

```
SKCB  REG, (COUNT)
```

where COUNT registers starting with REG are to be sinked.

### RTCB

After sinking, the registers may be retrieved by:

```
RTCB  REG, (COUNT)
```

specifying that COUNT registers counting backward from REG are to be retrieved from the data sink.

For example, if both registers had been sinked by:

```
SKCB  FONT, (2)
```

they could be retrieved by:

```
RTCB   CHAR, (2)
```

DOCHAR

Once the FONT register has been properly loaded, and the RCR has been initialized to the (negative of the) number of characters to be drawn, the instruction:

```
DOCHAR ADDR (RPTM)
```

can be used to draw the characters coded into words at ADDR.

This instruction does several things. It:

1. puts ADDR into the RAR;
2. moves the word now addressed by the RAR into the CHAR register;
3. increments the RAR;
4. extracts a byte from the CHAR register whose length is given by the "length" field of the FONT register and whose position (bit position of right-hand edge) is given by 35 minus the "pointer" field of the FONT register;
5. fetches the instruction whose address is the sum of this byte and the "base" field of the FONT register;
6. decreases the "pointer" field by the "length" field;
7. executes the instruction fetched (which may possibly stop this sequence);
8. increments the RCR and stops this sequence if non-negative;
9. returns to step 4 above if the "pointer" field is still non-negative;
10. resets the "pointer" field to 36 minus the "length" field if the "pointer" field is negative and returns to step 2 above.

If the LDS-1 has an active hardware character generator which recognizes the byte extracted in step 4 as one of its character codes, step 5 is skipped and step 7 is replaced by a signal to the character generator to draw the requested character.

If the RAR is already pointing to the right place, step 1 above may be skipped by using this form of the instruction:

```
DOCHAR  @(RPTM)
```

In fact, if the RAR is already correct, any other Channel Control register may be loaded, e.g.:

```
DOCHAR  RCR,-50(RPTM)
```

The idea behind the "base" field and the character codes (bytes extracted) is that somewhere in core is a dispatch table of the form:

```
BASE:     JMPPSH  CHAR0(PROGM)
          JMPPSH  CHAR1(PROGM)
            .
            .
            .
```

The address BASE should be in the "base" field and the sets of code at CHAR0, CHAR1, etc. should draw the characters whose codes are 0, 1, etc. respectively.

The reason why such instructions work is rather intricate. The JMPPSH pushes onto the stack the current PC (which is one greater than the address of the DOCHAR) marked with a change to the current mode, which is REPEAT. It also loads the PC with the address of the character-drawing subroutine and causes PROG mode to be entered, which permits the character to be drawn. A PEEL at the end of the subroutine restores the PC and returns to REPEAT mode, whence the RSR (Repeat Status Register) returns control to the Character String Interpreter.

Clearly the character subroutine should not destroy the values in the RAR, RCR, or RSR, which are needed for more decoding. The best practice is to push these registers at the beginning of each subroutine.

Consider the following example which draws the characters:

X Y X Y X

```
        LI RCR, -5                  ;FIVE CHARACTERS TO BE DRAWN
        LOCB  FONT, INFO(1)         ;LOAD FONT REGISTER
        DOCHAR WORDS(RPTM)          ;DRAW CHARACTERS
                    .
                    .
                    .
INFO:   XWD 341044, BASE            ;POINTER = 34 (OCTAL)
                                    ;LENGTH = 10 (OCTAL)
                                    ;ACCEPT POINTER, LENGTH
                                    ;DISABLE CHARACTER GENERATOR
                                    ;DISPATCH TABLE AT BASE
WORDS:  000004000020                ;BITS 0-8 : 0
                                    ;BITS 9-16 : 1
                                    ;BITS 17-24 : 0
                                    ;BITS 25-32 : 1
                                    ;BITS 33-36 IGNORED
        0                           ;BITS 0-8 : 0
                                    ;BITS 9-36 IGNORED
                    .
                    .
                    .
BASE:   JMPPSH      X(PROGM)        ;GO HERE ON CHARACTER CODE 0
        JMPPSH      Y(PROGM)        ;GO HERE ON CHARACTER CODE 1
                    .
                    .
                    .
   X:   PSH RAR,                    ;SAVE REGISTERS
        PSH RCR,
        PSH RSR,
        DRAWTR      [XWD 2,4]       ;DRAW AN X
        SETPTR      [XWD-2,0]
        DRAWTR      [XWD 2,-4]
        SETPTR      [XWD 2,0]
        PEEL                        ;RESTORE REGISTER AND PC, GO
                                    ;BACK TO REPEAT MODE
                    .
                    .
                    .
   Y:   PSH RAR,                    ;SAVE REGISTERS
        PSH RCR,
        PSH RSR,
        SETPTR      [XWD 1,0]       ;DRAW A  Y
        DRAWTR      [XWD 0,2]
        DRAWTR      [XWD-1,2]
        SETPTR      [XWD 1,-2]
        DRAWTR      [XWD 1,2]
        SETPTR      [XWD 2,-4]
        PEEL                        ;RESTORE REGISTERS AND PC, GO
                                    ;BACK TO REPEAT MODE
```

# CHAPTER 9

# PROGRAMMING EXAMPLES

This chapter is devoted entirely to examples of code for the LDS-1 system. A prior knowledge of the PDP-10 assembly language is assumed. The code shown in these examples is not guaranteed to be the optimal code for accomplishing a task, but is presented to illustrate the use of certain features of LDS-1.

## 9.1 Start Up

This example is a subroutine which initializes the system and starts the processor at a specified location. This subroutine is employed in nearly all the examples, so it behooves the reader to become familiar with the functions performed even if he doesn't care to understand it in detail. This subroutine is included in the MACRO-10 file which defines the operation codes for LDS-1.

```
                ;START UP SUBROUTINE TO INITIALIZE PROCESSOR
DP=130
DPPI=4
SINK:       BLOCK 177
STACK:      BLOCK 1
OPDEF     CONODP [CONO DP,DPPI]
INDISP:             0
                CONODP  515300
                MOVE    0,[JSR DPINTR]
                MOVEM   0,40+2*DPPI
                CONSO   DP,4000
                JRST    .-1
                DATAO   DP,[LI DIR,52405(PROGM)]
                CONSO   DP,4000
                JRST    .-1
                DATAO   DP,[LI SR,0]
                CONSO   DP,4000
                JRST    .-1
                DATAO   DP,[LOCLA SELINT,[XWD 401400,770000](1)]
                HRRZ    0,(16)
                ADD     0,[JMP 0]
                MOVEM   0,DSTQQQ+4
                CONSO   DP,4000
                JRST    .-1
                DATAO   DP,[JMP DSTQQQ]
                CONODP  400
                JRA     16,1(16)
DSTQQQ:         LI      SP,STACK
                LI      DSP,SINK
                LOCLSA  VIEW,[XWD 3777,3777](1)
                LOCLSA  WIND,[XWD 1000,1000](1)
                LOCLA   CDIR,[102521](1)
                STOP
```

The LDS-1 system is set up as unit 130 on the PDP-10 and is assigned priority interrupt level 4 by the assignment DPPI =4. A combination stack and sink are supplied. Remember that the stack is loaded toward lower addresses and the sink toward higher addresses. A new OPDEF is given for the CONO instruction. CONODP is a CONO directed toward the LDS-1 with the priority interrupt level always supplied.

The first instruction to the system is via the CONO. The system is master cleared, all interrupts turned off, and pause request issued. A JSR DPINTR is planted in the interrupt location. (DPINTR is a minimal interrupt routine which is supplied with INDISP, but not explained here). The CONSO loop causes the PDP-10 to wait until the LDS-1 system has returned to the PAUSE state. Then all conditions which might cause the system to stop are cleared. These conditions must be cleared via the DATAO instruction because the system will return to the STOP (or PAUSE) state if all stop conditions have not been cleared. Therefore, a sequence of instructions from memory would not necessarily clear all stop conditions. One must wait for the system to return to the PAUSE state before issuing another DATAO instruction. (NB, The DATAO instruction has no effect on the system if the LDS-1 is not in a PAUSE state.)

The stop conditions and means of clearing each are listed below:

| | |
|---|---|
| STOP on WCR + | Clear directive bit mask |
| STOP on HIT | Clear directive bit mask |
| PROGRAM STOP | Clear status register bit |
| Scope Selection Violation | Load proper permit and select bits (this can be done only by a DATAO) |

After clearing all stop conditions, a JMP POOH is placed at the end of the setup code located at DSTQQQ and the system is given the command to load the PC with DSTQQQ. Then the resume pulse is issued via a CONO. This causes the pause and stop flip flop to be cleared. (NB, The stop flip flop may be cleared for the execution of one instruction, but if a stop condition still exists, it will be set again. Therefore, issuing RESUME does not clear the condition, only the Stop flip flop.) The system then starts executing the code located at DSTQQQ. The first two instructions load the stack and sink pointers. The viewport is loaded to give full screen deflection and the window is loaded with the arbitrary numbers -1000, +1000 for both X and Y. Note the use of size absolute in conjunction with a 72-bit load. The planted JMP instruction is then executed which starts the user program.

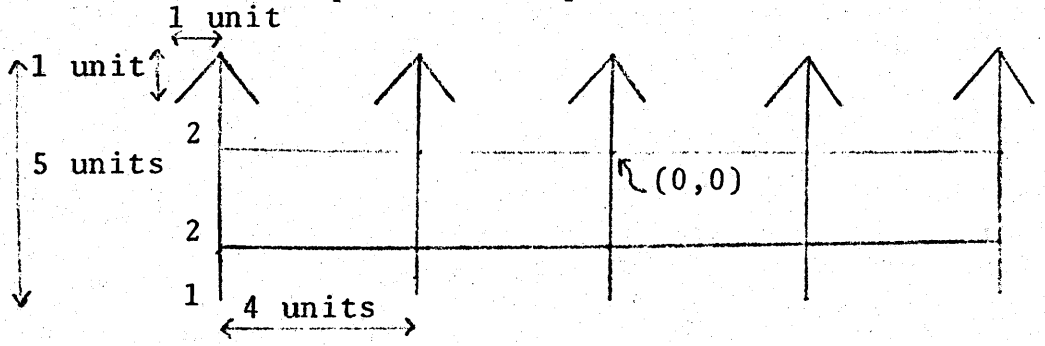
Included with the INDISP subroutine is the following macro definition:

```
DEFINE DSTART (POOH)
    <JSA 16,INDISP
     JUMP  POOH>
```

This macro will be used throughout the examples.

## 9.2 Example To Draw A 2D Picket Fence:

This example draws a picket fence as shown.




```
                LI   RCR, -4
                LINAA  CRSBRD(RPTM)   ;DRAW CROSSBOARDS
                SETPTA [XWD -10,2]    ;TOP OF FIRST PICKET
                LI RCR, -5            :FIVE PICKETS
DRAWP:          LIPSHM RCR,-3         ;THREE LINES PER PICKET
                DRAWFR  PICKET(RPTM)  ;DRAW PICKET
                PEEL                  ;RESTORE RCR
                SETPTR [XWD 4,0]      ;MOVE FOR NEXT PICKET
                IJNRCR  DRAWP         ;MORE PICKETS?
                        :
                        :
CRSBRD:         XWD -10,-2
                XWD  10,-2
                XWD  10, 0
                XWD -10, 0
PICKET:         XWD -1, -1
                XWD  0, -5
                XWD  1, -1
```

Note: The statements:

```
DRAWFR  PICKET(RPTM)
PEEL
```

Could be replaced by the equivalent:

```
DRAWFR PICKET (RPTM+PEELM)
```

## 9.3 2D Star

This example will display a star whose center is controlled via the switches and load the hit and angle counters in the console lights. The counts are cleared each time through the loop before any drawing instructions are issued. The setpoint data is controlled by the console switches. The RCR is used to control the number of THRURR type instructions executed in repeat mode. The RAR is loaded with the address of CNTR2D via the SETPTA instruction and will be incremented by SETPTA. Therefore, it is not necessary to load RAR in the THRURR instruction since the data is located immediately after CNTR2D.

```
OPDEF     THRURR  [DD+TO+SL]
TEST2D:   DSTART(SHOW2D)
          RSW     CNTR2D
          DATAO   PI,STACK
          JRST    TEST2D+2
SHOW2D:   LI      RCR,-4
          LOCLA   HITANG,[0](1)
          SETPTA  CNTR2D
          THRURR  @(RPTM)
          STCL    HITANG,STACK(1)
          JMP     SHOW2D
CNTR2D:           0
          XWD     600,600
          XWD     600,0
          XWD     600,-600
          XWD     0,-600
```

## 9.4 Repeat Mode Instruction File

This example shows processor code for a typical repeat mode instruction file. The processor is told to execute Table T1, followed by T2, followed by T3, etc. For example purposes, the three types of display tables are each different. The header for T1 shows that its data is to be interpreted as a polygon (i.e., set point, draw to, draw to, draw to, etc.) The Read Count Register is to be set to the negative number N1 which is one greater than the number of data items in the table. Notice when the Read Count Register counts up to zero, the table has been completely executed and the processor will return to the next instruction which is to execute T2.

The header of Table T2 shows that the table is to be interpreted as a list of address data. Each address specifies the location of one X,Y coordinate datum. The left half-word points to the coordinates of one end of a line, and the right half-word points to the coordinates of the other end. The Read Count Register is loaded with a negative number N2, whose magnitude is one more than the number of address pairs in the table. The figure drawn by T2 is the same polygon as in T1. It is drawn directly over the T1 polygon by drawing every other line in succession.

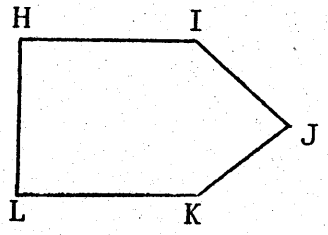
Table T3 shows the redrawing of both T1 and T2 using the push down stack.

TYPICAL REPEAT MODE INSTRUCTION FILE

```
                                        ;Display Code
        XQTA T1
        XQTA T2
        XQTA T3
        ;ETC

                                        ;T1, T2, T3 Are Three Kinds
N1=T1-L+1                               Of Display Tables To Execute
T1:     POLRR RCR,N1(RPTM)              ;Load Read Count Register
                                        With N1, Draw Polygon From
                                        Relative Coordinate Data
H:      XWD  0,1000                     ;Right And Left Half Words
                                        Separated By Comma
I:      XWD  400,0
J:      XWD  200,-500
K:      XWD -200,-500
L:      XWD -400,0

N2=T2-T2E+1
T2:     LINIRR RCR, N2(RPTM)            ;Load Read Count Register
                                        With N2, Setup For Drawing
                                        Lines From Address Pointers
                                        In Left And Right Half Words
                                        Separated By A Comma
```

```
                XWD T1+1, T1+2
                XWD T1+3, T1+4
                XWD T1+5, T1+1
T2E:            XWD T1+2, T1+3

T3:             JMPPSH .+1              ;Save PC, Setup For Table
                XQTA T1                 Of Code
                XQTA T2
                PEEL
```

The POLYGON



## 9.5 Chess Board

This example displays a chess board on the Scope. Cross hatching of the squares is controlled by program flag 0. Note the use of a processor subroutine (SQUARE). When exit from the subroutine is effected by the PEEL instruction, the RCR is also peeled since it was pushed, unmarked, when the subroutine was entered. Cross hatching is carried out in the SQUARE subroutine by testing PF0.

The RCR is used to control the column count, the row count, and the count for two repeat mode instructions. This multiple use of the RCR is made possible by correct use of the stack.

CHESSB (SHOWS CHESS BOARD)

```
;
CHESS:    DSTART(SCHESS)
          HALT
SCHESS:   LOCLA   WINDLB,DCHESS(2) ;LOAD WINDOW
          ST      (PF0)            ;SET PROGRAM FLAG 0
          SETPTA  [XWD 0,0]
          LI      RCR,-10          ;SET COLUMNS COUNT
CHESS1:   LIPSHM  RCR,-10          ;SET ROWS COUNT
CHESS2:   JMPPSH  SQUARE
          SETPTR  [XWD 0,100]      ;ADVANCE TO NEXT LINE
          IJNRCR  CHESS2           ;MORE LINES?
          SETPTR  [XWD 100,-1000]  ;ADVANCE TO NEXT COLUMN
          CM      (PF0)            ;CHANGE CROSS-HATCHING PARITY
          PEEL                     ;RETRIEVE COLUMN COUNT
          IJNRCR  CHESS1           ;MORE COLUMNS ?
          JMP     SCHESS+1         ;START ALL OVER AGAIN
;
SQUARE:   LIPSH   RCR,-5
          POLRR   TABCHS(RPTM)     ;DRAW A SQUARE
          SETPTR  @
          JIFCM   .+2(PF0)         ;CHECK CROSS-HATCHING PARITY
          PEEL
          LI      RCR,-13
          LINRR   @(RPTM)          ;CROSS-HATCHING
          PEEL
;
DCHESS:   XWD     0,0              ;WINDOW SIZE
          XWD     1000,1000
;
TABCHS:   XWD     10,10            ;DATA FOR SQUARES
          XWD     60,0
          XWD     0,60
          XWD     -60,0
          XWD     0,-60
          XWD     -10,-10
```

```
;
        XWD     10,50           ;DATA FOR CROSS-HATCHING
        XWD     20,20
        XWD     -20,-40
        XWD     40,40
        XWD     -40,-60
        XWD     60,60
        XWD     -40,-60
        XWD     40,40
        XWD     -20,-40
        XWD     20,20
        XWD     -70,-30
```

## 9.6 Boxing

This example produces two pictures, the first is in the upper left which displays a window and a line representing the diagonal of the instance. The other is in the lower right and displays that part of the instance (a transistor) within the window plus the diagonal line. The PDP-10 portion of the program changes the size and position of the window and instance.

