STD6EX

STD6LE - link loader

MAP - memory map

default BOUND 1000_8 - 16 K

BUILD

BOUND = LOW, HIGH

EVANS & SUTHERLAND COMPUTER CORP.

LINE DRAWING SYSTEM MODEL 2

SYSTEM REFERENCE MANUAL

MARKED UP FOR HALFTONE SYS.

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

August 1, 1971

901002-100

COPY ____

Prepared by: Russell Athay

TABLE OF CONTENTS

SYSTEM OVERVIEW

1.1 System Configuration

The LDS-2 is a general purpose computer with specialized facilities for graphic processing. In the shared memory configuration the LDS-2 operates as a second processor which shares memory with a host computer. In this configuration the LDS-2 is an independent processor in that it accesses and executes its own programs, but the LDS-2 is dependent upon the host computer for such functions as I/O and the regulation of its operation (i.e., starting and stopping the LDS-2, scheduling users, etc.). Figure 1.1 shows the configuration of the LDS-2. The following units make up the LDS-2:

The Channel Control. The Channel Control accesses memory to provide the instructions and data needed by the LDS-2. The Channel Control executes all of the general purpose processing instructions and interprets display instructions and provides commands and data to the display processing pipeline devices.

The Matrix Multiplier. The Matrix Multiplier can rotate, translate and scale the drawing to be displayed. The Matrix Multiplier also can iterate sets of difference equations to draw curves and families of 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 Clipping Divider also performs three-dimensional perspective division.

The Line Generator and Display Scopes. The Line Generator converts the digital specification of end-points into analog sweep voltages which are used to drive the deflection systems of the Display Scopes.

1.2 General Purpose Processing

The LDS-2 has a large and versatile instructions set, its own internal high-speed register memory, and facilities for interpreting complex data structures. Instructions are provided to perform the following tasks:

Arithmetic and logical operations
Shifting, masking, and bit manipulation
Comparisons and conditional skips
Program flow control and stack manipulation

LDS-2 DISPLAY SYSTEM CONFIGURATION

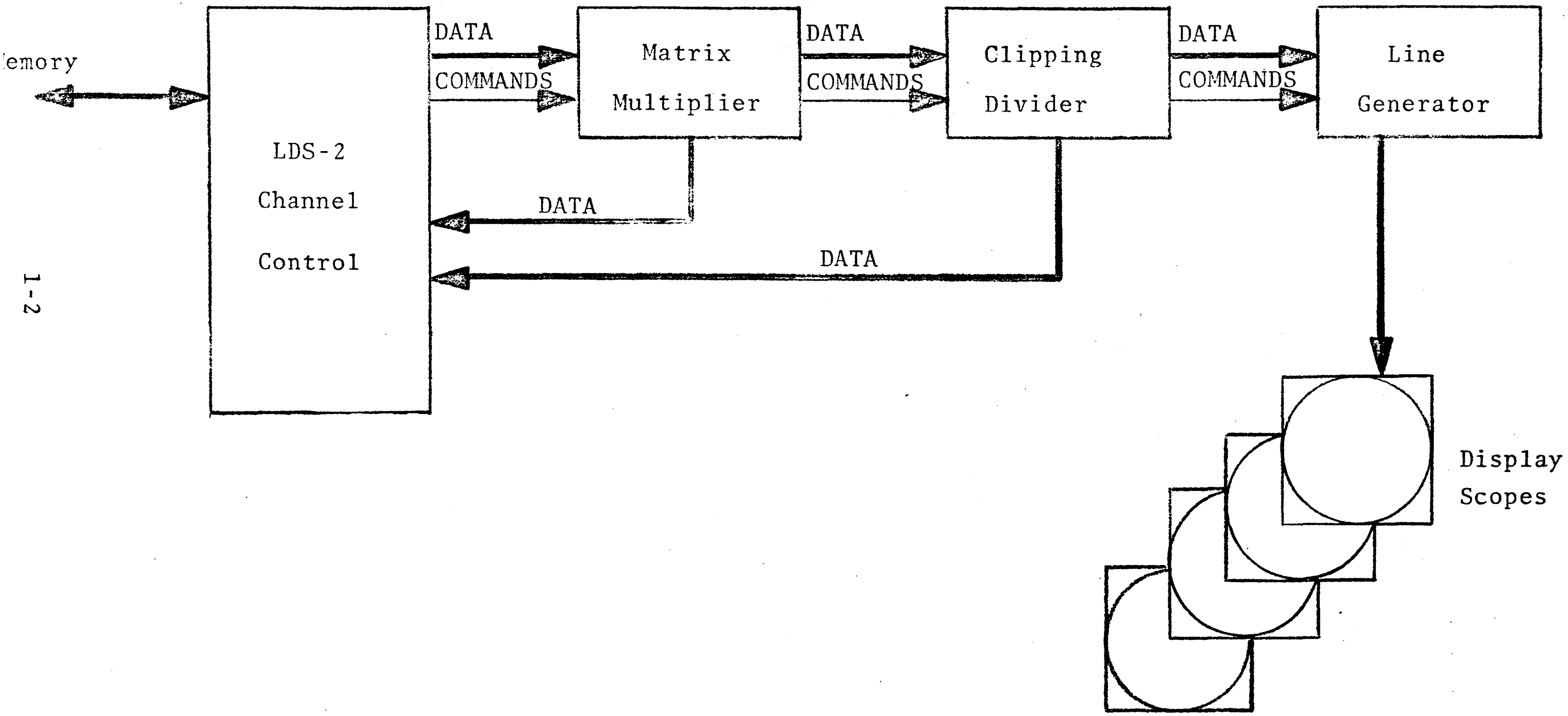


Figure 1.1

The LDS-2 Channel Control has a high-speed register memory which is composed of sixteen registers. While all but four of these registers are used for special functions, all registers may be manipulated with equal ease, and when the special function to which a register is dedicated is not being used, that register may be used as a general purpose accumulator.

The LDS-2 provides facilities for direct, indirect, and indexed addressing, but it is also a stack machine with very powerful and versatile stack manipulation facilities. Special stacks are operated to hold return locations and parameter information from the display processing pipeline. The user may also set up and operate additional general purpose stacks.

1.3 Graphic Processing

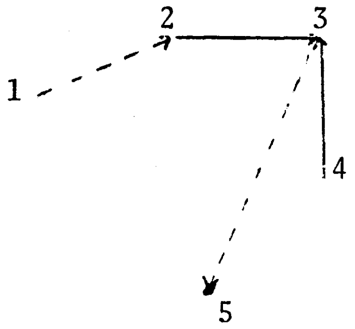
In addition to its general computing capabilities, the LDS-2 can interpret drawing definitions, perform graphic transformations on the drawing and display a picture on the Display Scope. For the purposes of this manual, the following words will take on special meanings to avoid confusion.

> Drawing. The drawing is the definition stored in memory which consists of two- or three-dimensional coordinate data and display instructions which determine how these coordinate values should be interpreted and how the points should be connected. The drawing is in "page coordinates."
>
> Picture. The lines and dots which finally appear on the Display Scope are referred to as the picture. The picture is in "scope coordinates."
>
> -2^{N-2} To $+2^{N-2}$
>
> Page Coordinates. The page is a virtual drawing space which stretches from -~~2^{22}~~ to +~~2^{22}~~ in each coordinate axis. The LDS-2 performs all arithmetic and graphic processing using two's complement arithmetic, so one may think of the page as a fixed point, two's complement drawing space. Since the page is extremely large, no checking is done to detect overflow of the page boundaries.
>
> -2^{11} To $+2^{11}$
>
> Scope Coordinates. Scope coordinates are centered about zero and stretch from -~~2^{15}~~ to +~~2^{15}~~ in X and Y. Before the drawing is displayed, it is mapped into scope coordinates and becomes the picture.

1.3.1 Drawing Instructions

The drawing instructions generally result in some movement of the beam on the scope. The upper half of Figure 1.2

DRAWING OPERATIONS



Basic Drawing Operations

1 is current point

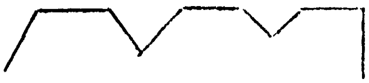
"Set point" to 2 (2 becomes current point)

"Draw to" 3 (3 becomes current point)

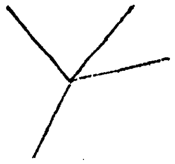
"Draw from" 4 (3 remains current point)

"Dot" 5 (5 becomes current point)

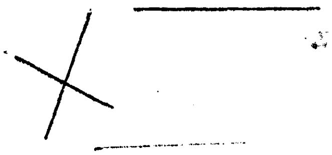
Complex Drawing Operations



"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. 2

illustrates the basic drawing operations that are available. These operations are done in relation to the "SAVE point" which indicates the current position of the beam on the scope. It is also possible to initiate a repeating series of the basic drawing operations with a single instruction, as shown on the lower half of Figure 1.2.

1.3.2 Data Forms

The coordinate data may be interpreted either as an absolute specification of the endpoint location, or as one of two forms of displacement specifications. The display instructions specify how the data are to be interpreted. Figure 1.3 illustrates the three manners of interpreting the coordinate data. "Absolute" data simply specify the position of the endpoint. "Relative" data are taken as an offset from the "current point." And "variable origin" causes the data to be taken as an offset from a user-specified "origin" which is held in the registers of the Channel Control.

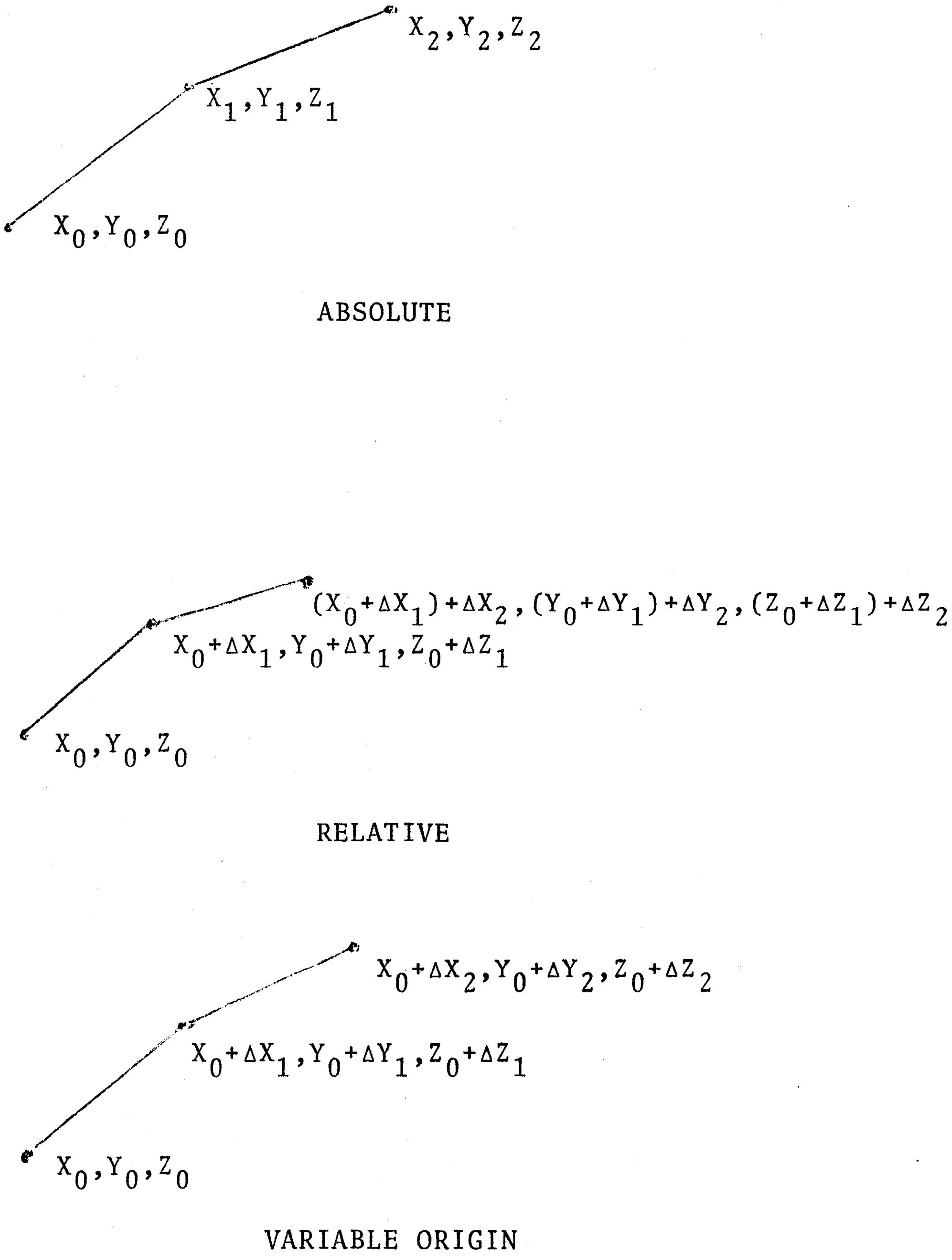
1.3.3 Dimension Modes and Coordinate Data Storage

The LDS-2 is always in one of four dimension modes, and these modes affect how many words of data are fetched for display instructions (both drawing instructions and pipeline register load/unload instructions). The two-dimensional mode causes the LDS-2 to pick up two contiguous words of data which are interpreted as the X coordinate and the Y coordinate. In "homogeneous" mode (sometimes referred to as 4D) the LDS-2 picks up four words of data which are interpreted as X Y Z and W. This data format is known as homogeneous coordinates, where W is the homogeneous element and is used as a scale factor. Data fed through the Matrix Multiplier should be in homogeneous coordinates. (See Appendix III for a description of homogeneous coordinates and their usage.) If the Matrix Multiplier is turned off, the four words of data fetched by the LDS-2 will be interpreted as X Y Zx and Zy, where Zx and Zy are generally the same. This is the form in which the Clipping Divider expects data.

Two special three-dimension modes are provided to allow more compact storage of data. These modes apply only to drawing instructions (i.e., pipeline load/unload instructions still pick up four words). "Matrix Multiplier three-dimensions" (MM3D) causes the LDS-2 to pick up three words which are interpreted as X Y and Z. The LDS-2 then supplies the fourth word, which is the fractional approximation for "1" (2^{23}-1) to serve for the "W" element. Since W is often "1", when using homogeneous coordinates, MM3D may often be used to save storage. MM3D should only be used, however, if the Matrix Multiplier is active.

If the Matrix Multiplier is not active and data are being fed directly to the Clipping Divider, "Clipping Divider

DATA FORMS


X_2, Y_2, Z_2
X_1, Y_1, Z_1
X_0, Y_0, Z_0
$(X_0+\Delta X_1)+\Delta X_2, (Y_0+\Delta Y_1)+\Delta Y_2, (Z_0+\Delta Z_1)+\Delta Z_2$
$X_0+\Delta X_1, Y_0+\Delta Y_1, Z_0+\Delta Z_1$
X_0, Y_0, Z_0
$X_0+\Delta X_2, Y_0+\Delta Y_2, Z_0+\Delta Z_2$
$X_0+\Delta X_1, Y_0+\Delta Y_1, Z_0+\Delta Z_1$
X_0, Y_0, Z_0

ABSOLUTE

RELATIVE

VARIABLE ORIGIN

Figure 1.3

three dimensions" (CD3D) may be used. This mode also causes the LDS-2 to pick up three words of data, but in this case the fourth word provided by the LDS-2 is a copy of the third word to give X Y Z Z, which is the form that the Clipping Divider expects.

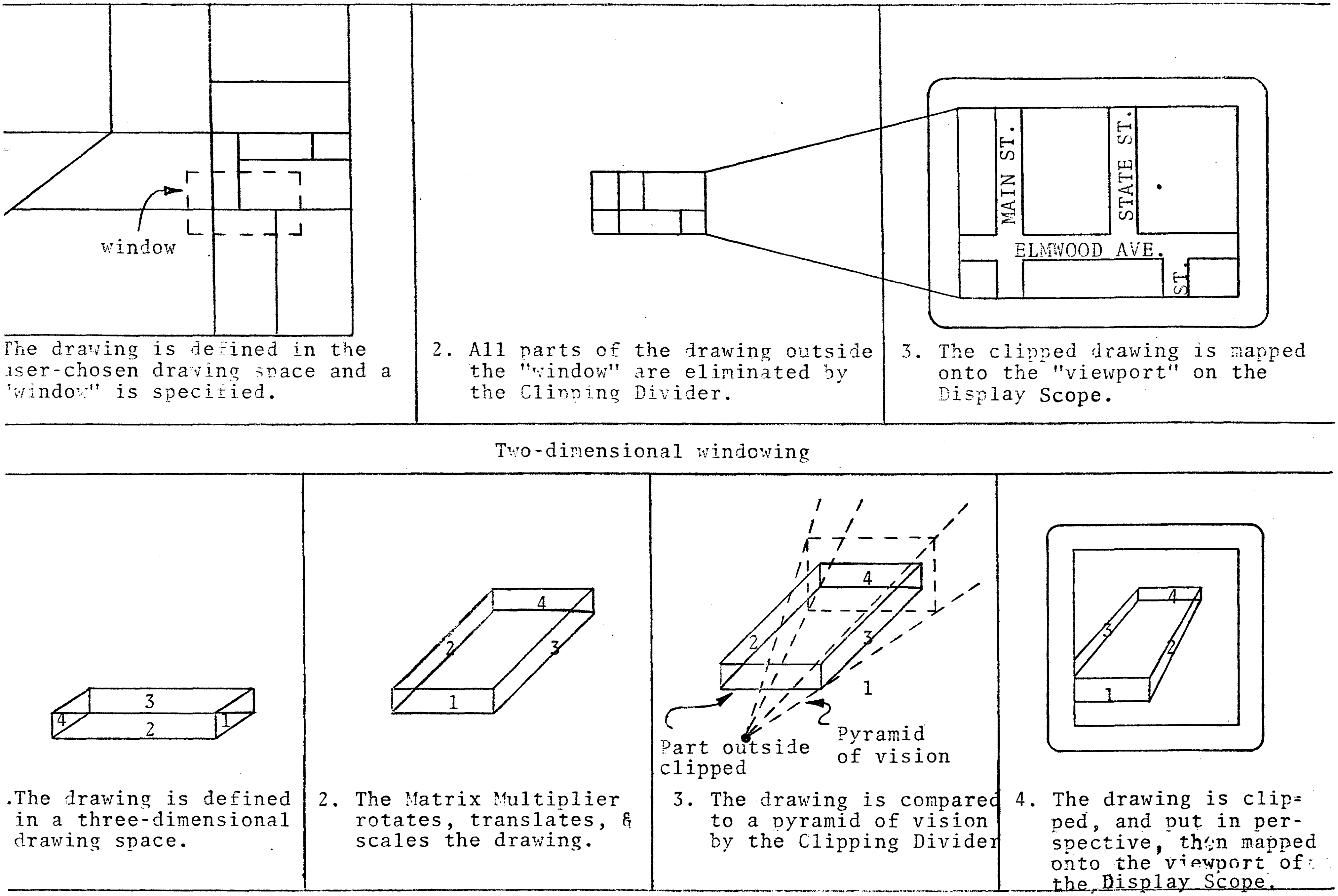
1.3.4 The Display Processing Pipeline Units

The display processing pipeline units perform graphic transformations on the coordinate data as they pass down the pipeline. The Matrix Multiplier and Clipping Divider contain their own internal storage registers to hold the parameters that are used in the graphic transformations. For instance, the Matrix Multiplier holds four 4 x 4 matrices. When the Matrix Multiplier is active, the coordinate data are multiplied by the values in the first matrix as they pass down the pipeline. These matrix multiplications may be used to rotate, translate, and scale the drawing. Similarly, registers in the Clipping Divider hold the "window" and "viewport" values which are used to map the coordinate data from page coordinates into scope coordinates. Details of the operation of the pipeline devices are given in Chapters 3, 4 and 5. Figure 1.4 gives a pictorial representation for the graphic processing performed by the LDS-2.

Because the parameters for the graphic processing are held internally by the devices which perform this processing, the data base can remain "pure;" that is, motion and other transformations can be implemented by changing the parameters in the pipeline registers rather than changing the coordinate data as it is stored in memory. The registers of the pipeline devices may be loaded or unloaded with these parameters by LDS-2 instructions.

1.4 Programming

The LDS-2 assembles its own programs and has its own assembly language (see Chapters 6 and 7 of this manual). Fortran callable support routines which generate code for the LDS-2 are also provided as an option. These routines allow the Fortran user to make use of the graphic capabilities of the LDS-2 through Fortran calls. The Fortran support routines are discussed in Chapter 8.



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

.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 1.4

CHAPTER 2

THE CHANNEL CONTROL

2.1 Function

The Channel Control functions as a general purpose processor and as the control unit for the LDS-2. The Channel Control has general computing capabilities which allow it to assemble its own programs and process both graphic and non-graphic data. But in addition to these general facilities, the Channel Control has special graphic capabilities which allow it to interpret display programs and to control the display processing pipeline units of the LDS-2.

2.2 Structure

A block diagram of the Channel Control is shown in Figure 2.1. The Channel Control operates out of the memory of the host computer by providing memory addresses and then either accepting or transmitting data or instructions. The Arithmetic and Logical Unit (ALU) provides the Channel Control with the ability to do general purpose data processing. The Channel Control has its own high-speed I/O buss which facilitates the communication between the Channel Control and several registers which function as I/O units to the Channel Control. These registers are described in Section 2.5.

2.2.1 Registers of the Channel Control

The Channel Control is organized around sixteen registers which form a high-speed register memory. Four of these registers serve as general purpose accumulators while the other twelve are dedicated to special functions. However, all registers may be manipulated with equal ease and all registers may be used with most instructions. It is thus possible to use the dedicated registers as general purpose accumulators if the function to which they are dedicated is not being used. For instance, if the system is not returning processed data from the pipeline back into memory, the WP and WC registers can safely be used as general purpose accumulators. Table 2.2 lists the registers of the Channel Control and briefly describes their functions. The use of these registers is more fully described in the course of this chapter.

2.2.2 Memory Addressing

The LDS-2 divides memory into pages of fixed length and fixed location. A page is $2^{(n-8)}$ words long where n is the number of bits per memory word in the system. For a 24-bit LDS-2 the page is 64K words long so paging considerations generally disappear. The address specified in addressing

STRUCTURE OF THE CHANNEL CONTROL

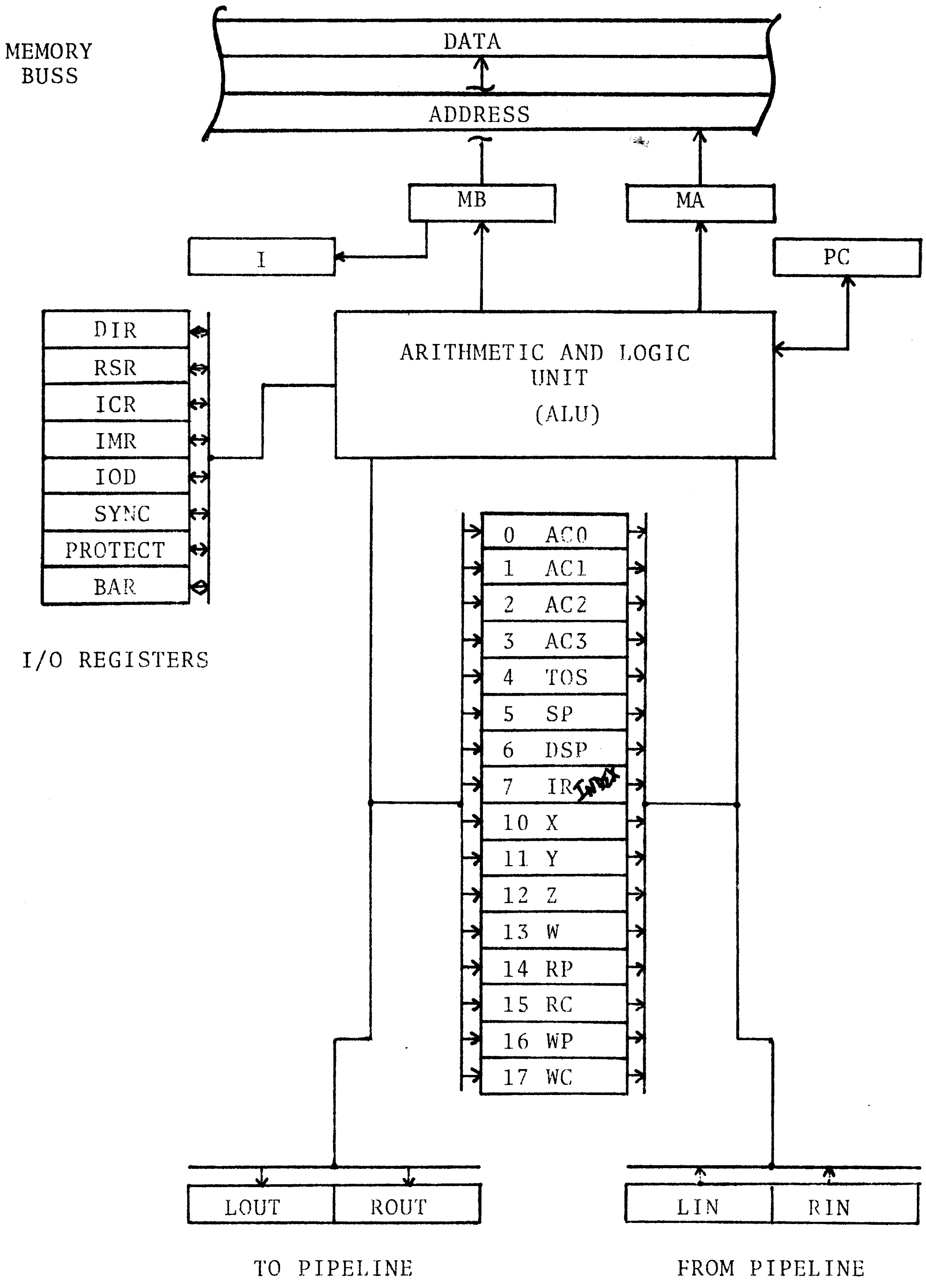


Figure 2.1

CHANNEL CONTROL REGISTER MEMORY

Register	Mnemonic	Dedicated Use	Functional Characteristics
0	AC0	undedicated	general purpose accumulator
1	AC1	undedicated	general purpose accumulator
2	AC2	undedicated	general purpose accumulator
3	AC3	undedicated	general purpose accumulator
4	TOS	Top Of Stack	top element of SP stack
5	SP	Stack Pointer	decrements before writing in the old PC for a pushjump
6	DSP	Data Sink Pointer	increments before writing in data from a sink operation
7	IR	Index	index register
10	X	X current point	updated automatically by drawing instructions
11	Y	Y current point	updated automatically by drawing instructions
12	Z	Z current point	updated automatically by drawing instructions
13	W	W current point	updated automatically by drawing instructions
14	RP	Read Pointer	points to the location of coordinate data tables
15	RC	Read Counter	increments once per coordinate point read through the RP
16	WP	Write Pointer	increments after writing data from pipe.
17	WC	Write Counter	increments once per word written through WP

Table 2.2

instructions is taken as an address within the page and is added to the 8-high order bits of the Program Counter (PC) to obtain the effective address. Direct addresses may not cross page boundaries (i.e., they must be within the current page).

Indirect addressing may be specified with all addressing instructions. When the indirect bit of the instruction word is set, the effective address is the contents of the location directly addressed. The directly addressed location must be within the current page, but the indirect address may be anywhere within the total addressing space. Only one level of indirection is available.

Some addressing instructions may also be indexed. Indexing causes the contents of the Index Register (IR) to be added to the address specified in the instruction in order to calculate the effective address. Since the IR is a full word length register, the effective address may lie anywhere within the total addressing space. If both indirection and indexing are specified, the indirection is performed before the indexing. Examples of the addressing scheme of the LDS-2 are given in Section 7.4.

2.3 General Computing Facilities

The LDS-2 has a large and versatile instruction repertoire which makes it convenient for a large variety of general purpose processing tasks. The availability of the sixteen registers in register memory and the stack mechanism add to the processing power of the LDS-2.

2.3.1 General Purpose Instructions

The general purpose instructions of the LDS-2 provide the following functions:

Load and store the Channel Control registers from memory or other registers

Program control (jump, pushjump, and execute)

Arithmetic and logical operations

Increment and decrement registers and skip on condition

Compare two registers and skip on condition

Arithmetic, logical and circular shifts

Masking

Stack control (push and pop with increment or decrement)

The individual instructions are explained in detail in Chapter 7, but it is useful to keep these general functions in mind while attempting to understand the LDS-2.

2.3.2 The Stack

The LDS-2 operates two special purpose stacks and allows the user to operate additional general purpose stacks. One of the special purpose stacks is known as the "data sink" and is used to store parameters from the pipeline registers. The data sink is described in Section 2.4.5. The other special purpose stack is used for storing return locations (i.e., old PC values). This stack operates in a special way because the top element of the stack is held in the Channel Control's TOP OF STACK (TOS) register rather than in memory. Thus, the STACK POINTER (SP) points not to the top of the stack, but rather to the last element stored in the memory portion of the stack, which is effectively the second element in the stack. Since the top element in the stack is in the TOS register it is immediately available to the user. When pushing the old PC onto this stack, the following process occurs:

$SP-1 \rightarrow SP$	decrement the stack pointer;
$TOS \longrightarrow C(SP)$	push the TOS;
$PC \longrightarrow TOS$	save the PC.

When the stack is popped to return the old PC the reverse path is followed:

$TOS \longrightarrow PC$	return old PC;
$C(SP) \rightarrow TOS$	pop into the TOS;
$SP+1 \rightarrow SP$	increment the stack pointer.

This whole process is invisible to the user so that he may simply consider the TOS as the top element in the stack.

In addition to these two special purpose stacks, the LDS-2 provides the user with convenient facilities for implimenting other stacks which may be used and manipulated under program control. Any of the Channel Control's registers may be used as a stack pointer with which to push the value held in another register onto the stack, or to pop an element off of a stack back into a register. This "stack pointer" may be incremented or decremented either before or after the register is pushed or popped, so that the user has the full range of possibilities for stack control. Because the LDS-2 has such convenient stack-control facilities, it is often best to treat the LDS-2 as a stack machine.

2.4 Graphic Facilities of the Channel Control

In addition to its general purpose computing capabilities, the LDS-2 Channel Control has special facilities for interpreting display-oriented instructions and controlling the LDS-2 display processing pipeline.

2.4.1 Display Instructions

The display instructions of the Channel Control fall into two groups:

> Drawing Instructions. The drawing instructions result in the transmission of the coordinate data to the processing pipeline. The drawing instructions define the topology of the drawing.
>
> Pipeline Load/Unload Instructions. The display processing pipeline units contain parameter registers. The values in these registers are used to process the coordinate data and thus affect the picture that is displayed. The Channel Control loads and unloads these registers either singly or in groups.

All of the display instructions require the Channel Control to generate a command for the pipeline and provide the necessary data. The Channel Control can fetch this data from memory or from its own internal registers.

2.4.2 The X, Y, Z, and W Registers

The X, Y, Z, and W registers of the Channel Control maintain the coordinates of the current point which is used as the base for relative and variable origin drawing instructions. A relative drawing instruction causes the incoming data to be added to the values in these registers before it is sent down the pipeline and the contents of the registers to be updated to the computed value of the new point. Variable origin instructions also cause the additions to be performed, but the contents of the registers are not updated, so that the next point will also be relative to the "variable origin."

The point held in the X, Y, Z, and W registers of the Channel Control usually corresponds to the "current point" held in the SAVE registers of the Clipping Divider. When processing a "variable origin" instruction, however, the X, Y, Z, and W registers are not updated in order to make all data relative to the "variable origin." The SAVE registers of the Clipping Divider are updated, however, thus at the end of a variable origin sequence the two sets of registers will contain different values. Because of this, it is a good idea to follow all variable origin instructions with either a "setpoint," a drawing operation in absolute mode, or another variable origin operation.

Note, that the relative pipeline load instructions do not use the X, Y, Z, and W registers as a base. For these instructions, data are sent to the pipeline in relative form and converted by the pipeline units themselves.

2.4.3 Data Fetching for Display Instructions

Addresses for the coordinate data for drawing instructions may come from one of two sources. The single point drawing instructions (see Section 7.14) specify an address as part of the instruction word. This address may be either direct or indirect and may be indexed (remember that indexing is performed after indirection). The table draw instructions (see Section 7.14) rely on the contents of the READ POINTER (RP) for the address. The contents of the RP may be used either as the direct address or as an indirect address which contains the effective address. If indirection is specified, indexing is also available, but if indirection is not specified (i.e., the contents of the RP are taken as the direct address), then indexing may not be specified. When indirection and indexing are specified, the contents of the INDEX REGISTER (IR) are added to the contents of the word addressed by the RP, and the result is used as the effective address. The pipeline load/unload instructions (see Section 7.13) rely on the RP just as the table draw instructions, but only direct addressing is available.

The RP is incremented after each use so that it can step through a contiguous table of data. The RP may be initialized to the beginning of a new table by loading it with the appropriate address.

The number of words of data fetched by the display instructions depends on the dimension mode of the Channel Control. The Channel Control has four modes:

Two Dimensions. In 2D two contiguous words of data are fetched which represent X and Y if the data are interpreted as coordinate data.

Three Dimensions for the Clipping Divider. This mode is abbreviated as CD3D. Three words are fetched for each point which represent X, Y, and Z. A fourth word is supplied to the pipeline by copying the last word which gives X, Y, Z, Z. This is the form that the Clipping Divider expects. Pipeline load/unload instructions behave as if the LDS-2 were in homogeneous mode.

Three Dimensions for the Matrix Multiplier. This second special three-dimensional mode (abbreviated MM3D) also fetches three words of data per coordinate point. In MM3D, however, the fourth word is supplied as the fractional representation of "1" (37777777) to give X, Y, Z, "1" which corresponds to the homogeneous representation with the homogeneous element equal to "1". Pipeline load/unload instructions behave as if the LDS-2 were in homogeneous mode.

Homogeneous Mode. In homogeneous mode four words of data are fetched for each element. If the data are interpreted as coordinate data, these four words represent X, Y, Z, and W, where W is the homogeneous element.

It is very important to remember that the dimension mode of the LDS-2 affects all display instructions. Special care must be taken when using pipeline load/unload instructions or incorrect data will be loaded into the pipeline registers. The pertinent considerations are outlined in detail in Section 7.13 dealing with these instructions.

2.4.4 Repeat 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 a "repeat" drawing instruction is recieved which indicates a "draw to," "draw from," "polygon," "star," "lines," or "dots" operation, the Channel Control automatically generates the appropriate series of basic drawing instructions. Finite-state machines within the Channel Control update the command, so that a single repeat instruction causes a series of drawing instructions to be sent down the pipeline. The drawing sequences and absolute/relative/variable origin combinations that are available with these instructions are discussed in Section 7.14. Pipeline load/unload instructions are inherently repeat. The address of the register loaded or unloaded is incremented after each iteration, so that a series of registers may be loaded or unloaded with a single instruction.

The iterations of the repeat instructions are counted by the READ COUNTER (RC). The RC is initialized with the negative (two's complement) of the number of elements (e.g., the number of coordinate points or the number of registers) and is incremented once after each data element has been fetched and passed to the pipeline. When the count reaches zero, the process is stopped and another instruction is fetched. If the count is initially zero, only one iteration will be performed. The count will never increment past zero and, thus, should never contain a positive number unless it was loaded with a positive number initially.

When the RC is not being used for repeat mode instructions, or when no other registers are available, it is convenient to use the RC as a counter for other purposes. The programmer can increment (or decrement) the counter under program control and test its results for zero. It is, of course, also possible to do this with any other of the Channel Control's internal registers.

2.4.5 The Data Sink

A special stack mechanism called the "data sink" is used to store information from the registers of the pipeline units. The DATA SINK POINTER (DSP) maintains the address of the last element written into the data sink. When pipeline registers are "sinked," the new information is written into memory and then the DSP is incremented. This information may then be "retrieved," in which case the DSP is decremented and then the register is reloaded. For retrieval operations the register addresses sent down the pipeline are decremented rather than incremented for repeat instructions, so that data are returned in the proper order.

