(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 223 557**
A2

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **86308825.8**

(22) Date of filing: **12.11.86**

(51) Int. Cl.⁴: **G 09 G 1/00**

(30) Priority: **15.11.85 US 798755**

(43) Date of publication of application: **27.05.87**
Bulletin 87/22

(84) Designated Contracting States: **DE FR GB**

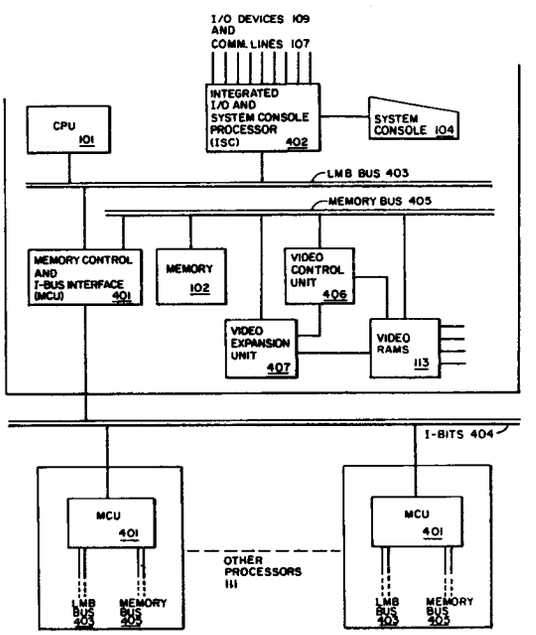
(71) Applicant: **DATA GENERAL CORPORATION, 4400 Computer Drive, Westboro Massachusetts 01580 (US)**

(72) Inventor: **O'Brien,Walter A., 66 George Hill Road, Grafton, MA 01519 (US)**
Inventor: **Rich,David L., 51 Church Street, Grafton, MA 01519 (US)**
Inventor: **Hurd,Charles C., 27 Woodleigh Road, Watertown, MA 02107 (US)**
Inventor: **Pogue,Michael, 1630 Worcester Road,Nr.328C, Framingham, MA 01701 (US)**

(74) Representative: **Pears, David Ashley et al, REDDIE & GROSE 16 Theobalds Road, London WC1X 8PL (GB)**

(54) Display control in a data processing system.

(57) In a distributed computer system, each CPU (101) accesses memory via an intelligent memory control unit (401) connected by a memory bus (404) to a local memory (102) and video memory (113) and by an interface bus to other memory control units (401) In the system. A video control unit (406) relieves the CPU (101) of much of the detailed work of modifying bit maps in the video memory (113). More specifically, in order to enhance the ability of the system to manage displays, especially in a technical graphics environment, where a single physical display supports a plurality of logical displays (windows), machine-language (graphics) instructions are provided which, in conjunction with logical display descriptors (forms) that describe each window, enable management and generation of display image data to be performed directly by the processing hardware of the digital computer system, minimizing the need for intervening software. Mechanisms are provided which gives the hardware the ability to defer complex protection and creation policies (faults) to operating system software. Data computed from the logical display descriptors may be encached, greatly enhancing the speed of consecutive operations on windows. Graceful creation is enhanced by permitting prosessing control to escape (trap) to software emulation handlers.



EP 0 223 557 A2

DISPLAY CONTROL IN A DATA PROCESSING SYSTEM

This invention relates generally to digital data systems, and more particularly to techniques for managing the display of data by such systems in an environment wherein a single display device may provide for a plurality of logical displays functioning independently of each other. Each logical display is known as a "window". Windows may all be displayed concurrently in their entirety, or some windows may be partially or completely covered by other windows.

Prior Art

References:

Our European Patent Application 85305716.4 (corresponding to U.S. Application 623908).

Graphics in Overlapping Bitmap Layers, Rob Pike, ACM transactions on Graphics, April, 1983.

SMALLTALK-80: The Language and its Implementation, Adele Goldberg and David Robson, Addison-Wesley, 1983. (Particularly

chapter 17, 18, 20).

Digital data systems have been equipped with display devices almost since their advent. The type of display that has taken pre-eminence as the most flexible for interfacing with the user is the cathode ray tube display. A recently evolved mode of the use of such displays is to permit several programs, or several processes to share the available space on a display, with each such program, or process being allocated a certain amount of display area. Each such area is known as a window.

Windows, then, may be thought of as independent logical displays co-existing on (or multiplexed onto) one physical display. An analogy is several sheets of paper on a desktop; they may be arranged so that all are simultaneously visible, or as they are manipulated some may completely cover (obscure) or partially cover (occlude) others. When obscured or occluded sheets are again uncovered, they still contain all the information that was temporarily invisible.

Windows on a display may likewise be manipulated so that some are sometimes partially or completely invisible on the display -- i.e., they present the appearance of being "covered" by other windows. A good embodiment permits the data in windows to be manipulated even while the affected windows or portions of windows are not visible on the screen, with subsequent "uncovering" revealing the manipulations that were performed on a window while invisible.

Windowing has heretofore been accomplished primarily by software. While such an implementation of windowing can provide

sufficient capability, it does so at the expense of computational overhead.

The user's requests, taking the form of software calls, must go through levels of interpretation by software in order to derive a series of machine-language instructions that the system can execute, even for wholly visible windows.

The software the user calls must be trusted software, since it requires a global perspective of all operations (both visible and invisble) performed to all logical displays. For example, one user may want to occlude another user's window; this requires reading the window to be occluded from the display memory and writing it ordinary system memory, then updating the appropriate display database to reflect the occlusion.

Operating system software is naturally trusted but usually results in a performance degradation, since more levels of interpretation are required to perform the user's request. Proper protection and operation can be provided, but only at a premium performace cost.

Another reason for the severity of this overhead is that the amount of descriptive information required for a window (logical display) is much greater than for a physical display.

Further, the descriptive information for each window must be completely reprocessed for every request -- there is no architectural provision for retaining (caching) the results of previous computations affecting those portions of the display not involved in a current change.

Summary Of The Invention

The present invention discloses a method of managing displays of a data system which includes users, a window manager, memory, a processor, a display, and a display interface. The processor is capable of executing machine language instructions that may directly (i.e., without intervening interpretation by software) manipulate displayed data. The method comprises providing a series of such instructions and providing a set of form descriptors, form descriptor identifiers, and operating system keys which describe ownership and the characteristics of windows on the display. The processor takes each such instruction along with the form identifier and operating system key of the user and in turn, tries to associate it with a form descriptor. If the processor discovers a match, it then determines the previous state of the display from the form descriptor, and assembles a new set of data to which the display interface is responsive to produce the modified display specified by the instruction. If an association can't be made, then a fault occurs to the operating system.

It is thus an object of the present invention to provide an improved data processing system.

It is another object of the present invention to provide data systems with the ability to efficiently manage windowed displays.

It is a further object of the present invention to provide data systems in which user-supplied instructions directly effectuate windowed displays with no need for intervening user software, that the windows can be changed (covered or uncovered) without user

software intervening.

It is an additional object of the present invention to provide data systems in which efficiency is enhanced by retaining the results of intermediate calculations relative to windowed displays, even when users are changed.

It is an additional object of the present invention to provide data systems in which protection is added to windows so that users are limited to their logical displays.

Other objects and advantages of the present invention will be understood by those of ordinary skill in the art, after referring to the description of the preferred embodiments and the appended drawings wherein:

Brief Description Of Drawings

Fig. 1 is a simplified block diagram of a prior art data processing system;

Fig. 2 is a simplified block diagram of the detail of block 113 in Fig. 1;

Fig. 3 is a more detailed block diagram of part of the system shown in Fig. 2;

Fig. 4 is a simplified diagram of a data processing system incorporating the invention;

Fig. 5 is a detailed block diagram of block 401 in fig. 4;

Fig. 6 is a detailed block diagram of blocks 406, 407 and 113 in fig. 4;

Fig. 7 is a block diagram showing data flow through the

graphics process board depicted in fig. 6 in pixel mode;

Fig. 8 is a block diagram showing data flow through the graphics process board depicted in fig. 6 in plane mode;

Fig. 9 is a more detailed block diagram of fig. 4;

Fig. 10 is a representation of a screen with a window;

Fig. 11 is a representation of a form descriptor;

Fig. 12 is an illustration of two windows, with one partially occluding the other;

Fig. 13 is a representation of the form descriptors for the windows in fig. 12;

Fig. 14 is a representation of pixel mapping which is used in the invention.

Detailed Description of Preferred Embodiments

Through out this document, the term GIS (Graphics Instruction Set) refers to previously mentioned U.S. Patent Application 623,908 (EP 65 305 716.4) ; whereas the term GIS II refers to this invention.

1 Discussion of Prior Art

1.1 System Prior Art

Referring to fig. 1, which is a block diagram of a typical

prior-art computer, the Central Processing Unit (CPU) 101 is the basic seat of intelligence in the computer and, as is indicated by its being depicted at the hub of all the other elements, is called upon to control all information transfers between those other elements.

CPU 101 is connected to memory 102 by memory bus 103, and must control all transfers over memory bus 103. System console 104 connects directly into CPU 101, which must control all transfers to system console 104. CPU 101 is connected to the external world by I/O bus 105, which connects to I/O controllers 108, through which transfers may be made to I/O devices 109; communications controller 106, through which transfers may be made to communication lines 107; and interprocessor controller 110, through which transfers may be made to other processors 111 comprising the distributed computer network. The controllers 106, 108, and 110 may be provided with some limited intelligence to control low-level details of transfers effected through them, but CPU 101 must provide all high-level control, setting up the controllers and overseeing returns of status information from them.

Alternatively, interprocessor bus 112 may be provided to interface with other processors 111; this may relieve some of the load on I/O bus 105, but does nothing to eliminate the problem of overhead on CPU 101.

Video RAMs 113 may be provided to contain "bit maps" of screen information for user terminals. CPU 101 provides bit map data and stores it in the RAMs in a form in which it may be displayed on user terminals.

1.2 Graphics Prior Art

1.2.1 Overview

Graphics memory is composed of thirty-two 64K Video RAMs (VRAMs) 113 and is organized into a 1K x 1K x 2 space. RS-343A monitor timing allows display of the entire array. A free-running blink clock selects one of two complete Palettes capable of mapping any pixel value to one of four levels of gray (0=black to 3=white). Palette I/O and other local operations are transacted through "Graphics Space", actually encoded as the IOC Auxilliary Processor (AP) Communication channel. In order to support the "Register-Transfer" function peculiar to VRAMs and additional diagnostic and VS boot-time character drawing, display memory timing and control logic 202 will arbitrate for the Memory Bus 103 as a requestor.

1.2.2 Video Memory

The 64K double-word video memory 113 is manipulated by the CPU as normal system memory. The screen is generated from a logical bit-map packed within a linear array of double-words which are ordered in the classical sense of left-to-right and top-to-bottom. Two bit pixels will be packed left-to-right with their most signi-

ficant bits toward the double-word most significant bit.

VRAM random access cycles are essentially identical with those of standard DRAMs. Their unique characteristic is the ability to transfer an entire 256 bit row of internal storage to a serial shift register 203 in one special access. This register may then be clocked independently of further random access activity. Additional controls allow multiplexing four 64 bit sections of this row register to aid in configurability.

1.2.3 Timing and Control

Dot and CRT timing is derived from a local oscillator operating at approximately 44 MHz. Due to the independent nature of VRAM serial clocking, no explicit synchronization with existing memory timing, other than the arbitration for register-transfer cycles, is required. Relatively simple multiplexing is all that is required to pass pixel values to the palette. The above capabilities can be satisfied by an intelligent micro-controller 206 (uC), the Intel 8051 being the best choice in that minimal cost and CEQs will result, although an 8031/2732 EPROM implementation is also possible.

1.2.4 Memory Bus Interface

A Memory Bus Arbiter 202 will grant the register-transfer, graphics, and palette cycles requested by the display memory uC. Standard protocol will be implemented in a PAL-based state machine.

ERCC syndrome bits will be generated logically for all reads both as an economical measure and due to the circumstance that VRAM outputs remain tri-stated during register-transfer cycles. Microcode will implement AP protocol via UABA references to write palette data and initiate any required commands.

1.2.5 Rotate and Merge Logic

In the course of analyzing the microcode necessary to implement the BITBLT instruction, a need was noted to accelerate graphics memory references on arbitrary bit boundaries. Consequently, the hardware required to implement this function as a Memory-Bus-resident device was developed: rotate and merge logic 205. A control bit specifies the direction of the merge sequence. A "merge-enable" bit is also available in order not to preclude a circular "rotate-only".

1.2.6 The Palette and DAC

The Palette 204 is organized as a 4 x 2 x 2 array arranged within a single double-word of storage. Two-bit Palette data written through the AP Graphics Space will encode the desired gray level to be associated with a given pixel value for each phase of the blink clock. Although direct reading of this register is not available, Microcode maintains an image of it in a single scratchpad location. The EDH13400 208 is actually a tri-DAC of which only the green channel will be driven. It not only performs sync-mixing, but is capable of direct 75-Ohm drive.

1.3 Prior Process

In figure 3, the system is seen to comprise a central processing unit (CPU 101), memory (102), video interface 320 consisting of video rams 113, video driver circuitry 324, and a video display 322. User software 307 and 309, which is resident in memory (102) may include all manner of software entities, including CPU instructions to manipulate the logical displays (311, 313, and 315) or software calls for windowing services. Such software calls invoke windowing software 317 also resident in memory (102) which interprets the user's calls and in turn may present to the CPU instructions which will result in carrying out the user's requests or call system software to carry out the user's requests. The windowing software 317 may interrogate the display databases 311, 313 or 315 to determine the previous state of the display, and will update the display database so that it reflects any changes brought about by the current call from user software.

Still referring to Figure 3, machine language instructions presented to CPU 101 by software are decoded by element 303 which, regardless of whether by "hard-wired" or microcoded means, directs arithmetic and logic unit (ALU) 305 in executing the instructions. The descriptive information in display databases 311, 313, 315 is then used to create a new screen bitmap 113, which would then contain a "screen image" of the display screen as it should now appear, reflecting the manipulations called for by the current calls from user software. Display interface 324 (regardless of

whether by programmed I/O or Direct Memory Access means) reads and processes the bit map to generate appropriate signals to display 322 causing it to display the information specified in bit map 113.

Note that the term "bit map" is used herein by convention, and may denote a character map for a character-oriented display, or a pixel map for a pixel-oriented display:

- On a character-oriented display, the smallest addressable element is a character position, which may be occupied by any character from a defined font of characters. The selection of which character is to occupy a particular character position is made by placing the binary code representing that character in the corresponding position of the bit map.
- On a pixel-oriented display, the smallest addressable element is essentially determined by the size of the "dot" that would be made to appear on the screen by the electron beam if it did not move. This is termed a "picture element", from which the term "pixel" was coined. In the simplest pixel bit map, a single bit position in the bit map represents each pixel position on the display; a "1" (one) at a position in the bit map denotes illumination "on" at the corresponding position on the screen, and "0" (zero) denotes "off". In more complex implementations, several bits in the bit map represent each pixel position on the display; a combination of several bits at a position in the bit map might denote the intensity level, or the color, or both, to be displayed at the corresponding pixel position on the screen.

The prior art approach can be successfully implemented, but it has the disadvantages of (1) introducing substantial overhead, because of the need for the system software to interpret all of the user's calls to effect protection or (2) allow all the users sharing the display direct access to the display, giving up protection.

2 Description of System Hardware

Referring to Figure 4, an overview block diagram of computers of the present invention employed in a distributed computing network, it is seen that CPU 101 is no longer configured at the hub of all the other elements. Over Local Memory Bus (LMB) 403, CPU 101 can communicate with integrated I/O and system console processor (ISC) 402, and memory control and I-Bus interface (MCU) 401, both of which contain sufficient intelligence to oversee their respective functions without close supervision by CPU 101. Without intervention by CPU 101, MCU 401 determines whether memory locations requested by CPU 101 are in local memory or not; if not, MCU 401 automatically performs the requested memory reference via I-Bus 404 in the memory associated with another computer on the network.

Communication between processors of the present invention configured as a distributed system is effected by memory references. All memory locations within the distributed system are accessible to any CPU -- a CPU may read from or write to a memory

location associated with another CPU on the distributed system with the same facility with which it may access any of the memory locations associated with itself. All memory access requests from a CPU 401 are passed over LMB bus 403 to MCU 401, which determines from the memory address whether the desired location is associated with the local processor (the processor containing the CPU and MCU) or one of the other processors comprising the network. If the former, MCU 401 accesses the local memory 102 (or video RAM 113, as appropriate) over memory bus 405 performing the requested read or write and obtaining data from CPU 101 over LMB bus 403 (if a write) or passing data to CPU 101 over LMB bus 403 (if a read). If the latter, MCU 401 passes the request over I-Bus 404 whence the MCU 401's of all other processors on the system examine the memory address; the processor having that address within its local memory performs the memory access, the data being passed over I-Bus 404 between the MCU 401 of the processor having the memory address and the MCU 401 of the requesting processor.

An arbitration scheme is provided to ensure that no processor can monopolize the I-Bus and that no processor can be deprived of the use of the I-Bus. This scheme is based on a rotating priority, wherein the processor that has just used the bus is given lowest priority and must wait till other requesting processors have used the bus before it can use the bus again.

Integrated I/O and System Console Processor (ISC) 402 contains a microprocessor and is provided to relieve CPU 101 of detail-level oversight of data transfers between the computer and I/O devices 109, communication lines 107, and system console 104.

LMB bus 403 is provided so that communication between CPU 101, ISC 402, and MCU 401 can take place without contention from any of the memory devices 102 or 113.

Video Control Unit (VCU) 406 is provided ahead of the video RAMs 113 to relieve CPU 101 of much of the detailed work of modifying bitmaps for controlling displays on user terminals.

Video Expansion Unit (VEU) 407 may optionally be provided to expand the pixel size from 8 to 24 bits. VEU 407 includes additional VRAM chips, but does not result in the creation of more VRAM locations-- it merely expands the size of the existing locations.

In the present embodiment, each computer is a 32-bit computer and is embodied on a single 15"x15" printed circuit board. Each board contains its own LMB Bus 403 which does not leave the board. Each board has a connection to I-Bus 404. Each board has a Memory Bus 405 which may leave the board and connect to optional expansion memory and video memory boards; up to 2 MBytes of memory may be accommodated on the processor board and are connected to Memory Bus 405; additional memory and video memory boards may be connected to the processor board's Memory Bus 405 to expand each computer's memory capacity.

Up to sixteen such computers (each with associated memory and video memory boards) may be accommodated in a single cabinet, the cabinet including a "backplane" comprising sockets into which all the boards are plugged, and permanent wiring interconnecting the sockets. I-Bus 404 is made up of backplane wiring and interconnects all the computers plugged into the cabinet to form a distributed computer network.

The sixteen computers may share a total memory space of 512 MBytes. As described above, any of the computers may access any location of the 512 MBytes, which may thus be regarded as a "global address space".

The Memory portion of the CPU contains the main memory control unit (MCU 401) and 2 Megabytes of main memory 102 itself. The MCU 401 also provides the control for an expansion memory bus 405 (called the MEM Bus) and the control for the global I-Bus 404. The MEM Bus 405 is also the connection for bit mapped video screens that are attached to the main memory address space. The only communication path between the CPU portion or I/O portion and Memory portion of the board is the LMB 403.

The Memory portion is entirely controlled by two gate arrays (see figure 5): CMOS-MEM gate array 561 and Bipolar-MEM gate array 562. These two gate arrays are basically traffic directors and error checking devices which control all the interactions that take place among the LMB 403, and I-Bus 404 and the MEM Bus 405.

The LMB 403 and the I-Bus 404 are the two busses that can initiate memory operations. The LMB 403 initiates all local memory accesses while the I-Bus 404 initiates all accesses of this particular node from other global nodes. The MEM bus 405 is essentially an internal bus to this memory portion which carries the actual address and data of the local RAM's themselves. This bus is "raw", unaligned, uncorrected data which is stored in the RAMs themselves. This MEM Bus has expansion capability so that up to 16 Mbytes can be addressed by this MCU (the two gate arrays) without adding more control. Thus, the MEM bus goes off-board so that

additional memory can be added either in the form of standard DRAMs or in the form of memory mapped graphics.