After the code for the upper left picture has been executed, the window and viewport are changed and the instance loaded with a 72-bit load (this is necessary to test Area In Common). AIC is tested with a group 2 instruction and if the instance has no area in common with the window, control is passed to NOBOX which displays the diagonal of the instance.

If there is AIC, the window and viewport are "sinked." The BOXSA command is given with the RAR set to point to MASTER. Since the box command is given as size absolute and MASTER contains 2000,2000, the master definition space is -2000 to 2000 in X and Y. The box command causes the viewport and window to be changed to reflect the mapping directly from master space to scope coordinates. The commands are then issued to draw the transistor. The old viewport and window are then retrieved from the data stack.

```
BOX2D:    DSTART(SUB2D)
BXLOOP:   RSW     14
          MOVEI   10,WS
          JSA     16,FIX
          HRRZ    15,WS
          LSH     15,1
          HRRM    15,WS+3
          XOR     15,[XWD 0,-1]
          HRRM    15,WS+1
          HLRZ    15,WS
          LSH     15,1
          HRLM    15,WS+4
          XOR     15,[XWD 0,-1]
          HRLM    15,WS+2
          LSH     14,-6
          MOVEI   10,WC
          JSA     16,FIX
          LSH     14,-6
          MOVEI   10,LSZ
          JSA     16,FIX
          LSH     14,-6
          MOVEI   10,LC
          JSA     16,FIX
          MOVEI   15,100
          SOJGE   15, .
          JRST    BXLOOP

;
;
;
FIX:      0
          TRNN    14,1
          JRST    XM
          HRLZI   15,1
          ADDM    15,0(10)
XM:       TRNN    14,2
          JRST    YP
          HRLZI   15,-1
          ADDM    15,0(10)
YP:       TRNN    14,4
          JRST    YM
          HRRZI   15,1
          ADD     15,0(10)
          AND     15,[XWD 0,-1]
          HRRM    15,0(10)
YM:       TRNN    14,10
          JRA     16,0(16)
          HRRZI   15,-1
          ADD     15,0(10)
          AND     15,[XWD 0,-1]
          HRRM    15,0(10)
          JRA     16,0(16)
```

```
;
OPDEF   LISR[DD+TO+SL]
;
SUB2D:  LOCLA   VIEWLB,VIEW11(4)
        SETPTA  WC
        LI      RCR,-5
        POLRR   @(RPTM)
        SETPTA  LC
        LISR    @0
        SETPTA  WC
        LOCLSR  @WIND,(1)
        LOCLA   VIEWLB,VIEW21(2)
        SETPTA  LC
        LOCLSR  @INST,(1)
        JIF     @NOBOX(AICF)
        SKCL    VIEWLB,(4)
        BOXSA   MASTER
        SETPTA  @
        LI      RCR,-20
        LINRR   @(RPTM)
        RTCLA   WINDRT,(4)
NOBOX:  SETPTA  LC
        LISR    @0
        JMP     SUB2D

MASTER: XWD     2000,2000
        XWD     -2000,0
        XWD     0,0
        XWD     2000,0
        XWD     0,1000
        XWD     0,-2000
        XWD     0,1400
        XWD     1000,400
        XWD     0,0
        XWD     0,1000
        XWD     0,-4000
        XWD     0,1000
        XWD     0,0
        XWD     -1000,400
        XWD     0,0
        XWD     100,0
        XWD     -100,0
        XWD     70,-70
        BLOCK   20
VIEW11: XWD     -3777,0
        XWD     0,3777
        XWD     400000,400000
        XWD     377777,377777
VIEW21: XWD     0,-3777
        XWD     3777,0
WC:     XWD     0,0
WS:     XWD     4000,4000
        XWD     0,-100000
        XWD     -100000,0
        XWD     0,100000
        XWD     100000,0
LC:     XWD     0,0
LSZ:    XWD     4000,4000
```

## 9.7 Subroutining with a New Box

This example shows how the Save Register in the Clipper can be used to see if there is any Area In Common Between the material about to be drawn and the material shown on the scope face. If there is area in common, the correct material is shown; if there is no area in common, the subroutine is skipped completely. The essence of this routine is the BOX instruction.

The BOX instruction forms a composite transformation from two linear transformations and places the parameters of the composite transformation in the window and viewport registers. The two transformations combined by the BOX instruction are each specified by two rectangular areas. One transformation is the former window-to-viewport transformation. The other transformation is the transformation from the rectangle of definition space specified during the BOX instruction itself into the rectangle of page space defined in the instance register.

Using the boxing instruction requires the following steps:

1. In the main routine, establish the part of page space to be occupied by the symbol. This is done by loading the instance register. There are two methods shown. Loading the instance register clears the AIC bit, and resets it if the symbol overlays the window.

2. In the subroutine test to see if there was an overlap between the old window and the instance space specified; and if there was none, go to the place where done.

3. Save the old window-to-viewport transformation by sinking the window and viewport registers.

4. Establish the portion of definition space to be seen. The BOX instruction computes the composite transformation, if any, and puts it into the window and viewport registers.

5. If there was overlap, display the subroutine content, transforming information from definition space directly onto the scope. The new transformation (a composite of the old window-to-viewport and definition - to - symbol transformation) applies. Coordinates are now treated in definition space.

Repeat steps 1 to 5 recursively if desired.

Typically operation 1 will be done in the main body of code prior to calling the symbol subroutine. Operations 2 through 6 will be done in the symbol subroutine. Thus, the main body of code needs to know only the position and size of the desired symbol and does not need to know how much of its definition space is used.

This process of calling subroutines with a new transformation is too costly in time and memory space to use for subroutines producing very simple fixed-size symbols such as single printed characters. Because it expands a symbol only if some portion of it is potentially visible, however, it is worthwhile to use it for relatively complex subroutines or subroutines for producing variable sized symbols. For example, each line of text displayed should certainly be called as a subroutine which makes the BOX test. The individual character calls with the line of text, however, would use simple subroutine calls to the character subroutines.

The character subroutines should use relative specification exclusively, so that the advance in position of successive characters in the line is provided by the information stored in the Save register.

As a note to the subroutine, the following reviews the four numerical registers in the clipping divider called the Window, Viewport, Save, and Instance registers. Each of these registers stores a four-component vector.

In the Window, Viewport and Instance registers, these four components deal with the left, right, bottom and top of a rectangular area defined by the register. The Window specifies the region of the page coordinate space to be shown. The Viewport register specifies the portion of the scope in which the material is to appear. The Instance register specifies the portion of page space to be occupied by a symbol.

In the Save register these four components specify a single point in two or three dimensions. In two dimensions, the Save register contains duplicate information in its first and last two components, XXYY. (In three dimensions the Save register contains XZYZ, but in the descriptions that follow assume only two-dimensional use.)

The window, viewport and instance registers can be addressed by the program in two ways:

1. Individually, as two-component halves.
2. Collectively, as full four-component vectors.

The two-component halves are called (WINDLB, WINDRT, VIEWLB, VIEWRT, INSTLB, INSTRT) where LB suffix indicates the left and bottom sections, and RT suffix indicates the right and top sections. Thus a specification of LOCLA WINDLB, [X1,Y1](1) will put X1 and Y1 into the left and bottom portions of the window register respectively.

The Window, Viewport and Instance registers can also be addressed using a relative, rather than an absolute mode as above. A relative specification of the form LOCLR WINDLB(1 will load two portions of the Window register with information derived by adding the new data from memory to the content of the Save register.

A four-component load specification obtains only two components fresh from memory. The other two components are made from the data remaining in the Save register. The Save register always contains the data that was last addressed. A full four-component load of the Window, Viewport, or Instance registers (or Save register itself) use the new data and the data in the Save register just as this data would be used to draw a line. The two ends of this virtual (invisible) line supply the four components required.

Suppose, for example, that the Save register contains [X1,X1,Y1,Y1]. A relative specification which loads the full window register might look like this:

LOCLR WIND, [dX,dY](1)

Where [dX,dY] is the address of a register storing dX,dY in its left and right halves. Such an instruction would generate a new endpoint, X2 Y2, just as would be generated to draw a line. The numbers placed in the window register then would be:

[X1,X2,Y1,Y2] where X2 = X1 + dX
Y2 = Y1 + dY

This full window load can be thought of as drawing a virtual (invisible) line from the lower left to the upper right of the rectangle desired.

Subroutining With New Box

```
                                ;In The Calling Routine

MAIN:   SETPTA LL               ;Establish Lower Left Of Symbol
        LOCLSR INST,SZ(1)       ;Put Symbol Data In Instance Register;
                                This Sets AIC Bit If The Symbol
                                Overlays The Window.  Note: Clipper
                                Save Register Supplies The Other
                                Coordinate.
        JMPPSH SUBR
        ;ETC

LL:     XWD XL,YB               ;Left Bottom Of Symbol Space
SZ:     XWD XR-XL,YT-YB         ;Size Of Symbol

MAIN:   SETPTA SY               ;OR Alternative Method
        LOCLA INST,SY+1(1)      ;Establish Symbol Space By Loading
                                Instance--Left Bottom--To Completion--
                                Right Top
        JMPPSH SUBR
        ;ETC

SY:     XWD XL,YB               ;Bottom Left Of Symbol
        XWD XR,YT               ;Top Right Of Symbol

SUBR:   JIF @DONE(AICF)         ;The Subroutine Called
        SKCL VIEWLB,(4)         ;Test The AIC Bit
                                ;Save In The Next Four Words Of The
                                Data Sink The Old Contents Of The
                                Window And Viewport
        SETPTA DEF              ;Set Bottom Left Of Definition Space
        BOXA DEF+1              ;Pick Up Top Right Of Definition Space
                                ;At This Point The Window And Viewport
                                Registers Have Been Setup With Their
                                New Values.  See Following
                                Explanation
        XQTA T1                 ;Draw The Symbol
        DRAWTR P
        ;ETC
        DRAWTR Q

DONE:   RTCLA WINDRT,(4)        ;Restore Window And Viewport From
                                The Stack
        PEEL

DEF:    XWD XL,YB               ;Desired part Of Definition Space To
                                See
        XWD XR,YT
```

## 9.8 3D Stars In Space

This, example draws five three-dimensional stars in locations specified by table STAR1.

```
OPDEF     LISR    [DD+TO+SL]      ;LINES SIZE RELATIVE
          LI      RCR,-5          ;5 STARS
          SETPTA  STAR1           ;POSITION FOR FIRST STAR
DRAWST:   PSHM    RAR             ;SAVE POINTER FOR STARS
          LIPSH   RCR,-10         ;10 (OCTAL) POINTS PER STAR
          LISR    STAR(RPTM)      ;DRAW A STAR
          PEEL                    ;RESTORE RCR, RAR
          SETPTA  @               ;POSITION FOR NEXT STAR
          IJNRCR  DRAWST          ;MORE STARS?
                .
                .
                .
STAR1:    XWD      100,200        ;STAR 1: X=100, Y=200
          XWD      300,300        ;      Z=300
          XWD       40,-70        ;STAR 2: X=40, Y=-70
          XWD      120,120        ;      Z=120
          XWD      500,0          ;STAR 3: X=500, Y=0
          XWD      600,600        ;      Z=600
          XWD        0,0          ;STAR 4: X=0, Y=0
          XWD       20,20         ;      Z=20
          XWD     -300,-100       ;STAR 5: X=-300, Y=-100
          XWD      400,400        ;      Z=400
STAR:     XWD        4,4          ;(4,4,4) to (-4,-4,-4)
          XWD        4,4
          XWD       -4,4          ;(-4,4,4) to (4,-4,-4)
          XWD        4,4
          XWD       -4,4          ;(-4,-4,4) to (4,4,-4)
          XWD        4,4
          XWD        4,-4         ;(4,-4,4) to (-4,4,-4)
          XWD        4,4
```

## 9.9 3D Picture

This example will display a set of parallel squares equally spaced along the Z axis. The X and Y starting point for the first square and the number of squares are controlled by the switches. Note that the directive must be loaded for 3D operation, and ZTOS must be selected in addition to OUTPUT TO SCOPE. The RCR is used for two purposes in this example. First, it is loaded with the negative of the number of squares requested. The RCR is then Pushed Marked and loaded with -4 to control repeat mode. Note that in 3D it is necessary to access two words for each data point, but the RCR refers to data points, not PDP-10 words. After the square is drawn with the DRAWTR instruction, a SETPTR is issued (without reloading the RAR) which moves Z out 40 units and goes into PEEL mode. PEEL mode causes the stacked value of the RCR to be unstacked and, since the PUSH instruction was marked, the mode is changed back to PROG. The IJNRCR causes the RCR to be incremented and the PC to be loaded with CNT3S+1 until the RCR goes to -1. (The RCR is tested before it is incremented.)

```
TEST3S:             DSTART(SHOW3S)
                RSW     0
                AND     0,[77]
                MOVN    0,0
                HRRM    0,CNT3S
                RSW     INPT3S
                BITSM(INPT3S,[0],[16],X3S)
                BITSM(INPT3S,[17],[35],Y3S)
                COMBIN(X3S,Y3S,XY3S)
                JRST    TEST3S+2

;
X3S:            0
Y3S:            0
INPT3S:         0
;
SHOW3S:         LI      DIR,4000
                SETPTA  PNT3S
CNT3S:          LI      RCR,-2
                LIPSHM  RCR,-4
                DRAWTR  TAB3S(RPTM)
                SETPTR  @(PEELM)
                IJNRCR  CNT3S+1
                JMP     SHOW3S+1
;
PNT3S:          XWD     400,400
                XWD     400,400
TAB3S           XWD     -1000,0
                XWD     0,0
                XWD     0,-1000
                XWD     0,0
                XWD     1000,0
                XWD     0,0
                XWD     0,1000
                XWD     0,0
                XWD     0,0
                XWD     40,40
;
                ;BITSM AND BITS SUBROUTINES
DEFINE BITSM(A1,A2,A3,A4)
<JSA 16,BITS
                LI      A1
                LI      A2
                LI      A3
                MOVEM   0,A4>
;
                ENTRY BITS      ; BITS(X,M,N) GETS BITS M-N OF X
                0
BITS:           0
                MOVEM   1,BITS-1
                MOVE    0,@0(16)    ; X TO AC0
                MOVE    1,@1(16)    ; LSH AC0 M BITS
```

```
LSH     0,0(1)
MOVE    1,@2(16)      ;AC1 = N
SUB     1,@1(16)      ;AC1 = N-M
SUBI    1,↑D35        ;AC1 = N-M-35
ASH     0,0(1)        ;RIGHT LSH (35-M+N) BITS
MOVE    1,BITS-1
JRA     16,3(16)
```