2.4.6 Returning Output to Memory

The processed output of the arithmetic devices may be returned to memory for use in further processing or for output to remote terminals. When one of the pipeline units has data ready to return to memory, it signals the Channel Control which stops its normal operation and records the data. The WRITE POINTER (WP) of the Channel Control is used to provide the memory address for recording the processed output. Since the WP is incremented after each use, the data are recorded in a contiguous table. The length of this table may be limited by loading the WRITE COUNT (WC) with the negative (two's complement) of the desired length of the table. When the WC reaches zero, the LDS-2 will be interrupted if the appropriate interrupt bit is enabled (see Section 2.5).

2.5 The I/O Structure

The Channel Control contains eight registers which are treated as I/O devices and manipulated with "input/output transfer" (IOT) instructions. IOT instructions are also used for special functions. All of the IOT instructions, except those indicated, are legal only when the LDS-2 is in executive mode.

The Channel Control is either in executive mode or user mode. In executive mode, all the implemented IOT instructions are legal, and the "permit" bits for scope selection (see Section 4.8) may be changed. Whenever an interrupt is received from either the LDS-2 itself or the host computer, the Channel Control goes to executive mode. The Interrupt Service Routine resets user mode before transferring control back to the user.

2.5.1 Status Registers

The DIRECTIVE register and REPEAT STATUS register hold information which controls the operation mode of the LDS-2 and the functioning of the pipeline devices. These are the only two registers available to the user. The DIRECTIVE register holds the dimension mode for the LDS-2, controls whether the pipeline devices are active, and contains status flags which are set by the pipeline.

DIRECTIVE

NEW BITS

8 SWITCHES NORMAL (READ ONLY)

9 VIDEO SETTLED (READ ~~ONLY~~ ONLY)

10, 11 } SURFACE BITS
00 → NON SURFACE
01 → SURFACE
11 → SMOOTHED, SURFACE

Bits	Function
0-1	Unused
2	Matrix Multiplier Active
3	Clipping Divider Active
4	No Overlap (i.e., each line is completely processed by all the pipeline devices before the next line is begun)
5-6	Dimension modes 00 2D 01 Homogeneous mode (4D) 10 MM3D (X Y Z with an assumed "1") 11 CD3C (X Y Z with a copy of the Z)
7	Interrupt on HIT
N-3 ~~21~~	HIT (from the Clipping Divider)
N-2 ~~22~~	Area In Common (from the Clipping Divider)
N-1 ~~23~~	Settled (i.e., all of the pipeline units have finished processing, pending data, and are waiting for input)

The REPEAT STATUS REGISTER (RSP) holds the pipeline load/unload and drawing commands that are sent down the pipeline, and is updated by the normal operation of the Channel Control. The RSP makes it possible to interrupt a repeat drawing or load/unload sequence. If during the time interrupt is being serviced other drawing instructions will be executed, the RSR should be saved and then reloaded to restore the user. If the interrupt results in going to a new user, the repeat bit of the RSR must be cleared; otherwise, the first load/unload or drawing instruction executed by the new user will use the old RSR rather than the information in the instruction. The appropriate actions are taken by the LDS-2 Interrupt Handler, so that the user does not have to worry about the RSR.

REPEAT STATUS REGISTER

Bits	Function for Load/Unload	Function for Drawing
0-1	Unused	Unused
2-4	Instruction Type (011 = load/unload)	Instruction Type (100 = drawing)
5-6	Load/Retrieve/Store/Sink	---
5-7	---	Present state of drawing operation finite-state-machine (FSM1)
7, 16-18	Device and Manner	---
16-18	---	Present state of data form finite-state-machine (FSM2)
N-5 to N-2 19-22	Address of Pipeline Register	---
N-1 ~~23~~	Repeat	Repeat

2.5.2 Interrupts

The LDS-2 has a two-level interrupt system. High-level interrupts come only from the host computer and cause the execution of a hard-wired address. Low-level interrupts may be caused by a variety of internal conditions which the LDS-2 has detected. These interrupts also cause the execution of a hard-wired address which contains a "pushjump" to the Interrupt Handler. The condition which caused the interrupt will have set a bit in the INTERRUPT CONDITIONS REGISTER (ICR). The bits in the ICR are masked against the bits in the INTERRUPT MASK REGISTER (IMR). If the interrupt bit is set and the mask bit is set, the LDS-2 will be interrupted. The Interrupt Handler interrogates the ICR to determine the cause of the interrupt, so it can take appropriate action. If the Interrupt Handler returns control to the user, it is first necessary for it to decrement the TOS in order to return the instruction which was interrupted rather than the next instruction.

INTERRUPT CONDITIONS REGISTER

Bits	Meaning
6	Scope Protection Violation (TUT TUT FORBID)--Note that there is no mask for this bit.
7	Memory Protection Violation
N-7 ~~17~~	Unimplemented Instruction (no mask)
N-6 ~~18~~	Nonexistent Memory
N-5 ~~19~~	Nonexistent I/O Device
N-4 ~~20~~	Real Time Clock
N-3 ~~21~~	Positive Write-Count Register (table overflow)
N-2 ~~22~~	Overflow (caused by an arithmetic instruction)
N-1 ~~23~~	Parity Error

INTERRUPT MASK REGISTER

Bits	Meaning
7	Memory Protection Violation
N-8 ~~16~~	High-level Interrupt Mask
N-7 ~~17~~	Low-level Interrupt Mask
N-6 → ~~18~~	Nonexistent Memory
N-5 → ~~19~~	Nonexistent I/O Device
N-4 → ~~20~~	Real Time Clock
N-3 → ~~21~~	Positive Write-Count Register
N-2 → ~~22~~	Overflow
N-1 → ~~23~~	Parity

When the LDS-2 is in user mode, most of the I/O devices are not accessible and are treated as "non-existent." The lower 8 bits of the device code of an illegal IOT are saved in the I/O DEVICE CODE ERROR REGISTER. If the interrupt mask is set, an interrupt will then be initiated. When the Interrupt Handler has determined that a nonexistent I/O device caused the interrupt, it checks the I/O DEVICE CODE ERROR REGISTER. The Interrupt Handler can then decide what to do on the basis of the information in this register. This mechanism provides a convenient communication between the user and the Interrupt Handler. For example, when the user's program needs input/output from the host computer, it can make the request by executing a specified "illegal" IOT (see Section 7.12).

2.5.3 Real Time Clocks

Four real time clock sources are available on the LDS-2. The LDS-2 itself has both a 60-cycle/second clock and a clock controlled by a variable potentiometer on the control panel which can be set between 10 and 100 cycles/second. In addition to these, the clock from the host computer is available and a clock from an external synchronization source. The selection of these clocks is made by setting the SYNC MASK REGISTER. This can only be done in executive mode.

SYNC MASK REGISTER

Bits	Function
N-5 ~~19~~	External Sync
N-4 ~~20~~	Real Time Clock from Host Computer
N-3 ~~21~~	60 Hz Real Time Clock
N-2 ~~22~~	Adjustable Clock

2.5.4 Memory Protection and Relocation

For an LDS-2 which is interfaced to a host computer ~~SEL-840, 512 word pages~~ can be protected. Each user is assigned an upper and lower bounds. The upper 8 bits of the bounds are loaded into the protection register.

PROTECTION REGISTER

Bits	Function
16-23	Lower Bounds
8-15	Upper Bounds

In order to facilitiate the passing of addresses between the LDS-2 and the host computer ~~SEL-840~~, the LDS-2 has been equipped with a BANK ADDRESS REGISTER (BAR), which is loaded at initialization with the same contents as the host computer's ~~SEL-840's~~ BAR (except for the first two quarters which are reserved on the SEL-840). It is thus possible to pass addresses from software on the ~~SEL-840~~ SHO-SAT to software on the LDS-2 without having to worry about BAR relocation. The BAR on the LDS-2 is active only when the LDS-2 is in user mode. In executive mode, addresses are interpreted as absolute.

BANK ADDRESS REGISTER

Bits	Function
2-5	00 Relocation
8-11	01 Relocation
14-17 ?	10 Relocation
(N-5)-(N-1) ~~20-23~~	11 Relocation

2.5.5 Special I/O Functions

In addition to loading and unloading registers, IOT instructions are used for several special functions as listed below. Note: that only the "skip-on-settled" function is available to the user.

Enable Interrupts. When a low-level interrupt is being serviced, other low-level interrupts are automatically locked out. At the end of the interrupt routine, it is necessary to enable these interrupts again. Similarly, when a high-level interrupt is serviced, other high-level interrupts are locked out, so that an "enable interrupt" IOT must be performed at the end of this routine also. The "enable interrupt" does not take effect until after the first "jump" instruction after the IOT.

Set User Mode. When an interrupt occurs, the LDS-2 goes into "executive" mode. In this mode, all of the defined IOT's are legal, and the scope selection registers can be set. User mode must be restored at the end of an interrupt service routine, or after the system has been initialized. User mode is not actually set until after the first "jump" instruction.

Sleep. Sleep is an idle state in which the LDS-2 does nothing but accept high-level interrupts.

Attention. When the LDS-2 needs to communicate with the host computer, the attention bit is raised. This IOT will cause an interrupt to the host computer.

Skip-on-Attention Clear. When the host computer has acknowledged the interrupt, it clears the attention bit. Before the LDS-2 issues another interrupt, it may want to check to see that the previous attention has been cleared. This is done by the "skip-on-attention clear" IOT.

Clear Protection Violation. When a protection violation occurs, a flip-flop is set which issues an interrupt. This flip-flop must be cleared by this IOT before going on to a new user.

Skip-On-Settled. This is the only special function IOT that is available to the user. Skip on settled causes the LDS-2 to skip the next instruction, if the pipeline is settled. This IOT is used when testing pipeline conditions (such as Area In Common) to insure that the pipeline is clear and the correct value for the condition can be read.

2.5.6 The Interface from the ~~SEL-840~~ Host Computer (HC) Side

The ~~SEL-840~~ host computer receives and issues interrupts through the ~~SEL-840~~ host computer I/O REGISTER of the LDS-2, which is an I/O device for the ~~SEL-840~~. host computer.

~~SEL-840~~ Host Computer I/O REGISTER

	Bits		Function
N-5	~~19~~	'200	Attention. When the LDS-2 issues an attention, this bit is set. It may also be loaded or unloaded from the ~~SEL~~ HC side of the interface.
N-4	~~20~~	'100	Attention Interrupt Mask. If this bit is set, the Attention bit will cause an ~~SEL-840~~ HC interrupt.
N-3	~~21~~		Stop State. When the LDS-2 is in the "sleep"

state, this bit is set. It can be read, but not set, by the ~~SEL 840~~ HC.

N-2 ~~22~~ Stop-State-Interrupt Mask. If this bit is set, the Stop-State bit will cause an ~~SEL 840~~ HC interrupt.

N-1 ~~23~~ LDS-2 Interrupt. By setting the LDS-2 Interrupt bit, the ~~SEL 840~~ HC issues an interrupt to the LDS-2. This bit is cleared automatically, when the interrupt is serviced by the LDS-2.

CHAPTER 3

THE MATRIX MULTIPLIER

3.1 Function

The Matrix Multiplier is the first arithmetic device in the LDS-2 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.

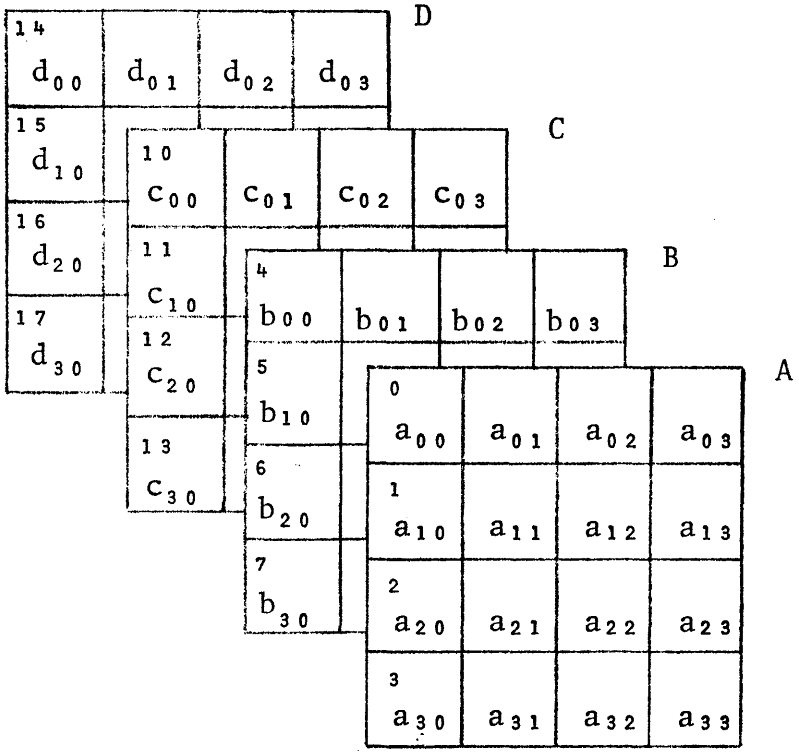
The basic configuration of the Matrix Multiplier and the addresses of the registers used for storing matrix elements are shown in Figure 3.1. Four matrices A, B, C, and D, each of dimension 4 x 4, are stored internally in a 4 x 4 x 4 matrix array of storage registers. The values in these registers may be manipulated by the "load," "store," "sink," and "retrieve" instructions. See Chapter 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

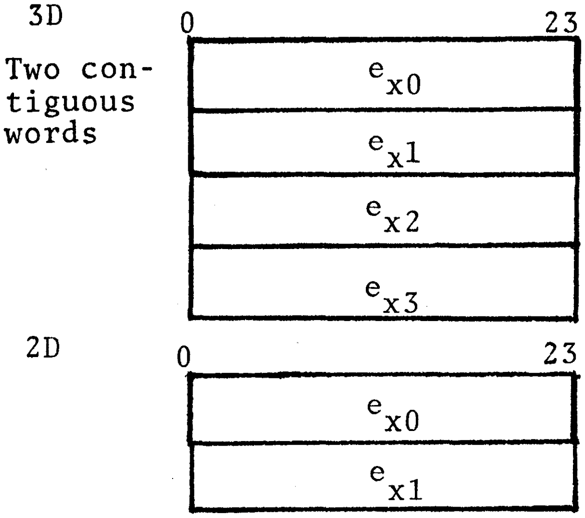
The Matrix Multiplier works on "homogeneous coordinates" (see Appendix III.) In homogeneous coordinates, three-dimensional coordinate data are represented by the four-component vector (X Y Z W), where X, Y, and Z are the normal orthogonal distances from the origin, and W is used as a scale factor. The 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 pipeline load/unload 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 Y Z_X Z_Y). 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:

$$[X\ Y\ 1]$$

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 $[T_1]\ [T_0]$. 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 2.4.5), 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

$$[X\ Y\ (W)] \quad \left[\begin{array}{cc:c} r_{00} & r_{01} & 0 \\ r_{10} & r_{11} & 0 \\ t_{20} & t_{21} & 1 \end{array}\right] \quad = \quad [X'\ Y'\ (W')]$$

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 yielding X/Z_X and Y/Z_Y, homogeneous 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 Matrix Multiplier 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 in repeat mode. In this case the RC 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 the current location are held on the top row of matrix A. Dataless drawing instructions (other than "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

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

Set Curve Operation

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

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

Other Drawing Instructions

Figure 3.4a

3D CURVES

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

top row specifies current absolute coordinate

Basic Representation

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

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

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

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

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

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" operation is used to generate each new curve of the surface patch.

3.8 Arithmetic Conventions

The word length of the Matrix Multiplier is 24 bits. The elements of input vectors and output vectors written into memory are all of this basic word length.

All arithmetic operations are performed treating elements as 2's complement signed (fixed point) *fractions*. Since the word length is 24 bits, the *algebraically largest* number that can be represented is $1-2^{-23}$, and the algebraically smallest number that can be represented is -1. In binary notation (with the binary point separating the sign bit from the fraction):

0.111111... is the algebraically largest number

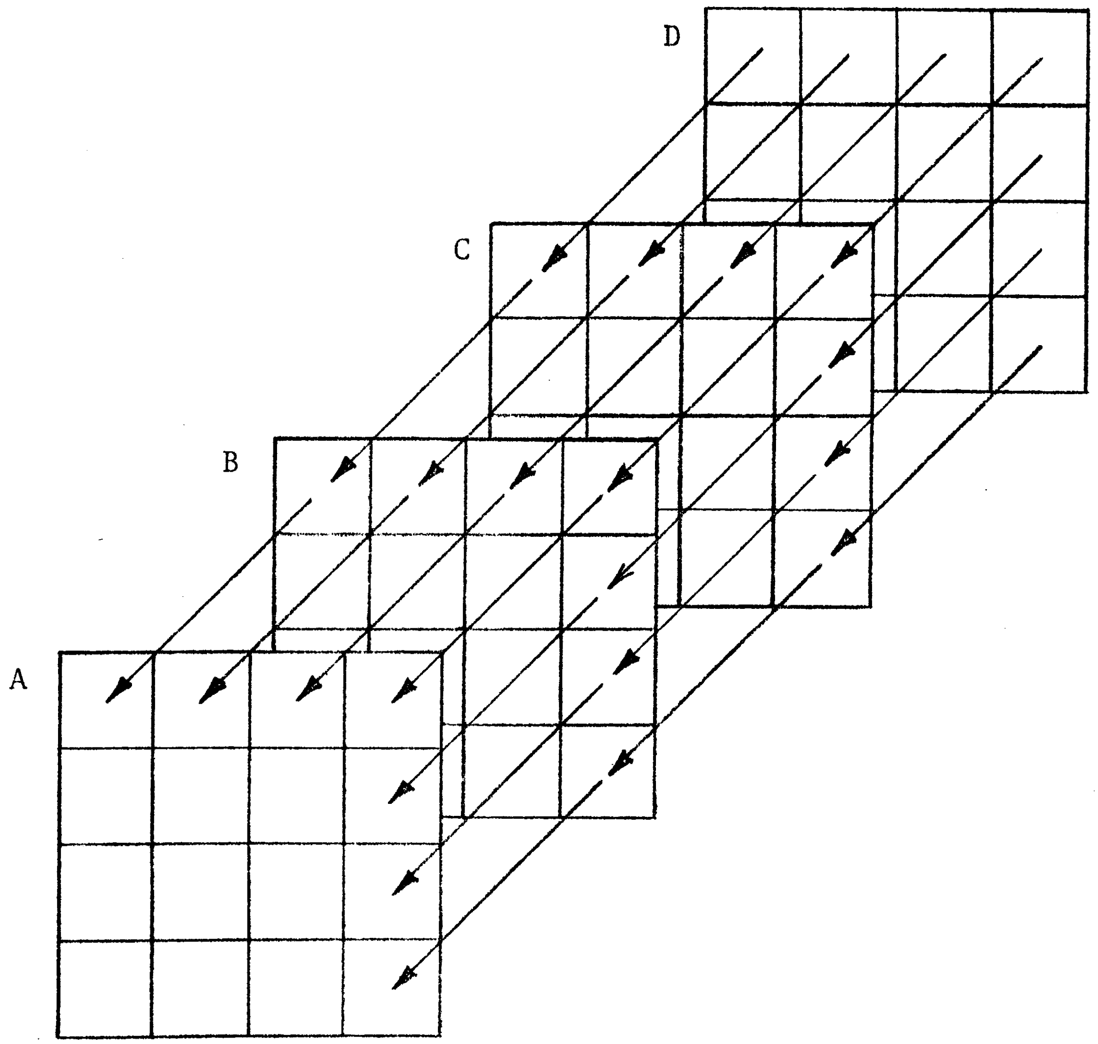
0.000000... is the unique representation for zero

1.000000... is the algebraically smallest number (-10.

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



$$\left.\begin{array}{l} A + QB \longrightarrow A \\ B + QC \longrightarrow B \\ C + QD \longrightarrow C \\ D + 0 \longrightarrow D \end{array}\right\} \text{For all 16 elements of each matrix}$$

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 (1.000000...). 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 or 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.

2D,3D (LDS-2 Dimension Modes) -- These bits determine whether the Channel Control supplies the Matrix Multiplier with a two-component or four-component input. 2D indicates a two-component (i.e., two-word) input, while <u>all</u> of the three-dimensional modes (including "homogeneous mode") indicate a four-component input. These rules apply for both drawing and register load/unload operations.

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:

3D

X'
Y'
Z'
W'

2D

X'
Y'

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

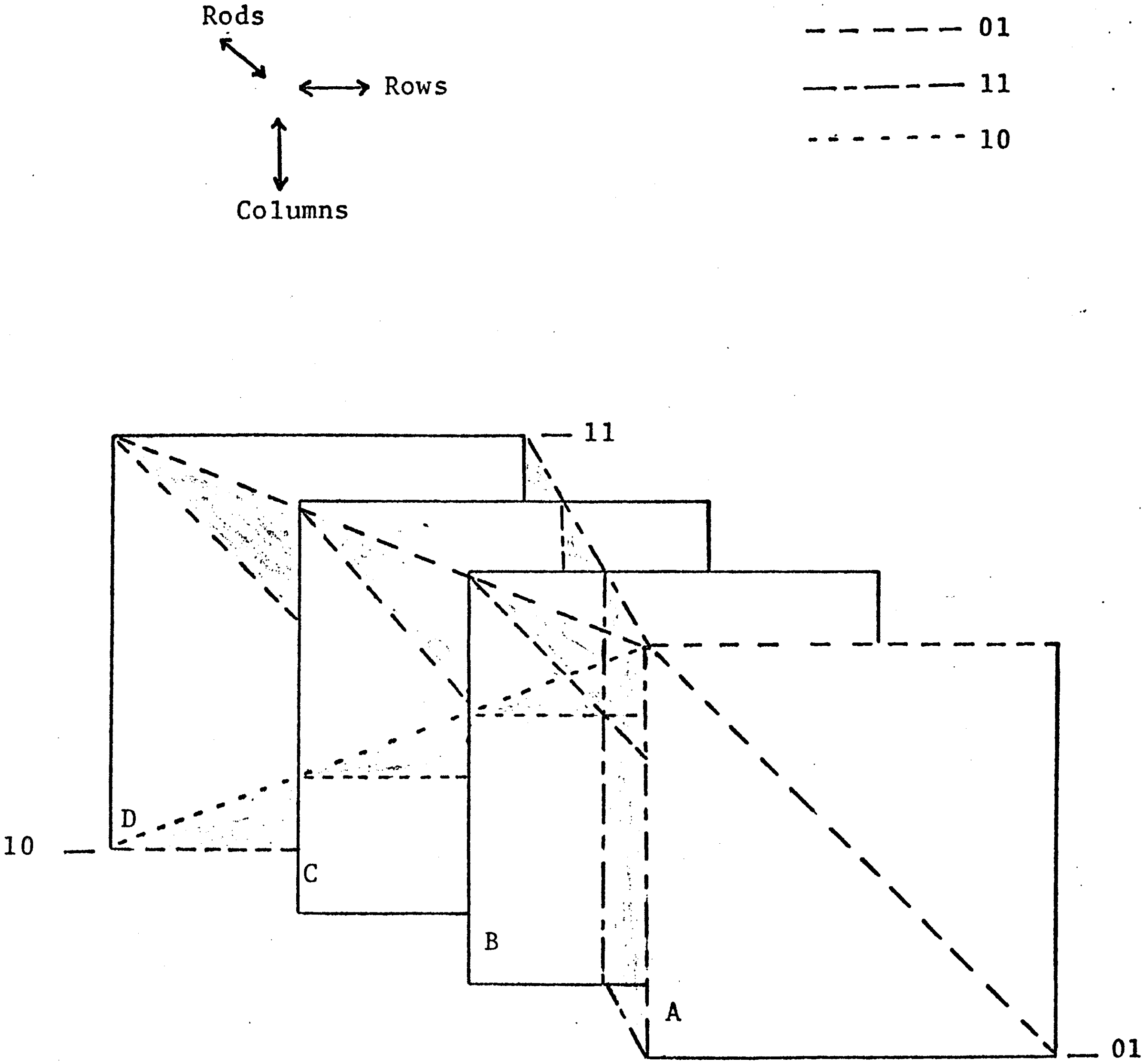
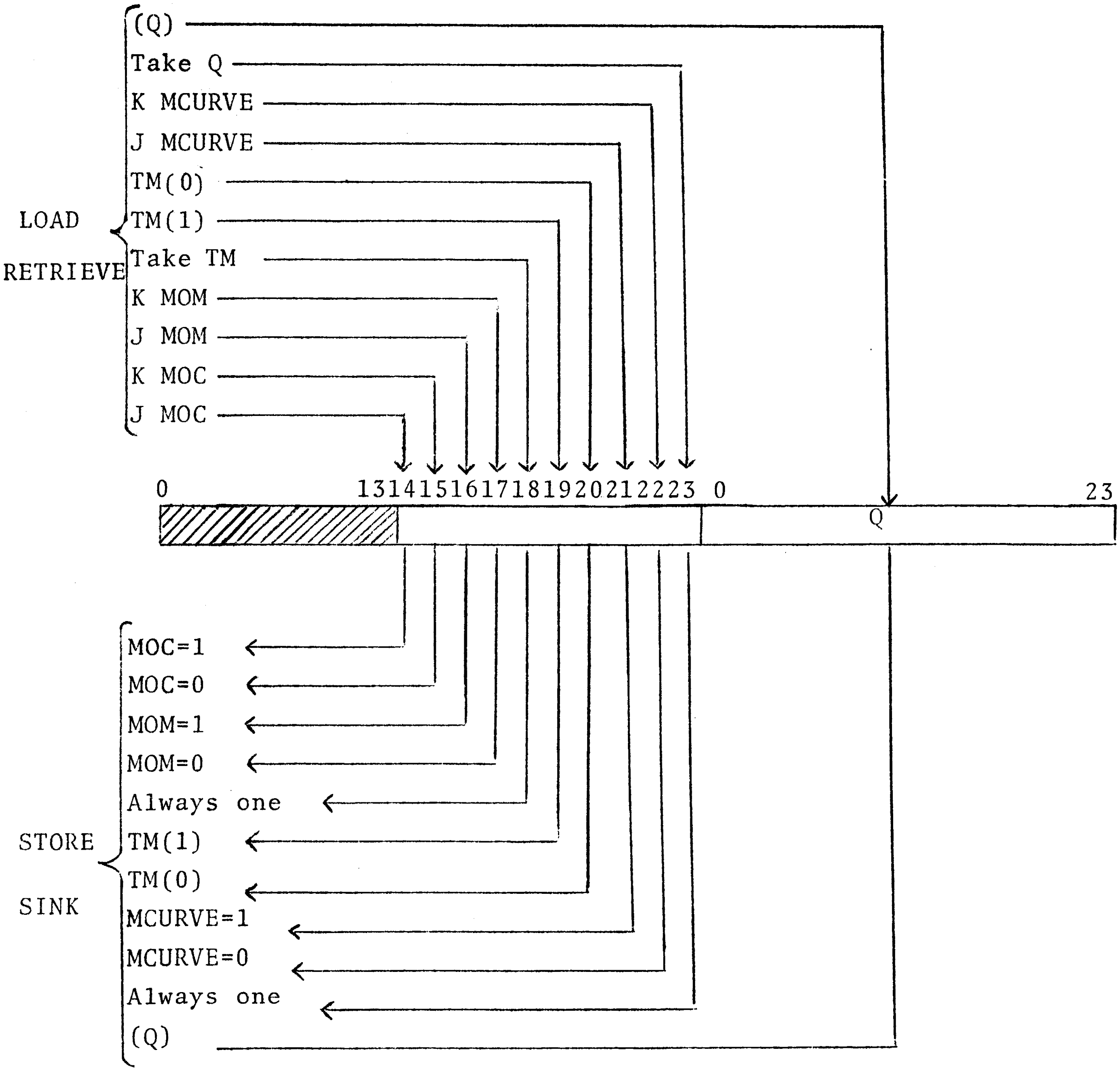


Figure 3.7

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 come 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 SAVE point which are retained by the LDS-2 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 SAVE 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 7. The address and structure of the SAVE register are shown in Figure 4.1.

4.3 Relative Data

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

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.

* Note: The Clipping Divider accepts only 23-bits of data from the Matrix Multiplier. The high-order bit is a sign-extension. This is done to prevent overflow within the Clipping Divider. The page for the LDS-2 is thus effectively 23 bits rather than 24.

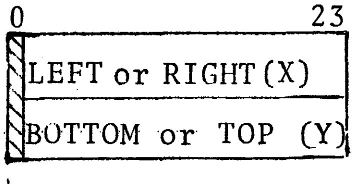
CLIPPING DIVIDER REGISTER CONFIGURATION

4-component addresses	2-component addresses					
14 SAVE	0 SAVELB	LEFT (X)	BOTTOM (Y)	1 SAVERT	RIGHT (X or Z_x)	TOP (Y or Z_y)
15 VIEW (VIEWPORT)	2 VIEWLB	LEFT (X)	BOTTOM (Y)	3 VIEWRT	RIGHT (X)	TOP (Y)
16 WIND (WINDOW)	4 WINDLB	LEFT (X)	BOTTOM (Y)	5 WINDRT	RIGHT (X)	TOP (Y)
17 INST (INSTANCE)	6 INSTLB	LEFT (X)	BOTTOM (Y)	7 INSTRT	RIGHT (X)	TOP (Y)
	10 NAME	NAME	NAME	11 CDIR	CDIR	▤
	12* HITANG	HIT, CORNER, EDGE COUNTS	ANGLE COUNTS	13* SELINT	SEL- ECT | PER MIT	INTENSITY

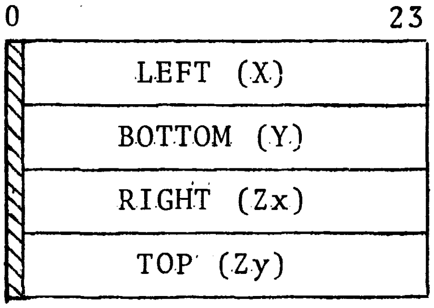
DATA FORMATS

* All bits not used, see figure 4.5 for exact formats.

2D



3D



Note: Bit 0 is a sign extention.

Note: The names associated with the registers are LDS-2 mnemonics which have been defined in the LDS-2 Assembly language.

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 -77777 to +77777 (i.e., 16 bits), but because the VIEWPORT register is a full 24-bit register and because only the 16 *least* significant bits are used to drive the scope, each boundary of the viewport should be specified to be between -77777 and +77777. 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 -17777777, +17777777 (in each axis) and a viewport specification of -77777, +77777 (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 1/4 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 is 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. If Z is negative, the line is clipped. Since Bit 0 of the Clipping Divider is a sign extension, *Z values should not be larger than 17777777*, or the line will be clipped.

In three-dimensions, perspective division becomes part of the process of mapping the coordinate data into scope coordinates. This perspective division yields 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

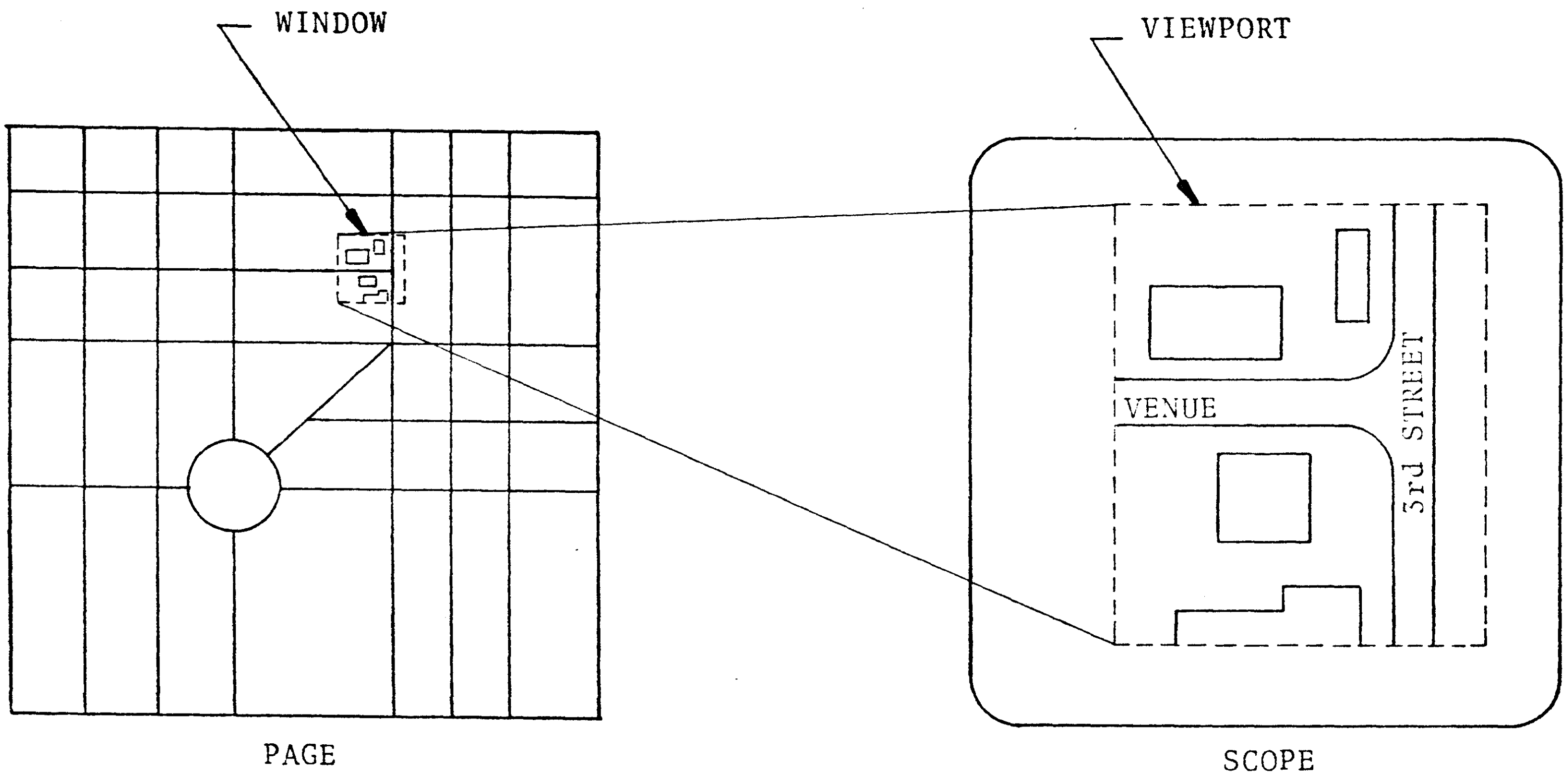


Figure 4.2

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 α is the desired viewing angle.

