75

A VIDEO GRAPHICS TERMINAL

A VERSATILE, HIGH SPEED, RASTER SCAN VIDEO GRAPHICS TERMINAL

By

PETER DOUGLAS MACDONALD, B.Sc.

A Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

April 1979

MASTER OF SCIENCE (1979)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario
Canada

TITLE: A Versatile, High Speed, Raster Scan Video Graphics Terminal

AUTHOR: Peter Douglas Macdonald, B.Sc. (Queen's University)

SUPERVISORS: Mr. G. J. Hicks
Professor K. A. Redish

NUMBER OF PAGES: viii, 112

ABSTRACT

The design of a flexible, high speed, raster scan graphics terminal is presented. The design is presented in general architectural terms rather than from a detailed circuitry point of view.

Control is divided between the 'main' microprocessor, an Intel 8086, and a subservient graphics controller which consists of a microprogrammable, bit-sliced, AM2903/2910 special purpose microprocessor. The high speed graphics controller is microprogrammed to accept basic line and circle generating commands. The configuration is felt to represent an efficient balance between simplicity and speed.

ACKNOWLEDGEMENTS

I wish to express my appreciation to my supervisors, Mr. G. Hicks and Professor K. Redish for their guidance and assistance during the preparation of this project. I would also like to acknowledge the very great contribution made by Mr. G. Hicks to the design proposed in this report. My special thanks to Mr. G. Hicks and Professor K. Redish for the patience and cooperation which they extended to me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Computers speak a language of bits, bytes, and registers. Man is more fluent with sketches, words, graphs, tables and digits. Each member of this unlikely pair, man and computer, can be indispensable to the solving of a problem at hand. Man is needed for his creative intuition and judgement. The computer is unmatched in its raw speed, its ability to remember large quantities of data, and in its willingness to perform simple repetitive tasks. In situations which require talents from both of the above categories, the interactive video graphics terminal provides a natural man-computer interface. Its usefulness is as an interpreter in the dialogue between man and machine. The development of this role has led to a wide range of computer graphics applications in fields ranging from engineering design to mathematical analysis to business data processing.

I. 1 Cathode Ray Tubes

At the heart of modern interactive graphics terminals, is the cathode ray tube (CRT). A CRT is large pear shaped tube which has been pumped free of air and closed off. At the narrow neck end of the tube there is an electron gun from which a continuous beam of electrons may be caused to emanate. At the other end of a tube there is a phosphorus display screen onto which the electron beam is focused. As electrons hit the screen and for a time afterwards, the phosphor glows and a spot of light is seen. The colour of this light depends on the type of phosphor used.

The electron beam is focused onto any point on the screen through the application of the proper 'horizontal deflection' and 'vertical deflection' analogue input signals. A third input signal controls the intensity of the beam and thus, indirectly, the intensity of the resulting screen dot. The range of this control includes the situation where the electron gun is turned off and there is no electron beam.

Interactive graphics terminals exchange information with a user on the one hand, and a 'host' computer on the other. Much of the information which is received from the user is passed on to the host computer. Information received from the host computer is translated, by the graphics terminal, into corresponding CRT beam control input signals. These signals, in turn, cause a visual image to be generated on the face of the CRT screen. In this way, the user and host computer communicate indirectly via the visual display.

Since phosphor glows for only a short time after the termination of an electron bombardment, CRT images are transient and need to be regenerated regularly in order to obtain a steady and coherent picture. The so called refresh rate is usually 30-60 Hz, depending on the type of phosphor used. Refresh rates which are too low cause an annoying 'flicker' whereby the display is noticeably discontinuous in time.

I. 2 Controlling CRT Displays

Two distinctive methods for generating and manipulating CRT displays have evolved. There are, accordingly, two general types of interactive graphics terminals, the directed beam terminals and the raster scan terminals. With directed beam terminals, digital display data is converted into analogue waveforms which are used to drive the

horizontal deflection and vertical deflection inputs of the CRT. As a result, the electron beam is directed, under program control, about the display screen. By controlling when the electron beam is on and when it is off, continuous or discontinuous outlines of display images are traced onto the screen. The approach yields good line quality.

With directed beam refresh terminals, the flicker problem ultimately results in an upper limit to the amount of information which can be displayed. The display must be simple enough that it can be completely generated in the time between two successive screen refreshes. High speed, but expensive, vector generators can be used to raise the image complexity limit.

Alternatively, some directed beam terminals use storage tubes which, unlike conventional CRT's, do not require that the image be refreshed. While this strategy eliminates flicker problems, storage tubes are expensive, inherently less bright, and wear out much more quickly than refresh CRT's. A further disadvantage is that the entire screen must be erased in order to delete any part of the picture.

With raster scan terminals, the screen is treated as a rectangular mesh of defineable dot positions. For example, a screen may be characterized by a matrix of 500 x 600 discrete positions. A graphics display is composed of the complete set of defineable dots, each of which is either bright or dark according to the specifications of a bit map. This bit map is maintained in an internal memory and is such that each bit in the map corresponds to a single screen position. During each screen refresh, the contents of the bit map are mapped onto the screen. Since the bit map is commonly maintained in random access read/write memory, the image can be selectively modified by simply

changing the contents of the bit map.

The words 'raster scan' refer to the manner in which the information contained in the bit map is mapped onto the display screen. The term, raster, refers to the complete set of horizontal lines defined by the discrete screen positions. Each raster line consists of a separate row of screen dots. At the start of each screen refresh, a new scan of the raster is begun. Starting with the electron beam focused on the top leftmost screen position, the beam is automatically made to scan through each raster line, in turn, from left to right. Upon reaching the end of each line, the leftmost dot of the next line down is selected as the starting position for the next horizontal scan. Upon reaching the bottom line, the beam is once again directed to the top raster line in preparation for the next screen refresh.

As the raster is scanned, the corresponding bits in the bit map are accessed, in sequence, and used as ON/OFF input signals for the electron gun. During the course of a complete raster scan, each bit in the bit map is accessed exactly once and used momentarily as the electron gun ON/OFF input signal. For each bit, this occurs at the precise moment that the electron gun is focused at the bit's corresponding screen position. In this manner, the information contained in the bit map is transformed into the graphics display.

For a time, the directed beam storage-tube terminals represented the only choice in relatively low cost graphics terminals. Memory prices rendered raster scan graphics impractical. With the advent of high density, inexpensive semiconductor memories, however, raster scan graphics terminals have become an economically attractive, alternative,

low-cost terminal. The more expensive refresh directed beam terminals remain uniquely useful in situations which require high speed and good line quality.

I. 3 Design Proposal

This report presents the design of a raster scan graphics terminal. In this design, commercially available LSI chips and, in particular, microprocessors are used to control the terminal. The graphics processing is distributed amongst two microprocessors which execute in parallel. A microprocessor can be thought of as a programmable logic device which can be made to synthesize any sequence of individual logic devices. In effect, it corresponds to the CPU of a digital computer. The term, microprocessor, usually refers to a single chip CPU although it can be extended to include 'bit-sliced' CPU's which consist of several LSI chips.

The control program of a microprocessor is stored in a separate memory. It is this program which gives the microprocessor its distinct personality. Since the control program is usually stored in Read Only Memory (ROM), it is thought of as being more 'firm' than regular software. Hence, it is commonly referred to as the control firmware.

The advantages of microprocessor-based design are several. Rather than design special purpose and, in comparison, inflexible hardware, many terminal control functions are easily programmed. The resulting firmware can be of a very general nature. Sophisticated, host independent capabilities can be programmed into the terminal without any accompanying increase in hardware complexity. Furthermore, future additions to the firmware are fairly readily accommodated. As more 'intelligence'

is built into the terminal, the burden on both the user and the host computer is reduced. Finally, the reduction in the number of discrete integrated circuits, results in low cost, low maintenance and high reliability.

A price for these benefits is paid for in processing speed. Hardwired logic can be made to operate faster than programmed logic. This compromise in speed provides the principle motivation for incorporating two microprocessors into the design. By distributing the graphics processing among two microprocessors rather than one, the terminal performance is significantly enhanced. The objective is the design of a capable raster scan graphics terminal which is able to keep pace with modern host-terminal transmission rates. These rates may be as high as 9600 baud (960 characters per second). This means that the terminal must be able to process incoming characters at the average rate of one character every millisecond.

In the proposed design, control is divided between the 'main' microprocessor, an Intel 8086, and a subservient graphics controller which consists of a bit-sliced AM 2903/2910 special purpose microprocessor. The main microprocessor supervises the hardware interfaces with the user, host computer, display hardware and graphics controller. It sends graphic commands to the high speed graphics controller whenever the contents of the bit map are to be altered. The configuration is felt to represent an efficient balance between simplicity and speed.

The graphics controller has been referred to as a 'bit-sliced' microprocessor. The precise meaning of this terminology shall be made clear in chapter 3. For the moment, it suffices to mention two prominent

aspects of the bit-sliced technology. Firstly, the bit-sliced approach enables the design of a highly specialized CPU specifically equipped for the task at hand. Secondly, bit-sliced microprocessors are microprogrammable. Microprogramming can be thought of as a level of programmable control below that provided by machine language. It is, in effect, the programming of the control unit of a CPU. It is inherently more complicated than regular machine language programming although, as a reward, the efficient use of microprogramming techniques can lead to further increases in speed.

In this report, the design is presented in general architectural terms rather than from a detailed circuitry point of view. A discussion of overall terminal design is followed by a more detailed look at the microprogrammable graphics controller. Finally, algorithms which provide basic graphics capabilities are developed.

CHAPTER 2

BASIC DESIGN

The hardware architecture of the video graphics terminal is shown in figure 2.1. Control firmware stored in the local memory is referred to as the operating system. It is executed by the main microprocessor (Intel 8086). Programmable control also resides in a separate control store contained in the bit-sliced graphics controller. The two programmable processors are able to execute in parallel. They provide for the basic flexibility of this design.

At any one time, the video display is entirely defined by the digital information contained in both the text memory and graphics bit map. The two separate memories are characteristic of the dual graphic and text roles of which the terminal is capable. The bit map defines the graphic image as described earlier. The text memory enables the terminal to act as a conventional alphanumeric terminal as well. The contents of the text memory are mapped onto the screen as characters. This is accomplished through the use of a character generator. Each alphanumeric character can be thought of as consisting of a rectangular matrix of dots. Each matrix is identical in size, depending only on character height and width. These matrix patterns are stored in the character generator and are accessed simply by providing the appropriate addresses.

In a typical application, the user's dialogue with the host computer is stored in the text memory while the graphic commands are

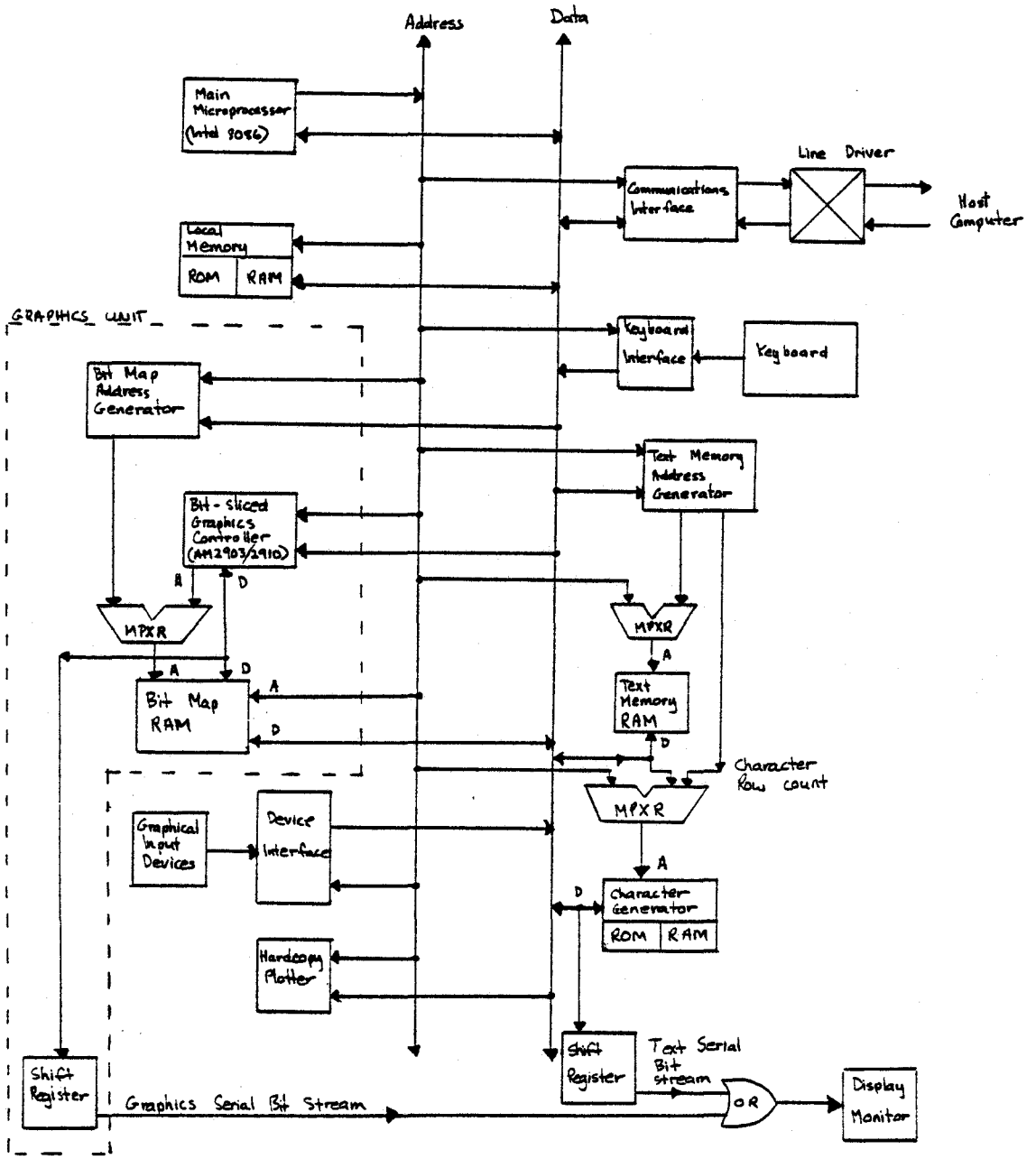


Fig. 2.1: The Video Graphics Terminal

sent to the graphics controller where they are transformed into graphics output to be stored in the bit map. The two independent text and graphics memories allow maximum user flexibility. Via the keyboard, the user may request that the contents of both memories be displayed simultaneously.Alternatively, either of the text or graphics displays may be viewed singly. At any moment, the user's dialogue with the host may be recalled to the screen for viewing or temporarily dismissed while the graphical display is studied.

II. 1 Five Concurrent Processes

The normal terminal operation can be understood in terms of five concurrent processes. The dialogue process enables communication between user, terminal and host computer to take place. The text definition and graphics definition processes are responsible for depositing information into the text memory and bit map respectively. The text display and graphics display processes are concerned with mapping this information onto the CRT display screen at the proper refresh rate.

Each of the five processes involves the operating system in some manner. The graphics and text display processes are interrupt driven. Each time a new frame needs to be displayed, the operating system is interrupted and the display process serviced. This is accomplished by sending the appropriate starting values to the graphics and text address generators. Both display processes then proceed under automatic hardware control. The operating system is free to resume its own processing. There are, however, important limitations to this freedom which will be discussed later. The 'background' or interrupt-enabled processing, consists of either the text definition or graphics definition process.

The dialogue process is also interrupt driven. The host computer communications interface issues data transmit and receive interrupts to the operating system. The dialogue process services these interrupts. The keyboard and other slow input devices which interface with the user, are conveniently polled at frame interrupt time, and therefore, do not require separate interrupts. It should be noted that part of the graphics display process is under control of the bit-sliced graphics controller and is independent of the operating system. Each of the five concurrent processes will now be described in detail. For these descriptions, the reader should refer to figure 2.1.

II. 1.1 Dialogue Process

The dialogue process resides in the operating system. It receives input in the form of ASCII characters from the user via the keyboard. If a received character does not correspond to a local command, it is passed onto the host computer where it is interpreted by mainframe software. The host computer replies in accordance with its interpretation by returning more encoded ASCII characters. These characters are received and stored in a character receive buffer in local memory which acts as an interface between the dialogue and definition processes. This arrangement can be used to afford the user either direct or indirect control over the display. Local commands can cause ASCII characters to be placed directly in the character receive buffer without their having come from the host computer.

Further dialogue with the user is provided by means of graphical input devices. Devices like the joystick, tracker ball, and mouse may be interfaced with the terminal (NEW). The 'steering' control of these

devices results in corresponding digital x and y coordinate values. Each input device is polled at the 60Hz frame interrupt rate. Every time a new effective x, y screen position is read, a request is issued to the graphical display process to change the position of the small cross or graphics cursor on the screen. Visual feedback enables the user to position the graphics cursor at any addressable point on the screen. Proper use of this feature greatly enhances the interaction between terminal and user. Among other functions, the user is able to specify lines through the use of endpoint positioning.

Notice the use made by the dialogue process of the 60Hz frame interrupt rate. The regularity of this interrupt makes it an effective real time clock. As such, it is frequently very useful in controlling the several 'rates' which are maintained by the operating system. The text cursor 'wink', automatic key repeat, keyboard polling, and text scrolling, are among some of the functions which require a real time clock.

II. 1.2 Text Definition Process

While the dialogue process is filling up the character receive buffer, the text definition and graphics definition processes are busy emptying it. Control of the text definition process also resides in the operating system. The characters which are passed to it are written in ASCII form into the text memory. At any one time, several screenfuls of alphanumeric text are stored in the text memory. Paging, scrolling and text cursor motion are accomplished by changing the first word address of the text display. General text editing capabilities may also be provided.

There are occasions where it is useful to write nonstandard text characters onto the screen. Although the standard ASCII character bit patterns are stored in ROM and can't be altered, the character generator also includes some RAM memory. This provision enables nonstandard character sets to be downloaded from the host computer.

II. 1.3 Graphics Definition Process

The graphics definition process is organized somewhat differently from the text definition process. Although some graphics processing is done by the operating system, most is delegated to the high speed graphics controller. This is necessary in order to handle situations where graphic commands are coming from the host computer at a very high rate. Whereas ASCII characters passed to the text definition process cause only a few text memory accesses, a character received by the graphics definition process may result in the drawing of a vector which requires hundreds of bit map accesses.

Any graphic image may be thought of as being composed of a set of vectors between specified points. From this perspective, a minimal requirement of the graphics controller is the ability to 'draw' vectors which are arbitrarily positioned on the screen. The graphics controller reads in the vector command sent by the operating system and then proceeds to modify the bit map accordingly. Single points are drawn by specifying zero length vectors. The graphics controller must also be able to 'erase' vectors. This is simply a matter of placing zeros in the bit map instead of ones. A complement mode is useful whereby bits in the map are 'flipped' rather than set or erased. This feature requires that the graphics controller be able to read as well as write to the bit map.

In many instances, it is desirable to include alphanumeric text as an integral part of the graphic display. For example, labels to diagrams must remain fixed with respect to the display. Regular text from the text memory is scrolled past the graphic display in the conventional manner of alphanumeric terminals. It is clear that the character bit patterns must be written directly into the display bit map. A convenient arrangement involves translating the bit patterns into graphical commands capable of being accepted by the graphics controller. This is accomplished by the operating system which reads the bit patterns from the character generator, translates them into a series of vector instructions, and sends these to the graphics controller. The operating system can also perform effective scale, rotating and slanting operations on the characters.

Up until now, the discussion has focused on the depositing of the proper display information into the text and graphics memories. Now, attention is shifted to getting information out of storage and onto the screen. Once every 1/60 sec,two processes are activated, whereby, the contents of both storage memories are simultaneously mapped onto the CRT screen.

II. 1.4 Text Display Process

The text display process will be described first. At the beginning of each screen refresh, the operating system sends the first word address of the text display to the text address generator. In hardware terms, the address generator simply consists of a few counting registers and some control logic. It provides the contiguous sequences of text memory addresses which correspond to the rows of text.

As an example, consider ASCII characters represented in the character generator as 16 by 8 dot matrices. Each matrix row is stored as a separate byte. The 8 bit ASCII character codes stored in the text memory are used as addresses to the corresponding character patterns. In order to fully specify a row within a character pattern, a four bit quantity must also be provided. Together, the four bit character row count and ASCII character code constitute a presentable character generator address.

