Feasibility of Hardware Sprite Acceleration with CoCoVGA

Brendan Donahe and Steve Spiller

April 1, 2018

Abstract

The Tandy/TRS-80 Color Computer's Motorola 6847 Video Display Generator (VDG) has long lacked the ability to accelerate graphics in the form of overlays or sprites. The CPU-cycle expense of drawing such sprites on a per-frame basis by software is high. Many systems subsequent to the Color Computer provide such hardware acceleration and, upon examination of their specs, can provide insight as to what aspects are useful to the programmer.

CoCoVGA is an upgrade for the Color Computer which already provides other types of color and text enhancements via attachment at the 6847's socket. Consideration of possible modifications to CoCoVGA's video display pipeline along with a potential methodology for uploading and manipulating sprites without sacrificing display refresh rate reveal that it is highly likely that a single sprite could be implemented in CoCoVGA. Beyond a single sprite, transparency makes the algorithm much more logic- and register-intensive, but still probable, despite the current > 60% utilization of the onboard Altera Cyclone IV EP4CE6.

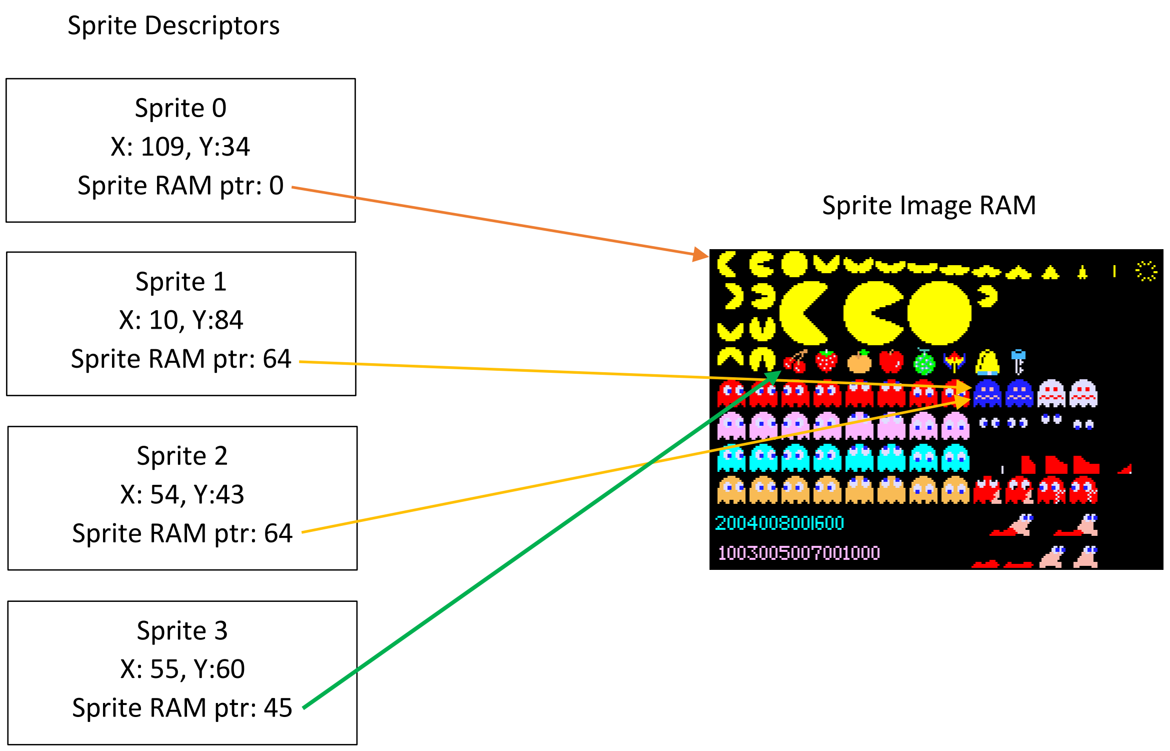
Introduction

The original TRS-80 Color Computer, a popular home computer from the early 1980s, has long relied heavily on its 8/16-bit processor, the Motorola 6809E, for graphics manipulation. That is, even by the standards of the day, set by the likes of Commodore and Atari, the video subsystem, driven by the Motorola 6847 VDG, was somewhat limited in a number of ways.

The Color Computer's 6809E processor has a rich, orthogonal instruction set which enables a programmer to copy rectangular blocks of pixel data in an indexed manner using the stack. This approach is often called "stack blasting". Color Computer programmers frequently use this method for their games and application software where graphical animation and movement are required. It is employed in such a way that any potentially-in-motion graphical objects (such as a mouse pointer or a video game character) are drawn onto the display during the vertical blanking region or to another graphics page which will be displayed following the next vsync. An example of stack blasting will be provided later in this document.

In contrast, hardware sprites are rectangular regions of pixels which are drawn atop the background image or other sprites (depending on a priority or depth order) automatically by the graphics hardware. The sprite images themselves are placed in particular locations in memory (a sprite RAM or ROM) such that the graphics hardware can index into these and examine them along with the background pixel values before deciding which pixel color to display. In other words, the sprite images are not copied onto the background image as it exists in RAM as we would do with software, but yet via the graphics hardware, they appear on the display when it selects between the sprite memory or the background image memory at the appropriate point while drawing the display pixel by pixel.

