Chapter 25

VGA Data Machinery
The Barrel Shifter, Bit Mask, and Set/Reset Mechanisms

In the last chapter, we examined a simplified model of data flow within the GC portion of the VGA, featuring the latches and ALUs. Now we're ready to expand that model to include the barrel shifter, bit mask, and the set/reset capabilities, leaving only the write mode to be explored over the next few chapters.

VGA Data Rotation

Figure 25.1 shows an expanded model of GC data flow, featuring the barrel shifter and bit mask circuitry. Let's look at the barrel shifter first. A barrel shifter is circuitry capable of shifting—or rotating, in the VGA's case—data an arbitrary number of bits in a single operation, as opposed to being able to shift only one bit position at a time. The barrel shifter in the VGA can rotate incoming CPU data up to seven bits to the right (toward the least significant bit), with bit 0 wrapping back to bit 7, after which the VGA continues processing the rotated byte just as it normally processes unrotated CPU data. Thanks to the nature of barrel shifters, this rotation requires no extra processing time over unrotated VGA operations. The number of bits by which CPU data is shifted is controlled by bits 2-0 of GC register 3, the Data Rotate register, which also contains the ALU function select bits (data unmodified, AND, OR, and XOR) that we looked at in the last chapter.
The barrel shifter is powerful, but (as sometimes happens in this business) it sounds more useful than it really is. This is because the GC can only rotate CPU data, a task that the CPU itself is perfectly capable of performing. Two OUTs are needed to select a given rotation: one to set the GC Index register, and one to set the Data Rotate register. However, with careful programming it’s sometimes possible to leave the GC Index always pointing to the Data Rotate register, so only one OUT is needed. Even so, it’s often easier and/or faster to simply have the CPU rotate the data of interest CL times than to set the Data Rotate register. (Bear in mind that a single OUT takes from 11 to 31 cycles on a 486—and longer if the VGA is sluggish responding to OUTs, as many VGAs are.) If only the VGA could rotate latched data, then there would be all sorts of useful applications for rotation, but, sadly, only CPU data can be rotated.

The drawing of bit-mapped text is one use for the barrel shifter, and I’ll demonstrate that application below. In general, though, don’t knock yourself out trying to figure out how to work data rotation into your programs—it just isn’t all that useful in most cases.

The Bit Mask

The VGA has bit mask circuitry for each of the four memory planes. The four bit masks operate in parallel and are all driven by the same mask data for each operation, so
they're generally referred to in the singular, as "the bit mask." Figure 25.2 illustrates the operation of one bit of the bit mask for one plane. This circuitry occurs eight times in the bit mask for a given plane, once for each bit of the byte written to display memory. Briefly, the bit mask determines on a bit-by-bit basis whether the source for each byte written to display memory is the ALU for that plane or the latch for that plane.

The bit mask is controlled by GC register 8, the Bit Mask register. If a given bit of the Bit Mask register is 1, then the corresponding bit of data from the ALUs is written to display memory for all four planes, while if that bit is 0, then the corresponding bit of data from the latches for the four planes is written to display memory unchanged. (In write mode 3, the actual bit mask that's applied to data written to display memory is the logical AND of the contents of the Bit Mask register and the data written by the CPU, as we'll see in Chapter 26.)

The most common use of the bit mask is to allow updating of selected bits within a display memory byte. This works as follows: The display memory byte of interest is latched; the bit mask is set to preserve all but the bit or bits to be changed; the CPU writes to display memory, with the bit mask preserving the indicated latched bits and allowing ALU data through to change the other bits. Remember, though, that it is not possible to alter selected bits in a display memory byte directly; the byte must first be latched by a CPU read, and then the bit mask can keep selected bits of the latched byte unchanged.

Listing 25.1 shows a program that uses the bit mask data rotation capabilities of the GC to draw bitmapped text at any screen location. The BIOS only draws characters
on character boundaries; in 640×480 graphics mode the default font is drawn on byte boundaries horizontally and every 16 scan lines vertically. However, with direct bitmapped text drawing of the sort used in Listing 25.1, it’s possible to draw any font of any size anywhere on the screen (and a lot faster than via DOS or the BIOS, as well).