4.6 Boxing

The boxing process is a special feature of the 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" operation of the LDS-2 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

THREE-DIMENSIONAL CLIPPING AND DIVISION

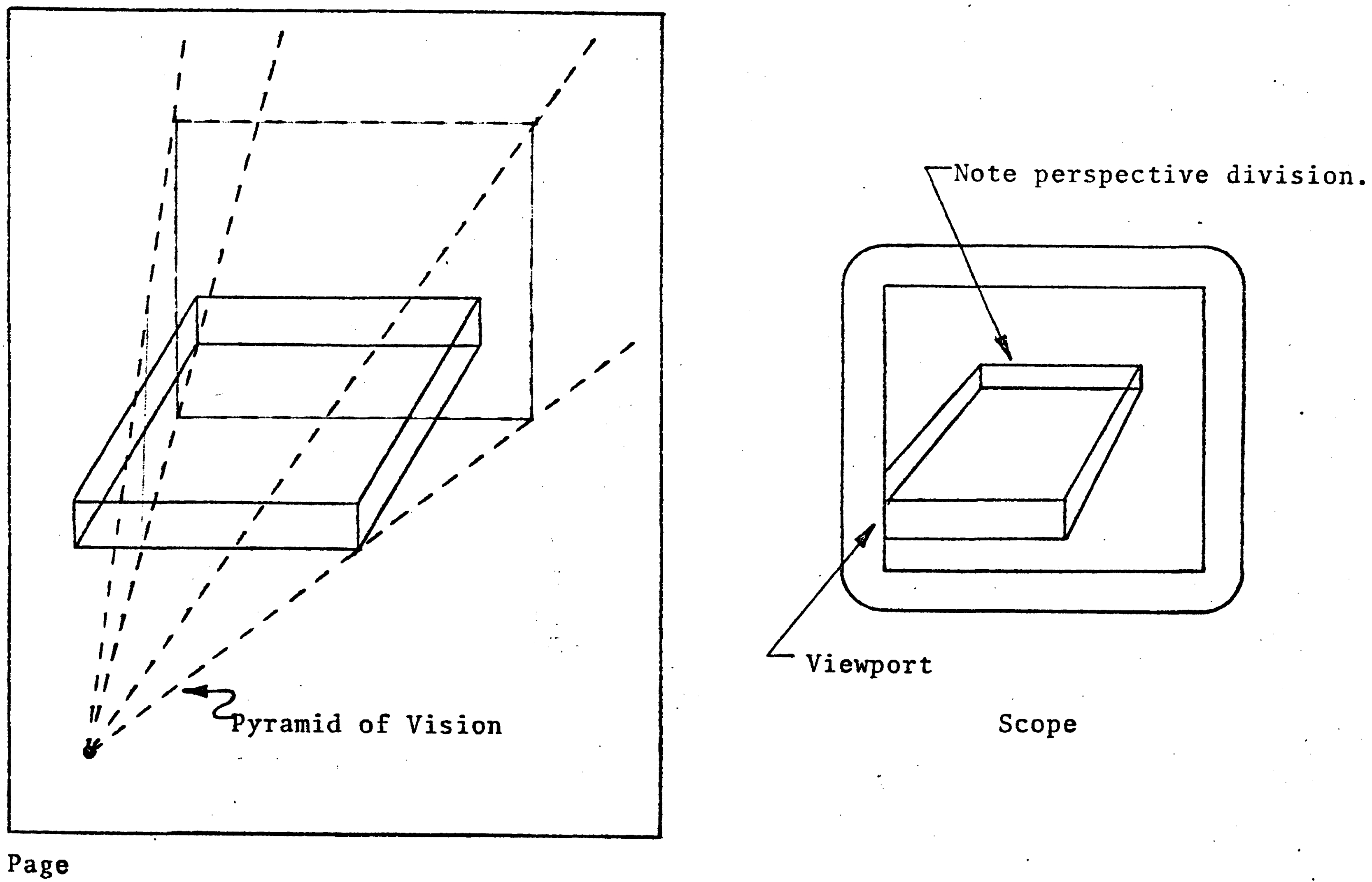


Figure 4.3

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 DIRECTIVE register of the Channel Control, where it can be tested prior to boxing. Please not that for the AIC bit to operate properly, the INSTANCE register must be the last register loaded with a 2D four-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 7.14. 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 DIRECTIVE register of the Channel Control where it can be tested. The HIT bit can also be enabled to interrupt the LDS-2. Once the HIT bit is set, it remains on until cleared by an IOT instruction. 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.

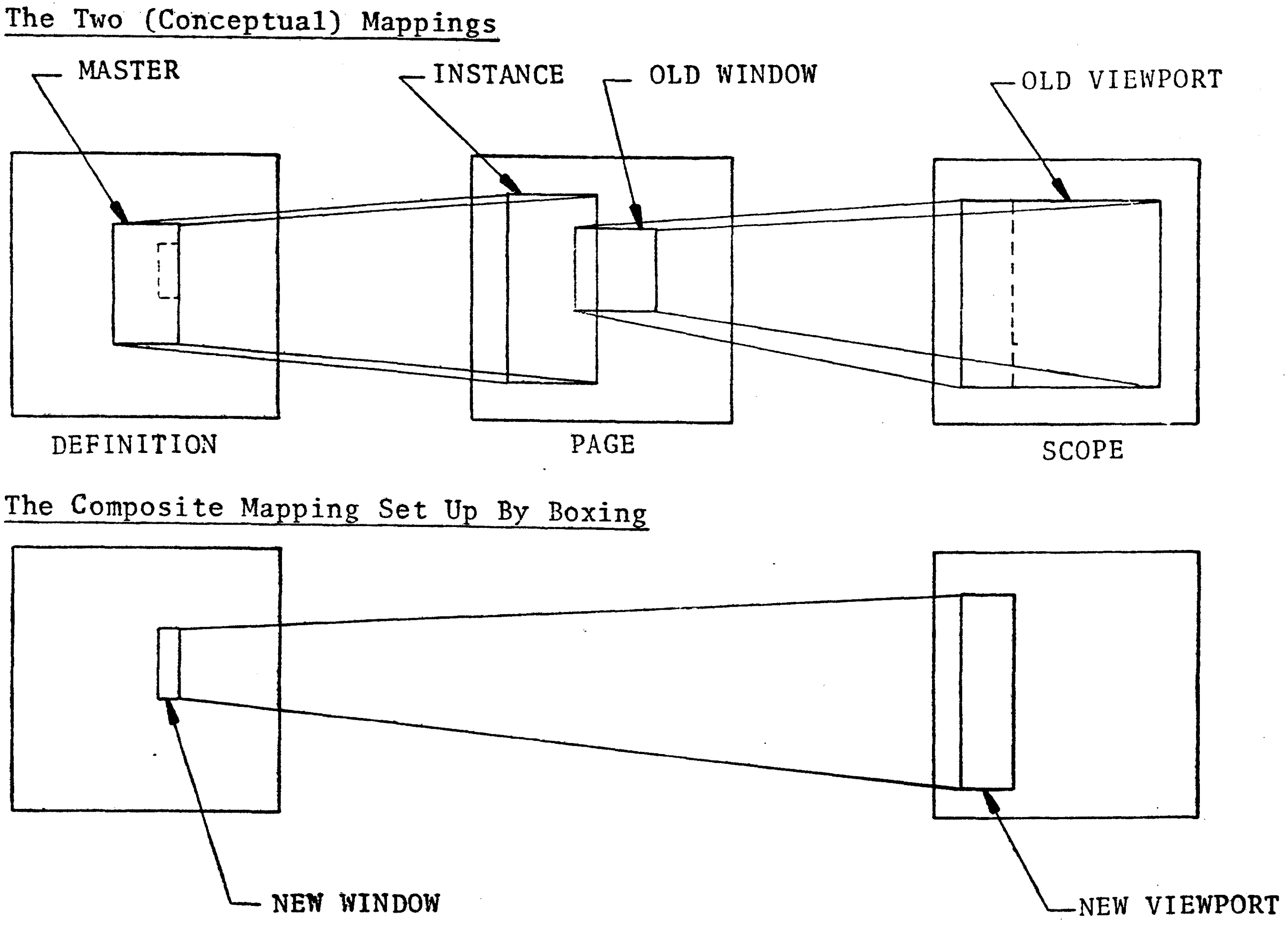
Several different counts that may be useful in examining the geometry of a drawing are maintained in the HITANG register. These counts are primarily useful for determining the relationship between polygons and the current window and, thus, will be explained assuming that a polygon is being drawn.

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.

BOXING



The Two (Conceptual) Mappings

The Composite Mapping Set Up By Boxing

Figure 4.4

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 in 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 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.

(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. Bits 2-9 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 LDS-2 (see Section 2.5). The permit bits can only be loaded when the LDS-2 is in executive mode.

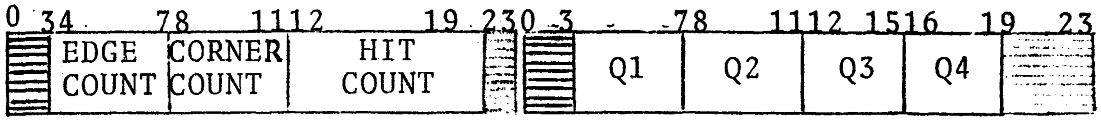
The last 24 bits of the SELINT register are used to specify the intensity. However, only bits 1 through 13 are actually used (see Section 5.2.1). Zero specifies greatest intensity, 37770000 specifies least intensities. The format for the SELINT register is shown in Figure 4.5.

4.9 The NAME Register

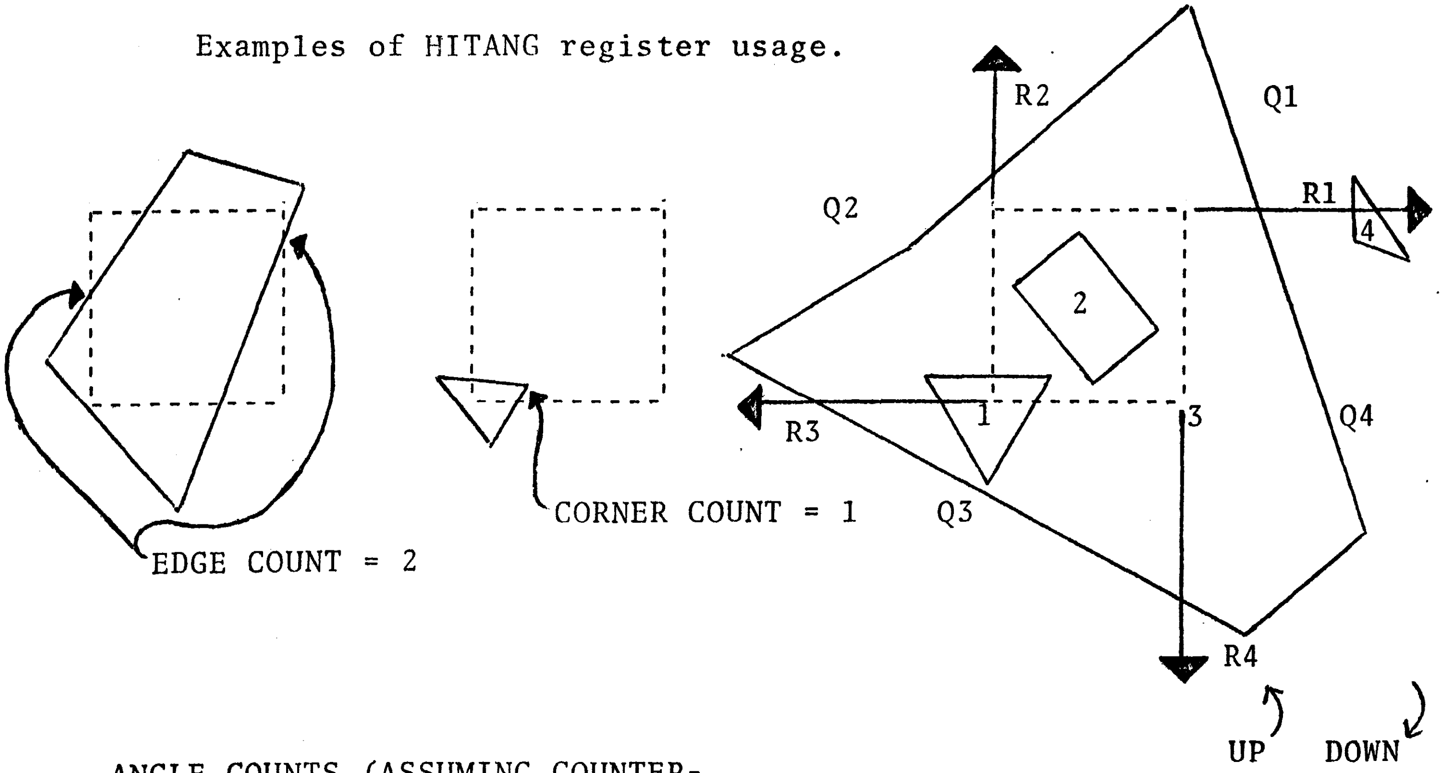
The NAME register of the Clipping Divider is an unassigned register, which can be used by the programmer as a storage register. The NAME register can be loaded, stored, sinked, or retrieved.

HITANG and SELINT REGISTERS

HITANG REGISTER



Examples of HITANG register usage.



ANGLE COUNTS (ASSUMING COUNTER-CLOCKWISE TRACE)

1. Intersects the window
2. Entirely within the window
3. Entirely surrounds the window
4. Outside the window

	Q1	Q2	Q3	Q4
1.	0	0	1	0
2.	0	0	0	0
3.	1	1	1	1
4.	0	0	0	0

SELINT REGISTER

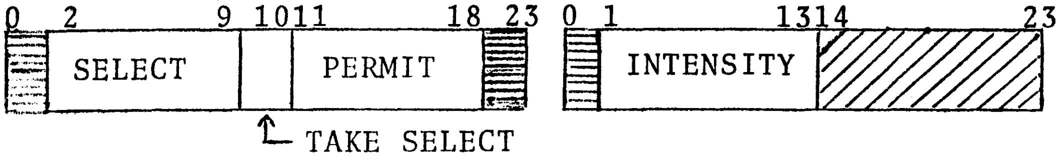
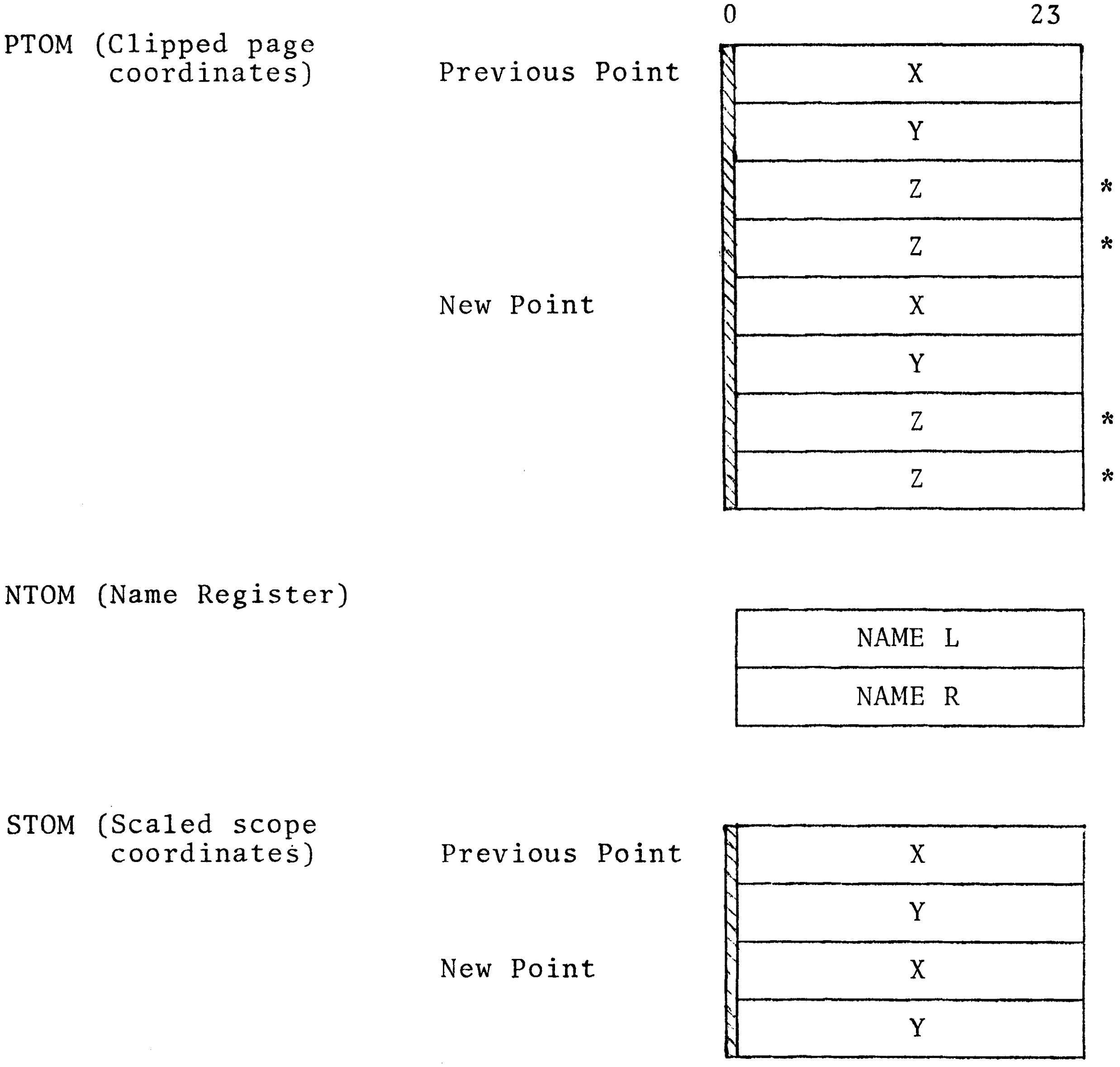


Figure 4.5

FORMAT FOR CLIPPING DIVIDER OUTPUT TO MEMORY



Note: Bit 0 is a sign extension.

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

* Omitted if 2D set.

Figure 4.6

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. 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 ΔX or ΔY.

For more efficient storage of the coordinate data, the "register draw" instructions should be used with SELFX or SELFY. If coordinate data are accessed from memory for the drawing instructions, the data will be interpreted as shown below:

SELFY
X_1
--
X_2
--
:

SELFX
--
Y_1
--
Y_2
:

4.11 Mode Control

The dimension mode bits of the Channel Control DIRECTIVE (DIR) register determine whether the Clipping Divider is in 2D or one of the 3D modes. The rest of the mode control information is stored in the Clipping Divider directive register (CDIR).

The bits of this register are as follows:

0-3	Unused	
4	STOS	Scaled output to scope.
5	STOM	Scaled output to memory (see figure 4.6 for format).
6	ZTOS	Z sent to scope to control intensity. (Otherwise the intensity bits of the SELINT register control intensity).
7	PTOM	Clipped page coordinates (before division) to memory (see figure 4.6 for format).
8	NTOM	NAME register contents to memory (see figure 4.6 for format).
9	Take bits 2-6	If not set bits 2-6 are not loaded.
10	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.
11	K (Clear) CURVE	
12	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).
13	K (Clear) MEF	
14	J (Set) Dashed Line	Causes the line drawn on the scope to be dashed rather than solid.

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

CHAPTER 5

THE LINE GENERATOR AND DISPLAY SCOPE

5.1 Function

The last units in the LDS-2 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 8-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 bits 1 through 13 of the value stored in the INTENSITY 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 2-9 of SELINT) of the Clipping Divider. These bits are masked against the bits in the Permit register (bits 11-17 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 in executive mode 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.

THE LDS-2 ASSEMBLER

6.1 General Characteristics

The LDS-2 Assembler takes source code written in LDS-2 Assembly Language and assembles it into object code which can be executed by the hardware of the LDS-2. The LDS-2 Assembly Language features symbolic representations for addresses and arguments, literals, automatic definition for symbols, and facilities for defining new mnemonics. The LDS-2 Assembler runs on the LDS-2, but the input is provided by the host computer. The details of the software interface between the LDS-2 and the host computer and instructions for calling the assembler are given in Chapter 9. Examples of LDS-2 Assembly Language usage and descriptions of the instructions are given in Chapter 7.

6.2.1 Symbols

A symbol is composed of from one to six alphabetic, numeric and non-reserved special characters (see Figure 6.1). Only those special characters which are not specifically designated for other purposes may be used in a symbol. A symbol may represent a statement label, an external name, or an equivalence relationship, such as a register name. When a symbol is defined within the program, it is flagged as either absolute or relocatable. If the assembly is in absolute mode, all symbols are absolute. Otherwise, any symbol which is a statement label (LAB) or derived from a statement label is relocatable. A symbol encountered in an expression may be automatically defined and assigned a location by placing a pound sign (#) immediately following it.

6.2.2 Numbers

A number consists of one or more of the digits 0-9. Numbers may be of any length; however, if the number is larger than the field into which it is to be placed, its excess high-order (left-hand) bits are discarded in order to make it fit. Numbers which begin with a preceding zero are interpreted as octal, while all other numbers are interpreted according to the prevailing radix, which is initially base ten. All numbers are considered to be positive integers.

6.2.3 Current Location Pointer

When the period (.) is encountered in a statement subfield, it is assumed to represent the current value of the location counter.

LDS-2 ASSEMBLER CHARACTER SET

ALPHABETIC CHARACTERS

A - Z

NUMERIC CHARACTERS

0 - 9

SPECIAL CHARACTERS

All other special ASCII characters, except as listed below, may be used in symbol formation.

SPECIAL CHARACTERS RESERVED FOR ASSEMBLER USE

.	Current location pointer
,	Subfield separator
'	Text string delimiter
;	Alternate statement terminator
@	Indirect address flag
%	Indexing flag
$	Statement continuation symbol
=	Literal delimiter
+	Addition operator
-	Subtraction operator
*	Multiplication operator; comment line indicator
/	Division operator
( )	Priority indication (used in expressions)
#	Auto-definition flag
[ ]	Used in OPDEF
< >	Reserved for future use
Space	Field separator
Carriage Return	Statement terminator

Figure 6.1

6.2.4 Expressions

An expression consists of one or more symbols, numbers and/or current location pointers, separated by combinations of the arithmetic operators "+", "-", "*", or "/". The last item in an expression must not be an operator. The expression is evaluated according to Fortran heirarchy - that is, "*" and "/" are evaluated first, then "+" and "-", except where overridden by the use of parentheses. If the expression contains a division by zero, the original dividend replaces the quotient. When successive operators of equal heirarchy are encountered, they are evaluated from left to right. All arithmetic is in fullword two's complement integer, so that fractional portions of quotients are discarded. Parentheses are permitted in an expression. As the expression is evaluated, its terms are checked for relocation compatibility, and the final evaluated expression must be either purely relocatable or purely absolute. Thus, assuming that "A" is an absolute symbol, "R" is a relocatable symbol, and "X" is any symbol, the following are illegal expressions:

```
R+R   (R+R-R is legal)

A-R   (A-R+R is legal)

X*R   (R*1, 1*R, 0*R, and R*0 are legal

X/R

R/X   (R/1 is legal)
```

If the expression begins with an operator, the assembler assumes an item preceding it, which has a value of zero and is in absolute mode. It is in this manner that negative numbers are handled. Once an expression has been evaluated, it is trimmed to fit the field into which it is to be placed in accordance with the same rules of modulus as for numbers (see Section 6.2.2).

6.2.5 Text Strings

A string of characters enclosed in single quotes (apostrophies) is called a text string. Such a string is interpreted by the assembler as a packed series of truncated ASCII characters, and is packed accordingly into successive computer words, six bit byte format, four characters per word, left justified. Any character may appear in the text string. A single quote, however, is represented by inserting two adjacent single quotes into the string. Unused portions of words containing text strings are blank-filled.

6.2.6 Literals

A literal may be used to replace the address in an operand field. When the literal is assembled, it is replaced by the address of the one-word memory location which contains the literal value. Thus, literals are automatically defined by using them. A literal must be preceded by an equal sign (=). The following types of literals are allowed.

Expressions

When an expression is used in a literal, it must be preceded by an equal sign. Should the expression contain a ".", however, the "." will be evaluated as the value of the location counter at the current statement; hence, precisely the same literal appearing in the next statement will be evaluated differently, and will be assigned a different memory location. If a literal expression contains a forward reference to a symbol, a new literal word will be reserved, even though the same expression may have occurred previously.

Text Strings

When a text string is used in a literal, the string, including the single quotes surrounding it, is preceded by an equal sign. If the string is greater than four characters in length, only the first four characters are accepted; the rest of the string is ignored.

6.2.7 Subfields

A subfield consists of either an expression, a text string, or a literal. The operand field of a statement is often composed of several subfields, each of which is terminated with a comma, or in the case of the last subfield, with a space or carriage return (or a semicolon, should another statement follow on the same line). All subfields other than the first must be preceded immediately by a comma. Two adjacent commas indicate a null subfield. A null subfield is assumed to be absolute, and to have a value of zero. If a subfield is the only one in the operand field, it may not be null, although it may contain zero. Should a dollar sign ($) immediately follow a subfield, the line will be assumed exhausted, the rest of the line will be ignored, and scanning for the next subfield will begin with the first non-blank character on the next line. The address subfield may contain either an expression or a literal, and is preceded optionally by the indirect-address flag (@) and/or the indexing flag (%) where permissible and applicable. These flags must precede other data in the subfield, but may occur in either order. If a subfield of an instruction which requires a relocatable expression is left null, an error is indicated by the assembler. The subfields of the EXTERN and ENTRY directives must be symbols, and the subfields of the DATA directive may contain expressions or text strings; all other subfields, except address subfields, are limited to expressions. Expression arithmetic involving external symbols is prohibited.

6.2.8 Fields

A field is a portion of a statement separated from other portions by one or more blank characters. It consists of one or more subfields.

6.2.9 Statements

The statement is the basic entity of the assembly language for the LDS-2. A statement consists of up to four fields separated from each other by one or more spaces.

The first or label field is optional, except in EQU or OPDEF directives, and with the exception of these two directives, is used to identify the memory location into which the current instruction or data word is to be inserted. The label must always be a symbol, and, with the exception of the EQU and OPDEF directives, its inclusion in the statement automatically causes it to be defined and given the value of the current location counter. If this field is omitted, at least one space must be inserted at the beginning of the statement. The first field in EQU and OPDEF directives is not interpreted as a label, but rather as a symbol or mnemonic which is to be set equal to some value.

The second field is always mandatory, and contains the instruction or directive mnemonic, which is a name following the format of a symbol, but in no way associated with labels; in fact, labels may be spelled exactly the same as instructions with no possibility of confusion. This field must be followed by at least one space, unless it has no operand and another statement follows on the same line, in which case it must be followed immediately by a semicolon.

The presence of the third or operand field depends entirely on the particular instruction or directive. This field is the only one which may contain subfields, and is used to specify the arguments of the instruction or directive. Should a symbol occur in this field, it is considered a reference to, rather than a definition of, a label. This field may also be followed immediately by a semicolon to indicate that another statement follows on the same line, or by a space or carriage return.

The fourth or comments field, which is always optional, except with the END statement which may not have a comment, ignored; and, therefore, any character may be included in the comment field, including the semicolon. If a comment exists (i.e., a semicolon or carriage return does not immediately follow the last mandatory field), only the carriage return or end of line may terminate the statement.

If a line begins with an asterisk (*), the entire line is treated as a comment and is not processed, but is listed in the assembly listing.

6.3 Assembler Directives

Directive statements are offered to allow the user to provide information to the assembler for the purpose of controlling the assembly of actual codes. Note: The label field of any of the directives listed below is optional, except for EQU and OPDEF directives.

6.3.1 Assembly-Control Directives

Format: LAB RADIX N

Where: N is a decimal number from 2 to 10, indicating the base of the number system used in evaluating the numbers used in subsequent statements.

This directive causes the prevailing radix for number interpretation to be modified. If this directive is not used, the radix will be assumed to be 10 (decimal). However, use of a leading zero will always cause the number to be interpreted as octal (Radix = 8).

Format: LAB DUP M,N

Where: M and N are expressions.

DUP causes the group of M instructions and directives following the DUP directive to be replicated N number of times. M must be greater than zero; N may be zero or greater. The default condition for M is one. Any directive, except END, may be included in the range of a given DUP. DUP's may be nested up to five levels deep; however, the boundaries of a given DUP range must completely enclose the boundaries of all DUP's occurring within that range. The number of statements in the range of the primary DUP is determined strictly by the space available in the symbol table.

Format: NAME EQU N

Where: NAME is a symbol; N is an expression.

This directive sets NAME equal to the value of N. If any symbols appear in N, their values must have been previously defined. If N is a relocatable expression, NAME will be flagged as relocatable; otherwise, it will be flagged as absolute. N may not contain an external symbol or an instruction or directive mnemonic.

Format: (1) NAME1 OPDEF NAME2

(2) NAME1 OPDEF NAME2 OPDFLD , FIELD1, FIELD2,...

(3) NAME1 OPDEF (Expression),FIELD1,FIELD2,...

Where: NAME1 is the name of the mnemonic which is being defined, NAME2 is the name of a previously defined mnemonic, OPFLDS is the appropriate operand field, and FIELD1,FIELD2, etc., have either of the following forms:

(1) (length of field, location of lowest-order bit,N)

(2) (length of field, location of lowest-order bit,A@%)

Form (a) is used for non-address fields, while Form (b) is used for address fields.

This directive is used to define new mnemonics for LDS-2 instructions. The names of mnemonics which are initially defined for the assembler may not be used for new definitions. Several of the instruction groups have various possibilities, not only for the names of mnemonics, but also for the way in which the operand fields are defined. Through the use of the OPDEF directive, the user has the option of defining alternate forms for LDS-2 instructions. (Note: See Appendix II for the OPDEF's which have been initially defined for the assembler.)

Format: LAB ORG N

Where: N is an expression.

This directive sets the location counter to the value of the expression. The value of the expression is required to be relocatable. If a label is associated with this directive, it assumes the old value of the location counter. The assembler initially assumes an ORG, where N points to the beginning of the first page, until it encounters another ORG.

Format: LAB LITORG

This directive causes all literals so far defined to be inserted into the program beginning at the current value of the location counter, and the literal table cleared. If the directive is labelled, the label will be assigned the address of the first literal. An automatic LITORG is generated upon encountering an END or PAGE directive (see below). Note: Once the literal table has been cleared by a LITORG, all references to previous literals are lost. Hence, the user must exercise caution in modifying the contents of a literal during execution of his program to provide a temporary storage area.

Format: LAB END N

Where: N is an expression.

This directive must be the last statement in the program, and signifies to the assembler that the input is complete. The expression N is optional, and if present, indicates the address at which execution is to begin. For relocatable assemblies, N is required to be relocatable. If the statement contains a label, the label will be assigned the address of the first literal at the end of the program, should one exist, and provided that the user has not used the auto-definition feature. Because the operand is optional, the END statement may not contain a comment field, unless the operand field is explicitly supplied. Otherwise, the Assembler will mistakenly treat the comment as an operand.

6.3.2 Object-Control Directives

Format: LAB EXTERN N,N,N,...

Where: The N are symbols.

This directive causes the symbols N,N,... to be interpreted to the loader as being defined in an external program, and instructs the loader to insert the proper linkage. If the symbol is also defined in the current program, a multiple-definition error will result.

Format: LAB ENTRY N,N,N,...

Where: The N are symbols.

This directive causes the symbols N,N,... to be made available to the loader for the purpose of defining symbols specified in other programs in EXTERN statements. A label used with this directive will be assigned the current value of the location counter and has no relation to the values of the operands of the ENTRY directive. If the symbols are not defined elsewhere in the program, an error will result.

6.3.3 Listing-Control Directives

Format: LAB LIST N

Where: N is an expression.

If N has a value of zero, all subsequent lines, until the next LIST directive, will not appear in the assembly-listing. If N is non-zero, the current line and all subsequent lines to the next LIST will be listed as follows: If N equal 2, the listing will be double-spaced; otherwise, it will be single-spaced.

Format: LAB SKIP N LAB SKIP N, 'text string'

Where: N is an expression.

This directive causes N blank lines to be inserted in the assembly-listing. If the number of lines to be inserted takes the listing past Line 56 of the current page, the listing will begin on the top line of the page following. N must be greater than zero. If the subfield is followed by another containing a text string, that string will appear on the heading line of all subsequent pages, until a similar SKIP directive is encountered. The line containing the SKIP directive is not listed, unless it has been labelled.

6.3.4 Storage-Allocation Directives

Format: LAB DATA N2,N3,...,NM

This directive causes M words to be reserved in memory, beginning at the address specified currently by the location counter. Into each word is placed the value of the corresponding subfield. The subfields may contain either expressions or text strings; if text, a sufficient number of words is reserved to accommodate the string. If the directive is labelled, the label is assigned the value of the location counter prior to incrementation; that is, the address of the first word generated by the DATA directive. Note: The "DATA" mnemonic may be omitted if the first operand subfield does not begin with a name.

Format: LAB BLOCK N

Format: LAB BLOCK N

Where: N is an absolute expression.

A block of N consecutive memory locations is reserved in the program, beginning at the address currently specified by the location counter. The counter is incremented by N. If the directive is labelled, the label is assigned the value of the location counter before incrementation.

6.4 Error and Warning Messages

If errors or possible errors are encountered during the assembly, error and warning messages will be printed in the listing. Some errors will cause termination of the assembly, while others are non-fatal. There are four levels of error messages:

1. Warning. The user is simply warned of a possible problem.

2. Error. An object module will be produced, but it will contain errors.

3. Fatal Error. The object module is discontinued, but the assembly and listing will continue.

4. Catastrophe. Assembly is immediately discontinued, probably due to an assembler error.

The following error messages are provided by the assembler:

TYPE	LEVEL	MEANING
A	1	Name too long; last part ignored
B	2	Number expressed in wrong radix
C	3	Symbol table overflow
D	2	Undefined symbol
E	2	Improperly nested parentheses
F	2	Misplaced arithmetic operator
G	2	Illegal placement of text string
H	2	Illegal use of external name
I	2	Multiply-defined symbol
J	2	Illegal use of relocatable name
K	4	Assembler error; get dump and call system man
L	2	Unresolved literal reference
M	3	Illegal DUP range
N	2	Undefined Mnemonic
O	2	Missing or garbled operand field
P	2	Literal Out of Address Field
Q	2	Flag illegal in this field
R	1	Too few subfields; remainder assumed null
S	2	Too many subfields
T	2	Duplicate flags
U	2	Field must be relocatable
V	2	Reference across page boundary
W	2	Displacement exceeds one page
X	4	Input data out of order
Y	2	Label missing or incorrect
Z	2	Illegal expression
]	2	Missing END statement

LDS-2 INSTRUCTION SET

7.1 Accessing Data for the Instructions

The necessary data for LDS-2 instructions may be accessed in one of three ways:

The address may be specified as part of the instruction word. This address may be a direct address or an indirect address. With most addressing instructions, indexing is also available. If both indirection and indexing are specified, the indirection is performed before indexing.

The address of the data may be contained in one of the Channel Control registers. For instance, in most drawing instructions the address is contained in the READ POINTER (RP). In the case of the drawing instructions, the address in the RP may be taken as an indirect address or an indirect and indexed address.

The data for the instruction may be contained in the Channel Control registers so that no memory reference is made at all.

7.2 Notation

For the descriptions of the instructions, the following special symbols are used:

R,R1,R2 These symbols are used to specify Channel Control register addresses. R1 → R2 means that the contents of R1 are loaded into Register R2.

N The symbol "N" specifies immediate data. Immediate data are taken as an unsigned (positive) integer. For a 24-bit system N may range from 0 to 7777 (octal).

b The symbol "b" is used to specify either the bit position or the number of bits to be shifted. Bits are numbered beginning with Bit 0 on the high-order (left end) of the word.

ADDR The address part of the instruction word is called ADDR. This is the address within the current page.

@ The "@" symbol is used to specify indirection. If this symbol precedes the ADDR portion of the instruction, ADDR will be taken as an indirect address.

% The "%" symbol specifies indexing. Although indirection may be specified whenever there is an ADDR field, indexing is only legal for some addressing instructions as specified in the detailed descriptions of the individual instructions.

e   The symbol "e" is used to represent the effective address. The effective address is the final memory address obtained after paging, indirection, and indexing have been performed.

C(e)   The contents of the memory location specified by the effective address are represented by C(e).

C(R)   The contents of the memory location referenced by the address contained in Register R are specified by C(R). Note, that in this case, it is assumed that R contains a memory address. R1→C(R2) means that the contents of Register R1 are deposited into the memory location specified by the contents of Register R2.

7.3 Loading and Storing the Channel Control Registers

The load and store instructions for the Channel Control registers allow data to be transferred between memory and a Channel Control register, between one register and another, and from the immediate data field of an instruction word into a register.

Mnemonic: LO (Get Sequence) LOad

Structure:

0	1 3	4 7	8 ~~15~~ ~~16~~ ~~19~~ ~~20~~ ~~[illegible]~~ N-1
@	0 0 0	R	ADDR

Format: LO R,@ADDR

Function: Load the contents of the effective address into register R. The previous contents of R are lost.

C(e) → R

Mnemonic: ST Put Seq. STore

Structure:

0	1 3	4 7	8 ~~15~~ ~~16~~ ~~19~~ ~~20~~ ~~28~~ N-1
@	0 0 1	R	ADDR

Format: ST R,@ADDR

Function: Store the contents of Channel Control Register R into the memory location specified by the effective address.

