High Speed Image Rasterization Using a Highly Parallel Smart Bulk Memory

Stefan Demetrescu

Technical Report No. 83-244

June 1983

This work has been sponsored by the Defense Advanced Research Projects Agency under contracts MDA903-79-C-0680 and N00039-83-K -043 1.

High Speed Image Rasterization Using a Highly Parallel Smart Bulk Memory

Stefan Demetrescu

Technical Report No. 83-244

June 1983

Computer Systems Laboratory
Departments of Computer Science and Electrical Engineering
Stanford University
Stanford, California 94305

Abstract

VLSI technology allows the efficient realization of a class of highly parallel architectures consisting of high density semiconductor memory with an on-chip processor which accesses the memory in large sections simultaneously. A processor is described which uses this architecture to rasterize lines, polygons and text quickly, providing the rasterization support required in high performance graphic raster displays and fast page printers. This on-chip processor translates high-level low bandwidth commands into low-level high bandwidth actions on chip, where the high bandwidth can be tolerated. This architecture is capable of achieving performance comparable to the "processor per pixel" approaches while avoiding the tremendous density penalty incurred by such approaches. Consequently, it is practical to build a very high performance high resolution system from a small number of these chips.

Key Words and Phrases: VLSI, Smart Bulk Memory, computer graphics. parallel computer architecture, array processing, parallel processing, smart memory, memory- access methods, raster grap hics, frame buffers, real- time grap hics, scan conversion

CR Categories: B.2.1, B.3.2, B.4.2. B.6.1. B.7.1, C.1.2. C.3, C.4, 1.3.1. I.3.3

1 Introduction and Background

The two methods for storing state information in digital VLSI designs are (i) memory as large collections of bits not all used simultaneously, commonly referred to as bulk memory, and (ii) memory as small collections of bits all used simultaneously, commonly referred to as registers.

Register memory is used when data is stored within a processing element. A small amount of data is stored (e.g. 16 bits) close to the place where it is required for the computation being performed. Because all of the data is stored or retrieved simultaneously, and because this block of stored data is not coordinated with any of the others, it is not possible to share circuitry so as to simplify the per-bit storage element. As a result, the density of register memory is low as compared with bulk memory.

Bulk memory is typified by static and dynamic random access memory (RAM). It is the densest form of storing digital information (that is, bits per unit area). The high density is achieved because: (i) the design can be highly regular, thus optimizing area, and (ii) each bit of information is allocated a very simple circuit. It is necessary to make up for this simplicity by adding support circuitry required to read and write this information. If the total number of memory cells is large, and the number of simultaneously accessed cells is small then the area penalty of this circuitry can he shared among many memory cells, so the extra area per cell is small.

. For example, a 16,384 bit dynamic RAM array stores each bit of information as either the presence or absence of charge in a small capacitor. These bits are arrayed as 64 "words" of 256 bits each (see Figure 1). All of the enabling transistors of each word are connected together. When it is desired to access one of the bits of the DRAM array, the proper word enable line is activated. This has the effect of releasing the charge stored in each of the 256 capacitors on the 256 "bit lines". Due to the small amount of charge stored in each of the enabled capacitors, the voltage difference generated on the bit lines is very small. As a result, it is necessary to amplify the signals with the aid of differential amplifiers. Subsequently, the proper one of the 256 bits is read or written and all 256 bits are stored back into their respective capacitors. Finally, the word enable line is deactivated, thus trapping the charge in the capacitors. As can be seen, the circuitry required per bit is very small. However, this requires a large amount of supportive circuitry (in the form of word select circuitry, differential amplifiers, etc.). This circuitry can be shared among the total number of bits each serves. In this example, each amplifier serves 64 bits so the total cost per bit due co the differential amplifier is very low.

For conventional Von Neumann processors, which operate on small amounts of state at a time, the bulk memory is the most natural form of storing the majority of state. This is because the total number of bits required for each operation is invariably a very small fraction of the total number of bits available. Thus. great gains in storage density can be made by sharing as much circuitry as possible.

For parallel processors, which operate on large amounts of state simultaneously, the register form of

storage appears better suited. For example, the class of parallel processors commonly known as systolic arrays [KL 80] almost invariably store all of their state as registers. Unfortunately, this approach to parallel computation leads to a situation in which any processing which requires a large amount of state is very space consuming due to the low density of register memory. For this reason, these state intensive computations are usually not considered to be practical using this approach.

Architectures which perform computations which require a large amount of state and which at the same time can benefit from parallelism represent another class of processors. In order for these processors to be practical, however, it is necessary that they take advantage of the high density afforded by bulk memory. This class of architectures will henceforth be referred to as *Smart Bulk* Memory systems.

This document describes the design of a Smart Bulk Memory system that is part of a general investigation of this class of VLSI architectures. Because this class of architectures is relatively unexplored, the author has chosen a sample problem which is well known and to which many architectural solutions have been suggested (both register and bulk memory based, and both serial and highly parallel) as an initial vehicle to explore this class of architectures. This is the problem of transforming graphical primitives (e.g. polygons, lines, text) into their raster approximations in order to display them on a raster device such as a television CRT.

2 The Rasterization Problem

In recent times computer graphics output has moved away from calligraphic displays (e.g., randomly scanned vector CRTs, pen plotters) and toward raster displays (e.g., television displays, raster page printers). This conversion is due to many reasons: (i) raster displays cost significantly less than other display methods, due in large part to cost reductions brought about by the consumer television industry, (ii) most raster displays rely on a frame buffer (i.e. a large memory which holds all of the pixels to be displayed) and the cost of semiconductor memory has recently declined sharply, (iii) raster displays can fill areas with solid colors (and shading) whereas calligraphic displays can only draw outlines (efficiently), and (iv) raster displays can display characters in many font styles more naturally and efficiently than calligraphic displays.

A graphical display system using a raster display is shown in Figure 2. Such a system is given high level descriptions of a two or three dimensional image in world coordinates (i.e. the coordinates which most naturally describe the image). This image is transformed and clipped using well known graphical methods [NS 79] into a two dimensional representation in terms of graphical primitives described in screen coordinates (i.e. device dependent coordinates). These transformations have recently been incorporated into a VLSI design [Cl 82]. At this point, the rasterizer must transform these primitives into their rasterized approximation and must place the primitives at the proper place on the partially completed rasterized image which the rasterizer maintains.

Unfortunately, the move from calligraphic to raster displays has brought new problems. One of these problems is the the large number of pixels which must be altered when drawing the graphical

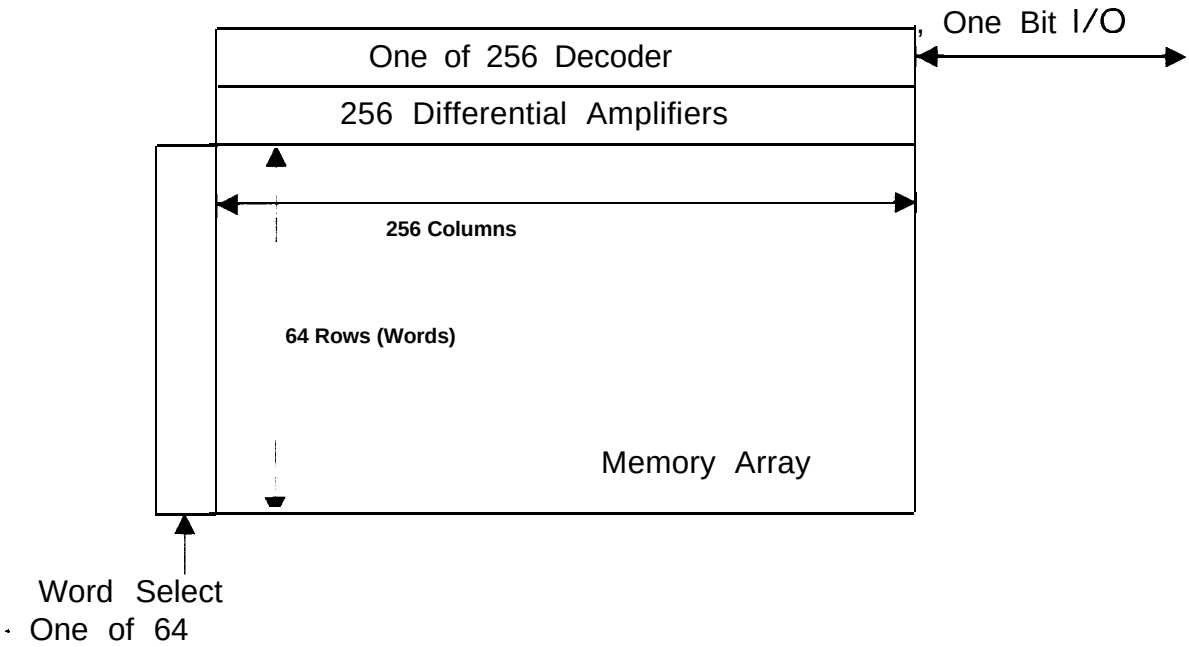


Figure 1a
64K by One Bit Dynamic RAM

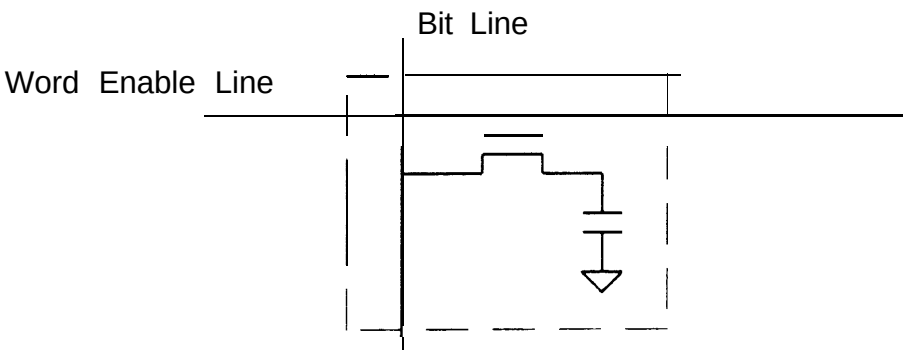


Figure 1 b
Dynamic RAM One Bit Cell

primitives (e.g., polygons). It is necessary not only to compute the vertices of the polygons but also to fill all of the pixels in the interior of the polygons with the desired color. Thus, the speed with which the polygons can be filled is typically much slower than the speed with which the position of the polygons can be calculated. For example, for a typical square (only 4 vertices) of 100 by 100 pixels, it is necessary to fill 10000 pixels.

As a result, the use of raster displays for real time images has been limited and expensive. For example, if an image of 1000 by 1000 pixels is to be imaged 30 times a second, one must typically access more than 30 million pixels per second. This is the rasterization problem.

3 A Smart Bulk Memory Rasterization System

In order to solve the rasterization problem, the process can be separated into two parts (Figure 3). The first part, performed by the Scan Line Processor, breaks up the graphical primitives into sequences of pixel lines to be filled in order to represent the primitives (the section titled Scan Line Processor Operations discusses this transformation further). The second part, iabelled as the Raster Processors, is responsible for maintaining the current raster image and for modifying the image as new commands are received from the Scan Line Processors.

Figure 4 shows a simplified block diagram of a typical rasterization system using Raster Processors. . The system shown is capable of displaying 1024 by 1024 pixels, each pixel being represented by one bit. The Raster Processors are each assumed to control 64 lines of 256 pixels each. Thus, each row in the figure can be thought of as representing 64 lines of 1024 pixels each. The Raster Processors are all bussed together so that the as far as the controlling Scan Line Processors are concerned, the four Raster Processors operate as if they were actually one large 64 by 1024 processor. It is necessary to have 16 rows of these processors in order to get 1024 scan lines. Rows do not depend on each other, so each row can operate in parallel with the others.

The Scan Line Processors are themselves connected on a common bus. On this bus are placed the commands which describe the primitives to be imaged. All of these processors can be connected on the same bus because they all operate on the same primitive at the same time. The processor controlling the Scan Line Processors need not be explicitly aware of how many of these processors are operating in parallel.