To illustrate the flow of a memory access, consider a CPU reference. The reference is initiated by the CPU via the LMB 403. The MCU 401 (combination of CMOS 561 and Bipolar MEM 562 gate arrays) recognizes the start of the memory operation. It then makes a determination of whether the reference was a local reference - i.e. to this node - or a global reference. Assuming it was local, the MCU generates the proper RAS and CAS (row address and column address) lines to access the required data. (The RAS and CAS lines are part of the MEM Bus 405). Either the memory array on the board itself (2 Mbytes) or an external expansion memory on the MEM Bus 405 will respond with the data. The MCU 401 now directs that data back onto the LMB 403 and signals the processor 101 that the data is available. If the data required aligning or correcting, the MCU 401 would have taken the data into the gate arrays themselves, manipulated it as required, and rebroadcast the data back onto the LMB 403 prior to signaling the processor 101.

Had the reference been global - i.e. not for this node, then the MCU would not have issued the reference on the MEM bus 405. Rather, the MCU would have begun arbitrating and re-initiating the reference onto the I-Bus 404. The responding I-Bus node 111 will return aligned, corrected data back via the I-Bus 404 at which time the MCU 401 will direct the data back onto the LMB 403, buffering the data as necessary.

3 Description of Graphics Hardware

Referring to Figure 4, VCU 406 provides high resolution color graphics (1280 x 1024), using 8 bits per pixel. Video Expansion Unit (VEU) 407 may optionally be included to expand the pixel size to 24 bits, giving the effect of a 24-bit VCU 406.

> Note: In the ensuing discussion, "VCU 406/8" shall mean that VEU 407 is absent and the pixel size is eight bits; "VCU 406/24" shall mean that VEU 407 is present and the pixel size is 24 bits; bald references to "VCU 406" shall apply regardless of pixel size.

VEU 407 includes augmentation of VRAMs 113; this does not provide additional VRAM locations, but expands the size of the existing locations from 8 to 24 bits. VCU 406 drives a 60 hertz non-interlaced 19" color monitor. The video outputs to the monitor are RGB (RED-GREEN-BLUE) sync-on-GREEN with 75 ohm drive impedance.

Pixel data retrieved from VRAMs 113 are not written to the screen directly, but are input to a table lookup function. The table, contained in a separate RAM and known as a "palette", outputs a 24-bit number. Eight of the 24 bits are converted from digital to analog to provide the RED video signal, eight provide the BLUE, and eight provide the GREEN. There are thus 2**24 (two to the 24th power) or 16 million colors which can be displayed. 8-bit pixels can display any 256 of the 16 million colors at any given time; the selection of which 256 may be altered by reloading the

palette RAM. 24-bit pixels can display any of the 16 million colors; the correspondence of pixel value to color may be altered by reloading the palette RAM.

VEU 407 must be used in conjunction with VCU 406 and connects to VCU 406 via 44 signals on the backplane. It provides an additional 16 bits per pixel bringing the total bits per pixel of the graphics display from 8 to 24. VEU 407, having circuitry analogous to that in VCU 406, will not be described in detail.

Pixel data from the host computer may be written directly into VRAMs 113, or may be combined according to various Boolean rules (discussed later) with data previously in VRAMs 113.

VCU 406 may be operated in "pixel mode" or "plane mode". Pixel mode provides more flexibility, since any of a great number of colors may be drawn at any screen position, but requires the host to forward every bit of every pixel of a desired display. Plane mode is provided to enhance performance, at the expense of limiting the number of colors that can be displayed for a given palette loading to 9 for VCU 406/8 or to 25 for VCU 406/24. Plane mode effects "planes" or "layers" of displays (eight planes for VCU 406/8, 24 planes for VCU406/24) wherein the color of each plane need be specified only once, "higher" planes may obscure "lower" planes, and the host need only send a single bit (denoting "ON" or "OFF") for each pixel position of each plane. For example, if it is desired to display a bar graph in which:

1) the background is blue;

2) a green grid is presented;

3) the bars are yellow; and

4) red labels may appear on the bars,

then it is necessary to:

1)

a) specify that the color of the background is blue by: (i) loading location 0 of the palette with a number that effects display of the desired blue; and (ii) clearing VRAMs 113 to all 0's, meaning that all palette lookups will access palette location 0 yielding the desired blue of the background;

b) load palette location 1 with a number that is displayed as the desired green for the grid lines;

c) load palette locations 2 and 3 with a number that causes display of the yellow desired for the bars;

d) load palette location 4, 5, 6, and 7 with a number that causes display of the red desired for the labels;

At this point, the VRAMs still containing all 0's, the screen will be entirely blue, the color specified in palette location 0;

2) provide one-bits (denoting "ON") to the least significant bit (the "lowest plane") of the pixels at positions corresponding to the screen positions comprising the desired green grid lines; (these pixels will then contain a value of 1, with the

result that palette lookups access palette location 1, yielding green; at this point the display will be a green grid on a blue background)

3) "OR" in one-bits to the next least significant bits of the pixels (the "second plane") at positions corresponding to all screen positions constituting the desired yellow bars; (these pixels will have values of 2 if ORed with a blue background pixel or 3 if ORed with a green grid line pixel-- in either case, palette lookup yields yellow. Thus, the green grid and blue background are not visible on the yellow bars-- those screen positions contain pure yellow, and not a superimposition or mixture of yellow, blue, and green.)

4) OR in one-bits to the next least significant bits (the "third plane") of the pixels at positions requisite to producing the desired labels. (These pixels will then have values of 4 or 5, either of which causes palette output of red.)

The desired bar graph is now displayed on the screen. Although some of the data is obscured (namely, the portions of the yellow bars that are under red labels; the portions of the green grid that are under yellow bars; the portions of the blue background that are under green grid lines) it is still present in VRAMs 113 and will again become visible when the overlaying data is removed. For example, if zero-bits are sent to the third plane, thus erasing the red labels, the yellow bars will again be fully visible with no need to reconstruct any portion of them; likewise,

if zero-bits are sent to the second plane to erase the yellow bars, the green grid will again be fully visible without having to reconstruct it.

3.1 Graphics Hardware Overview

Refer now to Figure 6. In the preferred embodiment, VCU 406 and VRAMs 113 are contained on a circuit board (Graphics Processor Board 301) which is a 15" x 15" 6 layer board with etch width of 8 mils and etch spacing of 8 mils. The board contains the following major components:

1) 64 256k VIDEO RAMS 113 with associated drivers and buffers.
2) Address mapping circuitry 602 for receiving addresses over Memory Bus 405 and translating same to physical addresses within VRAMs 113.
3) Data manipulating circuitry 603 for performing manipulations on data received over Memory Bus 405 or contained in VRAMs 113, and for storing manipulated data in VRAMs 113. Data manipulation circuitry 603 is mainly constituted by eight Graphics Data Processor (GDP) gate arrays 714 (on Figure 7).

The keyboard 611, mouse 613, and vertical blanking all interrupt the host processor via NMIs (non-maskable interrupts). The servicing of NMIs is very fast relative to normal interrupts because the host does not have to issue a VECT instruction, or

reschedule tasks upon receipt of the interrupt.

The mouse can interrupt the host as fast as every 33 milliseconds. Servicing of the mouse, which includes cursor plotting/replotting, should require no more 50 microseconds out of every 33 milliseconds of time. This is a total of .15% of the host's cpu time when the mouse is moving, which relatively speaking, is not often.

The keyboard constantly interrupts the host at an interval of 80 milliseconds. The time required to service the keyboard when it is idle is about 25 microseconds, a total of .03% of the hosts cpu time. If the keyboard is not idle, the time to service it is about 400 microseconds. With a typist who can type 60 words/minute (a word being 6 characters), the interrupt load is equal to about .12% of the hosts cpu time.

Vertical blanking interrupts can be used to pace the color palette updates. VCU 406 only updates palettes during vertical blanking. It takes 6 frames to fully update the 3 256-color palettes. Multiple attempts by the system to update the palettes in less than one frame will not be realized. The system can use the vertical blanking interrupt to indicate that VCU 406 has updated the palettes, and to issue another update if required. Since this function can be done via setting a flag, the time to service the interrupt is considered negligible.

Total load on the system, worst case, including both mouse and keyboard, is estimated to be approximately .27% of the total CPU time.

3.1.1 Pixel Mode Overview

Note: This discussion is in terms of 8-bit pixels; a discussion for 24-bit pixels would be analagous.

Referring to Figure 7, pixel data is sent from the host CPU over Memory Bus 405, is processed by the Graphic Data Processors 714, and eight-bit pixels are stored in "bit map" form in VRAMs 113.

Note:

- The term "host CPU" may refer to CPU 101 of the local processor, or some other node on the I-Bus.
- The "processing" performed by the Graphic Data Processors 714 may take the form of aligning pixels from the 32-bit bus word onto VRAM pixel boundaries, merging incoming pixel data with pixel data previously in VRAMs 113, and the like.

As is indicated by the bidirectionality of the arrow connecting GDPs 714 and VRAMs 113, GDPs 714 have the capability, in response to commands from the the host CPU, to extract bit map data

from VRAMs 113, perform manipulations upon it, and return it to VRAMs 113. Writing to VRAMs from the host is known as an "external" access"; the latter case is called an "internal" access.

Circuitry 604 extracts bit map data from VRAM's 113 and transforms each pixel to a desired video representation as directed by palette 609 under control of palette lookup table 605. Digital-to-analog converters (DACs) 606 transform the video representations to red, green, and blue video signals which are forwarded to the video monitor for display.

3.1.2 Plane Mode Overview

Plane mode data follows essentially the same path as pixel mode data, but is handled differently, as shown in Figure 8. With reference to the bar graph example set forth above, suppose that as part of writing the red labels on the yellow bars it is desired to write the character "A". A representation of the character "A" (element 815) is shown as it might appear in a "font" of characters (fonts, well known to those in the art, may be thought of as prestored bitmaps of often-used graphic entities). The prestored plane-mode (one bit per pixel) bitmap for "A" is shown as element 816. Where the bitmap contains a 1, the color denoted by foreground register 819 will be displayed at the corresponding screen position; where it contains a 0, the color denoted by background register 818 will be displayed. The discussion of the bar graph example in section 5.1 did not consider use of these registers; they enhance flexibility by permitting, for example, red labels on

a black background on the yellow bars. It is assumed here that the background register contains a value denoting the yellow of the bars and that the foreground register contains a value denoting the desired red of the labels.

3.2 Programming Overview

The video memory (VRAMs 113) contains video information which is continuously displayed on the screen. The smallest picture element that is addressable in the video memory is called a pixel. Each pixel contains information that corresponds to a value that the pixel can take on. For VCU 406/8, a pixel is represented by 8 bits of video memory, and can take on any one of 256 possible values. For VCU 406/24 a pixel is represented by 24 bits of video memory, and can take on any one of 16,777,216 (written as 16M for future discussion) possible values.

There are 1280 pixels displayed horizontally and 1024 pixels displayed vertically. There are actually 4048 VRAM locations horizontally, the last 768 of which are never displayed. This section of the video memory can be used to store temporary pictures, icons, character fonts, or small windows of data.

Each bit of the pixel is called a 'plane'. When configured as an 8 bit per pixel controller (VCU 406/8) there are 8 planes. When configured as a 24 bit per pixel controller (VCU 406/24), there are 24 planes.

The pixel plane bits are passed to the address field of a high speed RAM lookup table. The data returned by the RAM is then

passed to the video output stage. This table allows the pixel information stored in the video memory to be redefined before being displayed on the screen. This RAM is called the color palette.

On VCU 406/8, 8 bits of information are placed on the RAM address lines and 24 bits of data are returned. Because each pixel is 8 bits wide, it can take on any one of 256 values. The colors represented by the values can be chosen from a range of 16,777,216.

VCU 406/24, with respect to the color palettes, is essentially 3 VCU 406/8 designs in parallel.

3.3 Restrictions

VCU 406 is designed to read and write the video memory on pixel boundaries. To accomplish this, VCU 406 manipulates the data internally. Furthermore, on write cycles, VCU 406 performs a read-modify-write cycle internally. MCU 406 , the M-bus controller, is also capable of performing read-modify-write cycles, but cannot manipulate the data in the same manner as VCU 406. VCU 406 expects only simple reads and writes via the M-bus. If MCU 406 performs a read-modify-write to VCU 406, indeterminate results will occur.

To insure that MCU 406 executes only simple reads and writes to VCU 406 the following programming rules must be followed:

- For NORMAL space accesses to VCU 406 only even double word reads or writes are allowed. Specifically, odd double word reads or writes, byte reads or writes, and word reads or

writes are disallowed.

- o For OTHER space accesses to VCU 406, by definition, only even double word reads or writes are allowed. Specifically, odd double word reads or writes, byte reads or writes, and word reads or writes, are disallowed in the present embodiment.

3.4 Types Of Video Memory Accesses

VCU 406 is capable of the following types of video memory accesses:

1) Host processor to video memory accesses (and vice versa).
2) Video memory to video memory accesses.
3) Special character write accesses (from host processor to video memory.

The format in which the data is packed is BLOCK form - the 32 bit data word contains 4 consecutive 8 bit pixels for VCU 406/8; and 1 right justified 24 bit pixel for VCU 406/24 for video memory accesses. A special character write contains 32 consecutive pixels in a doubleword access and is independent of the bits per pixel.

3.5 On-board Graphics Hardware Caching

Video memory to video memory accesses (internal accesses) do not use the M-bus. During an INTERNAL READ data is read from the video

memory and stored in an on-board register. On an INTERNAL WRITE operation, the data that was previously stored in the on-board register from the INTERNAL READ is written back to the screen buffer, potentially at another address. Because internal accesses do not use the M-bus 403, the transfer of data is not bandwidth limited by the M-bus to 32 bits per access; but rather limited to the bandwidth of the video memory data lines. For VCU 406 it is 32 times the number of planes per pixel. This yields a transfer rate of 256 bits per transfer for VCU 406/8 and 768 bits per transfer for VCU 406/24. In both cases, this translates to 32 pixels per transfer.

3.5.1 X and Y, SOURCE and DESTINATION Registers

The SOURCE registers are normally used as pointers to a row and a column of the window to be moved or drawn. VCU 406 can use either pair of registers to read or write. This allows for easy handling of block moves and logic operations from video memory to video memory.

By providing seperate SOURCE and DEST registers for X and Y addresses, window moving in either the X or the Y plane can be accomplished much more efficiently because only one coordinate address needs to be sent after the initial X and Y address is loaded.

Note that both the SOURCE and the DEST registers can be used for reading and writing. Labels "source" or "dest" are used for

clarity only, although SOURCE register normally holds the top right coordinate of the window to be read and DEST register normally holds the top right coordinate of the window to be written.

3.5.2 Read/write Pixel

PIXEL operations may be used to draw lines, circles, and other pixel-by-pixel graphics efficiently. Pixel read and pixel write are just special cases of BLOCK READ and BLOCK WRITE respectively.

All accesses are double word accesses. However, if the pixel enable register contains a value of 00000001 (hex) then on a host write access, a pixel write is accomplished. The pixel to be written must always be right justified. Similarly, on a host read, the right most pixel of the double word read from the video memory is the pixel addressed, the rest of the pixels must be masked out by the host CPU. Note that the PIXEL ENABLE register is used by VCU 406 only on a write cycle.

PIXEL accesses require the setup of the following registers:

- o the X and Y SOURCE or DESTINATION registers,
- o the LALU register,
- o the PIXEL ENABLE register,
- o the PLANE ENABLE register.

3.5.3 Read/Write BLOCK

A double word access will access either 4 consecutive pixels on VCU 406/8 or 1 pixel on VCU 406/24. For VCU 406/8, each pixel is 8 bits wide, so 4 pixels will fit into a double word. For VCU 406/24 each pixel is 24 bits wide, so only 1 pixel will fit in a double word. This 24 bit pixel is RIGHT JUSTIFIED in the 32 bit double word. Note that, setting or clearing of the video memory is faster if a special character write access is performed instead.

BLOCK addressing may be used when it is required to move or draw a large rectangular blocks of graphics memory more efficiently. CHRBLT, 2DLINE,and BITBLT instructions use BLOCK transfers. Note that BLOCK accesses are NOT limited to double word boundaries of pixels, but are limited to pixel boundries.

BLOCK accesses require the setup of the following registers:

- o the X and Y SOURCE or DESTINATION registers
- o the LALU register
- o the PIXEL ENABLE register
- o the PLANE ENABLE register

3.6 VEU 407

VEU 407 is very similar in hardware design to VCU 406. The following paragraphs describe those differences.

There is no address generation, refresh control, or X and Y source and destination register sections on VEU 407. The address that goes to the video RAMs is passed via a cable to the VEU 407 board from the VCU 406 board.

There is no 8081 uP and associated circuitry (this includes the COM DATA register and COM STATUS register); the necessary video timing signals and data bus information are passed via a cable to VEU 407.

There are two video output chains (shifters, etc.) and twice as much memory (128 video RAMs) on VEU 407 as there is on VCU 406. Note that this memory does not yield any additional locations, but expands the size of the existing locations. There are only two DACs on VEU 407, one per video output stage.

There is no keyboard or mouse interface on VEU 407.

3.7 Converting VCU 406/8 to VCU 406/24

There exists a jumper cable that is connected on the backplane when both VCU 406 and VEU 407 are present in a system. This cable passes signals from VCU 406 to VEU 407 and vice-versa. One of the signals on the cable tells the VCU 406 board that a VEU 407 is connected so that it (VCU 406) configures itself appropriately.

The RED gun and GREEN gun cables are connected to the VEU 407 slot. The BLUE gun, keyboard cable, and mouse cable are connected to the VCU 406 slot.

4 Description of the Process

The prior art approach can be successfully implemented (as discussed above), but it has the disadvantages of introducing substantial overhead, because of the need to either (1) interpret all of the user's calls to effect protection while still sharing the display or (2) allow the user direct access to the display, giving up protection. The method of the present invention overcomes these disadvantages by

- o the construction of a data system in which special-purpose machine-language instructions are available for directly manipulating data in windowed displays, either logical or physical,
- o providing both privileged (protected) and non-privileged instructions which are directly executable by the CPU,
- o providing hardware faults into the operating system when protection policy decisions are required.

Note: CPU is meant to be the data processing system, and includes, but is not limited to, the parts of the data processing system depicted in fig. 9.

Since the user can use these instructions directly on the CPU, no intervening software is required to interpret them. Since they

execute directly in the CPU, which can include a highspeed memory for encaching descriptive information, there is no need to reprocess this descriptive information on every instruction. Thus the method of the present invention uses significantly less of the processing power of a given machine, or permits implementation of windowed displays on a smaller, less powerful machine than is possible for a prior art implementation.

The method of the present invention is depicted in fig. 4, but fig. 4 does not purport to depict all elements of a data system, but only those elements necessary to generate and produce displays.

The data system is again seen (see fig. 4) to comprise a CPU (101), memory bus controller and interface (401), a memory bus (404), memory (403), a video control unit (405), a video expansion unit (406), video rams (407) and a display (408). In this particular embodiment the display is a pixel-oriented video monitor and not a character-oriented terminal. A display interface is used that is appropriate to drive the video monitor.

User software is not constrained to software calls for manipulating the displays, but may now also contain instructions which will directly stimulate CPU 201 for that purpose. User software is always constrained to manipulate logical displays, which may or may not be on the physical display.

Some definitions are necessary at this point.

- User Graphics Instructions are non-privileged instructions that users can use to directly manipulate the logical display. The user is restricted to use these instructions only on

logical displays that the system software has specifically given the user access to by loading information into the CPU via privileged graphics instructions. Detailed description of the user graphics instructions can be found in Appendix A.