R → C(e)

Mnemonic: RLO Bin. Seq. Cont IV-6 Register LOad

Structure:

0 1 3	4 7	~~8~~ ~~15~~	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~25~~ N-1
0 1 1 0	R1		R2	0 0 1 0

Format: RLO R1,R2

Function: Load Register R1 with the contents of Register R2. The previous contents of R1 are lost.

R2 → R1

Mnemonic: RLOZ Register LOad and skip to Zero

Structure:

			N-8 N-5	N-4 N-1
0 1 3	4 7	8 ~~15~~	~~16~~ ~~19~~	~~20~~ ~~23~~
1 1 1 0	R1		R2	0 0 1 0

Format: RLOZ R1,R2

Function: Load Register R1 with the contents of Register R2 and skip the next instruction if R1 contains zero (after having been loaded from R2). The previous contents of R1 are lost.

R2 → R1

Mnemonic: ILO Immediate LOad

Structure:

			N-4 N-1
0 1 3	4 7	8 ~~15 16 19~~	~~20~~ ~~23~~
0 1 1 0	R	N	1 0 1 0

Format: ILO R,N

Function: Load Channel Control Register R with the immediate value N. The previous contents of R are lost.

N → R

Mnemonic: ILOM Immediate LOad Minus

Structure:

			N-5 N-4 N-1
0 1 3	4 7	8 ~~15 16~~ ~~19~~	~~20~~ ~~23~~
0 1 1 0	R	N	1 0 1 0

Format: ILOM R,N

Function: Load Channel Control Register R with minus (two's complement) value N. The previous contents of R are lost.

-N → R

7.4 Program Control

The normal sequential flow of the program may be changed by the following instructions. The "pushjump" and "popjmp" instructions use the PC-stack mechanism of the Channel Control. Remember that the top element in this stack is the TOS register, and that the STACK POINTER (SP) points to the second element in the stack. In the descriptions of the program control instructions, the following phrases take special meanings.

Push the PC The SP is decremented and the contents of the TOS register are copied into the memory location referenced by the new address in the SP. The contents of the PC (which contains LOC+1, where LOC is the address of the "push" instruction) are then copied into the TOS register.

Pop the PC The contents of the TOS register are loaded into the PC. The contents of the memory location referenced by the SP are then loaded into the TOS register, and the SP is incremented.

Mnemonic: J Jump

Structure:

0	1	3	4 7	8 ~~15 16 19 20~~ N-1 ~~25~~
@	0 1	%	0 0 1 1	ADDR

Format: J @%ADDR

Function: Load the program counter (PC) with the effective address.

e → PC

Mnemonic: PUSHJ PUSH Jump

Structure:

0	1	3	4 7	8 ~~15 16 19 20~~ N-1 ~~25~~
@	0 1	%	0 1 1 1	ADDR

Format: PUSHJ @%ADDR

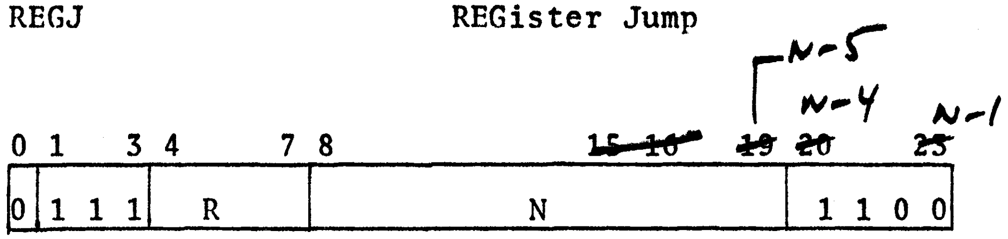
(PC of next instruction after "PUSHJ")

Function: Push the old PC onto the stack and load the PC with the effective address.

SP-1 → SP
TOS → C(SP)
(LOC+1)=PC → TOS
e → PC

Mnemonic: REGJ REGister Jump

Structure:



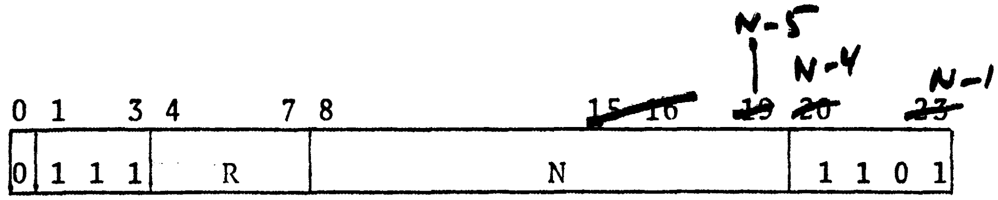
Format: REGJ R,N

Function: If an immediate value (N) is specified, it is added to the contents of Register R, and the results are loaded into the PC. Note: The immediate value N is an optional subfield and need not be specified if no offset is required. The comma, however, is required in any case.

N+(R)→ PC

Mnemonic: REGPJ REGister PushJump

Structure:



Format: REGPJ R,N

Function: Push the PC. If an immediate value N is specified, it is added to the contents of R, and the results are loaded into the PC. Note: The immediate value N is an optional subfield and need not be specified if no offset is required. The comma, however, is required in any case.

SP-1 → SP
TOS → C(SP)
PC → TOS
N+R → PC

REGJS

Mnemonic: REJS REGister Jump and pop the Stack

Structure:

N-5 N-4 N-1

0 1 3	4 7	8 ~~15 16~~ ~~19~~	~~20~~ ~~25~~
0 1 1 1	R	N	1 1 1 0

Format: REJS R,N

Function: If an immediate value (N) is specified, it is added to the contents of R, and the result is placed in the PC. The stack is then popped, which destroys the top element in the stack (i.e., the old contents of the TOS). This instruction may be thought of as a "grandfather return." Note: The N subfield is optional, however, the comma must still be present.

N+R → PC ;
C(SP) → TOS ; SP-1 → SP

Mnemonic: POPJ POPJump

Structure:

N-5 N-4 N-1

0 1 3	4 7	8 ~~15 16~~ ~~19~~	~~20~~ ~~25~~
0 1 1 1	0 1 0 0	0	1 1 1 0

Format: POPJ

Function: Pop the PC. This instruction serves as the standard subroutine return.

TOS → PC
C(SP) → TOS
SP+1 → SP

Mnemonic: POPJOF POPJump with OFset

Structure:

0 1 3	4 7	8 ~~15 16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
0 1 1 1	0 1 0 0	N	1 1 1 0

Format: POPJOF N

Function: Add the immediate value N to the contents of the TOS and pop the PC. Note: Since N is the only argument in the field, it must be present even though it may be zero.

N+TOS → PC
C(SP) → TOS
SP-1 → SP

Mnemonic: XEQ eXEcute a memory location as an insti

Structure:

0	1	3	4 7	8 ~~15 16 19 20~~ N-1 ~~23~~
@	0 1	%	1 0 1 1	ADDR

Format: XEQ @%ADDR

Function: Execute the contents of the memory location specified by the effective address as an instruction.

Mnemonic: REX Register EXecute

Structure:

0 1 3	4 7	8 ~~15 16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
1 1 0 1	R	0 0 0	0 1 1 1

Format: REX R

Function: Execute the contents of Register R as an instruction.

EXAMPLE 1: LDS-2 Addressing for a 24-bit system

If one assumes that

```
ADDR = 1056
C(ADDR) = 2736
C(2736) = 27
IR = 1
```

then

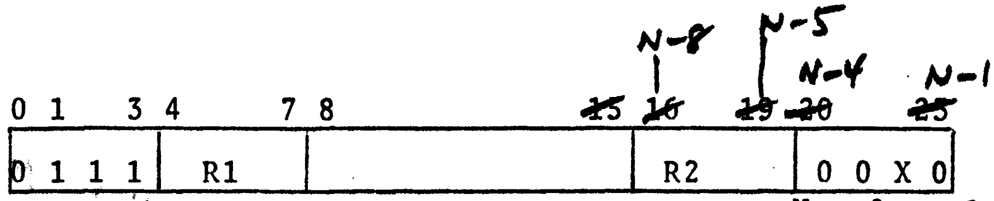
```
J  ADDR   sets the PC to 1056 ✓
J  @ADDR  sets the PC to 2736 ✓
J  %ADDR  sets the PC to 1057 ✓
J  @%ADDR sets the PC to 2737 ✓
```

7.5 Stack Control

In addition to the stack on which return locations from the program counter are saved, general-purpose stacks can be implemented easily with LDS-2 instructions. Any of the Channel Control registers may be used as a "stack pointer," and the mode of operation of the stack is under program control. The following instructions are used to implement general-purpose stacks.

Mnemonic: PUSH PUSH a register into memory

Structure:



X = 0 or 1

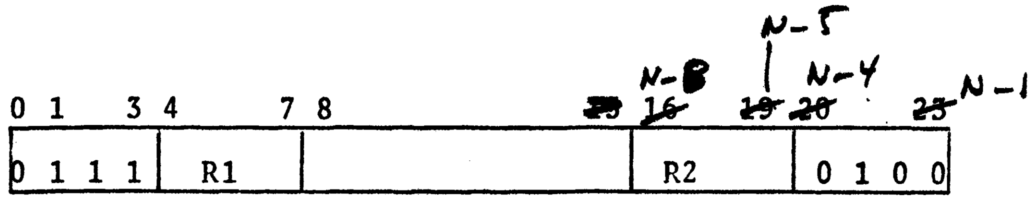
Format: PUSH R1,R2

Function: Push the contents of Register R1 into the memory location specified by the address contained in Register R2.

R1 → C(R2)

Mnemonic: IPUSH Increment and PUSH

Structure:



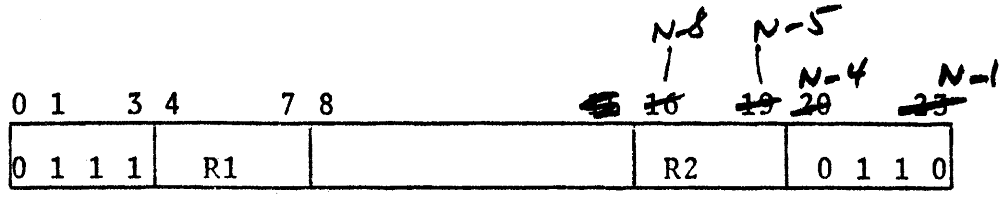
Format: IPUSH R1,R2

Function: Increment the contents of R2 by one, and push the contents of R1 into the memory location specified by the new address contained in R2.

R2+1 → R2
R1 → C(R2)

Mnemonic: PUSHI PUSH and Increment

Structure:



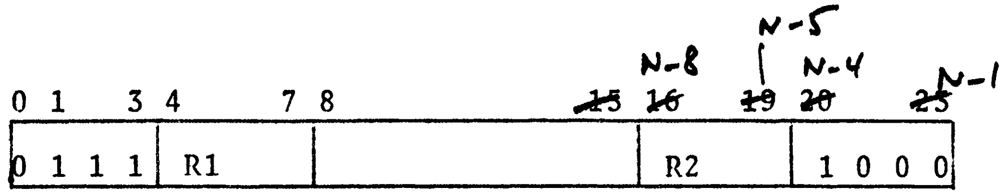
Format: PUSHI R1,R2

Function: Push the contents of Register R1 into the memory location specified by the address contained in R2 and then increment the contents of R2.

R1 → C(R2)
R2+1 → R2

Mnemonic: DPUSH Decrement and PUSH

Structure:



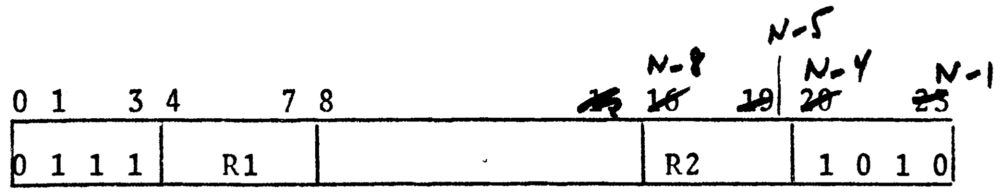
Format: DPUSH R1,R2

Function: Decrement the contents of R2 and then push the contents of R1 into the memory location specified by the new address contained in R2.

R2-1 → R2
R1 → C(R2)

Mnemonic: PUSHD PUSH and Decrement

Structure:



Format: PUSHD R1,R2

Function: Push the contents of R1 into the memory location specified by the address contained in R2 and decrement the contents of R2.

R1 → C(R2)
R2-1 → R2

Mnemonic: POP — POP a memory location into a register

Structure:

			N-8 N-5	N-4 N-1
0 1 3	4 7	8 ~~15~~	~~16~~ ~~19~~	~~20~~ ~~23~~
0 1 1 1	R1		R2	0 0 0 1

Format: POP R1,R2

Function: Pop the contents of the memory location specified by the address contained in R2 into Register R1.

C(R2) → R1

Mnemonic: IPOP — Increment and POP

Structure:

			N-8 N-5	N-4 N-1
0 1 3	4 7	8 ~~15~~	~~16~~ ~~19~~	~~20~~ ~~23~~
0 1 1 1	R1		R2	0 1 0 1

Format: IPOP R1,R2

Function: Increment the contents of Register R2 and pop the contents of the memory location specified by the new address in R2 into Register R1.

R2+1 → R2
C(R2) → R1

Mnemonic: POPI — POP and Increment

Structure:

			N-8 N-5	N-4 N-1
0 1 3	4 7	8 ~~15~~	~~16~~ ~~19~~	~~20~~ ~~23~~
0 1 1 1	R1		R2	0 1 1 1

Format: POPI R1,R2

Pop the contents of the memory location specified by the address contained in R2 into register R1 and increment the contents of R2.

C(R2) → R1
R2+1 → R2

Mnemonic: DPOP Decrement and POP

Structure:

0 1 3	4 7	8 ~~15~~	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
0 1 1 1	R1		R2	1 0 0 1

Format: DPOP R1,R2

Function: Decrement the contents of R2, and then pop the contents of the memory location specified by the new contents of R2 into Register R1.

R2-1 → R2
C(R2) → R1

Mnemonic: POPD POP and Decrement

Structure:

0 1 3	4 7	8 ~~15~~	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
0 1 1 1	R1		R2	1 0 1 1

Format: POPD R1,R2

Function: Pop the contents of the memory location specified by the address contained in R2 into Register R1 and then decrement the contents of R2.

C(R2) → R1
R2-1 → R2

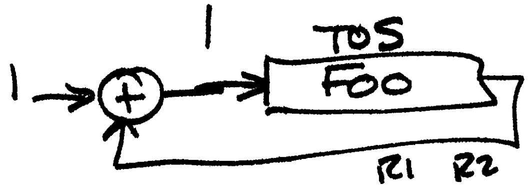
EXAMPLE 2: Subroutine Calling Sequence

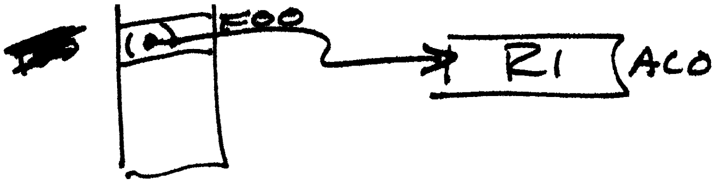
The push and pop instructions of the LDS-2 are very powerful for list processing. They also provide a nice facility for a subroutine calling sequence.

Calling Program

```
PUSHJ   SUBR
parameter 1
parameter 2
    .
    .
    .

parameter n
next instruction
```





```
SUBR    POPI    AC0,TOS    put parameter 1 in AC0
        POPI    AC1,TOS    put parameter 2 in AC1
          .
          .
          .
          .
        POPI    AC3,TOS    put parameter n in AC3
          .
          .
          .
          .
        POPJ               return to "next instruction"
```

Here we have used the TOS as a stack pointer to the parameter list and can pop the parameters from the calling program, as they are needed.

7.6 Arithmetic and Logical Operations

The arithmetic and logical operations are performed using the contents of two Channel Control registers or the contents of one register and an immediate value as arguments. Since these instructions do not have to reference memory, they are very fast. The arithmetic operations are performed using full-word, fixed-point, two's complement arithmetic. Logical operations are performed bit by bit according to the following truth tables.

OR

	0	1
0	0	1
1	1	1

XOR

	0	1
0	0	1
1	1	0

AND

	0	1
0	0	0
1	0	1

Mnemonic: ADD — ADD two registers

Structure:

0 1 3	4 7	8 15	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~25~~ N-1
0 1 1 0	R1		R2	0 0 0 0

Format: ADD R1,R2

Function: Add the contents of R1 and R2 and leave the results in R1.

R1+R2 → R1

Mnemonic: ADDNC — ADD two registers and skip on No Carryout

Structure:

0 1 3	4 7	8 ~~15~~	~~16~~ N-6 ~~19~~ N-5	~~20~~ N-4 ~~25~~ N-1
1 1 1 0	R1		R2	0 0 0 0

Format: ADDNC R1,R2

Function: Add the contents of R1 and R2 and leave the result in R1. Skip the next instruction, if the additions do not result in a carryout. This instruction is useful for double-precision arithmetic.

R1+R2 → R1
If no carryout, PC+1 → PC

Mnemonic: ADDI — ADD an Immediate value to a register

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
0 1 1 0	R	N	1 0 0 0

Format: ADDI R,N

Function: Add the immediate value N to the contents of Channel Control Register R and leave the results in R.

N+R → R

Mnemonic: ADDINC — ADD Immediate and skip on No Carryout

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
1 1 1 0	R	N	1 0 0 0

Format: ADDINC R,N

Function: Add the immediate value N to the contents of Channel Control Register R and leave the results in R. Skip the next instruction, if the addition does <u>not</u> result in carryout.

N+R → R
If no carryout, PC+1 → PC

Mnemonic: SUB — SUBtract

Structure:

0 1 3	4 7	8 ~~15~~ N-7	~~16~~ N-6 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
0 1 1 0	R1		R2	0 0 0 1

Format: SUB R1,R2

Function: Subtract the contents of Register R2 from the contents of Register R1 and leave the results in R1.

R1-R2 → R1

Mnemonic: SUBNB — SUBtract and skip on No Borrow

Structure:

0 1 3	4 7	8 ~~15~~	N-8 ~~16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
1 1 1 0	R1		R2	0 0 0 1

Format: SUBNB R1,R2

Function: Subtract the contents of R2 from the contents of R1 and leave the result in R1. Skip the next instruction, if the subtraction does not result in a borrow. SUBNB is useful for double-precision subtractions.

R1-R2 → R1
If no borrow, PC+1 → PC (skip)

Mnemonic: SUBI — SUBtract Immediate

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
0 1 1 0	R	N	1 0 0 1

Format: SUBI R,N

Function: Subtract the immediate value N from the contents of Register R and leave the results in Register R.

R-N → R

Mnemonic: SUBINB — SUBtract Immediate and skip on No Borrow

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
1 1 1 0	R	N	1 0 0 1

Format: SUBINB R,N

Function: Subtract the immediate value N from the contents of Register R and deposit the results in R. Skip the next instruction, if the subtraction does not result in a borrow.

R-N → R
If ~~no carryout,~~ PC+1 → PC (skip)

borrow

Mnemonic: OR logical OR

Structure:

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
0 1 1 0	R1		R2	0 1 0 1

Format: OR R1,R2

Function: Take the logical OR of the contents of R1 and R2 and deposit the results in R1.

R1 OR R2 → R1

Mnemonic: ORZ logical OR and skip on Zero

Structure:

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
1 1 1 0	R1		R2	0 1 0 1

Format: ORZ R1,R2

Function: Take the logical OR of the contents of Registers R1 and R2 and deposit the results in R1. Skip the next instruction, if the result is equal to zero.

R1 OR R2 → R1
If R1 = 0 PC+1 → PC (skip)

Mnemonic: XOR eXclusive OR

Structure:

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
0 1 1 0	R1		R2	0 0 1 1

Format: XOR R1,R2

Function: Take the exclusive OR of the contents of Registers R1 and R2 and deposit the results in Register R1.

R1 XOR R2 → R1

Mnemonic: XORZ — eXclusive OR and skip on Zero

Structure:

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
1 1 1 0	R1		R2	0 0 1 1

Format: XORZ R1,R2

Function: Take the exclusive OR of the contents of Registers R1 and R2 and deposit the results in R1. Skip the next instruction, if the results are equal to zero.

R1 XOR R2 → R1
If R1 = 0, PC+1 → PC (skip)

Mnemonic: XORNZ — XOR, do Not deposit results, skip on Zero

Structure:

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
1 1 1 0	R1		R2	0 1 1 0

Format: XORNZ R1,R2

Function: Take the exclusive OR of the contents of Registers R1 and R2, but do not deposit the results. Skip the next instruction, if the results are equal to zero.

If R1 XOR R2 = 0, PC+1 → PC (skip)

Mnemonic: AND — AND

Structure:

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
0 1 1 0	R1		R2	0 1 0 0

Format: AND R1,R2

Function: Take the logical AND of Registers R1 and R2 and deposit the results in Register R1.

R1 AND R2 → R1

~~Mnemonic: AND~~ ~~AND~~

~~Structure:~~

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
~~0 1 1 0~~	~~R1~~		~~R2~~	~~0 1 0 0~~

~~Format: AND R1,R2~~

~~Function: Take the logical AND of Registers R1 and R2 and deposit the results in Register R1.~~

~~R1 AND R2 → R1~~

Mnemonic: ANDZ AND and skip on Zero

Structure:

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
1 1 1 0	R1		R2	0 1 0 0

Format: ANDZ R1,R2

Function: Take the logical AND of the contents of Registers R1 and R2 and deposit the results in R1. Skip the next instruction, if the results are equal to zero.

R1 AND R2 → R1
If R1 = 0, PC+1 → PC (skip)

Mnemonic: ANDNZ AND, do Not deposit results, skip on Zero

Structure:

0 1 3	4 7	8 ~~15~~ N-9	~~16~~ N-8 ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
1 1 1 0	R1		R2	0 1 1 1

Format: ANDNZ R1,R2

Function: Take the logical AND of the contents of Registers R1 and R2 but do not deposit the results. Skip the next instruction, if the results are equal to zero.

If R1 AND R2 = 0, PC+1 → RC (skip)

EXAMPLE 3: Adding ~~two Columns of Numbers~~ Add Pairs of numbers.

Since the arithmetic and logical equations do not reference memory, it is best to use the stack mechanisms to do series of arithmetic operations.

Assume

A:		B:	
	5		4
	3		17
	4		3
	11		27

and that the WP, the WC, and the IR registers are not being used. Then

INSTRUCTION NOT used in TEXT yet.

```
                                                                  loc
ILO     IR,4         4 into count
LO      WP,=A        load WP with address of A                    +1
LO      WC,=B        load WC with address of B                    +2
LO      RP,=C        load RP with address of C                    +3
POPI    ACO,WP       A(n) → ACO, (WP)+1 -> WP                     +4
POPI    ACI,WC       B(n) → AC1   (WC)+1 -> WC                    +5
ADD     ACO,AC1      ACO+AC1 → ACO                                +6
PUSHI   ACO,RP       ACO →C(n), (RP+1) -> RP                      +7
DECE    IR           decrement count and stop, if                 +8
                     equal to zero
J       .-5          ~~to zero~~ (see Section 7.8)                +9
   .                 Jump to (loC+9)-5 = loc+4
   .
   .
   .
   .
```

will put

```
C:    9    count = 3
     20          = 2
      7          = 1
     38          = 0
```

7.7 Compare Instructions

The compare instructions allow the user to compare the contents of two registers or the contents of a register and an immediate value. There are conditional skip instructions, so that the next instruction will be skipped, if the condition specified (either equal or not equal) is satisfied.

Mnemonic: CE Compare two registers and skip if Equal

Structure:

		N-9	N-8 N-5	N-4 N-1
0 1 3	4 7	8 ~~15~~	~~16~~ ~~19~~	~~20~~ ~~23~~
1 1 0 1	R1		R2	1 0 1 0
				12

Format: CE R1,R2

Function: Compare the contents of Channel Control Registers R1 and R2 and skip the next instruction, if their contents are equal.

If R1 = R2, PC+1 → PC (skip)

Mnemonic: CNE Compare two registers and skip if Not Equal

Structure:

		N-9	N-8 N-5	N-4 N-1
0 1 3	4 7	8 ~~15~~	~~16~~ ~~19~~	~~20~~ ~~23~~
1 1 0 1	R1		R2	1 0 1 1
				13

Format: CNE R1,R2

Function: Compare the contents of Channel Control Registers R1 and R2 and skip the next instruction, if their contents are not equal.

If R1 = R2, PC+1 → PC (skip)

Mnemonic: CEMI Compare and skip if Equal to Minus Immediate value

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
1 1 0 1	R	N	1 1 1 0
			16

Format: CEMI R,N

Function: Compare the contents of Channel Control Register R with minus (two's complement) the value of the immediate N, and skip if they are equal.

If R = -N, PC+1 → PC (skip)

Mnemonic: CNEMI Compare and skip if Not Equal to Minus Immediate

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
1 1 0 1	R	N	1 1 1 1
			17

Format: CNEMI R,N

Function: Compare the contents of Channel Control Register R with minus (two's complement) the immediate value N and skip the next instruction, if they are not equal.

If R = -N, PC+1 → PC (skip)

Mnemonic: CEI Compare a register and skip if it equals the Immediate value

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
1 1 0 1	R	N	1 1 0 0
			14

Format: CEI R,N

Function: Compare the contents of Register R with the immediate value N and skip the next instruction, if they are equal.

If R = N, PC+1 → PC (skip)

Mnemonic: CNEI Compare and skip if Not Equal the Immediate value

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ ~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
1 1 0 1	R	N	1 1 0 1
			15

Format: CNEI R,N

Function: Compare the contents of Channel Control Register R with the immediate value N and skip the next instruction, if they are not equal.

If R = N, PC+1 → PC (skip)

7.8 Unary Instructions

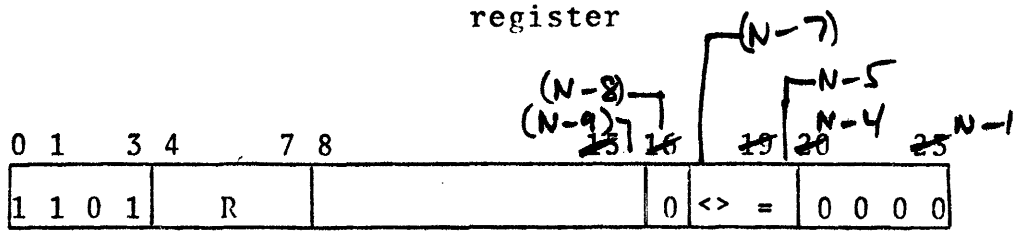
This class of instructions is referred to as the unary class, since the operations performed affect the contents of a single Channel Control register. These are also conditional skip instructions, so that if the condition specified in the mnemonic is satisfied, the next instruction is skipped. Conditions are specified by appending the following suffixes on the basic mnemonics:

E	equal
L	less than
LE	less than or equal
G	greater than
GE	greater than or equal
NE	not equal
A	always

In each case, comparison is made with an implied zero.

Mnemonic: DEC — DECrement a Channel Control register

Structure:



Format:

DEC R
DECE R
DECL R
DECLE R
DECG R
DECGE R
DECNE R
DECA R

Function: Decrement the contents of Channel Control Register R. Skip, if a condition is specified and satisfied.

R-1 → R
If condition is true, PC+1 → PC (skip)

Mnemonic: INC — INCrement a Channel Control register

Structure:

0 1 3	4 7	8 15	16	19	20 23
1 1 0 1	R		0	<> =	0 0 0 1

Format:

INC	R	INCG	R
INCE	R	INCGE	R
INCL	R	INCNE	R
INCLE	R	INCA	R

Function: Increment the contents of Channel Control Register R. Skip, if a condition is specified and satisfied.

R+1 → R
If condition is true, PC+1 → PC (skip)

Mnemonic: COM — COMplement a Channel Control register

Structure:

0 1 3	4 7	8 15	16	19	20 23
1 1 0 1	R		0	<> =	0 0 1 0

Format:

COM	R	COMG	R
COME	R	COMGE	R
COML	R	COMNE	R
COMLE	R	COMA	R

Function: Complement the contents of Channel Control Register R (one's complement). Skip, if a condition is specified and satisfied.

$\overline{R} \rightarrow R$
If condition is true, PC+1 → PC (skip)

Mnemonic: NEG — NEGate a Channel Control register

Structure:

0 1 3	4 7	8 15	16	19	20 23
1 1 0 1	R		0	<> =	0 0 1 1

Format:

NEG	R	NEGG	R
NEGE	R	NEGGE	R
NEGL	R	NEGNE	R
NEGLE	R	NEGA	R

Function: Negate the contents of Channel Control Register R (two's complement). Skip, if a condition is specified and satisfied.

-R ⟶ R
If condition is true, PC+1 ⟶ PC (skip)

Mnemonic: TST TeST a Channel Control register (NOP No OPeration)

Structure:

0 1 3	4 7	8 15	16	19	20 23
1 1 0 1	R		0	<> =	0 1 0 0

Format:

TST	R	TSTG	R
TSTE	R	TSTGE	R
TSTL	R	TSTNE	R
TSTLE	R	TSTA	R

Function: Test the contents of Register R (leaves R unchanged). Skip, if a condition is specified and satisfied. Note that TST without a condition appended is the NOP.

If condition is true, PC+1 ⟶ PC (skip)

Mnemonic: ZR ZeRo a Channel Control register

Structure:

0 1 3	4 7	8 15	16	19	20 23
1 1 0 1	R		0	<> =	0 1 0 1

Format:

ZR R
ZRE R

Function: Zero the contents of Register R. Skip, if the "A" is appended to the mnemonic.

0 ⟶ R
If ZRE, PC+1 ⟶ PC (skip)

Mnemonic: ABV

ABsolute Value of a Channel Control register

Structure:

0 1 3	4 7	8 15	16	19	20 23
1 1 0 1	R		0	<> =	0 1 1 0

Format:

ABV R
ABVE R
ABVG R
ABVGE R

Function: Take the absolute value of the contents of Channel Control Register R and skip, if a condition is specified and satisfied. Note that all conditions envolving "less than zero" are meaningless, since the test is made after the absolute value is taken. Similarly, ABVGE is equivalent to ABVA and replaces ABVA.

|R| → R
If condition is true, PC+1 → PC (skip)

Mnemonic: SLO

Switches LOad

Structure:

0 1 3	4 7	8 15	16	19	20 23
1 1 0 1	R		1	<> =	0 1 1 0

Format:

SLO R
SLOE R
SLOL R
SLOLE R

SLOG R
SLOGE R
SLONE R
SLOA R

Function: Load Register R from the data switches on the control panel, and skip the next instruction if a condition is specified and satisfied.

Switches → R
If condition is true PC+1 → PC

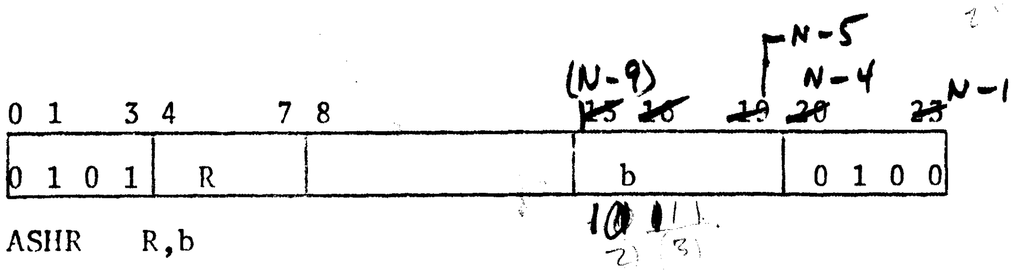
7.9 Shifting Instructions

These instructions shift the contents of the specified register either left or right the specified number of bits. Three types of shifting are available: arithmetic, logical, and circular. Arithmetic shifting right extends the sign bit on the left end of the word and shifts bits out the right end. Logical shifting right shifts zeros into the left end of the word and shifts bits out the right end. Logical shifting left shifts zeros into the right end of the word and bits out the left. Arithmetic shifting left has the same function and is the same instruction; however, two mnemonics are provided. In the above cases, all bits shifted out are lost. Circular shifting, on the other hand, cycles the bits out one end and back in the other so that no information is lost.

The logical and arithmetic shifts are also available for double registers, so that the two registers can be shifted as if they were a single register. However, the maximum number of places that can be shifted is still 23. (27) 2

Mnemonic: ASHR Arithmetic SHift Right

Structure:

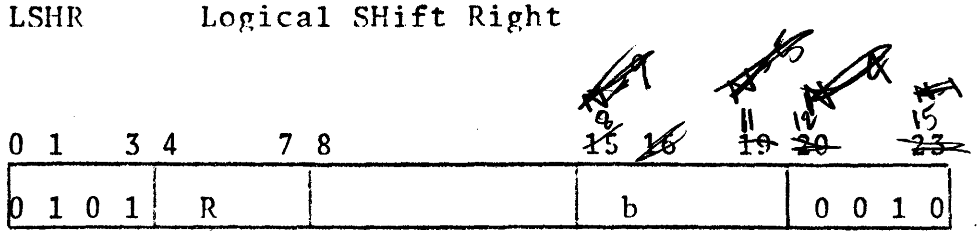


Format: ASHR R,b

Function: Shift the bits in Register R right b positions. Bits shifted out the right end of the word are lost. The sign bit (0) is extended to replace the bits shifted out of the left end of the word.

Mnemonic: LSHR Logical SHift Right

Structure:



Format: LSHR R,b

Function: Shift the bits in Register R right b positions. All bits are shifted. Bits shifted out the right end are lost, and zeros are shifted into the left end of the word.

Mnemonic: LSHL Logical SHift Left (may also be called ASHL)

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	0 0 1 1

Format: LSHL R,b

Function: Shift the bits of Register R left b bit positions. Bits shifted out the left end of the word are lost, and zeros are shifted into the right end of the word.

Mnemonic: CSHR Circular SHift Right

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	0 0 0 0

Format: CSHR R,b

Function: Shift the bits of Register R right b bit positions. Bits shifted out the left Right end of the word are shifted back into the ~~right~~ left end of the word so that no bits are lost.

Mnemonic: CSHL Circular SHift Left

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	0 0 0 1

Format: CSHL R,b

Function: Shift the bits of Register R b bit positions to the left. Bits shifted out the right left end of the word are shifted back into the left right end of the word so that no bits are lost.

Mnemonic: ASHRD Arithmetic SHift Right Double register

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	0 1 0 1

Format: ASHRD R,b

Function: Shift the bits of Registers R and R+1 to the right as though they were a single register. Bits shifted out the right of Register R are shifted into the left end of Register R+1, and bits shifted out the right end of Register R+1 are lost. The sign bit of Register R is extended to replace the bits shifted from the ~~out of~~ the left end of the word. The maximum shift is 23 bit positions.

Mnemonic: LSHRD Logical SHift Right Double register

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	0 1 1 0

Format: LSHRD R,b

Function: Shift the bits of Registers R and R+1 to the right as though they were a single register. Bits shifted from ~~out~~ the right end of Register R are shifted into the left end of Register R+1. Bits shifted out the right end of Register R+1 are lost. Zeros are shifted in the left end of Register R to replace bits shifted out. The maximum shift is 23 bit positions.

Mnemonic: LSHLD Logical SHift Left Double register (may also be called (ASHLD)

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	0 1 1 1

Format: LSHLD R,b

Function: Shift the bits of Registers R and R+1 to the left as though they were a single register. Bits shifted out the left end of Register R+1 are shifted into the right end of Register R. Bits shifted out the left end of Register R are lost. Zeros are shifted in the right end of Register R+1 to replace the bits shifted out. The maximum shift is 23 bit positions.

7.10 Masking Instructions

These two instructions mask out part of the contents of the specified register with zeros.

Mnemonic: MR Mask Right

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 1	R		b	1 0 0 0

Format: MR R,b

Function: Mask out all the bits to the right of, and including, bit b with zeros.

Mnemonic: ML Mask Left

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 1	R		b	1 0 0 1

Format: ML R,b

Function: Mask out all of the bits to the left of, and including, bit b with zeros.