4 The Raster Processor

The architecture of a high density semiconductor dynamic RAM is highly parallel (Figure 1). For the RAM shown, each time that one bit of memory is accessed. a whole word consisting of 256 bits is recalled.

It is possible to make each row (i.e. each word) of the RAM represent a scan line and each bit contained in each row to represent a pixel within that scan line. This can be viewed as a frame buffer of 64 lines of 256 pixels of one bit each if a 16K-bit RAM is used. Of course. many such

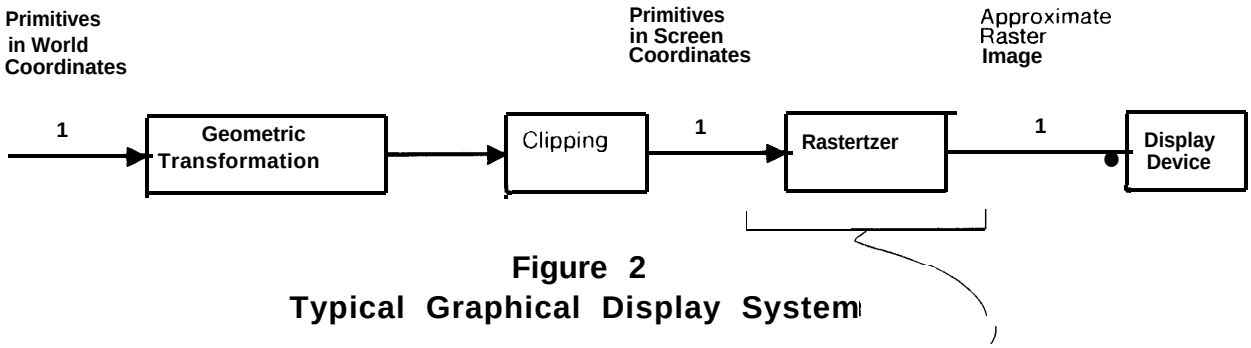


Figure 2
Typical Graphical Display System

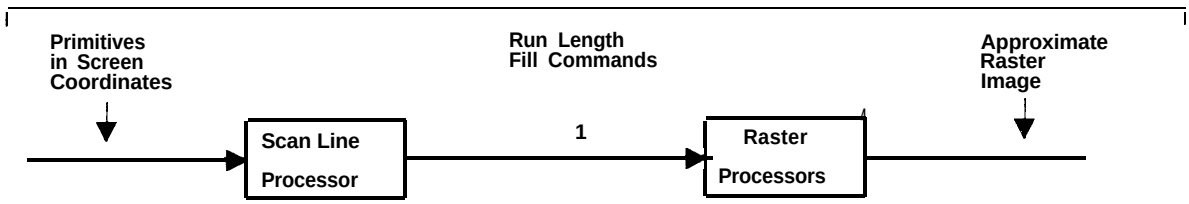
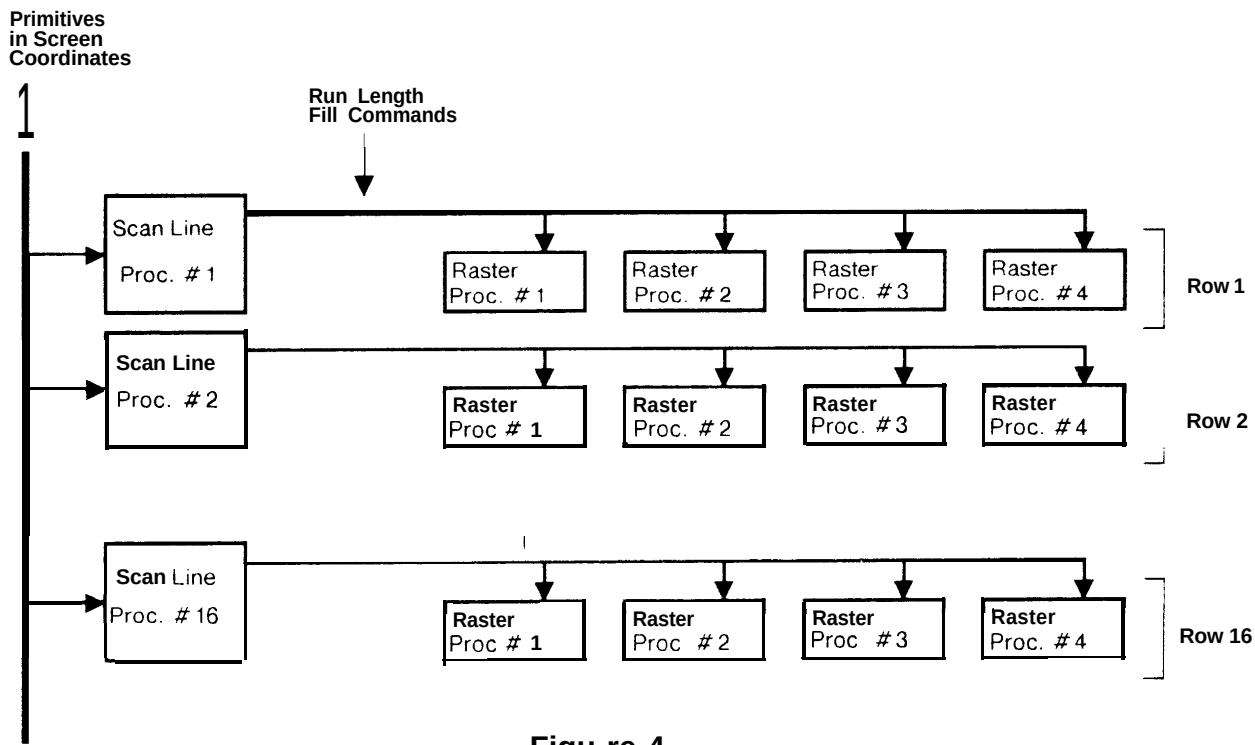


Figure 3
Rasterizer Block Diagram



Figu re 4
Basic System Configuration

chips must be used to tile an image of typical size with many bits per pixel.

It is then possible to exploit the fact that one can operate on one complete scan line every memory cycle. It is merely necessary to place a special purpose processor which is capable of filling many pixels on the scan line simultaneously with some desired pattern next to the differential amplifiers. This processor and memory combination (referred to as the *Raster Processor)* can be given simple, short commands (low-bandwidth off-chip communication) and can transform them to operations on many pixels (high- bandwidth on-chip operations).

In order to keep the Raster Processor as simple as possible, only the operations which must be next to the memory array to manipulate the wide memory word directly are placed on the chip. All operations that can be performed externally without a loss of parallelism or an increase in the communication bandwidth are left out. As a result, the task of transforming graphical primitives into sequences of line fill operations is not performed in the Raster Processor.

4.1 Raster Processor Operations

The Raster Processor can perform two basic classes of operations: (i) it can operate on a specified subrange of pixels within a given pixel line, and (ii) it can emit the completed raster in order to actually display the image. Appendix I contains a detailed description of the interface to the Raster Processor.

The operation which places images into the stored raster is transmitted to the Raster Processor in four sequential parts of 16 bits each (Figure 5 illustrates the effect of this operation for a particular halftone pattern). The first part specifies the pixel line on which the operation is to take place (i.e. which Y coordinate). This is followed by two X coordinates that specify the first and last pixels of the specified scan line to be affected. Finally, a 16 bit pattern is sent. This pattern is used to fill the specified area. If the area is larger than 16 bits, the pattern repeats. This pattern has two primary uses. When a polygon is being imaged, it represents the halftone pattern to be used in filling the interior of the polygon. When a character is being imaged, the pattern represents the actual rasterization of the character for that scan line.

Each processor can be programmed to respond to only a specified range of X and Y coordinates. Thus. many processors can be connected together on the same 16 bit command bus. Furthermore. many processors can be connected in parallel to store more than one bit per pixel.

When it is desired to extract the image from the Raster Processor. a command can be given which can output contiguous sections of 16 pixels from a specified pixel line on the 16 bit bus (the same one which is normally used for rasterization commands). Thus, the image can be extracted sequentially in order to display it.

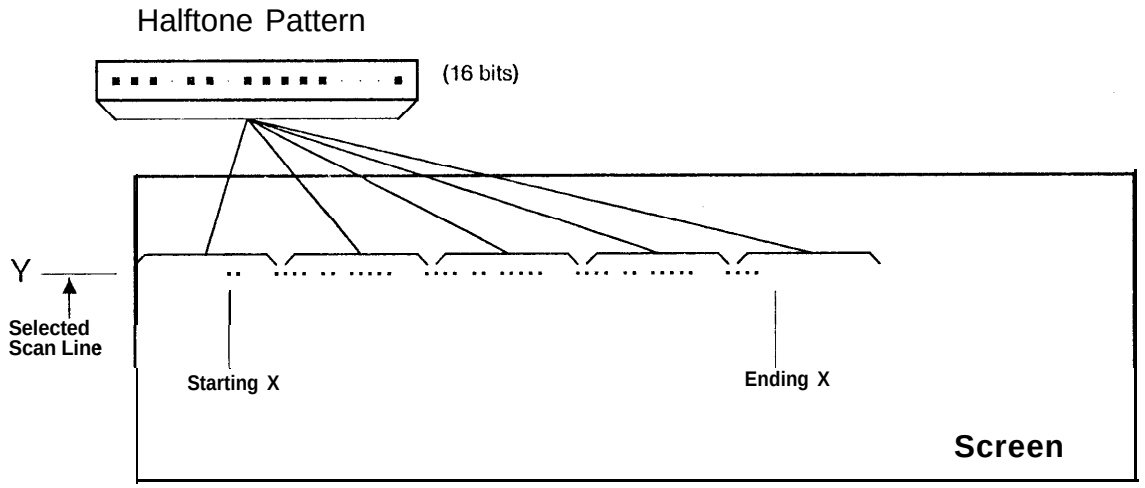


Figure 5
Scan Line Rasterize Operation

4.2 Raster Processor Design

The Raster Processor is composed of 6 main sections (see Figure 6): (i) the main memory array with required circuitry, (ii) the halftone ALU which operates on the incoming halftone pattern before it is used, (iii) the parallel comparator, which enables the proper subrange of the scan line, (iv) the scan line ALU which determines what value is to be stored back into the memory array given the input values from the parallel comparator, the halftone ALU (through the halftone bus) and the memory array, (v) the display latches which latch the scan line which is currently being sent to be displayed, and (vi) the control logic (i.e. control of the memory array, the parallel comparator, the ALUs, and the display latches).

The main memory array is a standard RAM design which is organized as rows and columns which are powers of 2 in size. No special modifications need to be made to the design. The exact kind of storage method (e.g. one transistor dynamic cell, 3 transistor dynamic cell, pseudostatic, fully static) is not important. The choice is to be made on the basis of an available proven design with the highest practical density. Furthermore, it is desirable to have an array much wider than it is long in order to achieve the largest amount of parallelism possible. For the remainder of this discussion, it will be assumed that a 16K-bit one transistor dynamic RAM is to be used which is organized as 64 words (i.e. rows) of 256 bits (i.e. columns) each.

The halftone ALU intercepts the incoming 16 bit halftone pattern and performs one of four operations on it: (i) no operation, (ii) logical bitwise inversion, (iii) turn to all 1s, (iv) turn to all 0s. The purpose of these options is to allow for the possibility of performing multiple value halftoning while imaging primitives (discussed in the System Design Issues section).

The parallel comparator takes an X coordinate as an input and produces N bits as an output (where N = number of bits in each word of the RAM array which is 256 in this discussion). Its purpose is to set all output bits whose position is less than the given X coordinate. Thus. the comparator is used to select the left and right limits of the pixels to be affected during an operation. These bits are used by the scan line ALU.