Imagine that the CRT screen is wide enough to accommodate exactly 80 characters in a row of text. At screen refresh time, the character row count is set to zero just as the electron beam is about to begin a horizontal trajectory along the top raster line of the terminal screen. The first word address of the text display is interpreted as the address of the text memory byte which corresponds to the leftmost character of the top text row to be displayed.

As the electron beam sweeps across the screen, the first raster line is displayed by generating a sequence of 80 contiguous text memory addresses. This sequence begins with the first word address. The result is a sequence of 80 ASCII character codes being presented, in combination with the zero valued character row count, as addresses to the character generator. The outputs of the character generator are just the topmost rows of the specified 80 characters matrix bit patterns. These bytes are loaded in sequence into the shift register and clocked out at video rate as a serialized bit stream to the electron gun.

Upon reaching the end of its horizontal trajectory, the electron

beam is shut off and swung quickly back (horizontal flyback) into position for the display of the next raster line. The next raster line is displayed by incrementing the character row count by one and generating the exact same sequence of text memory addresses. This process is continued until all 16 raster lines of the first row of text have been displayed. At this time, the row count is reset to zero and the first word address is incremented by 80. It now corresponds to the first character of the next row of text. When a complete screenful of text has been displayed, the electron beam is repositioned (vertical flyback) so as to be ready for the next complete raster.

II. 1.5 Graphics Display Process

The graphics display process is organized in a similar way. At the beginning of a screen refresh, the first word address is sent to the graphics address generator. The graphics bit map is row-ordered so that the raster-scan motion of the electron beam corresponds to traversing through contiguous words in the bit map. As a result, the graphics display process consists simply of the production, by the graphics address generator, of contiguous memory addresses. The bit map is read at appropriate intervals and the output is clocked at video rate to form a second serial bit stream to the electron gun.

The two resultant bit streams from the text and graphics display processes are subject to the same timing constraint. Both streams must be such that they can be OR'ed together to form a single digital signal representative of both the text and graphics displays. As well, depending on the display mode, either of the bit streams may be

inhibited. In this way, the display can be made to consist entirely of text or entirely of graphics. Alternatively, text and graphics may appear together on the screen.

II. 2 Fully Interlaced Display

Until now, we have assumed that during each screen refresh, all the raster lines are displayed. For television compatibility (HOL), a fully interlaced display may be implemented instead. This requires that during any screen refresh, only half the raster lines are displayed. Consider raster lines to be numbered from the top to the bottom of the screen. A fully interlaced display is one which is characterized by alternate display frames of only the evenly numbered lines and only the oddly numbered lines.

The implementation of a fully interlaced display has implications for both the text and graphics display processes. For example, the character row count in the text process is incremented by two during each horizontal flyback instead of by one as before. During the vertical flyback it is initialized according to the next frame type as either zero or one. Perhaps the most significant implications, however, has to do with the graphics display process. The simplest way to accommodate a fully interlaced display is to split the bit map into two. One contains the information for the even raster frames and the other, the information for the odd raster frames. The organization of both bit maps is still row-ordered. Depending on whether an even or odd frame is to be displayed next, the starting graphics display address is the first word address of either the even or odd bit maps.

II. 3 Hardcopy Dump

A sixth process may be discerned from the operation of the graphics terminal. It is different from the others in that it is only infrequently activated at the user's command. This process has to do with providing the user with a hardcopy of the graphic image. The simplest way to provide this capability is to enable the operating system to read the bit map directly. The dual port bit map memory shown in figure one affords this access. When a graphics hardcopy is desired, the bit map is read by the operating system and sent to the hardcopy plotter.

Until recently, a problem with this approach has been the relatively small address space of available microprocessors. Typically, they could address up to only 64 K bytes. The bit map alone may easily require 40 K 8 bit bytes. This would leave minimal space for the operating system and separate text memory. Use of the recently introduced Intel 8086 is one method of eliminating this problem (MOR). It has an addressing capability of 1 M byte.

II. 4 Process Coupling

Earlier, passing comment was made on limitations to the independence of the display and definition processes. This point is now addressed. The six processes which have been discussed, are coupled with one another by virtue of their shared resources. The integrity of their operation must be carefully guarded by properly ensuring mutual exclusion (HAN). A single resource must not be assigned to more than one process at the same time.

The text definition and text display processes both share access to the text memory and character generator. Conflict resolution logic is required to prevent memory from being simultaneously accessed. A simple solution is to inhibit the main microprocessor from bus access except during horizontal and vertical flybacks. With the Intel 8086, external bus access is stopped by driving the 'hold' pin high. Simple analysis reveals that this restriction on operating system processing does not prevent the terminal from operating effectively.

For example, consider the terminal specifications which are presented in table 2.1. A host-terminal transmission rate of 9600 baud requires that the terminal be able to process one character every millisecond. In the processing of a single character by the operating system, the dialogue process receives it. The text definition process writes it into text memory or interprets it as a text command of some sort. Alternatively, if passed to the graphics definition process, it is 'interpreted', and as a result, information is usually sent to the graphics controller. This processing must last, on average, no longer than one millisecond.

One millisecond corresponds to the time required to display approximately 15 raster lines. With the operating system processing restricted to horizontal and vertical flybacks, the actual operating system processing time is only 225 usec. This corresponds to approximately 625 Intel 8086 instructions which should be more than enough to accomplish the processing required.

TABLE 2.1

TYPICAL VIDEO GRAPHICS TERMINAL SPECIFICATIONS

Resolution	480 x 650 dots
Frame Interrupts	60 Hz
Fully Interlaced Display	
Horizontal Electron Beam Scan	53 usec
Horizontal Flyback	15 usec
Vertical Flyback	280 usec
Average Execution Time of Intel 8086 Instruction	.36 usec

Objective 9600 Baud (960 char/sec)
Host-terminal transmission rate or
approximately 1 character every msec

A necessary provision is, of course, that the graphics controller is fast enough. New information can only be sent to the graphics controller if it is ready to accept it. The performance of the graphics controller will be discussed after the bit-sliced design has been looked at in detail. It suffices to say at this point, that a basic rationale for the graphics controller is its parallel operation with the text and graphic display processes as well as with the operating system. Unlike the operating system, it continues to operate during the horizontal raster line time.

To realize this parallelism, special contention logic is required to administer the sharing of another resource, namely, the graphics bit map. A flag protocol is used to resolve conflicts between the graphics controller and graphics display process. The graphics display process raises a hardware flag whenever it requires access to the bit map. After the access, the flag is lowered until the need arises again. The graphics controller must consult this flag before every bit map access. If it is raised, it simply waits for it to be lowered before proceeding. The flag is raised by the display process some time in advance of the actual access. This provides the graphics controller with the time to finish any access of its own, which was begun before the flag was raised.

A third process interested in accessing the bit map is the graphics hardcopy display process. When a hardcopy plot is being made, the graphics definition and graphics display processes are suspended. The operating system does this by disabling the visual display and secondly, by refusing to send any further commands to the graphics

controller. Eventually, the graphics controller, after having finished with its last received command, rests in an idle loop. It sets a hardware idle flag in order to communicate this idle condition to the operating system. The operating system then initiates the bit map dump. Since both display processes are disabled, it proceeds without competition.

It should be pointed out that the graphics hardcopy display process corresponds to one of only two situations where the operating system accesses the bit map directly. In both cases the protocol is to disable the visual display and the graphic definition processes. If this were not the case, further hardware logic would be required to resolve conflicts between the graphics controller and operating system. The other instance where the bit map is accessed by the operating system, is during the clearing of the graphic display. The operating system does this by loading zeros directly into the bit map.

Further process coupling involves the dialogue and both definition processes. They all share the use of the character receive buffer. Since the dialogue process is interrupt driven, the contention logic is simple. It consists in momentarily disabling interrupts at appropriate places in the definition processes.

II. 5 Modularity

As mentioned earlier, the terminal has dual text and graphics roles. It is possible to split the hardware design along these lines. The dotted lines in figure 2.1 outline the various units which provide the terminal with graphics capabilities. These may be incorporated into a single functional unit. Essentially, the final product becomes

two. A conventional alphanumeric terminal which, when interfaced properly with the separate graphics unit, becomes a full fledged graphics terminal.

This flexibility is characteristic of the modular design approach. With this approach, a design is broken into relatively independent functional units or modules. The overall design problem is then reduced to the proper interfacing of these modular units. The details of each module's design are then faced separately. In the light of this design concept, it is now appropriate to focus attention on the graphics controller. It is this module which comprises the heart of the graphics unit. We have discussed its purpose within the context of the overall design. Now, it is time to look inside the black box.

CHAPTER 3

THE GRAPHICS CONTROLLER

The graphics controller is designed as a bipolar microprogrammable bit-sliced microprocessor. Compared with single-chip MOS microprocessors, the bit-sliced bipolar approach represents a fundamentally different philosophical direction. The current limitations, associated with bipolar technologies, on chip complexity, pin numbers,and chip size, dictate that the CPU be implemented on a multichip basis. This is realized by splitting the CPU and implementing the control and processing units on separate chips. The processing section or arithmetic logic unit (ALU) is itself dispersed over several chips. The manner of this dispersion is understood by imagining a single ALU which is vertically sliced into identical 'bit-slices'. These slices operate in parallel and may be cascaded to any width which is a multiple of the basic slice. The basic slice is usually only two or four bits wide. Although inherently less reliable, the use of more chips is justified with the increased performance and flexibility.

Hardware flexibility arises from the fact that the designer essentially builds his own customized CPU. The chips can be configured to provide a wide variety of digital system architectures. Unconventional word lengths can be provided by simply stacking the desired number of bit-sliced ALU chips together. As well, there is the programming flexibility which is inherent in the use of a microprogrammable control unit. The designer may define the system's

instruction set by a program (microprogram) stored in ROM. The microinstructions which constitute the microprogram, provide for a very low level control over the hardware resources. As such, they can be used to implement a very efficient graphical instruction set.

For our purposes, advantage is taken of this flexibility in order to design a special purpose processor. The processor instruction set is an atypical set designed for the express purpose of manipulating the bit map. Many of these instructions are relatively high level and correspond to hundreds of microinstructions. The graphics instructions are not fetched from a central memory in the manner of a conventional CPU but are received from the main microprocessor. In order to see how the graphics controller is realized, it is necessary to first introduce some fundamental concepts in microprogramming.

III. 1 Microprogramming

The central processing unit of a computer can be logically divided into a control unit and an execution network as shown in figure 3.1. The current instruction is contained in the instruction register (IR). In general, it requires several clock cycles to execute, depending on the particular instruction. The control unit decodes the instruction and as a result, emits control signals or commands to the execution network. A new set of commands is issued at every clock cycle. More precisely, at each cycle, the control unit sends a set of 'boolean' signals which defines the behaviour of the components of the execution network for the duration of that cycle. This boolean command vector can be regarded as a word, the command word.

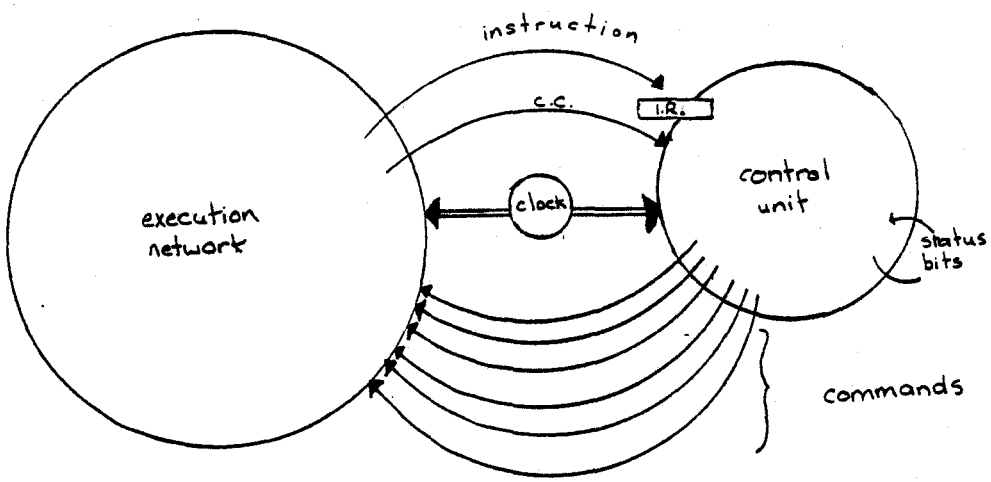


Fig. 3.1: The central processing unit of a computer can be logically divided into a control unit and an execution network (BOU).

Conditional instructions require information concerning the current status of the execution network. Such information is generally called the condition code (CC). It may, for example, consist of the value of a carry, the sign of the last result, or of an overflow status. This information is maintained by the execution network and made available to the control unit at appropriate moments.

After the current instruction has been executed, the control unit issues a sequence of command words which cause the next instruction to be placed in the instruction register. This sequence of command words corresponds to what is referred to as the fetch cycle. It is followed by an execute cycle whereby the instruction is executed. This alternate sequence of fetch and execute cycles continues for as long as the CPU is in operation. It should be stressed that each fetch and execute cycle consists of several clock cycles.

In the course of executing an instruction, the control unit is solely responsible for presenting the correct sequence of command words to the execution network. Consequently, the control unit must maintain a number of internal 'status bits' so as to keep track of the state of its own processing. The form of these status bits depends on the particular design of the control unit itself. At each clock cycle, the control unit must therefore perform two operations. It must generate a command word and update its internal status bits.

Traditionally, there have been two approaches to the realization of a suitable control unit. One method consists of using random hard-wired logic to generate each control word at every clock cycle. The second method is the microprogrammed solution. Command words are stored

in a memory called the control store. At each clock cycle, a new command word is available from memory and sent to the execution network. The internal status bits now correspond to the specification of the next command word to be used.

A single word contained in the control store is called a micro-instruction. It consists not only of the command word, but, as well, of the second set of boolean commands which are used to update the internal designation of the next microinstruction. Whereas, at each clock cycle, the command word is sent to the execution network, the next microinstruction command vector is sent to the next - micro-instruction - logic of the control unit itself.

Just as regular computer instructions can be combined to form a program, so a logically coherent sequence of microinstructions is called a microprogram. The terms, instruction, fetch and execute, are now prefixed with 'macro' to distinguish them from the micro-fetch and micro-execute cycles associated with the execution of each micro-instruction. The execution of each macroinstruction normally corresponds to the execution of a single associated microprogram.

Each microinstruction is logically divided into component fields. Each field corresponds to a functionally independent set of boolean commands. It can consist of varying numbers of bits. For example, the carry control, ALU function control, and next microinstruction address control, may be three separate microinstruction fields.

III. 2 Basic Architecture

Figure 3.2 shows the basic design of the graphics controller. The figure is divided into the control unit and the ALU. The ALU

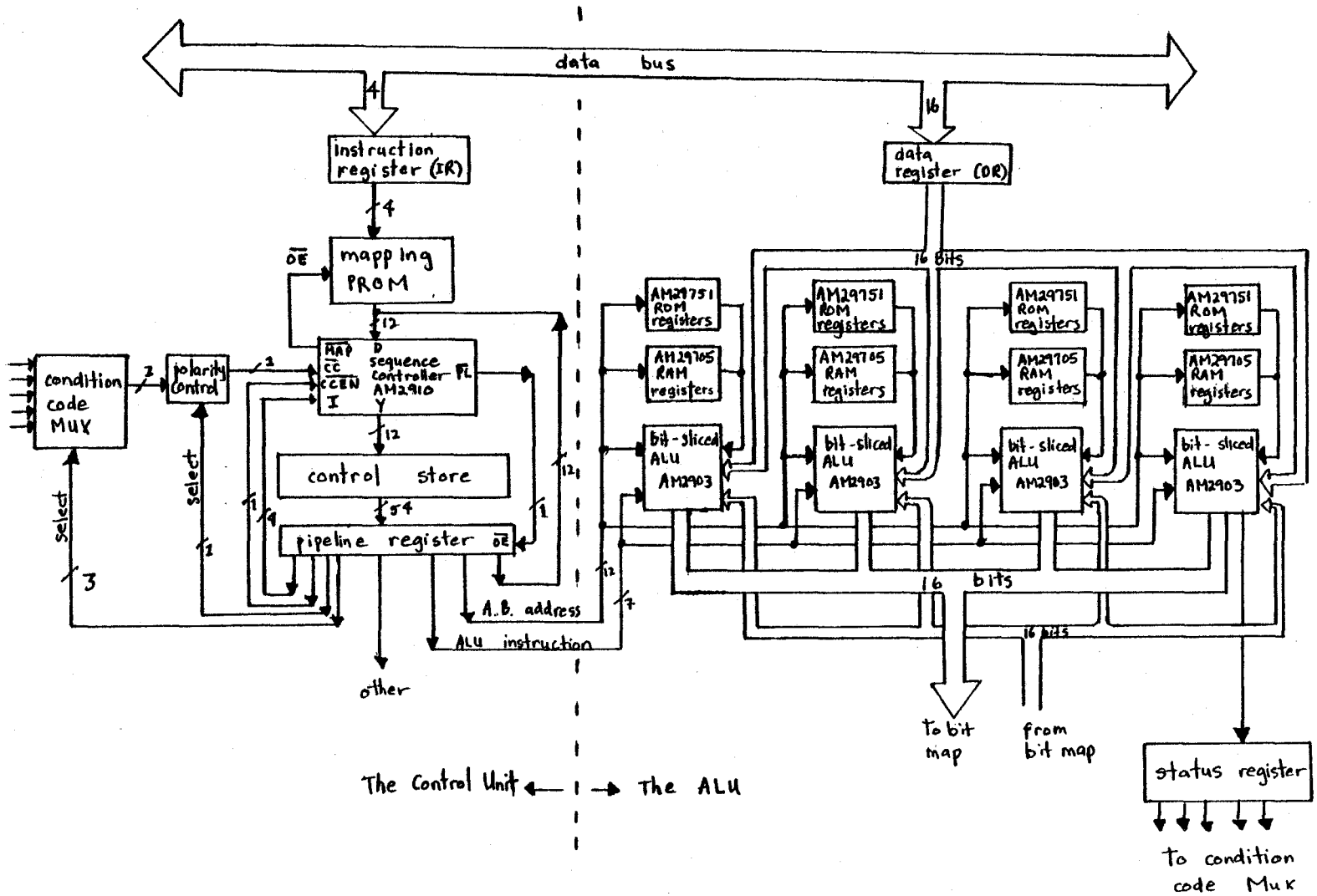


Fig. 3.2 : The Graphics Controller

consists of four 4 bit wide ALU chips (AM2903's) cascaded together to form a 16 bit processor. The AM2903 contains 16 internal working registers. This working space is increased through the addition of extra RAM (AM29705) and PROM (AM29751) bit-sliced register files. The final ALU contains 32 RAM and 16 PROM registers.

At each clock cycle, a new microinstruction is latched from the control store into what is called a 'pipeline register'. The contents of this register provide control inputs to the sequence controller (AM2910), the ALU, the bit map interface, and the data bus interface. In terms of the previous discussion, the latter three components constitute the execution network. The sequence controller is the next - microinstruction - logic of the control unit.

The execution network maintains arithmetic status information in the status register. This information, together with other various hardware I/O flags, constitutes the information from which the condition code is selected. It is available to the control unit through the condition code multiplexor.

The macroinstructions sent from the main microprocessor, include both an operation code (opcode), and several 16 bit words of data. The opcode is deposited in the instruction register while the data are sent to the data register. The mapping prom maps the contents of the instruction register into a starting microprogram address. During each macro-fetch cycle, this starting address is accepted by the sequence controller as the next microinstruction address. The instruction data are then read, under microprogram control, into the internal

register file. A 'handshaking' I/O protocol ensures that the opcode and data are received properly.

The pipeline register enables a time-saving technique known as 'pipelining'. With this technique, the next microinstruction is fetched at the same time that the current microinstruction is executed. By definition, the pipeline register always contains the microinstruction which is currently being executed. The contents of this register must remain fixed for the entire clock cycle. They define the current state of the execution network. While the contents of the pipeline register remain fixed, the next microinstruction can be fetched from the control store without affecting the integrity of the currently executing microinstruction.

This parallelism is one way of maximising the clocking frequency. In effect, there are two parallel processing paths. The first path involves the fetching of the next microinstruction. Boolean commands are sent from the control store to the sequence controller. These commands, in combination with the condition code input, cause the address of the next microinstruction to be presented to the control store. The corresponding memory word is fetched and set-up at the inputs to the pipeline register. This must all be accomplished during one clock cycle.