- Privileged Graphics Instructions are instructions that only privileged, system software can use. If a user attempts to use any privileged instruction (graphics or otherwise), a hardware-to-software fault is performed and the instruction is not executed by the CPU. Privileged instructions can only be executed from segment zero of Data General's MV series computers.
- A user's form identifier is an unique (in space) identifier, created by the system software, and used by both the hardware and the user to access a logical display (window). The form identifier is passed as a parameter with all user (non-privileged) graphics instructions.
- The hardware, when starting a graphics instruction, determines access based on the user's form identifier and the operating system's key. The hardware searches forms cache entry list, looking for a matching user's form identifier / operating system key pair. See Appendix B, section 2, for a more complete description of the form cache and how the user's form identifier is used.
- The operating system's key for this embodiment, is the Segment Base Register's (SBR) contents for that particular user's segment. This is a known unique value that both the operating system and hardware can use to safely determine which user is

using the CPU. The system software specifically loads the forms cache, using a privilege graphics instruction, with the triple (user form id, operating system key, address of the corresponding form descriptor).

- A form descriptor describes, or holds pointers to descriptors (such as attributes, cursor and rectangles) of a logical screen. The form descriptor contains such information as a pointer to the attributes block, the size of the window, and the location of the window on the physical screen, etc. See Appendix B, section 1 which describes the GIS II form descriptor and other related data structures for complete details. It should be noted that the form descriptor actually consists of databases used to describe the window (logical display).
- A window, when neither fully visible nor fully obscured, is broken into parts called rectangles. Each of these rectangles is describe by a rectangle descriptor. These rectangles descriptors are then strung together into a linked list, anchored off the form descriptor. Each rectangle (on the list) must be either fully visible or fully invisible. Background reading material on the use of rectangles can be found in [PIK]. Details on the rectangle descriptor can be found in Appendix B, section 1.
- Each window in the system can have a cursor. The cursor is usually used to point at specific objects on the screen (sometimes called ICONs) to perform some operation on. For example, a typical operation is to move a window on the screen by first moving the cursor to point at the upper left corner

of the window to move, pushing a key, and moving the cursor (and window) to the new place on the screen. Efficient cursor management requires extra effort, since the cursor is NOT part of the window, but can be thought of as a plane over the window. The form descriptor for each window points to a cursor descriptor. For details of the cursor descriptor see Appendix B, section 1.

- A Graphics Fault occurs when the CPU detects that (1) a user's form identifier is not in the forms cache entry, (2) the current graphics instruction intersects a visible cursor, (3) an unknown attribute index is referenced, or (4) an user's source form identifier can't be found. This fault is the CPU's way of informing the system software of a condition that it can't manage, and needs system software to resolve. For details of the this faulting mechanism, see Appendix B, section 4.
- A Graphics Unimplemented Instruction Trap occurs when the user tries to execute a graphics instruction that hasn't been implemented by the CPU. When this occurs a trap handler, that was defined by the user, is executed. At this point the trap handler can call on other software to emulate the unimplemented instruction, or abort. This provides a graceful way to migrate graphics functions between hardware and software. See Appendix A for more details.
- System software sometimes called operating system software is one or more programs that can execute privileged instructions.
- The window manager is software that is responsible for man-

aging windows in the system. Some embodiments may or may not make the window manager a part of the system software, but rather allow it to make privileged calls to the system. Some embodiments may also provide parts of the window manager software (the portions that do not need to be trusted) as user runtimes.

Still referring to fig. 10, it is seen that user software 311, must make software calls to a piece of system software called the window manager 317, to produce form descriptors. This is usually not a frequent operation, and can be slow. A user may not provide her own form descriptors, but must have the window manager provide them for her. The user then uses a unique form identifier returned to the user by the window manager that represents the form descriptor created for her. This is necessary in a multi-user, multi-program, or multi-process environment to arbitrate among the various users, programs, or processes that are contending for window space on the same physical screen.

Each user graphics instruction (for writing into windows) are presented by user software (307, 309) to CPU (101) where they are interpreted by the microcode unit (303), which in response to each instruction fetches an appropriate sequence of microinstructions to direct ALU (305) in performing the instruction.

4.1 The Protection Mechanism and Caching

Each such user graphics instruction must contain a user form identifier. Also known to the CPU/microcode is the current operating system key for the user currently running on the CPU. (The operating system key was set when the user was given the CPU via the SBR management instructions. In a different embodiment, this key can be generated differently, as long as it uniquely identifies all users in the system and can be agreed upon by the hardware and system software). The microcode then searches the forms entry cache looking for a matching form identifier/operating system key pair.

If the id/key pair is not found in the form id cache, a forms cache miss fault is generated, basically calling system software to resolve the fault. The operating system must then search its list of valid forms for this particular user to determine if a new form must be loaded into the cache (using the WGLFORM instruction) or if the user attempted to reference an illegal form.

If id/key pair is found in the forms entry cache then the form descriptor address is gotten from the form id cache. The form descriptor address is then used to see if the form descriptor is encached in the form descriptor cache. (In the preferred embodiment, the form descriptor is actually encached both on the CPU 101 and the Video Control Unit 406. In other embodiments, the descriptor may be wholly encached on either the CPU or VCU, or not cached at all. Also note, that other descriptors, related to the form such as the rectangle, attribute and cursor descriptors are

also cached to differing degrees). If the form descriptor is not cached, then it is read from memory 102 into the form descriptor cache. After the form descriptor is in cache, the graphics instruction can proceed.

In the ideal embodiment, many forms, of several users will be encache at one time. This cache can be a high speed scratch pad memory used by the CPU, or it can be wholly implemented on a graphic processor board. At the other extreme, it is possible to provide (a less expensive) implementation which provides little or no caching, and always reads the form descriptor from memory. Some form of caching will be necessary to store the form id, operating system key, form descriptor address triples to avoid faulting to the system software on every user graphics instruction. This caching could be a known location in main memory, or high speed scratch pad space in the CPU.

4.2 Coordinate Systems and Conversions

The reading of the form into the internal cached form consists of calculating screen-global coordinates from window-local coordinates provided by the system in the form descriptor. (The user is not required to know where on the screen her window is located, and can only find out by asking the window manager software).

Referring to fig. 10, a window positioned on a screen is depicted. Within the window are a user-specified origin "O" (1007) (the user may specify coordinates in the window relative to this origin) and a point "P" (1008) at which the user may wish to

operate. Listed on fig. 10 are the global coordinates of the corners of the screen, and the global and local coordinates (relative to the upper left corner (ULC) of the window) of points of interest of the window: its four corners E,F,G,H.

In the system-supplied form descriptor, the system is required to specify the height and width of the window (in pixels), and the local coordinates relative to the user's origin "0" of the window's ULC. Note that this implicitly specifies the location of the origin "0".

Fig. 11 depicts a form descriptor in system form. The transformation spoken of consists in calculating and filling in the global coordinates of the window's ULC (1009 and 1010) and a pointer to global 0,0 (1007). Since the form descriptor can now be encached as a forms cache entry, there is no need to recalculate such information on subsequent references to the same window, unless the window's global ULC changes.

Every bitmap employs a global coordinate system within which individual windows are allocated. Each form, in turn, defines its own local (internal) system that is independent of the location of the window within the containing bitmap. Display-affecting instructions are written in terms of local coordinates; translation to global coordinates is performed when the instructions are executed. Thus, with reference to fig.s 10 and 11, a form descriptor contains both local (1103, 1104) and global (1101, 1102) designations of the form's Upper Left Corner (ULC), point E (1003).

Suppose that it is desired to address point P 1008. The encached form descriptor supplies vectors AE (E's global coordina-

tes 1101,1102) and OE (E's local coordinates 1103, 1104). Instruction inputs supply vector OP (P's local coordinates: XL,YL relative to the user supplied origin 1007). The vector AP is calculated internally.

The cache then always contains the transformed versions of the "n" most recently used forms descriptors. This can greatly accelerate execution, since in practice many forms will often be used repetitively. (The determination of the value of "n" is left to the designers of a particular embodiment).

Since bitmap 1007 contains an image of what is to be displayed on the screen, execution of the present instruction may be regarded as complete when the bitmap is updated to contain the new display information specified by the current instruction. Translating the bit map 1007 to a visible display is a function of display interface 324, the timing of which is asynchronous to the timing of instruction execution.

Note that bitmap 1007 is contained in VRAM's (video random access memory chips). Depending on the particular embodiment, the VRAM's may or may not be accessible to the ALU just as any other portion of main memory.

4.3 Clipping

The method will not permit a user to write outside of the window he has referenced. The system software is responsible for defining the size of the user's window, by specifing in the form descriptor the bounding rectangle 1105, 1106, 1107, 1108 that the user's

references must stay within. For example, if a user instruction specified a horizontal line 400 pixels long in window 1002 that is only 250 pixels wide, only that portion of the line that fits within the window is written to the window 1002, and the rest is ignored. This is known as "clipping". The user is NOT informed that the clipping has been performed; further, the user does not need to know this. If the user was to be informed, then parallel operation of the graphics subsystem would greatly be limited, since the CPU would have to 'wait' for the graphics instruction to nearly complete to determine if clipping information needs to be returned.

In prior art, it was necessary to know if the instruction was clipped, since this provided the software a hint that the graphics instruction left one rectangle 1203 of a window. If this was the case, the software would then have to restart the instruction on the next rectangle from the rectangle descriptor list. In the current invention, since the rectangle list is known to the CPU and encached, it can efficiently walk the rectangle list to complete the instruction without software intervening.

4.4 Occlusion: Broken Windows

A complication occurs in the described processing when a window is occluded (partially covered) by another window. Referring to fig. 305, it is seen that Window A (502) is partially covering Window B (501). Prior to this occlusion, the processing was able to regard Window B as a single whole entity, but must now regard it as a complex entity made up of several simple entities called

rectangles. Here we introduce the term rectangle to denote a portion of a window. Window B is broken into three rectangles: B1 (503), B2 (504), and B3 (505). In order to keep the simple entities as simple as possible, the constraint is imposed that rectangles must always be rectangular in shape -- i.e., the method does not allow consideration of the occluded rectangle and the L-shaped remainder, but requires division of the L-shaped remainder into two rectangles, B2 and B3.

The user is not required to know that this occlusion has occurred because it may have been caused by a different user, program, or process, and hence does not bear the burden of knowing about the panes -- the system software does this automatically and in a manner that is transparent to the user. The system software creates a descriptor describing each pane. This descriptor is known as a rectangle descriptor. Form descriptors describe the entire window, whereas rectangle descriptors describe only an area of the window. See appendix B, section 1.4, for a detailed description of the rectangle description.

When occlusion requires breaking a window into rectangles, the processing will create new rectangle descriptors for the rectangles, and will link them off the form descriptor for the window. In future embodiments this automatic creation may take place under control of microcode, but in the present embodiment it is done by software, invoked by the microcode-to-software fault capability, to be discussed further on.

A form descriptor for a window essentially comprises a list of pointers to the rectangle descriptors that make up that window. In

the current embodiment, these pointers are kept by using a linked list, anchored in the form descriptor for the window, using physical addresses. It is interesting to note that the form descriptor can easily change shape and size since a size field is part of the descriptor, and that only the operating system software uses the form descriptor, not user software.

When a pane becomes occluded, the data that were displayed on it are not to be displayed any longer, indicating that the corresponding locations in bit map (1007) will be overwritten by the data for the occluding window (fig. 12 window A 1202). Rather than discard this information, which would necessitate recomputing it later when the pane is to become visible again (forcing the user software to be aware that the window was partially or fully occluded), or when the user wishes to manipulate data within the pane (recall that it is possible to manipulate data in an invisible pane), it is retained in a virtual bitmap in main memory.

Referring to fig. 13, which depicts the forms descriptor 1301 for Window B 1201 that reflects the occlusion situation depicted in Fig. 12, it is seen that the window has been broken into three rectangles (or panes) 1306, 1308, 1310. The window manager software, when told to move window B 1201 over the top of window A 1202, divides window B 1201 into the proper number of rectangles, creating new rectangle data descriptors, allocating virtual bitmap space (more properly, virtual rectangles) and copying the data from the physical display into memory as necessary. In order to prevent a user from changing the window while it is being modified by the window manager, the triple form identifier / os key / form descrip-

tor address is purged from the CPU; then any references the user attempts to make will result in a cache miss fault. At that point the window manager (with help from system software) will pend the user's path (by not reloading the cache and resuming the fault) until the window manager has finished modifying the window.

Subsequent user instructions to manipulate window B can now be handled. For example, (referring to Fig. 12) if the user specifies a line from I to J in window B, the processing will automatically translate it to three operations: one for line I-I' in rectangle B3 1205, one for line I'-J' rectangle B1 1203, and one for line J'-J in rectangle B2 1206. Handling these three requests will, as usual, result in recording the line in the bitmaps for the three rectangles. However, since rectangle B1 is occluded, line I'-J' is written into the off-screen bitmap and will not presently be visible on the display. The CPU can easily find the three rectangle necessary to perform the three line draw functions, simply by following the rectangle list.

An optimization has been included, called the bounded rectangle which is described in the form descriptor 1105, 1106, 1107, 1108; which describes a rectangle which the rectangles on the rectangle must fully fill. In this way, the CPU can check once at the beginning to see if any portion of the graphics operation is in the window.

Subsequent removal of window A will result in returning the form descriptor for window B to its original shape, moving rectangle B1's bit map information from the off-screen bitmap onto the physical bitmap and collapsing the three rectangles into one and

thus the complete line I-J will be visible even though part of it was invisible when drawn.

4.5 Cursor Management

Referring to fig. 12 a cursor 1207 is shown. Since the cursor is not part of a user's window proper, users do not need to be aware of where it is. Consequently, drawing over the top of the cursor should not destroy it, or make it invisible. Users can still have 'user' cursors that are not allowed to leave the window. This is very useful when providing a terminal like interface within a window. In the form descriptor for a window is a pointer 1109 to a cursor descriptor 1303. For more details of the cursor descriptor see appendix B, section 1.5. Cursors may be visible or invisible; if invisible the CPU knows it can safely ignore any intersections.

When the CPU detects that a graphics instruction intersects the cursor, a fault is taken; allowing the system software and window manager to decide what to do about the cursor, and resume the instruction.

4.6 Microcode-to-Software Escapes

This is the mechanism that permits taking complex issues out of microcode and moving them into software, while leaving the way clear to construct improved future embodiments with more functions performed by microcode. It also provides a guide for implementation staging and for migration of functionality up and down. This

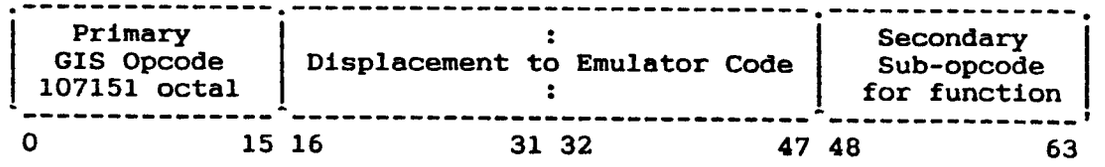
is performed by two mechanisms:

1) Microcode Faults: these are escapes into system software to deal with complex issues such as Form Cache Miss, Cursor Intersect, Unknown Attribute Index, and Invalid WGCHRBLT Source.
2) Unimplemented Instruction Traps: these are escapes into user software to deal with graphics instructions that may not exist in the current implementation. This allows the software to provide the richest possible instruction set, while migrating functionality up and down.

When an operation initiates, microcode must determine whether it can do the operation, and whether it can handle the forms on which it must operate. If the answer is no in either case, the instruction must fault to system software before it produces any side effects. A fault handler address is supplied in segment zero. The fault handler will determine what operation must be performed (load a form descriptor, moving a cursor, provide attribute information, or invalidate a reference), by using user and privileged instructions if necessary. The invoking program is resumed via WDPOP. For more details, see Appendix B, section 4.

Note in some embodiments although system software will receive the fault, it may then pass that information onto the window manager to handle. In this way, the window manager does not need to be a part of the system software proper, but is nonetheless, still a trusted part of the operating system.

When a graphics instruction starts, microcode must determine if the instruction is defined. If it isn't, an Undefined Instruction Trap (UIT) is performed and control is passed to the user provided Emulator routine. This routine's address is the 2nd and 3rd words of the instruction, which holds a program counter-relative offset (non-indirectable), of a software emulator/fault handler.



The process performs a LPSHJ to the routine. The emulator handler can then emulate the function, and return via an WPOPJ instruction. Recursive traps are thus supported implicitly.

4.7 Pixel Values and Palettes

The present embodiment uses a pixel value as simply an index into a palette. A palette is a special hardware map, which translates pixel values to (digital) beam intensities. Privileged instructions exist to set and retrieve pixel-to-color translations, i.e., load and store the palette.

Since the palette is a resource to be shared by more than one user, it must be protected by the system. This is accomplished by making the palette instructions privileged. System software is responsible for dividing the palette up into as many pieces as possible so that all users sharing the physical device can have as many useful 'colors' as possible. When the form descriptor is

created for the user, the range of possible palette entries that the user can use is specified by bit mask in the forms descriptor called the op mask.

When the user writes a pixel (via a write pixel or bitblt instruction) onto the physical bitmap, the CPU masks the users pixel bits by using this op mask, by masking the cooresponding pixel on the physical bitmap using the one's complement of the op mask, then ORing the user's pixel with bitmap pixel. If a combination rule was specified, or if the user specified a global operation mask, this is applied. (NOTE that masking is a transitive operation, leaving the designers to change the order). Optimizations are possible due to different combination rules and attributes the user has specified. See combinations rules and attributes in appendix A for more details.

An example seems necessary to clarify this. Suppose the user requires sixteen colors which is 4 bits/pixel, and that the physical bitmap has 256 colors which is 8 bits/pixels. Then it is possible to have 16 different windows, each with their own set of 16 colors using the same palette (16 * 16 = 256). This is shown in fig. 14. If we number the users from 0 to 15 (a four bit value), we can easily use that in the high half 1401 of each pixel to represent which 16 entry portion of the palette that user can have.

In the current embodiment, the system software must precharge, by using an WGRFLOOD instruction, each portion of the physical bitmap with the user number before the user can effectively use that area. In future embodiments, this could be moved into hardware, by including the user number to the forms descriptor.

A side effect of this mechanism is that when rectangles are occluded, and moved off the physical bitmap onto a virtual bitmap, only the exact number of bits per pixel that the user can use is required. Using the previous example, then when a rectangle is virtualized, it only requires 4 bits/pixel, half the space required when on the physical bitmap.

The palette can be split into asymetric partitions by giving different users different size masks. For example, if one user requires 32 colors, then it would have a five bit mask. Using this technique, the only real restriction is that the user always receives a power of two colors. This makes sense, since the user must always use a whole number of bits to represent a pixel.

4.8 Blinking

The present embodiment provides a blink clock. The blink clock provides a stimulus to switch between two palettes with a 50% duty cycle at a fixed rate of about 1.0-1.5 Hz. Entries in the two palettes are specified separately. This allows a given pixel value to alternate between two colors (or levels of intensity on a mono-chrome display). The chart below shows some of the effects possible when using this palette scheme for two-bit pixels.

Pixel Value	Phase-0 Palette	Phase-1 Palette	visual effect
00	black	black	off
01	dim white	dim white	dim
10	white	white	on
11	black	white	blink

4.9 Attributes

Attributes are stored in the attribute descriptor 1305, 1312 whose address is kept in the form descriptor 1110. Individual attributes are referenced by an index number into the attribute descriptor. Currently, all attributes are 32-bit quantities. These attributes are set by the WGWRATTR instruction, and read by the WGRDATTR which are non-privileged instructions.

Note: If an attribute in a Form Descriptor is changed while a GIS instruction is operating on that Form, the results are undefined.

Valid indices (and the attributes that they refer to) are:

0: Global Operation Mask
1: Global Combination Rule
2: Line Control Word
3: Line Foreground Color
4: Line Background Color
5: Linestyle
6: Character Control Word
7: Character Foreground Color
8: Character Background Color

For more details on the attribute descriptor see appendix A, section 1.1.

4.9.1 Combination Rules

Many of the instructions employ a combination rule to deal with superposition of source pixels on destination pixels. One does not always wish strictly to replace the destination pixel with the

source pixel; it may be desired to plant a pixel whose value is some function of source and destination. This implies that destination may also be an input.

In the context of a block transfer or line instruction, each bit in the source pixel VALUE is combined with the corresponding bit in the destination pixel VALUE, to form the new destination pixel VALUE. A standard Boolean function is used to specify the logical operation used for each bit position.