## 9.10 Moving Cart

The following example shows use of the Matrix Multiplier. It depicts a simple-minded cart driving around in a circle with its wheels turning.

```
STRT:     MOVE 0,RW
          FDVR 0,RC
          MOVEM 0,RATIO   ;RATIO = RW/RC
          DSTART (BEGIN)  ;START LDS-1 AT BEGIN
          SETZM THETAW    ;INITIALIZE

INCRMT:   MOVE 0,DELTAW
          FADRB 0,THETAW  ;INCREMENT THETAW
          FMPR 0,RATIO
          MOVEM 0,THETAC  ;INCREMENT THETAC

{ SINW = SIN(THETAW)   APPROPRIATE ROUTINES SHOULD BE
  COSW = COS(THETAW)   CALLED IN HERE TO COMPUTE SINES,
  SINC = SIN(THETAC)   COSINES, ETC.  IN FIXED-POINT
  COSC = COS(THETAC)   SIGNED FRACTIONS.                }

             ; UPDATE MATRICES
          MOVE 0,SINW
          HRLM 0,WHLMAT+3
          HRRM 0,WHLMAT+3
          MOVNS 0
          HRRM 0,WHLMAT+4
          MOVE 0,COSW
          HRRM 0,WHLMAT+2
          HRLM 0,WHLMAT+5
          HRRM 0,WHLMAT+5
          MOVE 0,SINC
          HRLM 0,CARMAT+1
          HRRM 0,CARMAT+1
          MOVNS 0
          HRLM 0,CARMAT+4
          MOVE 0,COSC
          HRLM 0,CARMAT
          HRLM 0,CARMAT+5
          HRRM 0,CARMAT+5
          JRST INCRMT
             ; CONSTANTS
RW:       6.0               ;RADIUS OF WHEELS
RC:       192.0             ;RADIUS OF TURN (300 OCTAL)
RATIO:    0
THETAW:   0                 ;ANGLE WHEELS HAVE TURNED
THETAC:   0                 ;ANGLE CART HAS TURNED
DELTAW:   .001              ;UNIT OF WHEEL ROTATION
          ; LDS-1 CODE
BEGIN:    LI DIR,104000     ; 3D, MM ACTIVE
          LOMDIR MD         ;LOAD MM DIRECTIVE
DRACAR:   LOMM 0,CARMAT(4)  ;PUT CART MATRIX INTO A
          LI RCR, -12       ;DRAW CART
          POLAA CART(RPTM)  ;    .
```

```
            LI RCR, -6          ;    .
            LINAA @ (RPTM)      ;    .
            PUSHMM 4, (4)       ;MOVE A INTO B
            LOMMP 4, WHLMAT(4);MOVE PRODUCT OF WHEEL 1
                    ;MATRIX AND B INTO A
            JMPPSH WHEEL        ;DRAW A WHEEL
            LOMMP 7, WHEEL2(1);RELOAD TRANSLATION
                    ;PART FOR WHEEL 2
            JMPPSH WHEEL        ;DRAW A SECOND WHEEL
            LOMMP 7,WHEEL3(1) ;RELOAD FOR WHEEL 3
            JMPPSH WHEEL        ;DRAW WHEEL 3
            LOMMP 7, WHEEL4(1);RELOAD FOR WHEEL 4
            JMPPSH WHEEL        ;DRAW WHEEL 4
            JMP  DRACAR         ;GO DO IT AGAIN
WHEEL:      LI RCR, -4          ;DRAW A WHEEL
            LINAA WHL (RPTM)    ;CONSISTING OF
            PEEL                ;FOUR SPOKES
                    ;MATRIX MULTIPLIER DIRECTIVE
MD:         XWD 11020,0         ;MOC ON, ALL ELSE OFF
                    ;MATRIX FOR CART
CARMAT:     XWD 0, 0            ;COS(THETAC), 0
            XWD 0, 0            ;SIN(THETAC), SIN(THETAC)
            XWD 0, 1            ; 0, 1
            XWD 0, 0            ; 0, 0
            XWD 0, 0            ;-SIN(THETAC), 0
            XWD 0, 0            ;COS(THETAC), COS(THETAC)
            XWD 0, -140         ;TX, TY
            XWD 1000, 1000      ;TZ, TZ
                    ;MATRIX FOR WHEELS
WHLMAT:     XWD 1, 0            ; 1, 0
            XWD 0, 0            ; 0, 0
            XWD 0, 0            ; 0, COS(THETAW)
            XWD 0, 0            ;SIN(THETAW), SIN(THETAW)
            XWD 0, 0            ; 0, -SIN(THETAW)
            XWD 0, 0            ;COS(THETAW), COS(THETAW)
            XWD 300, 0          ;TX, TY (WHEEL 1)
            XWD 0, 0            ;TZ, TZ (WHEEL 1)
                    ;TRANSLATION PART FOR WHEEL 2
WHEEL 2:    XWD 300, 0          ;TX, TY
            XWD 100, 100        ;TZ, TZ
                    ;TRANSLATION PART FOR WHEEL 3
WHEEL 3:    XWD 340, 0          ;TX, TY
            XWD 100, 100        ;TZ, TZ
                    ;TRANSLATION PART FOR WHEEL 4
WHEEL 4:    XWD 340, 0          ;TX, TY
            XWD 0, 0            ;TZ, TZ
                    ;CART, THIS PART DRAWN BY POLAA
CART:       XWD 300, 0          ;X, Y
            XWD   0, 1          ;Z, W
            XWD 300, 0          ; etc.
            XWD 100, 1
            XWD 340, 0
            XWD 100, 1
            XWD 340, 0
            XWD   0, 1
```

```
                XWD 300, 0
                XWD   0, 1
                XWD 300, 30
                XWD   0, 1
                XWD 300, 30
                XWD 100, 1
                XWD 340, 30
                XWD 100, 1
                XWD 340, 30
                XWD   0, 1
                XWD 300, 30
                XWD   0, 1
                          ; THIS PART DRAWN BY LINAA
                XWD 300, 30
                XWD 100, 1
                XWD 300, 0
                XWD 100, 1
                XWD 340, 30
                XWD 100, 1
                XWD 340, 0
                XWD 100, 1
                XWD 340, 30
                XWD   0, 1
                XWD 340, 30
                XWD   0, 1
                XWD 340, 0
                XWD   0, 1
                          ; WHEEL
WHL:            XWD   0, 6
                XWD   0, 1
                XWD   0, -6
                XWD   0, 1
                XWD   0, 0
                XWD   6, 1
                XWD   0, 0
                XWD  -6, 1
                END STRT
```

The program continually updates the angle through which each wheel has turned, and redefines accordingly the angle through which the cart has turned. It then finds the sine and cosine of each angle and updates the matrices for each.

The matrix for the cart has the form:

$$\begin{bmatrix} \cos\theta_c & 0 & \sin\theta_c & \sin\theta_c \\ 0 & 1 & 0 & 0 \\ -\sin\theta_c & 0 & \cos\theta_c & \cos\theta_c \\ TX & TY & TZ & TZ \end{bmatrix}$$

This matrix represents a rotation through angle $\theta_c$ in the X-Z plane and a translation by TX, TY, and TZ.

The matrix for the wheels is defined in the coordinate system of the cart. It has the form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_W & \sin\theta_W & \sin\theta_W \\ 0 & -\sin\theta_W & \cos\theta_W & \cos\theta_W \\ TX & TY & TZ & TZ \end{bmatrix}$$

representing a rotation through angle $\theta_W$ in the Y-Z plane and translations of TX, TY, and TZ. Since the four wheels differ only by a translation, this matrix serves for each with only the last row changing.

Notice that in both matrices the fourth column is a duplicate of the third. This is so that incoming vectors of the form:

$$[X \quad Y \quad Z \quad W]$$

will be output in the form the Clipper expects:

$$[X \quad Y \quad Z \quad Z] \ .$$

The LDS-1 begins by initializing the directive registers of the Channel Control and Matrix Multiplier.

It then loads Matrix A with the matrix for the cart, which moves it 1000 units deep into the screen and down 140 units, and rotates it by the angle $\theta_C$ in a horizontal plane from its initial position. The data defining the cart is then passed through Matrix A, coming in the form:

$$[X \quad Y \quad Z \quad 1]$$

and coming out in the form:

$$[X' \quad Y' \quad Z' \quad Z']$$

where: $X' = X \cos\theta_C - Z \sin\theta_C + TX$

$Y' = Y + TY$

$Z' = X \sin\theta_C + Z \cos\theta_C + TZ$

Then matrix A is copied into matrix B to prepare for the following step, which is multiplication of the matrix for the first wheel by the matrix for the cart, with the result going into matrix A. The data for the first wheel is then passed through A, which is equivalent to passing it first through the matrix for the wheel (which aligns the wheel properly on the cart) and then passing the result through the matrix for the cart, moving the wheel just as the cart has moved. This equivalence is easily seen by:

$$V \cdot (M_W \cdot M_C) = (V \cdot M_W) \cdot M_C$$

Next matrix A is set up for the second wheel, but since the second wheel differs from the first only by a translation, only the last row of matrix A need be updated. This is done by multiplying a new translation row by matrix B and leaving the result in the fourth row of A. The wheel is then drawn again, and it is translated correctly.

The third and fourth wheels are then drawn in similar fashion, and the whole process is repeated.

## 9.11 3D Surfaces Example

A family of the curves generated in 3D curve mode can be used to portray a surface patch. Execution of a NEWCRV instruction causes the matrix multiplier to iterate difference equations just like those performed along the columns in drawing curves, but instead along all 16 rods of the array. Thus, the array is the basic representation of the surface to be drawn, and the cubic difference equations are iterated in the columns of matrix A in order to draw each curve segment, and are iterated in the rods in order to permute A so that it draws a family of curves. If the transpose map is loaded with T=10, the rods and columns will be interchanged so as to draw the same surface patch in the other direction (criss-cross style). A program to draw surface patches is illustrated below (using LDS-1/PDP-10 assembly language code):

```
;
BEGIN:    LI       DIR,2000            ;2D DIRECTIVE
          LOMDIR   ,D1                 ;LOAD MDIR: T=00,MOC,CURVE
          LOCLA    VIEWLB,VIEWPT(2)    ;LOAD CLIPPER VIEWPORT
          LI       DIR,104000          ;MM ACTIVE,3D

;
DRAW:     LOMM     0,ARRAY(0)          ;LOAD ARRAY INTO MM
          LI       RCR,-100            ;#CURVES/FAMILY
          SKMM     0,(4)               ;SINK MATRIX A
          LIPSHM   RCR,-41             ;# LINES/CURVE +1
          POLCRV   (RPTM + PEELM)      ;DRAW CURVE, PEEL COUNT
          RTMM     3,(4)               ;RETRIEVE MATRIX A
          NEWCRV                       ;DO ITERATION ALONG RODS
          IJNRCR   .-5                 ;LOOP END TEST

;
; NOW DRAW OTHER WAY
;
          LOMM     0,ARRAY(14)         ;RELOAD FIRST 3 MATRICES
          LOMDIR   D2                  ;LOAD T=10 INTO MDIR
          LI       RCR,-40
          SKMM     0,(4)
          LIPSHM   RCR,-101
          POLCRV   (RPTM + PEELM)
          RTMM     3,(4)
          NEWCRV
          IJNRCR   .-5
          JMP      DRAW

;
;
D1:       001045010000                 ;FIRST DIRECTIVE FOR MM
D2:       0                            ;BECAUSE 3D DATA FETCH
          000060000000                 ;SECOND DIRECTIVE FOR MM,
                                            T=10
VIEWPT:   XWD -3777,-3777              ;CLIPPER VIEWPORT
          XWD 3777,3777
```

The above program requires 1,318 memory cycles, and draws 4,096 lines on the scope. If all the lines appear on the scope, this picture can be refreshed at about 30 frames/second. Very acceptable patches can be portrayed with far smaller counts.

## 9.12 Timesharing

This example program swaps two users on the same scope. The switches control the time that each user has the scope. The console lights show which user has the scope by displaying one or two.

Subroutine WAIT is the timeout subroutines.

SAVV is the subroutine to save a user in an area specified in the calling sequence. First a pause request is issued, then, via a DATAO, a new stack is formed and the status register and PC are both pushed marked. Then the processor is given control to execute the code at SAVV4 which saves all other registers plus the registers of the clipper. Note that the data stack pointer is pushed marked and then pushed marked again after saving the clipper registers.

RESTOR is the routine to restore a user from a saved area. First the status register is cleared to clear program stop which was set at the end of the SAVV routine. Then the stack pointer is set to the top of the stack for the saved user and PEEL mode entered which causes the data stack pointer to be unstacked (the DSP was the last thing pushed in SAVV and it was pushed marked). Then all registers of the clipper are retrieved via the restored DSP. PEEL mode is entered again which restores all registers but the PC, SR and SP. These three registers get restored via DATAO instruction and the resume is issued.

```
TESTSW:  DSTART(USER1)
         JSA     16,WAIT
         JSA     16,SAVV
         LI      SAVVU1
         DSTART(USER 2)
LOOPSW:  DATAO   PI,[2]
         JSA     16,WAIT
         JSA     16,SAVV
         LI      SAVVU2
         JSA     16,RESTOR
         LI      SAVVU1
         DATAO   PI,[1]
         JSA     16,WAIT
         JSA     16,SAVV
         LI      SAVVU1
         JSA     16,RESTOR
         LI      SAVVU2
         JRST    LOOPSW
WAIT:    0
         RSW
         AND     0,[7777]
WAIT1:   CONO    APR,1000
         CONSO   APR,1000
         JRST    .-1
         SOSLE
         JRST    WAIT1
         JRA     16,(16)

USER1:   LOCLA   VIEWLB,VWU1(4)
         LI      RCR,-5
         POLAR   TABSW(RPTM)
         JMP     USER1+1
USER2:   LOCLA   VIEWLB,VWU2(4)
         LI      RCR,-5
         POLAR   TABSW(RPTM)
         JMP     USER2+1
TABSW:   XWD     100,100
         XWD     -200,0
         XWD     0,-200
         XWD     200,0
         XWD     0,200
VWU1:    XWD     -3777,0
         XWD     3777,3777
         XWD     -200,-200
         XWD     200,200
VWU2:    XWD     -3777,-3777
         XWD     3777,0
         XWD     -100,-100
         XWD     100,100
         BLOCK   14
SAVVU1:  BLOCK   14
         BLOCK   14
SAVVU2:  BLOCK   14
```

```
;
; A SAVED-USER-TABLE ('USERSV')
;

;      -14  LI   DSP,(PROGM) ; AFTER SINKING THE CD
;      -13  LI   RSR,
;      -12  LI   DIR,
;      -11  LI   WCR,
;      -10  LI   RCR,
;       -7  LI   UR,



;       -6  LI   RAR,
;       -5  LI   WAR,
;       -4  LI   DSP,(PROGM)
;       -3  LI   PC,(PROGM)
;       -2  LI   SR,(PROGM)
;       -1  LI   SP,(PROGM)
; USERSV:  BLOCK 14 ; FOR THE CD
;
;
DEFINE IOMBR(A1)
[DATAO DP,A1
       CONSO   DP,4000
       JRST    .-1]

SAVV:  0
       MOVE    0,@0(16)
       HRRM    0,SAVV1
       HRRM    0,SAVV4
       CONO    DP,4000
       CONSO   DP,4000
       JRST    .-1
       IOMBR(SAVV1)
       IOMBR(SAVV2)
       IOMBR(SAVV3)
       CONO    DP,40000 ; RESUME
       CONSO   DP,40000
       JRST    .-1
       JRA     16,(16)
SAVV1: NWSTKM  0        ; 'USERSV' PLANTED HERE
SAVV2: LIPSHM  SR,0
SAVV3: JMPPSH  SAVV4
SAVV4: LIPSHM  DSP,0    ; 'USERSV' PLANTED HERE
       PSH     WAR,
       PSH     RAR,
       PSH     UR,
       PSH     RCR,
       PSH     WCR,
       PSH     DIR,
       PSH     RSR,
       SKCL    SAVELB,(14)
       PSHM    DSP,
       STOP
```

```
RESTOR: 0
        MOVE    0,@0(16)
        SUBI    0,14
        HRRM    0,RESTR1
        CONO    DP,4000
        CONSO   DP,4000
        JRST    .-1
        IOMBR(RESTR2)
        IOMBR([JMP RESTR1])
        CONO    DP,40000
        CONSO   DP,40000
        JRST    .-1
        CONO    DP,4000
        CONSO   DP,4000
        JRST    .-1
        IOMBR([PEEL])
        IOMBR([PEEL])
        IOMBR([PEEL])
        CONO    DP,40000
        JRA     16,(16)
RESTR1: LI      SP,0(PEELM)     ; 'USERSV'-14 PLANTED HERE
        RTCLA   SELINT,(14)
        PEEL
        STOP
RESTR2: LI      SR,0
;
```

## 9.13 Self Mode

This example demonstrates the plotting (SELF) mode. Two graphs are plotted; one in the upper left of the screen, and one in the lower right.

The upper left display uses SELFX; the lower right display uses SELFY and DO TWICE. SELFX is not given until after the viewport, window, and instance have been loaded and the starting point has been set. Then the DOTSRR instruction is used within three loops controlled by the RCR. After drawing the graph, SELFX is cleared. When SELFX is being used, the X data is always 10, since 10 is loaded into the X portion of the instance register.

The lower right graph uses SELFY and DO TWICE. The instance is loaded with -40 in the Y portion for the SELFY data. This time both an X and Y are given in the DOTSRR instruction. Normally, the Y would be ignored, but since DO TWICE is set, the normal X data (20 or -20) is used for X, then the normal Y data (60 or -60) is used for X.

```
;
DOTSLF:   DSTART(SHOWSF)
          HALF .
;
SHOWSF:   LI       DIR,2400            ; 2D, CLEAR DO TWICE
          LOCLA    VIEWLB,VIEW1(6)     ; LOAD VIEWPORT,WINDOW,
          SETPTA   [XWD 100,4000]      ; INSTACE
          LOCLA    CDIR,[ 5 ](1)       ; SELF X
          LI       RCR,-140
          DOTSRR   [XWD 0,10]
          IJNRCR   .-1
          LI       RCR.-300
          DOTSRR   [XWD 0,-10]
          IJNRCR   .-1
          LI       RCR.-140
          DOTSRR   [XWD 0,10]
          IJNRCR   .-1
          LOCLA    CDIR,[ 1 ](1)       ; CLEAR SELF
          LOCLA    VIEWLB,VIEW2(6)
          SETPTA   [XWD 4000,10000]
          LI       DIR,1000            ; DO TWICE
          LOCLA    CDIR,[ 3 ]          ;SELF Y
          LI       RCR,-20
          DOTSRR   [XWD 20,60]
          IJNRCR   .-1
          LI       RCR,-40
          DOTSRR   [XWD -20,-60]
          IJNRCR   .-1
          LI       RCR,-20
          DOTSRR   [XWD 20,60]
          IJNRCR   .-1
          JMP      SHOWSF
;
;
VIEW1:    XWD      -3777,0             ; VIEWPORT
          XWD      0,3777
          XWD      0,0                 ; WINDOW
          XWD      10000,10000
          XWD      10,0                ; INSTANCE (DUPLICATE IN-
          XWD      10,0                ; FORMATION IN BOTH HALVES)
VIEW2:    XWD      0,-3777
          XWD      3777,0
          XWD      0,0
          XWD      10000,10000
          XWD      0,-40
          XWD      0,-40
```

# CHAPTER 10

# MEMORY PROTECTION AND RELOCATION

## 10.1 General

The memory protection and relocation facilities provided for an LDS-1 which is interfaced to a PDP-10 operate in much the same manner as those provided by the CPU of the PDP-10. Two areas of memory are provided for the user's program. These areas are referred to as "low" memory and "high" memory and are mapped separately by the relocation and protection hardware of the LDS-1 as shown in figure 10.1. Low memory begins at location 0 (except that the first $17_8$ locations are not mapped) and high memory begins at location $400000_8$. The low relocation address is used to map memory addresses which lie between $20_8$ and the limit set by the "low bounds". The high relocation address is used to map addresses which lie between $400000_8$ (except when low bounds exceed $400000_8$ in which case the high relocation takes effect at the value of the low bounds plus 1777+1) and the limit set by the high bounds.