The second processing path is through the execution network. Control signals sent to the ALU cause arithmetical or logical operations to be performed on operands which are brought into the ALU from the data bus, bit map or the internal registers. After the results have stabilized, the contents of the status register are updated. The clocking period of the control unit and execution network must be at

least as long as the duration of the longest of the two parallel processing paths. Typically, for the AM2900 LSI series, the clock period is in the neighbourhood of 130 nsec.

Pipelining has a further implication. The status register is updated at the end of an execution network process path. However, its contents are required as input to the condition code multiplexor early in the fetching of the next microinstruction. This means that the condition code available to the control unit always corresponds to a previous state of the execution network. As a consequence, a microprogram branch, which is conditional on the results of the current microinstruction, cannot be specified until the next microinstruction.

III. 3 Microinstruction Fields

The graphics controller microinstruction contains four types of fields. These fields are concerned with the sequence controller, the ALU, the data bus interface, and the bit map interface, respectively. In the following discussion, each of these components will be discussed in more detail. As the discussion proceeds, the associated control fields will be defined. The result of this discussion is the specification of the complete set of microinstructions available for firmware control of the graphics controller.

For further details regarding the AMD (Advanced Micro Devices, Inc.) devices which are presented in the following discussion, the reader is referred to the appropriate technical specifications issued by AMD (AMD), (MIC).

III. 3.1 Sequence Controller

Figure 3.3 shows the block diagram of the AM2910 sequence controller. It operates according to signals received at the I input pins. During the execution of each microinstruction, the sequence controller presents the address of the next microinstruction to the control store. This address comes from one of four sources. The usual source is the microprogram counter register (UPC). In the graphics controller configuration, this register always contains an address one greater than the previous address. This provides for sequential access to the control store.

A second address source is the external input (D). For our purposes, this address ultimately comes from either the mapping prom output or from a part of the pipeline register. The sequence controller selects one of these two input sources by issuing either a mapping prom enable signal ($\overline{MAP}$) or a pipeline enable signal ($\overline{PL}$). If these output signals control tri-state output enables, for both input sources, then the D input pins can be driven directly, by both sources, without further contention logic. Note that the mapping prom and pipeline register must never both be enabled at the same time.

Input from the mapping prom signals the start of a new macro-execute cycle. It designates the start of the microprogram which executes the associated graphical instruction. Address input from the pipeline, on the other hand, affords a convenient branching capability. In this case, the microinstruction itself, contains the next microinstruction address.

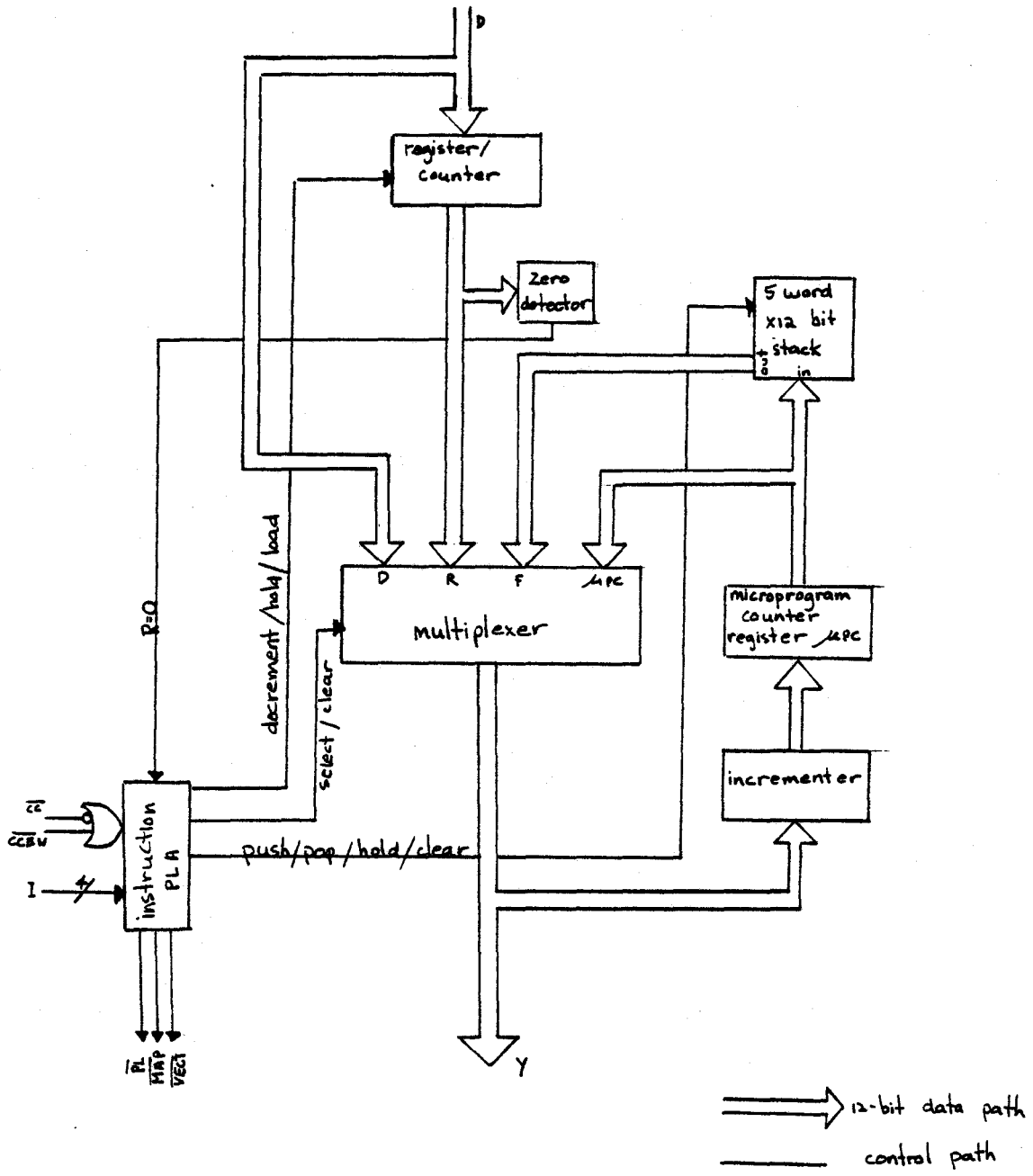


Fig. 3.3: The AM 2910 Microprogram Controller

HEX I_3-I_0	MNEMONIC	NAME	REG/ CNTR CON-TENTS	FAIL $\overline{CCEN}$ = LOW and $\overline{CC}$ = HIGH		PASS $\overline{CCEN}$ = HIGH or $\overline{CC}$ = LOW		REG/ CNTR	ENABLE
				Y	STACK	Y	STACK		
0	JZ	JUMP ZERO	X	0	CLEAR	0	CLEAR	HOLD	PL
1	CJS	COND JSB PL	X	PC	HOLD	D	PUSH	HOLD	PL
2	JMAP	JUMP MAP	X	D	HOLD	D	HOLD	HOLD	MAP
3	CJP	COND JUMP PL	X	PC	HOLD	D	HOLD	HOLD	PL
4	PUSH	PUSH/COND LD CNTR	X	PC	PUSH	PC	PUSH	Note 1	PL
5	JSRP	COND JSB R/PL	X	R	PUSH	D	PUSH	HOLD	PL
6	CJV	COND JUMP VECTOR	X	PC	HOLD	D	HOLD	HOLD	VECT
7	JRP	COND JUMP R/PL	X	R	HOLD	D	HOLD	HOLD	PL
8	RFCT	REPEAT LOOP, CNTR ≠ 0	≠ 0	F	HOLD	F	HOLD	DEC	PL
			= 0	PC	POP	PC	POP	HOLD	PL
9	RPCT	REPEAT PL, CNTR ≠ 0	≠ 0	D	HOLD	D	HOLD	DEC	PL
			= 0	PC	HOLD	PC	HOLD	HOLD	PL
A	CRTN	COND RTN	X	PC	HOLD	F	POP	HOLD	PL
B	CJPP	COND JUMP PL & POP	X	PC	HOLD	D	POP	HOLD	PL
C	LDCT	LD CNTR & CONTINUE	X	PC	HOLD	PC	HOLD	LOAD	PL
D	LOOP	TEST END LOOP	X	F	HOLD	PC	POP	HOLD	PL
E	CONT	CONTINUE	X	PC	HOLD	PC	HOLD	HOLD	PL
F	TWB	THREE-WAY BRANCH	≠ 0	F	HOLD	PC	POP	DEC	PL
			= 0	D	POP	PC	POP	HOLD	PL

Note 1: If $\overline{CCEN}$ = LOW and $\overline{CC}$ = HIGH, hold; else load. X = Don't Care

TABLE 3.1: AM2910 INSTRUCTION SET

A third address source is the register/counter (R). It is preloaded via the D input from the pipeline register. The five deep last-in first-out stack (F) is the fourth source. The stack provides for microsubroutine return linkages and looping capabilities. Microsubroutines may be nested up to five levels deep.

The register/counter R can also be made to act as a loop counter. It can be loaded from the D input and decremented each time through a loop. When its contents become equal to zero, the loop terminates. The arrangement is such that if it is preloaded with a number N and subsequently used as a loop termination counter, the loop will be executed N+1 times.

Table 3.1 presents the AM2910 instruction set. Many of these instructions are conditional. The input signal $\overline{CC}$ is used as the test criterion. A low signal corresponds to a pass. Further flexibility is provided by the $\overline{CCEN}$ input, which enables the conditional testing. When this signal is high, $\overline{CC}$ is ignored and the sequence controller operates as though the result of the condition test were a pass. The vector address enable signal ($\overline{VECT}$) allows a third external source to drive the D input. This capability is not utilized in the graphics controller design. As a result, the CJV instruction in table 3.1 is never used.

The microinstruction fields which control the operation of the AM2910 are shown in table 3.2 and are described below.

AM2910 Instruction - Controls operation of AM2910. Mnemonics correspond to those of table 3.1.

TABLE 3.2

CONTROL SEQUENCER FIELDS

AM 2910 Instruction 4 Bits	Condition Code Enable 1 Bit	Force Test Polarity 1 Bit	Condition Code 3 Bits		Pipeline Data 12 Bits
JZ	CD	NEG	C	Carry out	
CJS	UCD	POS	OVR	Overflow	
JMAP			Z	Zero	12 Bit
CJP			S	Sign	Data
PUSH			SO	Shift Out	Item
JSRP			IRF	IR Full	
CJV			DRF	DR Full	
JRP			MPF	Bit Map Free	
RFCT					
RPCT					
CRTN					
CJPP					
LDCT					
LOOP					
CONT					
TWB					

CD : conditional
UCD : unconditional
NEG : negative
POS : positive

Condition Code Enable - Provides the $\overline{CCEN}$ input signal. CCEN low corresponds to the CD (conditional) mnemonic.

Force Test Polarity - Provides the polarity select signal in figure 3.2.The input to the polarity control is either flipped (NEG) or passed to the AM2910 $\overline{CC}$ input unchanged (POS.).

Condition Code - Selects the condition code input from one of the eight condition code multiplexor inputs. The selected signal is subsequently passed to the polarity control.

Pipeline Data - The 12 bit pipeline data field which, if enabled, provides an external D input to the AM2910.

With the condition code multiplexor and polarity control shown in figure 3.2, the operation of the AM2910 can be made conditional on the status of the execution network. For example, a nonzero result in the ALU may lead to a branch in the microprogram. This would correspond to the following microinstruction field values.

AM2910 Instruction	Condition Code Enable	Force Test Polarity	Condition Code	Pipeline Data
CJP	CD	NEG	Z	Branch Address

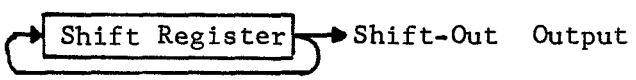
The meaning of the various possible condition code inputs will be explained more fully as the discussion of the execution network proceeds.

III. 3.2 ALU

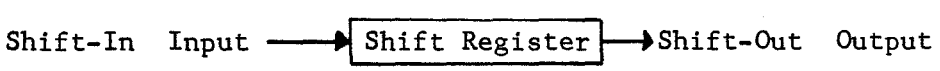
The ALU is fashioned from four AM2903 LSI chips. A principle reason for using these chips rather than the less expensive AM2901 chips, is that the internal register file of the AM2903 is easily expanded to virtually any required size. The AM2903 includes the necessary 'hooks' needed to accomplish this. On the other hand, the full range of capabilities available with the AM2903 has not been used in this design. In particular, the AM2903 has built-in floating point logic, which is not required for our purposes.

Figure 3.4 illustrates the functional capabilities of the proposed ALU. The diagram does not correspond to a specific component chip, but represents, instead, the complete ALU assembly. This assembly consists of the bit-sliced AM2903's, AM29751's and AM29705's as well as extra shift control logic. Specific chip assembly details may be found in the AMD literature (AMD).

The extra shift control logic specifies whether a shift operation is cyclic or linear. In any case, the actual shift operation is performed by the AM2903's. The control logic merely controls where the shift-in and shift-out bits come from and go to respectively. For example, in a cyclic shift, the shift-in bit is made equal to the shift-out bit, whereas, in a linear shift, the shift-in bit is driven from the external shift-in input as shown below.



Cyclic Right Shift



Right Shift (linear)

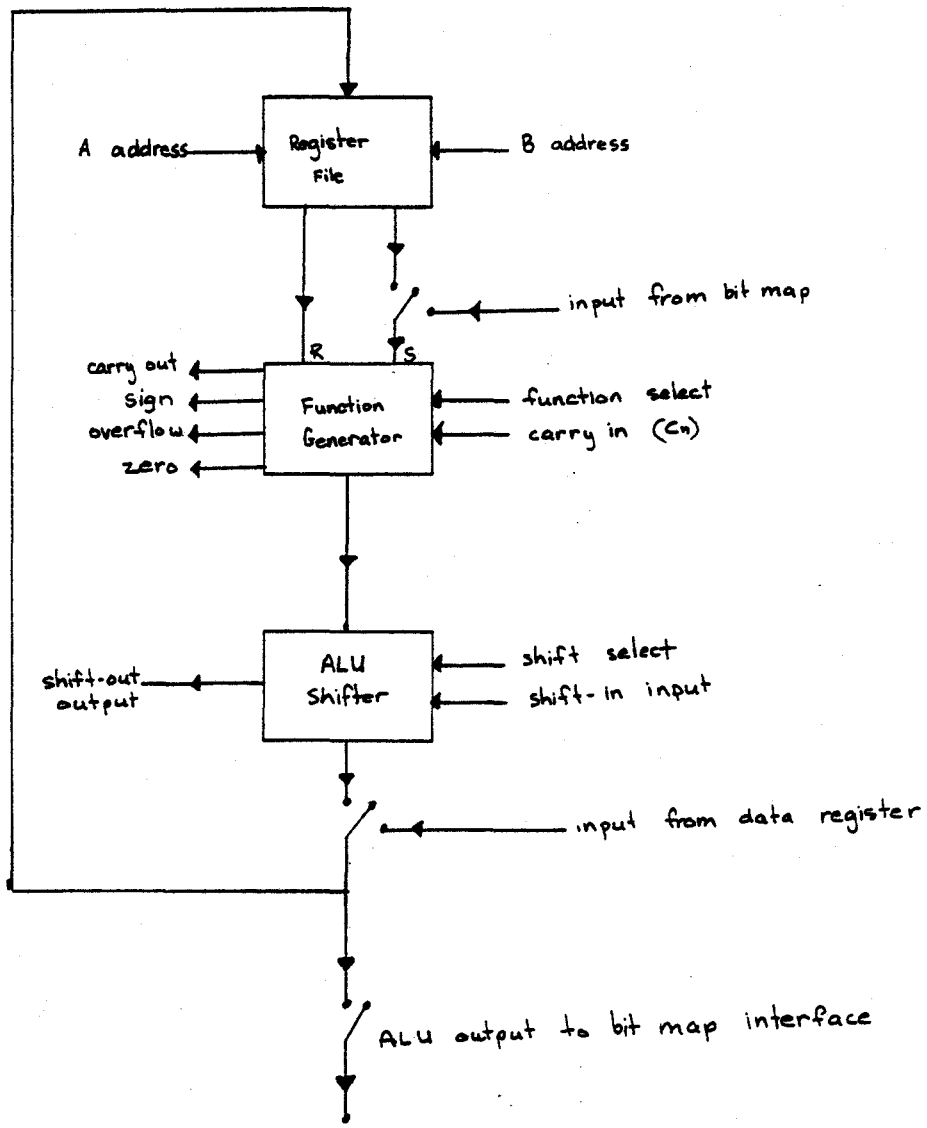


Fig. 3.4: The ALU

The operation of the arithmetic logic unit proceeds as follows. At the beginning of a microinstruction, the A and B addresses are presented to the ALU. As a result, two operands are sent to the function generator. Alternatively the B operand may come from the bit map. Next, an operation, as selected by the function select input, is performed. During the course of this operation, the carry out, sign, overflow and zero status flags are generated. The result is sent to the shifter, where, in like manner, the shifter operates on its input and as well, creates the shift-out output. After the result is stable, it may be written into the register file at the location specified by the B address or it may be sent to the bit map interface. Finally, the status register may be updated with the newly generated status flags. The microprogrammer has the option of inhibiting this update.

It is also possible for input to be accepted directly from the data register (fig. 3.2). In this case, the shifter output is ignored and the contents of the data register is written to the register file instead. Any operation to be performed on the data must wait until the next microinstruction. This situation is in contrast to input from the bit map, which, being presented directly to the function generator, can be read and operated on in the same microinstruction.

In the ALU's operation, the B address has a dual role. It is an operand address in the early part of the microinstruction cycle, and a result destination address in the latter. This constitutes what is known as a two address architecture. A three address architecture, whereby, separate destination and operand addresses are specified, is

also possible with AM2903 chips. However, if implemented, a three address configuration would involve more complex timing logic.

The preceding discussion enables further microinstruction fields to be defined. These are presented in table 3.3. Several of the ALU functions are expressed in terms of Cn, an input which is provided by the carry in field. For the shift control field, the only difference between a 'NOP' and a 'NS' command, is that a NOP does not cause the final result to be written into the register file. All other shift control instructions cause a final write operation. The NOP instruction is useful when the firmware designer wishes to idle the ALU without overwriting any of the data in the register file.

The enable status load field offers the firmware designer direct control over the updating of the status register. This feature can lead to a reduction in the number of microinstructions needed to accomplish a conditional operation. For example, consider the case of a microprogram 'jump' which is performed only if the value of a specific variable is zero. If the status register were automatically loaded at the end of every microinstruction, the variable in question would have to pass through the function generator during the clock cycle which immediately preceeds the execution of the conditional jump. In many instances, an extra microinstruction would be required for this purpose alone. If, on the other hand, the variable had been defined several microinstructions back, and the status load disabled ever since, then, the further microinstruction would not be required.

Notice that the input/output fields are classified as belonging to either the data bus or bit map interfaces. The interface

ALU Fields							Data Bus Interface	Bit Map Interface	
Function 4 bits	Carry In (Cn) 1 bit	Shift 3 bits	Shift-In Input 1 bit	A Addr. (R) 6 bits	B Addr. (S) 6 bits	Enable Status Load 1 bit	Data Reg. Input Enable 1 bit	Bit Map To ALU Enable 1 bit	ALU Output Enable 1 bit
HIGH	0	CRS	0			E	E	E	E
S-R-1+Cn	1	RS	1			DIS	DIS	DIS	DIS
R-S-1+Cn		NOP							
R+S+Cn		NS							
S+Cn		CLS							
$\overline{S}$+Cn		LS							
R+Cn									
$\overline{R}$+Cn									
LOW									
$\overline{R} \wedge S$									
$\overline{R \not\vee S}$									
$R \not\vee S$									
$R \wedge S$									
$\overline{R \vee S}$									
$\overline{R \wedge S}$									
$R \vee S$									

CRS : cyclic right shift
RS : right shift
NOP : no operation
NS : no shift
CLS : cyclic left shift
LS : left shift
E : enable
DIS : disable

TABLE 3.3 : Further microinstruction fields

fields are used rather infrequently in comparison to those of the ALU and sequence controller. Excepting microinstructions which perform I/O operations, the contents of these fields do not vary from one microinstruction to the next.