LISTING 25.1 L25-1.ASM

; Program to illustrate operation of data rotate and bit mask features of Graphics Controller. Draws 8x8 character at specified location, using VGA's 8x8 ROM font. Designed for use with modes OOh, OEh, OFh, 10h, and 12h.
; By Michael Abrash.

stack segment para stack 'STACK'
  db 512 dup(?)
stack ends

VGA_VIDEO_SEGMENT equ 0a000h ;VGA display memory segment
SCREEN_WIDTH_IN_BYTES equ 044ah ;offset of BIOS variable
FONT_CHARACTER_SIZE equ 8 ;# bytes in each font char
;
; VGA register equates.
;
GC_INDEX equ 3ceh ;GC index register
GC_ROTATE equ 3 ;GC data rotate/logical function
GC_BIT_MASK equ 8 ;GC bit mask register index
;
dseg segment para common 'DATA'
TEST_TEXT_ROW equ 69 ;row to display test text at
TEST_TEXT_COL equ 17 ;column to display test text at
TEST_TEXT_WIDTH equ 8 ;width of a character in pixels

TestString label byte
db 'Hello, world!',0 ;test string to print.
FontPointer dd ? ;font offset
dseg ends
;
; Macro to set indexed register INDEX of GC chip to SETTING.
;
SETGC macro INDEX, SETTING
  mov dx,G C_INDEX
  mov ax,(SETTING SHL 8) OR INDEX
  out dx,ax
endm

; Select 640x480 graphics mode.
;
mov ax,012h
int 10h
;
; Set driver to use the 8x8 font.
;
mov ah,11h ;VGA BIOS character generator function.
mov al,30h ;return info subfunction
mov bh,3  ;get 8x8 font pointer
int 10h
 call SelectFont

; Print the test string.
;
 mov si,offset TestString
 mov bx,TEST_TEXT_ROW
 mov cx,TEST_TEXT_COL

StringOutLoop:
 lodsb
 and al,a1
 jz StringOutDone
 call DrawChar
 add cx,TEST_TEXT_WIDTH
 jmp StringOutLoop

StringOutDone:

; Reset the data rotate and bit mask registers.
;
 SETGC  GC_ROTATE, 0
 SETGC  GC_BIT_MASK, Offh

; Wait for a keystroke.
;
 mov ah,1
 int 21h

; Return to text mode.
;
 mov ax,03h
 int 10h

; Exit to DOS.
;
 mov ah,4ch
 int 21h

Start endp

; Subroutine to draw a text character in a linear graphics mode
; (ODh, OEh, OFh, 010h, 012h).
; Font used should be pointed to by FontPointer.
;
; Input:
; AL - character to draw
; BX - row to draw text character at
; CX - column to draw text character at
;
; Forces ALU function to "move".
;
 DrawChar proc near
 push ax
 push bx
 push cx
 push dx
 push si
 push di
 push bp
 push ds

 VGA Data Machinery  467
; Set DS:SI to point to font and ES to point to display memory.

lds si,[FontPointer] ;point to font
mov dx,VGA_VIDEO_SEGMENT
mov es,dx ;point to display memory

; Calculate screen address of byte character starts in.

push ds ;point to BIOS data segment
sub dx,dx
mov ds,dx
xchg ax,bx
mov di,ds:[SCREEN_WIDTH_IN_BYTES] ;retrieve BIOS
:screen width
pop ds
mul di ;calculate offset of start of row
push di ;set aside screen width
mov di,cx ;set aside the column
and cl,0111b ;keep only the column in-byte address
shr di,1
shr di,1
shr di,1 ;divide column by 8 to make a byte address
add di,ax ;and point to byte

; Calculate font address of character.

sub bh,bh
shl bx,1 ;assumes 8 bytes per character; use
shl bx,1 ;a multiply otherwise
shl bx,1 ;offset in font of character
add si,bx ;offset in font segment of character

; Set up the GC rotation.

mov dx,Gc_INDEX
mov al,Gc_ROTATE
mov ah,cl
out dx,ax

; Set up BH as bit mask for left half, 
; BL as rotation for right half.

mov bx,0ffffh
shr bh,cl
neg cl
add cl,8
shr bl,cl

; Draw the character, left half first, then right half in the 
; succeeding byte, using the data rotation to position the character 
; across the byte boundary and then using the bit mask to get the 
; proper portion of the character into each byte. 
; Does not check for case where character is byte-aligned and 
; no rotation and only one write is required.

mov bp,Font_CHARACTER_SIZE
mov dx,Gc_INDEX
pop cx ;get back screen width
dec cx
dec cx ;-2 because do two bytes for each char
The bit mask can be used for much more than bit-aligned fonts. For example, the bit mask is useful for fast pixel drawing, such as that performed when drawing lines, as
we'll see in Chapter 35. It's also useful for drawing the edges of primitives, such as filled polygons, that potentially involve modifying some but not all of the pixels controlled by a single byte of display memory.