In the example below, sprite descriptors 1 and 2 are each referencing the same “scared ghost” sprite image, but note that they each have very different (X, Y) coordinates on screen. Therefore, 2 copies of the scared ghost would be output, each at their respective coordinates, by the graphics hardware.



The Commodore 64, Atari 400/800, and later, the Nintendo Entertainment System (NES) and Nintendo Game Boy all had some type of overlays or sprites handled by the hardware. Some examples of these are as follows:

System	**Commodore 64**	**Nintendo Entertainment System**	**Nintendo Game Boy**
Total sprites	8 (max of 8 per scanline)	64 (max of 8 per scanline)	40 (max of 10 per scanline)
Sprite size and color depth (in bits-per-pixel)	24x21 pixels @ 1bpp 12x21 pixels @ 2bpp	8x8 or 8x16 pixels @ 2bpp	8x8 or 8x16 pixels @ 2bpp
Sprite transparency	yes	yes	yes
Sprites “reusable” or “multiplexable” within single display	yes	yes	yes
Sprite mirroring	no	yes, both x and y	yes, both x and y

Typically, color index 0 is considered transparent such that the background or other sprites can be seen through a foreground, higher priority sprite. This enables a non-rectangular video game character or mouse cursor, for example, to travel across the background without obscuring the entire rectangular region around the sprite. For the 2bpp (bits-per-pixel) configurations above, 4 color indices are available (00, 01, 10, and 11), one of which would be transparent and the other 3 would be visible.

Mirroring enables a programmer to conserve sprite RAM by "flipping" one sprite's image either left-to-right or top-to-bottom, or both. For example, a video game character might need to travel both left or right, in which case it would face one way when travelling left and face the other way when travelling right. The example image shown above suggests that the Pac-Man arcade hardware did not have the ability to do this since there are orientations of Pac-Man pointing up, down, left, and right.

Reusability or multiplexing of sprites means that it is possible to take sprites which have already been fully displayed earlier on the video screen and, while the screen is still being drawn top-to-bottom, reprogram them to reappear again, with different parameters (X, Y, or image) later in the sweep. Timing the reprogramming of these sprite descriptors carefully can make it appear as though a system has many more sprites than the hardware actually supports.

Given hardware sprite acceleration examples such as the above, CoCoVGA, attached to the Color Computer by way of the 6847 socket and driving out VGA video signals, is in a reasonable position to provide software with the hardware assistance necessary to offload the CPU from various solutions such as stack-blasting. We will examine the CPU cycle cost of stack-blasting a sprite onto a blank canvas, and then evaluate what the software expense would be to communicate this to a sprite accelerator embedded within CoCoVGA, as well as the feasibility of implementing said accelerator.

Current State of the Art

Stack Blasting:

Since the Color Computer has no hardware sprite overlay ability, let us examine one of the more efficient methods for enabling software to draw sprites on the top of a background image. Here is an example 10x8 pixel x 4bpp space invader, rendered in VG6, a 16-color 128x96 video mode supported by CoCoVGA:

```
INVADER1A       fcb             $10,$10,$01,$00,$01 ;  * *  * *
                fcb             $01,$00,$10,$11,$00 ;   * ** *
                fcb             $10,$00,$01,$00,$00 ;    *  *
                fcb             $11,$10,$11,$11,$01 ;  ********
                fcb             $01,$10,$10,$11,$01 ;  ** ** **
                fcb             $11,$00,$11,$11,$00 ;   ******
                fcb             $10,$00,$01,$11,$00 ;    ****
                fcb             $00,$00,$00,$11,$00 ;     **
```

Note that the invader appears upside-down and with its bytes and nybbles in a somewhat scrambled order due to the particular stack-blasting algorithm chosen in this example. The character is drawn upside down and from right to left since pushing onto the stack causes the stack pointer to decrement. Below is the innermost loop of a 6809 stack-blasting routine. Note that prior to this code segment, the X index must point to the start of the sprite to be copied (in this example, the address `INVADER1A`), and U must point to the location in video RAM where the bottom right of the sprite should be drawn.

```
sb1
        ldy     ,x++    ; load 2 bytes (4 pixels) from sprite memory into Y, incrementing X by 2
        ldd     ,x++    ; load 2 bytes (4 pixels) from sprite memory into D, incrementing X by 2
        pshu    d,y     ; push D and Y onto the U stack
        lda     ,x+     ; load 1 byte (2 pixels) from sprite memory into A, incrementing X by 1
        pshu    a       ; push A onto the U stack
        cmpu    SPREND  ; have we reached the end of the sprite memory region?
        ble     sbd     ;   if so, return from this function
        tfr     u,d     ; otherwise, copy the U stack pointer to D
        subd    #59     ; decrement it by 59 bytes (118 pixels of 128 pixel-wide video mode)
        tfr     d,u     ; copy D back to U
        bra     sb1     ; loop back to blast next row of pixels from right to left
sbd     rts
```

For our purposes, the routine above takes 69 cycles to copy a row of 10 pixels from the sprite RAM into video RAM:

Instruction	Addressing Mode	Cycles
ldy	indexed with double increment	6+3
ldd	indexed with double increment	5+3
pshu	4 bytes	5+4
lda	indexed with increment	4+2
pshu	1 byte	5+1
cmpu	direct	7
ble	relative	3
tfr	immediate	6
subd	immediate	6
tfr	immediate	6
bra	relative	3
Total per row of pixels		69

That means that it takes 69*8 = 552 cycles to draw the 10x8 x 16-color sprite. It has the additional drawback that it has no “transparent” color, so it cannot render over the top of a patterned background. Doing this would require additional CPU cycles to read/modify/write the background composited with the sprite anywhere the sprite was transparent. Note that the above algorithm, if applied to an 8x8 pixel region, could drop the lda/pshu pair, saving 12 cycles. This means that an 8x8 pixel region still takes 57*8 = 456 cycles for the CPU to draw.