III. 3.3 Data Bus Interface

Figure 3.5 shows the data bus interface in more detail. The IR full and DR full, flags are used to establish a 'handshaking' input/output protocol. Using this protocol, the transfer of data from the main microprocessor to the graphics controller, proceeds under program control. For example, consider for the moment, the data path to IR. Before the main microprocessor can deposit a graphics instruction into the IR, it must wait for the graphics controller to clear the IR full flag. This flag is cleared immediately after each new macroinstruction is 'accepted' by the graphics controller. A macroinstruction is accepted by selecting the mapping from output as the next microinstruction address. Similarly, before the graphics controller can accept a new graphics instruction, it must wait for the main microprocessor to set the IR full flag. This flag is set immediately after each new macroinstruction is loaded, by the main microprocessor, into the IR. This protocol protects the integrity of the data transfer. In like manner, the DR full flag is used to regulate the transfer of data to the graphics controller via the data register.

It is clear that in order to establish the I/O protocol properly, provision must be made for microinstructions which are conditional to the value of either of the two I/O flags. Accordingly, both flags are used as inputs to the condition code multiplexor. The

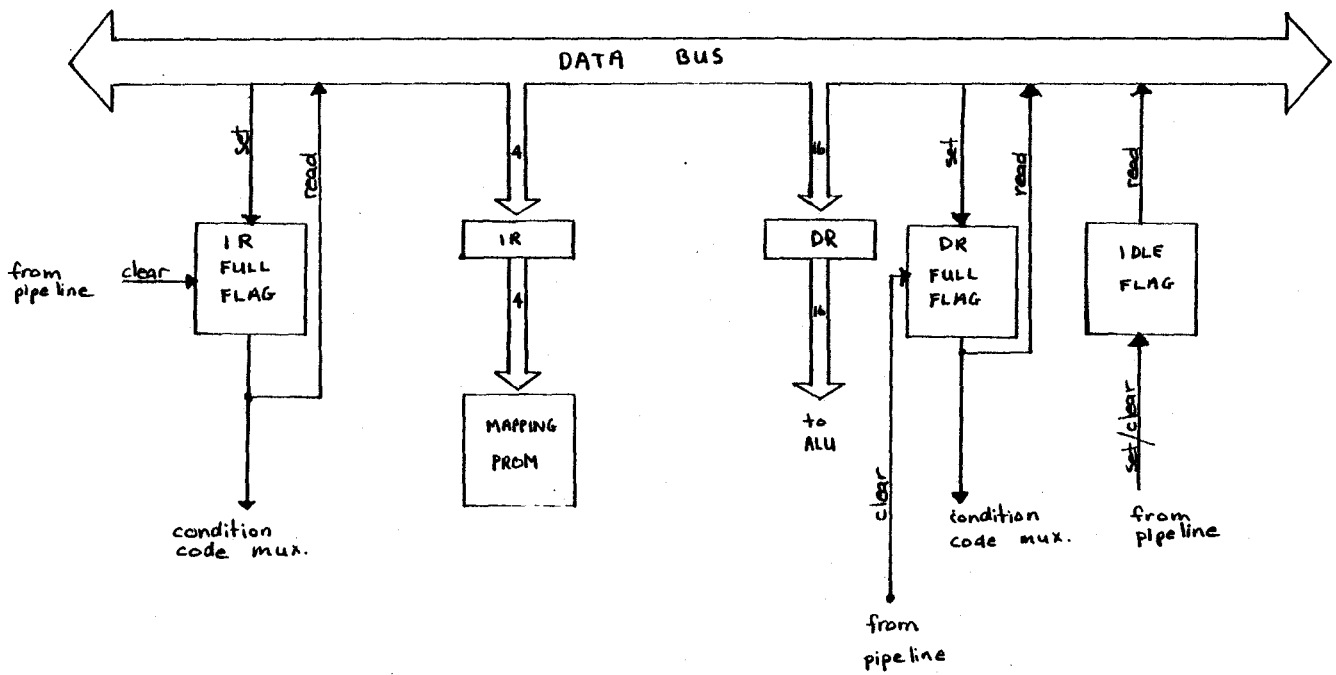


Fig. 3.5 : The Data Bus Interface

TABLE 3.4

DATA BUS INTERFACE MICROINSTRUCTION FIELDS

IR Full Flag	DR Full Flag	Idle Flag	Data Register Input Enable
1 Bit	1 Bit	1 Bit	1 Bit
CLR	CLR	IDLE	E
NCL	NCL	BSY	DIS

CLR = Clear IR Full Flag

NCL = No Clear

E = Enable

DIS = Disable

BSY = Busy

set conditions correspond to the 'DRF' and 'IRF' condition code field values shown in table 3.2

Another feature of figure 3.5 is the idle flag. This flag is set by the graphics controller whenever it is in an idle state, waiting for further graphics instructions. During this state, the bit map is not accessed by the graphics controller. As discussed in the preceding chapter, the main microprocessor must first test the idle flag before addressing the bit map directly.

The complete set of data bus interface microinstruction fields are shown in table 3.4. Except for I/O operations, these fields have the following fixed values.

IR Full Flag	DR Full Flag	Idle Flag	Data Register Input Enable
NCL	NCL	BSY	DIS

III. 3.4 Bit Map Interface

The final microinstruction fields to be defined are those of the bit map interface. Figure 3.6 shows the bit map interface in more detail. Note that 2^{20} bits may be stored in the 64K by 16 bit word RAM. This is large enough to support a screen resolution of 1024 by 1024 dots. The bit map interface fields are shown in table 3.5 and are defined below.

ALU Output Enable - The enable signal causes data to be sent from the ALU to the bit map interface.

Enable Register Load - The enable signal causes output from the ALU to be latched into one of the two destination registers. This destination is specified by the field which is defined next.

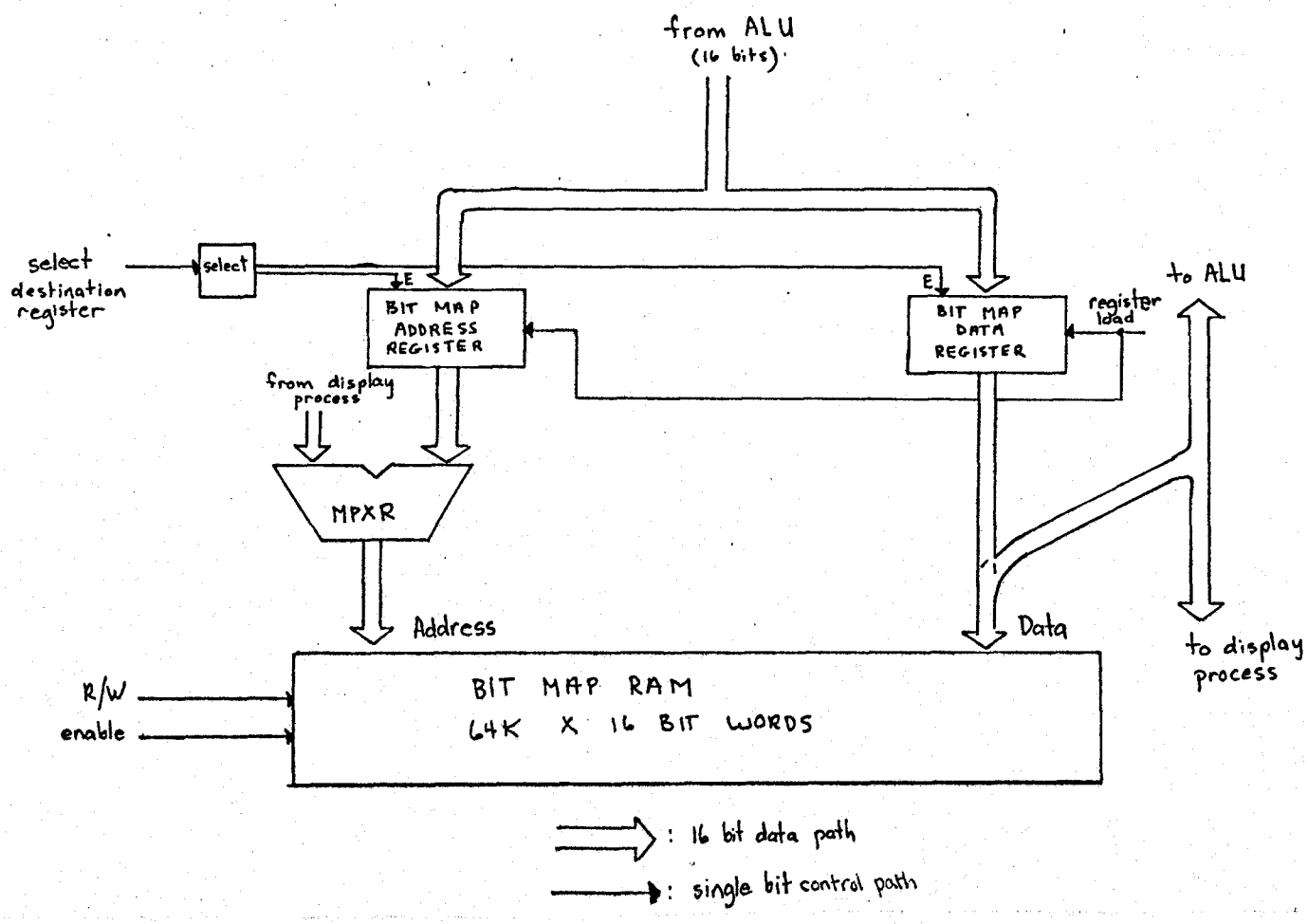


Fig. 3.6 : The Bit Map Interface

TABLE 3.5

BIT MAP INTERFACE MICROINSTRUCTION FIELDS

ALU Output Enable	Enable Register Load	Bit Map Address/Data	RAM Enable	RAM Read/Write	Bit Map to ALU Enable
1 Bit	1 Bit	1 Bit	1 Bit	1 Bit	1 Bit
E	E	ADD	E	R	E
DIS	DIS	DAT	DIS	W	DIS

D = Disable

E = Enable

ADD = Address register

DAT = Data register

R = Read

W = Write

Bit Map address or data - This field specifies the destination of output from the ALU as being either the bit map data register or the bit map address register. When the graphics controller performs bit map read/write operations, these registers drive the corresponding address and data lines of the bit map RAM. It is important to note that these registers are local to the graphics controller. The graphics display process uses its own separate address and data registers.

RAM Enable - Enables or disables the RAM read/write operations.

RAM Read/Write - Specifies whether a read or a write operation is to be performed by the bit map RAM. Note that depending upon the particular read/write access times of the RAM chosen for the bit map, read/write operations may require several microinstruction cycles to complete. In this case, RAM enable and read/write signals must be held fixed over several microinstructions.

Bit Map to ALU Enable - The enable signal causes bit map output available at the RAM data lines to be accepted as input to the ALU.

Except for I/O operations, the bit map interface fields have the following default values.

ALU Output Enable	Enable Register Load	Bit Map Address/Data	RAM Enable	RAM Read/Write	Bit Map to ALU Enable
X	DIS	X	DIS	X	DIS

where X = don't care

The bit map interface also provides a condition code multiplexor input. This input corresponds to the 'MPF' (Bit Map Free) condition code field value found in table 3.2. As discussed in the preceding chapter, the graphics display process and the graphics controller compete for access to the bit map memory. The graphic display process has highest priority. It is responsible for raising a bit map free flag whenever the graphics controller is to be allowed access to the RAM. This flag provides the corresponding condition code multiplexor input. Its value must be checked by the graphics controller before every bit map access.

The complete set of microinstruction fields has now been presented. It corresponds to a microinstruction which is 53 bits wide. Now, it is appropriate to turn our attention to the firmware control of the graphics controller. In the next chapter, a basic graphics instruction set is presented, along with several algorithms which enable its efficient realisation.

CHAPTER 4

THE GRAPHICS CONTROLLER FIRMWARE

A basic graphics instruction set for the graphics controller has been defined. This set consists of two types of instructions. The first type shall be referred to as the definition instructions. They are concerned with assigning values to so-called definition variables. These variables define the 'manner' in which curves are 'drawn'. For example, depending on their values, curves may be either solid or broken, erased or made visible. In contrast, the drawing instructions cause the actual drawing to occur. They, alone, cause a change in the contents of the bit map. There are two instructions of this type, VECTOR and CIRCLE, which generate line and circle segments respectively.

At the beginning of a graphics session, all the definition variables must be assigned values. Only then can a circle or vector instruction be executed properly. Once these variables have been initialized, drawing instructions may be executed in succession, one after the other. Alternatively, at any time, one or more definition instructions may be executed in between any two drawing instructions. This would have the effect of changing the drawing 'manner' from one curve segment to the next.

The variables, which are discussed in this chapter, correspond to 16 bit quantities. Two's complement arithmetic is used for the representation of negative values. Each variable can be classified as belonging to one of only three possible data types. There is a bit

mask data type, an integer data type, and a coded information data type.

IV. 1 Definition Variables

The definition variables are defined below. The first three definitions involve the assumption of a fully interlaced display (see Chapter 2).

EVODD: Coded data type. Specifies whether the current screen position corresponds to the even or odd raster bit map; 0: even; 1: odd.

POSITN: Integer data type. Represents the relative bit map address which corresponds to the current screen position. The absolute memory address is equal to POSITN + the first word address of the bit map specified by EVODD.

BTMSK: Bit mask data type. A single bit bit mask which, together with EVODD and POSITN, identifies the bit in a memory word which corresponds to the current screen position.

MODE: Coded data type. It specifies the drawing mode; 1: visible mode; 0: erase mode; -1: complement mode.

PATERN: Bit mask data type. It specifies the current dot pattern; one-valued bits correspond to screen dots which are, depending on the mode, made visible, erased or complemented. Zero-valued bits correspond to screen dots which are left unchanged.

PSCALE: Integer data type. It is a scaling factor which is applied to PATERN. Each bit in PATERN is made to correspond to PSCALE dots; PSCALE $>$ 0.

PATPOS: Bit mask data type. A single bit bit mask which specifies the current position in PATERN.

SCLPOS: Integer data type. Specifies the number of consecutive screen dot moves before the next bit in PATERN is in effect; SCLPOS > 0.

The first three variables defined above, shall be referred to as the position variables. Together, they define the bit in memory which corresponds to the current screen position. The concept of a current screen position is an important one. It implies that for each curve segment, there is both a starting, and a finishing endpoint. The current screen position is the starting endpoint from which subsequent drawing begins.

The position variables are continually updated during the execution of the drawing instructions. Upon termination of a drawing instruction, the position variables are left with values which represent the finishing position. This means that if two successive drawing instructions are executed, the two resultant curve segments will share an endpoint in common. Alternatively, a position instruction may be used to redefine the current screen position before the second segment is drawn. Thus, disjoint segments can be specified as well.

The mode variable, once set, remains unchanged unless it is explicitly reset. There are three drawing modes. The visible mode causes ones to be written into the bit map. These are translated by the graphics display process, into visible screen dots. As well, curves may be erased by specifying the erase mode, which causes zeros to be deposited into the bit map. The complement mode changes the bit map contents by complementing memory bits. This mode is of use in the displaying of temporary or dynamic images. In particular, with moving

images, there is the problem of blank gaps being left behind wherever previous images once intersected the static background display. This problem can be avoided through proper use of the complement mode (DIC).

The last group of definition variables provides for dotted curve segments. PATERN and PSCALE define the dotted pattern, whereas, PATPOS and SCLPOS define the position within the pattern. As curve segments are generated, the pattern position is cycled through the pattern. By way of understanding the meaning of the pattern variables more precisely, consider the process by which curve segments are generated.

Drawing instructions can be thought of as causing a representative sequence of moves from one screen dot to the next. Starting with the current screen position, each move is between contiguous dots, until the process terminates at the final predesignated position. Each move consists of selecting the next screen dot and making the appropriate bit map modifications. If the current pattern position corresponds to a zero-valued bit in PATERN, the selected screen dot is left unchanged. Otherwise, it is changed according to the dictates of the current drawing mode.

After each move, the current pattern position is updated. This update is accomplished by decrementing SCLPOS which acts as a scaling counter. Upon reaching zero, SCLPOS is automatically reset to PSCALE and a one bit cyclic shift operation is performed on PATPOS. In this manner, the dot pattern is repeatedly cycled through. As an example, a dotted line of alternately 30 visible dots and 10 blank dots may be specified by initializing PATERN to the binary string

'1110 1110 1110 1110' and by assigning PSCALE the value ten. PATPOS may be assigned any allowable value (single bit bit mask) depending upon whether the dotted line is to start on a blank or a visible line segment. Similarly, the initial value for SCLPOS determines the ultimate length of the starting line segment.

Once set, the actual dot pattern remains fixed, unless it is explicitly changed through the use of a definition instruction. The position within the pattern, however, is continually updated during the execution of drawing instructions. From this point of view, PATPOS and SCLPOS are similar to the position variables, whereas, PATERN, PSCALE and MODE form a second, more static category.

It should be pointed out that although the normal range of values for SCLPOS is between zero and PSCALE, the user is allowed the option of specifying an initial value for SCLPOS which is greater than PSCALE. This has the effect of scaling the starting bit in PATERN by a factor greater than PSCALE. For example, a dotted line could be generated, such that the first solid segment is longer than subsequent segments. This feature could be used to ensure that at both endpoints of a dotted line, there are solid segments.

IV. 2 Definition Instructions

Definition instructions translate incoming data into appropriate values of associated definition variables. These values are subsequently stored in the graphics controller register file. The data are in the form of 16 bit quantities which are sent from the main microprocessor to the graphics controller via the data register (figure 3.2). In these terms, the definition instructions are:

1. POSITION

   Data: PX, PY

   Define: EVODD, POSITN, BTMSK

2. MODE

   Data: MODE

   Define: MODE

3. PATTERN

   Data: PATERN, PSCALE

   Define: PATERN, PSCALE

4. PATTERN POSITION

   Data: PATPOS, SCLPOS

   Define: PATPOS, SCLPOS

Apart from the POSITION instruction, which is more complex, the above instructions simply involve writing received data directly into the ALU internal register file.

The POSITION instruction involves an actual translation. The main microprocessor specifies defineable screen positions with discrete x, y coordinate values. The corresponding coordinate system shall be referred to as the screen coordinate system. Its origin lies on the bottom leftmost screen dot. The unit distance is the distance between two adjacent horizontal, or two adjacent vertical, dots. The POSITION instruction must translate the discrete coordinate values, PX and PY, into their corresponding EVODD, POSITN and BTMSK values.

This translation is in terms of parameters which define the bit map:

GPHEVN: The first word address of the even raster bit map.

GPHODD: The first word address of the odd raster bit map.

WIDTH: The number of memory words which constitute a single raster line.

LENGTH: The number of raster lines per screen.

These parameters are stored in the ROM part of the internal register file. The POSITION instruction makes use of the following equations. A 16 bit memory word is assumed:

$$EVODD = MOD_2 \ (LENGTH - (PY + 1))$$

$$POSITN = \frac{LENGTH - (PY + 1)}{2} \ X \ WIDTH + PX/16$$

$$BTMSK = 2^{(15-MOD_{16} \ (PX))}$$

The division operations, in the above, are integer divisions. The quotient is truncated to an integral value. Since the divisors, in each case, are powers of two, the divisions correspond to simple binary shift operations. As well, the modular operations are with respect to bases, which are also powers of two. As a result, each modular operation simply involves the appropriate discarding of high order bits. The single multiplication is accomplished through a sequence of binary shift and add operations.

IV. 3 Drawing Instructions

Like the definition instructions, the drawing instructions also require data which are deposited, by the main microprocessor, into the data register. The received data are used to control the 'drawing' of curve segments on the display screen. The two drawing instructions are:

1. VECTOR

   Data: VX, VY

   Draw: Line Segment

2. CIRCLE

Data: X, Y, DOCT, FCORD

Draw: Circle Segment

A vector instruction causes a line segment to be drawn, from the current screen position, to the end point designated by VX and VY. These variables are defined below. Displacements are measured in the units of the screen coordinate system.

VX: Integer data type. It specifies the horizontal displacement, of the finishing endpoint, from the current screen position. VX may be either positive or negative.

VY: Integer data type. It specifies the vertical displacement, of the finishing end point, from the current screen position. VY may be either positive or negative.

A circle instruction causes an arc to be drawn, from the current screen position, to the endpoint specified by DOCT and FCORD. Points which lie exactly on the 'true' arc, are equidistant from a unique position which is referred to as the arc center. This position may lie outside screen boundaries. Its whereabouts are specified by X and Y. The CIRCLE variables are defined below. Again, distances are measured in the units of the screen coordinate system.