Basically, the bit mask is handy whenever only some of the eight pixels in a byte of display memory need to be changed, because it allows full use of the VGA's four-way parallel processing capabilities for the pixels that are to be drawn, without interfering with the pixels that are to be left unchanged. The alternative would be plane-by-plane processing, which from a performance perspective would be undesirable indeed.

It's worth pointing out again that the bit mask operates on the data in the latches, not on the data in display memory. This makes the bit mask a flexible resource that with a little imagination can be used for some interesting purposes. For example, you could fill the latches with a solid background color (by writing the color somewhere in display memory, then reading that location to load the latches), and then use the Bit Mask register (or write mode 3, as we'll see later) as a mask through which to draw a foreground color stencilled into the background color without reading display memory first. This only works for writing whole bytes at a time (clipped bytes require the use of the bit mask; unfortunately, we're already using it for stencilling in this case), but it completely eliminates reading display memory and does foreground-plus-background drawing in one blurry-fast pass.

---

**Tip**

This last-described example is a good illustration of how I'd suggest you approach the VGA: As a rich collection of hardware resources that can profitably be combined in some non-obvious ways. Don't let yourself be limited by the obvious applications for the latches, bit mask, write modes, read modes, map mask, ALUs, and set/reset circuitry. Instead, try to imagine how they could work together to perform whatever task you happen to need done at any given time. I've made my code as much as four times faster by doing this, as the discussion of Mode X in Chapters 47–49 demonstrates.

The example code in Listing 25.1 is designed to illustrate the use of the Data Rotate and Bit Mask registers, and is not as fast or as complete as it might be. The case where text is byte-aligned could be detected and performed much faster, without the use of the Bit Mask or Data Rotate registers and with only one display memory access per font byte (to write the font byte), rather than four (to read display memory and write the font byte to each of the two bytes the character spans). Likewise, non-aligned text drawing could be streamlined to one display memory access per byte by having the CPU rotate and combine the font data directly, rather than setting up the VGA's hardware to do it. (Listing 25.1 was designed to illustrate VGA data rotation and bit masking rather than the fastest way to draw text. We'll see faster text-drawing code soon.) One excellent rule of thumb is to minimize display memory accesses of all types, especially reads, which tend to be considerably slower than writes. Also, in
Listing 25.1 it would be faster to use a table lookup to calculate the bit masks for the two halves of each character rather than the shifts used in the example.

For another (and more complex) example of drawing bit-mapped text on the VGA, see John Cockerham's article, "Pixel Alignment of EGA Fonts," *PC Tech Journal*, January, 1987. Parenthetically, I’d like to pass along John’s comment about the VGA: “When programming the VGA, everything is complex.” He’s got a point there.

**The VGA’s Set/Reset Circuitry**

At last we come to the final aspect of data flow through the GC on write mode 0 writes: the set/reset circuitry. Figure 25.3 shows data flow on a write mode 0 write. The only difference between this figure and Figure 25.1 is that on its way to each plane potentially the rotated CPU data passes through the set/reset circuitry, which may or may not replace the CPU data with set/reset data. Briefly put, the set/reset circuitry enables the programmer to elect to independently replace the CPU data for each plane with either 00 or OFFH.

What is the use of such a feature? Well, the standard way to control color is to set the Map Mask register to enable writes to only those planes that need to be set to produce...
the desired color. For example, the Map Mask register would be set to 09H to draw in high-intensity blue; here, bits 0 and 3 are set to 1, so only the blue plane (plane 0) and the intensity plane (plane 3) are written to.