EXAMPLE 4: Shifting and Masking

Assume AC2 contains 0 and AC1 contains 77777777 and that the following sequence of instructions is performed.

```
ASHR   AC1,3     AC1=77777777
LSHR   AC1,3     AC1=07777777
LSHL   AC1,4     AC1=77777760
CSHR   AC1,1     AC1=37777770          77777760
CSHL   AC1,2     AC1=77777741          77777741
ASHRD  AC1,3     AC1=77777774
                 AC2=5000000
LSHRD  AC1,3     AC1=0777777
                 AC2=7500000
LSHLD  AC1,6     AC1=7777775
                 AC2=0000000
MR     AC1,6     AC1=7700000
ML     AC1,2     AC1=0700000
```

7.11 Bit Manipulation

The instructions of this class allow the user to independently test and manipulate individual bits within a register. Bits may be set or cleared, and the next instruction may be skipped if the specified bit is either one or zero. In the cases where both the testing and the setting and clearing are performed, the testing is performed first.

Mnemonic: SOB Skip on One Bit

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	1 0 0 0

Format: SOB R,b

Function: If bit b is equal to "1", then skip the next instruction.

Mnemonic: SZB Skip on Zero Bit

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	1 0 0 1

Format: SZB R,b

Function: If bit b is equal to "0", then skip the next instruction.

Mnemonic: CLB CLear Bit

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	1 0 1 0

Format: CLB R,b

Function: Clear bit b of the register specified.

Mnemonic: SETB SET Bit

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	1 0 1 1

Format: SETB R,b

Function: Set bit b of the register specified.

Mnemonic: SOBCL Skip on One Bit and Clear

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	1 1 0 0

Format: SOBCL R,b

Function: Test bit b of Register R. Skip the next instruction, if it equals "1" and clear bit b.

Mnemonic: SZBCL Skip on Zero Bit and Clear

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	1 1 0 1

Format: SZBCL R,b

Function: Test bit b of Register R. Skip the next instruction, if it equals "0" and clear bit b.

Mnemonic: SOBSET Skip on One Bit and SET bit

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	1 1 1 0

Format: SOBSET R,b

Function: Test bit b or Register R. Skip the next instruction, if bit b is equal to "1" and set bit b.

Mnemonic: SZBSET Skip on Zero Bit and SET bit

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 1	R		b	1 1 1 1

Format: SZBSET R,b

Function: Test bit b of Register r. Skip the next instruction, if bit b is equal to "0" and set bit b.

EXAMPLE 5: List Processing Loop

The following loop allows multilevel indirection for list processing. If we assume that pointer words are marked with a 1 in bit 0 and value words have a 0 in bit 0, then

```
     POP    AC0,AC0                  [(AC0)] -> AC0
     SOB    AC0,0
     J      .-2
```

will follow the pointer words down to the value word which is then left in AC0. For processing list structures where a value word is associated with each pointer word the following code can be used.

```
BEGIN        POPI   AC1,AC0            pointer to AC1
             POP    AC2,AC0            value to AC2
              .
              .
              .
             POPI   AC0,AC1            pointer to AC0
             POP    AC2,AC1            value to AC2
              .
              .
              .
             J      BEGIN
```

In this case AC0 and AC1 alternate as pointers so that the old pointer can be use to pick up the value word.

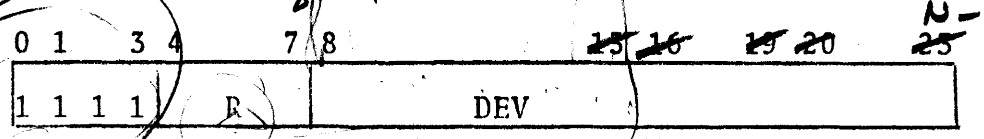
7.12 The IOT Instruction

As explained in Section 2.5, the LDS-2 has a series of registers which are treated as I/O devices. These registers may be loaded or unloaded via the IOT instruction. In addition to loading and unloading registers, the IOT instruction is used for special functions such as setting user mode, putting the LDS-2 to "sleep," or skipping on "settled" (see Section 2.5). Most of the IOT instructions are illegal, if the LDS-2 is in "user mode," so that if the user attempts to use them, the LDS-2 will be interrupted (if the mask is set). The device code bits of the illegal IOT instruction are saved in a register, so that the interrupt service routine can examine these bits and determine what to do. If the interrupt routine does not allow the device code, an interrupt will be sent to the host computer, and the job will be terminated. But if the interrupt routine knows how to service that device code, it can take appropriate action and then return control to the user's program. It is thus possible to use dummy device codes for communication between the user's program and the monitor of the LDS-2.

Mnemonic: IOT Input Output Transfer

See control V 8h 4-D4

Structure:



Format: IOT R,DEV

Function: Transfer information between Channel Control Register R and the I/O device specified by the DEV code. The DEV code also specifies the direction of the transfer. The DEV codes and the action taken are listed below.

DEV Octal Code	Function
000	Unused
001	Read Interrupt Conditions Register
002	Load Interrupt Conditions Register
3	Read Interrupt Mask Register
4	Load Interrupt Mask Register
5	Read I/O Device Code Error Register
6	Load I/O Device Code Error Register
7	Enable Interrupts (ION)
10	Set Sleep
11	Set User Mode
20	Read Switches
21	Load Lights
26	Load Sync Mask Register

27	Read Sync Mask Register
30***	Load Repeat Status Register (RSR)
31***	Read RSR
32***	Load Directive Register
33***	Read Directive Register
36***	Skip on Settled
40	Set the Attention Bit
41	Skip on Attention and Clear the Attention Bit
42	Load the Protection Register
43	Read the Protection Register
44	Clear Protection Violation
45	Read the BAR
46	Load the BAR

The following dummy codes are allowed by the interrupt handler:

370***	End of Execution String	IOT	,370 = RSTART
371***	Terminate Job Normally	IOT	,371 = STOP
372***	Input/Output Request to the Host Computer	IOT	,372 = IOR
373***	Call Software Character Generator	IOT	,373 = CHAR
374****	Disable Real-time Clock	IOT	,374 = CLKSTP
375****	Restore Clock to 30 Cycles	IOT	,374 = CLKSRT

*** Indicates that this code may be used by the user. All other codes are valid only in executive mode.

**** Available only to the highest priority user.

EXAMPLE 6: Changing the mode of the LDS-2

The following sequence of code can be used to change the mode of the LDS-2 to 2D:

```
IOT      AC0,33                     load AC0 with DIRECTIVE
CLB      AC0,5                      set bit 5 to 0
CLB      AC0,6                      set bit 6 to 0
IOT      AC0,32                     reload the DIRECTIVE
```

EXAMPLE 7: Multiply Routine

This routine multiplies two single word unsigned numbers in AC0 and AC1 and produces a single word product in AC0.

```
RLO       AC3,AC0         load multiplicand into AC3
ILO       AC2,~~24~~ 15   load counter to 24 bits
LSHL      AC0,1           shift multiplicand left
SZB       AC1,0           skip, if most significant bit is 0
ADD       AC0,AC3         accumulate product
LSHL      AC1,1           shift multiplier
DEC~~B~~L    AC2             decrement count
J         .-5             do again
POPJ                      return
```

EXAMPLE 8: Divide Routine

This routine divides the signed one-word dividend in AC0 by the signed divisor in AC1 to produce a signed quotient in AC0.

$\frac{AC0}{AC1} = AC0$

```
RLO       AC3,AC1
RLO       AC1,AC0
ZR        AC0
RLO       1R,AC3
XOR       1R,AC1
ABV       AC1
ABV       AC3
ILO       AC2,23
LSHLD     AC0,1
SUB       AC0,AC3
TSTL      AC0
J         .+3
ADD       AC0,AC3
SETB      AC1,23
DECL      AC2
J         .-7
COM       AC1
RLO       AC0,AC1
SZB       1R,0
NEG       AC0
POPJ                            return
```

7.13 Load/Unload Pipeline Registers

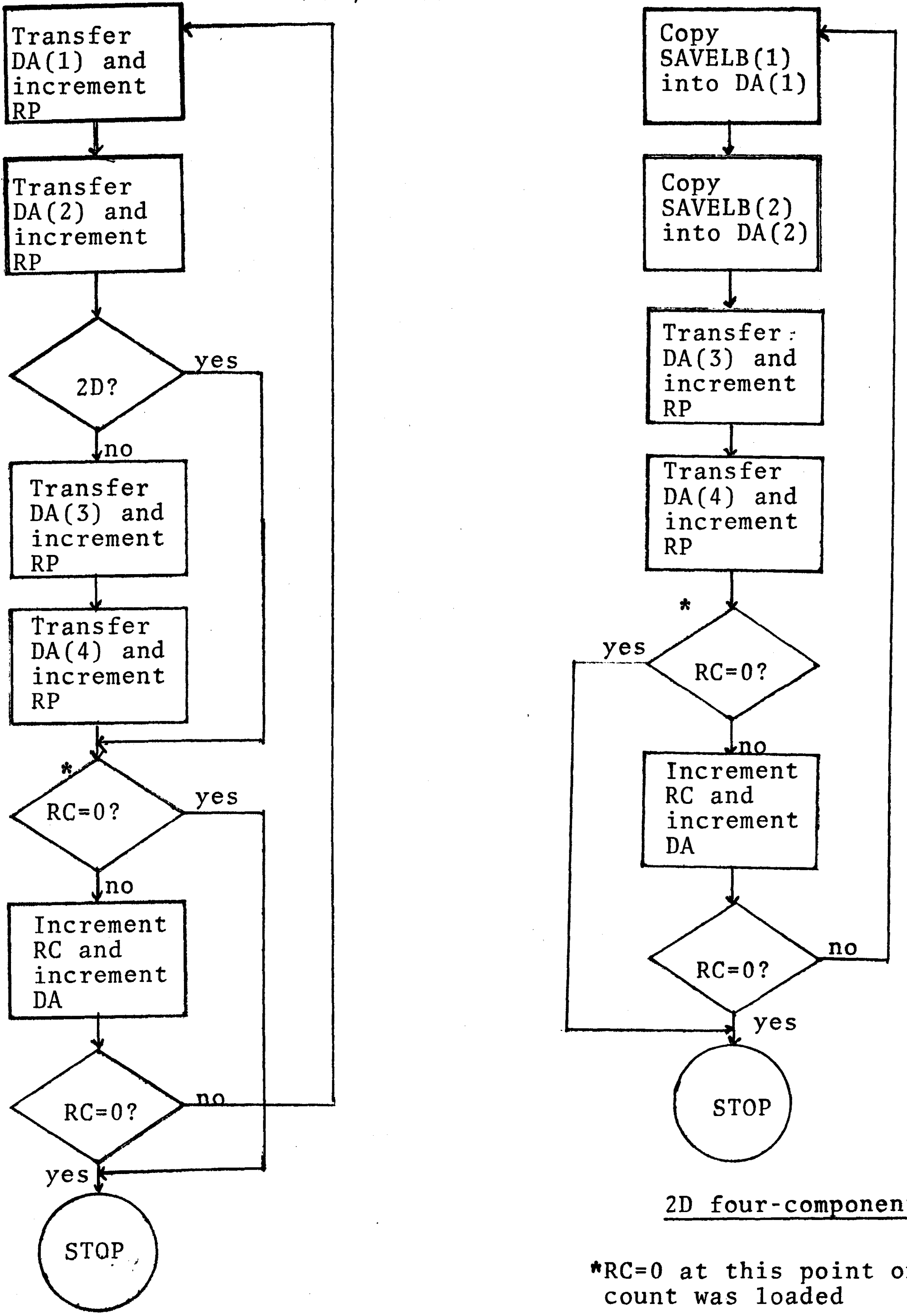
The pipeline processing units of the LDS-2 contain parameter and directive registers which control the processing performed by these units. The Channel Control sends the load/unload instruction down the pipeline and controls the transfer of data to or from the pipeline registers. Since the data in the pipeline registers affect the processing that is performed, the pipeline is allowed to settle so that all pending data will be processed before these instructions are executed. There are two general groups of these instructions: those which transfer data between the pipeline registers and memory, and those which transfer data between the pipeline registers and the Channel Control registers.

The memory load/unload instructions are inherently dynamic "repeat" instructions. It is useful to think of these instructions as transferring groups of registers, where the group may only include a single register. There are four types of register transfers: load, store, sink, and retrieve. For all these instructions the count of the number of registers to be transferred is specified by the contents of the READ COUNT (RC). If the RC contains a non-negative value, only one register will be transferred, and the RC is not incremented. If the count is negative, it is taken as the two's complement of the number of registers to be transferred and incremented after each register has been transferred. After these instructions are finished, the count will be zero (unless a positive number was initially loaded into the RC). For load and store instructions the contents of the READ POINTER (RP) are taken as the memory address into which or from which data are transferred. The RP is incremented as shown in the load/store algorithm of Figure 7.1. For sink and retrieve instructions the memory address is taken from the DATA SINK POINTER (see Section 2.4.5). The sink and retrieve algorithms are shown in Figure 7.2.

The register load/unload instructions (i.e., those which transfer data between Channel Control registers and the pipeline registers) transfer either one or two pipeline registers. Whether one or two registers are to be transferred, and which Channel Control registers will be used in the transfer, are specified in the "X" field of the register load/unload instructions. This "X" field may take on the following values:

SAC0	0	Single register beginning with AC0
SAC2	2	Single register beginning with AC2
SX	1	Single register beginning with X
SX	3	Single register beginning with Z
DAC0	4	Double register beginning with AC0

LOAD/STORE ALGORITHMS



Transfer DA(1) and increment RP
Transfer DA(2) and increment RP
2D?
yes
no
Transfer DA(3) and increment RP
Transfer DA(4) and increment RP
*
RC=0?
yes
no
Increment RC and increment DA
RC=0?
no
yes
STOP
Copy SAVELB(1) into DA(1)
Copy SAVELB(2) into DA(2)
Transfer DA(3) and increment RP
Transfer DA(4) and increment RP
*
RC=0?
yes
no
Increment RC and increment DA
RC=0?
no
yes
STOP

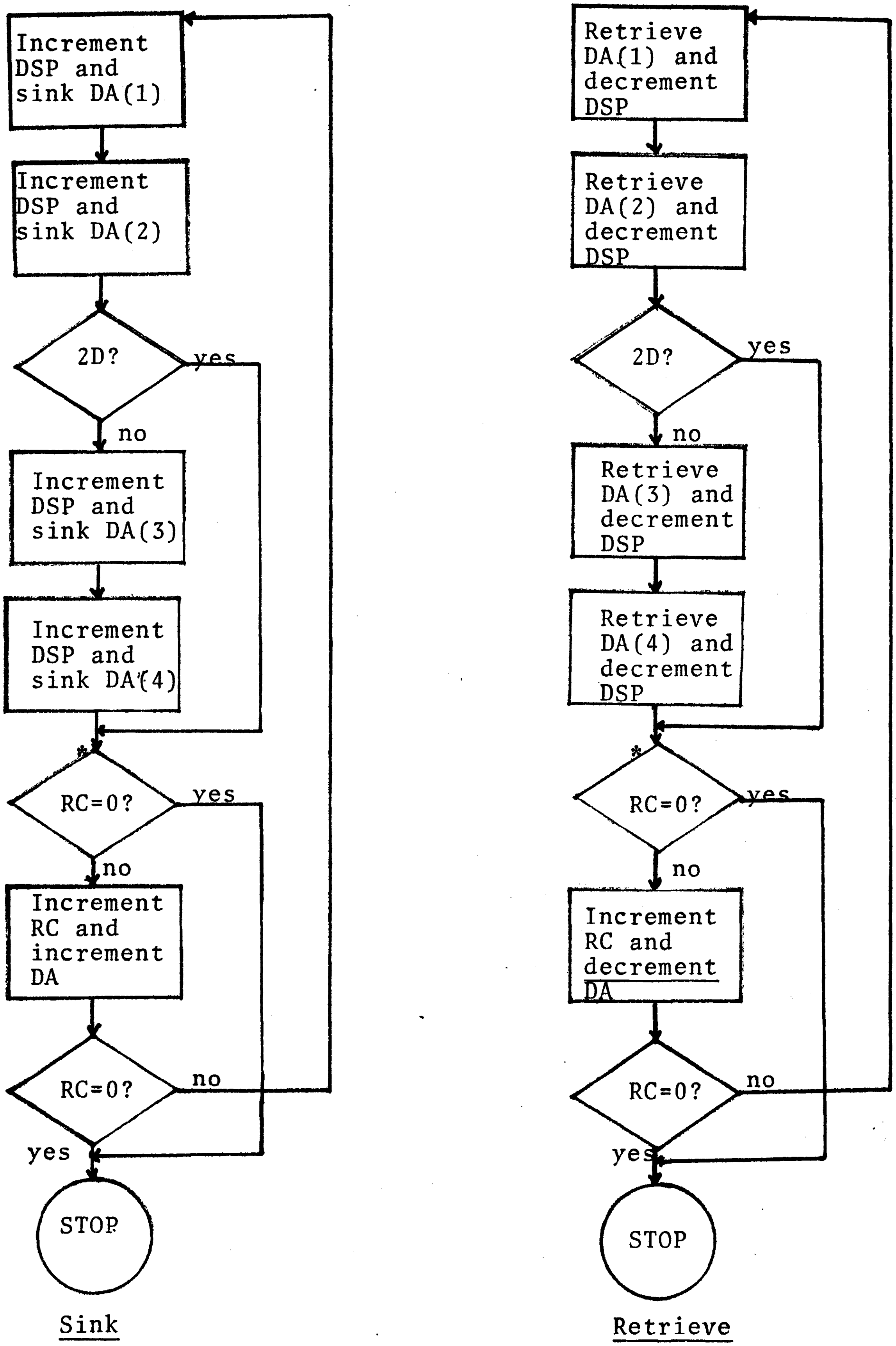
Normal 2D and 3D

2D four-component

*RC=0 at this point only if no count was loaded

Figure 7.1

SINK/RETRIEVE ALGORITHMS


Increment DSP and sink DA(1)
Increment DSP and sink DA(2)
2D?
yes
no
Increment DSP and sink DA(3)
Increment DSP and sink DA(4)
*
RC=0?
yes
no
Increment RC and increment DA
RC=0?
no
yes
STOP
Sink
Retrieve DA(1) and decrement DSP
Retrieve DA(2) and decrement DSP
2D?
yes
no
Retrieve DA(3) and decrement DSP
Retrieve DA(4) and decrement DSP
*
RC=0?
yes
no
Increment RC and decrement DA
RC=0?
no
yes
STOP
Retrieve

*RC=0 at this point only if no count was loaded.

Figure 7.2

DX 5 Double register beginning with X

For double register transfers, two consecutive pipeline registers are transferred. Register transfer instructions must be either load or store (i.e., there are no such things as register sink and retrieve instructions).

Bits 4-7 of the instruction word for all load/unload pipeline instructions constitute a "device and manner" code. The following device and manner codes are legal for the LDS-2:

CLA 0 Clipping Divider Absolute. The data are copied into or from the registers of the Clipping Divider.

CLR 1 Clipping Divider Relative (only valid for load- and retrieve-type instructions). The data are added to the Clipping Divider SAVE register, and the result is used to load the register.

CLSA 2 Clipping Divider Size Absolute. This manner is only legal for load and retrieve instructions and is only meaningful for loading four-word (four-component) Clipping Divider registers (i.e., Registers 14-17) in 2D. The incoming data are taken as a negative and positive displacement from the origin. That is, the negative of the data are loaded into the first two components, and the data are then loaded into the last two components.

CLSR 3 Clipping Divider Size Relative. The size relative manner is similar to the size absolute and has the same restrictions. The only difference is that with the size relative the data are taken as negative and positive displacements from the value in the SAVE register of the Clipping Divider.

MM 4 Matrix Multiplier Absolute. Data are simply copied into or from Matrix Multiplier registers.

MMR 5 Matrix Multiplier Relative. This manner is only legal for the load instructions. The data are first added to the old contents of the register to be loaded, and the result is then used to load the register.

MMP	6	Matrix Multiplier Product. This manner is also only legal for the load instructions. The incoming data are first multiplied by the matrix specified in the DA field (see following description), and the result is loaded into matrix A, beginning with row 0.
MDR	7	Matrix Multiplier Directive Register. The Matrix Multiplier Directive register is a two-word register which is treated as if it were a separate pipeline device. Data are transferred in absolute form.

Certain "illegal" combinations of instructions and device and manner codes are used for special operations of the Matrix Multiplier. A store instruction with a device and manner code of 5 is used for the "normalize" instruction, and with a device and manner code of 6 is used for a "push Matrix Multiplier" instruction. A sink instruction with a device and manner code of 5 is a "sink and slide" instruction. A retrieve instruction with a device code of 5 means "retrieve and slide," and with a device code of 6 means "pop Matrix Multiplier." Special mnemonics have been defined for all of these instructions.

The device address field (DA) of the load/unload instructions specifies the register within the device with which the transfer will begin. The register addresses for the Matrix Multiplier are simply the row numbers of the matrices. These registers are two words long, if the LDS-2 is in 2D; otherwise, they are four words long. Most of the register of the Clipping Divider can be addresses by two different addresses. Register $0\text{-}13_8$ are two-word registers (see Figure 4.1), and registers $14\text{-}17_8$ are four-word register addresses for Registers 0-7. Normally, two-word register addresses are used, when the LDS-2 is in 2D mode, and four-word addresses are used in the 3D modes. The major exception to this is when size absolute or size realtive loads are performed and when 2D four-component loads are performed (usually in preparation to boxing instructions; see the 2D four component load algorithm and Example 11).

The dimension mode of the LDS-2 determines how many words of data are sent down the pipeline for each register transferred. If the LDS-2 is in 2D mode, two words are transfered; otherwise, four words of data are sent down the pipeline. The programmer must, therefore, be careful to match his load/unload instruction addresses to the current mode of the LDS-2. If, for example, the LDS-2 is in one of the 3D modes and the user attempts to load one of the two component registers of the Clipping Divider, four words of data will get loaded into the two-word register, which will result in the last two words being written over the top of the first two.

Mnemonic: LOCLA LOad CLipping divider Absolute

Structure:

0 1 3	4 7	8 ~~15~~	N-8 ~~16~~ N-5 ~~19~~	N-4 ~~20~~ N-1 ~~23~~
0 1 0 0	0 0 0 0		DA	0 1 1 0

Format: LOCLA DA

Function: Load Clipping Divider register(s) with absolute data, starting with Register DA and continuing according to the load algorithm (see Figure 7.1).

Mnemonic: LOCLR LOad CLipping Divider Relative

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 0 1 0		DA	0 1 1 0

Format: LOCLR DA

Function: Load Clipping Divider Register(s) with relative data, starting with Register DA and continuing according to the "load" algorithm (see Figure 7.1). Relative data are added to the contents of the Clipping Divider SAVE registers to form the sum which is actually loaded into the register.

Mnemonic: LOCLSA LOad CLipping divider Size Absolute

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 0 1 0		DA	0 1 1 0

Format: LOCLSA DA

Function: Load Clipping Divider registers with size absolute data (see Figure 7.1).

Mnemonic: LOCLSR LOad CLipping divider Size Relative

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 0 1 1		DA	0 1 1 0

Format: LOCLSR DA

Function: Load Clipping Divider registers with size relative data (see Figure 7.1).

Mnemonic: LOMMA LOad Matrix Multiplier Absolute

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 1 0 0		DA	0 1 1 0

Format: LOMMA DA

Function: Load Matrix Multiplier register(s) with absolute data from memory, beginning with Matrix Multiplier Register DA and continuing according to the load algorithm (see Figure 7.1).

Mnemonic: LOMMR LOad Matrix Multiplier Relative

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 1 0 1		DA	0 1 1 0

Format: LOMMR DA

Function: Load Matrix Multiplier register(s) with relative data from memory, beginning with Matrix Multiplier Register DA and continuing according to the load algorithm (see Figure 7.1). Relative data for the Matrix Multiplier registers are added to the old data contained in the respective registers to calculate the sum that is actually loaded into the registers.

Mnemonic: LOMMP LOad Matrix Multiplier Product

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 1 1 0		DA	0 1 1 0

Format: LOMMP DA

Function: Load Matrix Multiplier registers with the matrix product of the matrix which begins with Register DA and the data from memory, and store the resulting product in matrix A, beginning with Register 0. Note: In most cases, DA should be either 4, 10, or 14 (octal), and the count in the RC should be -4. This causes a complete matrix to be multiplied by the incoming matrix to give a matrix product. This is true both in 2- and 3-dimensional modes. Since the product is stored in matrix A, a DA of 0 should not be specified with a LOMMP.

Mnemonic: LOMDR LOad Matrix multiplier DiRective register

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 1 1 1		0	0 1 1 0

Format: LOMDR

Function: Load the directive register of the Matrix Multiplier according to the load algorithm (see Figure 7.1).

Mnemonic: STCL STore CLipping divider

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 0 0 0		DA	0 1 1 0

Format: STCL DA

Function: Store the contents of registers in the Clipping Divider, beginning with Register DA and continuing according to the store algorithm (see Figure 7.1).

Mnemonic: STMM STore Matrix Multiplier

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 1 0 0		DA	0 1 1 0

Format: STMM DA

Function: Store the contents of Matrix Multiplier registers into memory, beginning with Register DA and continuing according to the store algorithm (see Figure 7.1).

Mnemonic: STMDR STore Matrix multiplier Directive Register

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 1 1 1		0	0 1 1 0

Format: STMDR

Function: Store the first half of the Matrix Multiplier Directive register into the memory location addressed by the contents of the RP. The RP is then incremented automatically, and the second half of the Directive register is stored into the memory location addressed by the new contents of the RP. Note: The RC should contain a count of zero or -1 before this instruction is executed, or the contents of the Matrix Multiplier Directive register will be recorded more than once.

Mnemonic: RLOCLA Register LOad CLipping Divider Absolute

Structure:

0 1 3	4 7	8 15	16 19	20	23
0 1 0 0	0 0 0 0		DA	0	X

Format: RLOCLA DA,X

Function: Load the Clipping Divider Register DA with data from the Channel Control registers specified by X. In 2D, two registers are transferred per coordinate point, and in the 3D modes four registers are transferred per coordinate point.

Mnemonic: RLOCLR Register LOad Clipping Divider Relative

Structure:

0 1 3	4 7	8 15	16 19	20	23
0 1 0 0	0 0 0 1		DA	0	X

Format: RLOCLR DA,X

Function: Load the Clipping Divider Register DA with data from the Channel Control register specified by X. Since the load is relative, the data are first added to the contents of the Clipping Divider SAVE registers, and the sum is loaded into the Register DA.

Mnemonic: RLOMMA Register LOad Matrix Multiplier Absolute

Structure:

0 1 3	4 7	8 15	16 19	20	23
0 1 0 0	0 1 0 0		DA	0	X

Format: RLOMMA DA,X

Function: Load either one or two Matrix Multiplier registers, starting with DA, with absolute data from Channel Control registers specified by X.

Mnemonic: RLOMMR — Register LOad Matrix Multiplier Relative

Structure:

0 1 3	4 7	8 15	16 19	20	23
0 1 0 0	0 1 0 1		DA	0	X

Format: RLOMMR DA,X

Function: Load either one or two registers of the Matrix Multiplier with relative data from the Channel Control registers specified by X. The data are first added to the old data in the corresponding Matrix Multiplier registers, and the sum is used to load the registers.

Mnemonic: RLOMMP — Register LOad Matrix Multiplier Product

Structure:

0 1 3	4 7	8 15	16 19	20	23
0 1 0 0	0 1 1 0		DA	0	X

Format: RLOMMP DA,X

Function: Load either one or two Matrix Multiplier registers (depending on X), beginning with Register 0, with the product of the contents of the Channel Control registers specified by X and the matrix which begins with DA.

Mnemonic: RLOMDR — Register Load Matrix multiplier Directive Register

Structure:

0 1 3	4 7	8 15	16 19	20	23
0 1 0 0	0 1 1 1		DA	0	X

Format: RLOMDR X

Function: Load the directive register of the Matrix Multiplier with the contents of the registers specified by X. If the mode of the LDS-2 or the value in the X field cause more than two registers to be transferred, the Matrix Multiplier Directive register will contain the last data loaded into it.

Mnemonic: RSTCL — Register STore CLipping divider

Structure:

0 1 3	4 7	8 15	16 19	20	23
1 1 0 0	0 0 0 0		DA	0	X

Format: RSTCL DA,X

Function: Store Clipping Divider register(s), beginning with DA, into the Channel Control registers specified by X.

Mnemonic: RSTMM — Register STore Matrix Multiplier

Structure:

0 1 3	4 7	8 15	16 19	20	23
1 1 0 0	0 1 0 0		DA	0	X

Format: RSTMM DA,X

Function: Store either one or two registers, beginning with DA, from the Matrix Multiplier into Channel Control registers specified by X.

Mnemonic: RSTMDR — Register STore Matrix multiplier Directive Register

Structure:

0 1 3	4 7	8 15	16 19	20	23
1 1 0 0	0 1 1 1		DA	0	X

Format: RSTMDR X

Function: Store the Matrix Multiplier Directive register into the Channel Control registers specified by X. If the combination of the mode of the LDS-2 and the value in the X field cause more than two registers of the Channel Control to receive data from the Matrix Multiplier Directive registers, several copies of the directive will be saved.

Mnemonic: RTCLA

ReTrieve CLipping divider Absolute

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 0 0 0		DA	0 1 1 1

Format: RTCLA DA

Function: Retrieve information from the data sink according to the retrieve algorithm into Clipping Divider registers (see Figure 6.2), beginning with DA.

Mnemonic: RTCLR

ReTrieve CLipping divider Relative

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 0 0 1		DA	0 1 1 1

Format: RTCLR DA

Function: Retrieve relative data from the data sink according to the retrieve algorithm into Clipping Divider registers, beginning with DA. The relative data are added to the contents of the Clipping Divider SAVE registers, and the sum is loaded in the registers. Note: Since data were sinked into the data sink in absolute format, one should not expect to get the same data back when using a relative retrieve.

Mnemonic: RTCLSA

ReTrieve CLipping divider Size Absolute

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 0 1 0		DA	0 1 1 1

Format: RTCLSA DA

Function: Retrieve Clipping Divider registers interpreting the data as size absolute (see Figure 7.2).

Mnemonic: RTCLSR ReTrieve CLipping divider Size Relative

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 0 1 1		DA	0 1 1 1

Format: RTCLSR DA

Function: Retrieve Clipping Divider registers interpreting the data as size-relative. (See Figure 7.2).

Mnemonic: RTMDR ReTrieve Matrix multiplier Directive Register

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 1 1 1		DA	0 1 1 1

Format: RTMDR

Function: Retrieve information from the data sink into the Matrix Multiplier Directive register.

Mnemonic: SKCL SinK CLipping divider

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 0 0 0		DA	0 1 1 1

Format: SKCL DA

Function: Sink the contents of Clipping Divider registers, beginning with DA, into the data sink. (See Figure 7.2.)

Mnemonic: RTMM ReTrieve Matrix Multiplier

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 1 0 0		DA	0 1 1 1

Format: RTMM DA

Function: Retrieve absolute data from the data sink. (See Figure 7.2.)

Mnemonic: RTMMS ReTrieve Matrix Multiplier and Slide

Structure:

0 1 3	4 7	8 15	16 19	20 23
0 1 0 0	0 1 0 1		DA	0 1 1 1

Format: RTMMS DA

Function: Retrieve absolute data from the data sink into Matrix Multiplier registers, beginning with DA, but before each load copy the old data into the corresponding row of matrix A.

Mnemonic: SKMM SinK Matrix Multiplier

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 1 0 0		DA	0 1 1 1

Format: SKMM DA

Function: Sink the contents of Matrix Multiplier registers, beginning with DA, into the data sink.

Mnemonic: SKMMS — SinK Matrix Multiplier and Slide

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 1 0 1		DA	0 1 1 1

Format: SKMMS DA

Function: Sink the contents of Matrix Multiplier registers, beginning with DA, into the data sink. After each register has been sinked, its contents are replaced with the contents of the corresponding row of matrix A. The contents of matrix A remain unchanged.

Mnemonic: SKMDR — SinK Matrix multiplier Directive Register

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 1 1 1		DA	0 1 1 1

Format: SKMDR

Function: Sink the contents of the Matrix Multiplier Directive register into the data sink.

Mnemonic: NOMM — NOrmalize the Matrix Multiplier

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 1 0 1		DA	0 1 1 0

Format: NOMM

Function: Normalize the Matrix Multiplier by shifting the data in its registers left the maximum number of positions or until some data word takes on a value between one half and one (i.e., the most significant bit is a 1). The maximum number of positions the words should be shifted is specified by the contents of the RC. If this count is zero, the words will only be shifted one place. The count in the RC will always be zero after the normalize instruction has been executed.

Mnemonic: PUSHMM PUSH Matrix Multiplier

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 1 1 0		DA	0 1 1 1

Format: PUSHMM DA

Function: Copy the contents of Matrix Multiplier registers, beginning with Register 0, into Matrix Multiplier registers, beginning with DA.

Mnemonic: POPMM POP Matrix Multiplier

Structure:

0 1 3	4 7	8 15	16 19	20 23
1 1 0 0	0 1 1 0		DA	0 1 1 0

Format: POPMM DA

Function: Copy the contents of Matrix Multiplier registers, beginning with DA, into Matrix Multiplier registers, beginning with Register 0.

EXAMPLE 9: Manipulating the Pipeline Registers

Assume that ARRAY1 and ARRAY2 contain 16 words, and that ARRAY3 contains 8 words, then in 3D:

```
     LO    RP,=ARRAY1
     ILOM   RC,4
     LOMMA    0               loads the four rows of Matrix A (begin-
                              ning with Row 0) with ARRAY1.

     ILOM   RC,4
     PUSHMM    4              copies Matrix A into Matrix B (begin-
                              ning with Row 4).

     LO    RP,=ARRAY2
     ILOM    RC,4
     LOMMP    4               multiplies [ARRAY2] [ARRAY1] and leaves
                              the product in Matrix A.  ARRAY1 is still
                              in Matrix B.

     LO    RP,=ARRAY3
     ILOM   RC,2
     LOCLA    VIEW            loads the VIEWPORT and WINDOW registers
                              with data from ARRAY3.
```

If the LDS-2 is in 2D:

```
     LO    RP,=ARRAY3
     ILOM   RC,4
     LOCLA    VIEWLB          loads the VIEWPORT and WINDOW registers
                              with data from ARRAY3.  (Note, that in
                              2D there are four registers.)

     LO    RP,=ARRAY3
     LOCLA    VIEWLB          loads the first half of the VIEWPORT
                              with the first two words of ARRAY3.
                              (Since no count was specified, only
                              one register was transferred.)

     LO    DSP,=SAVE
     ILOM   RC,4
     SKCL   VIEWLB            sinks the VIEWPORT and WINDOW registers
                              into memory at SAVE.

     ILOM   RC,4
     RTCLA  WINDRT            retrieves the VIEWPORT and WINDOW regis-
                              ters.  (Note, that the registers are
                              retrieved "backwards," so that the last
                              register sinked is the first retrieved.)
```

7.14 Drawing Instructions

The drawing instructions of the LDS-2 provide a great variety of ways to address data, to interpret data, and to move the beam. The six basic drawing instructions access data in different manners. The arguments of these instructions generally specify the movement of the beam and the absolut/relative/variable origin modes of interpreting the coordinate data. There are three sets of these arguments. The "single draw" instructions take a "manner" argument (MAN) which is interpreted as shown in Figure 7.3.

The "table draw" instructions rely on two sets of "finite-state machines." A series of drawing operations are performed by each instruction. Each time a drawing operation has been performed, both FSM1 and FSM2 are updated, as shown in Figures 7.4 and 7.5. To interpret the state graphs in these figures, it is useful to think of a marker that is placed on the state addressed by the FSM argument and then moved after each iteration following the arrows. For example, if FSM1 is POLY, then the finite-state machine will start in State 3 and issue a "setpoint" command to the pipeline. Then the finite state machine will then go to the next state, which in this case is 2, and a "draw to" command will be issued to the pipeline. Since State 2 goes to itself, the finite state machine will stay in that state and continue issuing "draw to commands" to the pipeline. The absolute/relative/variable origin finite-state machine works in exactly the same way.