X: Integer data type. It specifies the horizontal displacement, of the arc center, from the current screen position. X may be either positive or negative.

Y: Integer data type. It specifies the vertical displacement, of the arc center, from the current screen position. Y may be either positive or negative.

DOCT: Composite data type. The most significant bit is a code which specifies either a clockwise, or a counter clockwise, drawing direction; 0: counter clockwise; 1: clockwise. The remaining bits correspond to an integer data type. They specify the number of octant changes (see following discussion) required to complete the arc segment.

FCORD: Integer data type. Imagine two lines which pass through the arc center. One line is vertical and the other, horizontal. FCORD is the shortest distance between the final arc endpoint and the nearer of these lines; $0 \leq \text{FCORD} \leq \sqrt{(X^2 + Y^2)}/2$ where the upper limit is rounded down.

The position of the final arc segment endpoint can be deduced from the specified values of FCORD and DOCT. For this purpose, circles are imagined to be centered on a coordinate system, as shown in figure 4.1. The coordinate system is divided into eight equivalent sections, called octants. The horizontal and vertical octant boundaries are called square octant boundaries. Each octant has exactly one square and one diagonal octant boundary.

Imagine that the graphics controller has been instructed to draw a circle segment. Starting at the current screen position, the drawing proceeds in a clockwise or anti-clockwise fashion, depending on the value of the most significant bit of DOCT. Every 45 degrees, an octant boundary is crossed. This crossing from one octant into

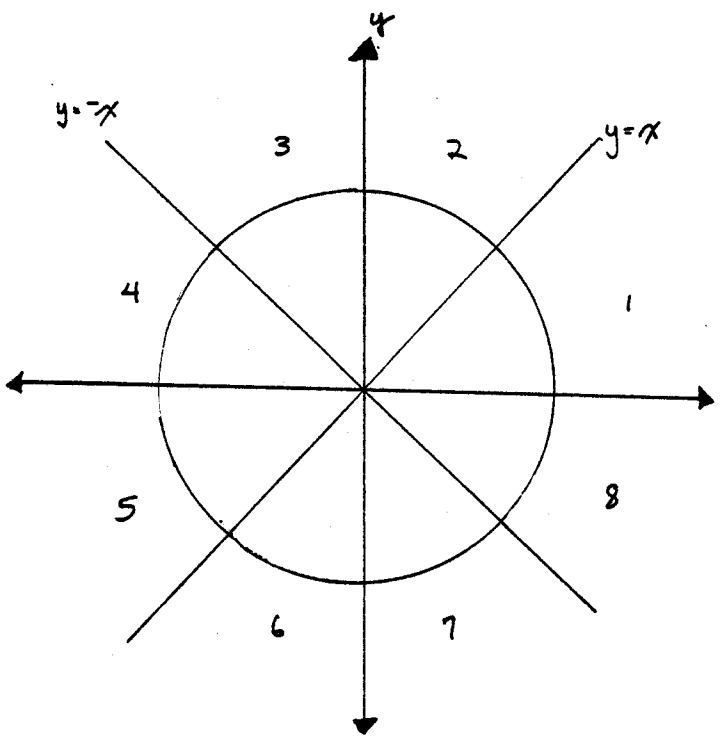


Fig. 4.1: The division of a circle into eight octants.

another, is called an octant change. DOCT specifies how many of these are required to complete the arc segment. After the required number of octant changes, the drawing terminates when the distance from the current octant's square octant boundary, is exactly FCORD. In this manner, the final arc segment endpoint is uniquely specified.

Note that an octant change occurs only when an octant boundary is actually crossed. As figure 4.2 illustrates, care must be taken in the assigning of values to DOCT, whenever an arc begins or ends on an octant boundary.

It should be pointed out that in order for graphics instructions to execute properly, they must be provided with 'correct' data. No testing is done by the controller firmware to ensure that incoming data is meaningful. For example, the microprogram, which corresponds to the CIRCLE instruction, does not test whether FCORD is within the required range. Moreover, all incoming data must be such that each specified curve segment lies entirely on the display screen. If this is not the case, unpredictable results may occur. For these reasons, there is a level of program control between the user and the graphics instruction set. This level of control resides in the main microprocessor.

IV. 4 Algorithms

Thus far, in this chapter, the graphics instruction set has been discussed in detail. In the remaining part of the chapter, algorithms, which were used to realize the VECTOR and CIRCLE instructions are presented.

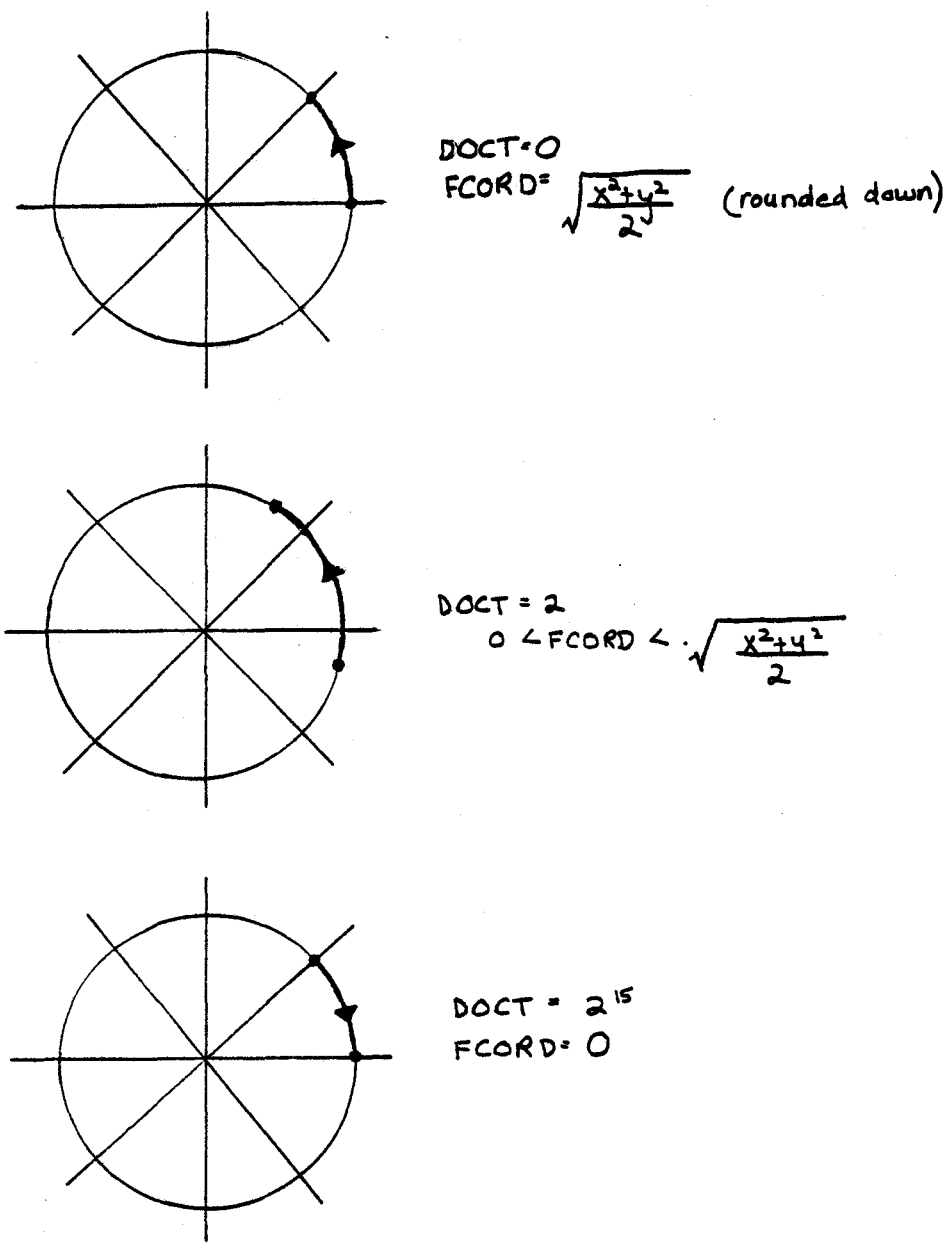


Fig. 4.2: FCORD and DOCT specify the final arc segment endpoint.

The algorithms were implemented with the graphics controller microinstruction set. Although the microcoded firmware is not presented as a formal part of this report, the actual lengths of critical sections will be used to provide an indication of the graphics controller performance capabilities.

The realization of the VECTOR and CIRCLE instruction was accomplished in two steps. First, the algorithms were implemented in the high level programming language, IFTRAN. Secondly, they were translated into microcode. The complete IFTRAN version of the circle and line generating algorithms, is to be found in Appendix A. In the discussion of algorithms, which follows, the reader is referred to Appendix A for greater detail. The IFTRAN package was tested and debugged on a CDC 6400 computer. For testing purposes, an on-line Versatec plotter was used in place of the display screen.

Every attempt was made to render the final translation, from IFTRAN to microcode, a trivial one. For example, the same modular divisions were made in both the IFTRAN and the microcode. Almost every IFTRAN routine corresponds to a similar routine in the microcode. The only exceptions are a group of IFTRAN subroutines, which are used solely for testing purposes. In these routines, a Versatec library is used to generate curves which serve as a basis of comparison for the circle and line generating algorithms being tested.

The IFTRAN and microcode are similar in other ways as well. For the most part, variables and coding structures are the same in both packages. There are, however, some important differences.

Most of these differences occur in the input/output routines. In particular, where the IFTRAN version sends information to the Versatec, the actual firmware makes corresponding changes to the bit map. By way of illustrating how the translation from IFTRAN to microcode was achieved, a sample microcoded routine is included in this report in Appendix B.

The line and circle generating algorithms are now presented. These algorithms fulfill certain basic requirements. They generate representative sets of contiguous dots, they are efficient, and they involve only simple binary arithmetic and logical operations. Much of the following development of these algorithms is owing to K.P. Horn (HOR). The line generating algorithm will be presented first.

IV. 4.1 Line Generator

Imagine a coordinate system to be centered on the current screen position. A line segment is to be drawn from the origin to the discrete screen position VX, VY. The equation of the corresponding line is just (VX)Y = (VY)X. Consider for the time being a line which lies in the first octant of the coordinate system. In other words, VX and VY are both greater than or equal to zero, and VX is greater than or equal to VY.

The screen can be thought of as a mesh of defineable screen dots. The solution set is the set of contiguous dots which best represent the line segment. For each column of dots, which intersects the line segment, there is a dot which lies immediately above the line, and a dot which lies immediately below the line. For each column, the closest of these two dots is to be included in the solution set.

Clearly, the closest dot is at a vertical distance from the line, of no more than 1/2 of a mesh unit. In contrast, the further dot is vertically removed from the line, by a distance which is at least 1/2 of a mesh unit and probably more. As a result, the solution set is bounded by the two lines, VX(Y-1/2) = (VY)X and VX(Y+1/2) = (VY)X, which lie on either side of, and are parallel to, the line segment which is to be generated.

There are also two vertical limits to the solution set. Namely, the lines X = 0 and X = VX. These limits, together with the previous two, define a bounded area. All dots within the bounded area are members of the solution set. Each column of dots, which intersects the bounded area, contains at most one dot within the band defined by the upper and lower parallel limits. Furthermore, in most instances, there is exactly one such dot. The exception occurs whenever the dot just below the prospective line segment, is as far from the line as the nearest one above it. In this situation, one dot lies on the upper boundary, while the other dot lies on the lower. There are no dots which lie entirely within the two limits. For the time being, the lower dot, alone, will be arbitrarily picked as belonging to the solution set.

The line-generating algorithm selects dots, in sequence, through incremental calculations. Essentially, the result of a test dictates whether the next dot chosen, is horizontally or vertically removed from its predecessor. Let the coordinates of the dot which was last selected, be X, Y. The next dot is chosen by testing whether the dot one column over (X+1, Y), falls below the lower limit.

If it does, the dot one column over and one row up (X+1, Y+1), must be the solution set member for that column. Otherwise, the dot at X+1, Y is selected. This process continues until X = VX.

The critical test amounts to whether or not D = 2 (VY(X+1) - VX(Y+1/2))>0. The 'greater than' test ensures that for equally good dots, the lower dot is selected. A 'greater than or equal to' test would result in the upper dot being selected. Since only fixed-point additions and subtractions are allowed, the factor of two is used to eliminate the fraction, 1/2. D is given an initial value of 2 VY-VX and incremented after every move. After each horizontal move, it is incremented by 2 VY, and after each diagonal move, by -2(VX-VY). It is easily seen that $-2(VX-Y)+1 \leq D \leq 2VY$. This means that only one more bit is required to store D, than is required for VX and VY.

The line generator is easily extended to include the general case. For example, the following variables may be used to specify motion in any octant. Once they are initialized correctly, any line-generating problem can be mapped into its equivalent, in the first octant.

INSTRX: Coded data type. Specifies direction of horizontal moves; - 1: decrement; 1: increment.

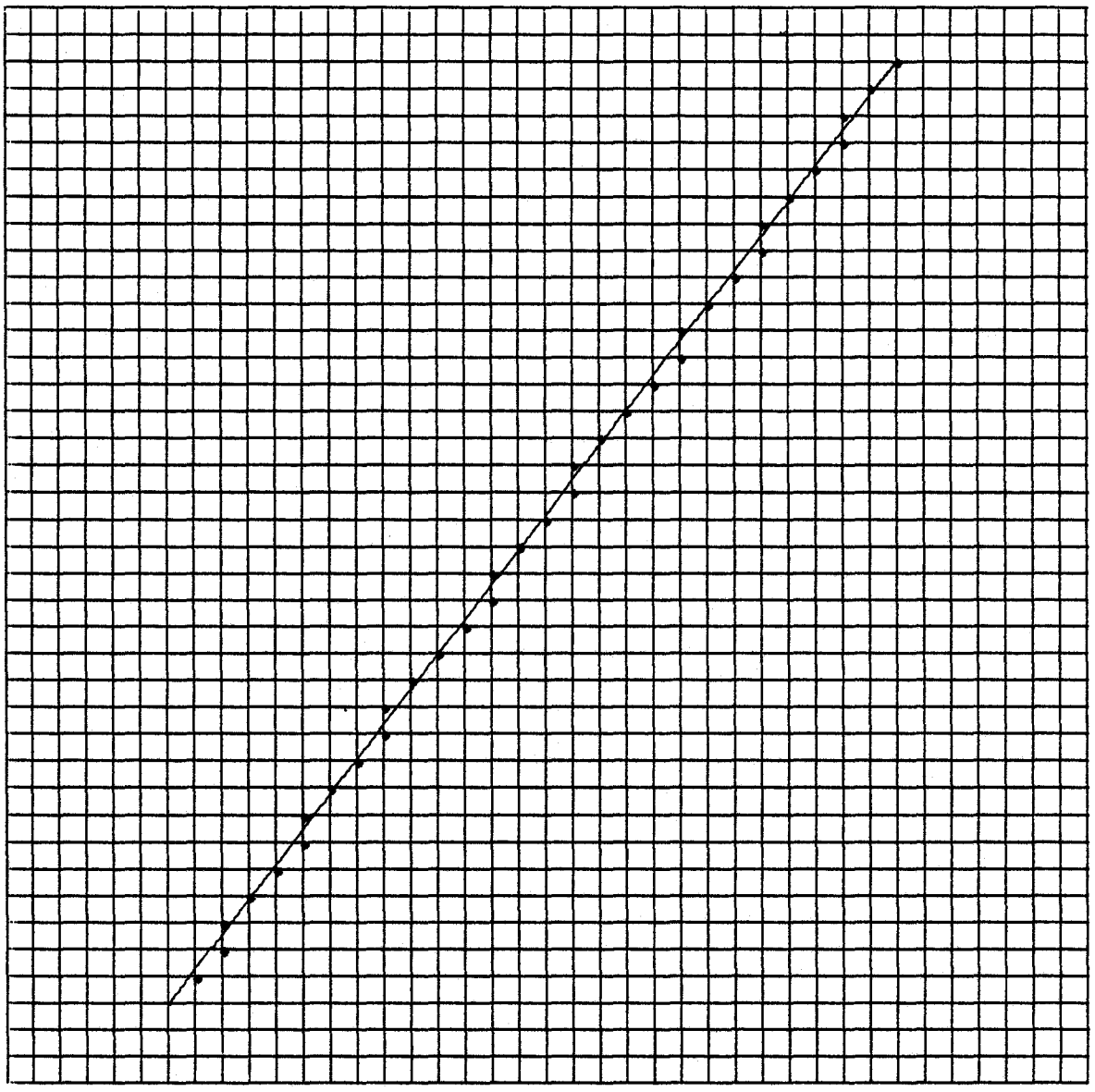
INSTRY: Coded data type. Specifies direction of vertical move; -1: decrement; 1: increment.

MOVEI: Coded data type. Specifies the coordinate which undergoes the greatest change; 1: vertical; 2: horizontal.

The above variables shall be referred to as defining the actual motion, whereas, VX and VY define the equivalent motion, As an example, consider a line which is to be drawn in the sixth octant. INSTRX and INSTRY are given 'decrement' values and MOVEI is made to designate the vertical coordinate. Finally, three transformations map the problem into the first octant. Reflections about the y and x axes are accomplished by setting VX = -VX and VY = -VY. A third reflection about the line y = x, involves swapping the values of VX and VY. The previous developed procedure is now used to generate the line. Only this time, the 'horizontal' move corresponds to a change in the value of the coordinate specified by MOVEI. Each 'horizontal' move is actually a downward vertical move. Similarly, the 'diagonal' move is now downward and towards the left.

There is a final, rather subtle point, which concerns the line generator. Consider a line segment which has been drawn in an upward direction. If this line segment is subsequently erased in a downward direction, the same dots must be made blank, as were once made visible. Most dot columns contain only a single 'best' dot, and as a result, there isn't a problem. Care must be taken, however, with columns which contain equally good dots. Equally good dots are to be found, for instance, at the center of line segments for which the greater coordinate displacement is even, and the lesser, odd.

The following example illustrates the problem of equally good dots. Consider a line which was originally drawn upwards into the first octant. Upon subsequent erasure, it is erased downwards as a line into the fifth octant. As discussed earlier, the line is erased



PX = 6
PY = 3
VX = 27
VY = 35

Fig. 4.3: An example of points selected by the line generator.

by effectively mapping the problem into its equivalent in the first octant. Unfortunately, after this mapping, what was once up, is now down. More specifically, the lower of two equally good dots, is now effectively on top. As a result, a different dot is erased than was originally made visible. In general, for any line which lies in an octant below the x axis, the dot selection test should be for $D \geq 0$ rather than for $D > 0$. This ensures that for equally good dots, the actual dot selected in drawing downwards is the same dot which is chosen in drawing upwards.

This solution can be implemented in a manner which involves no extra testing inside the dot-select loop. If D is incremented by one, the effect is of changing the $D > 0$ test to a $D+1 > 0$, or a $D > -1$, test. Since D is an integer, this is equivalent to the test, $D \geq 0$, which is, of course, exactly the desired result. Thus, for lines which fall in one of the lower octants, D is initially incremented by one. This is done only once, and before the dot-select loop is entered.

IV. 4.2 Circle Generator

The circle generator can be developed by analogy with the line generator. This time, the discrete x-y coordinate system lies in the center of the circle. The equation of the circle is $X^2 + Y^2 = R^2$, where R is the radius of the circle. The starting position is, by definition, a point on the circle. Consider, for the time being, a circle segment which lies entirely within the first octant, and is drawn counter-clockwise.

For each row of dots, which intersects the circle segment, there is a dot which lies immediately to the right of the arc, and a

dot which lies immediately to the left of the arc. In fact, much like before, the solution set is bounded by the two circles, $(X + 1/2)^2 + Y^2 = R^2$ and $(X - 1/2)^2 + Y^2 = R^2$. There are two horizontal limits to the solution set, which simply correspond to the initial and final (FCORD) values of Y. These four limits define a bounded area, within which all dots are members of the solution set. For each row of dots, which intersects the bounded area, there is but one dot within the band defined by the inner and outer arc limits.

The circle generator also selects dots, in sequence, through incremental calculations. Let the coordinates of the dot, which was last selected, be X, Y. The next dot is chosen by testing whether the dot one row up (X, Y + 1), falls outside the outer arc limit. If it does, the dot one row up and one column to the left (X - 1, Y + 1), must be the solution set member for that row. Otherwise, the dot at X, Y + 1 is selected. This process continues until Y = FCORD.