Remember, though, that planes that are disabled by the Map Mask register are not written to or modified in any way. This means that the above approach works only if the memory being written to is zeroed; if, however, the memory already contains non-zero data, that data will remain in the planes disabled by the Map Mask, and the end result will be that some planes contain the data just written and other planes contain old data. In short, color control using the Map Mask does not force all planes to contain the desired color. In particular, it is not possible to force some planes to zero and other planes to one in a single write with the Map Mask register.

The program in Listing 25.2 illustrates this problem. A green pattern (plane 1 set to 1, planes 0, 2, and 3 set to 0) is first written to display memory. Display memory is then filled with blue (only plane 0 set to 1), with a Map Mask setting of 01H. Where the blue crosses the green, cyan is produced, rather than blue, because the Map Mask register setting of 01H that produces blue leaves the green plane (plane 1) unchanged. In order to generate blue unconditionally, it would be necessary to set the Map Mask register to 0FH, clear memory, and then set the Map Mask register to 01H and fill with blue.

**LISTING 25.2 L25-2.ASM**

: Program to illustrate operation of Map Mask register when drawing to memory that already contains data.
: By Michael Abrash.
:
stack segment para stack 'STACK'
db 512 dup(?)
:
EGA_VIDEO_SEGMENT equ 0a000h ;EGA display memory segment
:
;EGA register equates.
;
SC_INDEX equ 3c4h ;SC index register
SC_MAP_MASK equ 2 ;SC map mask register
;
: Macro to set indexed register INDEX of SC chip to SETTING.
:
SETSC macro INDEX, SETTING
move dx,SC_INDEX
move ah,INDEX
out dx,al
inc dx
move ah,SETTING
out dx,al
dec dx
eendm
:
cseg segment para public 'CODE'
assume cs:cseg

472 Chapter 25
Setting All Planes to a Single Color

The set/reset circuitry can be used to force some planes to 0-bits and others to 1-bits during a single write, while letting CPU data go to still other planes, and so provides an efficient way to set all planes to a desired color. The set/reset circuitry works as follows:

```
start    proc    near

; Select 640x480 graphics mode.

    mov     ax,012h
    int     10h

    mov     ax,EGA_VIDEO_SEGMENT
    mov     es,ax        ; point to video memory

; Draw 24 10-scan-line high horizontal bars in green, 10 scan lines apart.

    SETSC   SC_MAP_MASK,02h       ; map mask setting enables only
                               ; plane 1. the green plane
    sub     di,di                 ; start at beginning of video memory
    mov     al,0ffh
    mov     bp,24                 ; # bars to draw

HorzBarLoop:
    mov     cx,80*10              ; # bytes per horizontal bar
    rep     stosb                 ; draw bar
    add     di,80*10              ; point to start of next bar
    dec     bp
    jnz     HorzBarLoop

; Fill screen with blue, using Map Mask register to enable writes
; to blue plane only.

    SETSC   SC_MAP_MASK,01h       ; map mask setting enables only
                               ; plane 0. the blue plane
    sub     di,di                 ; start at beginning of video memory
    mov     cx,80*480              ; # bytes per screen
    mov     al,0ffh
    rep     stosb                 ; perform fill (affects only
                               ; plane 0, the blue plane)

; Wait for a keystroke.

    mov     ah,1
    int     21h

; Restore text mode.

    mov     ax,03h
    int     10h

; Exit to DOS.

    mov     ah,4ch
    int     21h

start  endp

cseg    ends

end
```
For each of the bits 0-3 in the Enable Set/Reset register (Graphics Controller register 1) that is 1, the corresponding bit in the Set/Reset register (GC register 0) is extended to a byte (0 or OFFH) and replaces the CPU data for the corresponding plane. For each of the bits in the Enable Set/Reset register that is 0, the CPU data is used unchanged for that plane (normal operation). For example, if the Enable Set/Reset register is set to 01H and the Set/Reset register is set to 05H, then the CPU data is replaced for plane 0 only (the blue plane), and the value it is replaced with is OFFH (bit 0 of the Set/Reset register extended to a byte). Note that in this case, bits 1-3 of the Set/Reset register have no effect.

It is important to understand that the set/reset circuitry directly replaces CPU data in Graphics Controller data flow. Refer back to Figure 25.3 to see that the output of the set/reset circuitry passes through (and may be transformed by) the ALU and the bit mask before being written to memory, and even then the Map Mask register must enable the write. When using set/reset, it is generally desirable to set the Map Mask register to enable all planes the set/reset circuitry is controlling, since those memory planes which are disabled by the Map Mask register cannot be modified, and the purpose of enabling set/reset for a plane is to force that plane to be set by the set/reset circuitry.