The number of iterations performed by the reapeat drawing instructions is determined by the count contained in the READ COUNTER (RC). The RC should contain the two's complement of the number of operations to be performed. If the count is zero (or positive), only one iteration will be performed, and the count will not be incremented.

The "Matrix Multiplier draws" are used to draw curves and surface patches with the Matrix Multiplier. FSM1 operates in the same manner as in the table draw case, but for these instructions FSM2 is defined to be AA (2), so that the coordinate data are interpreted as absolute. It should be realized that the coordinate data for these drawing operations do not come from memory, but rather are provided by iterations of the Matrix Multiplier. For these instructions the Matrix Multiplier must be put in curve mode by loading the MDR.

The "register draw" instructions fetch data from the internal registers of the Channel Control rather than from memory. Both of the finite-state machines are used, but there can be only one or two iterations performed. The "X" argument of these instructions specifies whether it is a single point draw (i.e., one iteration) or a double point draw (i.e., two

MANNER CODES

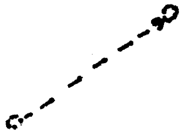
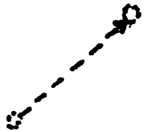
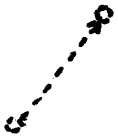
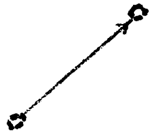
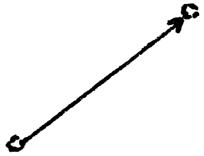
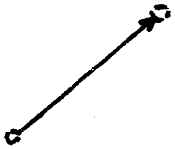
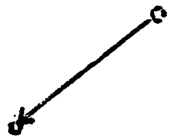
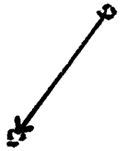
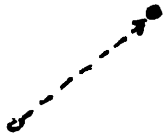
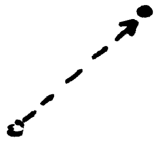
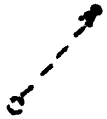
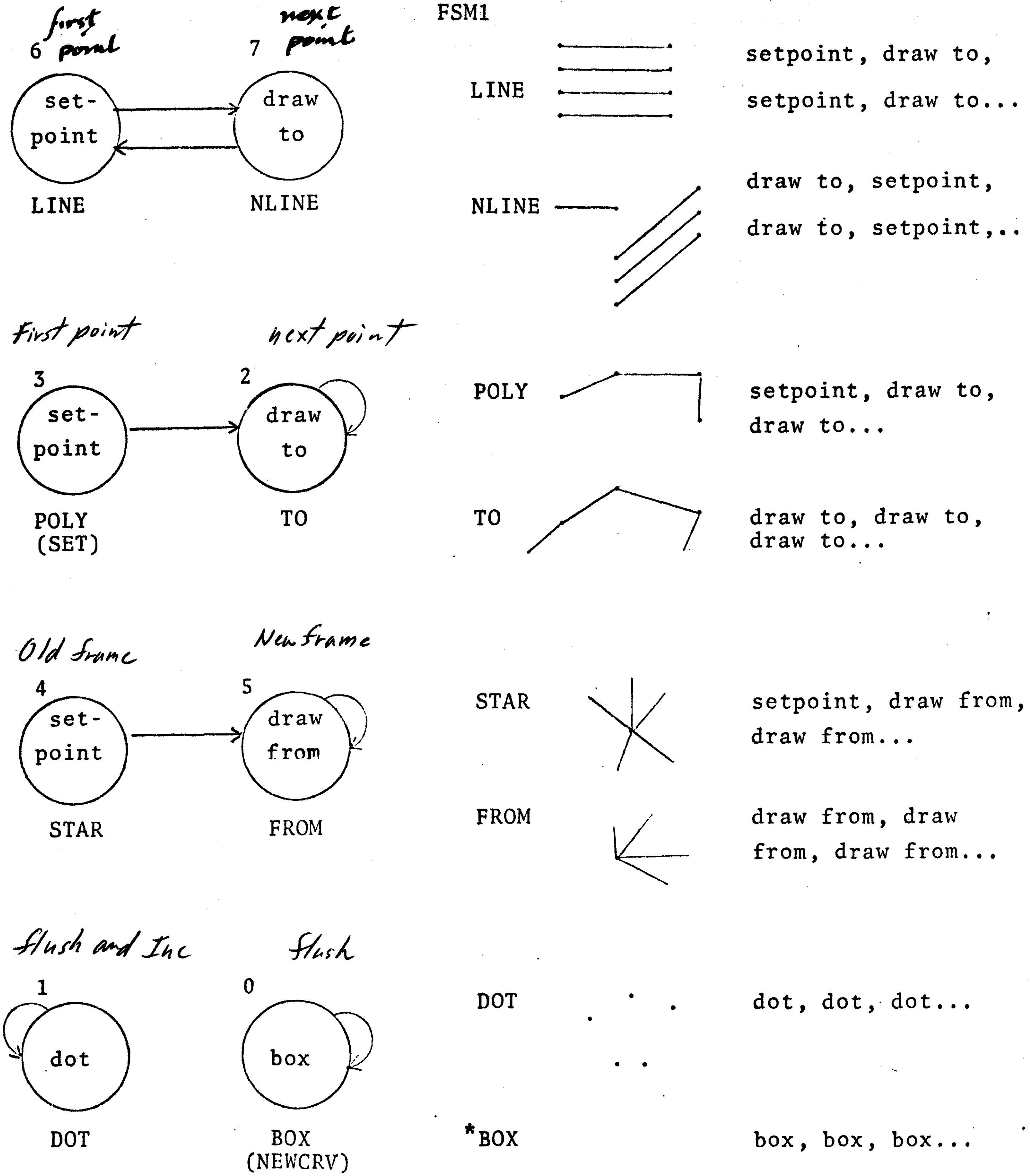
SETA = 0  $X_1,Y_1,(Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_1, Y_1, (Z_1)$	SETR = 1  $X_o+\Delta X_1, Y_o+\Delta Y_1, (Z_o+ \Delta Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_o+\Delta X_1, Y_o+\Delta Y_1, (Z_o +\Delta Z_1, Z_o+\Delta Y_1)$	SETV = 2  $X_o+\Delta X_1, Y_o+\Delta Y_1, (Z_o+\Delta Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_o, Y_o, (Z_o, Z_o)$	TOA = 4  $X_1, Y_1, (Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_1, Y_1, (Z_1, Z_1)$
TOR = 5  $X_o+\Delta X_1, Y_o +\Delta Y_1, (Z_o+ \Delta Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_o+\Delta X_1, Y_o+\Delta Y_1, (Z_o+ Z_1)$	TOV = 6  $X_o+\Delta X_1, Y_o+ \Delta Y_1, (Z_o+\Delta Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_o, Y_o, (Z_o, Z_o)$	FRMA = 16  $X_1, Y_1, (Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_o, Y_o, (Z_o, Z_o)$	FRMR = 17  $X_o+\Delta X_1, Y_o+ \Delta Y_1, (Z_o+\Delta Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_o, Y_o, (Z_o, Z_o)$
DOTA = 10  $X_1, Y_1, (Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_1, Y_1, (Z_1, Z_1)$	DOTR = 11  $X_o+\Delta X_1, Y_o+ \Delta Y_1, (Z_o+\Delta Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_o+\Delta X_1, Y_o+\Delta Y_1, (Z_o +\Delta Z_1, Z_o+\Delta Z_1)$	DOTV = 12  $X_o+\Delta X_1, Y_o+ \Delta Y_1, (Z_o+\Delta Z_1)$ $X_o, Y_o, (Z_o)$ SAVE: $X_o, Y_o, (Z_o, Z_o)$	○ = Position of beam The arrows are for expository purpose to indicate the direction of beam motion and do not actually appear on the scope. The Z coordinate applies, if in one of the 3D modes. BOXA =14 BOXR=15

Figure 7.3

THE "WHAT-TO-DO" CODES

FSM1



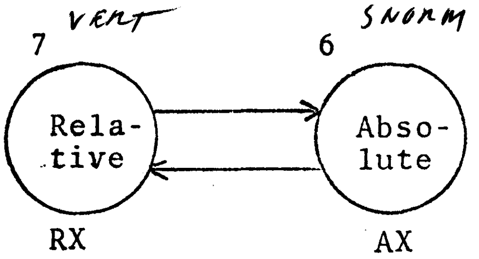
* Box does not move the beam, but rather sets up the parameters for subpictures (see Section 4.6)

Figure 7.4

SURFACE MODES IN IMC

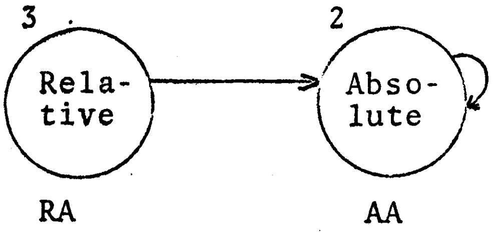
THE DATA FORM CODES

FSM2



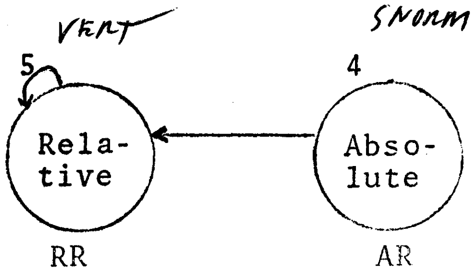
RX relative, absolute, relative...

AX absolute, relative, absolute...



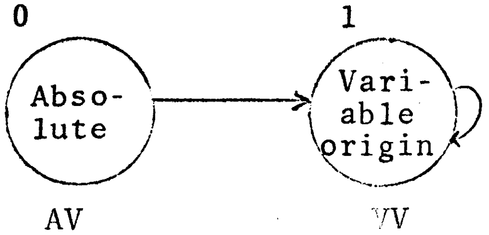
RA relative, absolute, absolute...

AA absolute, absolute, absolute...



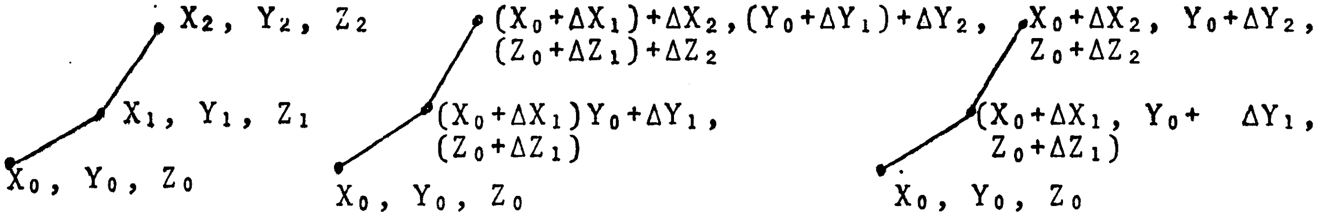
RR relative, relative, relative...

AR absolute, relative, relative...



AV absolute, variable origin, variable origin...

VV variable origin, variable origin, variable origin...


X_2, Y_2, Z_2
X_1, Y_1, Z_1
X_0, Y_0, Z_0
$(X_0+\Delta X_1)+\Delta X_2, (Y_0+\Delta Y_1)+\Delta Y_2, (Z_0+\Delta Z_1)+\Delta Z_2$
$(X_0+\Delta X_1) Y_0+\Delta Y_1, (Z_0+\Delta Z_1)$
X_0, Y_0, Z_0
$X_0+\Delta X_2, Y_0+\Delta Y_2, Z_0+\Delta Z_2$
$(X_0+\Delta X_1, Y_0+ \Delta Y_1, Z_0+\Delta Z_1)$
X_0, Y_0, Z_0

ABSOLUTE RELATIVE VARIABLE ORIGIN

Figure 7.5

iterations) and the address of the first Channel Control register from which data are to be taken.

In all cases except the "Matrix Multiplier draws," the number of words of data that are fetched per coordinate point is determined by the dimension mode of the LDS-2. For register draws, two registers are transferred in 2D, and four registers are transferred in all the 3D modes. For the single draw and table draw classes, there are two words fetched in 2D, three words fetched in the CD3D and MM3D modes, and four words fetched in the HOMOG mode. See Figure 7.6.

Mnemonic: SD Single Draw

Structure:

0 1 3	4 7	8 ~~15~~ ~~16~~ ~~19~~ ~~20~~ ~~23~~ N-1
0 0 1 0	MAN	ADDR

Format: SD MAN,@%ADDR

Function: Execute a single draw instruction fetching the data from the memory location referenced by the effective address. The MAN argument specifies the manner of the drawing instruction.

Mnemonic: TDR Table Draw Repeat

Structure:

0 1 3	4	7	8 ~~15~~	~~16~~ N-8	~~19~~ N-5	~~20~~ N-4 ~~23~~ N-1
0 1 0 0	0	FSM1		1	FSM2	1 1 1 0

Format: TDR FSM1,FSM2

Function: Execute a repeated series of drawing instructions fetching data from the memory locations addressed by the RP. The count in the RC specifies the number of operations to be performed, and the arguments FSM1 and FSM2 specify the type of operations to be performed.

DATA FORMATS FOR DRAWING INSTRUCTIONS

2D

X
Y

CD3D

X
Y
Z

The fourth word needed by the pipeline is a copy of Z to give X, Y, Z, Z, which is the format the Clipping Divider expects.

MM3D

X
Y
Z

The fourth word needed by the pipeline is a fraction approximation of 1 (i.e., all one's or 2^{23}-1), which provides a homogeneous component of "1" for the Matrix Multiplier.

HOMOG

X
Y
Z
W

W is the homogeneous element (see Appendix 2).

Figure 7.6

Mnemonic: TDIR Table Draw Indirect Repeat

Structure:

0 1 3	4	7	8 15	16	19	20 23
1 1 0 0	0	FSM1		X	FSM2	1 1 1 0

Format: TDIR FSM1,FSM2

Function: Execute a repeated series of drawing operations fetching data from the memory locations obtained by taking the contents of the memory locations addressed by the RP as addresses.

Mnemonic: TDIXR Table Draw Indirect and indeXed Repeat

Structure:

0 1 3	4	7	8 15	16	19	20 23
1 1 0 0	1	FSM1		X	FSM2	1 1 1 0

Format: TDIXR FSM1,FSM2

Function: Execute a repeated series of drawing operations as specified by arguments FSM1,FSM2. The effective address for the coordinate data is determined by taking the contents of the memory location addressed by the RP and adding the contents of the index register (IR).

Mnemonic: RD Register Draw

Structure:

0 1 3	4	7	8 15	16	19	20	23
0 1 0 0	0	FSM1		0	FSM2	1	X

Format: RD FSM1,FSM2,X

Function: Execute either one or two drawing operations (dependent on the value of X) according to the arguments FSM1,FSM2. Data for these operations are fetched from the registers of the Channel Control as specified by the X field.

Mnemonic: MMDR Matrix Multiplier Draw Repeat

Structure:

0 1 3	4	7	8 15	16	19	20 23
0 1 0 0	0	FSM2		1	0 1 0	1 1 1 1

Format: MMDR FSM1

Function: Execute a repeated series of drawing operations as indicated by FSM1 using data obtained by iteration of the Matrix Multiplier. FSM2 is defined to be equal to AA (2). The count for these instructions is held in the RC and incremented each iteration of the Matrix Multiplier (which corresponds to each individual drawing operation).

EXAMPLE 10: House Plan

The following routine will draw the outline of a simple 2D House Plan (see Figure 8.3):

```
     LO    SP,=SAVE
     IOT    AC0,33
     CLB    AC0,2
     IOT    AC0,32           turn off Matrix Multiplier
     LO    RP,=CLIP1         CLIP1 contains VIEWPORT and WINDOW data
     ILOM    RC,4
     LOCLA    VIEWLB         set VIEWPORT and WINDOW
     LO    RP,=PLAN          PLAN contains the drawing coordinates
     ILOM    RC,13
     TDR    POLY,AV          draw house plan
```

EXAMPLE 11: Boxing

Boxing may be used to draw subpictures at different locations on the picture. This routine draws symbols for a window in the house plan (see Figure 8.3). I1 contains the X and Y coordinates of the position of the window to be drawn:

```
     LO    RP,=I1
     LOCLA    SAVELB         left bottom corner of instance
     LOCLA    INST           set up instance
     IOT    ,42              skip until settled
     J    .=1
     IOT    AC0,33           read directive
     SZB    AC0,22           check area in common
     PUSHJ    WINDOW         jump to WINDOW routine
     POPJ                    return

WINDOW   LO    DSP,=SINK
     ILOM    RC,4
     SKCL    VIEWLB          save old WINDOW and VIEWPORT
     SD    SETA, MASTER
     SD    BOXA,MASTER+2     box to set up new parameters
     LO    RP,=W1
     ILOM    RC,5
     TDR    POLY,AA          draw frame
     ILOM    RC,4
     TDR    LINE,AA          draw cross piece
     RTCL    WINDRT          restore old WINDOW and VIEWPORT
     POPJ                    return
```

Note, that the instance is loaded with a 2D four-component load by first setting SAVELB with a LOCLA SAVELB and then loading INST with the right and top components. The master must also be set up in this manner, that is, the left and bottom components are set via a setpoint, and the right and top components with the box instruction.

EXAMPLE 12: 3D House

This example draws the frame of a house. The coordinate data for the example are given implicitly in Figure 8.3. Note how matrix transfer motions are concatenated.

```
LO    SP,=SAVE2
IOT    AC0,33
SETB  AC0,5
SETB   AC0,6         set MM3D
SET    AC0,          turn on Matrix Multiplier
IOT    AC0,32
LO    RP,=ARRAY1
ILOM   RC,4
LOMMA    0           load transformation matrix
LO    RP,=HOUSED     set RP to table of house data
ILOM   RC,5
TDR    POLY,AV       draw floor
ILOM   RC,5
TDR    POLY VV       draw ceiling
ILOM   RC,6
TDR    POL,VV        draw end wall
ILOM   RC,6
TDR    POL,VV        draw end wall
ILOM   RC,2
TDR    LINE,VV       draw roof
SD    SETV,WIN1      set for window 1
PUSHJ  WINDOW        jump to WINDOW subroutine (not included)
SD    SET,WIN2       set for window 2
PUSHJ  WINDOW        jump to WINDOW subroutine (not included)
PUSHMM   4           push transformation matrix to B
LO    RP,=DOORMT
ILOM   RC,4
LOMMP   4            multiply transformation matrices
LO    RP,=DOOR       set RP to door data
ILOM   RC,5
TDR    POLY,AA       draw door
ILOM   RC,4
POPMM   4            pop original transformation matrix
LO    RP,=DOORF      set RP to door frame data
ILOM   RC,4
SD    SETA,HOUSED    set point to corner of house
TDR    POLY,VV       draw door frame
```

FORTRAN SUPPORT ROUTINES

8.1 Function

The FORTRAN support routines provide the FORTRAN user the ability to define, manipulate, and display pictures with the LDS-2. The support routines are called by the FORTRAN program and prepare LDS-2 object code. Most of the calls do not place the code which has been generated directly into execution, but rather store the code in a user buffer. The generated routines can then be put into the LDS-2's execution string by DRAW calls. It is thus possible to execute the LDS-2 code in a user-specified order which may be different from the order in which the code was generated.

Most of the calls have the general form:

CALL SUB (NAME, LOC)

where:

SUB is the name of the particular support routine.

NAME is the identifier which will be associated with the code generated and should be either an integer or Holerith (up to four characters) value, and unique within the program.

LOC is the location of an array which usually contains both control information which is used to generate the code and the data which will be referenced by the LDS-2 code.

The information in the array referenced by LOC should be prepared by the FORTRAN program previous to the call. The data in these arrays are referenced directly by the LDS-2 code and may be changed dynamically, that is, they may be changed after the call has been made, or even while the code is in execution, but changing the control information will not change the code that has been generated, once the call has been made.

7.2 Data Formats

The arrays provided the support routines should contain integer values or names. This applies to both the control words and the data. The homogeneous element in three-dimensional data and the rotation elements for the Matrix Multiplier should be integer representations of fractions. That is, they should be integer values where the decimal point is assumed to be to the left of the word.

8.3 Preparation Calls

When the FORTRAN user is initiated on the LDS-2, default conditions for the state of the display system are set by the initializing routine. These conditions affect the modes of the LDS-2 pipeline devices, the dimension mode of the LDS-2, scope selection, and intensity control. The system is initiated with the LDS-2 in two-dimensional mode, the scope indicated on the job request record is selected, and maximum intensity is set. Default conditions for the parameter registers of the Clipping Divider are also provided as described in the calls which relate to these devices (see Section 8.4). All of the preparation calls generate code which goes directly into the user's execution string, unless the call is included within the scope of a GATH call (see Section 8.4).

Deleting the code generated by the preparation call via a KILL call, or turning this code off via an OFF call, does not restore any previous mode and, in fact, does not change any modes at all. Since the preparation calls set a state in the LDS-2, this state will remain until it is changed by another preparation call, or until another user is initiated. It is also important to realize that the dimension modes of the system affect the number of words of data which are processed per coordinate point and, thus, the data organization. A great deal of care must, therefore, be taken to insure that the prevailing mode corresponds to the data organization format of the data which are being processed.

CALL TWOD

TWOD sets the LDS-2 to two-dimensional operation. In 2D the LDS-2 picks up two words of data per coordinate point which are interpreted as X and Y. The LDS-2 is initially set to 2D, but if the mode has been changed by some other call, it is necessary to call TWOD in order to reset the mode to 2D.

CALL MM3D

MM3D sets the LDS-2 to a special three-dimensional mode. Three data words are required which specify the X, Y and Z coordinates of a point. The LDS-2 then supplies a "1" to give the fourth component expected by the Matrix Multiplier. The use of this mode allows the user to save storage and elminates the need to specify the fourth component of the homogeneous coordinates as long as that fourth component is a "1", which is often the case.

MM3D also turns the Matrix Multiplier on. The Matrix Multiplier is turned off at initialization, so MM3D must be called to turn it on. If one wishes to turn the Matrix Multiplier on, but does not wish to be in MM3D mode, it is simply necessary to call MM3D and then call either TWOD or HOMOG which changes the dimension mode of the LDS-2 but leaves the state of the Matrix Multiplier unchanged. MM3D also sets "depth cueing" (see INTSTY).

CALL CD3D

A second special three-dimension mode is called by CD3D. This mode is designed for data which are to be fed directly to the Clipping Divider. Again, three words are fetched per coordinate point, but in this case the fourth word supplied by the LDS-2 is a copy of the third word, thus giving X, Y, Z, Z, which is what the Clipping Divider expects. Since this mode is primarily of use when data are fed directly to the Clipping Divider, CD3D also turns the Matrix Multiplier off. When the user wants to turn the Matrix Multiplier off, but does not wish to be in CD3D mode, he can simply follow the CD3D call with a call to TWOD or HOMOG. CD3D also sets "depth cueing" (see INTSTY).

CALL HOMOG

HOMOG sets the LDS-2 to homogeneous coordinate mode where four words of data are expected for each point. Homogeneous coordinates are discussed in detail in Appendix III of the LDS-2 System Reference Manual. The four words of data are interpreted as X, Y, Z, and W, where W represents a fractional scale factor and is often "1". In working with homogeneous coordinates, it is important to realize that X, Y, and Z are interpreted as integer values, while W is interpreted as a fixed point fraction. Thus, the approximation for "1" which should be used in 37777777 (octal). HOMOG also sets "depth cueing" (see INTSTY).

CALL SELECT (Number of scopes, scope numbers)

SELECT allows the user to specify the scope(s) on which his picture is presented. The scope specified on the user job request record is initially selected so that it is only if the user wishes to change the scope(s) on which the picture is being presented that he must use the SELECT call. The scopes are numbered from 1 to n, where "n" is the number of available scopes.

CALL INTSTY (Intensity number)

This call allows the user to specify the intensity of the picture that is being drawn on the scope. An initial intensity value is set up for the user, but this value may be changed with the INTSTY call. Intensity values range from 0 (brightest) to 4096 (dimest). The intensity call also clears "depth cueing," which means that the intensity value rather than the Z coordinate of the point is used to control the intensity of the line. Depth cueing is restored by MM3D, CD3D, and HOMOG. It is thus possible to turn depth cueing on and off by careful ordering of the INTSTY and dimension calls.

CALL RFRATE (Cycles/second)

The highest priority user is allowed to specify the refresh rate through the use of this call. A default value of 30 cycles (1/30th of a second) is supplied, when the system is turned on so that RFRATE need be called only if some other refresh rate is desired. A RFRATE call by other than the highest priority user is ignored.

8.4 Definition and Manipulation Calls

The definition and manipulation calls are used to define pictures and to control the pipeline processing performed on these pictures. These calls generate code for the LDS-2, but do not put this code into the execution string of the user. The code is saved in the user's buffer until it is called by a DRAW call, which puts the code into execution. The order in which the code is executed is usually critical, but the order in which the code is generated by the calls is unimportant. It is thus possible to make the calls in any order that is convenient and then carefully control the execution of the code by using the appropriate DRAW calls.

Several calls can be grouped together as a single routine by the GATH call. GATH calls may be nested to allow the user to create tree-like structures of pictures and subpictures. Because the definition and manipulation calls do not go into immediate execution, and because all calls can be grouped into single routines which can be nested, the FORTRAN support routines provide the user great flexibility not only in defining and manipulating pictures, but also in structuring the display program generated by the support routines.

CALL DEF (NAME, LOC)

The DEF call is used to define drawings. The array referenced by LOC contains the coordinate data for the endpoints of the figure to be drawn and control information which determines how these points are to be connected. This array takes the following format:

<table>
<tr><td>Words/Point</td><td>No. of Sequences</td></tr>
<tr><td>Sequence</td><td>Mode</td></tr>
<tr><td colspan="2">Sequence Legnth--No. of Points</td></tr>
<tr><td colspan="2">X</td></tr>
<tr><td colspan="2">Y</td></tr>
<tr><td colspan="2">X or Y</td></tr>
<tr><td colspan="2">X,Y,Z, or W</td></tr>
<tr><td colspan="2">⋮</td></tr>
<tr><td>Sequence</td><td>Mode</td></tr>
<tr><td colspan="2">Sequence Legnth--No. of Points</td></tr>
<tr><td colspan="2">X</td></tr>
<tr><td colspan="2">Y</td></tr>
</table>

.

There may be either two, three, or four words per coordinate point depending upon the mode that the LDS-2 will be in when the code is put into execution. Since the different modes fetch different amounts of data per coordinate point, it is extremely important that the number of words per coordinate point corresponds to the mode that the LDS-2 is in at the time of execution. In constructing the first word of the array the number of sequences should be added to the code for the number of words per coordinate. This code is obtained by either of the following processes:

N=words per coordinate (2, 3, or 4)

NCODE=N*2**12

M=number of sequences

MWORD=NCODE+M

The decimal results of NCODE will be either 8192 (2 per coordinate) 12288 (3 per coordinate), or 16380 (4 per coordinate), and these numbers can be added directly to the number of sequences to build the word.

A "sequence" is one of the drawing sequences implemented by the LDS-2 (see Section 7.14 of the LDS-2 System Reference Manual). It should be noted that these sequence generally turn out to be different than their mnemonics imply if the count is only 1 or 2. For instance, a POLYGON sequence with a count of one is simply a setpoint, and with a count of 2 is simply

Drawing Sequences

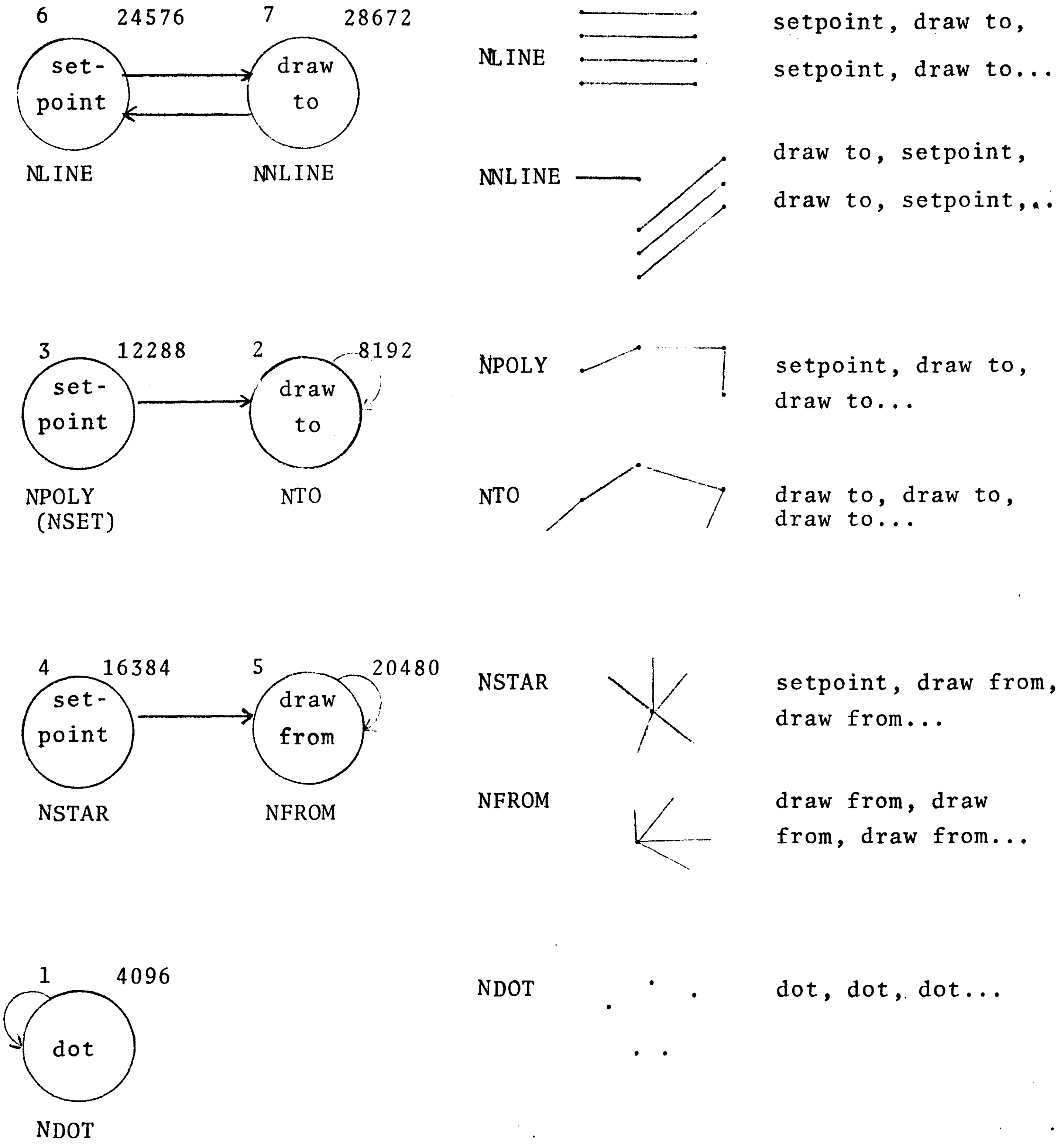
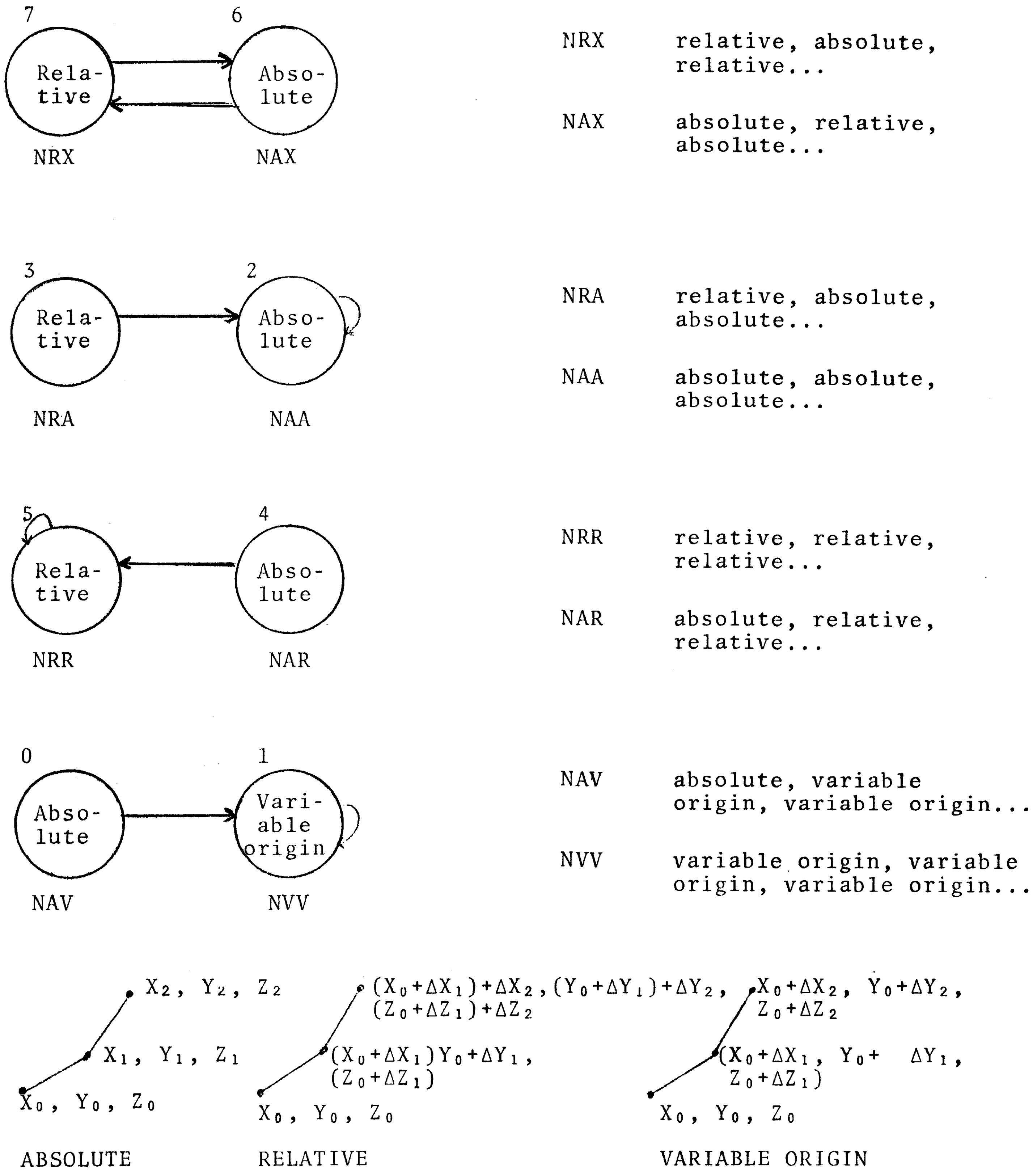


Figure 8.1

Data Modes


7
6
Relative
Absolute
NRX
NAX
3
2
Relative
Absolute
NRA
NAA
5
4
Relative
Absolute
NRR
NAR
0
1
Absolute
Variable origin
NAV
NVV
X_2, Y_2, Z_2
X_1, Y_1, Z_1
X_0, Y_0, Z_0
ABSOLUTE
$(X_0+\Delta X_1)+\Delta X_2, (Y_0+\Delta Y_1)+\Delta Y_2, (Z_0+\Delta Z_1)+\Delta Z_2$
$(X_0+\Delta X_1) Y_0+\Delta Y_1, (Z_0+\Delta Z_1)$
X_0, Y_0, Z_0
RELATIVE
$X_0+\Delta X_2, Y_0+\Delta Y_2, Z_0+\Delta Z_2$
$(X_0+\Delta X_1, Y_0+ \Delta Y_1, Z_0+\Delta Z_1)$
X_0, Y_0, Z_0
VARIABLE ORIGIN

NRX	relative, absolute, relative...
NAX	absolute, relative, absolute...
NRA	relative, absolute, absolute...
NAA	absolute, absolute, absolute...
NRR	relative, relative, relative...
NAR	absolute, relative, relative...
NAV	absolute, variable origin, variable origin...
NVV	variable origin, variable origin, variable origin...