In order to distribute the repeating halftone pattern bits to the corresponding bits of the memory words, it is merely necessary to send each of the 16 bits from the halftone ALU to every 16th position. This is achieved by running a 16 bit bus horizontally above the memory array (as shown in Figure 6). However. if it is desired to place patterns which are aligned with respect to the starting X coordinate (e.g. for rasterizing characters), it is necessary to circularly shift the pattern by the amount (X mod 16.) It is not necessary to burden the Raster Processor with performing a barrel shift operation since it can be performed by the controlling Scan Line Processor without any increase in the communication bandwidth between the two. This is advantageous since one Scan Line Processor is expected to control many Raster Processors.

The scan line ALU is the part of the processor which uses the values generated by the parallel comparator. the halftone pattern, and the scan line read from the memory in order to generate the new scan line to be stored back into the memory. It also receives a set of control lines which

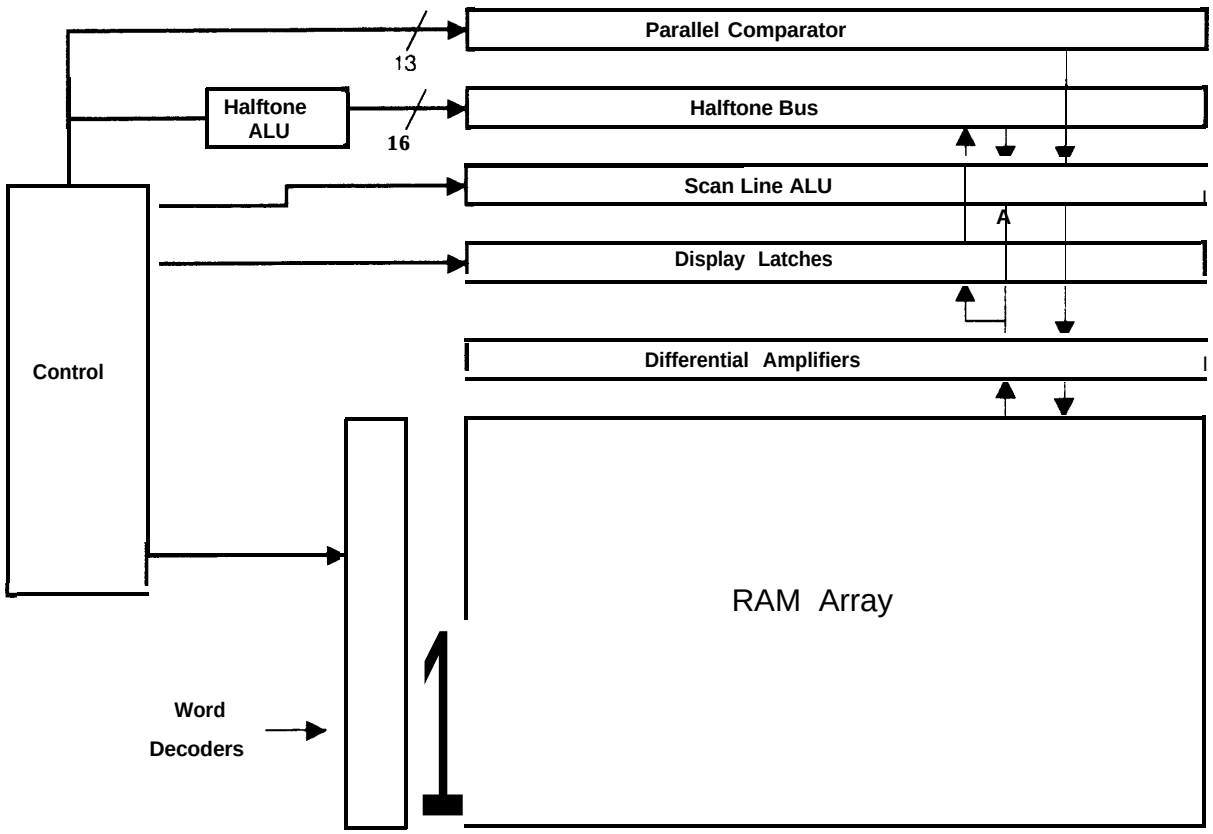


Figure 6

Raster Processor Block Diagram

indicate what function is to be performed during each cycle. The ALU is composed of one cell for each bit in the scan line. The function of each cell is described below. The description follows a typical cycle of the processor (see Figure 7 for a block diagram of the scan line ALU):

First, the (inclusive) starting coordinate (Xs) of the X extent to be affected is placed on the input of the parallel comparator (See Appendix II). Then, the inverse of the output of the parallel comparator is latched into latch L1. Thus. the latch array L1 contains a 1 for all locations along the scan line which are greater than or equal to Xs.

Second, the (exclusive) ending coordinate (Xe) of the X extent is placed on the input of the parallel comparator and the comparator's output is latched into latch L2. Thus, the latch L2 contains a 1 for all locations along the scan line which are less than Xe. The output of the AND gate shown in the figure (denoted as SEL(j)) is then a 1 for all X coordinates which lie in the range [Xs, Xe).

By this time, the RAM array has retrieved the current values of the pixels in the currently selected scan line. They are available on the lines labeled IR(j). Finally, the ALU can operate on the selected bits as desired and generate the pixels IW(j) to be written back into the memory array.

In order to keep the ALU as simple as possible, only the following minimal set of operations has been implemented: (i) No operation, (ii) Replace (i.e. store the halftone pixels instead of the current pixels for all selected pixels), (iii) Or (i.e. OR the halftone pixels with all pixels which have been selected), (iv) And: (i.e. AND the halftone pixels with all pixels which have been selected).

Of course, other functions are possible, at the expense of making the ALU larger. However, they are not required for the rasterization operations. The section titled Continuing and Future Work briefly discusses the possibility of turning the Raster Processor into a general purpose computing engine by using a more general purpose ALU and by exchanging some speed for generality.

5 Scan Line Processor

The Scan Line Processor transforms high level graphical primitives (i.e. polygons, lines, text). into sequences of commands to fill a specified section of a specified pixel scan line with a specified halftone pattern. These commands are transmitted to the Raster Processors it controls.

5.1 Scan Line Processor Operations

For simplicity, the Scan Line Processors only process polygons which are monotone in Y. That is, polygons which exhibit the property that any given horizontal scan line only intersects the boundary of the polygon at most twice. Convex polygons are a special case of monotone polygons. The polygon vertices are presented to the processor in descending Y order (see Figure 8). Each vertex is also labeled as to whether it is part of the left or the right edge of the polygon. The processor also contains a 16 by 16 halftone memory which can be loaded with the desired halftone pattern to be used while filling the polygon. Lines are rasterized as thin polygons.

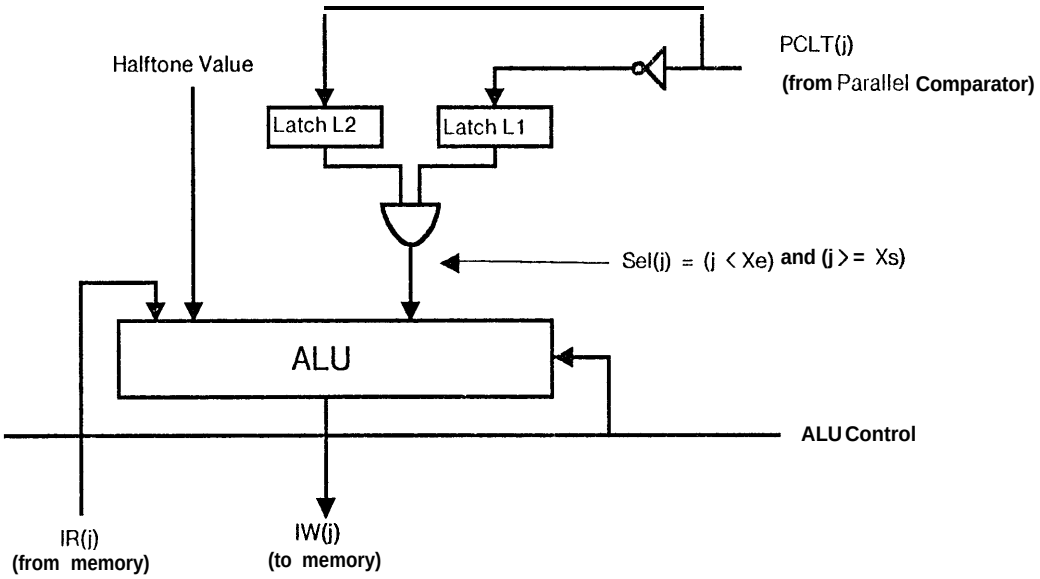


Figure 7
ALU for Pixel position j

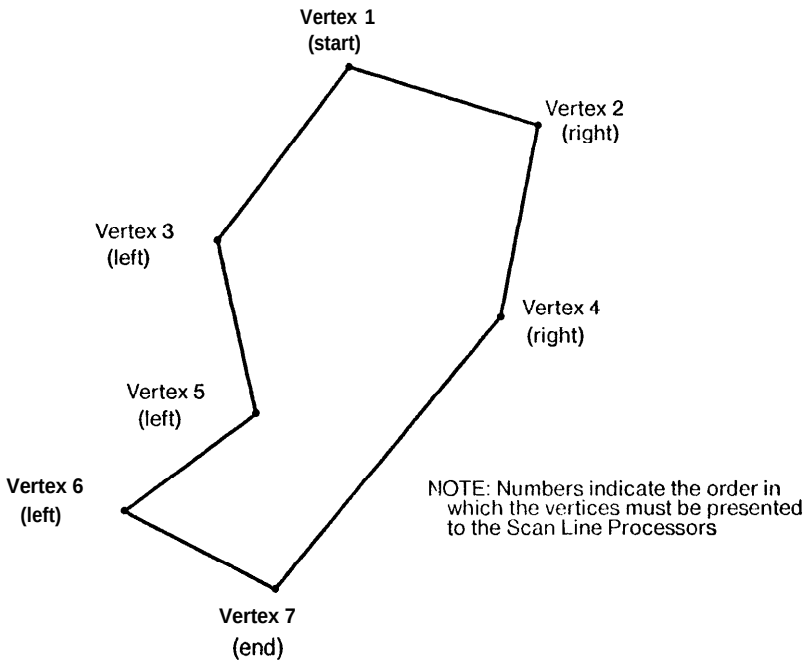


Figure 8
A Monotone Polygon

Characters are processed in one of three ways by the Scan Line Processor. Each approach is best suited for a different application. One approach is to specify the position of a character and to follow it by the pattern representing the character. This has the advantage of simplicity but suffers from the fact that one can not take advantage of any possible parallelism which could occur if more than one Scan Line Processor were operating simultaneously.

A second approach is to rasterize all identical characters on an image together (e.g. place all of the lower case "a" letters first, then all lower case "b", etc.). In this case, it is only necessary to send the raster description of a character to the Scan Line Processor once. Each Scan Line Processor can then temporarily store this information (only one character at a time). This can be followed by a list of positions where this character is to be placed. For example, if a screen has 4000 characters and roughly 60 different kinds of characters on it, it is expected that each character will be rasterized in 67 places. Thus, the time to load the pattern is effectively reduced by a factor of 67. Of course, this forces the parent processor to present all letters of the same kind at once.

A third approach is to endow each scan line processor with a font memory (or a font cache. which contains only the most used characters). In this case, it is only necessary to specify the code name and location of each character. Each processor can then retrieve the character description in parallel with the others and image the character simultaneously. This approach suffers from the fact that a large memory is usually necessary in order to store a respectable number of character descriptions. However, the characters can be imaged in any order.

This processor is only a support for the Raster Processor. Most of the power of this rasterization system stems from the parallelism achieved within the Raster Processor.

5.2 Scan Line Processor Design

The basic design of a Scan Line Processor is shown as a simplified block diagram in Figure 9. All commands are interpreted by the command decoder which proceeds to dispatch the commands to the appropriate processor. The polygon processor is in charge of rasterizing the current polygon until the end of either the right or the left current edge. When this occurs, the polygon processor awaits the next edge from the command processor. When the next edge is received, the polygon is rasterized further. The two edge processors are charged with calculating in parallel the beginning and ending X coordinates for the next scan line to be rasterized.