The critical test amounts to whether $S^+ = (X - 1/2)^2 + (Y + 1)^2 - R^2 + 3/4 > 0$. The fraction, 3/4, simply rounds S^+ up to the nearest integral value. By rounding up, rather than down, we ensure that none of the test results are changed. The superscript indicates a positive, or counter-clockwise, direction. Note that for circles, there is never a choice between two equally good dots. This is proven by the fact that $(X - 1/2)^2 + (Y + 1)^2 - R^2$ is never equal to zero.

The value of S^+ is updated as before. Since the starting position satisfies the equation $X^2 + Y^2 = R^2$, S^+ reduces to an initial value of 2Y - X + 2. After each vertical move, it is incremented by 2Y + 3, and after each diagonal move, by 2Y - 2X + 5.

Note that $-2(X - Y) + 6 \leq S^+ \leq 2Y + 3$. This means that only one more bit is required to store S^+, than is required for X and Y. Allowing for this extra bit plus an additional sign bit, radii of up to 2^{14} mesh units in length, may be specified.

If the arc is to be drawn in a clockwise direction, the choice is between moving one row down, and one row down and to the right. The test involves whether or not the inner, rather than the outer, arc boundary has been crossed. If $S^- = R^2 - (X + 1/2)^2 - (Y - 1)^2 + 1/4 > 0$, the dot at position X + 1, Y - 1 is selected, otherwise the dot at X + 1, Y is the correct choice. Again the fraction, 1/4, simply rounds S^- up to the nearest integral value. S^- has an initial value of $2Y - X - 1$. After each vertical move, it is incremented by 2Y - 3, and after each diagonal move, by $2Y - 2X - 5$. The limits for S^- are similar to those for S^+: $-2(X - Y) - 4 \leq S^- \leq 2Y - 3$.

We have considered the case of a circle segment which lies entirely within the first octant. Now, the question of how to generate an arc of arbitrary length, is addressed. For the time being, the arc must still begin in the first octant. Specifically, what happens when an arc which is being drawn in a clockwise direction runs into the diagonal octant boundary? The answer is what shall be termed a diagonal octant change.

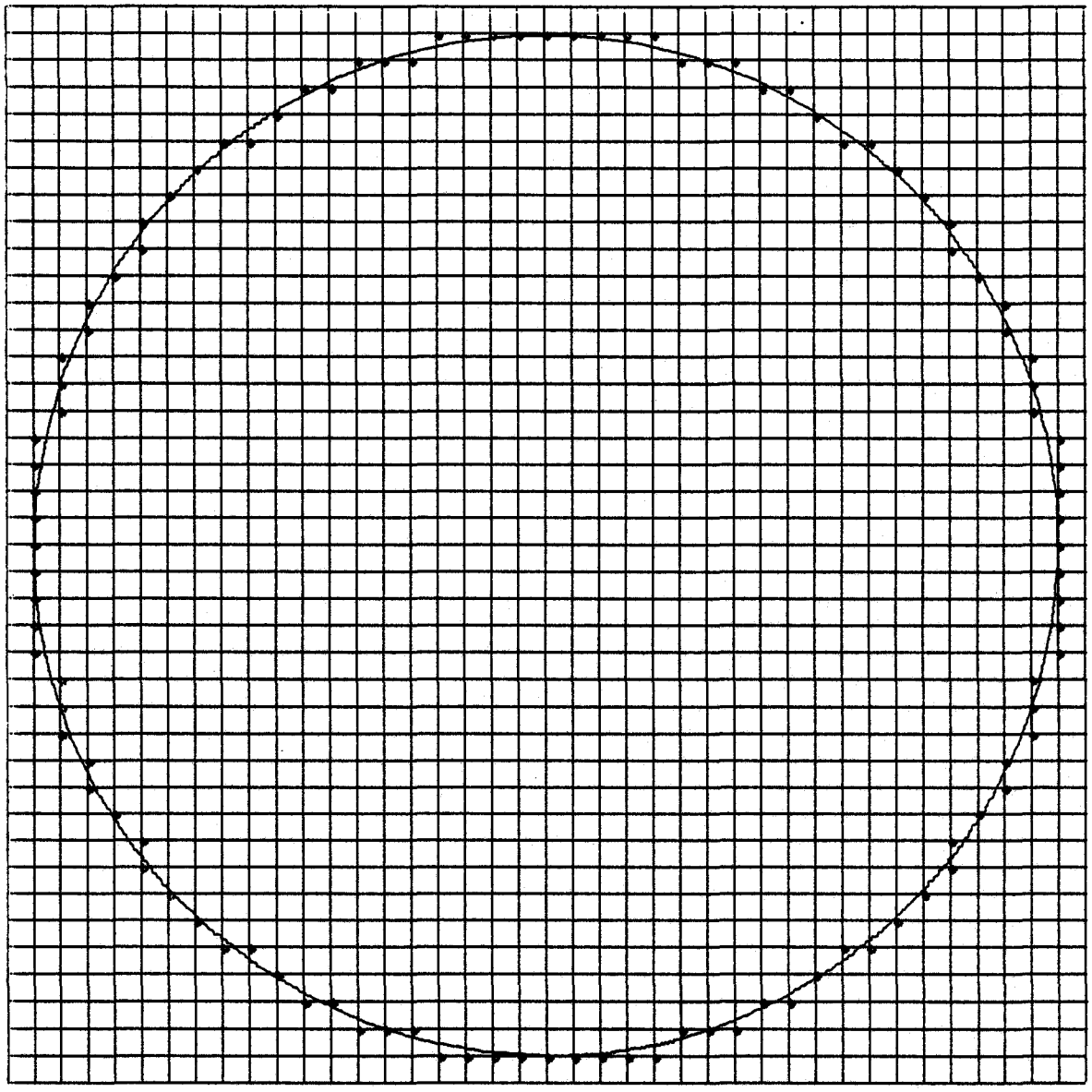
Moving from the diagonal octant boundary to the second octant's vertical boundary is symmetrically equivalent to reversing direction and returning to the first octant's horizontal boundary. This is exactly what is accomplished by a diagonal octant change.

As in the previous vector discussion, there are two types of motion. The actual drawing motion is defined by INSTRX, INSTRY and MOVEI as before. The equivalent motion remains in the first octant, and is described by X, Y, DX and DY. DX and DY represent the incremental changes for X and Y. For counter-clockwise motion, DX = -1 and DY = 1.

In a diagonal octant change, two things happen. First, the value of MOVEI is 'flipped'. A first octant vertical coordinate value becomes the second octant horizontal coordinate value. Secondly, the equivalent motion is reversed. DX and DY are multiplied by negative one and S^+ is transformed into S^-. As is easily deduced from their definitions, S^+ and S^- are related according to the equation, $S^- = -S^+ + 4Y - 2X + 1$. The equivalent motion now retraces the first octant arc, while the actual motion extends the original arc into the second octant.

In like manner, the arc can be drawn through the second octant and into the third. Of course, to accomplished this, a square octant boundary must be crossed. When this occurs, the equivalent motion again reverses, while the appropriate change is made to the actual motion. This change amounts to 'flipping' either INSTRX or INSTRY. If MOVEI corresponds to a vertical coordinate, then the value of INSTRX is changed. Otherwise INSTRY is the variable whose value changes.

At this point, it should be evident that whether the actual curve is drawn clockwise or anti-clockwise, it can be extended to a full circle. Every time an octant boundary is reached, an octant change is required. This continues until the number of octant changes,



```
PX =   39
PY =   20
X =   -19
Y =     0
DOCT =      7
FCORD =     0
```

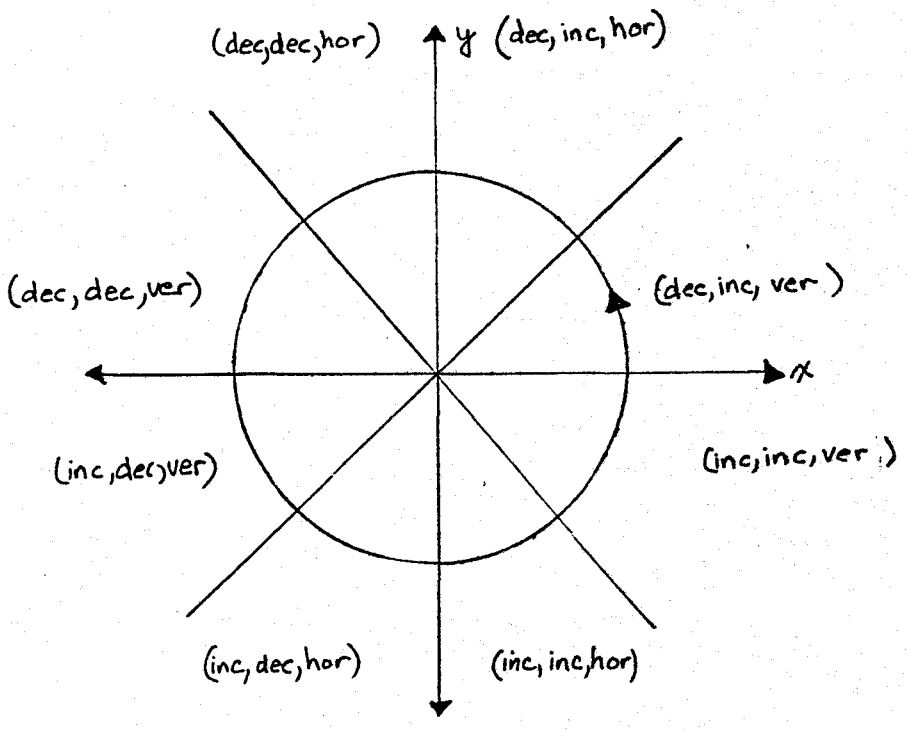
Fig. 4.4: Example of points selected by the circle generator.

which have occurred, is equal to the number specified by DOCT. The very next time Y = FCORD, the arc terminates.

The circle generator is easily made to accommodate arcs which begin in any octant. As before, the problem is simply mapped into its equivalent in the first octant. The actual motion, however, is defined according to the actual starting octant and direction. Figure 4.5 indicates the values of the actual motion variables, INSTRX, INSTRY and MOVEI, for the different octants. In this figure, a counter-clockwise direction is assumed. For a clockwise direction, INSTRX and INSTRY have values which are opposite to those of the figure. The following procedure is used to define the equivalent and actual motions properly.

First, the actual motion variables are set to their first octant values. If the arc is to be drawn in a counter-clockwise direction, INSTRX, INSTRY and MOVEI are given values of 'decrement', 'increment' and 'vertical' respectively. Secondly, the starting position (X, Y), and the drawing direction (DIRECT) are mapped into the first octant. As before, this is accomplished through reflections Rx, Ry and Rxy, about the x axis, the y axis, and the x-y diagonal, respectively. Each reflection results in a change in the value of DIRECT, the drawing direction. Finally, the inverse mapping is applied to the actual motion as represented by INSTRX, INSTRY and MOVEI. This inverse transformation maps the actual motion into the starting octant.

As an example, consider a starting arc position which lies in the sixth octant. The transformation Ry Rx Rxy, when applied to



Actual Motion: (INSTRX, INSTRY, MOVEI)

inc = increment
dec = decrement
hor = horizontal coordinate
ver = vertical coordinate

Fig. 4.5: As an ordered set, the actual motion variables may assume eight different values. For a given drawing direction, there is a one to one correspondence between the actual motion values and the eight octants.

X, Y and DIRECT, maps them into their equivalents in the first octant. The inverse transformation is simply Rxy Rx Ry. From figure 4.5, it can be seen that the transformation RX corresponds to 'flipping' the value for INSTRX. Similarly, the transformation RY corresponds to 'flipping' the value for INSTRY. Rxy is slightly more complicated. If INSTRX = INSTRY, the values of INSTRX, INSTRY and MOVEI are all changed. Otherwise, only the value of MOVEI is changed. The actual implementation of this scheme is simplified by the fact that Rx Ry = Ry Rx. Thus, the inverse of T = Ry Rx Rxy is T^{-1} = Rxy Ry Rx. This allows the Ry Rx part of the two mappings to be accomplished at the same time. For further details, the reader is referred to Appendix A.

This completes the presentation of the circle and line algorithms. In the next chapter, an estimate of the performance capabilities of the graphics controller is made.

CHAPTER 5

PERFORMANCE

A stated performance objective of the graphics terminal design is a host-terminal transmission rate of 9600 baud. In chapter two, it was concluded that the Intel 8086 based operating system is fast enough to process characters at the desired rate, providing that the graphics controller is able to keep up. The discussion was deferred until the graphics controller had been looked at in more detail. It is now appropriate to complete the analysis.

For this purpose, the graphics terminal is assumed to have the specifications found in table 2.1. In addition, it is assumed that a graphics controller microinstruction requires 125 nsec to execute. This is a typical clocking period for the particular AMD chip configuration used. Finally, a typical static RAM access time of 250 nsec is assumed to apply to bit map read/write operations. Read/write signals must be held stable for at least that long, before the operation can be assumed to have taken place.

Two worst-case situations shall be analyzed. In both instances, the host computer sends a continuous stream of 8 bit characters to the graphics terminal, at a rate of 960 characters per second. In the first case, these incoming characters are interpreted as corresponding solely to a sequence of line segment specifications. An image is being built up on the screen, which consists entirely of line segments. Similarly, in the second case, an image is being generated, which consists entirely

of circle segments. Each incoming character is interpreted as contributing to the specification of the next arc to be generated.

V. 1 Line Generation

In the line-drawing case, four characters are required to specify the displacements, VX and VY, of a line segment. For each incoming set of four characters, the operating system sends one VECTOR instruction to the graphics controller. Thus, during the time that the operating system takes to receive four characters, the graphics controller must be able to generate a complete line segment.

Upon completion of the graphics controller firmware, it was noted that the number of microinstruction executions required to generate a single vector dot, is 41. This is the number of microinstructions executed in one pass through the dot-select loop. For any given dot, the actual number may in fact be less. For example, the above count was made by assuming that a diagonal move was made in selecting the dot. A horizontal or vertical move requires fewer microinstructions. The important point, however, is that the number of microinstructions executed can never be greater than 41.

Forty-one microinstructions correspond to an execution time of 5.12 usec. During horizontal and vertical flybacks, this is the time taken to generate a single vector dot. During forward horizontal scans, however, the dot generation rate is less. Time is spent, before bit map accesses, in waiting for the graphics display process to relinquish its use of the bit map.

As discussed earlier, the graphics display process indicates its use of the bit map, by lowering a bit map free flag. During a

horizontal scan, 16 bits are mapped onto the screen every 1.3 usec. This means that every 1.3 usec, the display process requires the use of the bit map RAM for a 250 nsec read operation. However, the bit map free flag must be lowered in advance of each read requirement. This is necessary to provide ample time for the graphics controller to finish any bit map access, which was begun immediately before the bit map free flag was lowered. Enough time must be allowed for a complete read or a complete write operation to be performed.

A complete graphics controller read operation requires exactly four microinstructions. The first microinstruction issues the appropriate read signals. These signals are then held stable for two subsequent microinstructions. Finally, the execution of the fourth microinstruction brings the result into the ALU. In contrast, a write operation requires only three microinstructions. Whereas, the first microinstruction initiates the write operation, two further microinstruction cycles are required for the operation to be completed.

The net result is that the bit map free flag must be lowered for 250 nsec plus the duration of four microinstruction cycles, for each display process access. This means that out of every 1.3 usec, 750 nsec are spent with the bit map free flag in a lowered state. At any instant, the probability of the graphics controller not being able to immediately access the bit map, is $75/_{130}$. As a result, the dot generation process is lengthened by an average of $75/_{130} \times (\frac{1}{2} \times 750)$ nsec, for each bit map access. Note that $\frac{1}{2} \times 750$ nsec is just the average length of each access wait, whereas, $75/_{130}$ is the probability that a wait occurs. Since the dot select loop contains two bit map accesses,

a read and a write, the time required to generate each dot is increased by $2 \times (75/130) \times (\frac{1}{2} \times 750)$ nsec = .433 usec. Thus, during forward horizontal scans, a single vector dot is generated every 5.56 usec.

In the displaying of a single complete screen, there are 480 forward horizontal scans, 479 horizontal flybacks, and one vertical flyback. This leads to a time averaged dot generation time of

$$\frac{(480 \times 5.3 \text{ usec})(5.56 \text{ usec}) + (479 \times 15 \text{ usec})(5.12 \text{ usec}) + (280 \text{ usec})(5.12 \text{ usec})}{480 \times 53 \text{ usec} + 479 \times 15 \text{ usec} + 280 \text{ usec}}$$

$$= 5.46 \text{ usec/dot.}$$

The longest line segments which fit on the screen, consist of 650 dots. This many dots are generated in 650 x 5.46 usec = 3.55 msec. In other words, four incoming characters are processed in at least 3.55 msec. This corresponds to a baud rate of 11,300. In so far as line generation is concerned, it appears that the 9600 baud objective has been met.

V. 2 Circle Generation

The situation where circle segments, alone, are generated, is now analyzed. Seven eight bit characters are required to specify each arc. Values of X, Y and FCORD are derived from six incoming characters. The seventh character suffices to represent a value for DOCT. The analysis proceeds in exactly the same manner as for the previous case. This time, at most 47 microinstructions are executed each time an arc dot is generated. This leads to a time averaged dot generation time of 6.21 usec per dot.

The largest complete circle which may be generated is centered in the display screen and with a radius of 240 screen units. For this circle, the solution set is comprised of 1360 dots. This many dots is

generated in 6.21 usec x 1360 = 8.45 msec. In other words, seven incoming characters are processed in at least 8.45 msec. This corresponds to a baud rate of 8280 which is, of course, less than the 9600 baud objective. It should be pointed out, however, that the average arc specified is most likely to require somewhat less than 1360 dots. For arcs which require half as many dots, the baud rate is effectively doubled. Such arcs are still relatively large. Bearing this in mind, it appears that for circle generation, a host-terminal transmission rate of 9600 baud can, in fact, be maintained.

CHAPTER 6

CONCLUDING REMARKS

The main microprocessor and the graphics controller operate in parallel. The graphics controller generates circle and line segments at a rate which is fast enough to allow 9600 baud to be maintained. At the same time, the Intel 8086 executes approximately 600 instructions per each incoming character. This is enough to allow rather sophisticated graphics and text processing capabilities.

In part, high speeds are achieved at the expense of the extra complexity which is introduced through the use of microprogramming and bit-sliced technology. It is felt, however, that a favourable balance between simplicity and speed has been attained. State-of-the-art LSI technology is used in striking this balance. The use of standard, readily available LSI devices enables the design to be kept relatively simple. At the same time, recent breakthroughs in speed have lessened the severity of the ultimate tradeoff between speed and simplicity.

Programmable control provides the graphics terminal with a high degree of flexibility. Features such as general text editing, the specification of 'rubber band lines' (DIC),(NEW), image translation, image scaling and provision for user programming in a language such as PL/M (MCC), may be included as part of the operating system firmware. The graphics controller firmware may be extended as well. The microprogrammed graphics instruction set of chapter 4 requires 250 words (53 bits wide) of the 4K word address space of the AM2910 sequence controller (see fig. 3.2). Thus, the potential for extending the

graphics controller capabilities is very real. For example, the line and circle generators can be generalized to include the drawing of any conic section (PIT), (HOR).

On a closing note, comment is made on the basic modularity of the graphics terminal design. At the end of chapter 2, the graphics controller, the bit map, and the bit map address generator, were collectively referred to as a single modular graphics unit. This graphics unit can be successfully interfaced with any conventional, microprocessor-based, alphanumeric terminal. Modifications to the graphics controller firmware may be required, however, depending on whether or not a fully interlaced display is used.

The firmware was developed in a manner consistent with the aforementioned modularity. For example, the firmware is independent of the screen resolution. The number of displayable screen dots may vary up to a maximum number of 2^{20}, without necessitating any change to the firmware. For any particular resolution, the size of the corresponding bit map is fully specified by bit map parameter values stored in the internal ROM registers.

REFERENCES

(ALE) Alexandridis, N.A. "Bit-Sliced Microprocessor Architecture". Computer. June 1978, pp. 56-80.

(AMD) Advanced Micro Devices, Inc. AM2903 Four-Bit Bipolar Microprocessor Slice; AM2910 Microprogram Controller; Technical Data.