## 10.2 Memory Relocation Register

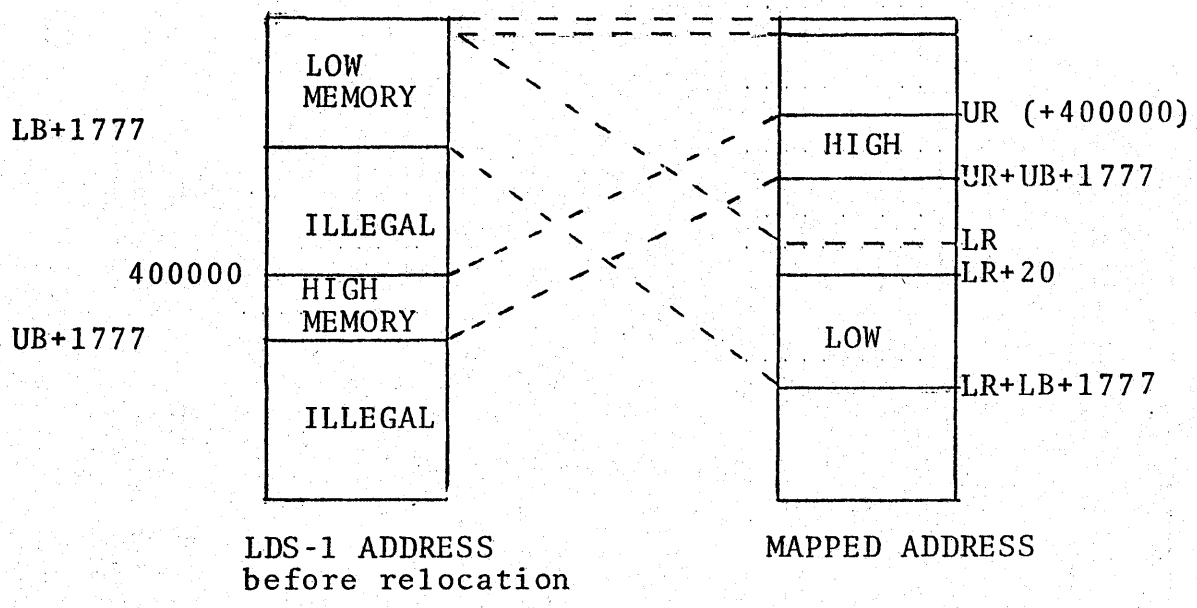
Four 8-bit registers and one control bit are used to implement the memory protection and relocation function as indicated in figure 10.1. The low relocation register contains the most significant 8 bits of the relocation address. The low bounds register specifies the number of $1024_{10}$ word blocks which can be accessed by the lower portion of the user's program. If the most significant 8 bits of the memory address are within the limit set by the lower bounds, they are added to the contents of the low relocation register; and the result is used as the most significant 8 bits of the mapped memory address.

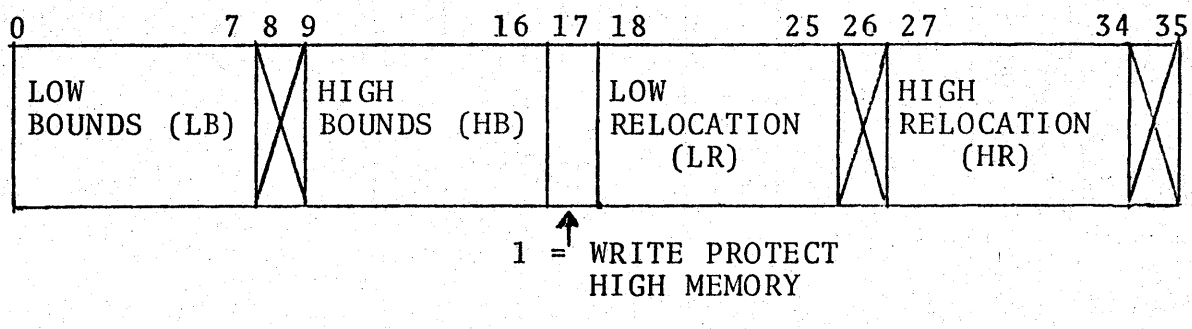
The high relocation register contains the most significant 8 bits of the high relocation address plus $400000_8$ (using wrap-around modular arithmetic). For example, if the high memory addresses were to be mapped into an area in memory beginning with $100000_8$, the high relocation register would contain the most significant 8 bits of $500000_8$. The high bounds register specifies the number of $1024_{10}$ blocks (+ the high 8 bits of $400000_8$) which are accessible to the high memory portion of the user's program. High memory can be write-protected by setting the write protect bit (see figure 10.1) so that only read cycles are allowed.

## 10.3 Memory Violation

If the memory address is out of the bounds for both low and high memory, the memory cycle is not started, and the

MEMORY PROTECTION AND RELOCATION




EXAMPLE OPERATION

| Word sent by DATAO 134 | 004 407 010 500 |
|---|---|
| (1) 000014 | 000014 |
| (2) 002036 | 012036 |
| (3) 403764 | 103764 |
| | (Read cycle only) |
| (4) 006000 | Memory Violation |
| (5) 410235 | Memory Violation |
| ↑ LDS-1 ADDRESS | ↑ MAPPED ADDRESS |

EXAMPLE MEMORY MAP

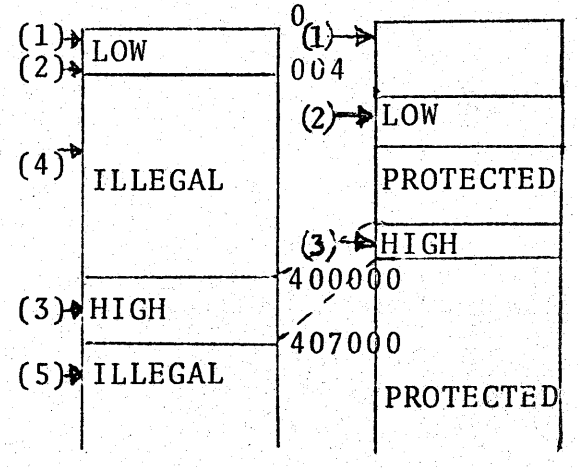



Figure 10.1

MEMORY VIOLATION level is raised which can be sensed by a CONI word and will cause an interrupt if the ALLOW MAP/ PROTECT INTERRUPT bit of the CONO word is set. The memory protection and relocation function is enabled by setting bit 21 of the CONO word. A PDP-10 DATAO instruction is used to load the four 8-bit registers and the control bit which sets write protect for high memory. The LDS-1 memory mapper is device 134 on the PDP-10. Some examples of the operation of the memory protection and relocation function are also given in figure 10.1.

# CHAPTER 11

# SWITCHES, BUTTONS, KNOBS AND LIGHTS (The SBKL Option)

## 11.1 General

The Evans & Sutherland SBKL Option provides the capability to simply and easily interface Switches, Buttons, Knobs, and Lights to the display system. The SBKL Option may include accommodations for between 1 and 4 user consoles, where each console consists of eight knobs, 16 input switches, and 16 lights. The SBKL Option provides very simple replacement of the console controls by the customer with controls arranged to suit his convenience since all operator control boxes are removable from the cables connecting them to the unit. The operator control boxes themselves contain nothing but switches, lights, potentiometers, cables, cable connectors, and ordinary wire. All electronic parts are located in the fixed electronic boxes of the SBKL Option.

## 11.2 Operation of the SBKL Option

The SBKL Option contains a register memory of 32 words for each pair of consoles, each register being 16 bits long. These registers are accessed in pairs by the LDS-1 as 16 words. These 16 registers may be loaded, stored, sinked or retrieved by LDS-1 using ordinary group 3 instructions. The SKBL Option regularly senses the contents of these registers and transfers whichever of them are appropriate to the indicator lights. Similarly, the SBKL Option regularly senses the position of the switches and transfers them to whichever of its storage registers is appropriate. And finally, the SKBL Option regularly senses the voltage output of the knobs, converts it to a digital value, and places that value in its corresponding internal register. Updating of the information takes place about every 2 milliseconds or about 500 times per second. Thus, although the programmer can load or store any of the registers in the SBKL Option, the registers assigned to switches and knobs will change spontaneously to reflect the condition of these input devices.

The SBKL Option is assigned to be device number 1110 (expanded memory section is 1111) (binary). Thus, the binary representation of a command to access it should be:

011 xx 1110 aaaa i nnnn address

where the 011 is the load store group, xx indicates load, store or retrieve, 1110 addresses the SBKL Option (first pair of

consoles), and aaaa is the octal address of the SBKL register being treated. Multiple load and store operations are possible and practical: the field nnnn gives the count of registers to be transferred, where binary 0000 transfers 16 registers, and binary 0001 transfers 1 register. For 4 user consoles, device code 1111 is used to access 16 additional registers.

The registers in a two-console SBKL Option are assigned as shown below:

```
xxxxxxxxxxLLLLLLLL,,xxxxxxxxxxLLLLLLLL
xxxxxxxxxxSSSSSSSS,,xxxxxxxxxxSSSSSSSS
xxxxxxAAAAAAAAAAxx,,xxxxxxAAAAAAAAAAxx
```

where the bits "L" drive the lights in order, the bits "S" are set according to the position of the switches, the bits "A" are set according to the position of the knobs, and the bits "x" are ignored, and will be set to zero on input devices.

The numerical values of the numbers in these registers are established by the switches and knobs, and control the lights. When an "L" bit is set, the corresponding light will be lit. When a switch is set in the neutral position, the corresponding "S" bit will be zero; when the switch is "closed," (on the standard tabletop unit, that means either pushed upward [for spring return] or locked downward [until manually returned]) the corresponding "S" bit will be a one. For the knobs, the value of A can range from 0000000000 to 1111111111. The value 0000000000 corresponds roughly to one extreme or knob rotation, the value of 1111111111 corresponds to the other extreme or knob rotation. Thus, the number AAAAAAAAAA should be considered to be a 10-bit unsigned binary quantity. No sign extension is provided; the x bits will appear always to be zero.

The full range of possible values for AAAAAAAAAA may never be reached. For one extreme of knob rotation, a small but non-zero value will be read; for the other extreme of knob rotation, a value slightly less than 1111111111 will be read. The programmer must measure these values and interpolate properly between them. The readings obtained at the extreme of knob rotation may vary slightly from time to time.

In programming with the SBKL Option, it is well to remember the sampling rate properties of the devices. Each knob is sampled about every 2 milliseconds or about 500 times per second. Thus, the values read will remain the same for about that period of time even if the knob is actually being turned.

THE STANDARD SBKL CONFIGURATION

| | Pair No. (octal) | | |
|---|---|---|---|
| USER #1 | 0 | Knob 0 (X) | Knob 0 (Y) |
| | 1 | Knob 1 (X) | Knob 1 (Y) |
| | 2 | Knob 2 (X) | Knob 2 (Y) |
| | 3 | Knob 3 (X) | Knob 3 (Y) |
| | 4 | Switches (1-8) | Switches (9-16) |
| | 5 | Unused | Unused |
| | 6 | Lights (1-8) | Lights (9-16) |
| | 7 | Unused | Unused |
| USER #2 | 10 | Knob 0 (X) | Knob 0 (Y) |
| | 11 | Knob 1 (X) | Knob 1 (Y) |
| | 12 | Knob 2 (X) | Knob 2 (Y) |
| | 13 | Knob 3 (X) | Knob 3 (Y) |
| | 14 | Switches (1-8) | Switches (9-16) |
| | 15 | Unused | Unused |
| | 16 | Lights (1-8) | Lights (9-16) |
| | 17 | Unused | Unused |

Figure 11.1

## 11.3 Instructions

Since the SBKL Option can be configured in several ways, it is "legal" to load, store, sink or retrieve any of the 16 registers. In any given configuration, of course, there are nonsensical combinations, like loading registers connected to switches.

All of the instructions defined in this section should be used only in 2D. The registers are numbered $0\text{-}17_8$.

### LOSBKL

To load a register pair, use the instruction:

```
LOSBKL  REG, ADDR(COUNT)
```

to specify the COUNT register pairs, starting at REG, are to be loaded from core beginning at ADDR.

### STSBKL

To store a register pair, use:

```
STSBKL  REG, ADDR(COUNT)
```

specifying that COUNT register pairs starting at REG are to be stored into memory locations beginning at ADDR.

### SKSBKL

Register pairs may be pushed onto the data sink by the instruction:

```
SKSBKL  REG,  (COUNT)
```

which pushes COUNT register pairs starting at REG onto the data sink.

### RTSBKL

Register pairs that have been pushed onto the data sink may be retrieved by the instruction:

```
RTSBKL  REG,  (COUNT)
```

specifying that COUNT register pairs counting backward from REG are to be retrieved from the data sink. Thus, the order of retrieving is the reverse of the order of sinking, so that each register pair receives its proper former value.

The instructions above apply to any SBKL configuration. A set of higher level mnemonics have been defined for the standard configuration, consisting of eight X-Y pairs of knobs, one set of sixteen switches, and one set of sixteen lights.

STKNOB

To interrogate a pair of knob settings, use the instruction:

```
STKNOB  NUM, ADDR(COUNT)
```

specifying that knob pairs starting at pair number NUM are to be stored into core at location ADDR. Knob pairs are numbered 0 - 3. COUNT should not be 0 or omitted, as this would specify 16 pairs, and NUM + COUNT should not exceed 4.

For example, the instruction:

```
STKNOB  2, LOC(2)
```

would store the values of knob pair 2 into location LOC and knob pair 3 into LOC+1.

STSWCH

To interrogate the switch settings, use the instruction:

```
STSWCH ADDR
```

where ADDR is the memory location in which the switch settings are to be stored. This instruction has a built-in COUNT of 1, and therefore this field should be omitted.

The memory word will correspond to the switch settings in the following way:

```
xxxxxxxxxxSSSSSSSS,,xxxxxxxxxxSSSSSSSS
```

where the two groups of S's correspond to the two banks of switches and the x's are always 0. An S of 0 means the switch is in the neutral position; and S of 1 means the switch is up (for spring return) or down (for manual return).

LOLITS

To load the register driving the lights, use the instruction:

```
LOLITS  ADDR
```

where ADDR contains the desired light settings, and a built-in COUNT of 1 is supplied.

The memory word should look like this:

```
xxxxxxxxxxLLLLLLLL,xxxxxxxxxxLLLLLLLL
```

where the groups of L's drive the two banks of lights (L = 1 turns on a light, L = 0 turns it off) and the x's are ignored.

# CHAPTER 12

# LDS-1 TABLET INTERFACE

## 12.1 Function

The LDS-1 Tablet Interface is an optional accessory to the LDS-1 Display Processor. Although it was designed primarily for tablets, it may be used in conjunction with any two-dimensional position encoder, and allows program-controlled transfer of single or multiple words from the encoding device to memory. The single word operation is useful for pointing and control functions, while multiple word operation is used primarily for automatic "inking".

## 12.2 Programming

The tablet interface is controlled by program flag 4 (PF4), and tablet Z values may be read by testing program flags 5 and 6. Words are written into memory at the location specified by the processor Write Address Register (WAR), and the number of words to be transferred is specified by the Write Count Register (WCR), exactly as in memory-to-memory operation. The word(s) written into memory have X in the left half and Y in the right half. The tablet center is (0,0), the upper right corner is (3777,3777) and the lower left corner is (-3777,-3777). This coordinate system is exactly like that of the scope. If the tablet (or other device) has less than 12 bits resolution, then the unused bits will be 0's (i.e. for a 10-bit tablet, the two rightmost bits of the left and right halves will be 0).

The tablet interface includes a timer(nominally 20 milliseconds) which is used to meter out inking points at a reasonable rate. The timer is initiated following each but the last write in a multiple transfer, and inhibits transfer of data for its duration. This allows single transfers to occur as rapidly as desired, but slows down multiple transfers.