The command processor is also responsible for filling the halftone memory (used for filling polygons) and the font memory. The font memory is either a one character memory or is a many character font cache memory.

l'he font processor is responsible for placing the current character in the raster. It reads the character pattern from the font memory and uses the barrel shifter to align the character pattern properly for placement by the Raster Processors (since the Raster Processors place the halftone pattern in locations fixed with respect to the raster).

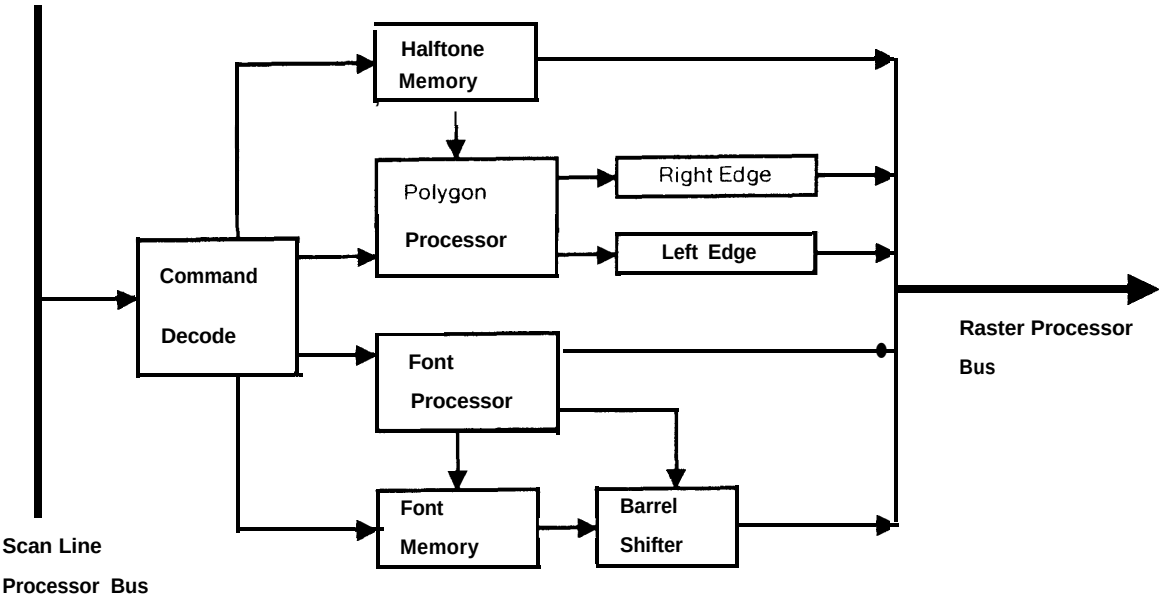


Figure 9
Scan Line Processor Block Diagram

6 System Design Issues

For the system shown in Figure 4, it is necessary to have 16 rows of processors in order to get 1024 scan lines. In the figure, each row of processors is shown having its own Scan Line Processor. If this is done, each row can operate in parallel with the others, thus achieving a 16 way parallelism. It is. however, possible to have one Scan Line Processor control more than one row of Raster Processors (see Figure 10, for example). In fact, it is possible to have only one Scan Line Processor controlling all of the rows. This decreases the parallelism achievable and thus decreases the maximum attainable performance.

The assignment of the scan lines on the screen to the Raster Processor rows affects performance. For example, it is possible to assign the first 64 lines to the first row, the second 64 to the second row, and so on. This has the disadvantage that when small, local objects are being rasterized, only a small number of the rows can operate simultaneously, since the object only intersects a small number of contiguous scan lines. It is much better to interleave the allocation of the scan lines to the processors. That is, each row can be assigned every 16th scan line. Thus, the first row is responsible for lines 0, 16, 32, 48, etc.. the second row is responsible for lines 1, 17, 33, 49, etc., and so on. So, even when a small object is rasterized, it will typically involve all of the processors if it spans more than 16 contiguous scan lines. As a result, almost all imaged objects can be expected to benefit from a 16 way parallelism.

This architecture is designed for real time graphics, so the ability to do double buffering (i.e. prepare one image while another one is being displayed) is essential. This can be achieved as shown in Figure 11. One set of Raster Processors is connected to the display circuitry while the other set is being controlled by the Scan Line Processors which are preparing the next frame. This allows the Scan Line Processors to be fully utilized.

It is also easily possible to design a system with multiple bits per pixel. For example. Figure 12 shows a system with 3 bits per pixel. In order to control multiple bit planes it is only necessary to add two separate control lines from the Scan Line Processor to each separate bit plane. The bulk of the control lines (the 16 lines) can still be shared between all of the processors in all bit planes.

The two select lines cause all of the processors in the bit plane to interpret the incoming halftone pattern in one of four ways: (i) the incoming pattern is used as is. (ii) the incoming pattern is inverted before it is used, (iii) the incoming pattern is ignored and all 1s are used in its place. (iv) the incoming pattern is ignored and all 0s are used in its place. The purpose of these options is to allow for the possibility of performing multiple value halftoning while imaging primitives. For example, if a system is only built with three bits per pixel. this allows each pixel to take on only one of 8 levels of gray. It is however. possible to halftone by using a mixture of two of the 8 gray scale values. For example. if one wanted to achieve an intensity of 5.5, one could fill the polygon with an alternating pattern of gray value 5 and gray value 6. This effect can be achieved by issuing pixel fill commands to the Raster Processor while commanding that the most significant bit plane use a halftone pattern of all 1s, the middle bit plane use the halftone pattern as given, and the least