Given the above algorithm on the 10x8 pixel region, we are limited to:

$$(1/60\ \text{Hz}) / ((1/0.895\ \text{MHz}) * 552\ \text{cycles}) = 27\ \text{sprites}$$

per vertical refresh, assuming we wish to animate and potentially relocate that many graphics objects. Certainly, this takes no game logic or audio into account, nor the time it takes to either clear an initial background screen or erase the existing characters from the current background screen before placing them again, but it does provide an upper bound for reference.

Example Hardware Acceleration:

By comparison, the Nintendo Game Boy, released in 1989, is a portable, battery-powered game system which has a 160x144 pixel by 4-level grayscale display (2 bits per pixel). It supports 40 total sprites, each 8x8 or 16x8 pixels, 10 of which can be displayed in any scanline. Of the 4-levels of gray, “color” 0 is considered transparent. Each sprite is described by the following parameters:

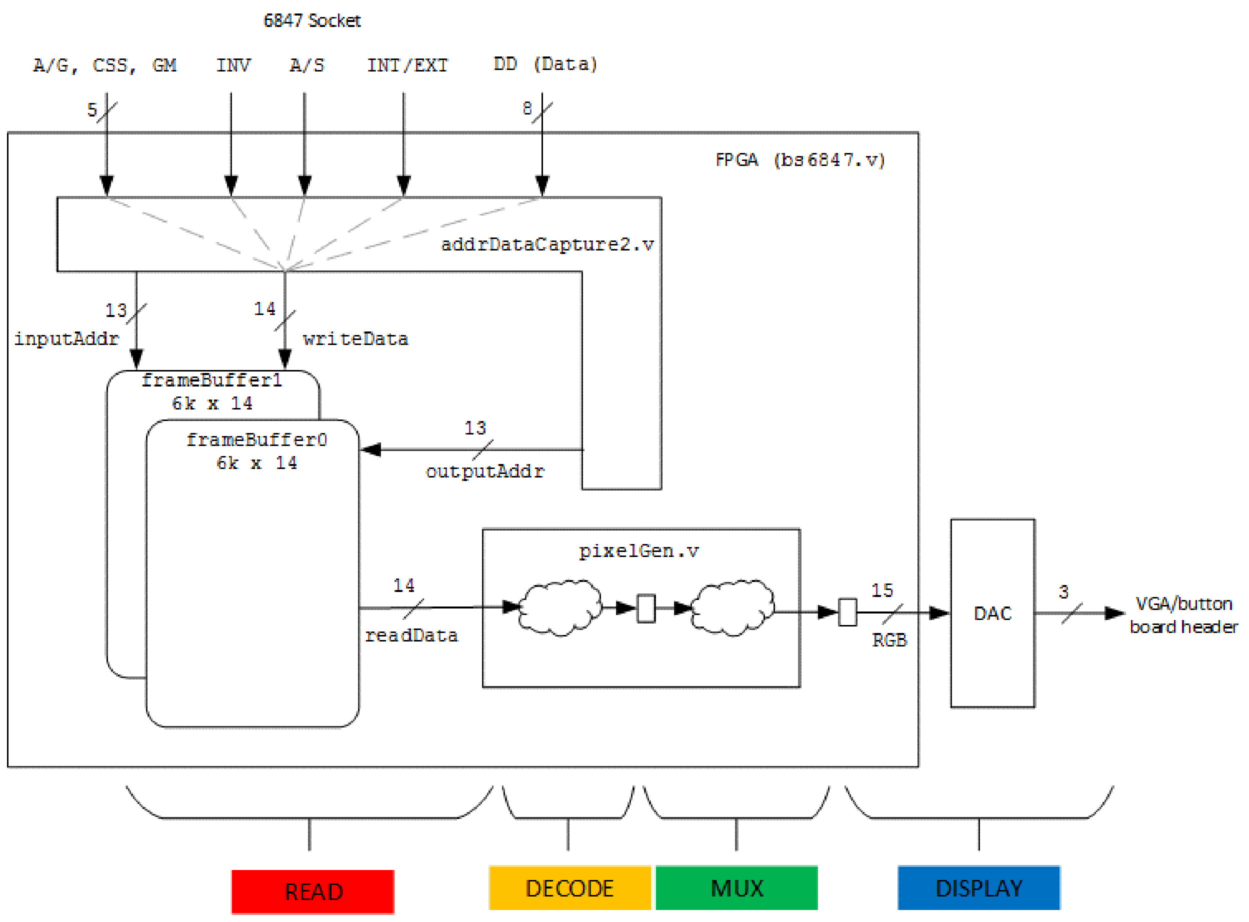
- 2 byte (x,y) location on screen
- 1 byte pattern number (0-255), which is basically a pointer into the sprite image memory
- 1 bit priority, that is, whether the sprite should be displayed on top or behind the background
- 1 bit y flip, which optionally mirrors the sprite image vertically
- 1 bit x flip, which optionally mirrors the sprite image horizontally
- 1 bit palette select, so each sprite can use either one of the two 4-grayscale palettes

Once the sprite image memory is initialized, only the above registers must be updated for action to occur on-screen.