The operation mask selects which pixel bits are modified by the operation. A zero means "do not do anything to this bit," while a one means "operate on this bit using RULE."

The COMBINATION RULE is applied on a bit-by-bit basis to each bit in the source pixel and destination pixel. The RULEs are defined as follows:

Rule Number	Bits 28 29 30 31	Action	Interpretation
0	0000	dest = 0	Clear destination bits
1	0001	dest = dest AND source	--
2	0010	dest = source AND ~dest	--
3	0011	dest = source	Move source to dest
4	0100	dest = dest AND ~source	--
5	0101	dest = dest	NOP
6	0110	dest = dest XOR source	Set if unequal
7	0111	dest = dest OR source	Set pixels
8	1000	dest = ~dest AND ~source	--
9	1001	dest = dest XNOR source	Set if equal
10	1010	dest = ~dest	Complement dest bits
11	1011	dest = source OR ~dest	--
12	1100	dest = ~source	Move complemented
13	1101	dest = ~source OR dest	--
14	1110	dest = ~(dest AND source)	--
15	1111	dest = 1	Set destination bits

Those RULEs with a dash in the interpretation column are seldom used and have no particularly meaningful interpretation. (The

exclusive nor, XNOR, operation is also known as the equivalence, EQV, operation.)

When any PIXEL is operated on, the following equations are used to calculate the resulting destination PIXEL:

```
DEST_PIXEL := [OPERATION_MASK AND (SOURCE_PIXEL RULE DEST_PIXEL)]
            + [~OPERATION_MASK AND DEST_PIXEL]
```

4.9.2 Line Attributes

Linestyle word, combined with the Line Control word is a way of drawing other than solid lines. The line style word is specified as a bit string of length 32, it controls which color (foreground or background) to use when drawing the line. For each draw position, the leftmost bit in the linestyle is examined. If set, the foreground color (pixel) is planted; if clear, the background color (pixel) is planted. The linestyle is rotated left one bit position, and the next draw position is computed.

The line foreground color, line background color and line control word are set by using the WGWRATTR (write attribute) instruction. For more details on line attributes see Appendix A, section 1.

Examples of linestyle (expressed in hexadecimal) are:

FFFFFFFF: solid	AAAAAAAA: somewhat grey
F0F0F0F0: almost dotted	FFF0FFF0: long dashes
FFFF0FF0: long/short dashes	FFFFF060: dash-dot
88888888: very faint	FFFF0660: dash-dot-dot

The Line Control word is used to determine whether to use the foreground and/or background colors, and to join multiple lines.

In the current embodiment, only four line control characteristics are defined: suppress leading or trailing pixel, suppress foreground or background color.

4.9.3 Character Attributes

A font organization has been chosen that uses one bit per pixel bitmap. Character drawing is controlled in a manner similar to the linestyle process. There is a character foreground and background color as well as a character control word.

Conceptually, a character drawn by WGCHRBLT from the character bitmap (one bit per pixel) onto the actual target bitmap. As each bit is read from the character bitmap, if the bit is set the foreground color (pixel) is planted; if the bit is reset the background color (pixel) is planted onto the target bitmap.

The character control word is used to determine whether to use the foreground and/or background colors. For more details on character attributes see Appendix A, section 1.4.

4.10 Access Methods

By way of defining the instructions, graphics practice reveals that certain "favorite" operations are performed frequently, including:

- o reading or writing a pixel's value,
- o moving a rectangular area of pixels around,
- o drawing a line or series of connected line segments,

- o filling an polygonal area with a pattern, and
- o drawing a string of ASCII characters.

These operations comprise three major methods of drawing on a display: pixels, figures, and characters.

4.10.1 Character Access Method

This method provide ways to plant text in a window. The Character Block Transfer (WGCHRBLT) instruction allows for arbitrary font specification.

4.10.2 Figure Access Method

Drawing a line is an important part of technical computer graphics. It is used in CAD/CAM packages, architectural design packages, and business graphics packages. Since this operation is performed so often, special instructions are provided. Both continuous (LINESEG) and incremental (BRESENHAM_STEP) forms of line drawing are included. Lines can be drawn closed, half-open, or fully open. The actual algorithm must be reversible so as to make things such as line erasure precise.

4.10.3 Pixel Access Method

This access method deals with individual pixels and rectangular areas of them. It can serve as the foundation of higher-level

accessing methods, so that users can create their own display manipulation instructions (for image manipulation, conic section generation, etc). Read Pixel and Write Pixel operators allow direct access to pixels. Although only these two operations are strictly necessary to do the job, higher level operations are much more common.

These are the only drawing instructions that do not take a combination rule specifier. They are intended as the simplest of all building blocks. The model of use is one of many write operations to the same form in rapid sequence.

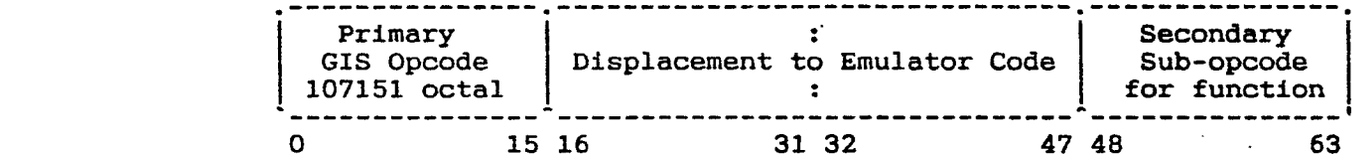
A Bit Block Transfer (often abbreviated 'BITBLT') operator is a very useful pixel-level operator. It is essentially a rectangular combination and assignment function. This is done especially when scrolling windows, moving windows around, creating and destroying windows that obscure other windows. A special rectangular fill operation is also useful for dealing with clearing screens and repartitioning windows.

BITBLT is the only operation that takes two forms, since certain restrictions are placed on source and target forms. Source logical pixels will be padded or chopped to conform to the target form's parameters.

4.11 Instruction Dictionary

All Graphics Instructions share a common instruction stream format. The first 16-bit word of all such instructions is octal 107151 (hexadecimal 8E69, Nova ADDOL# 2,0,SKP). The next two 16-bit words

hold a program counter-relative offset (non-indirectable), of a software emulator handler. The fourth 16-bit word contains a small, unsigned integer sub-opcode that specifies the particular function to be performed.



The instruction set is broken into two parts: privileged (system) instructions, and non-privileged (user) instructions. The following table lists the mnemonic, brief description and the decimal sub-code.

Privileged instructions:

WGLFORM	(load form)	16 decimal sub-code
WGPFORMS	(purge forms)	17
WGRDPAL	(read palette)	18
WGWRPAL	(write palette)	19

Non-privileged instructions:

WGRDPIXL	(read pixel)	20
WGWRPIXL	(write pixel)	21
WGRFLOOD	(rectangle flood)	22
WGPLINE	(poly line)	23
WGBITBLT	(bit block transfer)	24
WGCHRBLT	(character transfer)	25
WGRDATTR	(read attributes)	26
WGWRATTR	(write attributes)	27

See appendices A (non-privileged) and B (privileged) for detailed descriptions of these instructions.

Appendix A: Non-privileged Graphics Instructions

This document describes the initial set of non-privileged or 'User' GIS instructions as they will appear to a programmer using graphics. Some basic rules of GIS II are:

- o AC1 always contains a Form ID. This is not only consistent from the user's point of view but helps reduce microcode space by allowing common coding.
- o AC2 always contains a pointer to an instruction packet (if such a packet is defined). Again, this is consistent to the user and helps reduce microcode space.
- o AC3 is not used at all. This allows efficient invocation of GIS II subroutines from high-level languages that use AC3 to hold the frame pointer.
- o Attributes (such as line style, foreground color, etc.), are associated with the form and not with an individual instruction. This means that attributes are no longer provided with WGPLINE, WGBITBLT, and WGCHRBLT packets, but are set prior to issuing those instructions with the WGWRATTR instruction. This is the way existing graphics applications treat attributes. A benefit of this is that by restricting attribute processing to a single instruction, we further reduce microcode space.
- o All instructions that write to a FORM use the OPERATION MASK. WGBITBLT and WGCHRBLT use the destination FORM's OPERATION MASK only.
- o A majority of the GIS II instructions should be able to

execute on a parallel graphics processor. This dictates that no information is returned by these instructions.

A.1 Attributes

A.1.1 Attribute Block

Attributes are stored in the Form Descriptor and are referenced by an index number. Currently, all attributes are 32-bit quantities. These attributes are set by the WGWRATTR instruction

Note: If an attribute in a Form Descriptor is changed while a GIS instruction is operating on that Form, the results are undefined.

Valid indices (and the attributes that they refer to) are:

0: GLOBAL OPERATION MASK

1: GLOBAL COMBINATION RULE

2: LINE CONTROL WORD

3: LINE FOREGROUND COLOR

4: LINE BACKGROUND COLOR

5: LINESTYLE

6: WGCHRBLT CONTROL WORD

7: WGCHRBLT FOREGROUND COLOR

8: WGCHRBLT BACKGROUND COLOR

A.1.2 Operation Mask and Combination Rule

The operation mask and combination rule are attributes used to specify how pixels are to be combined for any instruction that writes to a FORM. These attributes are set by the WGWRATTR command.

In the context of a block transfer or WGPLINE instruction, each bit in the source pixel VALUE is combined with the corresponding bit in the destination pixel VALUE, to form the new destination pixel VALUE. A standard Boolean function is used to specify the logical operation used for each bit position. This function is called the COMBINATION RULE, or simply RULE.

The OPERATION MASK selects which pixel bits are modified by the operation. A zero means "do not do anything to this bit," while a one means "operate on this bit using RULE."

The COMBINATION RULE is applied on a bit-by-bit basis to each bit in the source pixel and destination pixel. The RULEs are defined as follows:

Rule Number	Bits 28 29 30 31	Action	Interpretation
0	0000	dest = 0	Clear destination bits
1	0001	dest = dest AND source	--
2	0010	dest = source AND ~dest	--
3	0011	dest = source	Move source to dest
4	0100	dest = dest AND ~source	--
5	0101	dest = dest	NOP
6	0110	dest = dest XOR source	Set if unequal

7	0111	dest = dest OR source	Set pixels
8	1000	dest = ~dest AND ~source	--
9	1001	dest = dest XNOR source	Set if equal
10	1010	dest = ~dest	Complement dest bits
11	1011	dest = source OR ~dest	--
12	1100	dest = ~source	Move complemented
13	1101	dest = ~source OR dest	--
14	1110	dest = ~(dest AND source)	--
15	1111	dest = 1	Set destination bits

Those RULEs with a dash in the interpretation column are seldom used and have no particularly meaningful interpretation. (The exclusive nor, XNOR, operation is also known as the equivalence, EQV, operation.)

When any PIXEL is operated on, the following equations are used to calculate the resulting destination PIXEL:

```
DEST_PIXEL := [OPERATION_MASK AND (SOURCE_PIXEL RULE DEST_PIXEL)]
            + [~OPERATION_MASK AND DEST_PIXEL]
```

The visible effect of applying a RULE during an instruction depends on the assignment of COLORs to the VALUEs of the source and destination pixels.

A.1.3 LINE Attributes

LINESTYLE together with the LINE CONTROL WORD are used to give a line a texture. LINESTYLE is a string of 32 bits that define a pattern (solid, dotted, dashed, etc.). As WGPLINE draws each pixel, it looks at each bit in the LINESTYLE, going from the most significant bit (MSB) to the least significant bit (LSB). If the selected LINESTYLE bit is 1 for a given pixel, then the pixel is planted with the LINE FOREGROUND COLOR. If the selected LINESTYLE

bit is 0 for a given pixel, then the pixel is planted with the LINE BACKGROUND COLOR. When the LSB of the LINESTYLE is reached, the processor returns to the MSB of the LINESTYLE. This process starts at the first pixel of the the first line segment, to the last pixel of the last line segment.

The LINE CONTROL WORD is used to suppress the FOREGROUND and/or BACKGROUND colors when drawing a polyline. It is also used to suppress the initial and/or final endpoint of the polyline. Suppression of a pixel means that it will be left unaffected by the instruction.

The structure of the LINE Control Word is:

Bit #	Description
0	Draw the foreground pixels if clear. Suppress the foreground pixels if set.
1	Draw the background pixels if clear. Suppress the background pixels if set.
2	Draw the initial point if clear. Suppress the initial point if set.
3	Draw the final point if clear. Suppress the final point if set.
4-31	Reserved for future use. Must be set to zero.

Note: If the line being drawn is a single point, i.e. all the endpoints of the polyline are the same coordinate, the point is drawn only if bits 1 & 2 of the LINE control word are both clear.

A.1.4 WGCHRBLT Attributes

The WGCHRBLT CONTROL WORD is used to suppress the FOREGROUND and/or BACKGROUND colors when character plotting.

The structure of the WGCHRBLT Control Word is:

Bit #	Description
0	Draw the foreground pixels if clear. Suppress the foreground pixels if set.
1	Draw the background pixels if clear. Suppress the background pixels if set.
2-31	Reserved for future use. Must be set to zero.

A.2 Traps and Interrupts

A.2.1 Unknown Instruction Trap

The GIS provides a mechanism for future expansion to include instructions in addition to those listed here. If the processor encounters a graphics instruction with an unknown sub-opcode or if it cannot process the instruction for some other reason, it performs an unknown-graphics-instruction trap. This trap makes the processor perform an LPSHJ instruction function. Then, the undefi-

ned graphics instruction can be emulated by software, or some other appropriate action can be taken. Return from software is by the WPOPJ instruction.

Each GIS instruction has an emulator associated with it. The address of this emulator is given as a PC-relative displacement in the instruction itself (Bits 16-47).

In the event that the GIS instruction is implemented in software instead of microcode, the processor pushes PC+4 on the wide stack. Calculates the effective address of the emulator routine. Loads the PC with the effective address. Continues sequential operation at the word addressed by the updated PC.

A.2.2 Interrupts

All GIS instructions are interruptable and restartable, with an interrupt latency of about 15 microseconds. When an interrupt occurs, a GIS instruction saves their current state on the wide stack, and takes the interrupt, storing the program counter value for the currently executing GIS instruction in locations 2 and 3 of segment 0. Control passes to the address specified by location 1 of segment 0. Bit 2 (IRES) of the Processor Status Register (PSR) is set to 1 when a GIS instruction is interrupted.

When interrupt service is complete, control passes back to the GIS instruction, which pops the saved state off the wide stack and continues with its execution. Bit 2 (IRES) of the Processor Status Register (PSR) is cleared.

A.3 The Instruction Dictionary

A.3.1 WGRDPIXL -- Read Pixel Value

WGRDPIXL [displacement to emulator]

Input

AC0: Unused

AC1: Form ID

AC2: Address of a WGRDPIXL packet (X,Y)

Output

AC0: Pixel value

AC1: Unchanged

AC2: Unchanged

Under the control of AC1, and AC2, reads the VALUE associated with the FORM specified by (X,Y) and returns that VALUE in AC0.

AC1 contains the key to the FORM DESCRIPTOR of the FORM containing the pixel.

AC2 contains a pointer to the X and Y coordinates of the pixel in the FORM.

Upon completion, the contents of AC1, and AC2 are unchanged. If the pixel with the given coordinates is in the specified FORM,

then AC0 contains the pixel VALUE as a 32-bit number, right justified and zero extended; otherwise, AC0 is unchanged.

DWord #	Mnemonic	Definition
1	PIXEL_X	A signed 32-bit integer for the X-coordinate location.
2	PIXEL_Y	A signed 32-bit integer for the Y-coordinate location.

A.3.2 WGWRPIXL -- Write Pixel Value

WGWRPIXL [displacement to emulator]

Input

AC0: Pixel value

AC1: Form ID

AC2: Address of a WGWRPIXL packet (X,Y)

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

Under the control of AC1, and AC2, writes the VALUE in AC0 into the pixel specified by (X,Y) of the associated FORM.

AC0 contains the right-justified VALUE to be written into the FORM.

AC1 contains the key to the FORM DESCRIPTOR of the FORM

containing the pixel.

AC2 contains a pointer to the X and Y coordinates of the pixel in the FORM.

This instruction uses the OPERATION MASK and COMBINATION RULE when planting pixels.

After execution, the contents of the accumulators are unchanged. This instruction has no effect if the pixel specified by (X,Y) lies outside the FORM, or if that pixel in the form is write inhibited.

DWord #	Mnemonic	Definition
1	PIXEL_X	A signed 32-bit integer for the X-coordinate location.
2	PIXEL_Y	A signed 32-bit integer for the Y-coordinate location.

A.3.3 WGRFLOOD -- Flood a Rectangle

WGRFLOOD [displacement to emulator]

Input

AC0: Pixel value

AC1: Form ID

AC2: Address of a WGRFLOOD packet

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

Under the control of AC1, and AC2, writes the VALUE in AC0 into a rectangular area of pixels in the associated FORM.

AC0 contains the right-justified VALUE to be written into the FORM. This is an nonnegative 32-bit integer.

AC1 contains the key to the FORM DESCRIPTOR of the FORM containing the pixel(s).

AC2 contains a pointer to the packet describing the rectangular area in the FORM.

This instruction uses the OPERATION MASK and COMBINATION RULE when planting pixels.

After execution, the contents of the accumulators are unchanged. This instruction has no effect on pixels lying outside the FORM, or on pixels on the FORM that are write inhibited.

If either the X_EXTENT or Y_EXTENT is set to zero, this instruction has no effect.

DWord #	Mnemonic	Definition
1	D_ULC_X	A signed 32-bit integer for the X-coordinate of the location of the upper-left-hand corner of the rectangle.
2	D_ULC_Y	A signed 32-bit integer for the Y-coordinate of the location of the upper-left-hand corner of the rectangle.
3	X_EXTENT	Width of the rectangle in pixels.
4	Y_EXTENT	Height of the rectangle in pixels.

A.3.4 WGPLINE -- Draw a poly line

WGPLINE [displacement to emulator]

Input

AC0: Number of lines to draw

AC1: Form ID

AC2: Address of a WGPLINE packet

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

Draws one or more line segments between the endpoints specified in the WGPLINE packet with the current LINESTYLE. As WGPLINE draws each pixel, it looks at each bit in the LINESTYLE, going from the most significant bit (MSB) to the least significant bit (LSB). If the selected LINESTYLE bit is 1 for a given pixel, then the pixel is planted with the LINE FOREGROUND COLOR. If the selected LINESTYLE bit is 0 for a given pixel, then the pixel is planted with the LINE BACKGROUND COLOR. When the LSB of the LINESTYLE is reached, the processor returns to the MSB of the LINESTYLE. This process starts at the first pixel of the the first line segment, to the last pixel of the last line segment.

AC0 contains the number of line segments in the poly line

AC1 contains the key to the FORM DESCRIPTOR of the FORM to draw the line segments in.

AC2 contains a pointer to a packet containing coordinates of two or more endpoints.

This instruction uses the OPERATION MASK and the COMBINATION RULE when planting pixels.

After execution, the contents of the accumulators are unchanged. This instruction has no effect on pixels lying outside the FORM, or on pixels on the FORM that are write inhibited.

DWord #	Mnemonic	Definition
1	X1	A signed 32-bit integer for the X-coordinate of the location of the first endpoint.
2	Y1	A signed 32-bit integer for the Y-coordinate of the location of the first endpoint.
3	X2	A signed 32-bit integer for the X-coordinate of the location of the second endpoint.
4	Y2	A signed 32-bit integer for the Y-coordinate of the location of the second endpoint.
.	X.	...
.	Y.	...
2n-1	Xn	A signed 32-bit integer for the X-coordinate of the location of the n-th endpoint.
2n	Yn	A signed 32-bit integer for the Y-coordinate of the location of the n-th endpoint.

A.3.5 WGBITBLT -- Bit Block Transfer

WGBITBLT [displacement to emulator]

Input

AC0: Source form ID

AC1: Destination form ID

AC2: Address of a WGBITBLT packet

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

Moves a RECTANGLE (a rectangular area of pixels) from the source PIXEL FORM to the destination PIXEL FORM, performing an OPERATION as it moves the pixels. If the source and the destination FORM IDs are the same, the transfer is performed is such a way that a pixel is copied before being written to.