Figure 8.2

a line. Figure 8.1 shows the sequences that are allowed, their octal code, and the decimal equivalents after shifting the codes to the left half of the word.

The allowable modes are also those which are implemented on the LDS-2. Figure 8.2 lists these modes, their octal codes and the decimal equivalents. In constructing the sequence/mode control word it is simply necessary to add the two decimal equivalents for the appropriate codes and store the result in the proper word of the array.

The third word in the array and the word after each sequence/mode word contains the number of coordinate points and not the number of data words in the sequence. The data words should contain integer values, as should all of the other words of the array.

The following two examples show the contents of the DEF arrays for the floor plan of a simple house and a three-dimensional drawing of the same house. The numbers shown in the array are decimal.

FORTRAN EXAMPLE 1: Two-dimensional House Plan

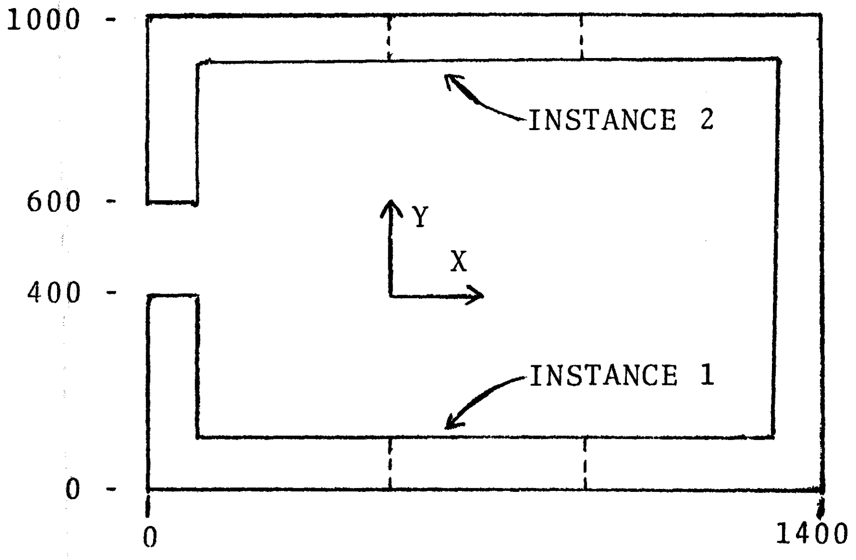
```
     NPLAN(1) = 2*2**12+1          2 words per point, 1 sequence
     NPLAN(2) = NPOLY+NAV          Polygon sequence, first point
                                   absolute, the rest are variable
                                   origin
     NPLAN(3) = 13
     DATA                          See Figure 8.3
       .
       .
       .
     CALL DEF (4HPLAN,NPLAN)
```

FORTRAN EXAMPLE 2: Three-dimensional House and Door Frame

```
     NHOUSE(1) = 3*2**12+6         2 Words per point, 6 sequences
     NHOUSE(2) = NPOLY+NAV         Floor
     NHOUSE(3) = 5
     DATA                          See Figure 8.4
       .
       .
       .
     NHOUSE(19) = NPOLY+NVV        Ceiling
     NHOUSE(20) = 5
     DATA
       .
       .
       .
     NHOUSE(36) = NPOLY+NVV        End wall
     NHOUSE(37) = 6
     DATA
       .
       .
       .
```

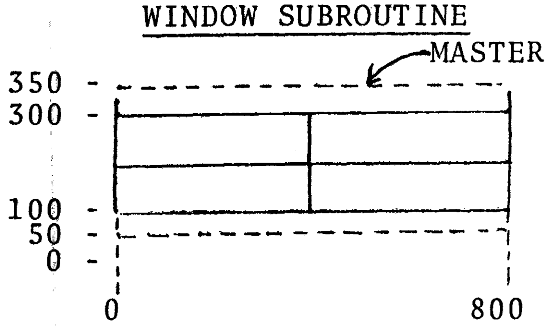
```
NHOUSE(56) = NPOLY+NVV          End wall
NHOUSE(57) 6
DATA
   .
   .
   .
NHOUSE(76) = NLINE+NVV          Roof
NHOUSE(77) = 2
DATA
   .
   .
   .
NHOUSE(96) = NPOLY+NVV          Door frame
NHOUSE(97) = 14
CALL DEF(4HHOUS,NHOUSE)
DATA
   .
   .
   .
NDOOR(1) = 3*2**12+1
NDOOR(2) = NPOLY+NVV
NDOOR(3) = 5
DATA
   .
   .
   .
CALL DEF (4HDOOR,NDOOR)
```

2D HOUSE PLAN



NPLAN		
0	0	(absolute setpoint)
0	400	
100	400	
100	100	
1300	100	
1300	900	
100	900	
100	600	
0	600	
0	1000	
1400	1000	
0	0	

NW1N	(WINDOW)
0	0
2000	1000



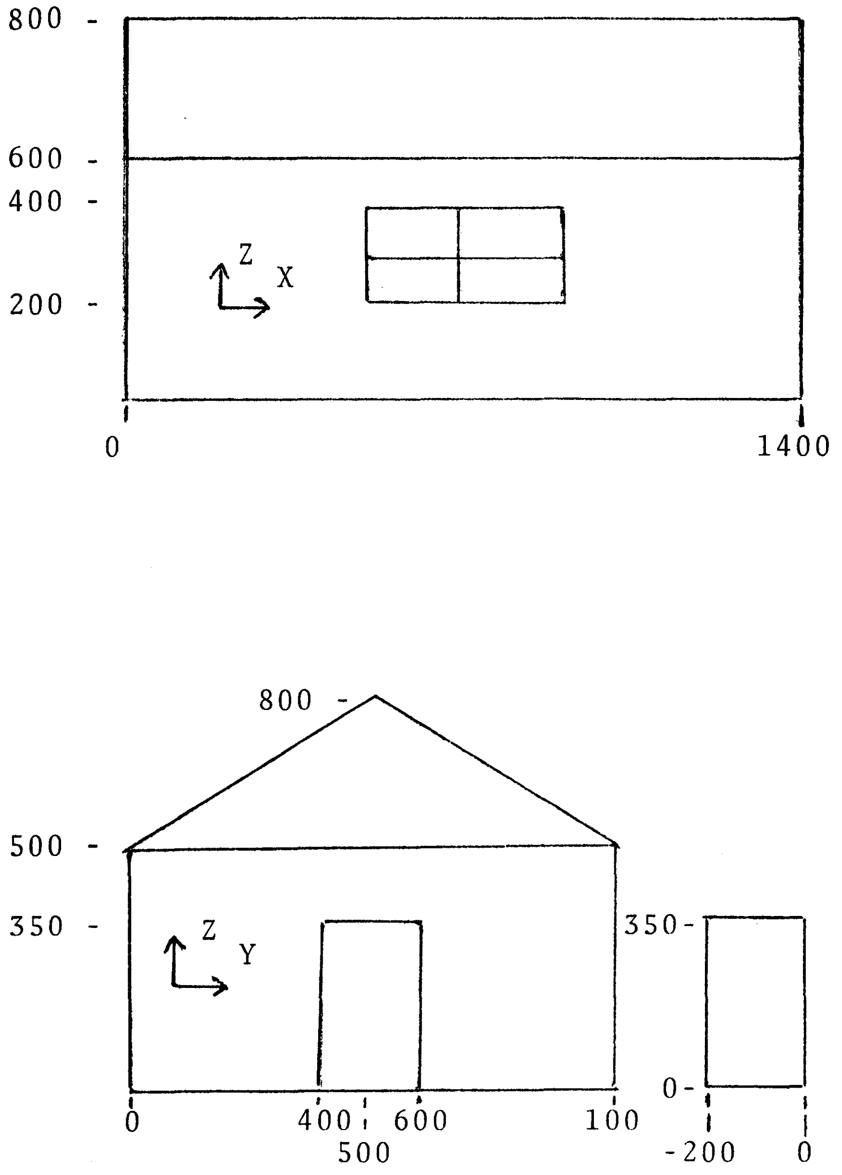
INSTANCE 1	
500	0
900	100

INSTANCE 2	
500	900
900	1000

MASTER	
0	50
800	350

Figure 8.3

3D HOUSE



	NHOUSE		
0	0	0	floor
0	1000	0	
1400	1000	0	
1400	0	0	
0	0	0	
0	0	500	ceiling
0	1000	500	
1400	1000	500	
1400	0	500	
0	0	500	
0	0	0	end 1
0	0	500	
0	500	800	
0	1000	500	
0	1000	0	
0	0	0	
1400	0	0	end 2
1400	0	500	
1400	500	800	
1400	1000	500	
1400	1000	0	
1400	0	0	
0	500	800	roof
1400	500	800	
	NDOORF		
0	400	0	
0	400	350	
0	600	350	
0	600	0	
	NDOOR		
0	0	0	
0	-200	0	
0	-200	350	
0	0	350	
0	0	0	

Figure 8.4

CALL WIND (NAME, LOC)

The WIND call generates the necessary code to change the WINDOW registers of the Clipping Divider. The four elements in the array are used to define the new window.

Left
Bottom
Right
Top

The coordinates define the left bottom and right top corners of the window rectangle and should be given in the same coordinate system as the drawing (i.e., the same coordinate system as is used for the DEF's that are processed while the window is in effect). The code generated will load the values in the arrays into the WINDOW registers when the code goes into execution. Those values will stay in the WINDOW registers, until either the mode is changed from 2D or another WIND call is executed. Thus, deleting the code which set the window via a KILL call does not change the window. When the LDS-2 is in 2D, each picture element is compared with the window, and only those portions of the picture which lie within the window are displayed.

A window is defined as the system is initialized; this window stretches from -32757 to +32767 in both the X and the Y directions. Unless another window is defined, this window is in effect.

The following example sets up a window around the floor plan described in the description of the DEF call. Section 4.4 of the LDS-2 System Reference Manual should be consulted for further information on the use of the window.

FORTRAN EXAMPLE 3: Window

```
     NWIN(1) =    0
     NWIN(2) =    0
     NWIN(3) = 2000
     NWIN(4) = 1000

     CALL WIND (3HWIN,NWIN)
```

CALL VIEW (NAME, LOC)

The VIEW call is used to set the VIEWPORT registers of the Clipping Divider which define the portion of the scope face onto which the picture is to be mapped. The viewport is defined in the same manner as the window, that is, by giving its left bottom and right top corners in an array.

Left
Bottom
Right
Top

In contrast to the WINDOW coordinates, however, the viewport coordinates are specified in the coordinate system of the scope. Thus, the values used should range between -32767 and +32767. A viewport is necessary regardless of the mode of the LDS-2. Anything which lies within the field of vision (which is a pyramid of vision defined by the planes x=+Z and Y=+Z in the threedimensional modes and the window in two dimensions) is mapped onto the viewport. If the viewport is not the same shape as the field of view (i.e., not square in 3D or not the same shape as the window in 2D), the picture will be stretched in either the X or Y direction. The viewport can cover the whole scope face or any rectangular portion of the scope face. A viewport is defined as the system is initialized for each user to cover the whole scope (i.e., -32767, -32767, +32767, +32767).

FORTRAN EXAMPLE 4: Viewport

The following viewport covers the upper half of the screen:

```
NVIEW(1) = -32767
NVIEW(2) = 0
NVIEW(3) = 32767
NVIEW(4) = 32767

CALL VIEW (4HVIEW,NVIEW)
```

CALL BOX (NAME, LOC), CALL COPY (NAME, LOC)

The BOX and the COPY routines are used to draw repeated copies of two-dimensional subpictures. These routines allow the user to define a subpicture which can then be placed at several positions on the main picture and even appear in different sizes. The BOX call is used to set up the basic subpicture parameters and should be called by a COPY call each time the subpicture is to appear. The BOX call takes the following format:

Name
Left
Bottom
Right
Top

The first element in the array is the name of the picture elements that are to appear in the subroutine. This should either be the name of a DEF call or the name of a GATH call which contains the definition of the subpicture. The four data words define the left bottom and right top corners of a "master" rectangle. This master rectangle serves the same function for the subpicture that the window does for the main drawing. Any part of the subpicture which is outside the master is not included in the copies of the subpicture, and the size of the master affects the size of the subpicture.

The COPY call generates the code to place a copy of the subpicture onto the main drawing. The array referenced by LOC takes the following form:

Name
Left
Bottom
Right
Top

The first word of the array should contain the name of the BOX call which was used to define the subpicture. The data words define the left bottom and right top corners of the "instance" rectangle. Everything that is within the master rectangle defined in the BOX call is mapped onto the instance rectangle. If the instance rectangle lies partially outside the current window, only those portions of the subpicture which lie within the portion of the instance that is within the window will be displayed. If the instance lies wholly outside the current window, the code which defines the subpicture is not processed at all, since nothing would appear on the scope anyway. This fact can be used to define very large data bases, where only a small portion is ever displayed at one time. By defining each portion of the drawing with a BOX call and drawing that code with COPY calls, large sections of code and data which lie entirely outside the window can be skipped entirely, thus improving the performance of the system.

Since the boxing process calculates new window and viewport values, the old values in the WINDOW and VIEWPORT registers are saved when the COPY call is executed (by a DRAW call) and then restored. The code to save and restore these registers is actually generated by the BOX call, but since the BOX call cannot be put into execution except through a COPY call to the BOX, it is convenient to think of this as happening when the COPY call is executed.

It is important that a BOX call be called only by COPY calls (and not by DRAW calls), and that the name in the COPY call array be the name of a BOX call. It is also important that these calls are executed with the LDS-2 in 2D mode.

FORTRAN EXAMPLE 5: BOX and COPY

The following calls can be used to place symbols for two windows on the floor plan of Example 1 (see Figure 8.3).

```
NW(1) = 2*2**12+2              Definition of window
NW(2) = NPOLY+NAA              Symbol
NW(3) = 5
DATA
  .
  .
  .
NW(14) = NLINE+NAA
NW(3) = 4
DATA
  .
  .
  .
CALL DEF (1HW,NW)
NBX(1) = W
NBX(2) = 0                     Left of Master
NBX(3) = 50                    Bottom of master
NBX(4) = 800                   Right of master
NBX(5) = 250                   Top of master
```

```
CALL BOX(2HBX,NBX)
NCPY1(1) = BX
NCPY1(2) = 500                  Left of instance 1
NCPY1(3) = 0                    Bottom of instance 1
NCPY1(4) = 900                  Right of instance 1
NCPY1(5) = 100                  Top of instance 1
CALL COPY(4HCPY1,NCPY1)
NCPY2(1) = BX
NCPY2(2) = 500                  Left of instance 2
NCPY2(3) = 900                  Bottom of instance 2
NCPY3(4) = 900                  Right of instance 2
NCPY2(5) = 1000                 Top of instance 2
CALL COPY(4HCPY2,NCPY1)
```

CALL MM (NAME, LOC)

The MM call is used to manipulate the values in the registers of the Matrix Multiplier. When the Matrix Multiplier is on, data that are sent down the pipeline are multiplied by the first of the four matrices which can be stored in the Matrix Multiplier. The MM call allows the user to set these matrices to the appropriate values. The array referenced by the LOC· should take the following form:

Matrix	Action
Element 1	
Element 2	
Element 3	
⋮	
Element 16	

The legal values for "Matrix" are 1, 2, 3, and 4 for the four matrices. The following actions may be indicated in the "Action" field of the control word.

1. Load the Matrix Multiplier matrix specified in the array with the data in the array. This data should contain the elements of the 4 X 4 matrix desired. Figure 3.2 shows how these words are stored into the matrix and which of the data elements should be considered as integers and which as fractions.

2. Store the values in the matrix specified into the array.

3. Multiply the matrix specified in the array by the data in the array and leave the result in Matrix 1. Since Matrix 1 is used to contain the result, it cannot be specified as the multiplicand.

4. Push the data in Matrix 1 into the matrix specified in the array. This destroys the old value of this matrix.

5. Pop the value from the specified matrix back into Matrix 1.

Since the "push" and "pop" actions do not require data, the 16 array words for the elements of the matrix are ignored and do not need to be provided. Since these instructions require no memory references when they execute, the code generated is much faster than the code generated by the "load" and "store" actions. If only a limited amount of temporary matrix storage is required, it is best to use matrices 2, 3, and 4 for storage and use the push and pop actions to store into and retrieve from temporary storage.

It is also possible to use the Matrix Multiplier in 2D operation. In this case only the first two elements of each row are used in the matrix transformation. Thus, the data in elements 3 and 4, 6 and 7, and 10 and 11 are not used and should be set to 0. The last four elements 13-16 are also unused in 2D operations.

The following examples show how matrix transformations can be used to show the desired view of the house defined in the DEF example, and how matrix transformations can be concatenated to show a door which opens and closes in the proper position.

FORTRAN EXAMPLE 6: Using the Matrix Multiplier

These calls load the Matrix Multiplier with a rotation and translation matrix for the house; multiply that matrix by another to calculate the transformation matrix for the door of the house, and then return the first matrix.

```
NROTMT(1) = 1*2**12+1
DATA FOR NROTMT
   .
   .
   .
CALL MM(4HROMT,NROMT)           Rotation and translation matrix
                                for house
   .
   .
   .
NPSH = 2*2**12+4                Push Matrix 1 into Matrix 2
                                to SAVE
CALL MM(3HPSH,NPSH)
NDOORM(1) = 2*1**12+3
DATA FOR NDOORM
   .
   .
   .
CALL MM(3HDRM,NDOORM)           Calculate new matrix for
                                door
   .
   .
   .
```

```
NPOP(1) = 2*2**12+5                Restore original rotation
                                   and translation matrix
CALL MM(3HPOP,NPOP)
```

CALL TEXT (NAME, LOC)

The TEXT routine is used to display characters on the screen. Previous to this call, the beam should be set to the position of the first character. The TEXT array should have the following format:

<table>
<tr><td colspan="4">Size</td></tr>
<tr><td colspan="4">No. of Words</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

"Size" specifies the size of the characters in page coordinates, that is, the size of the characters in relation to the rest of the drawing. The window to viewport mapping then determines the size of the characters on the screen. Thus, if the window is defined as -1000 to +1000, and the user wants 50 characters per line (i.e., across the whole face of the scope), then the size should be 2000/50 = 40.

The TEXT routine calls the software character generator, when it is put into execution by a DRAW call.

CALL GATH (NAME), CALL NOG (NAME)

The GATH routine is used to gather all the code generated by all calls which occur between the GATH and its corresponding NOG into a single routine. Thus, when the GATH call is put into execution (by a DRAW call which references it), all the code that has been generated by calls within the scope of the GATH will also be put into execution. Because GATH puts all the code within its range into execution, BOX calls and the DEF calls they reference should not be included within the range of a GATH, or they will be executed directly rather than through the COPY call. If, however, the calls within the GATH have name parameters, they may still be referenced individually. Calls which normally generate code that goes directly into execution (i.e., the drawing and preparation calls) may also be executed directly, but stored in the user's buffer until the GATH routine is executed, or until they themselves are referenced by another DRAW call. GATH calls may be nested to 20 levels. It is possible to nest GATH calls in two ways. They can be nested from the top by including a GATH call within the scope of another GATH call, or they may be nested from the bottom by first defining the lowest level GATH call and then referencing that call by a DRAW which is included in a higher level GATH.

NOG closes the GATH routine. If the name on NOG is the name of a higher level GATH call, then all the GATH calls nested below that level, as well as the GATH named, will be closed.

FORTRAN EXAMPLE 7: Using the GATH Call

The DEF and MM calls for the 30 house can easily be included in a GATH call, so that the whole sequence of code can be referenced by referencing the GATH call.

```
     CALL GATH (1H)
     CALL DEF (4HHOUS,NHOUSE)
     CALL MM (3HPSH,NPSH)
     CALL MM (3HDRM,NDOORM)
     CALL DEF (4HDOOR,NDOOR)
     CALL MM (3HPOP,NPOP)
     CALL NOG (1HH)
```

CALL REPEAT (NAME, LOC)

This call generates an LDS-2 subroutine which, when referenced by a DRAW call, will execute each of the named subroutines in the array the designated number of times. The named subroutines are linked one after the other, until all subroutines have been placed in the chain. The chain of calls to named subroutines will then be executed the designated number of times, when referenced by a DRAW call. Each name in the array must be that of a previously generated LDS-2 subroutine. BOX calls and the DEF's they reference should not be included in the Repeat Table. COPY calls may, however, be included.

Repeat Count	No. of Names
Name 1	
Name 2	
Name 3	
• • •	

CALL LDS (NAME, LOC)

The LDS call allows the FORTRAN user to escape into machine language in order to perform functions that are not provided by other FORTRAN calls. LOC points to an array which should contain valid LDS-2 instructions which have already been assembled. The last instruction must be a POPJ to return back to the FORTRAN program. No checking is done to see that the code in the array is legal, so this is a "use at your own risk" call.

8.5 The Drawing Calls

The drawing calls allow the user to control the execution of the code generated by the definition and manipulation calls. These calls cause the code generated by other calls to be added to the execution string of the user, deleted from the execution string, or destroyed entirely.

CALL DRAW (NAME, LOC)

The DRAW call is used to put code generated by the definition and manipulation calls or within a GATH call into execution. If the DRAW call is itself within a GATH, the code does not go into execution until the GATH is referenced by another DRAW call, or until the DRAW call itself is referenced by another DRAW call. The array for the DRAW call should include the names of the routines to be executed in the order in which the user wishes them to be executed.

Number of Names
Name 1
Name 2
Name 3
. . .

The names in the array should be names assigned to calls which have previously been made by the user's program.

CALL OFF (LOC), CALL ON (LOC)

The OFF call is used to remove code generated by the support routines from the execution string. The array referenced by LOC contains the names of the routines deleted.

Number of Names
Name 1
Name 2
Name 3
. . .

It is assumed that the routines named in the array are currently in the execution string. If they are not, there is no need to reference them in the array and an error message will be given. Even though the code is removed from the execution string, its place in the execution string is maintained, so that by using an ON call the code will be returned to its original place in the execution string. The ON call array lists the names of the routines to be turned back on.

Number of Names
Name 1
Name 2
Name 3
. . .

The names in the array need not be in the same order as they were in the OFF array, but it is not legal to include any names in this array which were not included in a previously executed OFF array. An error message will be given, if this is done. By strategic use of OFF and ON calls, it is possible to "blink" all or parts of the picture.

CALL KILL (LOC)

The KILL call is used to destroy the routines that were generated by the calls named in the array.

Number of Names
Name 1
Name 2
Name 3
. . .

If the named code is in execution, it will be removed from the execution string and destroyed. If it is not in execution, it will simply be destroyed. In either case, no further references may be made to the code. If a DRAW call is named in the array, the DRAW routine will be destroyed and the routines referenced by the DRAW will be removed from the execution string. However, the routines referenced by the DRAW call are not destroyed and may be referenced by later calls. If the LOC parameter contains an "0," all of the user's code will be removed from execution, but may be referenced later. All of the user's code is destroyed automatically at the termination of his FORTRAN program.

Remove this section

SOFTWARE INTERFACE

9.1 General

The software interface provided for the LDS-2 and the SEL-840 schedules users on the system and handles the interrupts that occur. Within the framework of the Interrupt Handlers, such services as setting the real time clock, handling I/O service requests, and interpreting and displaying characters are performed. Figure 9.1 shows the basic structure of the software interface.

9.2 The Schedulers

When the user enters a job, the SEL-840 Scheduler builds an entry in the Schedule Table and checks to see if the LDS-2 is in stop state (sleep). If the LDS-2 is stopped, the SEL-840 Scheduler issues an interrupt to the LDS-2, which initiates the LDS-2 Scheduler. The LDS-2 Scheduler then interrogates the Schedule Table and starts up the user's program. If the LDS-2 is already running, the SEL-840 Scheduler simply adds the user to the Schedule Table. After each user has finished, the LDS-2 traps to the LDS-2 Scheduler, which removes the finished user and checks the Schedule Table to find the next user.

9.3 Interrupt Handlers

When the LDS-2 is interrupted, it traps to the LDS-2 Interrupt Handler, which determines the cause of the interrupt and takes the appropriate action. If the cause is an error condition, the LDS-2 Interrupt Handler sets a status word and interrupts the SEL-840. The SEL-840 Interrupt Handler then interrogates this status word and takes the appropriate action; which, in this case, is to terminate the job, print an error message, and the value of the LDS-2 PROGRAM COUNTER (PC) at the time the interrupt was caused. The following error conditions will cause job termination:

Non-existent Instruction
Non-existent I/O Device (i.e., illegal IOT)
Parity Error
Scope Selection Villation
Memory Protection Violation
Non-existent Memory

The LDS-2 Interrupt Handler also handles interrupts from the real time clocks and calls the LDS-2 Scheduler to restart the first user when the end of a refresh cycle comes. The LDS-2 always refreshes at a constant rate when under executive control. If all the users are done, the LDS-2 goes into a waiting loop until the clock interrupt terminating that refresh cycle comes.

SOFTWARE INTERFACE

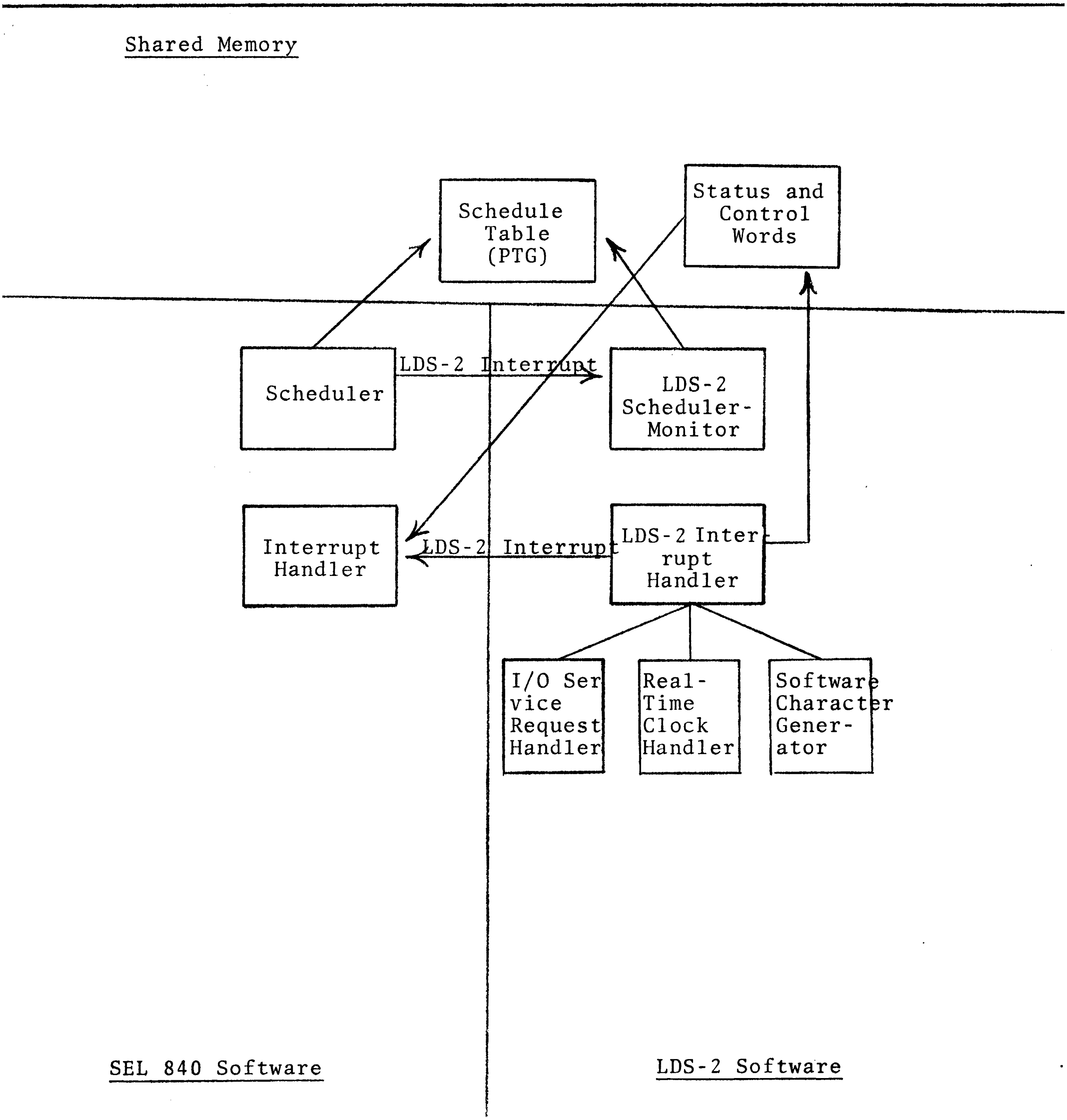


Figure 9.1

As explained in Sections 2.5 and 7.12, certain "illegal" IOT instructions are used for communicating special requests to the LDS-2 executive routines. These instructions cause an interrupt which is interpreted by the LDS-2 Interrupt Handler.

IOR. The IOR mnemonic is an IOT ,372, which is interpreted as a request for input-output service. The Interrupt Handler takes the contents of AC0 as the address of a user-prepared I/O packet, which should have been prepared according to SEL-840 I/O packet specifications. The Interrupt Handler will interrupt the SEL-840 to perform the requested I/O and then return to one of three locations.

CALL+1	if there is an error in the packet
CALL+2	if the user requested completion status
CALL+3	for normal completion (does not mean that the I/O itself has been completed)

CHAR. The CHAR mnemonic is an IOT ,373, which is interpreted as a call to the software character generator. The Interrupt Handler expects AC0 to contain the address of the first word of the text array. The text array should be formatted as shown in Section 8.7.

RSTART. RSTART (IOT ,370) should be the next to the last statement in an Assembly Language program, if it is not to be repeatedly executed (e.g., to refresh a picture). This IOT is interpreted by the Interrupt Handler as indicating the end of the user's execution string. The user's program is restarted at the beginning, when his turn comes up again.

STOP. STOP is an IOT ,371 and is taken as an indication that the user's program is done and causes it to be terminated.

CLKSTP. Real-time clock interrupts may be stopped with an IOT ,374 or CLKSTP. This instruction is ignored, unless the user has highest priority. Once CLKSTP has been issued, the executive is circumvented until some other interrupt comes, so the user must jump to the beginning of his display program to refresh the picture. All other users will be locked out when the highest priority user turns off the clock.

CLKSRT. An IOT ,375 is used to restart the clock. Again, only the highest priority user may use this instruction.

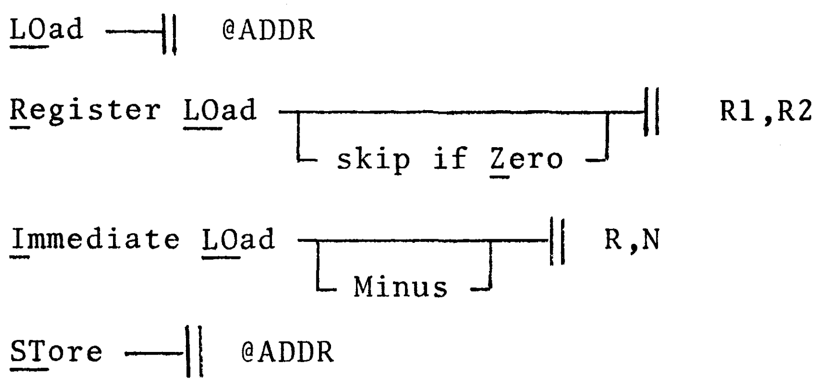
APPENDIX I

LDS-2 Mnemonic Construction

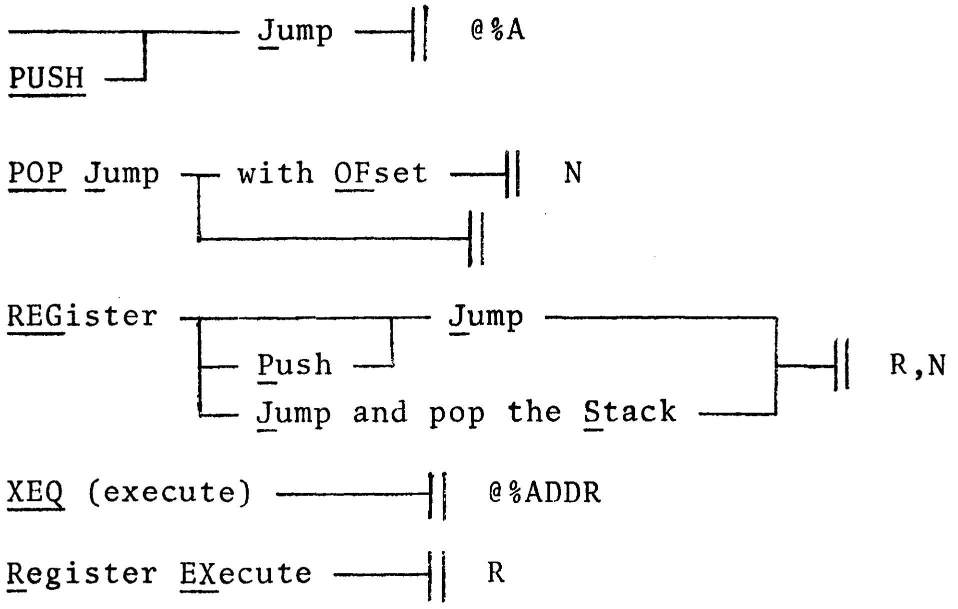
The figures on the following pages show how LDS-2 Assembly Language mnemonics are constructed. Mnemonics are built by following a path from left to right and concatenating the underlined (and capitalized) parts of the words encountered. For example, under Stack Control the first set of mnemonics expands to PUSH, POP, IPUSH, IPOP, DPUSH, and DPOP. The arguments for each set of instructions are given after the parallel vertical lines at the end of the string.