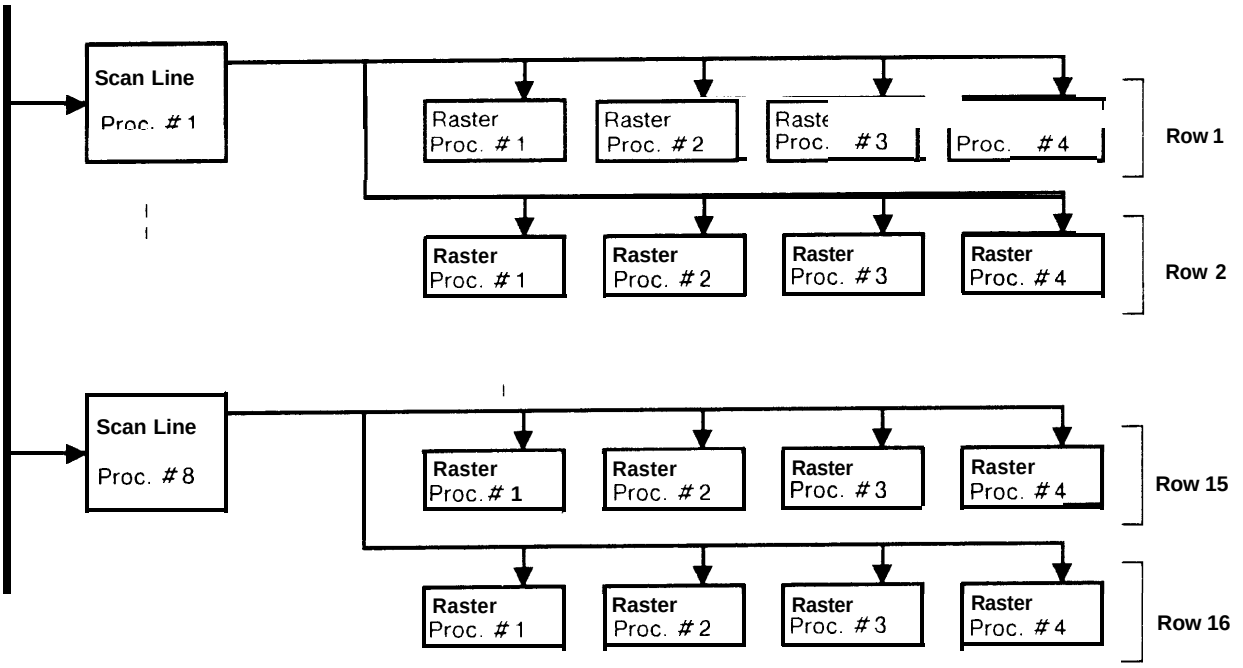


Figure 10

System Where One Scan Line Processor Controls Two Rows of Raster Processors

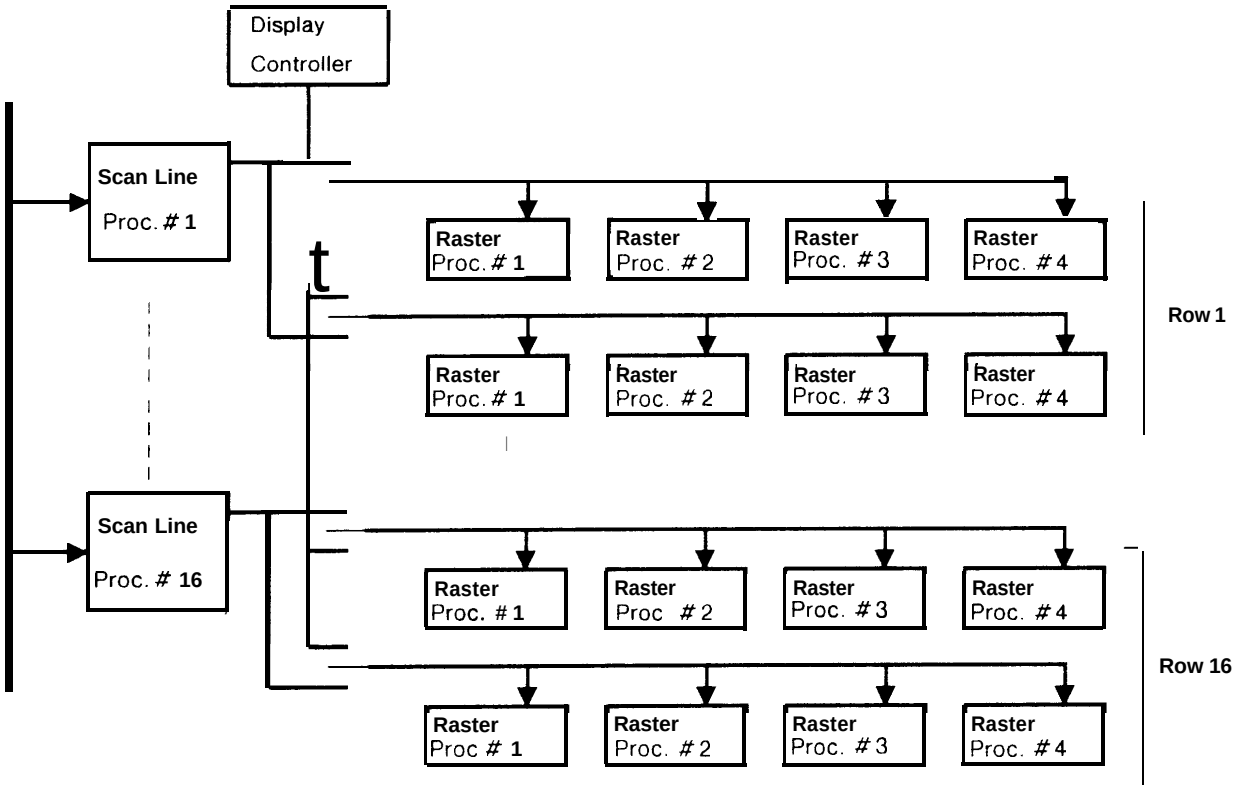


Figure 11

Double Buffered System

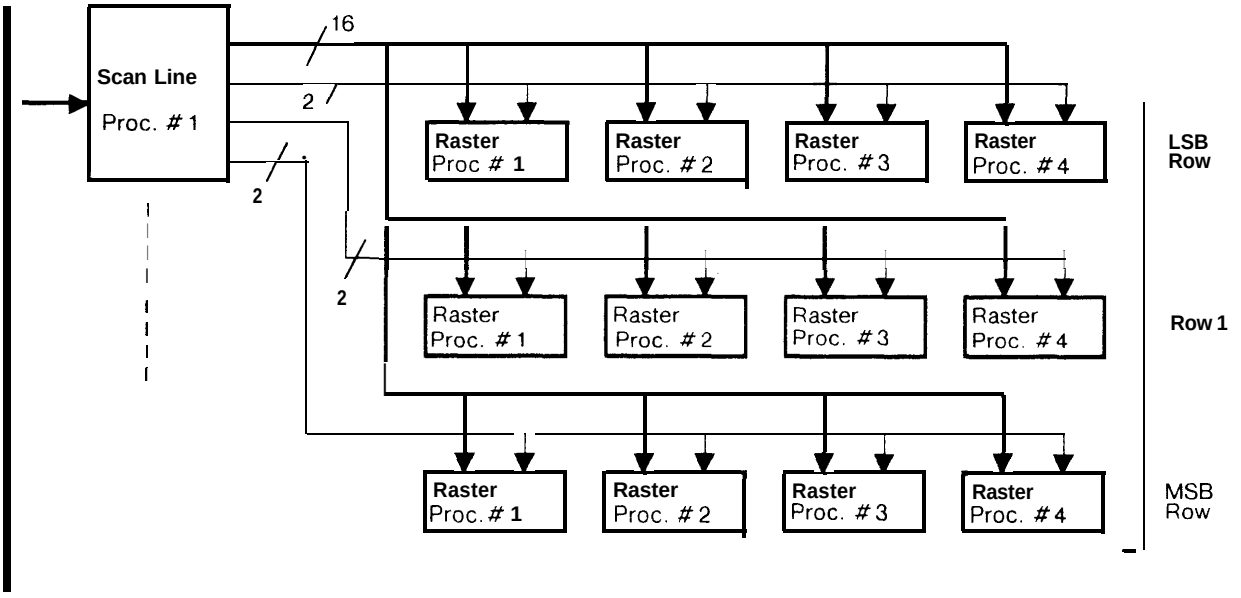


Figure 12

System With Three Bits Per Pixel

significant bit plane use the pattern inverted. This has the effect of placing a 6 in all pixel locations where the halftone pattern is 1 and 5 in all pixels where the halftone pattern is 0.

7 Comparison With Existing Architectures

It is instructive to compare the maximum attainable performance of this new architecture (as shown in Figure 4) with that of other architectures which have been designed to perform the same function. Any such comparison can not be "fair" because many of the processor architectures have functionality not available in the others. The author has picked one dimension shared by all designs and compared them along that dimension, while keeping in mind that other aspects of the architectures are being ignored.

In this comparison, only the time required to rasterize polygons, lines and text is addressed. The size of the raster is taken to be 1024 by 1024 pixels, and each pixel is assumed to be represented by one bit. Consequently, only the features relevant to this computation will be discussed. This comparison is not meant to establish which architecture is best, because that is highly dependent on the application to which the architecture is being put. and the relative importance of the features. For example, in some applications, such as computer generated animation and special effects, speed is not even the primary concern. In such applications the quality of the image (e.g. antialiasing, smooth shading) is much more important.

Furthermore, it is impossible to compare the alternatives by measuring the "real" time needed to perform the various rasterization operations because many of the designs have a speed potential which has not been realized in hardware but which could easily be accomplished without any conceptual breakthroughs. Thus, the designs are compared on the basis of a standard of minimum achievable "memory cycle time", that is, the number of accesses to memory which are necessary to place the rasterized representation of the image in the memory. This, of course, assumes that the bottleneck is always the memory access time and that the processing takes little time as compared to the memory access time, or that it can be hidden by the memory access time. The author believes that this is a realistic assumption and that it provides a good measure of the fundamental performance limitations of the various approaches to rasterization.

Table 1 compares the performance of the Raster Processor against that of five other architectures which rasterize graphical primitives. These architectures are briefly discussed below. The minimum number of memory cycles that each of the architectures requires to rasterize the indicated graphical primitives is listed.

The conventional approach to raster displays has been to implement a frame buffer (subsequently referred to as the FB architecture). This is typically no more than some part of the main computer memory of a conventional Von Neumann computer where each pixel is assigned a unique address. One memory cycle time is required for each pixel access.

In order to take advantage of the fact that many displays do not require more than one bit per pixel

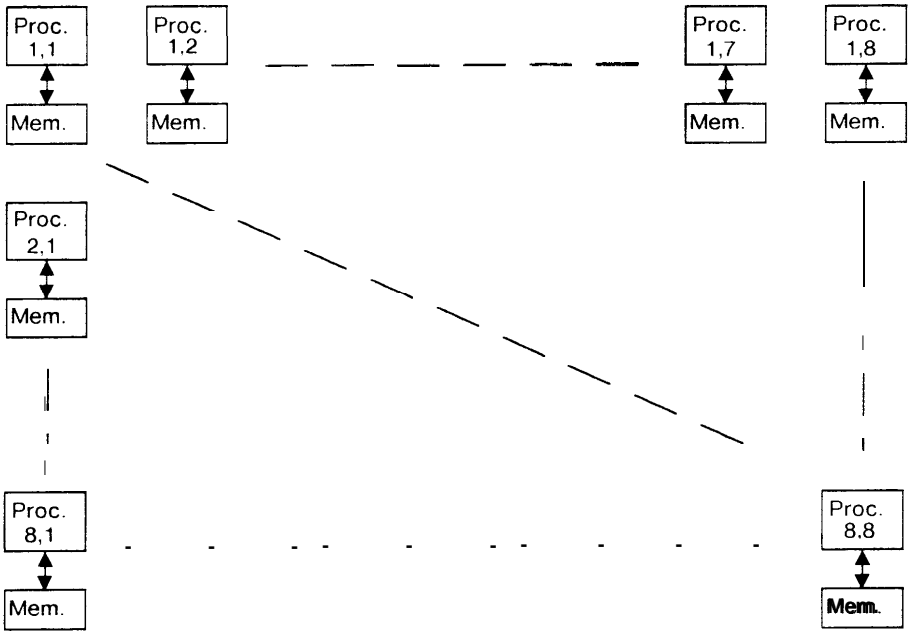


Figure 13
8 by 8 System Block Diagram

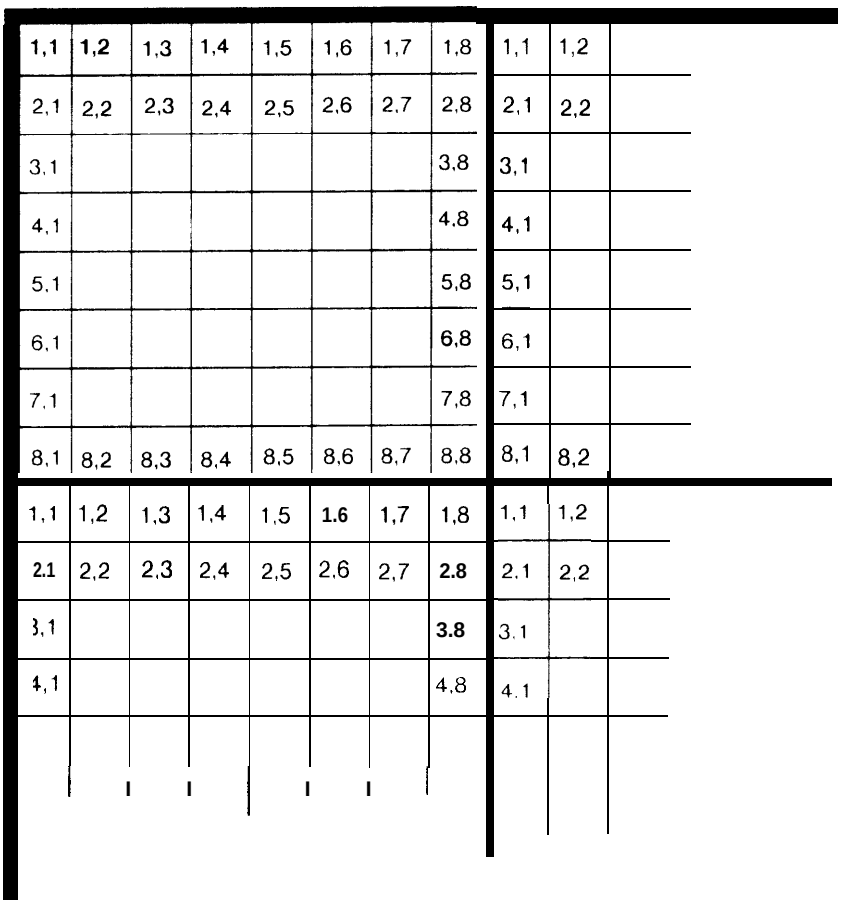


Figure 14
Assignment of Screen Pixels to Processors
for the 8 by 8 Architectures

(i.e. the pixel is either black or white) some architectures have grouped these pixels into "words" (of 16 bits each typically) of main computer memory. This allows 16 pixels to be accessed simultaneously (subsequently referred to as the 16FB architecture). This scheme has typically been augmented by special hardware capable of filling the memory at the full memory bandwidth either by stealing main processor memory cycles [TM 82] or by employing a separate bulk memory which is maintained by a simple processor capable of accessing any arbitrary 16 contiguous horizontal pixels in one cycle [BB 80]. This approach is up to 16 times faster than the FB when large areas are to be filled. However, the edges of primitives (e.g. polygons) typically do not fall on 16 pixel boundaries, hence memory accesses sometimes occur which do not alter all 16 bits. In fact, one pixel wide lines often require that only one pixel be modified for each 16 pixel block which the line intersects. In such cases, the 16FB is no faster than the FB.

In recognition of this bottleneck, some suggestions have been made to make the process of rasterization more parallel. Two approaches ([GS 81], [CH 80]) have been basically organized as shown in Figure 13. Basically, these methods use an array of 8 by 8 conventional processors each connected to some bulk memory (referred to as the 8by8 architecture). Each processor is responsible for every 64th pixel of the image (as shown in Figure 14) and is connected to the memory representing those pixels. These processors are capable of executing special purpose instructions for filling their part of the memory. In effect, this approach can be thought of as 64 processors each managing an image which is 64 times smaller than the total image. During each memory cycle, the 8by8 approach can access at most 64 pixels. As with the 16FB however, primitives which do not set large blocks of contiguous memory simultaneously (such as one pixel wide lines) do not achieve the full parallelism.

In order to decrease the rasterization time even more, architectures have been suggested which contain some significant amount of computation at each pixel (e.g. [FP 81]). These methods are commonly referred to as the "processor-per-pixel" approach (referred to as the PperP architecture). In principle, this architecture is clearly the fastest since any raster operation (e.g. filling polygons, drawing lines, placing fonts) can be accomplished in one cycle time if the processors are sufficiently complex. However, because a typical image of 1024 by 1024 pixels has more than one million pixels (and consequently one million processors), the processors and their interconnections have to be kept rather simple.

In order to meet this constraint, the architecture described in [FP 81] uses a simple serial processor at each pixel capable of isolating the pixels which fall within a convex polygon in 20 cycles for each edge of the polygon (independent of the total area of the polygon). Even though the processor has been simplified extensively, the amount of circuitry per pixel is still significantly bigger than that required in the FB, 16FB or the 8by8 methods because these methods take advantage of high density dynamic RAM to store the raster and because the processing elements are shared among many pixels. Thus. this approach differs from all of the previously described ones because the amount of hardware required (i.e. the number of chips) is much larger, and the cost is much higher.