AC0 contains the key to the source FORM DESCRIPTOR.

AC1 contains the key to the destination FORM DESCRIPTOR.

AC2 contains a pointer to the packet describing the rectangular area in the source and destination FORM.

Upon completion, the contents of AC0, AC1, and AC2 are unchanged.

A RECTANGLE is defined by a POINT specifying the upper lefthand corner and the X and Y EXTENT of the rectangle within a FORM. The RECTANGLE, then, is a list of 2 signed and 2 unsigned 32-bit integers. The X-coordinate is the first double word in the list, the Y-coordinate is the second, the X EXTENT is the third,

and the Y EXTENT the fourth. The source RECTANGLE must be contained within the source FORM and must fit inside the destination FORM when moved. If it does not lie entirely within one of the FORMs, the WGBITBLT instruction will cause the RECTANGLE to be clipped so that it does fit in either FORM. This instruction will not write to pixels on the FORM that are write inhibited.

If either the X_EXTENT or Y_EXTENT is set to zero, this instruction has no effect.

This instruction uses the OPERATION MASK and the COMBINATION RULE when planting pixels.

No pixel will ever be taken from outside the source FORM, or drawn outside the boundaries of the destination FORM.

DWord #	Mnemonic	Definition
1	D_ULC_X	A signed 32-bit integer for the X-coordinate of the location of the upper-left-hand corner of the destination rectangle.
2	D_ULC_Y	A signed 32-bit integer for the Y-coordinate of the location of the upper-left-hand corner of the destination rectangle.
3	X_EXTENT	Width of the rectangle in pixels.
4	Y_EXTENT	Height of the rectangle in pixels.
5	S_ULC_X	A signed 32-bit integer for the X-coordinate of the location of the upper-left-hand corner of the source rectangle.
6	S_ULC_Y	A signed 32-bit integer for the Y-coordinate of the location of the upper-left-hand corner of the source rectangle.

A.3.6 WGCHRBLT -- Character Block Transfer

WGCHRBLT [displacement to emulator]

Input

AC0: Source Form ID

AC1: Destination Form ID

AC2: Address of a WGCHRBLT packet

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

Copies a rectangular area of pixels from a 1-bit per pixel SOURCE FORM, expanding each pixel to a FOREGROUND COLOR or BACKGROUND COLOR, performing an OPERATION as it copies the pixels, in accordance with the values of the FOREGROUND SUPPRESS and BACKGROUND SUPPRESS flags.

Note: The SOURCE FORM must be 1-bit per pixel.

AC0 contains the key to the FORM DESCRIPTOR of the SOURCE FORM.

AC1 contains the key to the FORM DESCRIPTOR of the DESTINATION FORM.

AC2 contains a pointer to the packet describing the rectangular area in the source and destination FORM.

This instruction uses the OPERATION MASK and the COMBINATION RULE when planting pixels.

If either the X_EXTENT or Y_EXTENT is set to zero, this instruction has no effect.

The source RECTANGLE must be contained within the source FORM and must fit inside the destination FORM when moved. If it does not lie entirely within one of the FORMs, the WGCHRBLT instruction will cause the RECTANGLE to be clipped so that it does fit in either FORM. This instruction will not write to pixels on the FORM that are write inhibited.

If the source form is not 1-bit per pixel and not on a virtual bitmap, an Invalid WGCHRBLT Source Fault will occur.

Upon completion, the accumulators are unchanged.

DWord #	Mnemonic	Definition
1	D_ULC_X	X coordinate of the location of the upper left-hand corner of the destination location for the 'character'.
2	D_ULC_Y	Y coordinate of the location of the upper left-hand corner of the destina tion location for the 'character'.
3	X_EXTENT	Width of the character to be plotted in pixels.
4	Y_EXTENT	Height of the character to be plotted in pixels.
5	S_ULC_X	X coordinate of the location of the upper left-hand corner of the 'character'.
6	S_ULC_Y	Y coordinate of the location of the upper left-hand corner of

the 'character'.

A.3.7 WGRDATTR -- Read Attribute

WGRDATTR [displacement to emulator]

Input

AC0: Attribute Index

AC1: Form ID

AC2: Address to store attribute

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

Reads an attribute indexed by AC0 from the FORM DESCRIPTOR referenced by AC1 and stores it at the address specified by AC2.

AC0 contains the attribute index.

AC1 contains the key to the FORM DESCRIPTOR.

AC2 contains a word pointer to store the attribute.

If the value in AC0 is greater than 8, an Invalid Attribute Index Fault will occur.

Refer to the chapter "ATTRIBUTES" for more information.

A.3.8 WGWRATTR -- Write Attribute

WGWRATTR [displacement to emulator]

Input

AC0: Attribute index

AC1: Form ID

AC2: Address to read attribute from

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

Writes an attribute indexed by AC0 to the FORM DESCRIPTOR referenced by AC1 with the value at the address specified by AC2.

AC0 contains the attribute index.

AC1 contains the key to the FORM DESCRIPTOR.

AC2 contains a word pointer to read the attribute from.

If the value in AC0 is greater than 8, an Invalid Attribute Index Fault will occur.

Refer to the chapter "ATTRIBUTES" for more information.

Appendix B: Privileged Graphics Instructions

B.4 Data Structures

B.4.1 Comments and Guarantees

The data structures described here constitute a contract between operating systems and microcode for implementing GIS II. This set of information is the sum total required by microcode from operating systems.

An operating system can determine that GIS II microcode exists on the system. If an operating system is only concerned with the existence or non-existence of GIS II, it can attempt to issue a GIS II instruction (WGPFORMS is fairly harmless). If the instruction generates a UIT (Unimplemented Instruction Trap) or transfers control to its error handler, the operating system can assume that GIS II does not exist on the system. If the instruction executes, the operating system can assume that GIS II does exist. If an operating system is concerned with the level of GIS II support (which instructions are supported, what level of optimization exists, etc), there is currently no way to find this out.

The following guarantees can be made about these data structures:

- they are always resident
- they do not cross page boundaries

- they are aligned on a doubleword boundary

The maximum size of any data structure is one page.

Most of the addresses stored in these data structures are physical addresses. This allows the microcode to follow linked lists without having to translate logical addresses.

B.4.2 The Form Descriptor

A form is the object upon which all GIS operations are performed. The Form Descriptor describes the form itself and points to related databases (cursor descriptor and attributes). The Form Descriptor is created in response to a user request.

Form Descriptor:

DWord #	Mnemonic	Description
1	LENGTH	Unsigned 32-bit integer. The length of the Form Descriptor in words.
2	ATTR_BLK	Unsigned 32-bit integer. Physical address of the Attribute Block.
3	BR_ULC_X	Signed 32-bit integer. Represents the X coordinate of the upper-left-corner of the bounding rectangle with respect to the local origin of the form.
4	BR_ULC_Y	Signed 32-bit integer. Represents the Y coordinate of the upper-left-corner of the bounding rectangle with respect to the local origin of the form.

5	BR_EXT_X	Unsigned 32-bit integer. Width of the bounding rectangle in pixels.
6	BR_EXT_Y	Unsigned 32-bit integer. Height of the bounding rectangle in pixels.
7	GBL_ULC_X	Signed 32-bit integer. Represents the X coordinate of the upper-left-corner of the form with respect to the origin of the physical bitmap.
8	GBL_ULC_Y	Signed 32-bit integer. Represents the Y coordinate of the upper-left-corner of the form with respect to the origin of the physical bitmap.
9	LOC_ULC_X	Signed 32-bit integer. Represents the X coordinate of the upper-left-corner of the form with respect to the origin of the form.
10	LOC_ULC_Y	Signed 32-bit integer. Represents the Y coordinate of the upper-left-corner of the form with respect to the origin of the form.
11	FLAGS	Unsigned 32-bit integer. A set of flag bits. The individual bits and their interpretations are described below.
12	FORM_MASK	Unsigned 32-bit integer. A set of bits. The interpretation of these bits is described below.
13	RECT_LIST	Unsigned 32-bit integer. Physical address of a Rectangle Descriptor.
14	CURSOR_DESC	Unsigned 32-bit integer. Physical address of a Cursor Descriptor.
15	DEV_TYPE	Unsigned 32-bit integer. Device type of the physical bitmap.
16	P_BMAP_ADDR	Unsigned 32-bit integer. Microcode ID for the video board.
17	P_X_PITCH	Unsigned 32-bit integer. Number of bits per pixel for the physical bitmap. This number must be within the limits 1 <= X <= 32.
18	P_Y_PITCH	Unsigned 32-bit integer. Number of pixels per line for the physical

		bitmap. This number must be a power of two.
19	P_LOG2_XPITCH	Unsigned 32-bit integer. The log (to the base two) of the X pitch of the physical bitmap.
20	P_LOG2_YPITCH	Unsigned 32-bit integer. The log (to the base two) of the Y pitch of the physical bitmap.
21	V_BMAP_ADDR	Unsigned 32-bit integer. Logical address of the start of the virtual bitmap memory.
22	V_X_PITCH	Unsigned 32-bit integer. Number of bits per pixel for the virtual bitmap. This number must be within the limits 1 <= X <= 32.
23	V_Y_PITCH	Unsigned 32-bit integer. Number of pixels per line for the virtual bitmap. This number must be a power of two.
24	V_LOG2_XPITCH	Unsigned 32-bit integer. The log (to the base two) of the X pitch of the virtual bitmap.
25	V_LOG2_YPITCH	Unsigned 32-bit integer. The log (to the base two) of the Y pitch of the virtual bitmap.

The Bounding Rectangle encompasses all rectangles on the rectangle list for the form. The ULC of the bounding rectangle is equal to the ULC of the uppermost, lefthand rectangle in the rectangle list. The extents of the bounding rectangle are the sum of the extents of the rectangles in the rectangle. list. Tiling a form with rectangles is described in the next section.

The flag bits currently defined in the FLAGS field of the Forms Descriptor are:

Bit #	Description
0	If set, the rectangle list contains one or more

rectangles on the physical bitmap. If clear, no rectangles in the rectangle list are on the physical bitmap.

1 If set, the rectangle list contains one or more rectangles on the virtual bitmap. If clear, no rectangles in the rectangle list are on the virtual bitmap.

2 If set, the rectangle list contains one or more rectangles that are not on either the virtual bitmap or the physical bitmap. If clear, all rectangles in the rectangle list are on either the virtual bitmap or the physical bitmap.

3 If set, the rectangle list contains one or more write-inhibited rectangle. If clear, none of the rectangles in the rectangle list are write-inhibited.

4 If set, the rectangle list consists of a single rectangle. If clear, there are multiple rectangles in the rectangle list.

5 If set, the palette index supplied on a WGWRPAL (Write Palette) instruction is replicated from one byte to three bytes if the palette is on a 24 bit/pixel bitmap.

6-31 Reserved for future use. These bits should be zero, but the microcode will perform no validation.

The Form Mask defines the pixel depth of the bitmaps associated with a form. For a bitmap with "n" bits per pixel, the low-order "n" bits of the Form Mask are set to one. The remaining bits in the Form Mask are set to zero. When performing a GIS instruction, the Form Mask is ANDed with the Operation Mask (contained in the Attribute Block) to produce a mask that determines which bits within a pixel should be operated upon.

The ULC of the virtual bitmap always corresponds to the ULC of the form, regardless of the local coordinate system established for the form.

If the form has only a physical bitmap associated with it, the virtual bitmap portion of the Form Descriptor will be filled with zeros. An "empty" bitmap description will never be used by microcode so long as no rectangles on the form's rectangle list refer to the non-existent bitmap. If a rectangle does refer to a non-existant bitmap, the results are undefined. Microcode will probably trap.

If a GIS II instruction is in progress on a particular Form Descriptor, only the following information can be safely altered:

1) Global ULC
2) Local ULC
3) The rectangle bits in the FLAGS
4) The rectangle list

Changing any other information will produce undefined results.

B.4.3 The Attribute Block

Values such as foreground color and line style are stored in the Attribute Block. Initially, the Attribute Block is filled with a set of default values. All of these values can be examined by the user with the WGRDATTR (Read Attribute) instruction and altered with with the WGWRATTR (Write Attribute) instruction. The Attribute Block is created when a Form Descriptor is created.

Attribute Block:

DWord #	Mnemonic	Description
1	LENGTH	Unsigned 32-bit integer. The length of the Attribute Block in words.
2	OP_MASK	Unsigned 32-bit integer. A mask that determines which bits within a pixel will be affected by a GIS operation.
3	COMBO_RULE	Unsigned 32-bit integer. Specifies one of 16 boolean functions to be applied during a GIS operation.
4	LINE_CTRL	Unsigned 32-bit integer. A set of flag bits that govern the drawing of lines. The individual bits and their interpretations are described below.
5	LINE_F_COLOR	Unsigned 32-bit integer. Pixel value of the foreground color used when drawing lines.
6	LINE_B_COLOR	Unsigned 32-bit integer. Pixel value of the background color used when drawing lines.
7	LINE_STYLE	Unsigned 32-bit integer. A set of bits that determines the texture of any lines that are drawn. Bits that are set indicate that a foreground color pixel should be planted. Bits that are clear indicate that a background color pixel should be planted.
8	CHAR_CTRL	Unsigned 32-bit integer. A set of flag bits that govern the drawing of characters. The individual bits and their interpretations are described below.
9	CHAR_F_COLOR	Unsigned 32-bit integer. Pixel value of the foreground color used when drawing characters.
10	CHAR_B_COLOR	Unsigned 32-bit integer. Pixel value of the background color used

when drawing characters.

The flag bits currently defined in the LINE_CTRL field of the Forms Descriptor are:

Bit #	Description
0	Draw the foreground pixels if clear. Suppress the foreground pixels if set.
1	Draw the background pixels if clear. Suppress the background pixels if set.
2	Draw the initial point if clear. Suppress the initial point if set.
3	Draw the final point if clear. Suppress the final point if set.
4-31	Reserved for future use. These bits should be zero, but the microcode will perform no validation.

The flag bits currently defined in the CHAR_CTRL field of the Forms Descriptor are:

Bit #	Description
0	Draw the foreground pixels if clear. Suppress the foreground pixels if set.
1	Draw the background pixels if clear. Suppress the background pixels if set.
2-31	Reserved for future use. These bits should be zero, but the microcode will perform no validation.

B.4.4 The Rectangle Descriptor

Although the user perceives a form as a unit, in reality the form may be divided into multiple pieces. For convenience, each piece is a rectangle. A Rectangle Descriptor describes one of the set of rectangles that make up a form. A rectangle can reside entirely on the physical bitmap, entirely on the virtual bitmap, or on neither bitmap.

The collection of rectangles that make up an entire form is called a "tiling" of that form. Certain constraints are placed on a tiling. Rectangles may not overlap, must completely tile the bounding rectangle for the form, and may not lie outside of the form. If either of these conditions are violated, undefined results will occur.

Rectangle Descriptor:

DWord #	Mnemonic	Description
1	NEXT	Unsigned 32-bit integer. Physical address of the next Rectangle Descriptor in the list or minus one.
2	FLAGS	Unsigned 32-bit integer. A set of flag bits. The individual bits and their interpretations are described below.
3	ULC_X	Signed 32-bit integer. Represents the X coordinate of the upper-left-corner of the rectangle with respect to the origin of the form.
4	ULC_Y	Signed 32-bit integer. Represents the Y coordinate of the upper-left-corner of the rectangle with respect to the origin of the form.

5	EXT_X	Unsigned 32-bit integer. Width of the rectangle in pixels.
6	EXT_Y	Unsigned 32-bit integer. Height of the rectangle in pixels.

The flag bits currently defined in the FLAGS field of the Rectangle Descriptor are:

Bit #	Description
0	If set, the rectangle is on the physical bitmap.
1	If set, the rectangle is on the virtual bitmap.
2	If set, the rectangle is not on any bitmap.
3	If set, the rectangle is write-inhibited.
4-31	Reserved for future use. These bits should be zero, but the microcode will perform no validation.

Bits #0, #1, and #2 are mutually exclusive. Exactly one bit can be set.

B.4.5 The Cursor Descriptor

A cursor is some pattern that is drawn on the bitmap screen to represent the position of a pointing device.

There are two different types of cursors. The first type is an image, such as an arrow. This cursor is defined by the rectangle that contains the visible portion of the image. The second type of cursor is a cross-hair. This cursor is defined by the

endpoints of the horizontal and vertical lines that form the cross-hair. In the case of a full-screen cross-hair, the horizontal and vertical lines always span the entire width and height (respectively) of the bitmap screen. The endpoints of a full-screen cross-hair are always on the edge of the bitmap screen. In the case of a cross-hair that is simply very large, the cross-hair may be clipped to the edge of the bitmap screen. The endpoints of a large cross-hair define the visible portion of the cross-hair.

For each of these two types of cursors, there is a different descriptor. The first doubleword of the Cursor Descriptor, however, remains the same in all cases. This is the Flags word, which contains bits that determine the format of the descriptor that follows.

The Cursor Descriptor is used by the microcode to determine if a given GIS operation could intersect the cursor. If no intersection is possible, microcode can perform the GIS operation directly. If an intersection could occur, microcode must ask the operating system to erase the cursor before the GIS operation can be performed.

Microcode must access the Cursor Descriptor in two stages. The first access retrieves the Flags word. If the cursor is invisible, microcode must not attempt to access the rest of the descriptor. If the cursor is visible, microcode reads in the rest of the descriptor based on the type of cursor.

When a Cursor Descriptor changes, all forms that refer to that descriptor must be purged from the Form Cache prior to the change.

Image Descriptor:

DWord #	Mnemonic	Description
1	FLAGS	Unsigned 32-bit integer. A set of flag bits. The individual bits and their interpretations are described below.
2	ULC_X	Signed 32-bit integer. Represents the X coordinate of the upper-left-corner of the rectangle with respect to the origin of the physical bitmap.
3	ULC_Y	Signed 32-bit integer. Represents the Y coordinate of the upper-left-corner of the rectangle with respect to the origin of the physical bitmap.
4	EXTENT_X	Unsigned 32-bit integer. Width of the rectangle in pixels.
5	EXTENT_Y	Unsigned 32-bit integer. Height of the rectangle in pixels.

Cross-hair Descriptor:

DWord #	Mnemonic	Description
1	FLAGS	Unsigned 32-bit integer. A set of flag bits. The individual bits and their interpretations are described below.
2	H_START_X	Signed 32-bit integer. Represents the X coordinate of the left endpoint of the horizontal line with respect to the origin of the physical bitmap.
3	H_START_Y	Signed 32-bit integer. Represents the Y coordinate of the left endpoint of the horizontal line with respect to the origin of the physical bitmap.
4	H_END_X	Signed 32-bit integer. Represents the X coordinate of the right endpoint of the horizontal line with respect to the origin of the physical bitmap.

5	H_END_Y	Signed 32-bit integer. Represents the Y coordinate of the right endpoint of the horizontal line with respect to the origin of the physical bitmap.
6	V_START_X	Signed 32-bit integer. Represents the X coordinate of the top endpoint of the vertical line with respect to the origin of the physical bitmap.
7	V_START_Y	Signed 32-bit integer. Represents the Y coordinate of the top endpoint of the vertical line with respect to the origin of the physical bitmap.
8	V_END_X	Signed 32-bit integer. Represents the X coordinate of the bottom endpoint of the vertical line with respect to the origin of the physical bitmap.
9	V_END_Y	Signed 32-bit integer. Represents the Y coordinate of the bottom endpoint of the vertical line with respect to the origin of the physical bitmap.

The flag bits currently defined in the FLAGS field of the Cursor Descriptor are:

Bit #	Description
0	If set, the cursor is visible. If clear, the cursor is invisible.
1	If set, the cursor is an image cursor.
2	If set, the cursor is a cross-hair cursor.
3-31	Reserved for future use. These bits should be zero, but the microcode will perform no validation.

Only one of the cursor type bits may be set at any one time.