Areas for Improvement

Current CoCoVGA FPGA Design:

In order to consider the addition of sprite acceleration to the CoCoVGA FPGA design, we should review its current implementation. The first release of the CoCoVGA FPGA commonly available to the public was revision 0.87. The sections pertinent to this discussion are depicted here:



The CoCoVGA FPGA design contains 2 6k x 14-bit frame buffers. Video data fetched by the 6847 as well as mode at its input pins are written into one frame buffer while the other is simultaneously read by the VGA display engine. This double-buffering or ping-pong buffering technique is employed to avoid tearing that would appear on the VGA display given the inexact frame timing (i.e. asynchronous interface) between the VGA display and the 6847's native refresh rate of ~60Hz (NTSC) or ~50Hz (PAL). There are situations in which both the write and read pointers are in the same buffer, however, the VGA refresh rate is slightly faster than the 6847's, keeping the VGA read pointer ahead of any writes. Note that this will, occasionally, cause frames to be repeated on the VGA display. Additionally, there will always be between 1/60 to 2/60 (1/50 to 2/50 for PAL) of a second delay between the data being written as it appears at the 6847 socket and when it is delivered to the VGA output.

Bit-banging a series of 4 5-bit values into CoCoVGA by way of the A/G (alphanumeric/graphics), CSS (color set select), and GM[2:0] (the 3-bit graphics mode) pins provides the ability to access its write-only soft switches and palette registers. This "combination lock" is only recognized during the 6847's vertical blanking region, making it extremely unlikely that software would accidentally unlock this feature. Once this secret code is entered, this instructs CoCoVGA to comprehend the upcoming frame as register settings instead of video data. This frame is not stored for display by CoCoVGA's double-buffering technique. Instead, it will show the already fully buffered and previously displayed video frame during this time and resume normal operation at the start of the next 6847 vertical blanking region, when it will start buffering the video data as before. Currently, the only page of soft switches and registers implemented in the FPGA reside in page 0.

In this revision, the video display pipeline following the frame buffers is as follows for all video except RG6 (PMODE 4), if employing a complex artifact color generation mode such as MESS/MAME or smArtifact:

	pixelClk n	pixelClk n+1	pixelClk n+2	pixelClk n+3	pixelClk n+4	pixelClk n+5
VGA pixel n	READ	DECODE	MUX	DISPLAY		
VGA pixel n+1		READ	DECODE	MUX	DISPLAY	
VGA pixel n+2			READ	DECODE	MUX	DISPLAY

In the diagram above, the READ stage's input is a combinatorial calculation of the next address of interest in the video buffer RAM based mostly on VDG mode, but also takes into account various configuration options such as 64-column text mode.

Output from the video buffer RAM is handled in the DECODE stage. Based on 6847 and enhanced mode configuration, CoCoVGA combinatorially selects either a hardcoded or software-defined color from the enabled palette.

The MUX stage selects between border and pixel colors, depending on what part of the display is currently being drawn, producing 15-bit red/green/blue data. (Note that in the earliest boards such as the AMC2, only 9 of these bits are used, the 3 most significant of each color.)

To allow for the best quality video output, the 15-bit RGB data is flopped one last time. The DISPLAY stage is directly output from these flops within the FPGA to the off-chip resistor-ladder video DAC (Digital to Analog Converter) to be driven to the VGA display.

With complex artifact color generation, prefetch of the previous byte is required during READ in order to apply the 6-tap filter across all bits, and then requires many more delay pipeline stages (an additional 12 cycles within MUX) before DISPLAY of the pixel following decode.

Description of Improvements

Software Interface Considerations:

Given the ability by software to upload data to CoCoVGA's soft switches during a frame's time, it makes sense to employ an additional page or pages to upload the sprite pixel data into the sprite image RAM. Whereas page 0 (an alphanumeric 32x16 byte mode) was used for some of the simpler soft switches and palette registers, a higher-resolution CG6 or RG6 page may be preferable due to the sheer amount of data that must be uploaded to CoCoVGA's sprite image RAM. Since the page is selected by the 5 mode pins of the 6847 following the combination lock, there are 32 possible pages we can select from. This will be revisited later once we estimate the number and size of each image to be placed in the sprite image RAM. Because of the frame latency involved with this approach, this upload operation must be performed sparingly.