Listing 25.3 illustrates the use of set/reset to force a specific color to be written. This program is the same as that of Listing 25.2, except that set/reset rather than the Map Mask register is used to control color. The preexisting pattern is completely overwritten this time, because the set/reset circuitry writes 0-bytes to planes that must be off as well as OFFH-bytes to planes that must be on.

**LISTING 25.3  L25-3.ASM**

: Program to illustrate operation of set/reset circuitry to force setting of memory that already contains data.
: By Michael Abrash.

: stack segment para stack 'STACK'
  db 512 dup(?)
stack ends
:
EGA_VIDEO_SEGMENT equ 0a000h ;EGA display memory segment
:
: EGA register equates.
:
SC_INDEX equ 3c4h ;SC index register
SC_MAP_MASK equ 2 ;SC map mask register
GC_INDEX equ 3ceh ;GC index register
GC_SET_RESET equ 0 ;GC set/reset register
GC_ENABLE_SET_RESET equ 1 ;GC enable set/reset register
:
: Macro to set indexed register INDEX of SC chip to SETTING.
:
SETSC macro INDEX, SETTING
  mov dx,SC_INDEX
  mov al,INDEX
  out dx,al

474 Chapter 25
inc dx
mov al,SETTING
out dx,al
dec dx
endm

; Macro to set indexed register INDEX of GC chip to SETTING.

SETGC macro INDEX, SETTING
mov dx,GC_INDEX
mov al,INDEX
out dx,al
inc dx
mov al,SETTING
out dx,al
dec dx
endm

cseg segment para public 'CODE'
assume cs:cseg
start proc near

; Select 640x480 graphics mode.

mov ax,012h
int 10h

mov ax,EQAVIDEO_SEGMENT
mov es,ax ;point to video memory

; Draw 24 10-scan-line high horizontal bars in green, 10 scan lines apart.

SETSC SC_MAP_Mask,02h ;map mask setting enables only
sub di,di ;start at beginning of video memory
mov al,0ffh
mov bp,24 ;# bars to draw
HorzBarLoop:
mov cx,80*10 ;# bytes per horizontal bar
rep stosb ;draw bar
add di,80*10 ;point to start of next bar
dec bp
jnz HorzBarLoop

; Fill screen with blue, using set/reset to force plane 0 to 1's and all
; other plane to 0's.

SETSC SC_MAP_Mask,0fh ;must set map mask to enable all
planes, so set/reset values can
be written to memory
SETGC GC_ENABLE_SET_RESET,0fh ;CPU data to all planes will be
replaced by set/reset value
SETGC GC_SET_RESET,01h ;set/reset value is 0ffh for plane 0
;(the blue plane) and 0 for other
;planes
sub di,di
mov cx,80*480 ;# bytes per screen
mov al,0ffh ;since set/reset is enabled for all
planes, the CPU data is ignored-
;only the act of writing is
;important
Manipulating Planes Individually

Listing 25.4 illustrates the use of set/reset to control only some, rather than all, planes. Here, the set/reset circuitry forces plane 2 to 1 and planes 0 and 3 to 0. Because bit 1 of the Enable Set/Reset register is 0, however, set/reset does not affect plane 1; the CPU data goes unchanged to the plane 1 ALU. Consequently, the CPU data can be used to control the value written to plane 1. Given the settings of the other three planes, this means that each bit of CPU data that is 1 generates a brown pixel, and each bit that is 0 generates a red pixel. Writing alternating bytes of 07H and 0E0H, then, creates a vertically striped pattern of brown and red.

In Listing 25.4, note that the vertical bars are 10 and 6 bytes wide, and do not start on byte boundaries. Although set/reset replaces an entire byte of CPU data for a plane, the combination of set/reset for some planes and CPU data for other planes, as in the example above, can be used to control individual pixels.