Tablet writing is enabled by setting PF4 and disabled by clearing PF4. PF4 is automatically cleared by "system clear" and by a positive WCR. The WAR and WCR should be loaded before setting PF4. The usual coding for a single transfer is:

```
LI      WCR,-1
LALST   WAR, WHERE(PF4)
JIF     . (PF4)  ;PF4 CLEARED WHEN DONE
```

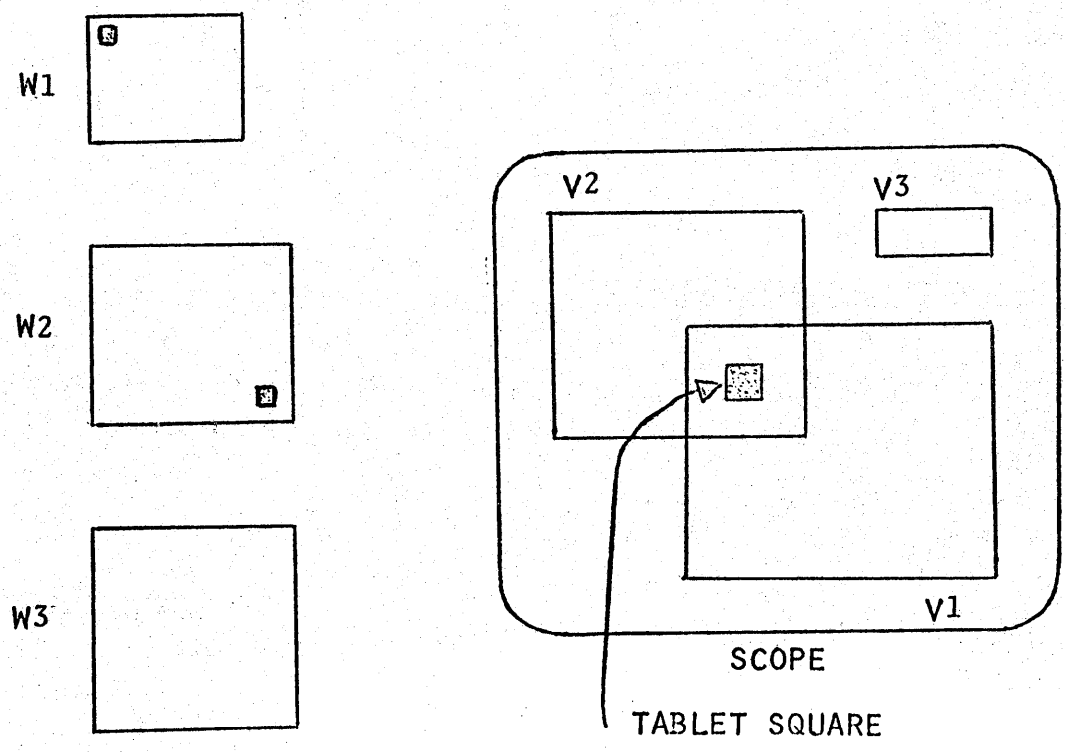
Multiple transfers require a buffer area. One good strategy is to display this buffer using a window of (3777,3777) size absolute, and to fill it initially with values outside this window. In this way, the inking points will appear as they are written into memory. Example:

```
;  SETUP SUBROUTINE FOR INKING
;

TSETUP:    LI       RCR,-400
           SETPTA   [XWD 400000,400000]
           LI       RAR,IBUF
           STCL     SAVELB,@(2)
           IJNRCR   .-1
           LI       WCR,-1000
           LALST    WAR,IBUF(PF4)
           PEEL
;
;  LINK TO THIS SUBROUTINE TO DISPLAY INK
;

INK:       LOCLSA   WIND,[XWD 3777,3777] (1)
           LI       RCR,-1000
           DOTSAA   IBUF(RPTM) ; OR POLAA IS NICE
           PEEL
;
IBUF       BLOCK    1000
```

## 12.3 Additional Programming Hints

Use of a tablet requires some periodic checks, which are generally done at the beginning or end of each complete display frame. To show the program user where he is pointing, read the tablet each frame and display a cursor at this point. This is not necessary (or possible) during inking, but while inking the Z value given by the tablet should be checked each frame to see if the pen has been lifted and the inking should be stopped. When using the tablet for pointing (i.e. at light buttons, picture features) there are two basic strategies. The first is to test once each frame (usually at the beginning) whether the tablet is pointing at any of a list of features. The second is to leave the tablet position in a particular memory location and incorporate the tests into the code for each feature. The first method is often preferred if the pointing is to perform a gross change (such as selecting another picture) and if the items pointed at are relatively static (such as light buttons). The second method is often necessary for more complex interaction.

One generally thinks of the boundaries of the tablet corresponding to the boundaries of the scope, or perhaps of some viewport on the scope. Pictures are defined in page coordinates, which bear an indirect and perhaps a variable relation to the scope. For hit tests, it is necessary to map the tablet position "backwards" from scope to page. Consider the place pointed at to be a small square on the scope, centered on the position of the tablet with a "palsy allowance" defined by the size of the square.



Each time a window/viewport combination is loaded, we can call a subroutine which sinks the window and viewport, then exchanges them, loads the tablet square as an instance, tests area-in-common; and if there is AIC, does a Box operation (master doesn't matter). The new viewport result of the Box operation is the appropriate window for hit testing.

# CHAPTER 13

# LORGNETTE COLOR/STEREO SYSTEM

## 13.1 Introduction

The Lorgnette (pronounced "Lorn-yet") is a light-weight, hand held viewing system which, with associated electronic circuitry, permits the user to observe computer generated displays in programmed color or stereo or both. The Lorgnette system allows several viewing units to be driven in exact synchronism, thus enabling multiple users to see the color and/or stereo pictures simultaneously.

## 13.2 Principles of Operation

The basic Lorgnette system includes two motor driven viwing units. Each viewing unit contains a six-segment disk which rotates at about 10 revolutions per second, inside a transparent plastic housing. The user holds the viewing unit directly in front of his face so that the axis of rotation, if extended, would pass directly through the bridge of his nose. The user's eyes look through diametrically opposite sides of the rotating disk. Marks are provided on the front of the viewer housing to assist in maintaining correct viewing alignment.

Three viewing disks are furnished with each viewing unit, and are easily interchangeable without tools. These disks are keyed so that they can be installed on the motor hub in only one position. The color disk is used for viewing displays in color and contains segments in the order red, green, blue, red, green, blue. A second disk contains segments in the order clear, opaque, clear, opaque, clear, opaque. This is the stereo disk and provides for uncolored stereo viewing because the user's left and right eye views are alternately blanked by an opaque segment. The final disk is the color/stereo disk and contains segments in the order red, opaque, blue, opaque, green, opaque. This disk permits the viewer to see colored stereo pictures.

The color or stereo effect of the Lorgnette is achieved because successive frames of information presented on the display are seen by the user through different filters. This technique of color or stereo viewing is known as "field sequential" color or stereo. The eye retains the images seen and fuses them into a single colored or three-dimensional image.

For example, if a stereo image is to be seen, the clear and opaque disk will be used. The user will see alternate frames with each eye. If these frames represent slightly different perspective views of a three-dimensional object, the user's eye will fuse them into a single three-dimensional impression. This form of "time division stereo" works just as well as the more usual "space division stereo" in which the separate images are presented simultaneously but separated for the two eyes by optical means. Observers without Lorgnette viewing units will see both images super-imposed on the screen and will not get the stereo effect.

The rotation of the colored disks and the presentations of the pictures on the CRT must be synchronized. Information to be seen by the users as "red" must be presented during the time that the red segment of the disk is in place in front of the user's eyes. Obviously, all disks that are used on a single display must themselves be driven in exact synchronism. If the display is to be sunchronized to the rotating disks, signals must be provided to the Lorgnette system. It is the purpose of the Lorgnette Power and Synchronization unit to provide for all such synchronization signals.

Color pictures can be seen on any display with the Lorgnette but because many blue and green phosphors are deficient in red output, the red colors may be weak. Stereo presentations require that the right eye image be extinguished before the left eye is uncovered by the rotating disk. Thus, a fast-decaying phosphor is required for stereo presentations. The well-known television phosphor, P4, is suitable for both stereo and color because it is both fast and white.

## 13.3 The Lorgnette Motor Drive Circuit

The Lorgnette is made possible by a special type of synchronous motor. This motor has a permanent magnet rotor and a four-phase star-connected stator. Electronic detectors within the motor detect the position of the rotor. The signals delivered by these detectors are used to control the currents in the stator so that the motor will start and run at approximately the correct speed. In addition, synchronous running currents are provided to ensure that all of the motor units turn in exact synchronization.

The synchronous drive currents and the starting and damping drive currents combine to provide for stable synchronous operation of the motor. The starting currents provide a torque on the motor which decreases with the motor velocity. The synchronous running currents provide a torque on the motor which is proportional to the error in the motor's position from proper phase.

The damping term is very important. Without proper damping, any synchronous drive system will "hunt". In the Lorgnette system, this damping is provided by decreasing the starting torque as the Lorgnette velocity increases. When correctly adjusted, the starting torque should be just sufficient to drive the motor against friction and windage at about the correct synchronous speed. A TOGGLE-SWITCH is provided on the Lorgnette Power and Synchronization panel to remove the synchronous drive while adjusting the starting and damping torque. When this SWITCH is UP, the Lorgnette motors should neither speed up nor slow down. Each motor has a screwdriver adjustment, accessible from the front of the panel, which changes the magnitude of the damping term. The potentiometer is just sufficient to drive the motor against friction and windage about synchronous speed. When correctly adjusted, the motor will neither speed up nor slow down when the synchronous drive is interrupted by FLIPPING the SYNC SWITCH UP. Because the Lorgnette motor will operate satisfactorily over a wide range of misadjustment, the adjustments can be checked infrequently. FOR NORMAL OPERATION, THE SYNC SWITCH SHOULD BE IN THE "DOWN" POSITION.

## 13.4 Lorgnette Synchronization

Synchronizing signals for the Lorgnette motors are provided by the clock card. The clock card accepts a driving signal of about 120 Hz and provides for drive of the motors of one-twelfth of the drive frequency. The Lorgnette system is delivered with the synchronization drive derived from the power line, but other drive signals between about 100 and 150 Hz can be used. The drive signal must be sufficient to switch the one-shot multivibrator located on the clock card. Signals which vary between about ground and about +3 to +5 volts will work. Signals larger than +5 volts should be avoided.

The one-shot multivibrator drives a six-stage shift register. The output of the shift register is fed back to its input in such a way that a unique shift register sequence is generated. This shift register sequence has 12 unique states which are shown below. Alternatively, the shift register sequence and timing signals might be provided by the computer to which the Lorgnette system is connected.

The shift register outputs are used to generate two sets of signals. One set of signals is used to drive the motors. These signals are equally spaced in time and each two-twelfths of a period long. The four signals are equally spaced in time so that there is a one-twelfth time period between them when none of them are high. The synchronizing signals are disabled whenever the SYNC switch is UP. The other set of signals $\Phi 1$, $\Phi 2$, and $\Phi 3$ are used to indicate the position of the disks to

the computer. These signals change six times during a complete cycle. In addition, every other output pulse of the multivibrator itself is provided to the computer on a line called "CLOCK". This CLOCK signal is an indication that the joint between segments of the disk is in front of the observer's eye. The multivibrator delay is set at the factory to about four milliseconds.

SHIFT REGISTER SEQUENCE

| Φ1 | Φ2 | Φ3 | Color Disk | Stereo Disk | Color Stereo Disk |
|---|---|---|---|---|---|
| 0 | 0 | 0 | Red | Right | Red Right |
| 1 | 0 | 0 | Green | Left | Green Left |
| 1 | 1 | 0 | Blue | Right | Blue Right |
| 1 | 1 | 1 | Red | Left | Red Left |
| 0 | 1 | 1 | Green | Right | Green Right |
| 0 | 0 | 1 | Blue | Left | Blue Left |

Fig. 1

Obviously, the phase relationship between the phasing signals delivered to the computer and the actual position of the disks must be adjusted to match. Each Lorgnette motor and viewing unit has a mechanical phasing adjustment. This adjustment is achieved by rotating the motor mounting disk with respect to its housing. The motor mounting should be adjusted so that the color disk segments correspond to the output coding and so that the transition between colors occurs at the center of the multivibrator pulse. This adjustment should change very little from time to time and so a few degrees of rotation should suffice. The motor housing disk has a mark put on it at the factory to indicate approximately the correct phasing. This mark should be at or near the center of the handle. If the motor is removed from its mounting plate or the hub for mounting disks is removed, the mark may have to be moved.

A quick check of phasing may be performed by connecting an oscilloscope to the test BNC jack located on the panel and setting the horizontal sweep and triggering until 3 complete cycles of the waveform are stationary on the screen. When viewed through the Lorgnette, the trace should be solid color during the time the waveform is "low" and the transition between colors should occur while the waveform is "high".

## 13.5 Power Supply and Regulator

The Lorgnette contains its own power supply and filter. This power supply provides a reference voltage and regulated voltage to each of the motor driver cards. The reference voltage is regulated by a zener diode located in the main power supply and is used to supply +5V to the clock card integrated circuits.

The regulated voltage is supplied by a series regulator located on the regulator subassembly and furnishes power to the motor driver cards and the motors. It is factory adjusted to +8.0 volts and normally should not be changed. Refer to the Maintenance Manual for adjustment procedures.

## 13.6 Interfacing the Lorgnette

The signals Φ1, Φ2, Φ3 and the CLOCK destined for the computer are delivered to the level shifter card. The level shifter card takes special power voltages and ground from the computer and delivers the three Φ signals and the CLOCK at the electrical levels appropriate to the particular computer in question.

Level connectors are available for 0 and -3 volts (DEC standard), and 0 and +5 volts (TTL levels). The appropriate level shifter card will be supplied for the particular computer to which the Lorgnette system is applied.

## 13.7 Programming the Lorgnette

There are four signals coming from the Lorgnette to indicate the color in front of the eye. Three of the four levels indicate the color and the fourth indicates when the other three levels are valid.

The valid bit (valid when low) is called "clock enable" and wired to program flag 7 in the Channel Control STATUS REGISTER (SR). The three phase signals come in as program flags 14, 15, and 16. Phase 0 is on 14, phase 1 is on 15, and phase 2 is on 16.

These four levels can be sensed by use of Group 2 instructions:

| Condition | Tests |
|---|---|
| PF 7 | Clock enable (low for ok) |
| PF14 | Phase 0 |
| PF15 | Phase 1 |
| PF16 | Phase 2 |

The color codes are as follows:

| Condition | | | | Stereo | Color | Color Stereo |
|---|---|---|---|---|---|---|
| 7 | 14 | 15 | 16 | | | |
| 0 | 0 | 0 | 0 | Right | Red | Red - Right |
| 1 | 0 | 0 | 0 | Changing | Color Changing | Color Changing |
| 0 | 1 | 0 | 0 | Left | Green | Green - Left |
| 1 | 1 | 0 | 0 | Changing | Color Changing | Color Changing |
| 0 | 1 | 1 | 0 | Right | Blue | Blue - Right |
| 1 | 1 | 1 | 0 | Changing | Color Changing | Color Changing |
| 0 | 1 | 1 | 1 | Left | Red | Red - Left |
| 1 | 1 | 1 | 1 | Changing | Color Changing | Color Changing |
| 0 | 0 | 1 | 1 | Right | Green | Green - Right |
| 1 | 0 | 1 | 1 | Changing | Color Changing | Color Changing |
| 0 | 0 | 0 | 1 | Left | Blue | Blue - Left |
| 1 | 0 | 0 | 1 | Changing | Color Changing | Color Changing |

The following is a programming example, using the Lorgnette on the Evans & Sutherland Line Drawing System, Model 1, to draw a set of boxes. A blue box is seen very close to the observer; a green box at an intermediate distance; and a red box very far away.

```
;          LORGNETTE TEST, DEMO, AND PROGRAMMING EXAMPLE
;          (PDP-10/LDS-1)
;
;
;    VALID BIT IS CONDITION 7, AND IS VALID IF 0
;    CODE IS CONDITIONS 14, 15, 16, AS SHOWN BELOW
;
;    LORGNETTE CODE:
;    ***************
;
;         CODE     COLOR     EYE
;         000      RED       R
;         100      GREEN     L
;         110      BLUE      R
;         111      RED       L
;         011      GREEN     R
;         001      BLUE      L
;         000
;                  ETC.
;
;    THE CODES APPEAR IN THE ORDER SHOWN --
;    THE COLOR/STEREO WHEEL COMBINES COLOR & STEREO
;    ACCORDING TO THE ABOVE TABLE --
;
;
;
START:   DSTART(DISPAT)          ;PDP-10 CODE TO START LDS-1
         HALT
;
LDS-1 CODE:
;
DISPAT:  JIF      @.(PF7)
         JIF      .(PF7)         ;FALLS OUT WHEN JUST BECAME VALID
         LI       RCR,-10
         IJNRCR   .              ;DELAY
;
         JIF      .+4(PF14)
         JIF      C011(PF15)     ;(01X) GRN R
         JIF      C001(PF16)     ;(001) BLU L
         JMP      C000           ;(000) RED R
         JIF      @C100(PF15)    ;(10X) GRN L
```

```
                JIF     @C110(PF16)         ;(110) BLU R
                JMP     C111                ;(111) RED L
;
;
C000:           SETPTA  [XWD -400,0]        ;RED RIGHT
                JMPPSH  BOX
                JMP     DISPAT
;
C100:           SETPTA  [XWD 0,0]           ;GREEN LEFT
                JMPPSH  BOX
                JMP     DISPAT
;
C110:           SETPTA  [XWD 400-100,0]     ;BLUE RIGHT
                JMPPSH  BOX
                JMP     DISPAT
;
C111            SETPTA  [XWD -400,0]        ;RED LEFT
                JMPPSH  BOX
                JMP     DISPAT
;
C011:           SETPTA  [XWD 0-40,0]        ;GREEN RIGHT
                JMPPSH  BOX
                JMP     DISPAT
;
C001            SETPTA  [XWD 400,0]         ;BLUE LEFT
                JMPPSH  BOX
                JMP     DISPAT
;
;
BOX:            LI      RCR,-40
                LIPSHM  RCR,-4
                DRAWTR  BOXT(RPTM+PEELM)
                IJNRCR  .-2
                PEEL
;
BOXT:           XWD     100,0
                XWD     0,100
                XWD     -100,0
                XWD     0,-100
;
;
                END     START
```

## 13.8 Special Applications

The Lorgnette disk normally rotates clockwise as viewed in the normal way, that is, from the shaft end of the motor. The color sequence is red, green, blue, with the color indicating signals as shown in Table 1. With such rotation, an observer's right eye looks through the color filters which move down, and the left eye looks through color filters moving up.

For television applications, it may be desirable to have both eyes observe the screen through wheels which are moving down. To accomplish this, two Lorgnette motor and viewing units should be used. The observer will put his nose between the two units, as shown in figure 2. The direction of rotation for the left eye should be clockwise, the direction should be counter-clockwise for the right eye.