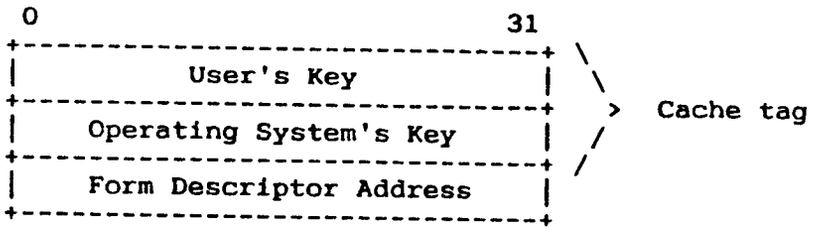
Setting both of these bits will produce undefined results.

B.5 The Form Cache

In order for a GIS II instruction to operate on a form, the target form must be loaded into the Form Cache. If the target form is not in the Form Cache, microcode will generate a "Form Cache Miss" fault (GIS II faults are described in more detail later). The operating system controls access to forms by loading or not loading a particular form into the Form Cache in response to a "Form Cache Miss" fault.

A form is uniquely identified in the Form Cache by a cache tag consisting of the address of the Form Descriptor and two keys. The first key is the user's form ID. This is the number that is supplied on a GIS II instruction. The second key is some process-specific number chosen by the operating system. For MV-class machines, this number is the contents of the Segment Base Register (SBR) for the ring on whose behalf the form was loaded into the Form Cache. This is not only process-specific but ring-specific, a fact that allows ring maximization on forms. For other architectures, some other value may be appropriate.

A Form Cache entry, therefore, looks like this:



To determine if the form required by a particular GIS II instruction is in the Form Cache, microcode obtains the user's form ID and the operating system's key and searches the Form Cache for a cache tag containing those values. If a tag is found whose contents match both keys, the GIS II instruction continues, using the form pointed to by the Form Descriptor address. If no match can be found, the microcode faults to the operating system.

It should be noted that microcode will only check the operating system's key if the GIS II instruction was issued from Rings One to Seven. If a GIS II instruction is issued from Ring Zero, the microcode will only search the Form Cache for a tag that contains the given form ID.

While processing of a GIS II instruction, any forms in the Form Cache that are unused by the instruction may be purged.

B.6 Privileged Instruction Dictionary

The following instructions are privileged instructions that may only be issued from Ring Zero.

B.6.1 WGLFORM -- Load the Form Cache

Performs an unconditional load of the form specified by AC2 into the Form Cache. If the form being loaded exists in the cache

already, the version in the cache is overwritten. The cache tag associated with the form is built from AC0 and AC1.

WGLFORM [displacement to error handler]

Input

AC0: Operating System's Key

AC1: User's Form ID

AC2: Physical address of the Form Descriptor

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

B.6.2 WGPFORMS -- Purge the Form Cache

Purges either a selected form from the Form Cache or all forms in the Form Cache. A cache tag is built from AC0 and AC1 if only a single form is being purged.

WGPFORMS [displacement to error handler]

Input

AC0: One of the following:

- Zero. Purge the form specified by AC1 from the Form Cache.

- Non-zero. Purge all entries in the Form Cache.

AC1: One of the following:

- Unused if AC0 is non-zero.
- User's Form ID if AC0 is zero.

AC2: Unused

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

B.6.3 WGRDPAL -- Read Palette

Reads an entry from the palette specified by AC1 into the packet specified by AC2. The palette entry to read is specified by AC0. The values read from the palette may not be the values originally written to the palette.

WGRDPAL [displacement to error handler]

Input

AC0: Palette index

AC1: Microcode ID for video board

AC2: Logical address of a palette entry packet

Output

AC0: Unchanged

AC1: Unchanged

AC2: Unchanged

Palette Entry Packet:

DWord #	Mnemonic	Description
1	RED_P0	Unsigned 32-bit integer. Red color intensity for phase zero of the blink clock.
2	GREEN_P0	Unsigned 32-bit integer. Green color intensity for phase zero of the blink clock.
3	BLUE_P0	Unsigned 32-bit integer. Blue color intensity for phase zero of the blink clock.
4	GREY_P0	Unsigned 32-bit integer. Grey level for phase zero of the blink clock.
5	RED_P1	Unsigned 32-bit integer. Red color intensity for phase one of the blink clock.
6	GREEN_P1	Unsigned 32-bit integer. Green color intensity for phase one of the blink clock.
7	BLUE_P1	Unsigned 32-bit integer. Blue color intensity for phase one of the blink clock.
8	GREY_P1	Unsigned 32-bit integer. Grey level for phase one of the blink clock.

B.6.4 WGWRPAL -- Write Palette

Writes an entry into the palette specified by AC1 from the packet specified by AC2. The palette entry to write is specified by AC0.

WGWRPAL [displacement to error handler]

Input

AC0: Palette index

AC1: Microcode ID for video board

AC2: Logical address of a palette entry packet

Output

AC0: Unchanged

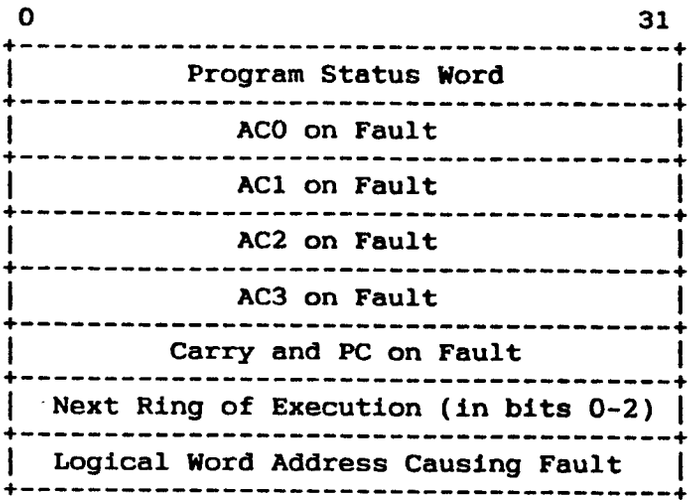
AC1: Unchanged

AC2: Unchanged

B.7 Faults Under GIS II

B.7.1 The Structure of the Current Fault World

The structure of a fault context block for any processor other than an MV/6000 or an MV/8000 is:



The following AC values are passed to the operating system on a fault:

AC0: Undefined

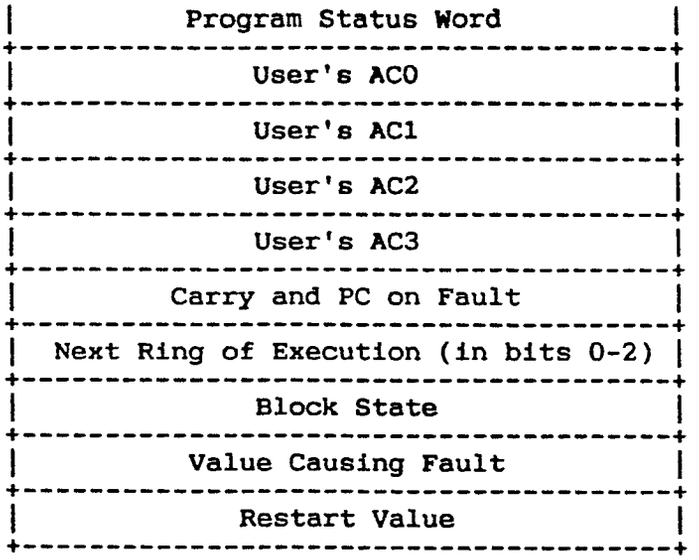
AC1: Fault code

AC2: Undefined

AC3: Undefined

B.7.2 The Structure of the GIS II Fault World

The structure of a GIS II fault context block is:



The following AC values are passed to the operating system on a GIS II fault:

AC0: Undefined

AC1: Fault code

AC2: Undefined

AC3: Undefined

When a GIS II fault occurs, the "Restart Value" field of the context block contains a zero. If this field remains unchanged when the operating system resumes the fault (via WDPOP), the microcode will continue the faulting instruction. If, however, the operating system stores some non-zero value in the "Restart Value" field prior to resuming the fault, the microcode will stop processing the faulting instruction and begin execution of the next sequential instruction.

B.7.3 Fault Codes

The fault codes currently defined are:

Fault Code	Fault Type
0	Multiple ERCC Fault
1	Page Table Depth Fault
2	Page Table Page Fault
4	Normal Object Reference Fault

The GIS II fault codes are:

Fault Code	Fault Type	Contents of "Value Causing Fault"
5	Form Cache Miss	The ID of the form that was not found in the Form Cache.
6	Cursor Intersect	The Form ID supplied on the GIS instruction.
7	Unknown Attribute Index	The Form ID supplied on the WGRDATTR or WGWRATTR instruction.
8	Invalid WGCHRBLT Source	Undefined.
9-12	[Reserved for future use]	

The "Unknown Attribute Index" and "Invalid WGCHRBLT Source" faults can only be taken at the start of a GIS instruction. Once the microcode begins performing the GIS instruction, these faults are invalid.

The "Form Cache Miss" and "Cursor Intersect" faults may be taken at any time.

None of these faults should be generated at interrupt level. It is guaranteed that the Form Cache will always be capable of containing the source form and destination form. It is Software's responsibility to load the cache with the correct forms in order to prevent faults when running at interrupt level.

B.7.4 Fault Dictionary

This dictionary gives the situation in which each GIS II fault is taken and the actions that the operating system performs to service the fault. In all cases, the microcode will generate these faults using the existing page fault mechanism. Microcode state will be stored in the GIS II fault context block described above. The WDPOP instruction will be used to resume from these faults.

B.7.4.1 Form Cache Miss

This fault is generated if the Form ID given on a particular GIS instruction could not be found in the Form Cache.

The fault/service/resume procedure is:

1) Microcode faults to the operating system with the user's Form ID in the "Value Causing Fault" field of the context block.

2) The operating system searches for the specified form in its internal databases.
3) If no form is found, the operating system traps the process using the same mechanism as an "Inward Address Trap". The trap code will be "Invalid Form ID Trap".
4) If a form is found, the operating system issues an WGLFORM instruction to load the Form Descriptor.
5) The operating system passes control back to the microcode with a WDPOP instruction.

B.7.4.2 Cursor Intersect

This fault is generated if a particular GIS instruction could corrupt the cursor.

The fault/service/resume procedure is:

1) Microcode faults to the operating system with the user's Form ID in the "Value Causing Fault" field of the context block.
2) The operating system erases the cursor associated with the specified form.
3) The operating system passes control back to the microcode with a WDPOP instruction.

B.7.4.3 Unknown Attribute Index

This fault is generated if the attribute index provided on a WGRDATTR or WGWRATTR instruction falls outside of the Form Descriptor's attribute block.

The fault/service/resume procedure is:

1) Microcode faults to the operating system with the user's Form ID in the "Value Causing Fault" field of the context block.
2) If the Attribute Index specified by the user does not refer to a valid "soft" attribute, the operating system traps the process.
3) If the Attribute Index specified by the user refers to a valid "soft" attribute, the operating system sets the "Restart Value" field of the context block to some non-zero value and passes control back to the microcode with a WDPOP instruction.

B.7.4.4 Invalid WGCHRBLT Source

This fault is generated if the source form given on a WGCHRBLT instruction does not meet the following restrictions:

1) The rectangle list consists of a single rectangle on the virtual bitmap.
2) The virtual bitmap is 1 bit/pixel deep.

The fault/service/resume procedure is:

1) Microcode faults to the operating with nothing in the "Value Causing Fault" field of the context block.
2) The operating system traps the process.

CLAIMS:

1. A digital computer system, comprising main memory means (102) for storing instructions and data; processing means (101) for performing operations on data in response to the instructions; and display means for displaying representations of data; characterised in that the system controls the displays by the steps of:

a) storing in the main memory means (102) logical form descriptors for describing organization of data to be displayed;

b) storing in the main memory means (102) instructions for specifying first data and for specifying representations of first data which are to be displayed, and for describing the position within the organization described by the logical form descriptors at which the representation of the first data are to be displayed;

c) storing in the main memory means (102) form descriptor identifiers for representation of logical form descriptors;

d) storing in the main memory means (102) operating system keys for representation of users of the digital computer system;

e) selecting an instruction and a form identifier;

f) calculating, using the processing means (101), second data using the operating system key representing the user of the digital computer system and the matching form identifier, the second data being the form descriptor;

g) calculating, in the processing means (101), third data which is a function of the selected instruction, the second data, and certain first data specified by the selected instruction, the third data being a representation of what is to be displayed; and

h) forwarding the third data to the display means for representation of this data to be displayed.