LISTING 25.4 L25-4.ASM

; Program to illustrate operation of set/reset circuitry in conjunction with CPU data to modify setting of memory that already contains data.
; By Michael Abrash.

stack segment para stack 'STACK'
    db 512 dup(?)
stack ends

EGA_VIDEO_SEGMENT equ 0a000h ;EGA display memory segment

SC_INDEX equ 3c4h ;SC index register
SC_MAP_MASK equ 2 ;SC map mask register
GC_INDEX equ 3ceh ;GC index register
GC_SET_RESET equ 0 ;GC set/reset register
GC_ENABLE_SET_RESET equ 1 ;GC enable set/reset register

; Macro to set indexed register INDEX of SC chip to SETTING.
;
SETSC macro INDEX, SETTING
    mov dx, SC_INDEX
    mov al, INDEX
    out dx, al
    inc dx
    mov al, SETTING
    out dx, al
    dec dx
endm

; Macro to set indexed register INDEX of GC chip to SETTING.
;
SETGC macro INDEX, SETTING
    mov dx, GC_INDEX
    mov al, INDEX
    out dx, al
    inc dx
    mov al, SETTING
    out dx, al
    dec dx
endm

cseg segment para public 'CODE'
assume cs:cseg
start proc near
;
; Select 640x350 graphics mode.
;
    mov ax, 010h
    int 10h
;
    mov ax, EGA_VIDEO_SEGMENT
    mov es, ax ;point to video memory
;
; Draw 18 10-scan-line high horizontal bars in green, 10 scan lines apart.
;
    SETSC SC_MAP_MASK,OEh ;map mask setting enables only plane 1, the green plane
    sub di, df ;start at beginning of video memory
    mov a1, 0ffh
    mov bp, 18 ;# bars to draw
    HorzBarLoop:
        mov cx, 80*10 ;# bytes per horizontal bar
        rep stosb ;draw bar
        add di, 80*10 ;point to start of next bar
        dec bp
    jnz HorzBarLoop
;
; Fill screen with alternating bars of red and brown, using CPU data
to set plane 1 and set/reset to set planes 0, 2 & 3.

VGA Data Machinery 477
There is no clearly defined role for the set/reset circuitry, as there is for, say, the bit mask. In many cases, set/reset is largely interchangeable with CPU data, particularly with CPU data written in write mode 2 (write mode 2 operates similarly to the set/reset circuitry, as we’ll see in Chapter 27). The most powerful use of set/reset, in my experience, is in applications such as the example of Listing 25.4, where it is used to force the value written to certain planes while the CPU data is written to other planes. In general, though, think of set/reset as one more tool you have at your disposal in getting the VGA to do what you need done, in this case a tool that lets you force all bits in each plane to either zero or one, or pass CPU data through unchanged, on each write to display memory. As tools go, set/reset is a handy one, and it’ll pop up often in this book.

Notes on Set/Reset

The set/reset circuitry is not active in write modes 1 or 2. The Enable Set/Reset register is inactive in write mode 3, but the Set/Reset register provides the primary drawing color in write mode 3, as discussed in the next chapter.
Be aware that because set/reset directly replaces CPU data, it does not necessarily have to force an entire display memory byte to 0 or 0FFH, even when set/reset is replacing CPU data for all planes. For example, if the Bit Mask register is set to 80H, the set/reset circuitry can only modify bit 7 of the destination byte in each plane, since the other seven bits will come from the latches for each plane. Similarly, the set/reset value for each plane can be modified by that plane's ALU. Once again, this illustrates that set/reset merely replaces the CPU data for selected planes; the set/reset value is then processed in exactly the same way that CPU data normally is.

A Brief Note on Word OUTs

In the early days of the EGA and VGA, there was considerable debate about whether it was safe to do word OUTs (OUT DX,AX) to set Index/Data register pairs in a single instruction. Long ago, there were a few computers with buses that weren't quite PC-compatible, in that the two bytes in each word OUT went to the VGA in the wrong order: Data register first, then Index register, with predictably disastrous results. Consequently, I generally wrote my code in those days to use two 8-bit OUTs to set indexed registers. Later on, I made it a habit to use macros that could do either one 16-bit OUT or two 8-bit OUTs, depending on how I chose to assemble the code, and in fact you'll find both ways of dealing with OUTs sprinkled through the code in this part of the book. Using macros for word OUTs is still not a bad idea in that it does no harm, but in my opinion it's no longer necessary. Word OUTs are standard now, and it's been a long time since I've heard of them causing any problems.