However, a new approach for updating the sprite configuration information (X/Y, palette select, etc.) must be utilized in order to enable movement of sprites at the full frame rate of 1/60th of a second. Investigating the feasibility of updating such information via bit-banging the 5 GM, CSS, and A/G pins, figure 8 of the 6847 spec provides the duration of the vertical blank:

tWFS = 32 * tPHST = 32 * (227.5*1/f) where f is the 6847 clock at 3.579545 MHz

tWFS = ~2.033ms

The CPU clock on the Color Computer 1 and 2 runs at 0.895 MHz, which is a period of ~1.11us. This means that the CPU has:

tWFS/1.11us ~= 1831 CPU clock cycles

in which to unlock CoCoVGA and provide additional sprite programming information by way of the 6847 mode pins.

Because these 5 6847 pin controls are contained within the same byte as 3 other PIA (Peripheral Interface Adapter) bits at address $FF22, the data to be written to this register should be carefully formatted, leaving the 3 least significant bits unmodified:

7	6	5	4	3	2	1	0
6847 A/G - alphanumeric vs graphics	6847 GM[2] - MSB of graphics mode	6847 GM[1]	6847 GM[0] - LSB of graphics mode	6847 CSS - color set select	RAM size input	Single-bit sound output	RS-232C data input

If the X index register is pointed to the start of a pre-calculated combination lock and sprite information array, a code sequence such as the one below could be used repetitively to write it to the PIA. The 6809E reference manual lists the cycle counts for these operations:

Instruction	Addressing Mode	Cycles
lda ,x+	indexed with increment	6
sta $ff22	direct	4
Total per VDG mode pin write		10

This means that 1831 CPU cycles/10 CPU cycles per VDG pin write = 183 VDG mode writes during the vertical blanking region are possible, assuming that the outer loop was unrolled.

Since the CSS, A/G, and GM pins are exactly that and not registers, there is no way for CoCoVGA to determine whether a new write has occurred unless at least one bit of that interface changes. For that reason, one of these five bits must toggle on every write to make it known that the data has changed. The most straightforward way to accomplish this is to assign one of the five bits for this purpose, leaving a nybble to carry sprite parameter data for each write. At 4 bits per VDG mode write, that means a theoretical maximum of 183 nybbles = 732 bits (91 bytes) of information could potentially be written to CoCoVGA during the vertical blanking region, including a new 4 entry (2 byte) combination lock, leaving 89 bytes.

The sprite descriptor stream can be designed in such a way as to enable software to update any sprite by providing the IDs of the n sprites to modify:

Byte number	Sprite ID	Data
0	j	ID (j) – not stored, only used to index into registers
1	j	flags (enable, X/Y mirror controls, palette select, size) – 5 bits
2	j	X location – 8 bits
3	j	Y location – 8 bits
4	j	image pointer – 7 bits
5	k	ID (k)
6	k	flags
7	k	X location
8	k	Y location
9	k	image pointer
…	…	…
5n	N/A	ID $ff to terminate

That means 5 bytes per sprite descriptor update plus an additional termination byte of ID $ff, limits us to updating a maximum of 88/5 = 17 sprites' parameter sets per vertical blanking region. In the above configuration, each set of sprite parameters will consume 28 bits. A special state machine will be required to save these values into the appropriate location as they are written one nybble-at-a-time.

Given that 2600 logic elements (LEs) are still free in the CoCoVGA FPGA design as of revision 0.87, and each LE contains a single 1-bit flip-flop, this provides an upper bound for the number of sprite descriptor registers that can be implemented. It is highly unlikely to achieve full utilization of each LE, so it is almost guaranteed that we will be unable to access each of these LEs' flops. A conservative utilization estimate of these remaining logic elements' flip-flops would be around 1/3, or approximately 900. In this situation, our upper limit of sprites would be around 32 (28 bits * 32 sprites = 896). Since more than 32 sprites are desirable, strong consideration should be given to storing this information in a RAM, which might enable up to 64 sprites (16-bit wide RAM * 2 entries * 64 sprites = 2kbytes). Certainly, this requires more logic and time to populate as well as to extract the data from. The 20 CPU cycles per byte required to provide each of the above listed parameters, a state machine should be able to use the initial ID byte of each sprite to index into the RAM and write the data in appropriate locations as it arrives without issue.

Display Pipeline and Caching:

Given the previously described pipeline stage, if we first consider a single sprite, we must perform additional operations in these stages:

- READ
  1. calculate X/Y coordinate that is about to be displayed
  2. compare X/Y against sprite descriptor register values containing current sprite locations
  3. if a match is found, calculate addresses into sprite image RAM
  4. start sprite image RAM read
- DECODE
  5. select between original (background) pixel color and sprite (taking transparency into account)

6. calculate pixel color

- MUX

7. no change

- DISPLAY

8. no change

As soon as multiple sprites can access the same sprite image RAM, the above method becomes unusable because there are simply not enough cycles to perform multiple RAM lookups into the same sprite image RAM while the scanline is being drawn. Therefore, to implement multiple sprites, it becomes necessary to cache sprite row data. During the VGA horizontal blanking region (that is, between the visible parts of scanlines), the hardware can sequence through each set of sprite parameters to determine whether the sprite will be visible on the upcoming scanline. If it is visible it must:

1. index into the location of sprite image RAM for the appropriate row of the specified sprite
2. read that row of sprite RAM into flops for zero-cycle-access later when the scanline row is being drawn