It is important to note that the [FP 81] PperP design is primarily intended to rasterize three

Table 1

Comparison of the Number of Memory Cycles Required by Six Rasterization Architectures to Image Various Shapes

Shape	FB	16FB	8by8	PperP	RECT	Raster Proc. System
Horizontal Line Length = 1024	1024	64	128	60	1	1
Horizontal Line Length = 128	128	8	16	60	1	1
Vertical Line Length = 1024	1024	1024	128	60	1	64
Vertical Line Length = 128	128	128	16	60	1	8
45 deg. Line Length = 1024	724	724	91	60	724	46
45 deg Line Length = 128	91	91	12	60	91	6
Axis Aligned Square 1024 x 1024	1048576	65536	16384	80	1	64
Axis Aligned Square 128 x 128	16384	1024	256	80	1	8
45 degree Square 128 x 128	~16384	~1024	~256	80	182	12
45 degree Square 16 x 16	~256	~16	~4	80	24	2
128 edge Circle Approx. R = 64	~12868	~804	~202	2560	128	8
Equilateral Triangle Edge Length = 128	~7095	~443	~110	60	111	
Character 16 x 8	128	8-16	2	?	~16	1
Character 32 x 16	512	32-48	8	?	~32	2
Character 32 x 64	2048	128-256	32	?	~64	8

NOTE: The PperP is the only processor which does not use a bulk memory. Hence, the number of "memory cycles" is actually the number of cycles required by the serial pixel processors to image the given shape.

dimensional images and is capable of performing hidden surface elimination and smooth shading in color. However, only its ability to rasterize two dimensional primitives at one bit per pixel is compared here.

In an effort to minimize the size of the per-pixel circuitry, [Wh 82] has suggested building a two dimensional array of memory elements. Any rectangular subset of these memory cells can be set or cleared in one cycle time by specifying the starting and ending X and Y values (referred to as the RECT architecture). Thus, the per-pixel circuitry is reduced to a memory cell with two select lines (one for X and one for Y) (see Figure 15). Note that this cell is significantly bigger than a dynamic RAM bit cell (see Figure 1). Nevertheless, due to the large amount of shared circuitry on the edges of the array, the circuitry per pixel is much less than that of the PperP approach.

This architecture can image any rectangular area which is aligned with the axes in one cycle time. However, if it is expected that the polygons and lines will seldom be aligned on the axes (as is the case for arbitrarily rotated images, for example), the analysis in [Wh 82] indicates that this method is on the average only twice as fast as the traditional FB method for drawing lines, and the fastest method to rasterize arbitrary polygons using this architecture is to rasterize them one scan line at a time. Thus, the number of memory cycles required to rasterize any polygon is on the average equal to the number of scan lines which the polygon occupies. Furthermore, characters must be imaged by specifying a list of all rectangular areas needed to image that character. For a typical character, this will be equal to twice the number of scan lines which the character occupies (i.e. there are 2 black pixel runs per scan line in a typical character).

As Table 1 indicates, no architecture is clearly superior to the others for all of the graphical primitives listed. Of the previously known architectures, the PperP exhibits the best performance except for: (i) axis aligned rectangular areas (in that case, the RECT architecture is superior), and (ii) objects with many edges or with small areas (due to the fact that the PperP incurs a fixed cost per edge).

The Raster Processor architecture exhibits performance comparable or superior to the best previously known architecture (the PperP) without having to abandon the area cost advantages of a bulk memory architecture. Furthermore, the Raster Processor architecture is well suited for rasterizing all of the broad range of shapes listed, whereas the performance of the others degrades significantly for some shapes.

8 Effects of Increasing VLSI Density

The Raster Processor architecture is not only desirable for the present density achievable in VLSI but can be scaled up "gracefully" as VLSI densities increase. Furthermore. increasing the density of the design does not force degradation of the Raster Processor performance due to communication constraints.

In fact, the design can be scaled in different ways depending on whether cost or performance

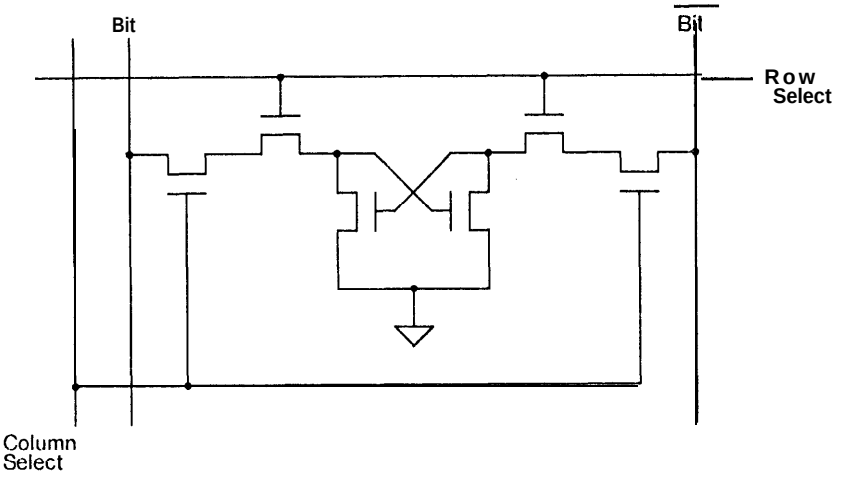


Figure 15
Memory Cell in RECT Architecture

optimization is desired. There are three principal ways to scale up the design of the Raster Processor: (i) the size of the memory array can be increased in both width and height, (ii) the size of the memory array can be increased in width only, or (iii) the Scan Line Processor can be incorporated into the chip along with all for the Raster Processors which it controls. Each of these approaches has its advantages.

If the size of the memory array is increased in both width and height, the number of transistors per pixel is decreased because the processing circuitry is shared among many more pixels. As a result, the chip area cost is reduced. However, this has the effect of decreasing the parallelism achievable, because the active processors become responsible for more scan lines. The number of pins on such a chip will not, however, be increased. The same chip interface can be used for memories of up to 8192 pixels per word (i.e. pixels per scan line) and up to 8192 words (i.e. a 64 megabit RAM).

If it is not desired to give up any of the parallelism, it is possible to increase the width of the memory array (i.e. the number of pixels per scan line) without increasing the height. This has the effect of keeping the number of transistors per pixel constant, hence keeping the area cost constant. Again, this does not increase the number of pins per chip. Note. however, that the width can only be increased up to the point where it equals the total width of the raster (e.g., for a 1024 by 1204 raster image, the maximum useful width for a Raster Processor is 1024).

If further integration is desired without giving up any of the parallelism, it becomes necessary to incorporate the Scan Line Processor on the chip along with a number of Raster Processors each of which is responsible for a number of full width pixel scan lines (see Figure 4). Since all Scan Line Processors are placed on a common bus, the number of pins on this kind of chip is equal to the number of pins required to communicate with the Scan Line Processor bus.

9 Continuing and Future Work

The next goal of this project is to build a prototype graphics system using the Raster Processor architecture in order to demonstrate its feasibility and to get experimental confirmation of the performance which has been projected for the design.

The first step is to fabricate a number of working Raster Processors in VLSI technology. For example, to build a double buffered 1024 by 1024 one bit per pixel display requires 128 working Raster Processors each controlling a block of 16K pixels (this is equal to the number of 16K RAMs that would be required to make a frame buffer of comparable size). Because of the difficulty of getting many prototype chips fabricated. it may be necessary to initially construct a system from as few as two processors, although the full parallelism of the architecture can not be exhibited.

It is feasible to build the Scan Line Processors (which control the Raster Processors) on a VLSI chip. This is not necessary for demonstrating the principle, however, since few of these processors are required for a complete system. The power of the graphics svstem is really derived from the Raster Processors. Therefore. the initial Scan Line Processor may be constructed from discrete

components.

Future research related to the Raster Processor will focus on methods of improving the design to include other desirable graphics features such as antialiasing, smooth shading, and hidden surface elimination. In principle any of the many well known scan line algorithms can be implemented in the Scan Line Processors. However. the challenge is to design these processors to generate a new scan line section every memory cycle time in order to take advantage of the full parallelism possible.

Research is also continuing on the study of other Smart Bulk Memory architectures. As was noted earlier, it is possible to replace the special purpose ALU on the Raster Processor with a more general purpose one bit ALU. It is then possible to make a programmable parallel processor capable of performing many tasks which were previously only possible by special purpose parallel architectures. Preliminary analyses have indicated that an architecture of this kind can be programmed to perform the function of self sorting arrays or content addressable memories by simply changing the program. Such an implementation would exhibit the same high degree of parallelism and high storage density (or low cost) realized with the Smart Bulk Memory Raster Processor.

10 Conclusion

Smart Bulk Memory architectures have a great unexplored potential. A special purpose architecture of this kind has been explored which is capable of rasterizing graphical primitives with performance comparable to that of the fastest existing architectures (i.e. the processor-per-pixel approaches). However, due to the high density storage of the image, the new architecture is much less expensive in integrated circuit area than other processors which exhibit similar performance. Preliminary analyses indicate that this type of architecture can be extended to perform many other parallel computations less expensively than previous approaches.

11 Acknowledgments

The author thanks Dr. David Cheriton and Mike Spreitzer for their many useful suggestions and discussions regarding this work. Dr. James Clark also read and commented on a version of this document.

References

[BB 80] Bechtolsheim, A.; Baskett, F.
High performance Raster Graphics for Microcomputer Systems.
In *SIGGR APH Conference Proceedings,* Seattle, July 1980.

[CH 80] Clark, J. H.; Hannah, M. R.
Distributed Processing in a High-Performance Smart Image Memory.
In *Lambda,* Vol I, No. 4, 1980, pages 40-45.

[Cl 82] Clark, James H.
The Geometry Engine: A VLSI Geometry System for Graphics.
In *SIGGRAPH Conference Proceedings,* Boston, July 1982, pages 127-133.

[FP 81] Fuchs, Henry: Poulton. John
PIXEL-PLANES: A VLSI-Oriented Design for a Raster Graphics Engine.
In *Lambda,* Vol. II, No. 3, 1981, pages 20-28.

[GS 81] Gupta, Satish; Sproull, Robert F.; Sutherland, Ivan E.
A VLSI Architecture for Updating Raster-Scan Displays.
in *SIGGR APH Conference Proceedings.* Dallas, August 1981, pages 71-78.

[KL 80] Kung, H. T.: Leiserson. Charles E.
Algorithms for VLSI Processor Arrays.
In: Mead. Carver: Conway, Lynn
Introduction to VLSI Systems, Section 8.3.
Addison- Wesley, 1980.

[NS 79] Newmann, W.; Sproull, R.F.
Principles of Interactive Computer Graphics, second edition.
McGraw Hill, 1979.

[TM 82] Thacker, C. P.: McCreight, E. M.: Lampson, B. W.: Sproull, R. F.; Boggs, D. R.
Alto: A Personal Computer.
In: Siewiorek, D. P.; Bell. C. G.: Newell, A. eds
Computer Structures: Principles and Examples, Chapter 33.
McGraw Hill, 1982.

[Wh 82] Whelan, Daniel S.
A Rectangular Area Filling Display System Architecture.
In *SIGGR APH Conference Proceedings.* Boston, July 1982, pages 147- 153.

Appendix I

Raster Processor Interface Details

Input Command Format

Table 2 lists the interface pins of the Raster Processor. All communication to and from the processor is performed synchronously with the system clocks. All signals are expected to be valid at the end of each clock cycle. The clock cycle time is expected to be in the neighborhood of 100 ns.. The function of the signals is described below.

The major communication path to and from the processor is the set of 16 data lines (D0-D15). During the normal rasterization process, these 16 lines are used to send commands and data to the processor. During the imaging mode (i.e. when the image is being extracted from the processor), these 16 lines are used to carry the pixel values.

Each command to the processor is composed of four 16 bit values sent during four consecutive clock cycles. Some commands require that the Chip Select pin be enabled when the first word of the command is sent in order for that command to have any effect, while others do not. Figure 16 summarizes the format of the rasterization commands which are described in detail below. Note that the motivation for some of the command features only becomes evident when one considers the interaction among many processors operating together in a complete system.