0223557

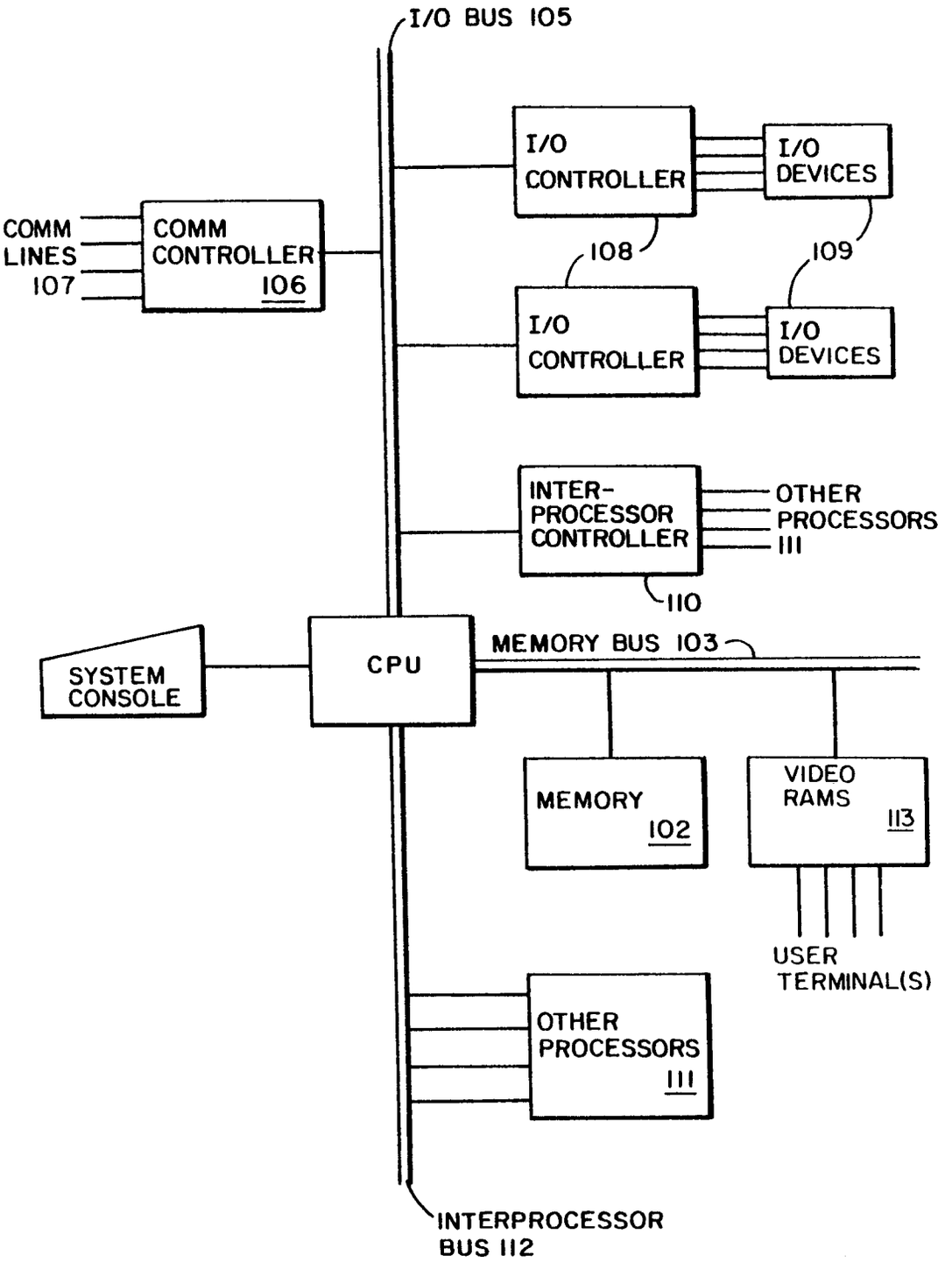


FIG. 1 PRIOR ART

0223557

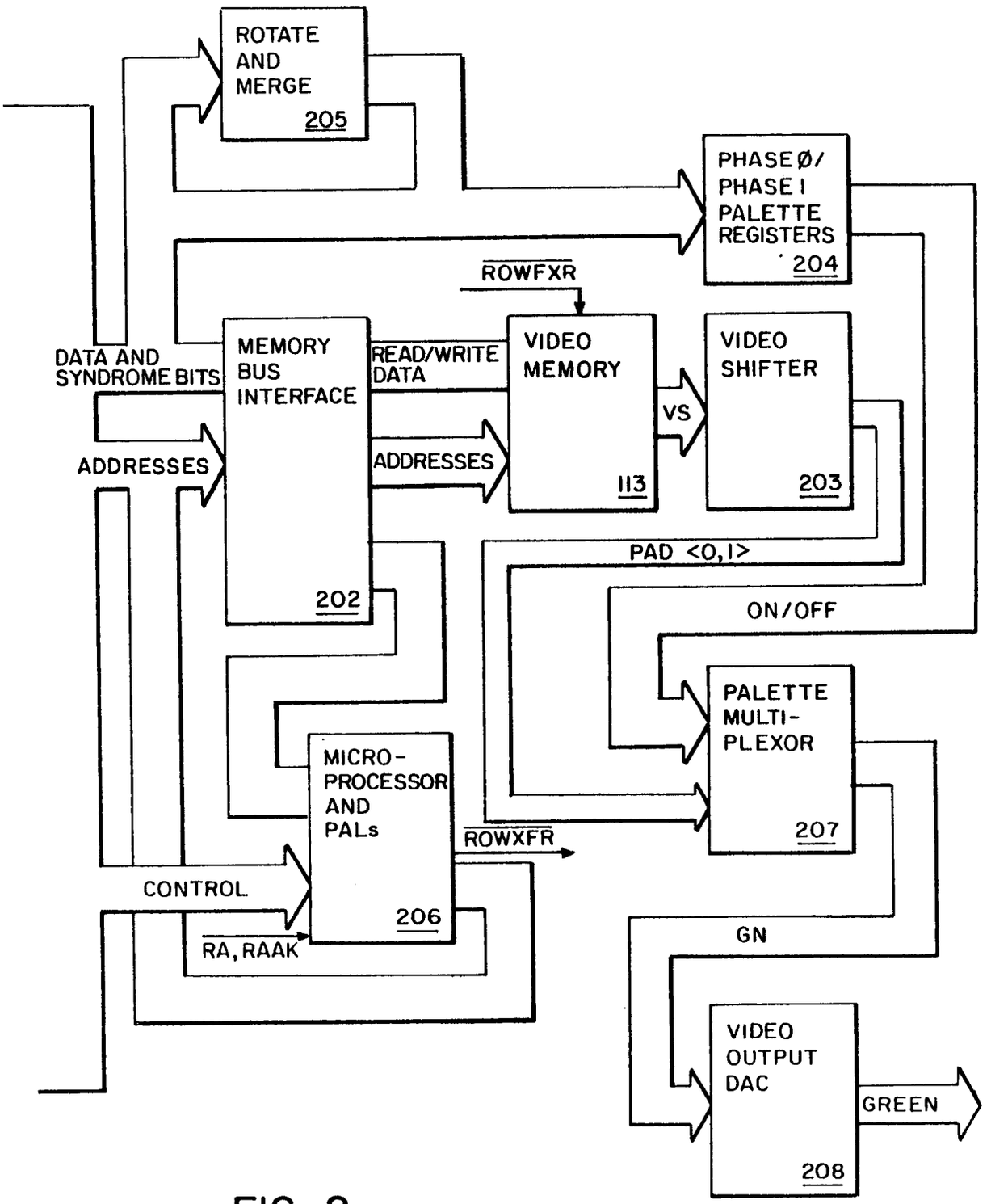


FIG. 2

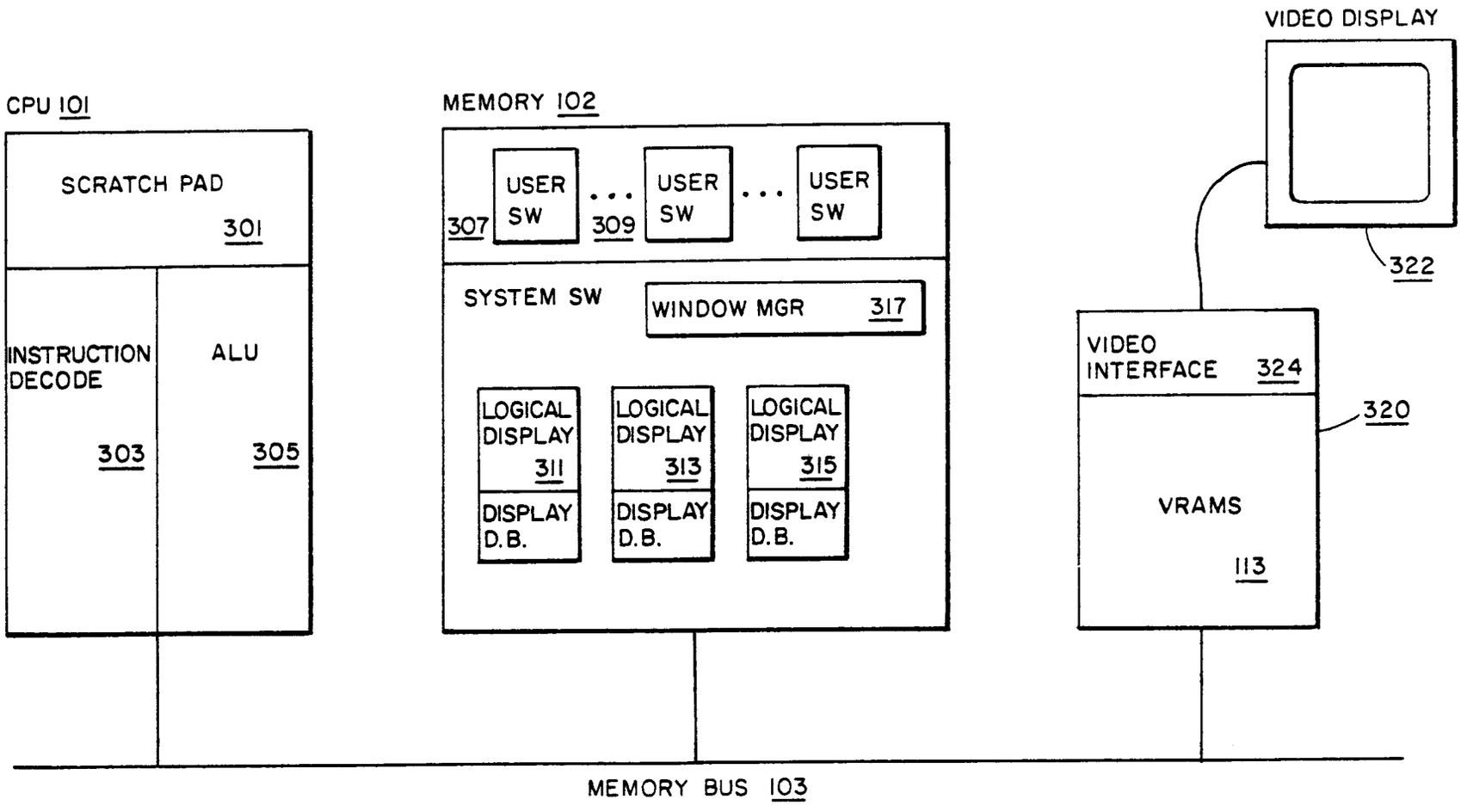


FIG. 3 PRIOR ART DATA PROCESSING VIDEO DISPLAY SYSTEM

0223557

0223557

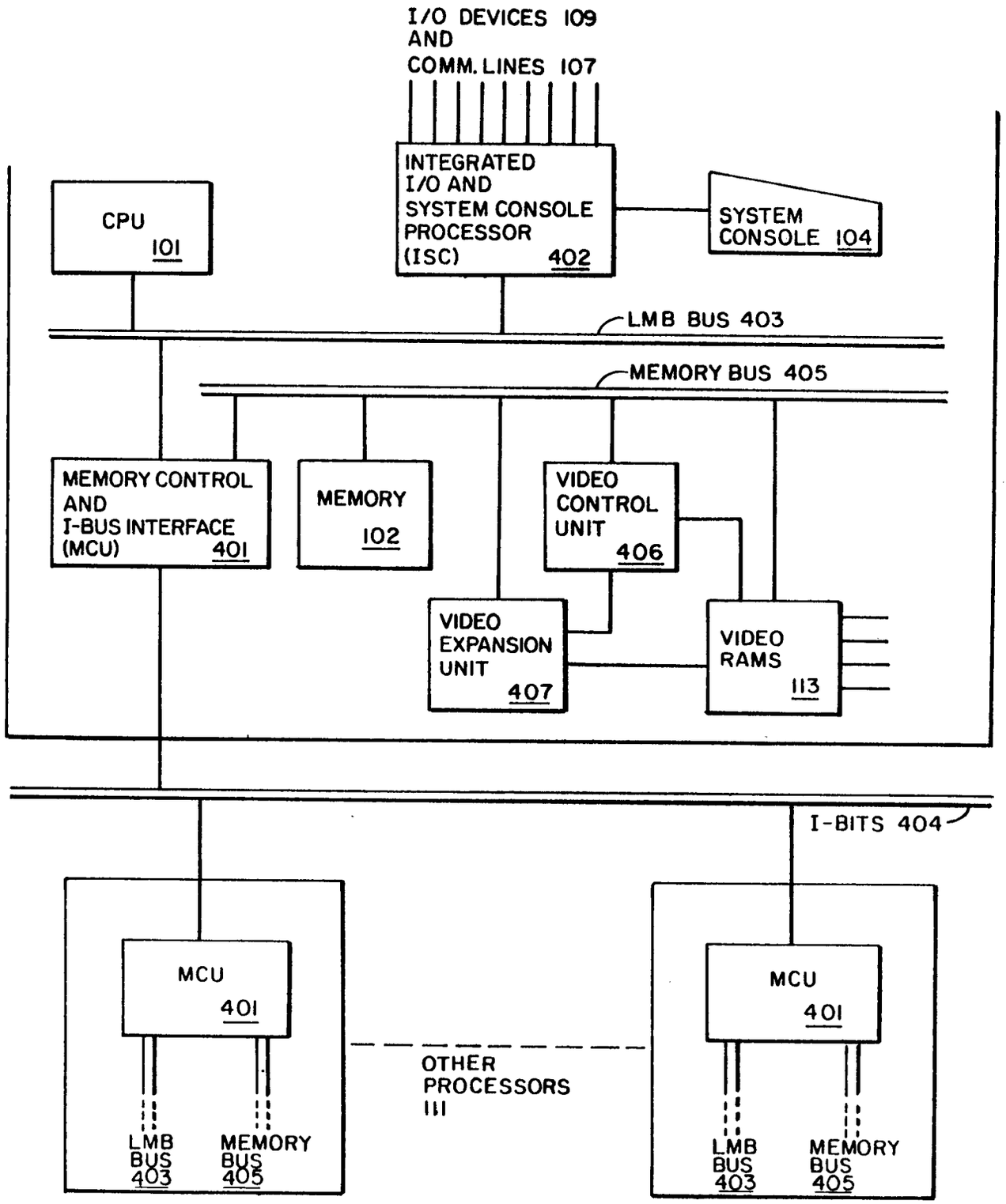


FIG. 4 OVERVIEW

0223557

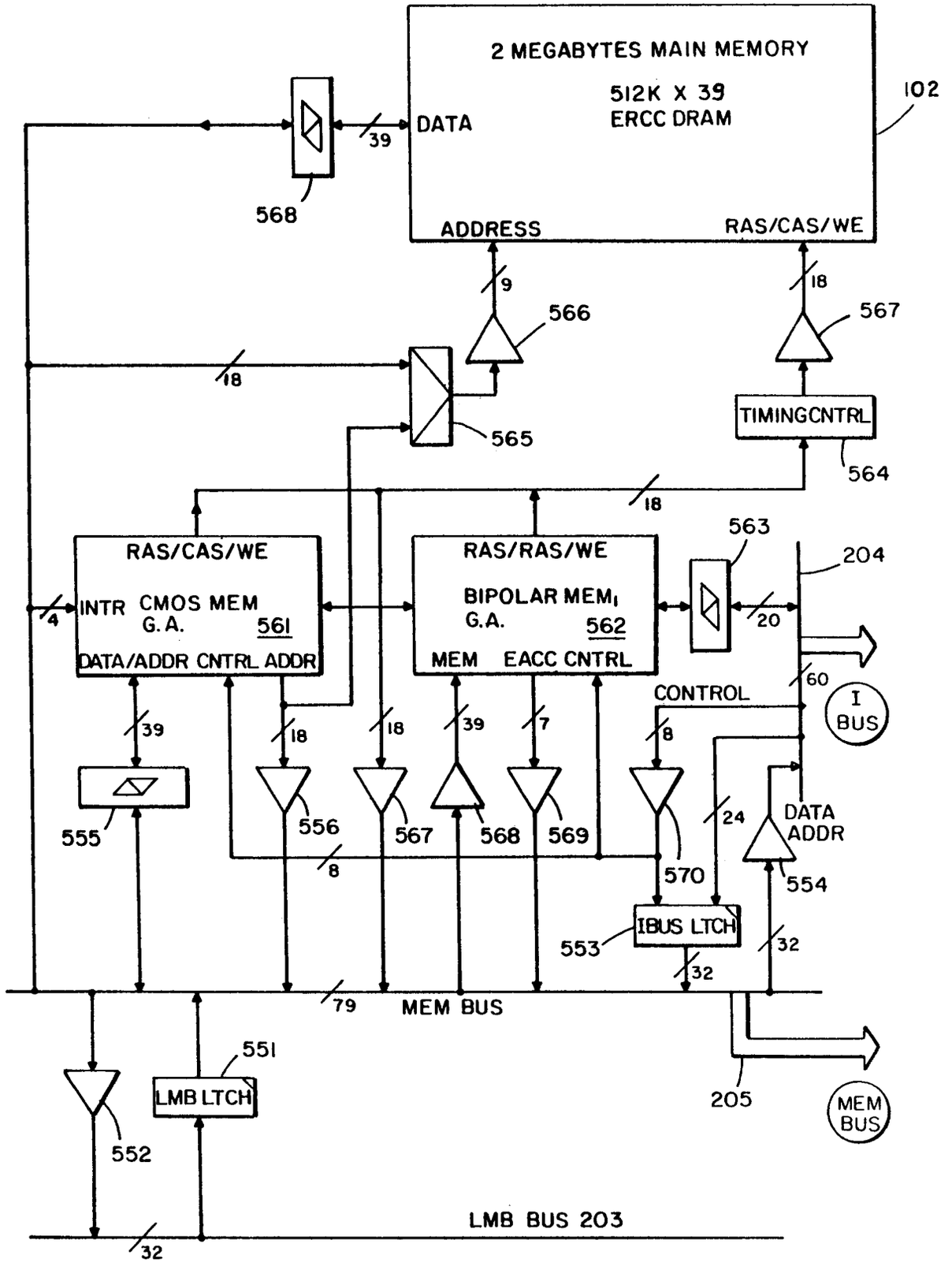


FIG. 5 MCU 401

0223557

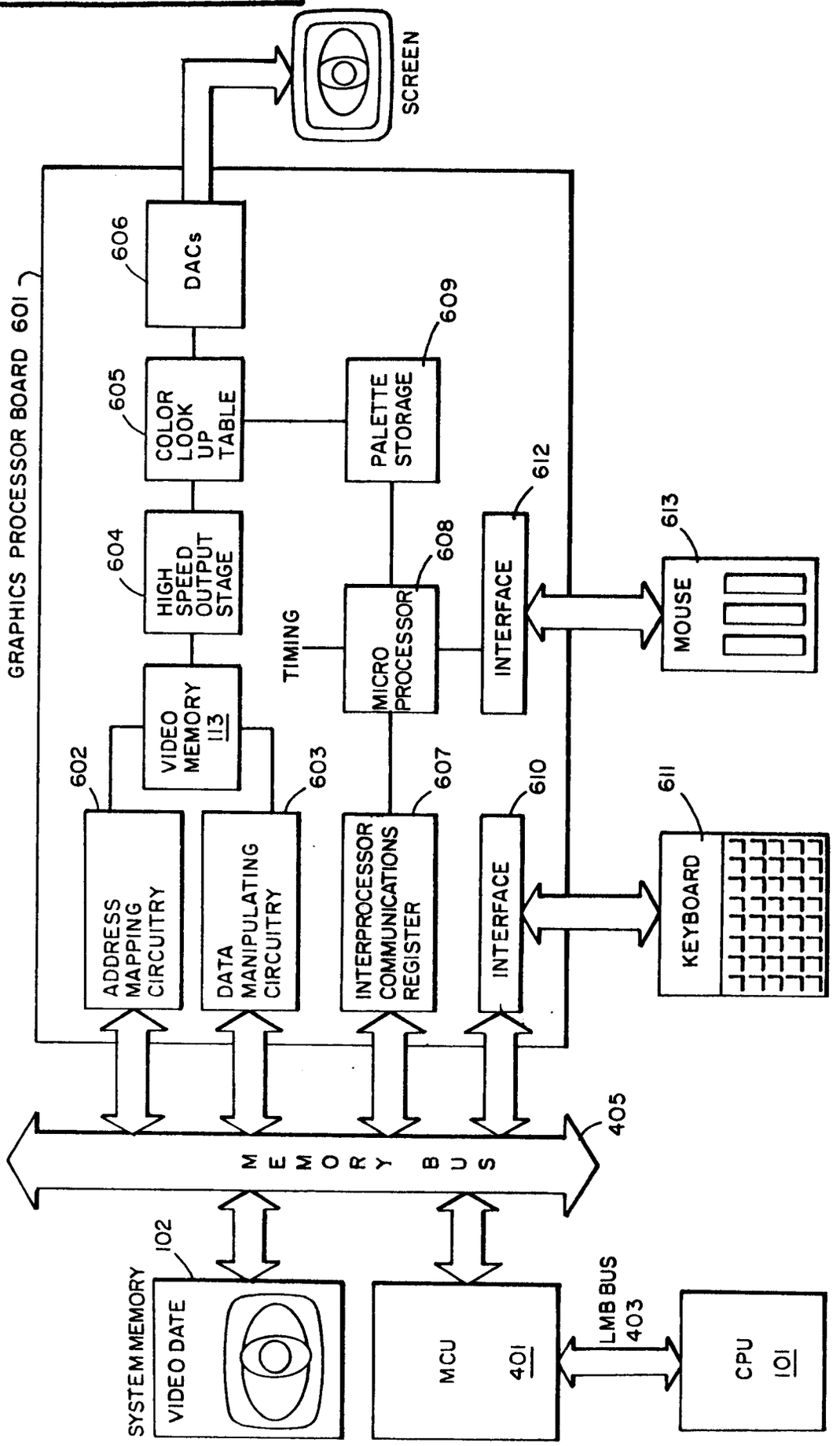


FIG. 6 GRAPHICS FUNCTIONAL OVERVIEW

0223557

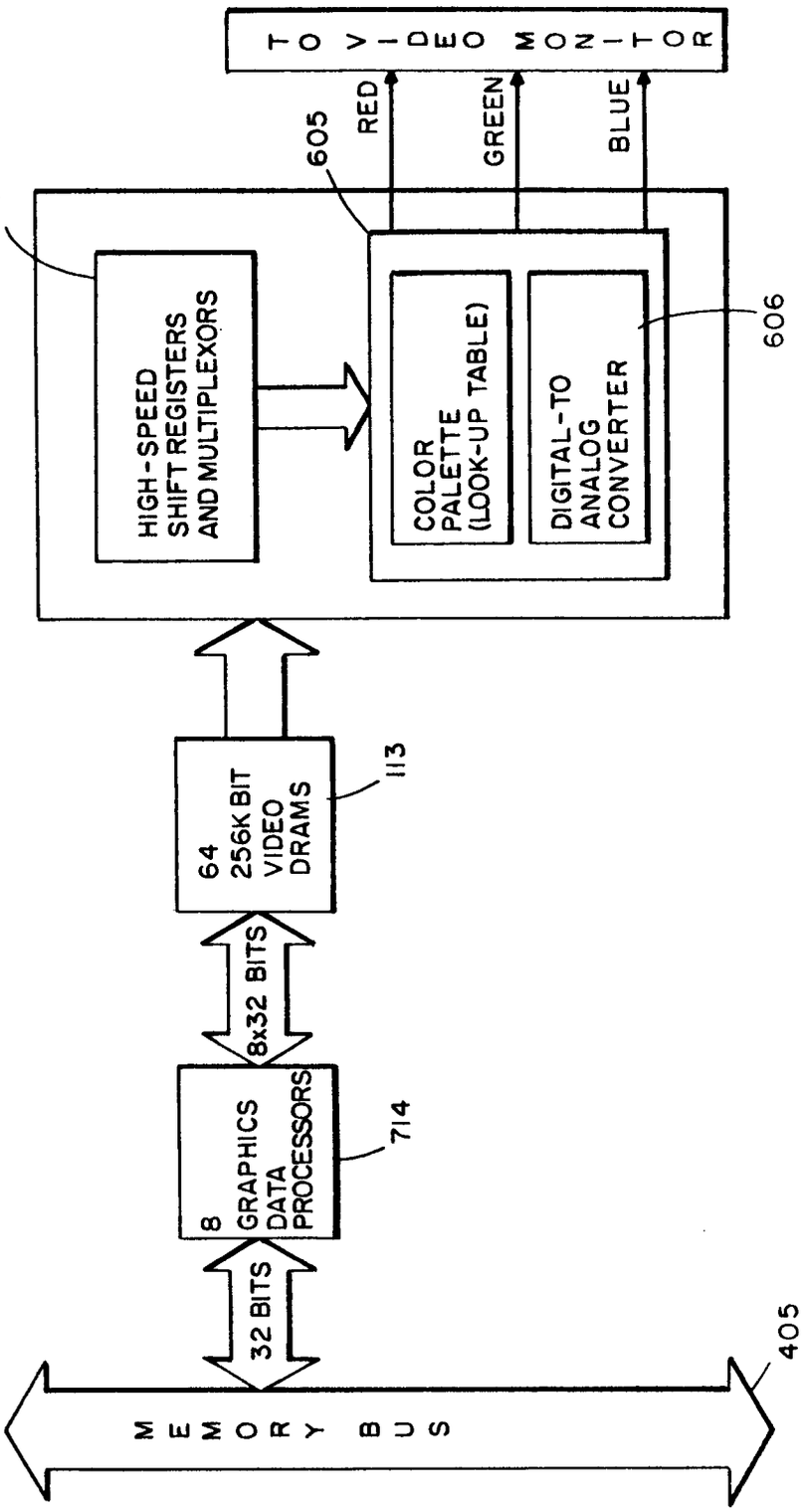


FIG. 7 PIXEL DATA FLOW MOLD

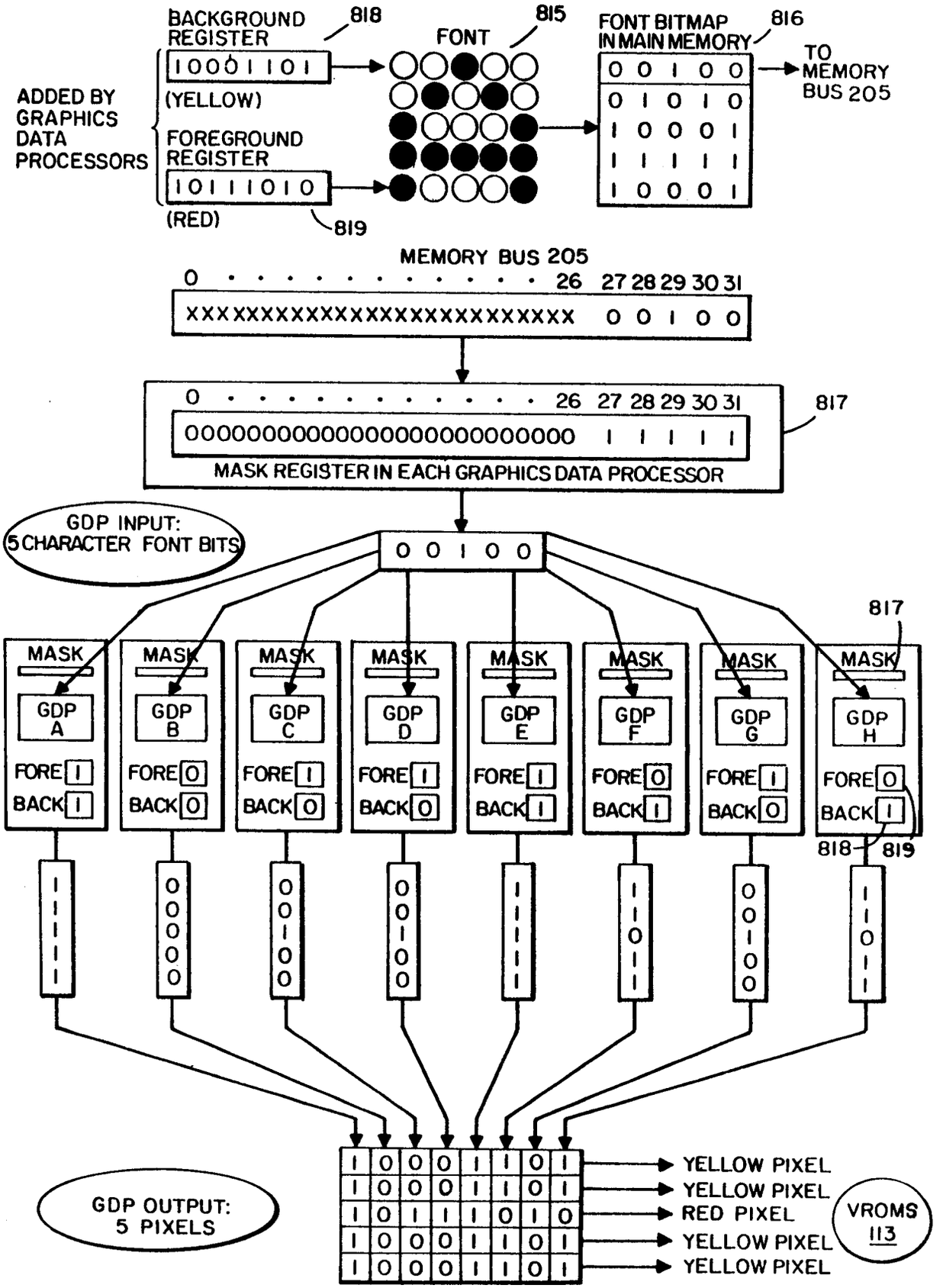


FIG. 8 PLANE MODE DATA FLOW

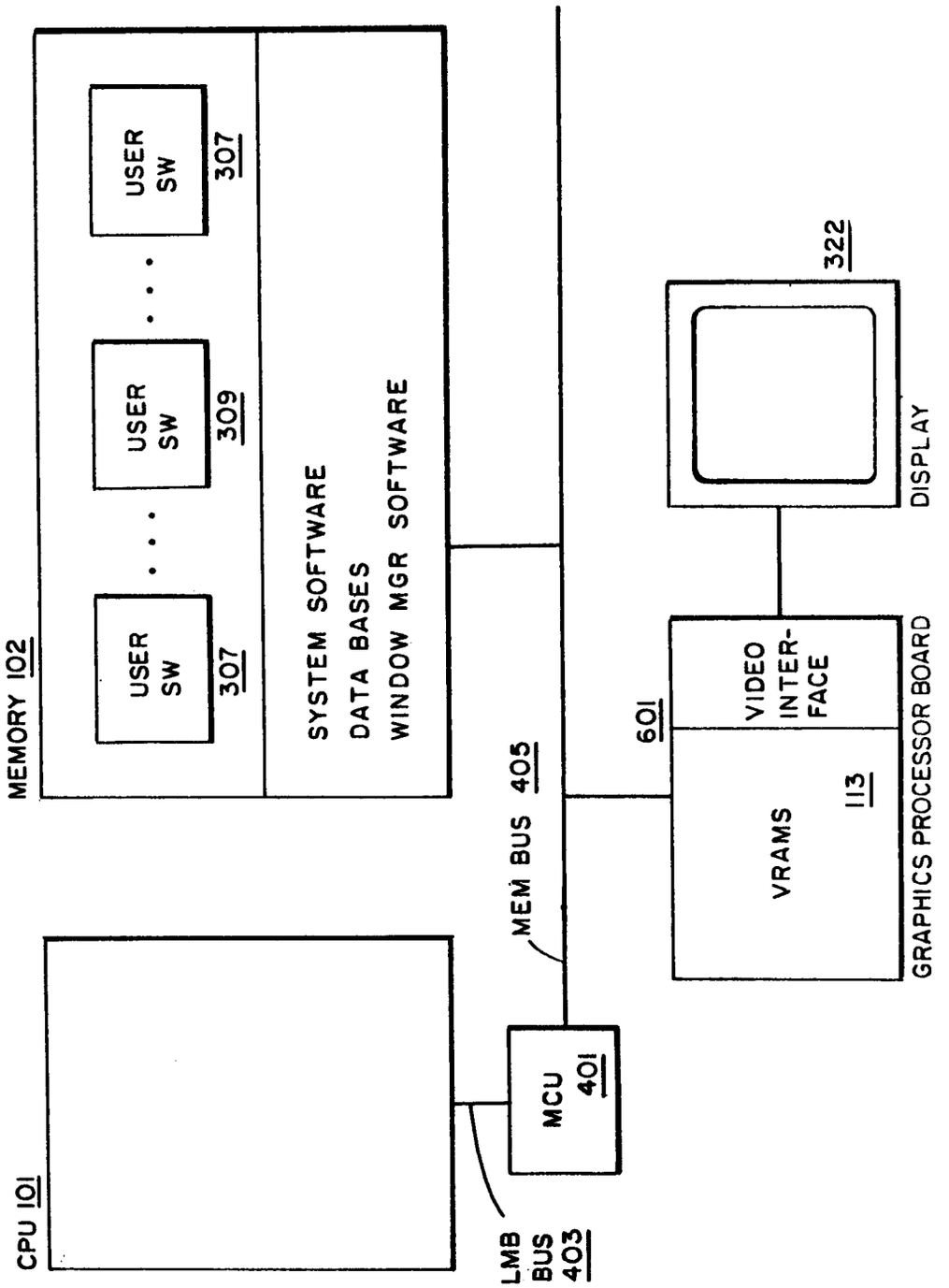