TECHNIQUE FOR VIEWING RASTER PICTURES

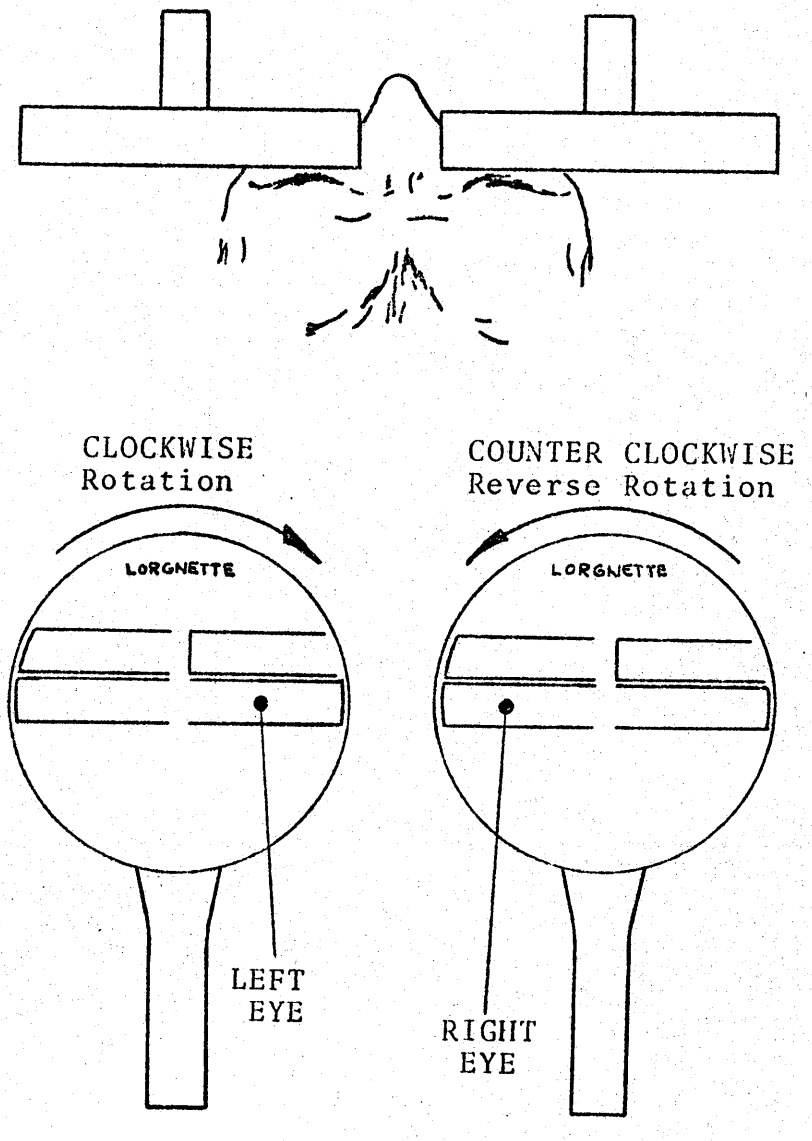



FIGURE 2

# APPENDIX I

## LDS-1 PROCESSING TIMES

The LDS-1 will process and display an "average" picture (i. e. a picture with a typical distribution of long and short lines) of 2500 lines. This figure includes use of full pipeline processing. If more lines than this are presented, the picture may begin to flicker.

Because the LDS-1 operates as an asynchronous pipeline, it is not always significant to calculate how long an individual instruction will take. Group 0 instructions, which require only the Channel Control to execute require 2.5 microseconds, but if these instructions are executed while other parts of the pipeline are still processing previous data, these instructions may be performed without adding to the total processing time for the system.

The Matrix Multiplier will multipy the incoming data by the transformation matrix in from 6 to 10 microseconds (depending on the setting of the Matrix Multiplier clock).

The period of the Clipping Divider clock is 0.5 usec. We are attempting to operate the clock considerably faster than this, probably in the 0.35 usec. region. Clipping Divider processing of dots and lines starts with 5 clock times of "setup." Center size specification and processing dots requires one additional clock time during the "setup" phase. Processing then proceeds to the "clipping" step, in which any portion of the picture outside the window is eliminated. Dots require only a single clock time in this phase of the computation. The time required for a line depends on its position relative to the window. If both ends of a line are within the window, the "clipping" phase is completed in 1 clock time. Otherwise the processing time is from 1 to 20 clock periods, tending toward smaller times if the line is entirely outside the window. The maximum clipping time is $[\log_2 L]$ clock periods where L is the larger of the X and Y line lengths. If the dot or any segment of the line was within the window, the processing then proceeds to the perspective division and the window-to-viewport mapping. The processing time here, following 1 "setup" step, is from 1 to 20 clock times depending upon the size of the window and the viewport. The maximum processing time will be $[\log_2 W]$ clock periods where "W" is the larger of the dimensions of the viewport or the larger of the dimensions of the window, whichever is smaller. The worst case for processing lines is 46 clock times (for 20 bit numbers) or 42 clock times (for 18 bit numbers) and the best time for rejecting a line is 6 clock times. Sending to memory requires several clock times, sending to the scope requires from 1 to 2 clock times, and sending depth cueing information to the scope requires several clock times.

After the usual "setup" operations, boxing requires 2 clock times for determining that the window and instance have no area in common, or from 12 to 31 steps to compute a new window and viewport. Loading and unloading Clipping Divider viewing parameters requires from 4 to 6 clock times.

# APPENDIX II

## MNEMONICS

### II.1 The Instructions

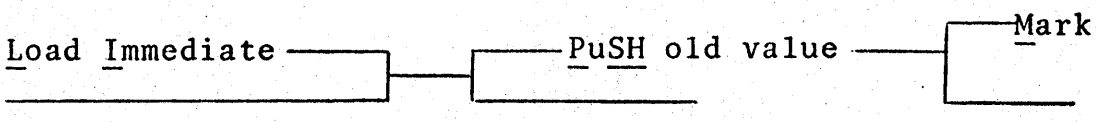
| MNEMONIC | ARGUMENTS | INDIRECT BIT |
|---|---|---|
| GROUP 0 | | |
| LI---- | CC REGISTER,DATA(NEXT MODE) | ** |
| PSH- | CC REGISTER,(NEXT MODE) | ALWAYS SET |
| JMP--- | ADDRESS(NEXT MODE) | NO JUMP |
| NWSTK- | ADDRESS(NEXT MODE) | NO SP-CHANGE |
| XQTA | ADDRESS | ** |
| NOP | | ALWAYS SET |
| XQT | | ALWAYS SET |
| RPT | THE ADDRESS PART IS IGNORED | ALWAYS SET |
| PROG | | ALWAYS SET |
| PEEL | | ALWAYS SET |
| GROUP 2 | | |
| LIF-- | CC REGISTER,DATA(CONDITION) | REVERSE DECISION |
| LAL-- | CC REGISTER,DATA(CONDITION) | ** |
| JIF-- | ADDRESS(CONDITION) | REVERSE DECISION |
| JAL-- | ADDRESS(CONDITION) | NO JUMP |
| CL | (CONDITION) | ALWAYS SET |
| ST | (CONDITION) | ALWAYS SET |
| CM | (CONDITION) | ALWAYS SET |
| IJ--CR | ADDRESS | REVERSE DECISION |
| STOP | THE ADDRESS PART IS IGNORED | ALWAYS SET |
| GROUP 3 | | |
| LO---- | REGISTER,ADDRESS(N) | ** |
| ST---- | REGISTER,ADDRESS(N) | ** |
| RT---- | REGISTER,(N) | ALWAYS SET |
| SK---- | REGISTER,(N) | ALWAYS SET |
| NOMM | (MAX. COUNT) | ALWAYS SET |
| PUSHMM,POPMM | MM-REGISTER,(N) | ALWAYS SET |
| GROUP 4 | | |
| SETPT- | ADDRESS-OF DATA(NEXT MODE) | ** |
| DRAW-- | ADDRESS-OF-DATA(NEXT MODE) | ** |
| LIN-- | ADDRESS-OF-DATA(RPTM) | ** |
| POL-- | ADDRESS-OF-DATA(RPTM) | ** |
| DOTS-- | ADDRESS-OF-DATA(RPTM) | ** |
| STAR-- | ADDRESS-OF-DATA(RPTM) | ** |
| BOX-- | ADDRESS-OF-DATA | ** |

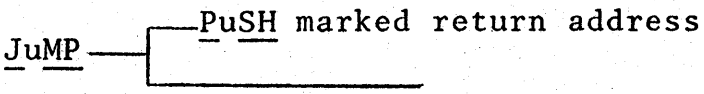
| | | |
|---|---|---|
| GROUP 5 | | |
| LINI-- | ADDRESS-OF-POINTER(RPTM) | ** |
| POLI-- | ADDRESS-OF-POINTER(RPTM) | ** |
| GROUP 6 | | |
| ---CRV | (NEXT MODE) | ALWAYS SET |
| GROUP 7 | | |
| DOCHAR | ADDRESS-OF-DATA(RPTM) | ** |

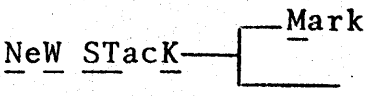
**If the indirect bit is set, the information in the address field of this instruction is not loaded into the processor. Memory address registers used will be incremented after use in the normal manner.

## II.2 Construction of Mnemonics for E&S Display Codes

GROUP 0

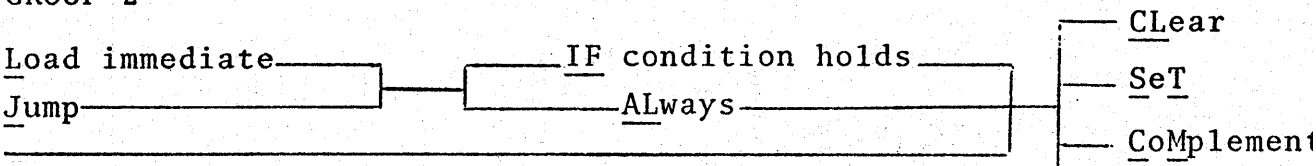


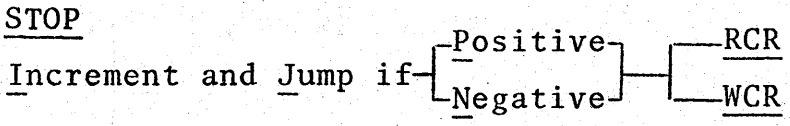




NOP , XQTA , XQT , RPT , PEEL , PROG

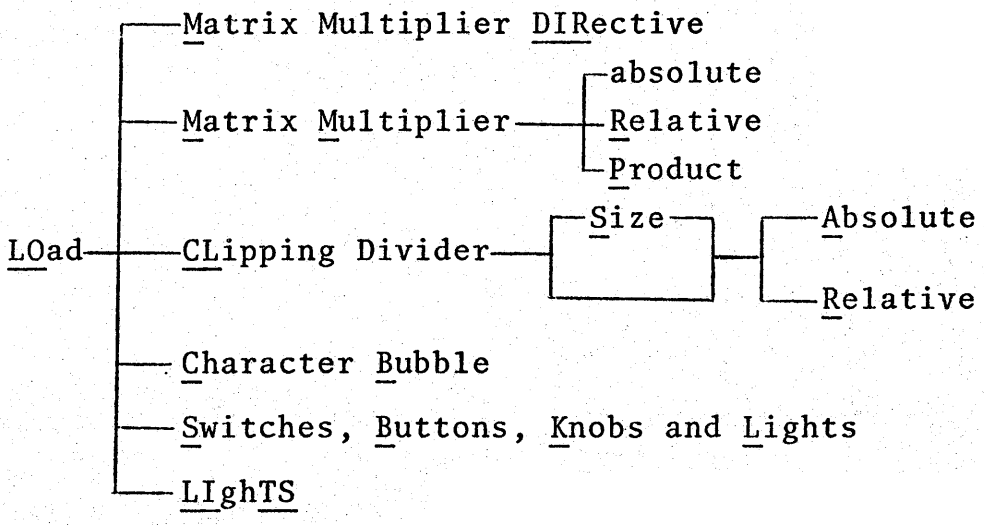
GROUP 2

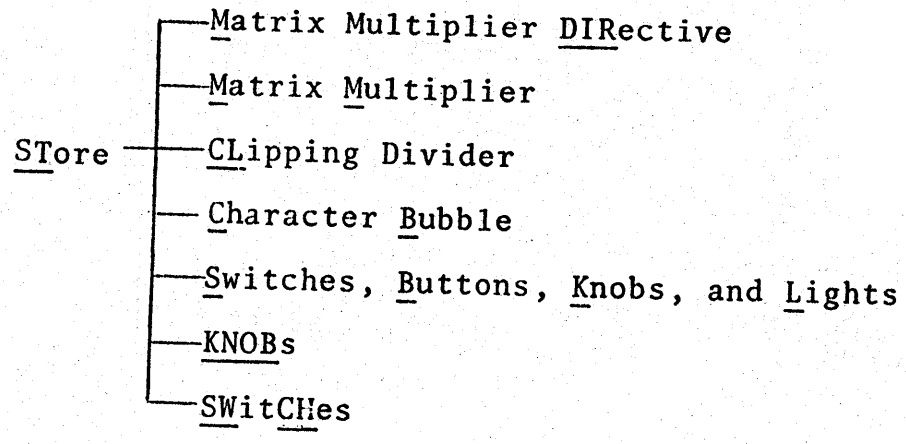


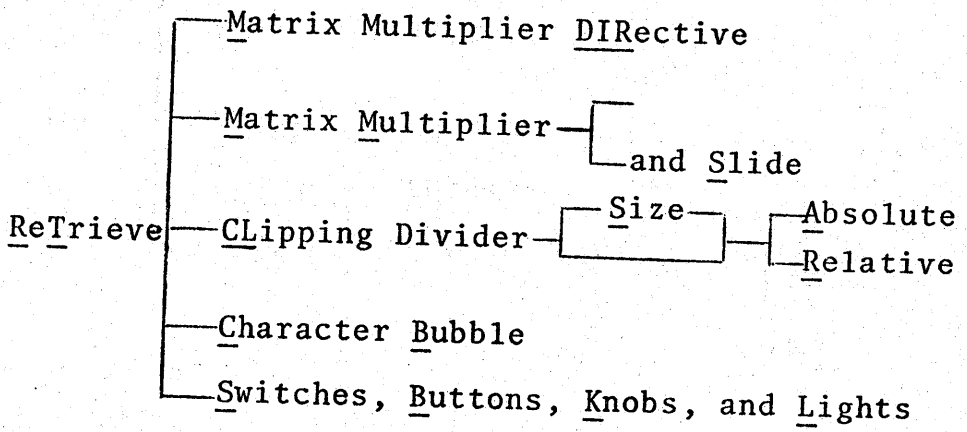

STOP

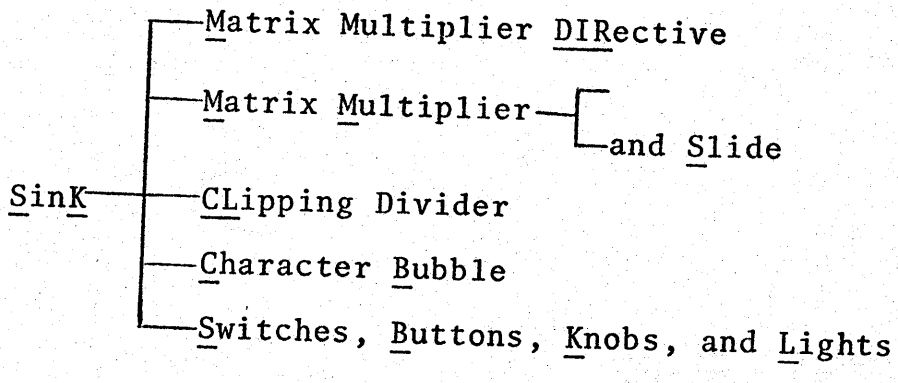


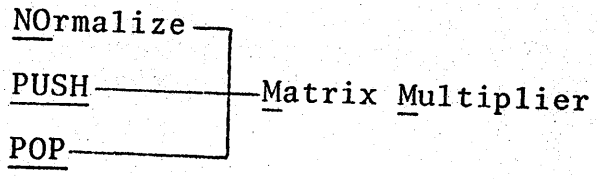

GROUP 3



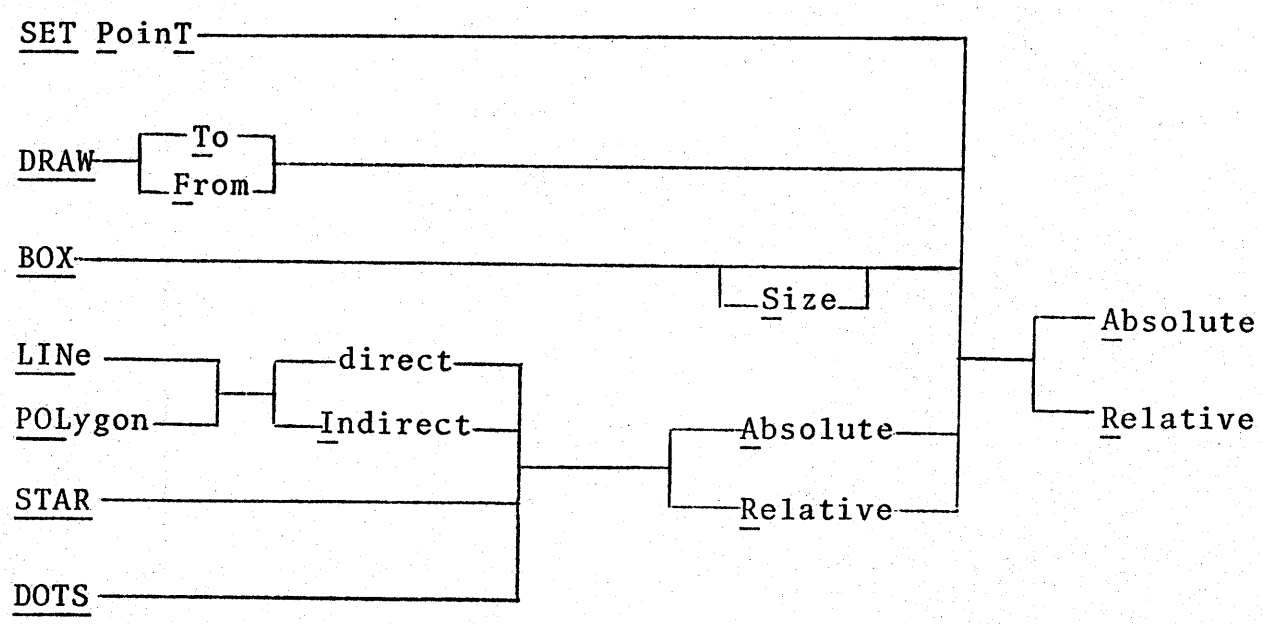

STore
Matrix Multiplier DIRective
Matrix Multiplier
CLipping Divider
Character Bubble
Switches, Buttons, Knobs, and Lights
KNOBs
SWitCHes