(BAS) Baskett, F. and Shustek, L. " The Design of a Low Cost Video Graphics Terminal". SLAC PUB-1715. Stanford Linear Accelerator Center, Stanford University, Stanford, Ca. Feb. 1976

(BOU) Boulaye, G.G. Microprogramming. Halsted Press, N.Y. (1975).

(DAV) Davidson, S. and Shriver, B.D. "An Overview of Firmware Engineering". Computer. May 1978, pp. 21-33.

(DIC) Dickinson, P.D. "Versatile Low-Cost Graphics Terminal Is Designed for Ease of Use". Hewlett-Packard Journal. January 1978, pp. 2-16.

(HAN) Hansen, B. Operating System Principles. Prentice Hall, Inc., Englewood Cliffs, N.J. (1973).

(HOL) Holm, W.A. How Television Works. N.V. Philips' Gloeilampenfabrieken, Eindhoven, Holland. (1958).

(HOR) Horn, K.P. "Circle Generators for Display Devices". Computer Graphics and Image Processing. Vol. 5, (1976), pp. 280-288.

(MCC) McCracken, D.D. A Guide to PL/M Programming for Microcomputer Applications. Addison-Wesley Publishing Company, Inc., Philippines (1978).

(MIC) Mick, J.R. and Brick, J. Advanced Micro Devices: Microprogramming Handbook. Advanced Micro Devices, Inc., Ca. (1976)

(MOR) Morse, S.P. and Pohlman, W.B. and Ravenel, B.W. "The Intel 8086 Microprocessor: A 16-bit Evolution of the 8080". Computer. June 1978, pp. 18-27.

(NEW) Newman, W.M. and Sproull, R.F. Principles of Interactive Computer Graphics. McGraw-Hill, Inc., N.Y. (1973).

(OSB I) Osborne, A. An Introduction to Microcomputers: Volume I - Basic Concepts. Adam Osborne and Associates, Inc., Ca. (1976).

(OSB II) Osborne, A. and Jacobson, S. and Kane, J. An Introduction to Microcomputers: Volume II - Some Real Products. June 1977. Revision, Adam Osborne and Associates, Inc., Ca. (1976)

(PIT) Pitteway, M.L.V. "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter". Computer Journal. Vol. 10, (1967-68), pp. 282-289.

(RED) Redfield, S.R. "A Study in Microprogrammed Processors: A Medium Sized Microprogrammed Processor". IEEE Transactions on Computers. Vol. C-20, No. 7, July 1971, pp. 743-750.

(SHU) Shustek, L. "The Internals of the Video Graphics Terminal". SLAC Report 199. Stanford Linear Accelerator Center, Stanford University, Stanford, Ca. December 1976.

APPENDIX A

Line and Circle Generators (IFTRAN)

```
      PROGRAM CIRCLE( INPUT , OUTPUT )
C*****************************************************************
C
C     PURPOSE:                          AN ALGORITHM FOR GENERATING DIGITAL
C                                       APPROXIMATIONS TO CIRCULAR CURVES IS
C                                       TESTED. THE ALGORITHM GENERATES AN
C                                       OPTIMUM SET OF CONTIGUOUS DOTS, ALL
C                                       OF WHICH LIE ON A DISCRETE GRID,
C                                       SUCH AS IS CHARACTERISTIC OF RASTER
C                                       SCAN DISPLAY DEVICES. THE USER MUST
C                                       SPECIFY THE GRID SIZE ( SEE SMLATE )
C                                       AS WELL AS THE ARC SPECIFICATIONS
C                                       ( SEE READIN ) IN ORDER TO PRODUCE
C                                       THE GRID, ARC, AND CONTIGUOUS SET OF
C                                       GRID POINTS AS OUTPUT ON A VERSATEC
C                                       PLOTTER.
C     CALLS:                            READIN, SMLATE, INITAL, DRAWC, PLOT
C     CALLED BY:                        NONE
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       NONE
C     GLOBAL VARIABLES INITIALIZED:     NONE
C     LOCAL VARIABLES:                  X , Y , DOCT , FCORD , PX , PY
C
C*****************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      COMMON / PLTVAR / S , DS1 , DS2 , DDS1 , DDS2 , DX , DY
      COMMON / VERSTC / SCALE
C READ ARC SPECIFATIONS
      CALL READIN( PX , PY , X , Y , DOCT , FCORD )
C DRAW GRID AND ARC ON VERSATEC
      CALL SMLATE( PX , PY , X , Y , DOCT , FCORD )
C INITIALIZE CIRCLE VARIABLES
      CALL INITAL( X , Y , DOCT )
C GENERATE CONTIGUOUS GRID POINTS
      CALL DRAWC( X , Y , DOCT , FCORD , PX , PY )
C END OF PLOT
      CALL PLOT( 14.0 , 0.0 , -3 )
      CALL PLOT( 0.0 , 0.0 , 999 )
      STOP
      END
```

```
      SUBROUTINE READIN( PX , PY , X , Y , DOCT , FCORD )
C*****************************************************************************
C
C     PURPOSE:                          READS USER"S ARC SPECIFICATIONS
C     CALLS:                            NONE
C     CALLED BY:                        CIRCLE
C     INPUT:                            PX , PY , X , Y , DOCT , FCORD
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       NONE
C     GLOBAL VARIABLES INITIALIZED:     PX,PY -   THE INITIAL GRID POSITION
C                                                 IN UNITS OF DOTS FROM THE
C                                                 ORIGIN AT THE LOWER LEFT
C                                                 HAND CORNER.
C                                       X,Y -     THE DISPLACEMENT, IN DOTS,
C                                                 OF THE ARC CENTER FROM PX,
C                                                 PY.
C                                       DOCT -    BITS 0 - 14: THE TOTAL
C                                                 NUMBER OF OCTANT CHANGES
C                                                 (RELATIVE TO THE ARC
C                                                 CENTER ) ENCOUNTERED IN
C                                                 COMPLETING THE ARC.
C                                                 BIT 15: IF ZERO DRAW
C                                                 COUNTER - CLOCKWISE ELSE
C                                                 CLOCKWISE.
C                                       FCORD -   UPON TERMINATION, THE
C                                                 ABSOLUTE VALUE OF THE MOST
C                                                 RAPIDLY VARYING COORDINATE
C                                                 ( RELATIVE TO ARC CENTER )
C     LOCAL VARIABLES:                  NONE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      READ*, PX , PY , X , Y , DOCT , FCORD
      RETURN
      END
```

```
      SUBROUTINE SMLATE( PX , PY , X , Y , DOCT , FCORD )
C*****************************************************************************
C
C     PURPOSE:                          PLOTS GRID AND ARC ON VERSATEC
C     CALLS:                            OCTANT,FINALPT,PLOT,ARC,LETTER
C     CALLED BY:                        CIRCLE
C     INPUT:                            SCALE
C     OUTPUT:                           VERSATEC PLOT
C     GLOBAL VARIABLES INHERITED:       PX , PY , X , Y , DOCT , FCORD
C     GLOBAL VARIABLES INITIALIZED:     SCALE -   NUMBER OF FINEST VERSATEC
C                                                 DIVISIONS TO FINEST GRID
C                                                 DIVISION
C     LOCAL VARIABLES:                  ZX,ZY,OCT,ZX2,ZY2,ZX1,ZY1,ZXC,ZYC,
C                                       ZOTHER,CLKWISE,I,DISPLAY
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / VERSTC / SCALE
      LOGICAL CLKWSE
      DIMENSION DISPLAY( 14 )
      READ*,SCALE
C DRAW VERTICAL GRID LINES
      ZX = 0.0
      UNTIL( ZX .GT. 10.4 )
         CALL PLOT( ZX , 0.0 , 3 )
         CALL PLOT( ZX , 10.4 , 2 )
         ZX = ZX + ( .005 * SCALE )
      END UNTIL
C DRAW HORIZONTAL GRID LINES
      ZY = 0.0
      UNTIL( ZY .GT. 10.40 )
         CALL PLOT( 0.0 , ZY , 3 )
         CALL PLOT( 10.4 , ZY , 2 )
         ZY = ZY + ( .005 * SCALE )
      END UNTIL
C CALCULATE FINAL ENDPOINT OF ARC
      IF( DOCT .GE. 2**15 )
         CLKWSE = .TRUE.
      ELSE
         CLKWSE = .FALSE.
      ENDIF
      CALL OCTANT( -X , -Y , CLKWSE , OCT )
      IF( CLKWSE )
         OCT = MOD( ( OCT - 1 ) + ( 8 - ( DOCT - 2**15 ) ) , 8 ) + 1
      ELSE
```

```
         OCT = MOD( OCT - 1 + DOCT , 8 ) + 1
      ENDIF
      ZOTHER = SQRT( FLOAT( X**2 + Y**2 - FCORD**2 ) )
      CALL FINALPT( OCT , FCORD , ZOTHER , ZX2 , ZY2 )
C CONVERT TO ABSOLUTE COORDINATES AND DRAW ARC
      ZXC = ( PX + X ) * SCALE * .005
      ZYC = ( PY + Y ) * SCALE * .005
      ZX1 = PX * SCALE * .005
      ZY1 = PY * SCALE * .005
      ZX2 = ZX2 * SCALE * .005 + ZXC
      ZY2 = ZY2 * SCALE * .005 + ZYC

      IF( CLKWSE )
         CALL ARC( ZX2 , ZY2 , ZX1 , ZY1 , ZXC , ZYC , .005 )
      ELSE
         CALL ARC( ZX1 , ZY1 , ZX2 , ZY2 , ZXC , ZYC , .005 )
      ENDIF
C PRINT DISPLAY INFORMATION ON VERSATEC PLOT
      ENCODE(140 , 1 , DISPLAY ) 5HPX = , PX , 5HPY = , PY , 4HX = , X ,
     1 4HY = , Y , 7HDOCT = , DOCT , 8HFCORD = , FCORD , 8HSCALE = ,
     1 SCALE
    1 FORMAT( A5 , I3 , 12X , A5 , I3 , 12X , A4 , I4 , 12X , A4 , I4 ,
     1 12X , A7 , I5 , 8X , A8 , I4 , 8X , A8 , I3 , 9X )
      DO( I = 1 , 7 )
         CALL LETTER( 20 , .15 , 0.0 , 11.2 , 6.0 - I * .2 ,
     1        DISPLAY( 2 * I - 1 ) )
      ENDDO
      RETURN
      END
```

```
      SUBROUTINE OCTANT( X , Y , CLKWSE , OCT )
C*****************************************************************************
C
C     PURPOSE:                          DETERMINES WHICH OCTANT THE POINT
C                                       DESIGNATED BY X,Y IS LOCATED IN
C     CALLS:                            NONE
C     CALLED BY:                        SMLATE
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       X,Y,CLKWSE
C     GLOBAL VARIABLES INITIALIZED:     OCT -      CALCULATED OCTANT
C     LOCAL VARIABLES:                  NONE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      LOGICAL CLKWSE
      IF( X .GT. 0 )
         IF( Y .GT. 0 )
            IF( X .GT.  Y )
               OCT = 1
            ELSE
C SPECIAL CARE MUST BE TAKEN FOR POINTS ON OCTANT BOUNDARY
               IF( X .EQ. Y .AND. CLKWSE )
                  OCT = 1
               ELSE
                  OCT = 2
               ENDIF
            ENDIF
         ELSE
            IF( X .GT. IABS( Y ) )
               IF( Y .EQ. 0 .AND. ( .NOT. CLKWSE ) )
                  OCT = 1
               ELSE
                  OCT = 8
               ENDIF
            ELSE
               IF( X .EQ. -Y .AND. ( .NOT. CLKWSE ) )
                  OCT = 8
               ELSE
                  OCT = 7
               ENDIF
            ENDIF
         ENDIF
```

```
ELSE
    IF( Y .GT. 0 )
        IF( IABS( X ) .GT. Y )
            OCT = 4
        ELSE
            IF( X .EQ. 0 .AND. CLKWSE )
                OCT = 2
            ORIF( X .EQ. -Y .AND. ( .NOT. CLKWSE ) )
                OCT = 4
            ELSE
                OCT = 3
            ENDIF
        ENDIF
    ELSE
        IF( IABS( X ) .GT. IABS( Y ) )
            IF( Y .EQ. 0 .AND. CLKWSE )
                OCT = 4
            ELSE
                OCT = 5
            ENDIF
        ELSE
            IF( Y .EQ. X .AND. CLKWSE )
                OCT = 5
            ORIF( X .EQ. 0 .AND. ( .NOT. CLKWSE ) )
                OCT = 7
            ELSE
                OCT = 6
            ENDIF
        ENDIF
    ENDIF
ENDIF
RETURN
END
```

```
      SUBROUTINE FINALPT( OCT , FCORD , ZOTHER , ZX2 , ZY2 )
C*****************************************************************************
C
C     PURPOSE:                          DETERMINES FINAL ENDPOINT OF ARC
C     CALLS:                            NONE
C     CALLED BY:                        SMLATE
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       OCT , FCORD ,
C                                       ZOTHER -  OTHER FINAL COORDINATE
C     GLOBAL VARIABLES INITIALIZED:     ZX2,ZY2 - FINAL POSITION RELATIVE TO
C                                                 ARC CENTER
C     LOCAL VARIABLES:                  NONE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      CASE OF ( OCT )
      CASE( 1 )
          ZX2 = ZOTHER
          ZY2 = FCORD
      CASE( 2 )
          ZX2 = FCORD
          ZY2 = ZOTHER
      CASE( 3 )
          ZX2 = -FCORD
          ZY2 = ZOTHER
      CASE( 4 )
          ZX2 = -ZOTHER
          ZY2 = FCORD
      CASE( 5 )
          ZX2 = -ZOTHER
          ZY2 = -FCORD
      CASE( 6 )
          ZX2 = -FCORD
          ZY2 = -ZOTHER
      CASE( 7 )
          ZX2 = FCORD
          ZY2 = -ZOTHER
      CASE( 8 )
          ZX2 = ZOTHER
          ZY2 = -FCORD
      END CASE
      RETURN
      END
```

```
      SUBROUTINE INITAL( X , Y , DOCT )
C*****************************************************************************
C
C     PURPOSE:                          INITIALIZES CIRCLE-GENERATING
C                                       VARIABLES
C     CALLS:                            MAP
C     CALLED BY:                        CIRCLE
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       X,Y,DOCT
C     GLOBAL VARIABLES INITIALIZED:     S -       DIAGONAL MOVE IF GT 0 ELSE
C                                                 MOVE1
C                                       DS1 -     ADD TO S AFTER MOVE1
C                                       DS2 -     ADD TO S AFTER DIAGONAL
C                                                 MOVE
C                                       DDS1 -    ADD TO DS1 AFTER EVERY
C                                                 MOVE ; ADD TO DS2 AFTER
C                                                 MOVE1
C                                       DDS2 -    ADD TO DS2 AFTER DIAGONAL
C                                                 MOVE
C                                       DX -      ADD TO X AFTER HORIZONTAL
C                                                 MOVE
C                                       DY -      ADD TO Y AFTER EVERY MOVE
C     LOCAL VARIABLES:                  DIRECT -  IF GT 0, COUNTER-CLKWISE
C                                                 ELSE CLKWISE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      COMMON / FLTVAR / S , DS1 , DS2 , DDS1 , DDS2 , DX , DY
C SEPARATE DIRECTION BIT FROM DOCT AND INITIALIZE DIRECT
      IF( DOCT .GE. 2**15 )
         DIRECT = -1
         DOCT = DOCT - 2**15
      ELSE
         DIRECT =1
      ENDIF
C TRANSFORM X,Y TO COORDINATES OF INITIAL POSITION W.R.T. ARC CENTER
      X = -X
      Y = -Y
C MAP X, Y, DIRECT INTO FIRST OCTANT
      CALL MAP( X,Y , DIRECT )
```

```
C INITIALIZE VARIABLES ACCORDING TO DIRECTION
      IF( DIRECT .GT. 0 )
         S = 2 * Y - X + 2
         DS1 = 2 * Y + 3
         DS2 = 2 * Y - 2 * X + 5
         DDS1 = 2
         DDS2 = 4
         DX = -1
         DY = 1
      ELSE
         S = 2 * Y - 2 * X - 1
         DS1 = 2 * Y - 3
         DS2 = 2 * Y - 2 * X - 5
         DDS1 = -2
         DDS2 = -4
         DX = 1
         DY = -1
      ENDIF
      RETURN
      END
```

```
      SUBROUTINE MAP( X , Y , DIRECT )
C*****************************************************************
C
C     PURPOSE:                          MAPS X,Y AND DIRECT FROM THE START-
C                                       ING OCTANT INTC THE FIRST OCTANT.
C                                       THE INVERSE TRANSFORM IS APPLIED
C                                       TO MOVE1,INSTRX,INSTRY TO MAP THE
C                                       ACTUAL MOVEMENT FROM FIRST TO START-
C                                       ING OCTANT.
C     CALLS:                            NONE
C     CALLED BY:                        INITAL
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       X,Y,DIRECT
C     GLOBAL VARIABLES INITIALIZED:     INSTRX -   INSTRUCTION FOR HORIZONTAL
C                                                  MOVE
C                                       INSTRY -   INSTR FOR VERTICAL MOVE
C                                       MOVE1 -    VALUE OF 1 : VERTICAL MOVE
C                                                  2 : HORIZONTAL MOVE
C     LOCAL VARIABLES:                  XYFLAG , TEMP
C
C*****************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      LOGICAL XYFLAG
      XYFLAG = .FALSE.
C SET INSTRX, INSTRY AS IF IN FIRST OCTANT, GT 0 MEANS INCREMENT , LT 0
C MEANS DECREMENT
      IF( DIRECT .GT. 0 )
         INSTRX = -1
         INSTRY = 1
      ELSE
         INSTRX = 1
         INSTRY = -1
      ENDIF
C REFLECT X,Y,DIRECT IN LINE Y = X IF NECESSARY
      IF( IABS( Y ) .GT. IABS( X ) )
         TEMP = Y
         Y = X
         X = TEMP
         DIRECT = -DIRECT
         XYFLAG = .TRUE.
      ENDIF
```

```
C REFLECT X,Y,DIRECT AND INSTR),INSTRY IN LINE X = 0
      IF( Y .LT. 0 )
         Y = - Y
C INSTRX IS FLIPPED
         INSTRX = -INSTRX
         DIRECT = -DIRECT
      ENDIF
C REFLECT X,Y,DIRECT AND INSTR),INSTRY IN LINE Y = 0
      IF( X .LT. 0 )
         X = -X
         INSTRY = -INSTRY
         DIRECT = -DIRECT
      ENDIF

C REFLECT INSTRX,INSTRY,MOVE1 IN LINE Y = X
      IF( XYFLAG )
         IF( INSTRX .EQ. INSTRY )
            INSTRY = -INSTRY
            INSTRX = -INSTRX
         ENDIF
         MOVE1 = 2
      ELSE
         MOVE1 = 1
      ENDIF
      RETURN
      END
```

```
      SUBROUTINE DRAWC( X , Y , DOCT , FCORD , PX , PY )
C*****************************************************************
C
C     PURPOSE:                          GENERATES THE CONTIGUOUS SET OF GRID
C                                       POINTS WHICH BEST REPRESENT ARC
C     CALLS:                            MOVEPT,DIAG,SQUARE
C     CALLED BY:                        CIRCLE
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       X,Y,MOVE1,S,DS1,DS2,DDS1,DDS2,DX,DY
C                                       DOCT,FCORD,PX,PY
C     GLOBAL VARIABLES INITIALIZED:     NONE
C     LOCAL VARIABLES:                  BESTPT -  1 : VERTICAL MOVE
C                                                 2 : HORIZONTAL MOVE
C                                                 3 : DIAGONAL MOVE
C*****************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      COMMON / PLTVAR / S , DS1 , DS2 , DDS1 , DDS2 , DX , DY
      COMMON / VERSTC / SCALE
C TEST IF STARTING MOVE CROSSES OCTANT BOUNDARY
      IF( Y + DY .GT. X .OR. Y + DY .LT. 0 )
         DOCT = DOCT + 1
      ENDIF
C GENERATE POINTS SEQUENTIALLY THROUGH INCREMENTAL CALCULATIONS
      UNTIL( DOCT .LE. 0 .AND. Y .EQ. FCORD )
         Y = Y + DY
C TEST FOR OCTANT CHANGE, FORCE X,Y TO REMAIN IN FIRST OCTANT
         IF( Y .GT. X )
            CALL DIAG( DOCT , X , Y )
         ORIF( Y .LT. 0 )
            CALL SQUARE( DOCT , X , Y )
         ORIF( Y .EQ. X .AND. S .GT. 0 )
            BESTPT = 3
            CALL MOVEPT( BESTPT , PX , PY )
            CALL DIAG( DOCT , X , Y )
         ENDIF
C CHOOSE BEST NEXT POINT
         IF( S .GT. 0 )
            BESTPT = 3
            S = S + DS2
            DS2 = DS2 + DDS2
            X = X + DX
         ELSE
```

```
C MOVE1 IS A VERTICAL OR HORIZONTAL MOVE
            BESTPT = MOVE1
            S = S + DS1
            DS2 = DS2 + DDS1
         ENDIF
         DS1 = DS1 + DDS1
         CALL MOVEPT( BESTPT , PX , PY )
      END UNTIL
      RETURN
      END

      SUBROUTINE MOVEPT( BESTPT , PX , PY )
C*****************************************************************************
C
C     PURPOSE:                          PERFORMS INCREMENTAL MOVE FROM PX,PY
C                                       AND PLACES DOT ON GRID
C     CALLS:                            SPOT
C     CALLED BY:                        DRAWC
C     INPUT:                            NONE
C     OUTPUT:                           VERSATEC PLOTTER
C     GLOBAL VARIABLES INHERITED:       BESTPT,PX,PY,INSTRX,INSTRY,SCALE
C     GLOBAL VARIABLES INITIALIZED:     NONE
C     LOCAL VARIABLES:                  NONE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      COMMON / VERSTC / SCALE
C IF BIT ZERO OF BESTPT IS SET , MOVE VERTICALLY ACCORDING TO INSTRY
      IF( AND( COMPL( MASK( 59 ) ) , BESTPT ) .NE. 0 )
         IF( INSTRY .GT. 0 )
            PY = PY + 1
         ELSE
            PY = PY - 1
         ENDIF
      ENDIF
C IF BIT ONE OF BESTPT IS SET, MOVE HORIZONTALLY ACCORDING TO INSTRX
      IF( AND( SHIFT( COMPL( MASK( 59 ) ) , 1 ) , BESTPT ) .NE. 0 )
         IF( INSTRX .GT. 0 )
            PX = PX + 1
         ELSE
            PX = PX - 1
         ENDIF
      ENDIF
C PLACE DOT
      CALL SPOT( PX * .005 * SCALE - .01 , PY * SCALE * .005 - .01 , .02
     1   , 1H* , 0.0 )
      RETURN
      END
```

```
      SUBROUTINE DIAG( DOCT , X , Y )
C*****************************************************************************
C
C     PURPOSE:                          DIAGONAL OCTANT CHANGE
C     CALLS:                            REVERS
C     CALLED BY:                        DRAWC
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       MOVE1,INSTRX,INSTRY,DOCT,X,Y,S,DS1,
C                                       DS2,DDS1,DDS2,DX,DY
C     GLOBAL VARIABLES INITIALIZED:     NONE
C     LOCAL VARIABLES:                  NONE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      COMMON / PLTVAR / S , DS1 , DS2 , DDS1 , DDS2 , DX , DY
C FLIP MOVE1
      MOVE1 = XOR( MOVE1 , 3 )
C REVERSE PRESENT DIRECTION
      Y = Y - 1
      S = -S + 4 * Y - 2 * X + 1
      DS1 = DS1 - 6
      DS2 = DS2 - 10
      DDS1 = -DDS1
      DDS2 = -DDS2
      DX = -DX
      DY = -DY
      Y = Y + DY
C DECREMENT OCTANT CHANGE COUNT
      DOCT = DOCT - 1
      RETURN
      END
```

```
      SUBROUTINE SQUARE( DOCT , X , Y )
C*****************************************************************************
C
C     PURPOSE:                          SQUARE OCTANT CHANGE
C     CALLS:                            REVERS
C     CALLED BY:                        DRAWC
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       X,Y,DOCT,MOVE1,INSTRX,INSTRY,S,DS1,
C                                       DS2,DDS1,DDS2,DX,DY
C     GLOBAL VARIABLES INITIALIZED:     NONE
C     LOCAL VARIABLES:                  NONE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      COMMON / FLTVAR / S , DS1 , DS2 , DDS1 , DDS2 , DX , DY
C FLIP DIAGONAL MOVE
      IF( MOVE1 .EQ. 1 )
         INSTRX = -INSTRX
      ELSE
         INSTRY = -INSTRY
      ENDIF
C REVERSE PRESENT DIRECTION
      S = -S - 2 * X + 1
      DS1 = DS1 + 6
      DS2 = DS2 + 10
      DDS1 = -DDS1
      DDS2 = -DDS2
      DX = -DX
      DY = -DY
      Y = Y + 2 * DY
C DECREMENT OCTANT CHANGE COUNT
      DOCT = DOCT - 1
      RETURN
      END
```

```
      PROGRAM VECTOR( INPUT , OUTPUT )
C*****************************************************************************
C
C     PURPOSE:                          AN ALGORITHM FOR GENERATING DIGITAL
C                                       APPROXIMATIONS TO STRAIGHT LINE
C                                       SEGMENTS IS TESTED. THE ALGORITHM
C                                       GENERATES AN OPTIMUM SET OF
C                                       CONTIGUOUS DOTS, ALL OF WHICH LIE ON
C                                       A DISCRETE GRID, SUCH AS IS
C                                       CHARACTERISTIC OF RASTER SCAN
C                                       DISPLAY DEVICES. THE USER MUST
C                                       SPECIFY THE GRID SIZE ( SEE SMLATV )
C                                       AS WELL AS THE STRAIGHT LINE
C                                       SPECIFICATIONS ( SEE READV ) IN
C                                       ORDER TO PRODUCE THE GRID, LINE AND
C                                       CONTIGUOUS SET OF GRID POINTS AS
C                                       OUTPUT ON A VERSATEC PLOTTER
C     CALLS:                            READV,SMLATV,MAPV,DRAWV,PLOT
C     CALLED BY:                        NONE
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       NONE
C     GLOBAL VARIABLES INITIALIZED:     NONE
C     LOCAL VARIABLES:                  PX,PY,VX,VY
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      COMMON / VERSTC / SCALE
C READ STRAIGHT LINE SPECIFICATIONS
      CALL READV( PX , PY , VX , VY )
C DRAW GRID AND STRAIGHT LINE ON VERSATEC
      CALL SMLATV( PX , PY , VX , VY )
C GENERATE CONTIGUOUS GRID POINTS
      CALL MAPV( VX , VY )
      CALL DRAWV( PX , PY , VX , VY )
C END OF PLOT
      CALL PLOT( 14.0 , 0.0 , -3 )
      CALL PLOT( 0.0 , 0.0 , 999 )
      STOP
      END
```

```
      SUBROUTINE READV( PX , PY , VX , VY )
C*****************************************************************************
C
C     PURPOSE:                          READS LINE SEGMENT SPECIFICATIONS
C     CALLS:                            NONE
C     CALLED BY:                        VECTOR
C     INPUT:                            PX,PY,VX,VY
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       NONE
C     GLOBAL VARIABLES INITIALIZED:     PX,PY -   THE INITIAL GRID POSITION
C                                                 IN UNITS OF DOTS FROM THE
C                                                 ORIGIN AT THE LOWER LEFT
C                                                 HAND CORNER.
C                                       VX,VY -   THE DISPLACEMENT, IN DOTS,
C                                                 OF THE FINAL GRID POSITION
C                                                 FROM PX,PY . THE LINE
C                                                 SEGMENT IS DRAWN FROM THE
C                                                 INITIAL TO FINAL GRID
C                                                 POSITION
C     LOCAL VARIABLES:                  NONE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      READ* , PX , PY , VX , VY
      RETURN
      END
```

```
      SUBROUTINE SHLATV( PX , PY , VX , VY )
C*****************************************************************************
C
C     PURPOSE:                          PLOTS GRID AND LINE SEGMENT ON
C                                       VERSATEC
C     CALLS:                            PLOT,LETTER
C     CALLED BY:                        VECTOR
C     INPUT:                            SCALE
C     OUTPUT:                           VERSATEC PLOT
C     GLOBAL VARIABLES INHERITED:       PX,PY,VX,VY
C     GLOBAL VARIABLES INITIALIZED:     SCALE -   NUMBER OF FINEST VERSATEC
C                                                 DIVISIONS TO FINEST GRID
C                                                 DIVISION
C     LOCAL VARIABLES:                  ZX,ZY,DISPLAY,I
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / VERSTC / SCALE
      DIMENSION DISPLAY( 10 )
      READ* , SCALE
C DRAW VERTICAL GRID LINES
      ZX = 0.0
      UNTIL( ZX .GT. 10.4 )
         CALL PLOT( ZX , 0.0 , 3 )
         CALL PLOT( ZX , 10.4 , 2 )
         ZX = ZX + ( .005 * SCALE )
      END UNTIL
C DRAW HORIZONTAL GRID LINES
      ZY = 0.0
      UNTIL( ZY .GT. 10.4 )
         CALL PLOT( 0.0 , ZY , 3 )
         CALL PLOT( 10.4 , ZY , 2 )
         ZY = ZY + ( .005 * SCALE )
      END UNTIL
C DRAW LINE SEGMENT USING VERSATEC ROUTINES
      CALL PLOT( PX * .005 * SCALE , PY * .005 * SCALE , 3 )
      CALL PLOT( ( PX + VX ) * .005 * SCALE , ( PY + VY ) * .005 * SCALE
     1 , 2 )
C PRINT DISPLAY INFORMATION ON VERSATEC PLOT
      ENCODE( 100 , 1 , DISPLAY ) 5HPX = , PX , 5HPY = , PY , 5HVX = ,
     1 VX , 5HVY = , VY , 8HSCALE = , SCALE
    1 FORMAT( A5 , I3 , 12X , A5 , I3 , 12X , A5 , I3 , 12X , A5 , I3 ,
     1 12X , A8 , I3 , 9X )
      DO( I = 1 , 5 )
         CALL LETTER( 20 , .15 , 0.0 , 11.2 , 5.8 - I * .2 , DISPLAY( 2
     1*    I - 1 ) )
      ENDDO
      RETURN
      END
```

```
      SUBROUTINE MAPV( VX , VY )
C*****************************************************************************
C
C     PURPOSE:                          THE COORDINATES VX,VY ARE MAPPED
C                                       INTO THEIR EQUIVALENTS IN THE FIRST
C                                       OCTANT. INSTRX , INSTRY , MOVE1 ARE
C                                       INITIALIZED ACCORDING TO REAL
C                                       DIRECTION OF MOTION
C     CALLS:                            NONE
C     CALLED BY:                        VECTOR
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       VX,VY
C     GLOBAL VARIABLES INITIALIZED:     INSTRX -   INSTRUCTION FOR
C                                                  HORIZONTAL MOVE
C                                       INSTRY -   INSTR FOR VERTICAL MOVE
C                                       MOVE1 -    VALUE OF 1: VERTICAL MOVE
C                                                  2: HORIZONTAL MOVE
C     LOCAL VARIABLES:                  TEMP
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
C INITIALIZE INSTRX,INSTRY - GT 0 MEANS INCREMENT , LT 0 MEANS DECREMENT
      IF( VX .GE. 0 )
         INSTRX = 1
      ELSE
         INSTRX = -1
         VX = -VX
      ENDIF
      IF( VY .GE. 0 )
         INSTRY = 1
      ELSE
         INSTRY = -1
         VY = -VY
      ENDIF
C MOVE1 CORRESPONDS TO MOTION IN DIRECTION OF GREATEST CHANGE
      IF( VY .GT. VX )
         MOVE1 = 1
         TEMP = VX
         VX = VY
         VY = TEMP
      ELSE
         MOVE1 = 2
      ENDIF
      RETURN
      END
```

```
      SUBROUTINE DRAWV( PX , PY , VX , VY )
C*****************************************************************************
C
C     PURPOSE:                          GENERATES THE SET OF CONTIGUOUS
C                                       GRID POINTS WHICH BEST REPRESENTS
C                                       THE LINE SEGMENT
C     CALLS:                            MOVEPT
C     CALLED BY:                        VECTOR
C     INPUT:                            NONE
C     OUTPUT:                           NONE
C     GLOBAL VARIABLES INHERITED:       PX,PY,VX,VY,MOVE1,INSTRY
C     GLOBAL VARIABLES INITIALIZED:     NONE
C     LOCAL VARIABLES:                  D -       DIAGONAL MOVE IF GT 0 ,
C                                                 ELSE MOVE1
C                                       DD1 -     ADD TO D AFTER MOVE1
C                                       DD2 -     ADD TO D AFTER DIAGONAL
C                                                 MOVE
C                                       BESTPT -  0: NO MOVE
C                                                 1: VERTICAL MOVE
C                                                 2: HORIZONTAL MOVE
C                                                 3: DIAGONAL MOVE
C
C*****************************************************************************
      IMPLICIT INTEGER ( A - Y ) , REAL( Z )
      COMMON / MOVE / INSTRX , INSTRY , MOVE1
      COMMON / VERSTC / SCALE
C TEST FOR GENERATION OF A SINGLE POINT
      IF( VX .EQ. 0 )
         BESTPT = 0
         CALL MOVEPT( BESTPT , PX , PY )
      ELSE
C GENERATE CONTIGUOUS POINTS SEQUENTIALLY THROUGH INCREMENTAL
C CALCULATIONS
         DD1 = 2 * VY
         D = DD1 - VX
         DD2 = D - VX
C FUDGE TO ENSURE SAME POINTS CHOSEN DRAWING FORWARDS AND BACKWARDS
         IF( INSTRY .LT. 0 )
            D = D + 1
         ENDIF
         UNTIL( VX .LE. 0 )
            IF( D .GT. 0 )
               D = D + DD2
               BESTPT = 3
            ELSE
               D = D + DD1
```

```
C MOVE1 IS A VERTICAL OR HORIZONTAL MOVE
                BESTPT = MOVE1
            ENDIF
            CALL MOVEPT( BESTPT , PX , PY )
            VX = VX - 1
        END UNTIL
    ENDIF
    RETURN
    END
```