"Rasterize" Command

The first command is the most heavily used one. Its purpose is to allow the ALU (described in the section titled Raster Processor Architecture) to operate on a section of a given scan line with a given pattern. The command specifies on which scan line the operation is to take place (Y, a 13-bit value), what the first selected X coordinate is (Xs. a 13 bit value) and what the last selected X coordinate is (Xe, a 13 bit value). It is required that Xs be less than or equal to Xe. Note that the selection includes Xs but does not include Xe. Furthermore, the command specifies a 16 bit halftone pattern which is to be used when operating on the selected area. The halftone pattern is repeated every 16 pixels independent of Xs or Xe. The operation that the ALU is to perform on the various input data is also specified (see Section titled Raster Processor Architecture).

"Set Processor Position and Interleave" Command

Clearly, each Raster Processor can only handle a small number of scan lines and pixels per scan line. For example, in the 16K-bit dynamic RAM, there are only 64 scan lines (i.e. 6 bits) and only 256 pixels per scan line (i.e. 8 bits). Why then, does one specify 13 bits for both X and Y

Table 2

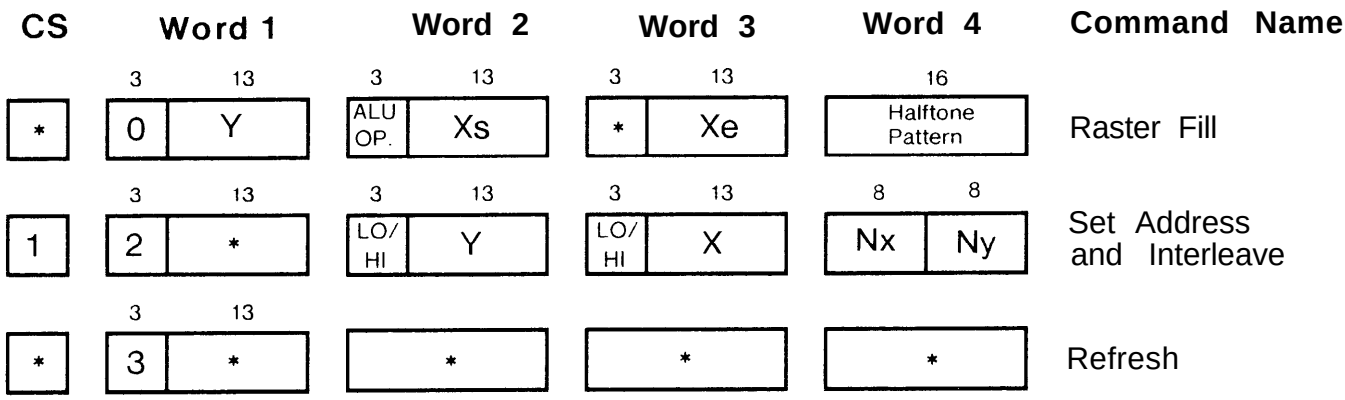
Pins of the Raster Processor

DO-015	16
Clocks	3
Power	3
co, Cl, C2	3
Chip Select	1
Total	26

Table 3

Function of Control Lines

co	Cl	C2	
0	0	0	Rasterization Mode: use incoming halftone pattern as is.
0	0	1	Rasterization Mode: invert halftone pattern before using it.
0	1	0	Rasterization Mode: ignore incoming halftone pattern and use all 0s instead.
0	1	1	Rasterization Mode: ignore incoming halftone pattern and use all 1s instead.
1	0	0	Imaging Mode: Clear Y display counter, latch the next scan line, and select the first 16 pixels for output.
1	0	1	Imaging Mode: Increment the Y display counter, latch the next scan line and select the first 16 pixels for output.
1	1	0	Imaging Mode: Output the next 16 pixels of the latched scan line during next clock cycle.
1	1	1	Imaging Mode: No Operation.



* - Don't care

Xs - Start of X range (inclusive)

Xe - End of X range (exclusive)

Y - Y Coordinate of selected scan line

Halftone Pattern - Pattern with which to fill scan line

ALU OP - Raster ALU operation:

0 - No Op
1 - Replace
2 - OR
3 - AND

LO/HI - Determine interleave:

0 - Use low order interleave
1 - Use high order interleave

Nx, Ny - Parameters for low order interleave.
Each specifies that the number of processors in the system is 2↑Nx or 2↑Ny respectively.

Figure 16
Raster Processor Commands

coordinates? In order to assemble a complete system, it is desirable to place many processors on a common bus. One approach is to connect the data bus of many of the processors for an image together. This destroys much of the possible parallelism, but it is a "low cost" approach. In such a situation, it is desirable for each processor to determine which commands pertain to the part of the image for which it is responsible. As a result, each processor only processes those requests which affect it.

This processor architecture provides 4 ways of splitting up the image space between a 2-dimensional , array of processors (2 possibilities in the X direction, and 2 possibilities in the Y direction). For example, assume that it is desired to make an image of 256 scan lines by 512 pixels per line out of 8 processors, each of which can handle 64 scan lines of 256 pixels per line. The processors are shown in Figure 17.

One approach (referred to here as high order interleave) is to assign each processor a contiguous area of the image as shown in Figure 18. It is simple for each processor to determine whether the current modification is to affect any of the contents of its memory. It must only compare all but the lowest 6 bits of the Y coordinate against the coordinate range that has been assigned to this processor. To get the internal scan line number, it is only necessary to use the low order 6 bits of the Y coordinate.

The other approach (referred to here as low order interleave) is to interleave the Y scan lines among the processors so that each processor is responsible for every 4th scan line (Figure 19). Again, it is simple for each processor to determine whether it is responsible for a given scan line. It must only compare the low order 2 bits of the Y coordinate against the low order two bits which it has been assigned. To get the internal scan line number, it is only necessary to shift the Y coordinate down by 2 bits. Note that if low order interleave is used, it is necessary to assign a number of processors equal to a power of two in the direction of the low order interleave. In this example, 4 processors must be assigned to the Y direction, even if 3 processors could contain all of the image.

Of course. this interleave method can also be applied to the X coordinates independently of the interleave chosen for the Y coordinates.

Thus, the second command in Figure 16 is used to set the processor's X and Y high/low interleave and the coordinates which this chip is to recognize. Note that this command is only recognized if the chip select line is enabled. Thus. it is possible to give this command to all processors and yet still have only one of them interpret it. This, of course, implies that the chip select line is not connected in parallel to any of the chips in a system.

'*Refresh'* Command

Dynamic memory arrays require that each capacitor of each row be read and re-written every so often (typically, every 2 milliseconds). Thus. it is up to the system generating commands to the