LDS-2 MNEMONIC CONSTRUCTION

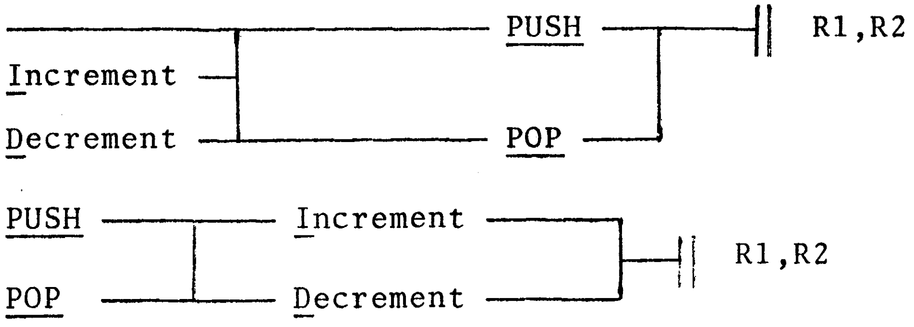
Load and Store Channel Control Registers



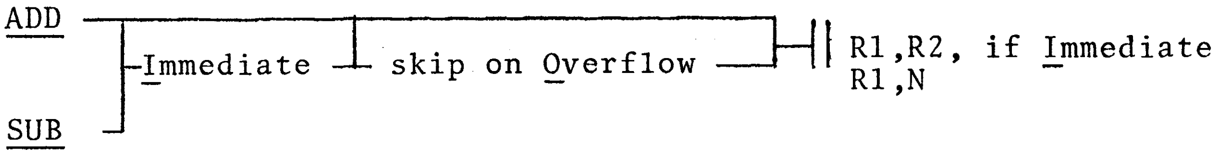
Program Control

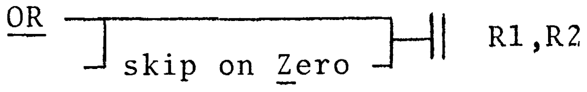


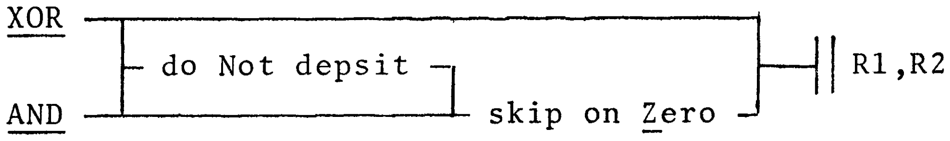
Stack Control



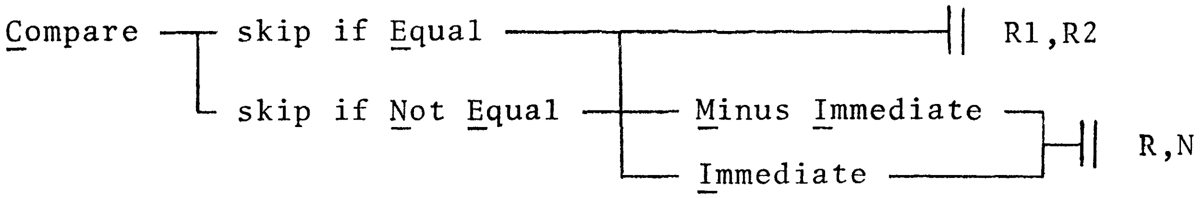
Arithmetic Operations



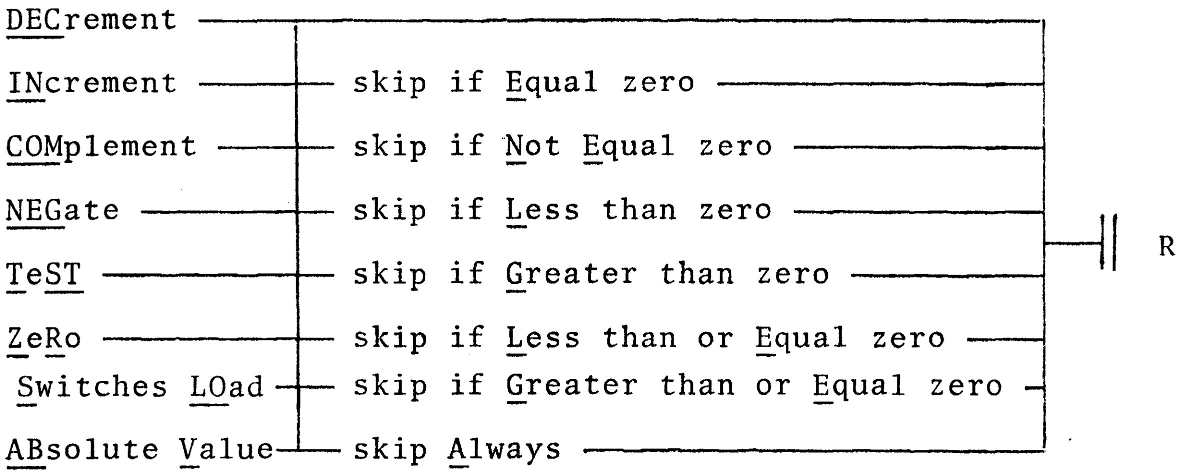




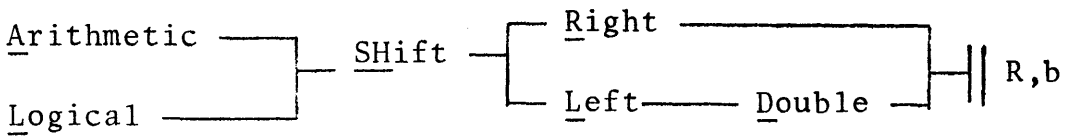
Compare

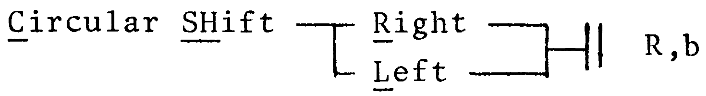


Unary

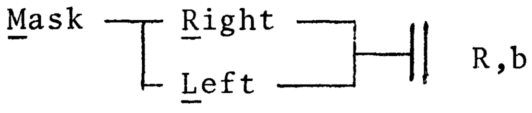


Shift Instructions

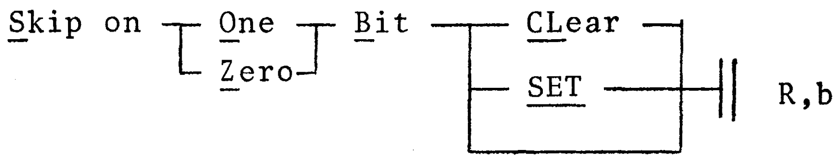


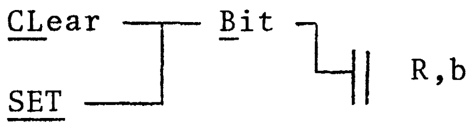


Masking Instructions

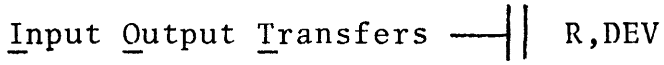


Bit Manipulation



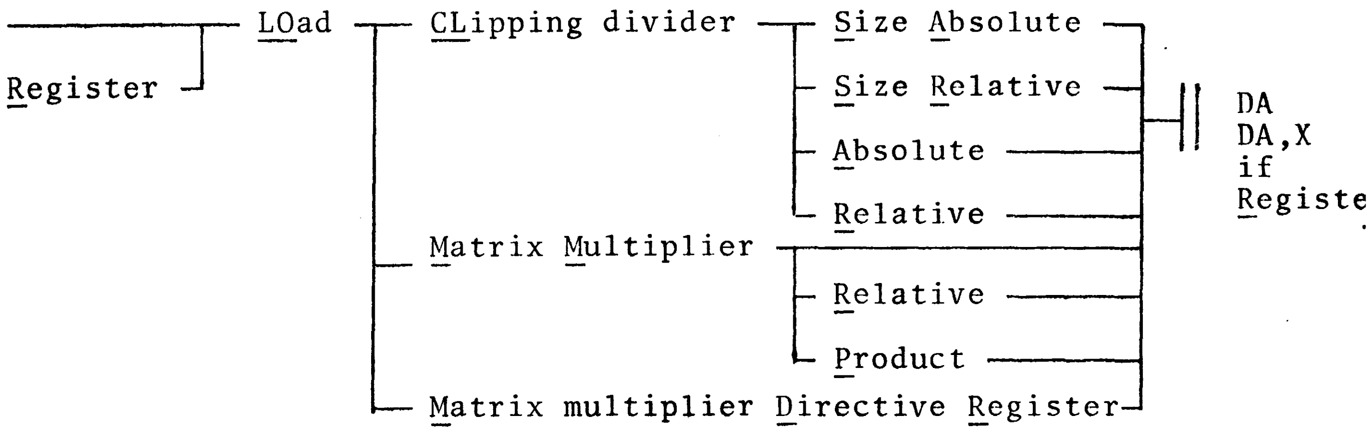


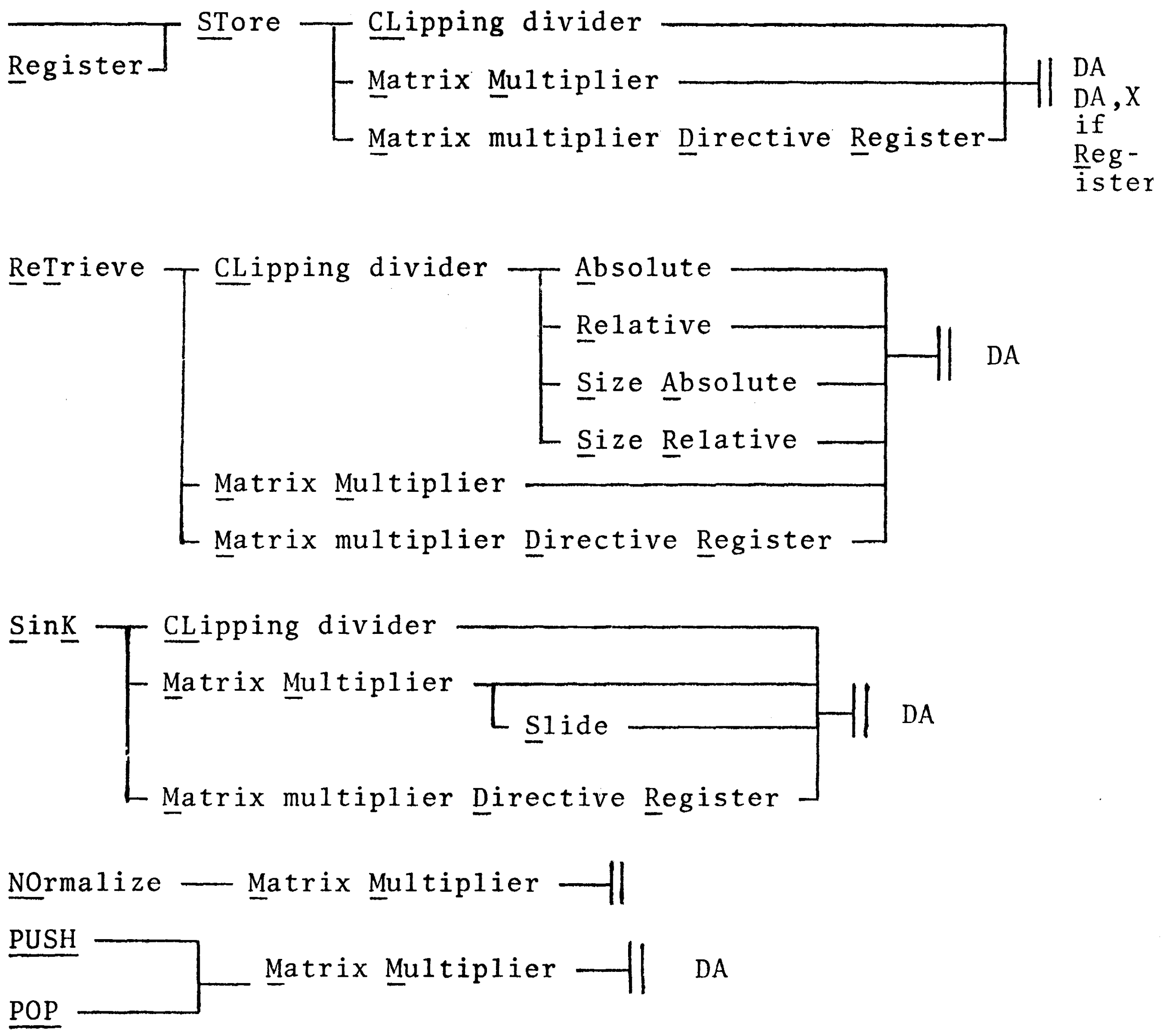
Input/Output Transfers



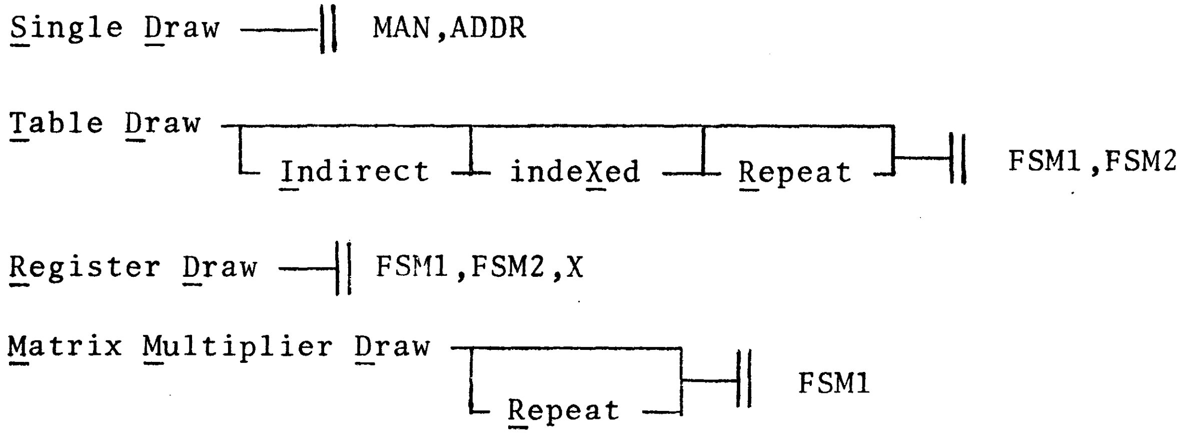
SLEEP

Pipeline Load/Unload Instructions





Drawing Instructions



APPENDIX II

OPDEF's and EQU's

The following list gives the permanently defined OPDEF's and EQU's for the LDS-2 Assembler. These OPDEF's and EQU's are reserved mnemonics, which may not be used for other purposes. An attempt to use one of the permanently defined mnemonics will result in an error message from the Assembler. Chapter 6 of this manual explains the format and meaning of both the OPDEF and the EQU directives.

```
←        EQU      24                     (THIS MUST BE SET TO THE WORD SIZE)
←←       EQU      6                      (SET THIS TO NUMBER OF BITS/TEXT CHAR.
↑        EQU      256                    (SET THIS SO THAT ↑=(2**(←-16))  )
!        EQU      ↑*16384/5 MULTIPLIER FOR 1ST DIGIT OF DEC, FRACT.
!!       EQU      !/1Ø                             "          2ND        "
!!!      EQU      !!/1Ø                            "          3RD        "
!!!!     EQU      !!!/1Ø                           "          4TH        "
!!!!!    EQU      !!!!/1Ø                          "          5TH        "
!!!!!!   EQU      !!!!!/1Ø           "         6TH        "          "
ACØ      EQU      Ø
AC1      EQU      1
AC2      EQU      2
AC3      EQU      3
TOS      EQU      4
SP       EQU      5
DSP      EQU      6
IR       EQU      7
X        EQU      Ø1Ø
Y        EQU      Ø11
Z        EQU      Ø12
W        EQU      Ø13
RP       EQU      Ø14
RC       EQU      Ø15
WP       EQU      Ø16
WC       EQU      Ø17
LO       OPDEF    (Ø),(4,7,N),(←-8,←-1,A@)
ST       OPDEF    (Ø1ØØØØ*↑),(4,7,N),(←-8,←-1,A@)
RLO      OPDEF    (Ø6ØØØØ*↑+2),(4,7,N),(4,←-5,N)
RLOZ     OPDEF    (Ø16ØØØØ*↑+2),(4,7,N),(4,←-5,N)
ILO      OPDEF    (Ø6ØØØØ*↑+Ø12),(4,7,N),(←-12,←-5,N)
ILOZ     OPDEF    (Ø16ØØØØ*↑+Ø12),(4,7,N),(←-12,←-5,N)
ILOM     OPDEF    (Ø6ØØØØ*↑+Ø13),(4,7,N),(←-12,←-5,N)
ILOMZ    OPDEF    (Ø16ØØØØ*↑+Ø13),(4,7,N),(←-12,←-5,N)
J        OPDEF    (Ø214ØØ*↑),(←-8,←-1,A@%)
PUSHJ    OPDEF    (Ø234ØØ*↑),(←-8,←-1,A@%)
XEQ      OPDEF    (Ø254ØØ*↑),(←-8,←-1,A@%)
SOB      OPDEF    (Ø5ØØØØ*↑+Ø1Ø),(4,7,N),(4+←/17,←-5,N)
SZB      OPDEF    (Ø5ØØØØ*↑+Ø11),(4,7,N),(4+←/17,←-5,N)
CLB      OPDEF    (Ø5ØØØØ*↑+Ø12),(4,7,N),(4+←/17,←-5,N)
SETB     OPDEF    (Ø5ØØØØ*↑+Ø13),(4,7,N),(4+←/17,←-5,N)
SOBCL    OPDEF    (Ø5ØØØØ*↑+Ø14),(4,7,N),(4+←/17,←-5,N)
SZBCL    OPDEF    (Ø5ØØØØ*↑+Ø15),(4,7,N),(4+←/17,←-5,N)
SOBSET   OPDEF    (Ø5ØØØØ*↑+Ø16),(4,7,N),(4+←/17,←-5,N)
SZBSET   OPDEF    (Ø5ØØØØ*↑+Ø17),(4,7,N),(4+←/17,←-5,N)
ASHR     OPDEF    (Ø5ØØØØ*↑+Ø4),(4,7,N),(4+←/17,←-5,N)
LSHR     OPDEF    (Ø5ØØØØ*↑+Ø2),(4,7,N),(4+←/17,←-5,N)
LSHL     OPDEF    (Ø5ØØØØ*↑+Ø3),(4,7,N),(4+←/17,←-5,N)
ASHL     OPDEF    LSHL
CSHR     OPDEF    (Ø5ØØØØ*↑),(4,7,N),(4+←/17,←-5,N)
CSHL     OPDEF    (Ø5ØØØØ*↑+Ø1),(4,7,N),(4+←/17,←-5,N)
ASHRD    OPDEF    (Ø5ØØØØ*↑+Ø5),(4,7,N),(4+←/17,←-5,N)
LSHRD    OPDEF    (Ø5ØØØØ*↑+Ø6),(4,7,N),(4+←/17,←-5,N)
LSHLD    OPDEF    (Ø5ØØØØ*↑+Ø7),(4,7,N),(4+←/17,←-5,N)
ASHLD    OPDEF    LSHLD
MR       OPDEF    (Ø15ØØØØ*↑+Ø1Ø),(4,7,N),(4+←/17,←-5,N)
ML       OPDEF    (Ø15ØØØØ*↑+Ø11),(4,7,N),(4+←/17,←-5,N)
```

```
DEC         OPDEF       (Ø15ØØØØ*↑),(4,7,N)
DECE        OPDEF       (Ø15ØØØØ*↑+Ø2Ø),(4,7,N)
DECL        OPDEF       (Ø15ØØØØ*↑+Ø1ØØ),(4,7,N)
DECLE       OPDEF       (Ø15ØØØØ*↑+Ø12Ø),(4,7,N)
DECG        OPDEF       (Ø15ØØØØ*↑+Ø4Ø),(4,7,N)
DECGE       OPDEF       (Ø15ØØØØ*↑+Ø6Ø),(4,7,N)
DECNE       OPDEF       (Ø15ØØØØ*↑+Ø14Ø),(4,7,N)
DECA        OPDEF       (Ø15ØØØØ*↑+Ø16Ø),(4,7,N)
INC         OPDEF       (Ø15ØØØØ*↑+Ø1),(4,7,N)
INCE        OPDEF       (Ø15ØØØØ*↑+Ø21),(4,7,N)
INCL        OPDEF       (Ø15ØØØØ*↑+Ø1Ø1),(4,7,N)
INCLE       OPDEF       (Ø15ØØØØ*↑+Ø121),(4,7,N)
INCG        OPDEF       (Ø15ØØØØ*↑+Ø41),(4,7,N)
INCGE       OPDEF       (Ø15ØØØØ*↑+Ø61),(4,7,N)
INCNE       OPDEF       (Ø15ØØØØ*↑+Ø141),(4,7,N)
INCA        OPDEF       (Ø15ØØØØ*↑+Ø161),(4,7,N)
COM         OPDEF       (Ø15ØØØØ*↑+Ø2),(4,7,N)
COME        OPDEF       (Ø15ØØØØ*↑+Ø22),(4,7,N)
COML        OPDEF       (Ø15ØØØØ*↑+Ø1Ø2),(4,7,N)
COMLE       OPDEF       (Ø15ØØØØ*↑+Ø122),(4,7,N)
COMG        OPDEF       (Ø15ØØØØ*↑+Ø42),(4,7,N)
COMGE       OPDEF       (Ø15ØØØØ*↑+Ø62),(4,7,N)
COMNE       OPDEF       (Ø15ØØØØ*↑+Ø142),(4,7,N)
COMA        OPDEF       (Ø15ØØØØ*↑+Ø162),(4,7,N)
NEG         OPDEF       (Ø15ØØØØ*↑+Ø3),(4,7,N)
NEGE        OPDEF       (Ø15ØØØØ*↑+Ø23),(4,7,N)
NEGL        OPDEF       (Ø15ØØØØ*↑+Ø1Ø3),(4,7,N)
NEGLE       OPDEF       (Ø15ØØØØ*↑+Ø123),(4,7,N)
NEGG        OPDEF       (Ø15ØØØØ*↑+Ø43),(4,7,N)
NEGGE       OPDEF       (Ø15ØØØØ*↑+Ø63),(4,7,N)
NEGNE       OPDEF       (Ø15ØØØØ*↑+Ø143),(4,7,N)
NEGA        OPDEF       (Ø15ØØØØ*↑+Ø163),(4,7,N)
TST         OPDEF       (Ø15ØØØØ*↑+Ø4),(4,7,N)
NOP         OPDEF       [TST        Ø]
TSTE        OPDEF       (Ø15ØØØØ*↑+Ø24),(4,7,N)
TSTL        OPDEF       (Ø15ØØØØ*↑+Ø1Ø4),(4,7,N)
TSTLE       OPDEF       (Ø15ØØØØ*↑+Ø124),(4,7,N)
TSTG        OPDEF       (Ø15ØØØØ*↑+Ø44),(4,7,N)
TSTGE       OPDEF       (Ø15ØØØØ*↑+Ø64),(4,7,N)
TSTNE       OPDEF       (Ø15ØØØØ*↑+Ø144),(4,7,N)
TSTA        OPDEF       (Ø15ØØØØ*↑+Ø164),(4,7,N)
ZR          OPDEF       (Ø15ØØØØ*↑+Ø5),(4,7,N)
ZRE         OPDEF       (Ø15ØØØØ*↑+Ø25),(4,7,N)
ZRL         OPDEF       (Ø15ØØØØ*↑+Ø1Ø5),(4,7,N)
ZRLE        OPDEF       (Ø15ØØØØ*↑+Ø125),(4,7,N)
ZRG         OPDEF       (Ø15ØØØØ*↑+Ø45),(4,7,N)
ZRGE        OPDEF       (Ø15ØØØØ*↑+Ø65),(4,7,N)
ZRNE        OPDEF       (Ø15ØØØØ*↑+Ø145),(4,7,N)
ZRA         OPDEF       (Ø15ØØØØ*↑+Ø165),(4,7,N)
ABV         OPDEF       (Ø15ØØØØ*↑+Ø6),(4,7,N)
ABVE        OPDEF       (Ø15ØØØØ*↑+Ø26),(4,7,N)
ABVL        OPDEF       (Ø15ØØØØ*↑+Ø1Ø6),(4,7,N)
ABVLE       OPDEF       (Ø15ØØØØ*↑+Ø126),(4,7,N)
ABVG        OPDEF       (Ø15ØØØØ*↑+Ø46),(4,7,N)
ABVGE       OPDEF       (Ø15ØØØØ*↑+Ø66),(4,7,N)
ABVNE       OPDEF       (Ø15ØØØØ*↑+Ø146),(4,7,N)
ABVA        OPDEF       (Ø15ØØØØ*↑+Ø166),(4,7,N)
```

```
SLO       OPDEF     (Ø150000*↑+Ø204),(4,7,N)
SLOE      OPDEF     (Ø150000*↑+Ø224),(4,7,N)
SLOL      OPDEF     (Ø150000*↑+Ø304),(4,7,N)
SLOLE     OPDEF     (Ø150000*↑+Ø324),(4,7,N)
SLOG      OPDEF     (Ø150000*↑+Ø244),(4,7,N)
SLOGE     OPDEF     (Ø150000*↑+Ø264),(4,7,N)
SLONE     OPDEF     (Ø150000*↑+Ø344),(4,7,N)
SLOA      OPDEF     (Ø150000*↑+Ø364),(4,7,N)
REX       OPDEF     (Ø150000*↑+Ø7),(4,7,N)
CE        OPDEF     (Ø150000*↑+Ø12),(4,7,N),(4,←-5,N)
CNE       OPDEF     (Ø150000*↑+Ø13),(4,7,N),(4,←-5,N)
CEI       OPDEF     (Ø150000*↑+Ø14),(4,7,N),(←-12,←-5,N)
CNEI      OPDEF     (Ø150000*↑+Ø15),(4,7,N),(←-12,←-5,N)
CEMI      OPDEF     (Ø150000*↑+Ø16),(4,7,N),(←-12,←-5,N)
CNEMI     OPDEF     (Ø150000*↑+Ø17),(4,7,N),(←-12,←-5,N)
ADD       OPDEF     (Ø60000*↑),(4,7,N),(4,←-5,N)
ADDNC     OPDEF     (Ø160000*↑),(4,7,N),(4,←-5,N)
ADDI      OPDEF     (Ø60000*↑+Ø10),(4,7,N),(←-12,←-5,N)
ADDINC    OPDEF     (Ø160000*↑+Ø10),(4,7,N),(←-12,←-5,N)
SUB       OPDEF     (Ø60000*↑+Ø1),(4,7,N),(4,←-5,N)
SUBNB     OPDEF     (Ø160000*↑+Ø1),(4,7,N),(4,←-5,N)
SUBI      OPDEF     (Ø60000*↑+Ø11),(4,7,N),(←-12,←-5,N)
SUBINB    OPDEF     (Ø160000*↑+Ø11),(4,7,N),(←-12,←-5,N)
OR        OPDEF     (Ø60000*↑+Ø5),(4,7,N),(4,←-5,N)
ORZ       OPDEF     (Ø160000*↑+Ø5),(4,7,N),(4,←-5,N)
XOR       OPDEF     (Ø60000*↑+Ø3),(4,7,N),(4,←-5,N)
XORZ      OPDEF     (Ø160000*↑+Ø3),(4,7,N),(4,←-5,N)
XORNZ     OPDEF     (Ø160000*↑+Ø6),(4,7,N),(4,←-5,N)
AND       OPDEF     (Ø60000*↑+Ø4),(4,7,N),(4,←-5,N)
ANDZ      OPDEF     (Ø160000*↑+Ø4),(4,7,N),(4,←-5,N)
ANDNZ     OPDEF     (Ø160000*↑+Ø7),(4,7,N),(4,←-5,N)
PUSH      OPDEF     (Ø70000*↑),(4,7,N),(4,←-5,N)
IPUSH     OPDEF     (Ø70000*↑+Ø4),(4,7,N),(4,←-5,N)
PUSHI     OPDEF     (Ø70000*↑+Ø6),(4,7,N),(4,←-5,N)
DPUSH     OPDEF     (Ø70000*↑+Ø10),(4,7,N),(4,←-5,N)
PUSHD     OPDEF     (Ø70000*↑+Ø12),(4,7,N),(4,←-5,N)
POP       OPDEF     (Ø70000*↑+Ø1),(4,7,N),(4,←-5,N)
IPOP      OPDEF     (Ø70000*↑+Ø5),(4,7,N),(4,←-5,N)
POPI      OPDEF     (Ø70000*↑+Ø7),(4,7,N),(4,←-5,N)
DPOP      OPDEF     (Ø70000*↑+Ø11),(4,7,N),(4,←-5,N)
POPD      OPDEF     (Ø70000*↑+Ø13),(4,7,N),(4,←-5,N)
REGJ      OPDEF     (Ø70000*↑+Ø14),(4,7,N),(←-12,←-5,N)
REGPJ     OPDEF     (Ø70000*↑+Ø15),(4,7,N),(←-12,←-5,N)
REGJS     OPDEF     (Ø70000*↑+Ø16),(4,7,N),(←-12,←-5,N)
POPJ      OPDEF     (Ø72000*↑+Ø16)
POPJOF    OPDEF     [POPJ],(←-12,←-5,N)
IOT       OPDEF     (Ø170000*↑),(4,7,N),(←-8,←-1,N)
SLEEP     OPDEF     [IOT      ,Ø10]
SETA      EQU       Ø
SETR      EQU       1
SETV      EQU       2
TOA       EQU       4
TOR       EQU       5
TOV       EQU       6
DOTA      EQU       Ø10
DOTR      EQU       Ø11
DOTV      EQU       Ø12
```

```
BOXA      EQU      Ø14
BOXR      EQU      Ø15
FRMA      EQU      Ø16
FRMR      EQU      Ø17
BOX       EQU      Ø
NEWCRV    EQU      BOX
DOT       EQU      1
FROM      EQU      5
STAR      EQU      4
TO        EQU      2
POLY      EQU      3
SET       EQU      POLY
NLINE     EQU      7
LINE      EQU      6
RX        EQU      7
AX        EQU      6
RA        EQU      3
AA        EQU      2
AR        EQU      4
RR        EQU      5
VV        EQU      1
AV        EQU      Ø
SACØ      EQU      Ø
SAC2      EQU      2
SX        EQU      1
SZ        EQU      3
DACØ      EQU      4
DX        EQU      5
SAVELB    EQU      Ø
SAVERT    EQU      1
VIEWLB    EQU      2
VIEWRT    EQU      3
WINDLB    EQU      4
WINDRT    EQU      5
INSTLB    EQU      6
INSTRT    EQU      7
NAME      EQU      Ø1Ø
CDIR      EQU      Ø11
HITANG    EQU      Ø12
SELINT    EQU      Ø13
SAVE      EQU      Ø14
VIEW      EQU      Ø15
WIND      EQU      Ø16
INST      EQU      Ø17
LOCLA     OPDEF    (Ø4ØØØØ*↑+Ø6),(4,←-5,N)
LOCLR     OPDEF    (Ø4Ø4ØØ*↑+Ø6),(4,←-5,N)
LOCLSA    OPDEF    (Ø41ØØØ*↑+Ø6),(4,←-5,N)
LOCLSR    OPDEF    (Ø414ØØ*↑+Ø6),(4,←-5,N)
LOMMA     OPDEF    (Ø42ØØØ*↑+Ø6),(4,←-5,N)
LOMMR     OPDEF    (Ø424ØØ*↑+Ø6),(4,←-5,N)
LOMMF     OPDEF    (Ø43ØØØ*↑+Ø6),(4,←-5,N)
LOMDR     OPDEF    (Ø434ØØ*↑+Ø6)
STCL      OPDEF    (Ø14ØØØØ*↑+Ø6),(4,←-5,N)
STMM      OPDEF    (Ø142ØØØ*↑+Ø6),(4,←-5,N)
STMDR     OPDEF    (Ø1434ØØ*↑+Ø6)
RLOCLA    OPDEF    (Ø4ØØØØ*↑),(4,←-5,N),(3,←-1,N)
RLOCLR    OPDEF    (Ø4Ø4ØØ*↑),(4,←-5,N),(3,←-1,N)
```

```
RLOMMA    OPDEF     (Ø42ØØØ*↑),(4,←-5,N),(3,←-1,N)
RLOMMR    OPDEF     (Ø424ØØ*↑),(4,←-5,N),(3,←-1,N)
RLOMMP    OPDEF     (Ø43ØØØ*↑),(4,←-5,N),(3,←-1,N)
RLOMDR    OPDEF     (Ø434ØØ*↑),(3,←-1,N)
RSTCL     OPDEF     (Ø14ØØØØ*↑),(4,←-5,N),(3,←-1,N)
RSTMM     OPDEF     (Ø142ØØØ*↑),(4,←-5,N),(3,←-1,N)
RSTMDR    OPDEF     (Ø1434ØØ*↑),(3,←-1,N)
RTCLA     OPDEF     (Ø4ØØØØ*↑+Ø7),(4,←-5,N)
RTCLR     OPDEF     (Ø4Ø4ØØ*↑+Ø7),(4,←-5,N)
RTCLSA    OPDEF     (Ø41ØØØ*↑+Ø7),(4,←-5,N)
RTCLSR    OPDEF     (Ø414ØØ*↑+Ø7),(4,←-5,N)
RTMDR     OPDEF     (Ø434ØØ*↑+Ø7)
SKCL      OPDEF     (Ø14ØØØØ*↑+Ø7),(4,←-5,N)
RTMM      OPDEF     (Ø42ØØØ*↑+Ø7),(4,←-5,N)
RTMMS     OPDEF     (Ø424ØØ*↑+Ø7),(4,←-5,N)
SKMM      OPDEF     (Ø142ØØØ*↑+Ø7),(4,←-5,N)
SKMMS     OPDEF     (Ø1424ØØ*↑+Ø7),(4,←-5,N)
SKMDR     OPDEF     (Ø1434ØØ*↑+Ø7)
NOMM      OPDEF     (Ø1424ØØ*↑+Ø6)
POPMM     OPDEF     (Ø143ØØØ*↑+Ø6),(4,←-5,N)
PUSHMM    OPDEF     (Ø143ØØØ*↑+Ø7),(4,←-5,N)
SD        OPDEF     (Ø2ØØØØ*↑),(4,7,N),(←-8,←-1,A@%)
TD        OPDEF     (Ø4ØØØØ*↑+Ø16),(3,7,N),(3,←-5,N)
TDR       OPDEF     (Ø4ØØØØ*↑+Ø216),(3,7,N),(3,←-5,N)
TDI       OPDEF     (Ø14ØØØØ*↑+Ø16),(3,7,N),(3,←-5,N)
TDIR      OPDEF     (Ø14ØØØØ*↑+Ø216),(3,7,N),(3,←-5,N)
TDIX      OPDEF     (Ø144ØØØ*↑+Ø16),(3,7,N),(3,←-5,N)
TDIXR     OPDEF     (Ø144ØØØ*↑+Ø216),(3,7,N),(3,←-5,N)
RD        OPDEF     (Ø4ØØØØ*↑+Ø1Ø),(3,7,N),(3,←-5,N),(3,←-1,N)
MMD       OPDEF     (Ø4ØØØØ*↑+Ø57),(4,7,N)
MMDR      OPDEF     (Ø4ØØØØ*↑+Ø257),(4,7,N)
          END
```

APPENDIX III

A NOTE ON HOMOGENEOUS COORDINATES AND THE LDS-2

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 2. 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-2 system, the basic three-dimensional coordinates describing objects is stored in main memory in four consecutive words. These four 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=$00...00_2$. 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 a 24-bit LDS-2 system might be a cube of space centered at Zero and running from $-2^{23} = 10...0_2$ to $2^{23}-1 = 01...1_2$; 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 $W \to \infty$. 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

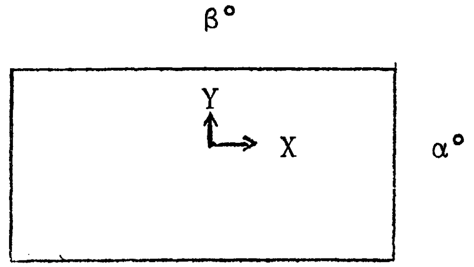
Homogeneous Coordinates — [CDST] Clipping Divider Switching Transformation — Clipper Input

$$[X,\ Y,\ Z,\ W]\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 1\\ 0 & 0 & 0 & 0\end{bmatrix} = [X,\ Y,\ Z_x,\ Z_y]$$

[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)

$$\begin{bmatrix} X, & Y, & Z, & 1 \end{bmatrix} \times \left[\begin{array}{ccc} & 3\times3 & \\ \hline j & k & l \end{array}\right] = \begin{bmatrix} X'', & Y'', & Z'' \end{bmatrix}$$

4X4 HOMOGENEOUS TRANSFORMATION
(ROTATION, SCALING AND TRANSLATION)

$$\begin{bmatrix} X, & Y, & Z, & 1 \end{bmatrix} \times \left[\begin{array}{c|c} & 0 \\ 3\times3 & 0 \\ & 0 \\ \hline 3\times1 & 1 \end{array}\right] = \begin{bmatrix} X'', & Y'', & Z'', & 1 \end{bmatrix}$$

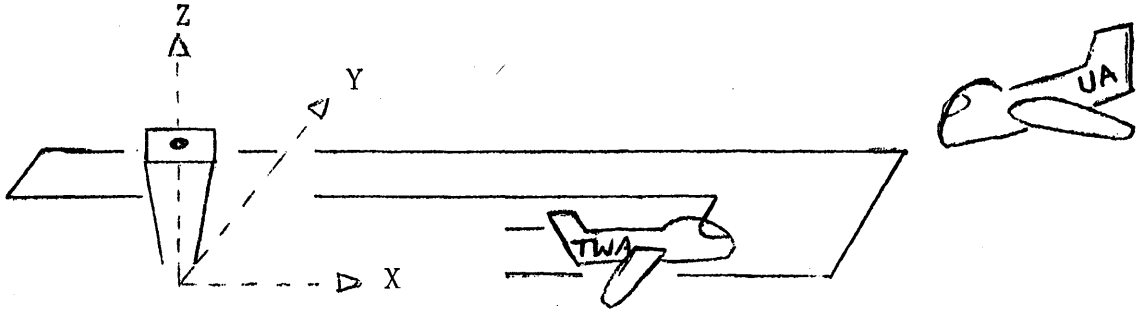
Figure AIII.2

Delete

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