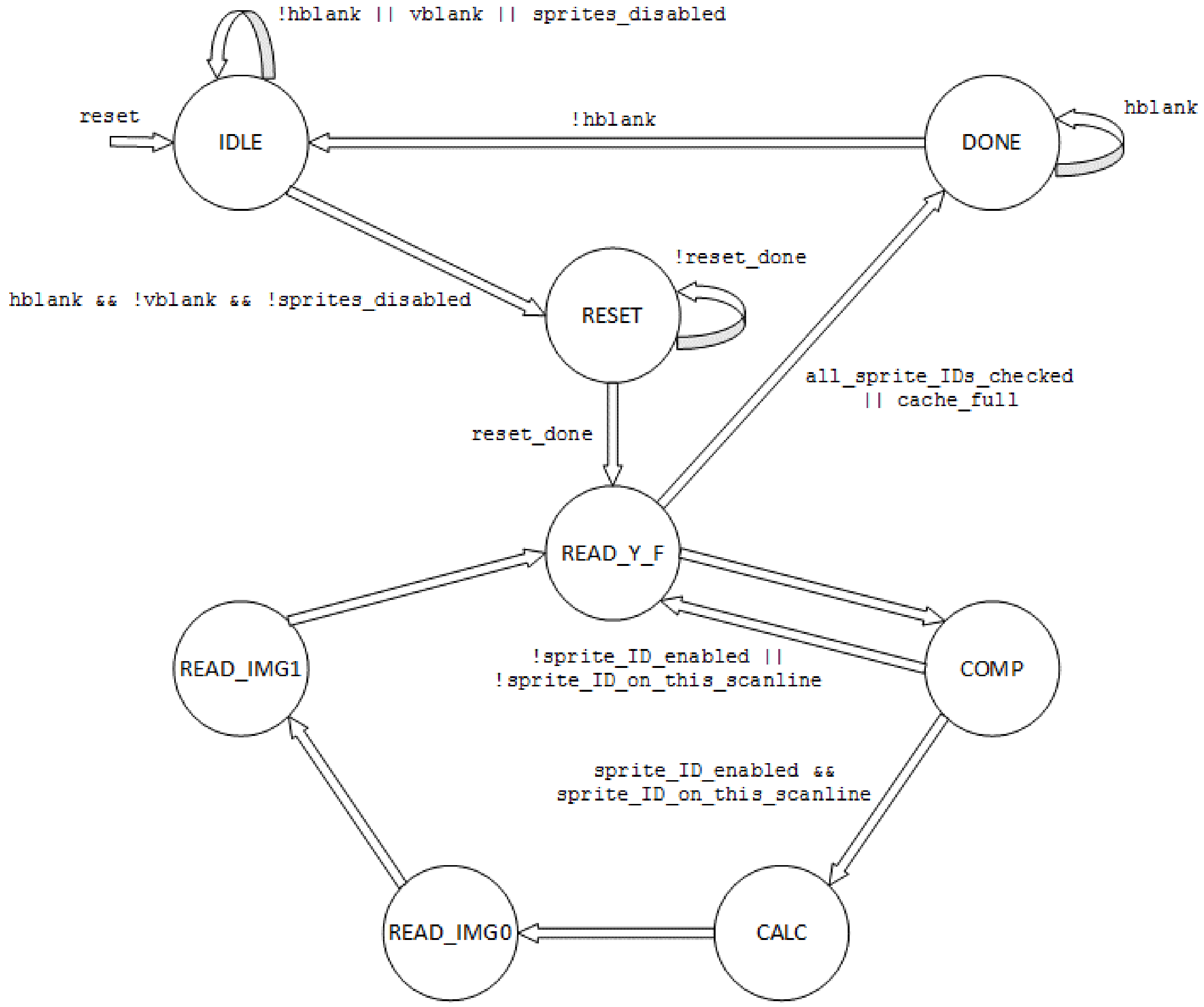
It is partially due to this complexity that other systems limit the number of sprites visible upon a scanline to the highest 8 or 10 priority sprites. For the sake of argument, we elect to display the highest 10 priority (lowest ID) sprites.

With this in mind, let us briefly think about our sprite size, which must be carefully considered given the amount of time for updating sprite parameters during the vertical blanking region and the amount of time and number of flops needed to cache data from visible sprites during horizontal blanking. A wide sprite size can potentially reduce the need to use multiple sprites to represent one on-screen game character, however, more row data must be read for each sprite. Tall sprites, depending on placement and assuming that some rows consist of all transparent pixels, have a higher likelihood of unnecessarily conflicting with other objects that could potentially be visible on each scanline. A narrow sprite size might require 2 or 4 adjacent sprites to represent a single character, also reducing the number of complete objects to appear on a single scanline. A balance must be achieved between height and width, and given prior art in this area, a configurable sprite height of 8 or 16 (selected by the size sprite parameter) seems appropriate. Regarding width of the sprite, RAMs in the Cyclone IV can be configured to be 16 bits wide, so taking 2 consecutive words results in a 32-bit row size, producing 32 pixels at 1bpp, 16 at 2bpp, and 8 at 4bpp. Because the RAM remaining in revision 0.87 of CoCoVGA's Cyclone IV is ~100kbits, and there are other future enhancements being considered, a target of less than ~70kbits is necessary. Selecting a height of rows of pixels 16 enables more than 130 sprite images in this amount of RAM (each one 32x16 = 512 bits). 96 sprite images would result in 48kbits (6kbytes), which also happens to be a convenient size for uploading via a RG6 or CG6 mode page.