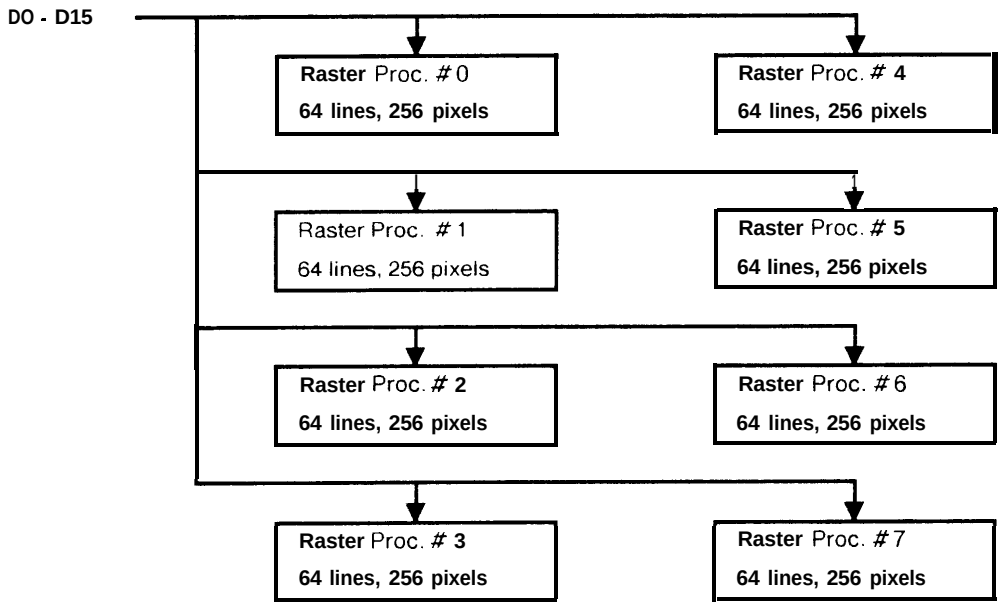


Figure 17

A 256 Line by 512 Pixel System

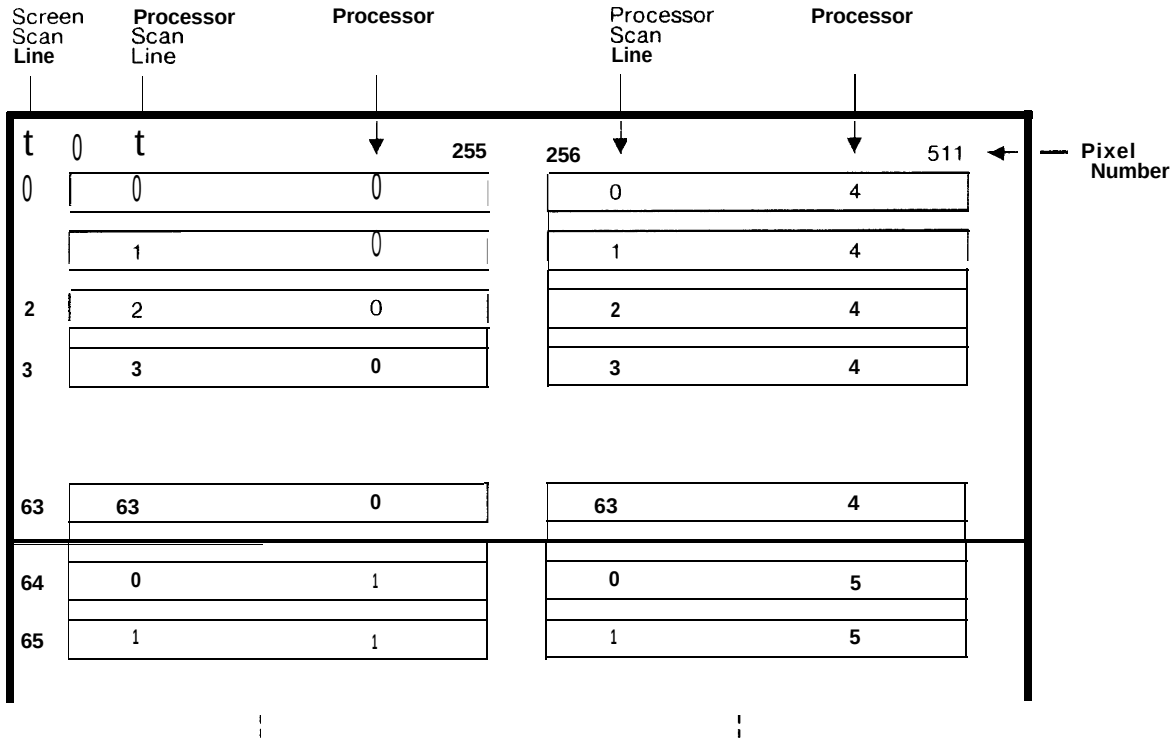


Figure 18

Effect of High Order Y Interleave

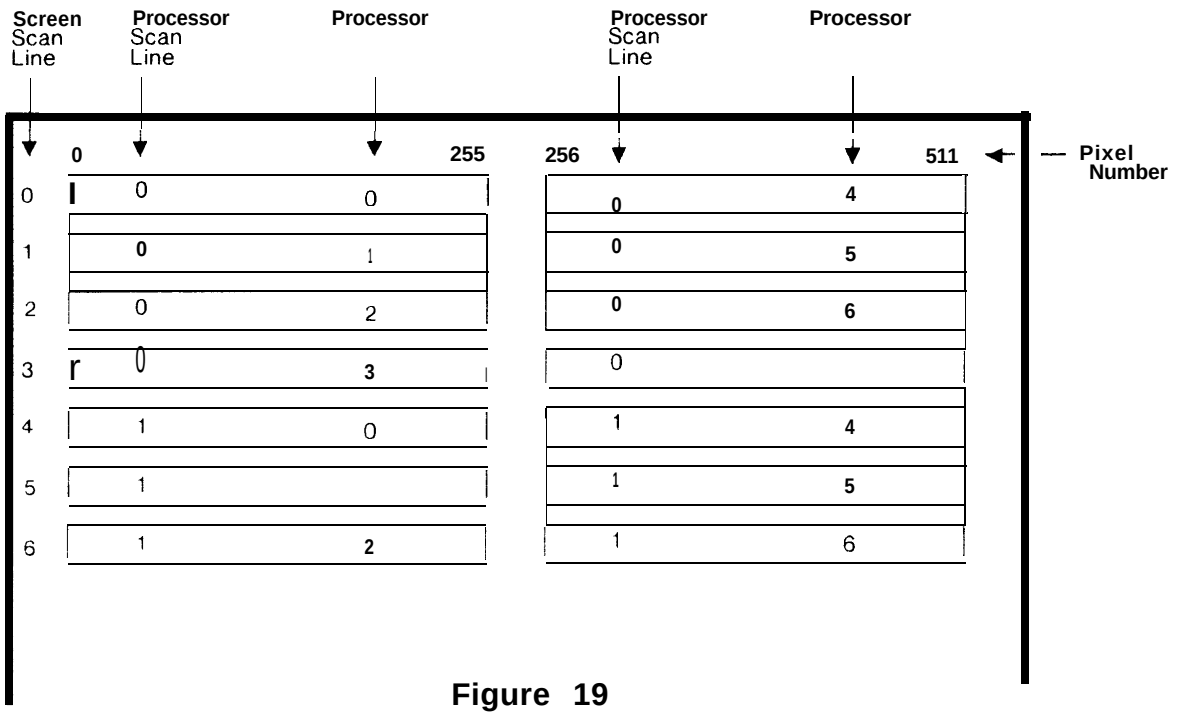


Figure 19

Effect of Low Order Y Interleave

Raster Processors to intersperse a number of "refresh" commands equal to the number of rows in the memory processor every 2 milliseconds. Internally, each processor possesses a refresh counter. Each time a "refresh" command is given, the refresh counter is incremented, its contents are used as a row address to the memory array (the scan line address) and a "no-op" ALU operation is done on that scan line, thus refreshing that scan line without modifying it. Note that if the system can guarantee by other means that all RAM rows are accessed every 2 milliseconds it is not necessary to ever use the refresh command. Note also that the refresh command must be used whenever the system does not have the next operation ready for the processors (as a "no-op" filler).

If, however, the memory array is static, then the only use for the "refresh" command is as a filler command, when no other command is yet available.

Display Control

A graphic display system has two major functions. One is to fill the frame buffer with the proper pattern, and the second is to display that pattern on a display device. As the resolution increases, the speed with which the pixels must be displayed increases also, since the frame refresh rate is kept constant (in order to avoid flicker). For example, if the image is to be displayed at 30 Hz and it is composed of 1024 by 1024 pixels, then the time to display each pixel is about 32 nanoseconds. Thus, it is essential to have a well designed method to extracting the image data from these processors quic kly.

In order to avoid having to "push" the speed of the technology, one must extract data from these processors many bits at a time. The data lines (D0-D15) can be used as output lines during the display cycle. It is expected that these chips will be used in double buffered applications where one set of chips is being filled with an image while the other set of chips is being displayed. Thus, the data lines of the chips whose contents is being displayed are not used for rasterization commands (Figure 11).

It is desired to be able to recall sequential 16 pixel sections from sequential scan lines. This can be achieved by in effect using the halftone distribution lines "backward" and selecting blocks of 16 pixels to scan out. Furthermore, it is necessary to have a counter which selects sequentially increasing scan line numbers. All of these functions are controlled by the Mode Control lines described below.

Furthermore, when the processor is in display mode and the memory array is not otherwise busy, it constantly performs refresh cycles if the memory is a dynamic RAM.

Mode Control Lines

Because the data lines are used to output data during display mode, three other lines are provided for controlling the display process. These lines are labeled CO, Cl, C2 (Table 3). In fact these three lines control the current mode of the processor (i.e. display or fill mode).

When CO is false, the processor is in the rasterization mode, and the data lines are used to send four word rasterization commands as described above. Lines C2 and Cl control the special ALU which processes the incoming halftone pattern. This feature is used for performing multi-value halftoning.

When C2 it true, the processor is commanded to go into the display mode. In this mode, as opposed to the filling mode, a command may be given every clock cycle (with the restrictions listed below) and the data lines D0-D15 are used as outputs. The 4 commands are described below. All commands must be accompanied by a true Chip Select line in order to be effective. If the chip select line is false. the data lines are turned off (tri-stated) and no operation is performed during that clock cycle. The 4 commands are:

a) Clear the Y display counter, perform a memory cycle to retrieve scan line 0, latch scan line 0. and select the first 16 pixels for output. Tri-state output lines during the next cycle. This command must be followed by 8 No Operation instructions. These instructions are necessary to allow the processor to finish any in-progress refresh cycle (up to 4 clock cycles) and then to recall the requested scan line (4 clock cycles).

b) Increment the Y display counter, perform a memory cycle to retrieve this scan line, latch the current scan line, and select the first 16 pixels for output. Tri-state output lines during the next cycle. This command must be followed by 8 No Operation instructions.

c) Place the next 16 pixels of the selected scan line on the data bus during the next clock cycle.

d) No Operation: Do nothing, enable or disable the output lines as indicated by the Chip Select line. If the lines are enabled, the currently selected set of 16 pixels is presented on the data bus.

Appendix II

Parallel Comparator Details

Given an N-bit number (herein after referred to as the N-bit number B) the parallel comparator is to generate 2^N outputs (numbered 0 through $2^N - 1$). Each output (defined as PCLT(j)) can be associated with one of the values of the N-bit number B and behaves as follows (See figure 20):

PCLT(j) is true iff j < B for j = 0, 1, 2, . . . 2^N-1

The comparator can be realized as the tree shown in Figure 21. Note that each node in the tree is labeled with 2 numbers: i and j. The first number i is the level of the node in the tree where the top "virtual" root is at level N and the bottom nodes are at level 0. The second number j is the distance of the node from the left of the drawing in its level (the leftmost node is 0). Let B be decomposed into its N bits as follows:

B = [B(N-1),B(N-2),. . . B(O)]
where B(0) is the least significant bit

Each node (i, j} takes 2 inputs from node {i + 1, Floor(j/2)}, 2 inputs b(i) and bc(i), and generates 2- outputs:

Inputs (and assumptions on inputs):
EQ{i+1, Floor(j/2)} true iff Floor(j/2) = [B(N-1) . . B(i + 1)]
LT{i+1, Floor(j/2)} true iff Floor(j/2) < [B(N-1) . . B(i+ 1)]
b(i) true iff B(i) = 1
bc(i) true iff B(i) = 0

outputs:
EQ{i, j} true iffj = [B(N-1) . . B(i)]
LT{i, j} true iffj < [B(N-1) . . B(i)]

That is, EQ{i, j} is true if j is equal to the number represented by the most significant bits of B (bits N-l to i), and LT{i, j} is true if j is less than the number represented by the most significant bits of B (bits N-l to i). Furthermore, it is necessary to assume that the virtual root generates outputs as follows:

EQ{N,0} is true
LT{N,0} is false

Then. by letting i = 0, it is clear that:

EQ{0,j} is true iffj = B
LT{0,j} is true iff J < B

Thus the outputs of the bottom nodes implement the desired function PCLT. That is:

$$LT\{0,j\} = PCLT(j)$$

It now only remains to derive the boolean equations relating the outputs of each node to its inputs. These relations are:

$$EQ\{i,j\} = EQ\{i+1, Floor(j/2)\} \text{ AND } (B(i) = LSB(j))$$
$$LT\{i,j\} = LT\{i+1, Floor(j/2)\} \text{ OR}$$
$$(EQ\{i+1, Floor(j/2)\} \text{ AND } B(i) = 1 \text{ AND } LSB(j) = 0)$$

where LSB(j) = least significant bit of j. Fortunately, it is possible to rewrite these relations more simply by noting that. for any given node, LSB(j) is a constant. Thus the above equations reduce to:

If LSB(j) = 0:

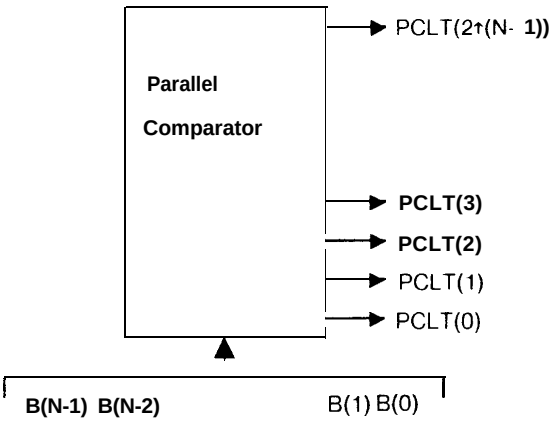
$$EQ\{i,j\} = EQ\{i+1, Floor(j/2)\} \text{ AND } b(i)$$
$$LT\{i,j\} = LT\{i+1, Floor(j/2)\} \text{ OR}$$
$$(EQ\{i+1, Floor(j/2)\} \text{ AND } b(i))$$

If LSB(j) = 1:

$$EQ\{i,j\} = EQ\{i+1, Floor(j/2)\} \text{ AND } b(i)$$
$$LT\{i,j\} = LT\{i+1, Floor(j/2)\}$$

By inspection, it is clear that these equations generate the desired results if the previously stated assumptions are made on the inputs. Thus, this parallel comparator can be constructed with $2^{(N+1)}-2$ nodes each of which implements the simple boolean relationships just described.

Examples



B = 0	B = 1	B = 2	B = 3
0	0	0	0
0	0	0	0
0	0	0	1
0	0	1	1
0	1	1	1

Figure 20
The Prallel Comparator

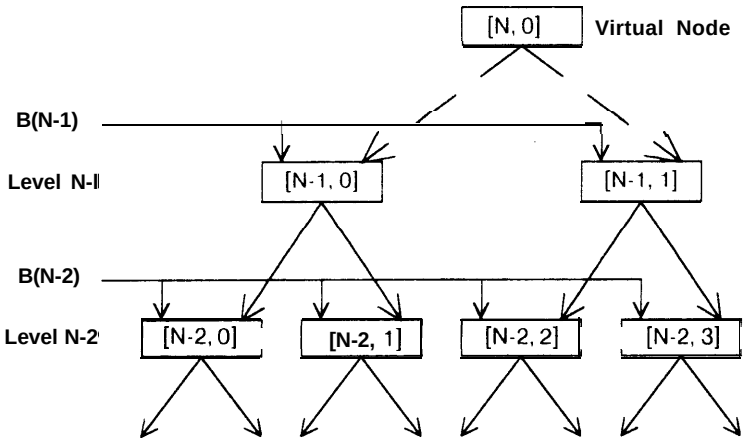


Figure 21a
The Parallel Comparator Comparison Tree

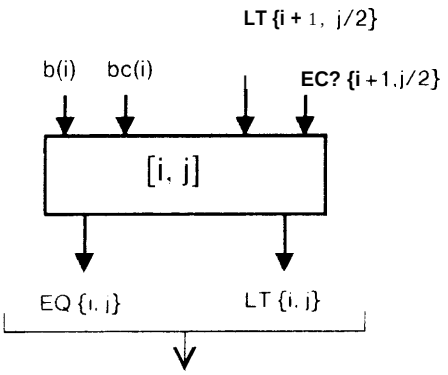


Figure 21 b
Comparison Tree Node Detail