ReTrieve
Matrix Multiplier DIRective
Matrix Multiplier
and Slide
CLipping Divider
Size
Absolute
Relative
Character Bubble
Switches, Buttons, Knobs, and Lights

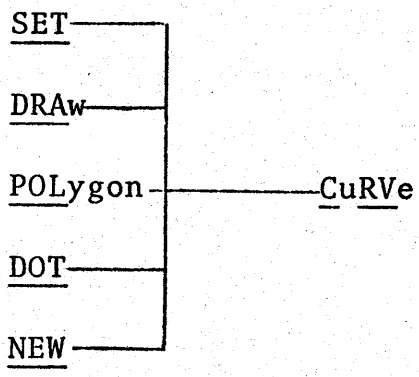

SinK
Matrix Multiplier DIRective
Matrix Multiplier
and Slide
CLipping Divider
Character Bubble
Switches, Buttons, Knobs, and Lights


NOrmalize
PUSH
POP
Matrix Multiplier

GROUPS 4 and 5




GROUP 6




GROUP 7

DO CHARacters

## II.3 Definition of Arguments for the E&S Display System Instructions

THE DISPLAY PROCESSOR (DP-REGISTER):

| | | |
|---|---|---|
| RAR | = | 0 |
| WAR | = | 1 |
| PC | = | 2 |
| SP | = | 3 |
| P1 | = | 4 |
| P2 | = | 5 |
| DSP | = | 6 |
| UR | = | 7 |
| RCR | = | 10 |
| WCR | = | 11 |
| DIR | = | 12 |
| RSR | = | 13 |
| SR | = | 14 |
| NEXT | = | 17 |

THE MODE (NEXT-MODE):

| | | |
|---|---|---|
| XQTM | = | 10 |
| RPTM | = | 4 |
| PEELM | = | 2 |
| PROGM | = | 1 |

CONDITIONS WHICH MAY BE CHECKED (CONDITION):

| | | | |
|---|---|---|---|
| PF0 | = | 0 | Program Flag #0 |
| PF1 | = | 1 | Program Flag #1 |
| PF2 | = | 2 | Program Flag #2 |
| PF3 | = | 3 | Program Flag #3 |
| PF4 | = | 4 | Tablet Read Signal |
| PF5 | = | 5 | Tablet Z Value |
| PF6 | = | 6 | Tablet Z Value |
| PF7 | = | 7 | Lorgnette Clock |
| RCRN | = | 10 | RCR Less Than -1 |
| WCRN | = | 11 | WCR Less Than -1 |
| HITF | = | 12 | Hit Flag |
| AICF | = | 13 | Area In Common Flag |
| PF14 | = | 14 | Lorgnette Color Code |
| PF15 | = | 15 | Lorgnette Color Code |
| PF16 | = | 16 | Lorgnette Color Code |
| STOPF | = | 17 | Stop Flag |

THE CLIPPING DIVIDER REGISTERS (CD-REGISTER):

| | | |
|---|---|---|
| SAVELB | = 0 | Two Components |
| SAVERT | = 1 | Two Components |
| VIEWLB | = 2 | Two Components |
| VIEWRT | = 3 | Two Components |
| WINDLB | = 4 | Two Components |
| WINDRT | = 5 | Two Components |
| INSTLB | = 6 | Two Components |
| INSTRT | = 7 | Two Components |
| NAME | = 10 | Two Components |
| CDIR | = 11 | Two Components, Second Ignored |
| HITANG | = 12 | Two Components, Hit Count + Angle Count |
| SELINT | = 13 | Two Components, Select + Intensity |
| SAVE | = 14 | Four Components |
| VIEW | = 15 | Four Components |
| WIND | = 16 | Four Components |
| INST | = 17 | Four Components |

THE CHARACTER BUBBLE REGISTERS:

| | |
|---|---|
| FONT | = 1 |
| CHAR | = 0 |

THE SBKL REGISTERS:

| | |
|---|---|
| LITS | = 6 |
| SWCH | = 4 |

BITS FOR THE DIRECTIVE REGISTERS:

| DIR BITS | | CDIR BITS | |
|---|---|---|---|
| JMMA | = 100000 | STOS | = 100000000000 |
| KMMA | = 40000 | STOM | = 40000000000 |
| JNO | = 20000 | ZTOS | = 20000000000 |
| KNO | = 10000 | PTOM | = 10000000000 |
| J3D | = 4000 | NTOM | = 4000000000 |
| K3D | = 2000 | TTO | = 2000000000 |
| JDT | = 1000 | JCURVE | = 1000000000 |
| KDT | = 400 | KCURVE | = 400000000 |
| JSOH | = 10 | JMEF | = 200000000 |
| KSOH | = 4 | KMEF | = 100000000 |
| JSOWCR | = 2 | JDL | = 40000000 |
| KSOWCR | = 1 | KDL | = 20000000 |
| | | SELFX | = 4000000 |
| | | SELFY | = 2000000 |
| | | TSELF | = 1000000 |

| MDIR | |
|---|---|
| JMOC | = 10000000000 |
| KMOC | = 4000000000 |
| JMOM | = 2000000000 |
| KMOM | = 1000000000 |
| TM3 | = 700000000 |
| TM2 | = 600000000 |
| TM1 | = 500000000 |
| TM0 | = 400000000 |
| JMCUR | = 40000000 |
| KMCUR | = 20000000 |
| TAKEQ | = 10000000 |

## II.4 The Complete List of Instructions

GROUP 0 - LOAD IMMEDIATE INSTRUCTIONS

| | | | |
|---|---|---|---|
| LI | = | [000 ,0] | Load Immediate |
| LIPSH | = | [040 ,0] | Load Imm. Push-Old-Value |
| LIPSHM | = | [060 ,0] | Load Imm. Push-Old-Value Marked |
| PSH | = | [LIPSH ,@0] | Push-Old-Value, Without Loading |
| PSHM | = | [LIPSHM ,@0] | Push-Old-Value Marked, Without Loading |
| NOP | = | [LI ,@0] | No-operation |
| JMP | = | [LI PC,0] | Jump |
| JMPPSH | = | [LIPSHM PC,0] | A Marked [JMP 0] Is Saved In Stack |
| NWSTK | = | [LIPSH SP,0] | Unmarked [LI SP,0] Is Saved In Stack |
| NWSTKM | = | [LIPSHM SP,0] | A Marked [LI SP,0] Is Saved In Stack |
| XQTA | = | [LI RAR,(XQTM)] | Execute the Instruction |
| XQT | = | [NOP ,(XQTM)] | Enter Execute Mode |
| RPT | = | [NOP ,(RPTM)] | Enter Repeat Mode |
| PEEL | = | [NOP ,(PEELM)] | Enter Peel Mode |
| PROG | = | [NOP ,(PROGM)] | Enter Program Mode |
| LIS* | = | [020 ,0] | LI + LOC+1 to RAR |
| RJMP* | = | [LIS PC,0] | RAR Jump |

*See Appendix V

GROUP 2 - CONDITIONAL LOAD IMMEDIATE

```
LIF    =  [200 ,0]          Load Immediate if Condition Holds
LIFCL  =  [210 ,0]          LIF And Clear
LIFST  =  [220 ,0]          LIF And Set
LIFCM  =  [230 ,0]          LIF And Complement

LAL    =  [240 ,0]          Load Always, LAL Is Slower Than LI
LALCL  =  [250 ,0]          LAL And Clear
LALST  =  [260 ,0]          LAL And Set
LALCM  =  [270 ,0]          LAL And Complement

JIF    =  [LIF PC,]         Jump If Condition Holds
JIFCL  =  [LIFCL PC,]       JIF And Clear
JIFST  =  [LIFST PC,]       JIF And Set
JIFCM  =  [LIFCM PC,]       JIF And Complement

JAL    =  [LAL PC,]         Jump Always, JAL Is Slower Than JMP
JALCL  =  [LALCL PC,]       JAL And Clear
JALST  =  [LALST PC,]       JAL And Set
JALCM  =  [LALCM PC,]       JAL And Complement

IJNRCR =  [JIFST ,(RCRN)]   Increment RCR, JMP If Negative
IJNWCR =  [JIFST ,(WCRN)]   Increment WCR, JMP If Negative
IJPRCR =  [IJNRCR ,@0]      Increment RCR, JMP if Positive
IJPWCR =  [IJNWCR ,@0]      Increment WCR, JMP If Positive

CL     =  [LALCL ,@0]       Clear Condition
ST     =  [LALST ,@0]       Set Condition
CM     =  [LALCM ,@0]       Complement Condition
STOP   =  [ST (STOPF)]      Stop The Processor

JIFDED =  [JIF ,(STOPF)]    Jump if Stopped
```

GROUP 3 = EXTERNAL DEVICE REGISTER TRANSMISSION

```
LOCLA   = [300 ,0]          Load Clipper Absolute
LOCLR   = [301 ,0]          Load Clipper Relative
LOCLSA  = [302 ,0]          Load Clipper Size Absolute
LOCLSR  = [303 ,0]          Load Clipper Size Relative
LOMM    = [304 ,0]          Load Matrix Multiplier Absolute
LOMMR   = [305 ,0]          Load Matrix Multiplier Relative
LOMMP   = [306 ,0]          Load Matrix Multiplier Product
LOMDIR  = [307 ,0(1)]       Load Matrix Multiplier Directive
LOCB    = [314 ,0]          Load Character Bubble Absolute
LOSBKL  = [316 ,0]          Load SBKL Absolute
LOLITS  = [LOSBKL  6,0(1)]  Load Lights

STCL    = [320 ,0]          Store Clipper Absolute
STMM    = [324 ,0]          Store Matrix Multiplier Absolute
NOMM    = [325 ,@0]         Normalize Matrix Multiplier
POPMM   = [326 ,@0]         Push Matrix Multiplier
STMDIR  = [327 ,0(1)]       Store Matrix Multiplier Directive
STCB    = [334 ,0]          Store Character Bubble Absolute
STSBKL  = [336 ,0]          Store SBKL Absolute
STKNOB  = [STSBKL ,0]       Store Knob
STSWCH  = [STSBKL  4,0(1)]  Store Switches

RTCLA   = [340 ,@0]         Retrieve Clipper Absolute
RTCLR   = [341 ,@0]         Retrieve Clipper Relative
RTCLSA  = [342 ,@0]         Retrieve Clipper Size Absolute
RTCLSR  = [343 ,@0]         Retrieve Clipper Size Relative
RTMM    = [344 ,@0]         Retrieve Matrix Multiplier Absolute
RTMMS   = [345 ,@0]         Retrieve Matrix Multiplier and Slide
RTMDIR  = [347 ,@0(1)]      Retrieve Matrix Multiplier Directive
RTCB    = [354 ,@0]         Retrieve Character Bubble Absolute
RTSBKL  = [356 ,@0]         Retrieve SBKL Absolute

SKCL    = [360 ,@0]         Sink Clipper Absolute
SKMM    = [364 ,@0]         Sink Matrix Multiplier Absolute
SKMMS   = [365 ,@0]         Sink Matrix Multiplier and Slide
PUSHMM  = [366 ,@0]         Pop Matrix Multiplier
SKMDIR  = [367 ,@0(1)]      Sink Matrix Multiplier Directive
SKCB    = [374 ,@0]         Sink Character Bubble Absolute
SKSBKL  = [376 ,@0]         Sink SBKL Absolute
```

Instructions in Groups 4-6 are defined using the following field definitions:

GROUP NAME:

| | | | |
|---|---|---|---|
| DD | = | 400 | Do Direct |
| DI | = | 500 | Do Indirect |
| DN | = | 600 | Do Internal |

THE WHAT-TO-DO MACHINE:

| | | | |
|---|---|---|---|
| LS | = | 060 | Lines = (SET-DRAWTO-SET-DRAWTO...) |
| LT | = | 070 | ????? = (DRAWTO-SET-DRAWTO-SET...) |
| PO | = | 030 | POLYG = SET-(DRAWTO-DRAWTO...) |
| TO | = | 020 | TO = (DRAWTO-DRAWTO...) |
| SS | = | 040 | STAR = SET-(DRAWFROM,DRAWFROM...) |
| FR | = | 050 | FROM = (DRAWFROM-DRAWFROM...) |
| DT | = | 010 | DOTS = (DOT-DOT) |
| BX | = | 000 | BOXES = (BOX-BOX...) |

THE ABS/REL-MODES MACHINE:

| | | | |
|---|---|---|---|
| RX | = | 007 | (REL-ABS-REL-ABS...) |
| AX | = | 006 | (ABS-REL-ABS-REL...) |
| RA | = | 003 | REL-(ABS-ABS...) |
| AB | = | 002 | (ABS-ABS...) |
| AR | = | 004 | ABS-(REL-REL...) |
| RE | = | 005 | (REL-REL...) |
| SL | = | 001 | (SIZE REL-SIZE REL...) |
| SA | = | 000 | (SIZE ABS-SIZE ABS...) |

GROUPS 4-6 - DISPLAY INSTRUCTIONS:

```
SETPTA  =  [DD+LS+AB ,0]   Set-Point-Absolute
SETPTR  =  [DD+LS+RE ,0]   Set-Point-Relative
DRAWTA  =  [DD+TO+AB ,0]   Draw-To-Absolute
DRAWTR  =  [DD+TO+RE ,0]   Draw-To-Relative
DRAWFA  =  [DD+FR+AB ,0]   Draw-From-Absolute
DRAWFR  =  [DD+FR+RE ,0]   Draw-From-Relative
LINAA   =  [DD+LS+AB ,0]   Line-(Absolute-Absolute)
LINAR   =  [DD+LS+AX ,0]   Line-(Absolute-Relative)
LINRA   =  [DD+LS+RX ,0]   Line-(Relative-Absolute)
LINRR   =  [DD+LS+RE ,0]   Line-(Relative-Relative)
LINIAA  =  [DI+LS+AB ,0]   Line-Indirect-(Absolute-Absolute)
LINIAR  =  [DI+LS+AX ,0]   Line-Indirect-(Absolute-Relative)
LINIRA  =  [DI+LS+RX ,0]   Line-Indirect-(Relative-Absolute)
LINIRR  =  [DI+LS+RE ,0]   Line-Indirect-(Relative-Relative)
POLAA   =  [DD+PO+AB ,0]   Polygon-Absolute's
POLAR   =  [DD+PO+AR ,0]   Polygon-Absolute-Relative's
POLRR   =  [DD+PO+RE ,0]   Polygon-Relative's
POLRA   =  [DD+PO+RA ,0]   Polygon-Relative-Absolute's
POLIAA  =  [DI+PO+AB ,0]   Polygon-Indirect-Absolute's
POLIAR  =  [DI+PO+AR ,0]   Polygon-Indirect-Absolute-Relative's
POLIRR  =  [DI+PO+RE ,0]   Polygon-Indirect-Relative's
POLIRA  =  [DI+PO+RA ,0]   Polygon-Indirect-Relative-Absolute's
STARAA  =  [DD+SS+AB ,0]   Star-Absolute's
STARAR  =  [DD+SS+AR ,0]   Star-Absolute-Relative's
STARRR  =  [DD+SS+RE ,0]   Star-Relative's
STARRA  =  [DD+SS+RA ,0]   Star-Relative-Absolute's
DOTSAA  =  [DD+DT+AB ,0]   Dots-Absolutes
DOTSAR  =  [DD+DT+AR ,0]   Dots-Absolute-Relative's
DOTSRR  =  [DD+DT+RE ,0]   Dots-Relative's
DOTSRA  =  [DD+DT+RA ,0]   Dots-Relative-Absolute's
BOXA    =  [DD+BX+AB ,0]   Box-Absolute
BOXR    =  [DD+BX+RE ,0]   Box-Relative
BOXSA   =  [DD+BX+SA ,0]   Box-Size-Absolute
BOXSR   =  [DD+BX+SL ,0]   Box-Size-Relative
SETCRV  =  [DN+LS+AB ,@0]  Curve-Set-Point
DRACRV  =  [DN+TO+AB ,@0]  Curve-Draw-To
POLCRV  =  [DN+PO+AB ,@0]  Curve-Polygon
DOTCRV  =  [DN+DT+AB ,@0]  Curve-Dots
NEWCRV  =  [DN+BX+AB ,@0]  Curve-New
```

GROUP 7 - CHARACTER DRAWING

```
DOCHAR  =  [700 ,0]        Draw Characters
```

# APPENDIX III

## A NOTE ON HOMOGENEOUS COORDINATES AND THE LDS-1

### III.1 Introduction

This note is designed as an operational, as opposed to a theoretical, note on homogeneous coordinates and the Evans & Sutherland Line Drawing System Model 1. The use of homogeneous coordinates operationally and conceptually simplifies many of the problems in presenting and manipulating three-dimensional objects with a computer graphic system. The degree of simplification gained is apparent in the airport examples discussed at the end of this Appendix. These examples are significant because they are indicative of the general class of problems which involve multiple moving bodies in three-space.