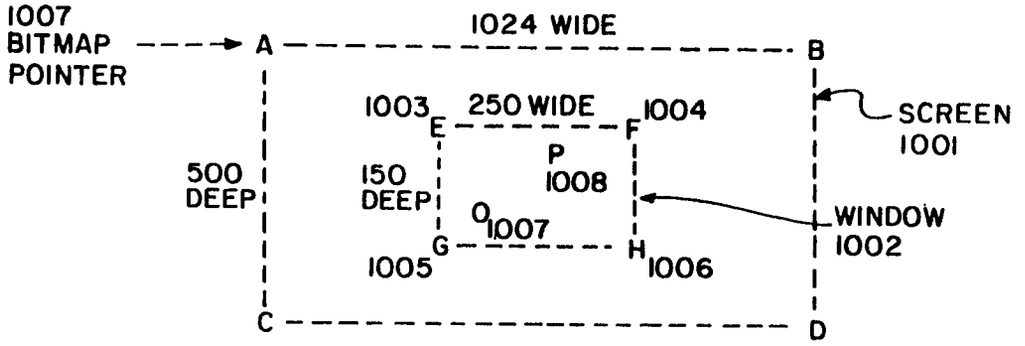


FIG. 9 WINDOWING DISPLAY DATA SYSTEM BLOCK DIAGRAM



A: GLOBAL 0,0
B: GLOBAL 1023,0
C: GLOBAL 0,499
D: GLOBAL 1023, 449
E: GLOBAL 100,200 = LOCAL 0,0
F: GLOBAL 349,200 = LOCAL 249,0
G: GLOBAL 100,349 = LOCAL 0,149
H: GLOBAL 349,349 = LOCAL 249,149

FIG. 10 COORDINATE CONVERSION

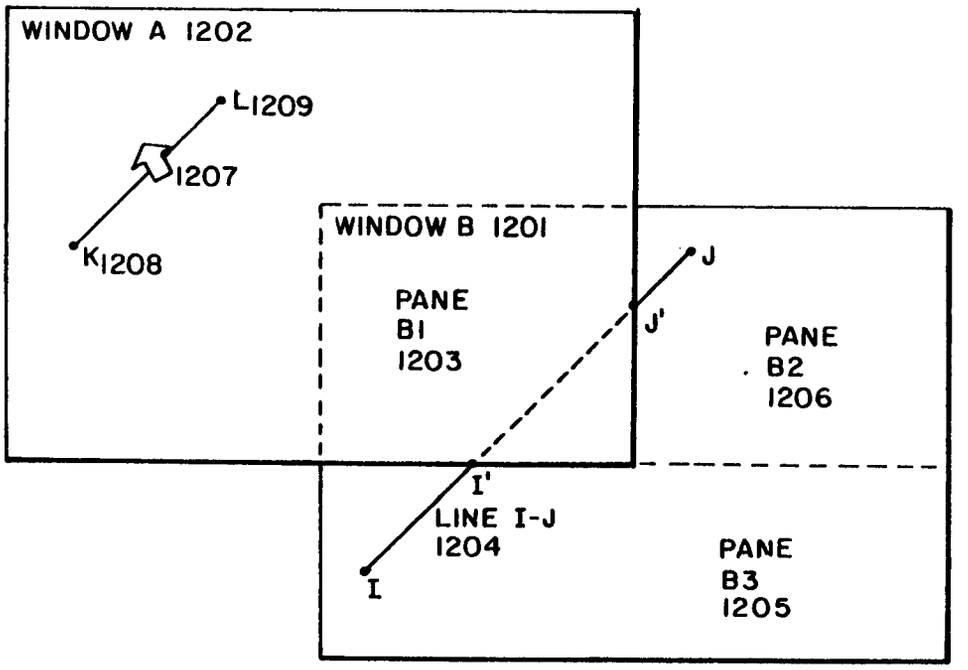


FIG. 12 TWO WINDOWS

0223557

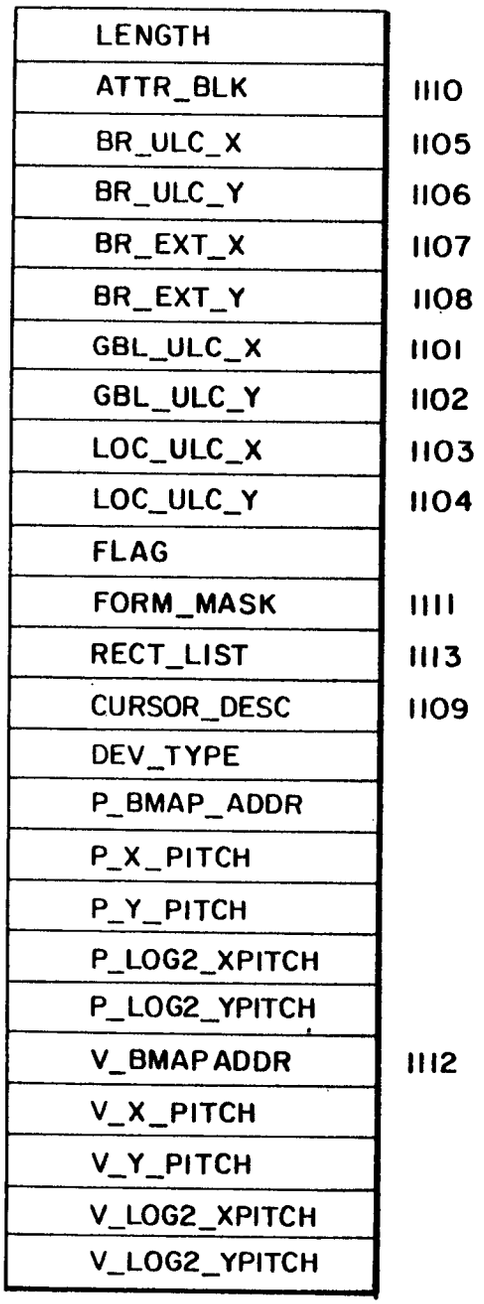


FIG. 11 FORM DESCRIPTOR

0223557

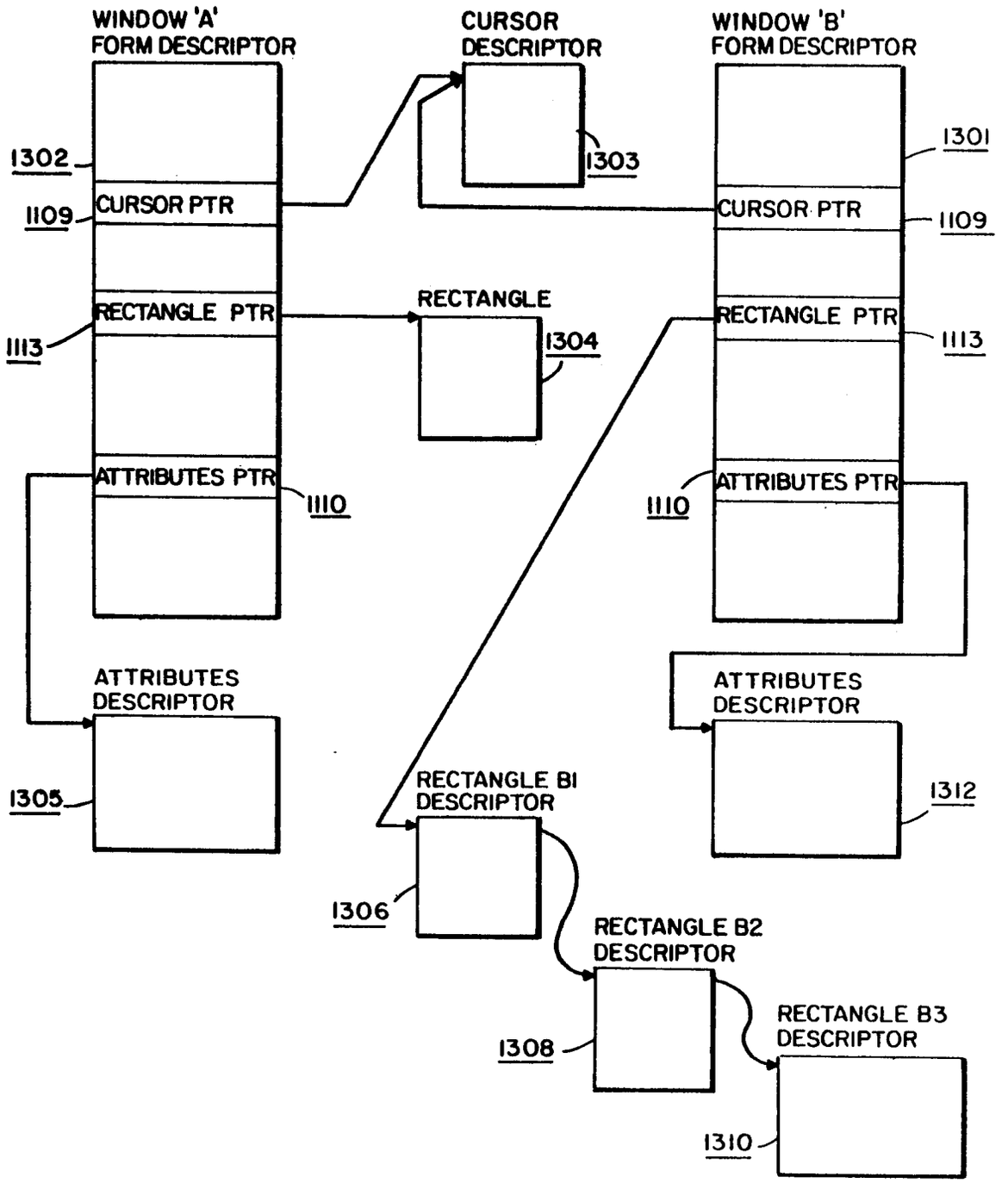


FIG. 13 FORM DESCRIPTORS FOR WINDOWS A & B OF FIG. 12

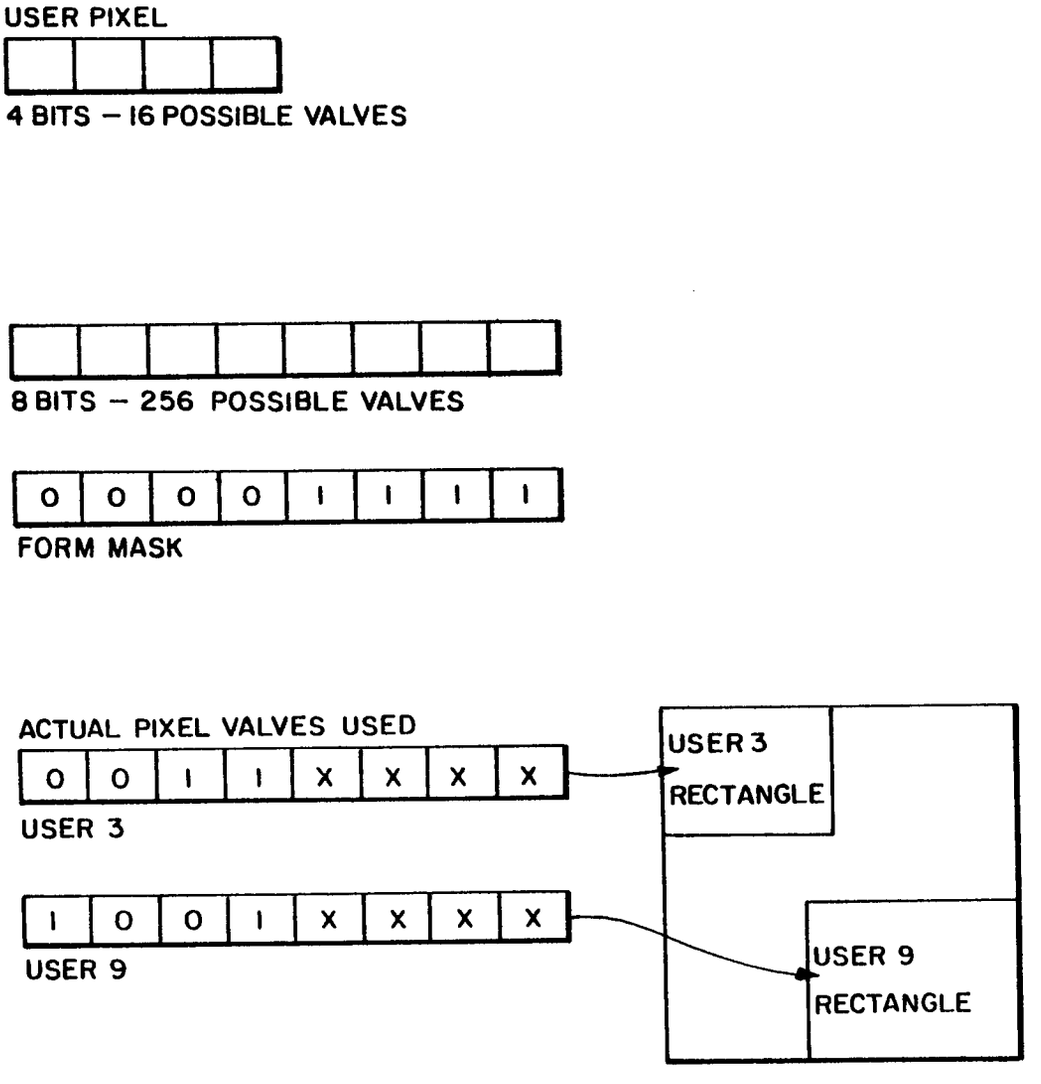


FIG. 14 PIXEL MAPPING EXAMPLE TO SHARE PALETTE