Returning to the concern of how to perform sprite image row caching, the time spent in the VGA horizontal blanking region is the total horizontal scan cycles minus the visible "green field" region (since the hardware is blanked during the solid color border of the Color Computer emulated display). The entire width of a 640x480 VGA display, including pixels that are not drawn, is 800 pixels. The visible field of Color Computer video is 256 6847 pixels, which are each doubled in width to 512 VGA pixels. Therefore, the time available during each horizontal scan is 800 – (256*2) = 288 25MHz VGA pixel clock cycles.

If sprite parameter sets are placed in 16-bit wide RAM, it is appropriate to collect the critical data together such that a single read provides information about Y position and the flags. Of the flags, the sprite enable can be used to skip a sprite image read, and the size (height) information combined with the sprite's Y position is necessary to determine whether this sprite is within this scanline.

The state machine to perform caching functionality appears as follows:


!hblank || vblank || sprites_disabled
reset
IDLE
!hblank
DONE
hblank
!reset_done
hblank && !vblank && !sprites_disabled
RESET
all_sprite_IDs_checked || cache_full
reset_done
READ_Y_F
READ_IMG1
!sprite_ID_enabled || !sprite_ID_on_this_scanline
COMP
sprite_ID_enabled && sprite_ID_on_this_scanline
READ_IMG0
CALC

The state machine remains in the IDLE state until the horizontal blanking region is reached and the main sprite enable flag is enabled by way of a page 0 enhanced mode control. In this stage, the sprite_ID_counter will be set to 0.

In the RESET state, the state machine iterates through the sprite image cache, clearing each set of flops to an invalid/unwritten state.

Because the sprite parameters are to be stored in a RAM, the read cycle to the critical Y position and flags information are only initiated in the READ_Y_F state for the sprite parameters specified by the sprite_ID_counter. If the sprite_ID_counter has reached the end of all of the sprite descriptors or if all 10 entries of the cache are now full, then the caching operation is complete until the next horizontal blanking period, so the state machine can complete by way of DONE and IDLE.

The data from READ_Y_F is returned in the COMP state, where this sprite ID is checked for enable and whether it is within this scanline. In other words, taking the sprite's Y position and size into account, the COMParison:

sprite_Y[sprite_ID_counter] <= scanline_Y < (sprite_Y[sprite_ID_counter] + height)

provides indication to either continue working with this sprite (enter the CALC state) or to increment the sprite_ID_counter and return to READ_Y_F to get the next sprite ID's parameters. Additionally, the read for the X position and image pointer for the current sprite ID are started in this cycle with the assumption that the data will be needed in the CALC state. If the CALC state is not reached, of course, the data returned from the sprite parameter RAM will be ignored.

In the CALC state, the sprite image pointer index and X position is returned from the prior read. By way of an address CALCulation with Y and flags, the sprite image pointer is converted to a sprite image RAM address. This address is immediately used to start the read for the first 16 bits of row image data. Also in this stage, the X position and flags are stored in the cache for this sprite, since they will be needed when displaying the sprite. Note that the Y position is no longer necessary, because only the image row that pertains to this scanline will be copied from the sprite image RAM into the cache.

The READ_IMG0 and READ_IMG1 states accept return data from each of their respective prior stages' read operations and store the 16-bit words of the sprite image RAM row into flops for use when drawing the image later. The READ_IMG1 state also takes the action to increment the sprite_ID_counter for the next READ_Y_F.

Starting with the smallest sprite ID and incrementing through all sprites' parameter sets in order gives the sprites a fixed priority, with ID 0 being most important to appear and the highest ID sprite being the least.

It will take 1 cycle to traverse from the INIT state to RESET, and RESET will consume 10 cycles (one for each cache entry) before reaching the first READ_Y_F, leaving 288-1-10 = 277 pixel clock cycles. This means, given the typical traversal of 5 cycles per sprite to cache, that in addition to whatever top 10 priority sprites are found, 275 – (10*5) = 245 cycles remain. Because it takes 2 cycles to determine whether a sprite is enabled and within the current scanline, there is potential to check 245/2 = 122 more. Here, the limiting factor is more likely to be flop storage for the sprite parameter sets or the pipeline's pixel priority mux size than the hblank time to perform these calculations and caching reads.

Each of the 10 priority-ordered cache entries contains the following information:

- 1 valid bit
- 8 bits of X position
- 2 bits of flags for X mirroring and palette selection
- 32 bits of sprite image row pixel data

Each cache entry therefore consumes 43 flops for a total of 43*10 = 430 register bits.

The state machine must track the current cache entry with a 4 bit write pointer, have a 6 bit sprite ID counter, and a 3 bit machine state. Temporary data, specifically Y mirroring, Y position, and size will consume a total of 10 bits of flops. Certainly 430 + 6 + 3 + 10 = 449 flops is not unreasonable, especially when compared to the ~700 register bits consumed only by palette registers.