For a full LDS-1 system, the basic three-dimensional coordinates describing objects is stored in main memory in four consecutive "half-words." These four half-words represent the "homogeneous" three-space coordinate vector [X, Y, Z, W]. The first three components X, Y, Z are the normal orthoginal three-space distances from the origin of coordinates of the particular object. The fourth component, W, is a scale factor for the first three components.

The X, Y and Z components are binary 2's complement numbers arrayed about Zero=$00\ldots00_2$. The binary point, analogous to the decimal point, can be thought to be located at the user's discretion. Thus in one representation of the whole three-space, the user might be thinking of a "cube" of space "centered" at Zero and running to approximately ± Unity in each direction; if so, the user would be thinking of the binary point being located one binary place to the right of the left end of the half-word. Another natural representation with an 18-bit LDS-1 system might be a cube of space centered at Zero and running from $-2^{17} = 10\ldots0_2$ to $2^{17}-1 = 01\ldots1_2$; in this case, the binary point would be located at the right end of the half-word. Regardless of the assumed binary point, the X, Y, and Z values can still represent any scale for the object or space in question. The location assumed for the binary point is independent of this choice of scale for the object.

The W component is often stored as unity to represent a unity scale for the homogeneous coordinate. If W were half of unity, the coordinate would represent a point (or distance) twice as far from the origin. If W were Zero, the coordinate would represent a relative value. Since a relative coordinate is the difference between two absolute coordinates, this can easily be shown for coordinates with equal W's:

$$[X, Y, Z, 1] - [X', Y', Z', 1] = [\Delta X, \Delta Y, \Delta Z, 0]$$

The set of four-element homogeneous coordinate vectors that describe an object can be transformed by the LDS-1 Matrix Multiplier. There are 16 elements in this matrix and, contrary to coordinate data, they are considered to have a fixed binary point. The elements are signed fractions in 2's complement representation. Thus, the binary point is assumed to be located to the right of the left end of the half-word. Unity = $01...1_2$, is the largest positive fraction that can be represented as a matrix element. For convenience in the example matrices that follow, this is written "1."

## III.2 Conventions for the Homogeneous Coordinates

Some of the literature about homogeneous coordinates considers Z as the distance from the projection plane to the object, and W as the distance from the observer's "eye point" to the object. However, in many applications, it is inconvenient or impossible to calculate the location of the projection plane. An example is the projection screen for a pilot in an aircraft simulator; this application may need a virtual screen at infinity. In contrast to this potential problem of the location of the projection plane, the location of the eye point is known in almost all applications. The Evans & Sutherland Clipping Divider considers the Z information presented to it as the distance from the eye point to the object.

Before proceeding, a comment about orthographic projection is in order. In the "Z from the projection plane" coordinate system, the perspective presentation seen on the projection plane approaches an orthographic projection as the eye position is moved farther and farther from the plane, i. e. as $W \rightarrow \infty$. In the "Z from the eye point" system, which is used exclusively in what follows, orthographic projections are made by using a transformation matrix which makes the resulting scope coordinates depend upon W (the homogeneous coordinate scale factor), but not on Z (the distance from the viewpoint). As an interesting example, consider a star in the sky which is located infinitely far from the viewer. Since the star is infinitely far away, it has a coordinate of [X, Y, Z, 0]. If this point were orthographically projected onto a screen, it is almost certain to be projected to a point on the screen that is very far from the area of the screen that represents the viewport. In effect, the orthographic projection of the star by the Clipping Divider would entail dividing by 0. This would make the scope coordinates $X_s$ and $Y_s$ extremely large, (i. e. off the scope).

## III.3 Conventions of the Clipping Divider

In addition to the "Z from the eye point" coordinate system, three other conventions used by the Clipping Divider must also be understood. The first convention is that the Clipping Divider treats its four component vector input as if it were [X, Y, $Z_x$, $Z_y$] rather than [X, Y, Z, W]. That is, $Z_x$ is assumed to be the Z distance for X and the $Z_y$ the Z distance for Y. Since [X, Y, $Z_x$, $Z_y$] describes a single point, normally

$Z = Z_x = Z_y$ for information presented to the Clipping Divider. The transformation from [X, Y, Z, W] data stored in memory to the [X, Y, $Z_x$, $Z_y$] data presented to the Clipping Divider can be handled by the Matrix Multiplier. Examples are given at the end of this Appendix. The Clipping Divider algorithm then processes this input information to get an intermediate result [X', Y', $Z'_x$, $Z'_y$]. Following this, the algorithm divides X' by $Z'_x$ and Y' by $Z'_y$ to get the final X and Y scope coordinates to be passed to the display.

The second convention is that the Clipping Divider hardware operates as if the field of view were 90° in both X and Y. Consequently, the $Z_x$ and $Z_y$ presented as input should have been scaled to provide the desired field of view. Again, this transformation can be handled by the Matrix Multiplier as shown in the examples at the end of this Appendix. The normal procedure is to scale $Z_x$ and $Z_y$ to values that equal X and Y at the edge of the desired field of view. For fields of view less than 90°, this scaling reduces Z, and can be represented as an appropriate fractional number in the Matrix Multiplier.

The third convention is that the Clipping Divider always treats its input information in a left-hand coordinate system. Thus, positive X increases to the right and positive Y increases upward, while positive Z increases away from the eye point perpendicular to the center of the screen.

These conventions used by the Clipping Divider need cause no trouble; they can be handled by appropriate transformations made by the Matrix Multiplier. In fact, the natural way to handle all transformation information is to combine them into a single 4 x 4 transformation matrix. A matrix for the first transformation, the Clipping Divider Switching Transformation [CDST], can be written as in the top of figure AIII.2 when $Z = Z_x = Z_y$.. The matrix for the second Field of View Transformation [FVT] is shown in the bottom row of figure AIII.1. The desired field of view is defined by $\alpha°$ and $\beta°$. This transformation [FVT] can then be combined with [CDST] to get the final Switching and View transformation [SV].

Since the Matrix Multiplier can multiply matrices, [SV] can be combined with any other transformation by the Matrix Multiplier. One method is to load [SV] into the Matrix Multiplier (and probably the Data SINK for later use) as the LDS-1 starts. It can, thereafter, be combined automatically with each individual transformation which has been stored with individual picture elements. An alternate method is to use software to combine the [SV] transformation with each individual picture element's transformation before beginning the display. The first method makes the data base more "pure" and requires less software, while the second allows the LDS-1 to operate faster when displaying the picture.

## III.4 Position - Viewpoint Matrices.

An Object's Position matrix (denoted [OP]) is the 4 x 4 homogeneous coordinate matrix that specifies an object's location and orientation with respect to the origin of three-space coordinates. It is derived from concatenating the information describing the object's rotation, scaling and translation, as shown in figure AIII.2. The concatenation of a [0, 0, 0, 1] column makes the matrix square.

This resulting square [OP] matrix always has an inverse. Moreover, since the [OP] describes the object position from the origin of three-space, the inverse $[OP]^{-1}$, describes the three-space position from the origin of the object! Thus, the [OP] can be thought of as describing the "view of," and the $[OP]^{-1}$ can be thought of as describing the "view from," the object in question. Use will be made of this relationship below.

## III.5 The Airport Problem

The picture in figure AIII.3 allows several operational relationships to be written down just as the LDS-1 system will execute them. We will assume for the sake of simplicity that all viewers have the same field of view (i. e. $\alpha°$ and $\beta°$) so that there is only one [FVT], and thus only one [SV]. Other position matrices are defined as noted in Table 3.

First, what does one see from the base of the control tower (the origin of three-space coordinates) looking straight up? One sees the space, the Trans-World plane in its correct position, and the United Airlines plane in its correct position, (assuming the field of view is large enough). Thus, to start a picture, the display program could:

1) load [SV] into the DATA SINK (for later use) and the Matrix Multiplier
2) draw the objects fixed in three-space
3) multiply the [SV] matrix in the Matrix Multiplier by [UAP]
4) draw the United Airline plane
5) load the Matrix Multiplier with [SV] from SINK
6) multiply by [TWP], and draw the Trans-World plane

What does the control tower operator see? He sees the three-space and the objects just as before, except from his translated position up the Z axis and looking along a direction rotated from the three-space Z axis. This transformation is defined in figure AIII.3 as [CTP]. The program would:

1) load [SV] into the DATA SINK and Matrix Multiplier
2) multiply $[CTP]^{-1}$
3) draw the objects fixed in three-space
4) continue as in previous example

Note that there may be no reason to draw the control tower itself (which is assumed to be part of the three-space). This is especially true if none of the control tower appears to the control tower operator. Omitting the tower may save program execution time at the cost of a little more care in initially organizing the data.

What does the United Airlines pilot see? He sees the space, and TWA at the TWA location. Consequently, a program might:

1) load [SV] into SINK and Matrix Multiplier
2) multiply $[UAP]^{-1}$
3) draw the three-dimensional space
4) multiply [TWP]
5) draw the Trans-World plane

Again, this assumes that none of the United plane is visible to the United pilot.

## TRANSFORM MATRICES

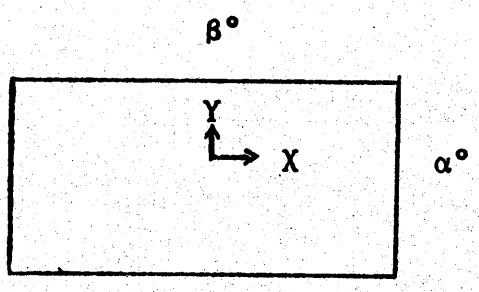
Homogeneous Coordinates — [CDST] Clipping Divider Switching Transformation — Clipper Input

$$[X, Y, Z, W] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = [X, \ Y, \ Z_x, \ Z_y]$$

[CDST] — [FVT] Field of View Transformation — [SV] Final Switching and View Transformation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \tan \alpha/2 & 0 \\ 0 & 0 & 0 & \tan \beta/2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \tan \alpha/2 & \tan \beta/2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Where the chosen angles of view represent a viewport described by:




$\alpha, \beta < 90°$

Figure AIII.1

# HOMOGENEOUS COORDINATES

COORDINATES X TRANSFORMATION = NEW COORDINATES

**3X3 TRANSFORMATION**
(ROTATION AND SCALING)

$$\begin{bmatrix} X, & Y, & Z \end{bmatrix} \times \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} X', & Y', & Z' \end{bmatrix}$$

**3X4 TRANSFORMATION**
(ROTATION, SCALING AND TRANSLATION)

$$\begin{bmatrix} X, & Y, & Z, & 1 \end{bmatrix} \times \begin{bmatrix} 3\times3 & \\ \hline j & k & l \end{bmatrix} = \begin{bmatrix} X'', & Y'', & Z'' \end{bmatrix}$$

**4X4 HOMOGENEOUS TRANSFORMATION**
(ROTATION, SCALING AND TRANSLATION)

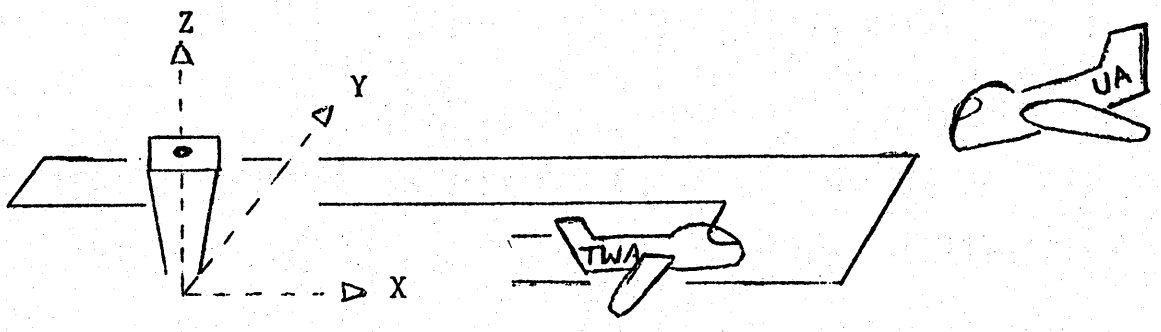
$$\begin{bmatrix} X, & Y, & Z, & 1 \end{bmatrix} \times \begin{bmatrix} & 3\times3 & & 0 \\ & & & 0 \\ & & & 0 \\ \hline 3\times1 & & & 1 \end{bmatrix} = \begin{bmatrix} X'', & Y'', & Z'', & 1 \end{bmatrix}$$

Figure AIII.2

THE AIRPORT PROBLEM

Origin of the three-space at base of control tower.

Origin of each plane assumed to be at pilot's eye point.




Associated Matricies

[UAP] = United Airlines Position. Matrix giving the position and orientation in three-space of the United Airlines plane from the origin of three-space coordinates.

[TWP] = Trans World Airlines Position. Matrix as above.

[CTP] = Control Tower observer's Position. Matrix as above.

Figure AIII.3

# APPENDIX IV

## QUICK REFERENCE MATERIAL

The following pages contain information which has been extracted from earlier chapters and is intended for use as quick reference material. In most cases, the information is a direct copy of figures or data located elsewhere in this manual. In the case of figures, the figure number has been retained so that the reader can direct himself back to the chapter from which the figure was extracted if he needs more information.

Included are the following figures and pages:

# APPENDIX V

## LIS INSTRUCTIONS

With special hardware modification an additional instruction is available in group 0. The "LIS" instruction is a load immediate of the indicated register, but in addition the address used to fetch the instruction (+1) is stuffed into the RAR. The LIS instruction takes the form shown in figure AV.1. An LIS of the PC is defined in the mnemonics as RJMP.

To illustrate the usefulness of these instructions, let us consider an instruction sequence such as:

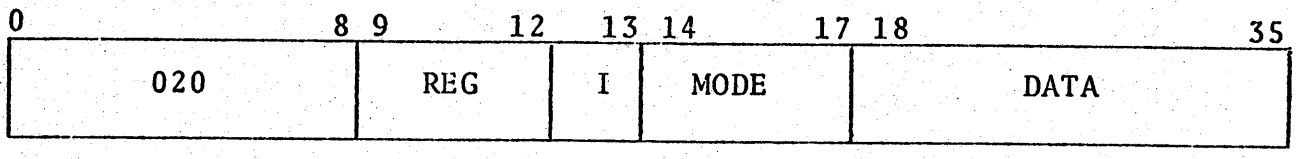
```
        RJMP N1(XQTM)
        POLAA RCR,-5 (RPTM)
        DATA1
         :
         :
        DATA5
         :
         :
N1:     RJMP N2(XQTM)
        POLAA RCR,-7(RPTM)
        etc.
```

This instruction sequence facilitates a chain of polygon instructions which do not have to be contiguously located in memory. The RJMP loads the PC with N1 and then puts LOC + 1 in the RAR. Since execute mode is indicated, the Channel Control uses the RAR for the next instruction fetch. The RAR is incremented as it is used for this instruction fetch so it is pointing to the data for the polygon instruction. When the count in the RCR goes positive the Channel Control again enters program mode and uses the PC (which was loaded with N1 by the RJMP) for the next instruction fetch. This sequence can then be repeated to chain a group of polygon instructions.

Mnemonic: LIS

Assembler definition: [020000000000]

Structure:




Function: Load immediate REG if I=0.
LOC (of instruction) +1⟶RAR

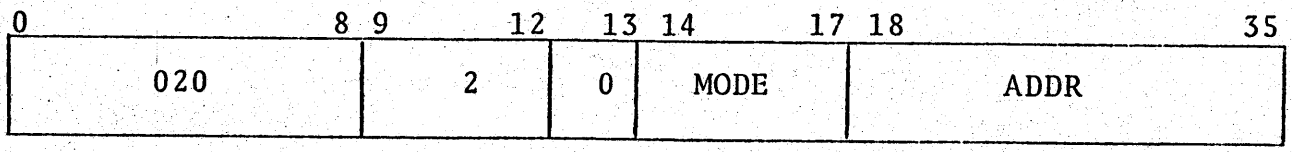
Format: LIS REG,DATA(MODE) ;or
LIS @(MODE)

where: REG is the register
DATA is the new data
MODE is the new mode (optional)
@ sets the I bit

Mnemonic: RJMP

Assembler definition: [LIS PC,0]

Structure:




Function: ADDR PC
LOC (of instruction) +1⟶RAR

Format: RJMP ADDR(MODE)

where: ADDR is the address to be loaded into PC
MODE is the new mode (optional)

Figure AV.1