APPENDIX B

Sample Microprogram

Working variables are mapped into the R/W part of the internal register file as shown below. The global variables are used in the implementation of both drawing instructions. The local variables are those which are local to either CIRCLE or VECTOR.

Global Variables

Register (R/W)	Variable
1	POSITN
2	BTMSK
3	EVODD
4	MODE
5	PATERN
6	PSCALE
7	PATPOS
8	SCLPOS
9	INSTRX
10	INSTRY
11	MOVEI
12	BESTPT
13	ADDR
14	MAPWRD

Local Variables

Register (R/W)	Variable	
	VECTOR	CIRCLE
15	VX	X
16	VY	Y
17	D	DOCT
18	DD1	FCORD
19	DD2	DIRECT
20	-	XYFLAG
21	-	S
22	-	DS1
23	-	DS2
24	-	DDS1

Register (R/W)	Variable	
	VECTOR	CIRCLE
25	-	DDS2
26	-	DX
27	-	DY
28	-	-
29	-	-
30	-	-
31	-	-
32	-	-

The following constants are stored in the 16 register ROM part of the internal register file.

Register (Read-Only)	Constant
1	width
2	gphevn
3	gphodd
4	2^{15}
5	2
6	3
7	4
8	1
9	10
10	length
11	-
12	-
13	-
14	-
15	-
16	-

With the exception of the ADDR and MAPWRD, definitions for the above variables and constants can be found in appendix A or in chapter 4. ADDR is a bit map address and MAPWRD, a bit map word.

The following routine is representative of the graphics controller firmware. The routine (DRAWC) is invoked during the execution of a CIRCLE instruction. It generates the contiguous set of grid points which best represents the specified circle segment. For comparison, see the corresponding IFTRAN version in appendix A.

Note that in the following, the data bus interface and bit map interface fields are not shown as a part of each microinstruction. During the entire routine, these fields remain fixed at their default values as is discussed in chapter 3.

DRAWC -- Invoked During Circle Generation

	Control Sequencer Fields					ALU Fields							
	AM2910 Instruction	Condition Code Enable	Force Test Polarity	Condition Code	Pipeline Data	Function	Carry In (Cn)	Shift	Shift In	A Address	B Address	Enable Status Load	
Drawc	CONT	X	X	X	X	R+Cn	0	NS	X	R16	R32	X	R32 = Y
+1	CONT	X	X	X	X	R+S+Cn	0	NS	X	R27	R32	E	R32 = R32 + DY
+2	CJP	CC	neg	S	Drawc +4	R-S-1+Cn	1	NS	X	R15	R32	E	IF R32 ≥ 0 go to Drawc+4
+3	CJP	UCP	X	X	Drawc+6	S+Cn	1	NS	X	X	R17	X	DOCT = DOCT +1 ; go to DRAWC +6
+4	CJP	CC	neg	S	Drawc+6	X	X	NOP	X	X	X	X	IF X-R32 ≥ 0 go to DRAWC +6
+5	CONT	X	X	X	X	S+Cn	1	NS	X	X	R17	X	DOCT = DOCT +1
+6	CONT	X	X	X	X	$\overline{S}$+Cn	1	NOP	X	X	R17	E	
+7	CJP	CC	pos	S	Drawc+9	S-R-1+Cn	1	NOP	X	R18	R16	E	IF -DOCT <0 go to Drawc+9
+8	CRTN	CC	pos	Z	X	X	X	NOP	X	X	X	X	IF Y=FCORD Return
+9	CONT	X	X	X	X	S+R+Cn	0	NS	X	R27	R16	X	Y= Y + DY
+10	CONT	X	X	X	X	S-R-1+Cn	1	NOP	X	R16	R15	E	
+11	CJP	CC	neg	S	Drawc+14	S+Cn	0	NOP	X	X	R16	E	IF X-Y ≥0 go to Drawc +14
+12	CJS	UCP	X	X	Diag	X	X	NOP	X	X	X	X	Call Diag
+13	CJP	UCP	X	X	Drawc+21	X	X	NOP	X	X	X	X	go to Drawc + 21
+14	CJP	CC	neg	S	Drawc+17	$\overline{S}$+Cn	1	NOP	X	X	R21	E	IF Y ≥0 go to Drawc +17
+15	CJS	UCP	X	X	Square	X	X	NOP	X	X	X	X	Call Square
+16	CJP	UCP	X	X	Drawc+21	X	X	NOP	X	X	X	X	go to Drawc +21
+17	CJP	CC	neg	S	Drawc+21	S-R-1+Cn	1	NOP	X	R15	R16	E	IF -S ≥0 go to Drawc+21
+18	CJP	CC	neg	Z	Drawc+21	X	X	NOP	X	X	X	X	IF Y-X ≠0 go to Drawc +21
+19	CJS	UCP	X	X	Movept	R+Cn	0	NS	X	C6	R12	X	Bestpt = 3 ; call movept
+20	CJS	UCP	X	X	Diag	X	X	NOP	X	X	X	X	Call diag
+21	CONT	X	X	X	X	$\overline{S}$+Cn	1	NOP	X	X	R21	E	
+22	CJP	UCP	neg	S	Drawc +27	R+Cn	0	NS	X	R11	R12	X	IF -S ≥ 0 go to Drawc +27 ; Bestpt=move1
+23	Cont	X	X	X	X	R+Cn	0	NS	X	C6	R12	X	Bestpt = 3
+24	cont	X	X	X	X	S+R+Cn	0	NS	X	R23	R21	X	S= S+ DS2
+25	cont	X	X	X	X	S+R+Cn	0	NS	X	R25	R23	X	DS2 = DS2 + DDS2
+26	CJP	X	X	X	Drawc+29	S+R+Cn	0	NS	X	R26	R15	X	X = X+ DX ; go to Drawc +29
+27	cont	X	X	X	X	S+R+Cn	0	NS	X	R22	R21	X	S = S+ DS1
+28	cont	X	X	X	X	S+R+Cn	0	NS	X	R24	R23	X	DS2 = DS2 + DDS1
+29	CJS	UCP	X	X	Movept	S+R+Cn	0	NS	X	R24	R22	X	DS1 = DS1 + DDS1 ; call Movept
+30	CJP	UCP	X	X	Drawc+7	$\overline{S}$+Cn	1	NOP	X	X	R17	E	go to Drawc + 7

Test if first move crosses Octant boundary

loop which generates points

Terminating condition

Test for octant change ; Force x, y to remain in first octant

Choose best next point a diagonal move

a vertical or diagonal move

loop back

X = Don't Care
R = R/W Register
C = Read only Register