Revisiting the pipeline once again, the multi-sprite-capable display pipeline is similar to the version appropriate for a single sprite, the main exceptions being that Y position and image lookup no longer needs to be taken into account:

- READ
  1. calculate X coordinate that is about to be displayed
  2. compare X against all sprite descriptor register values in the cache
  3. for any matches found, select appropriate bits from cached image registers
- DECODE
  4. select between original (background) pixel color and sprites (taking transparency into account)
  5. calculate pixel color
- MUX
  6. no change
- DISPLAY
  7. no change

Note that the cached sprite selection will have to be evaluated during each READ in parallel to discern which sprites are in range of the current X display location, figure out which are transparent, and have these results ready for the next cycle when the pixel to decode can be selected from either one of these sprites or the background color.

Conclusions and Future Work

We have seen that without hardware acceleration of sprites, it can easily cost more than 400 cycles to copy an 8x8 region of 4bpp pixels from one memory to another in order to draw each sprite, even using stack blasting. By comparison, a hardware accelerator could be used to manipulate up to any 17 sprites during a vertical blanking region combination lock entry of 40 cycles, followed by 100 cycles per sprite, independent of the size (8x8 or 8x16). Such a hardware improvement could therefore more than quadruple performance, either by way of sprite size or by allowing the CPU to perform other tasks such as game logic or production of audio.

The hardware accelerator could perform any number of the following operations to a selected set of sprites on a per-frame basis:

- enable or disable it, making it “flash” on and off or fade in or out, depending on the timing
- change the direction it’s pointing
- select a different sprite image to provide the appearance of animation or
- change its palette, potentially causing fade, flash, or animation effects
- move it to a new location

One aspect not yet examined is the expense of formatting sprite parameter data into 4 data bits and a toggling strobe bit. The assumption is that it would be reasonable to calculate the new locations, flags, and sprite image indices and convert this info to the appropriate format while the video frame is being drawn so that in the next blanking region, the array would be fully populated and ready to bit-bang. Ideally, this should be experimented with further to prove that these operations are not overly onerous.

Uploading sprites to CoCoVGA is intended to be a 1-time or otherwise very infrequent operation, usually performed at the start of program execution, and therefore not considered in this efficiency estimate. Note that by uploading the sprite images to CoCoVGA's sprite image RAM, they no longer need to be housed in the Color Computer's general purpose RAM, making this region available to be used in other ways.

Related to the feasibility of implementation in the unused hardware remaining in the current FPGA, the most challenging estimate to perform is related to the number of flops truly available. This is not a number directly provided from the hardware documentation or synthesis tool, since the accessibility to registers is based on how the LEs' LUTs (Look-Up Tables) and registers can be employed as the Verilog design is mapped to them. This aspect will require experimentation and review of the synthesis reports.

Regarding expected RAM utilization, these parameters are clearly tracked and reported by the Quartus Prime synthesis tool, so we can check that our total expected additional RAM usage fits:

Usage	Size
Expected usage for other extended features not described here	~30kbits
Sprite image RAM – 96 images	~48kbits
Sprite parameter RAM – 64 sprites	~2kbits
Total expected additional RAM usage	~80kbits
Current remaining RAM in FPGA	~100kbits

A quality of some systems' hardware sprite acceleration engines is that of providing collision detection between objects. In other words, a flag or interrupt could optionally be triggered if the graphics hardware discovered that sprites were touching or overlapping. This is an extremely useful feature to have for programming games, however, due to the nature of the Color Computer's architecture, the 6847 is an output-only device (with the exception of the address, vsync, and hsync signals, which are either not connected or used for other purposes) so there is no way to read back collision information if such a feature were added to the CoCoVGA FPGA design.

Another feature seen in other systems is the ability to multiplex or reuse sprites after they have been drawn but for objects which appear lower in the display. This simply will not be possible with CoCoVGA, due to the fact that it would be necessary to comprehend where in the display the VGA scan happens to be. This is not possible because the vsync and hsync signaled from the 6847 socket happen to be those of the 6847, which is drawing into the input side of the double-buffer at the NTSC or PAL refresh rate, not reading from the output side at the VGA refresh rate.

All-in-all, if software would be written to use the features of hardware accelerated sprites provided by CoCoVGA, then this would definitely be a worthwhile addition to make to the FPGA design. The above evaluation is optimistic enough that attempted implementation is warranted.

Sources

https://en.wikipedia.org/wiki/Game_Boy

http://devrs.com/gb/files/gbspec.txt

https://www.c64-wiki.com/wiki/Sprite

http://ece545.com/S15/reports/F06_NES.pdf

http://www.cocovga.com/documentation/software-mode-control/

http://www.colorcomputerarchive.com/coco/Documents/Datasheets/MC6847%20MOS%20Video%20Display%20Generator%20(Motorola).pdf

http://www.lomont.org/Software/Misc/CoCo/Lomont_CoCoHardware_2.pdf

https://en.wikipedia.org/wiki/TRS-80_Color_Computer

http://www.lemon64.com/forum/viewtopic.php?t=14988

https://en.wikipedia.org/wiki/Sprite_%28computer_graphics%29#Hardware_sprites

Motorola Inc. *MC6809 - MC6809E Microprocessor Programming Manual*. Motorola Inc., 1981.

https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf