



# Video

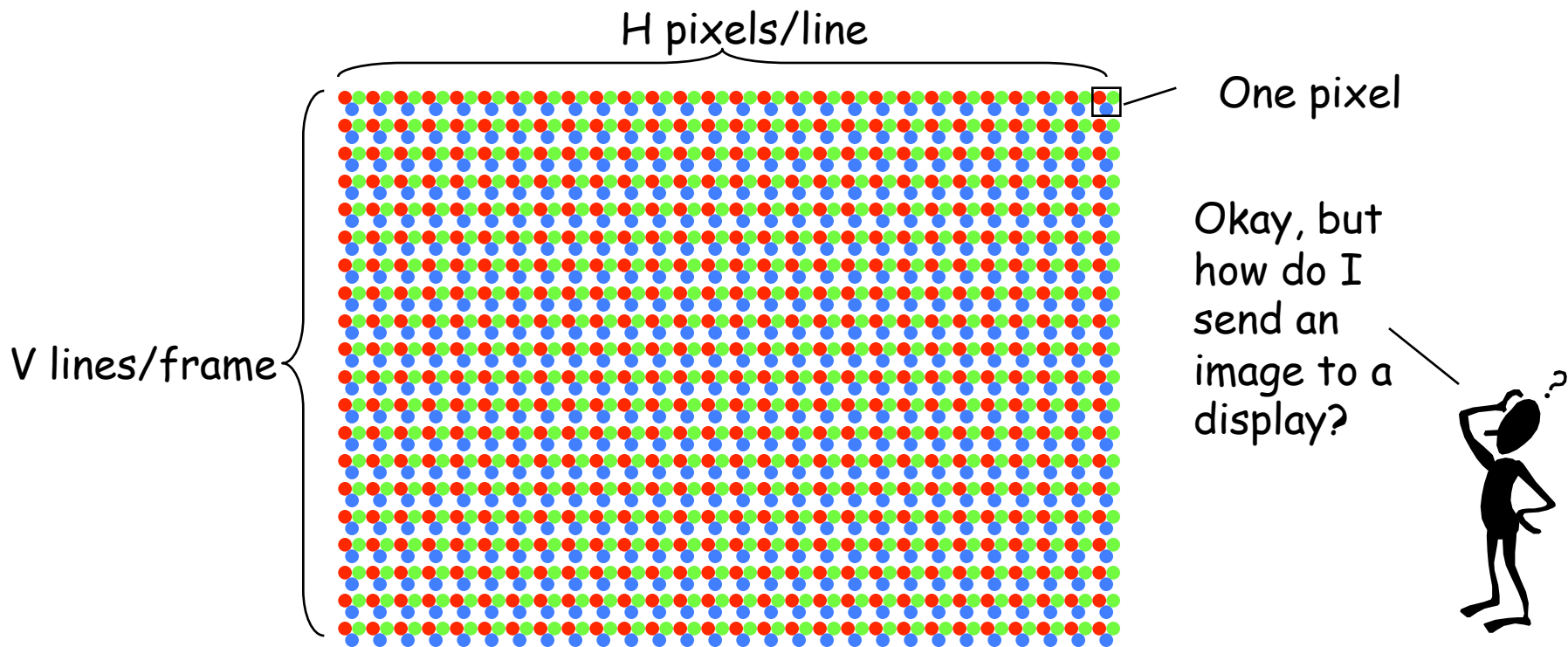
- Generating video sync signals
- Decoding NTSC video
  - color space conversions
- Generating pixels
  - test patterns
  - character display
  - sprite-based games

Updated  
fir31.filtered  
on website

Lab #4 due Thursday, project teams next Monday

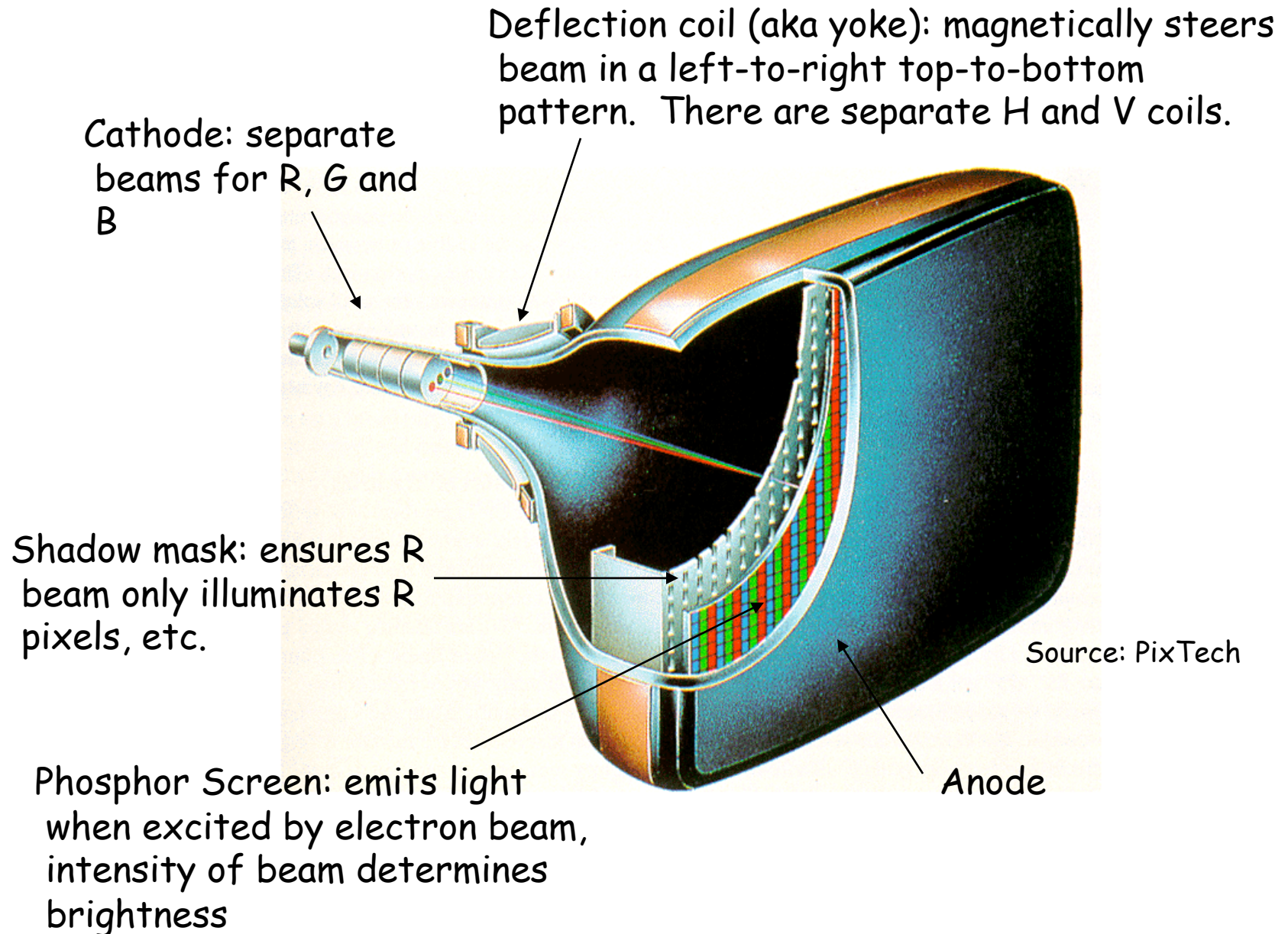
# The CRT: Generalized Video Display

Think of a color video display as a 2D grid of picture elements (pixels). Each pixel is made up of red, green and blue (RGB) emitters. The relative intensities of RGB determine the apparent color of a particular pixel.

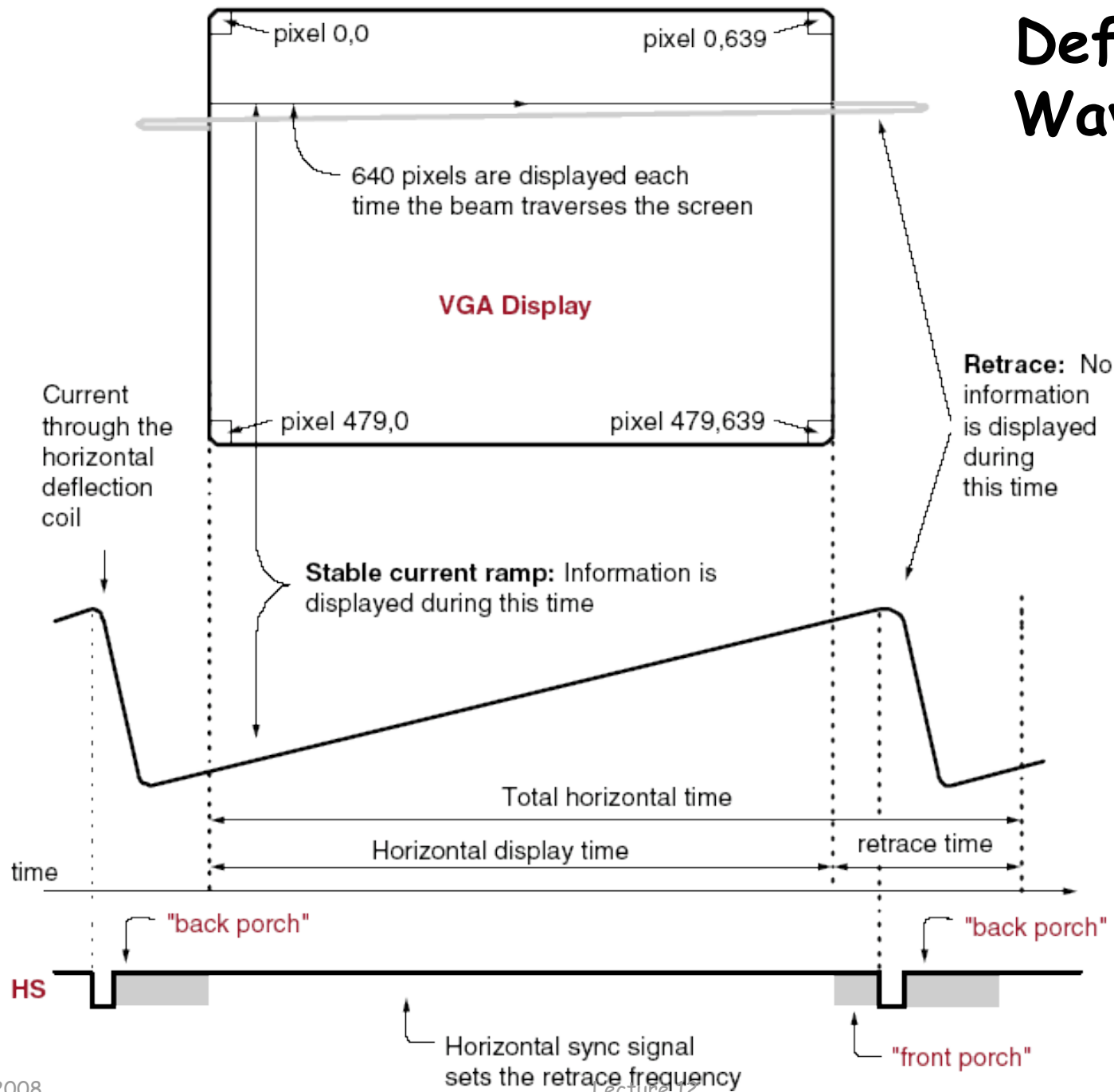


Traditionally  $H/V = 4/3$  or with the advent of high-def  $16/9$ .  
Lots of choices for H,V and display technologies (CRT, LCD, ...)

# Background: Cathode Ray Tubes

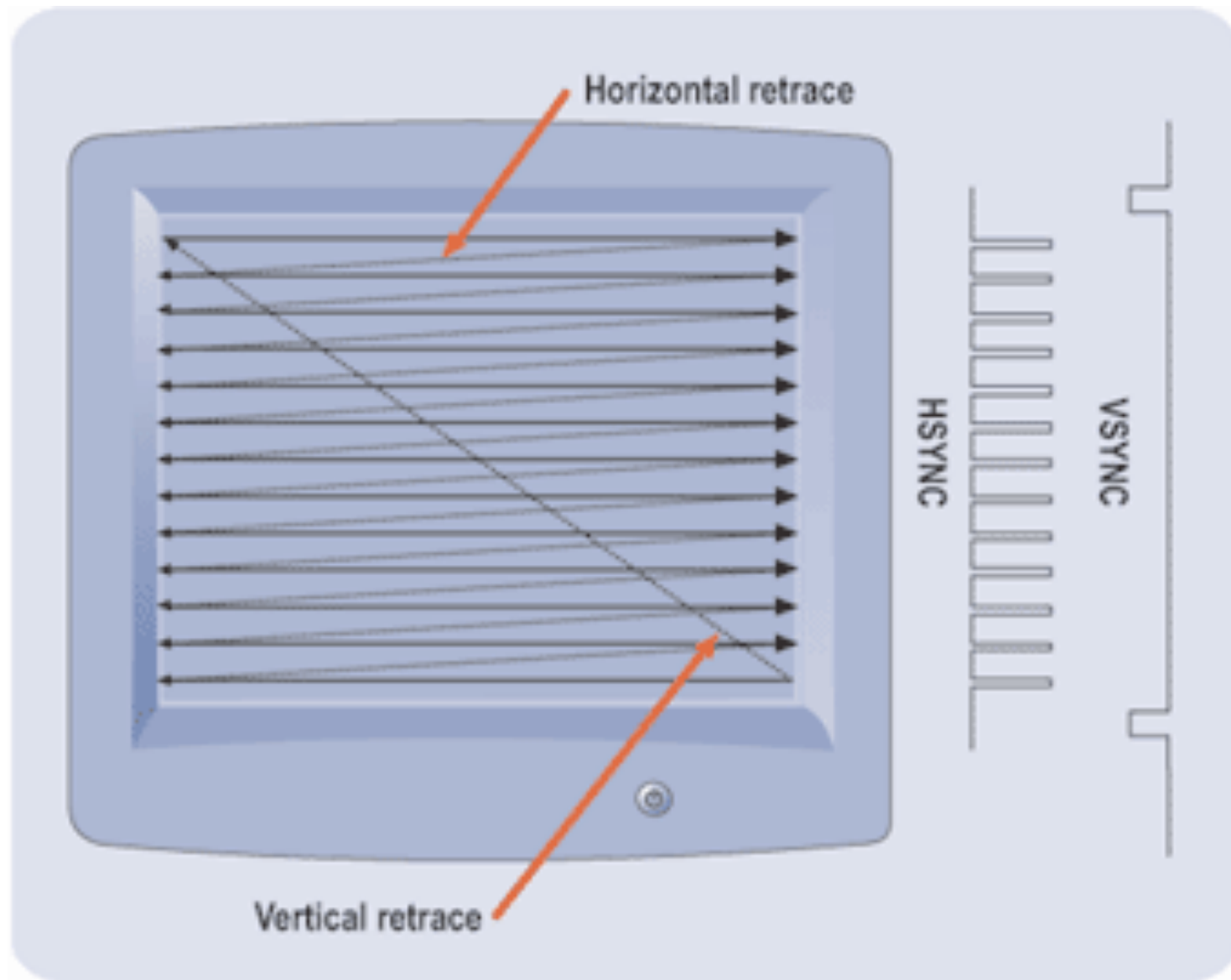


# Deflection Waveforms



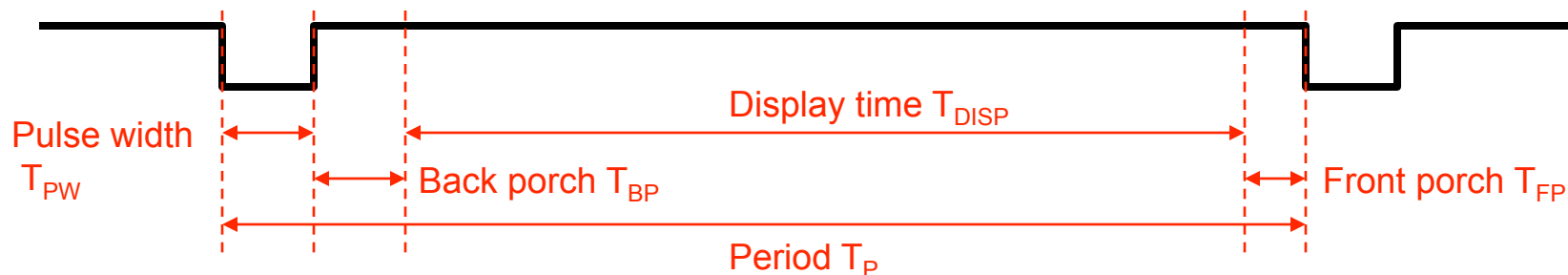
Source: Xilinx Spartan-3 Starter Kit Board User Guide

# Sync Signals (HS and VS)



# Sync Signal Timing

The most common ways to send an image to a video display (even displays that don't use deflection coils, eg, LCDs) require you to generate two sync signals: one for the horizontal dimension (HS) and one for the vertical dimension (VS).



<i>Format</i>		<i>CLK</i>	<i>P</i>	<i>PW</i>	<i>BP</i>	<i>DISP</i>	<i>FP</i>
VGA	HS (pixels)	25Mhz	794	95	47	640	13
	VS (lines)	--	528	2	33	480	13
XGA	HS (pixels)	65Mhz	1344	136	160	1024	24
	VS (lines)	--	806	6	23	768	9

# Interlace

Non-interlaced (aka progressive) scanning:

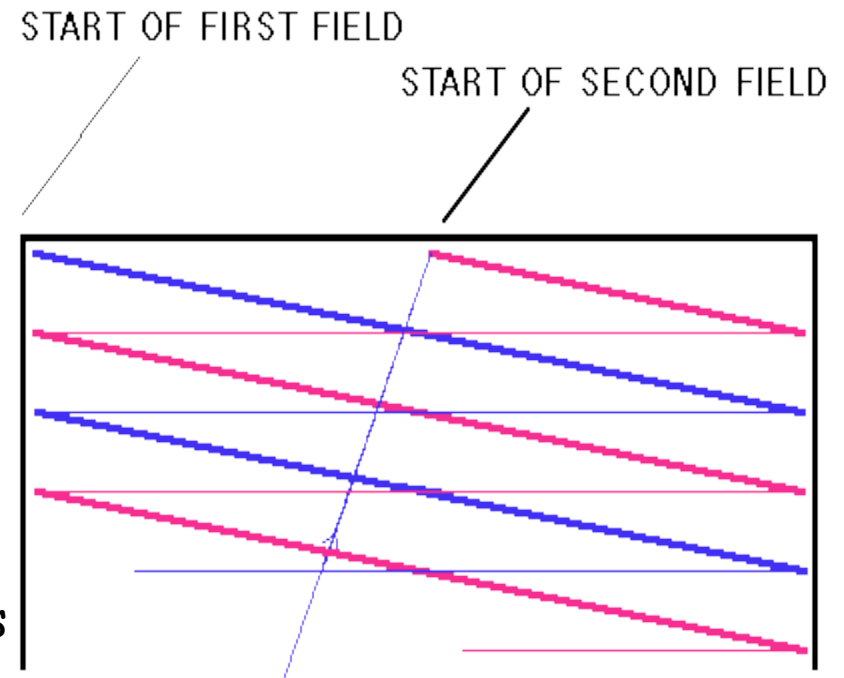
- VS period is a multiple of HS period
- Frame rate  $\geq 60\text{Hz}$  to avoid flicker

Interlaced scanning:

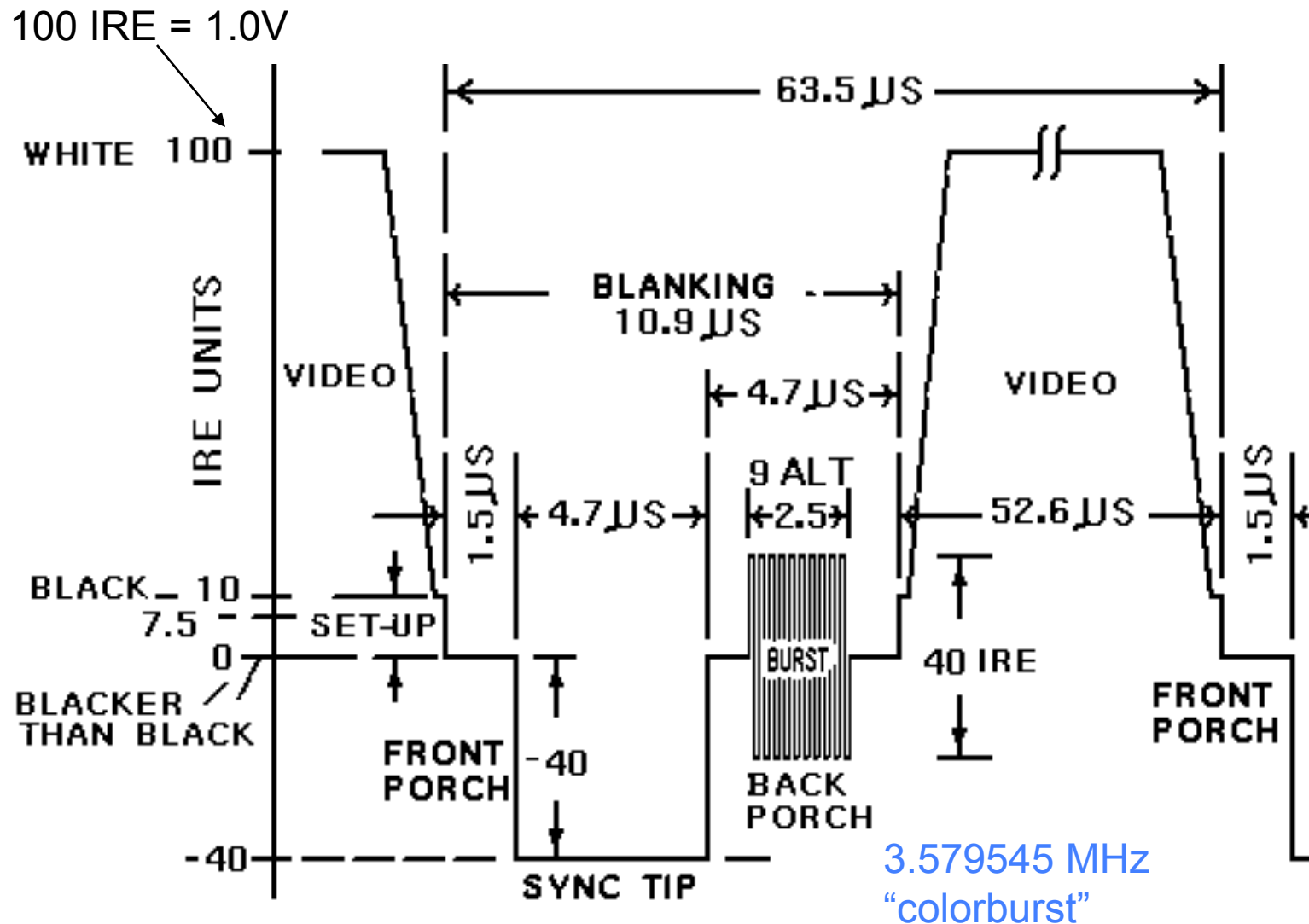
VS period is *not* a multiple of HS period, so successive vertical scans are offset relative to horizontal scan, so vertical position of scan lines varies from frame to frame.

NTSC example:

- 525 total scan lines (480 displayed)
- 2 fields of 262.5 scan lines (240 displayed). Field rate is 60Hz, frame rate = 30Hz



# NTSC\*: Composite Video Encoding



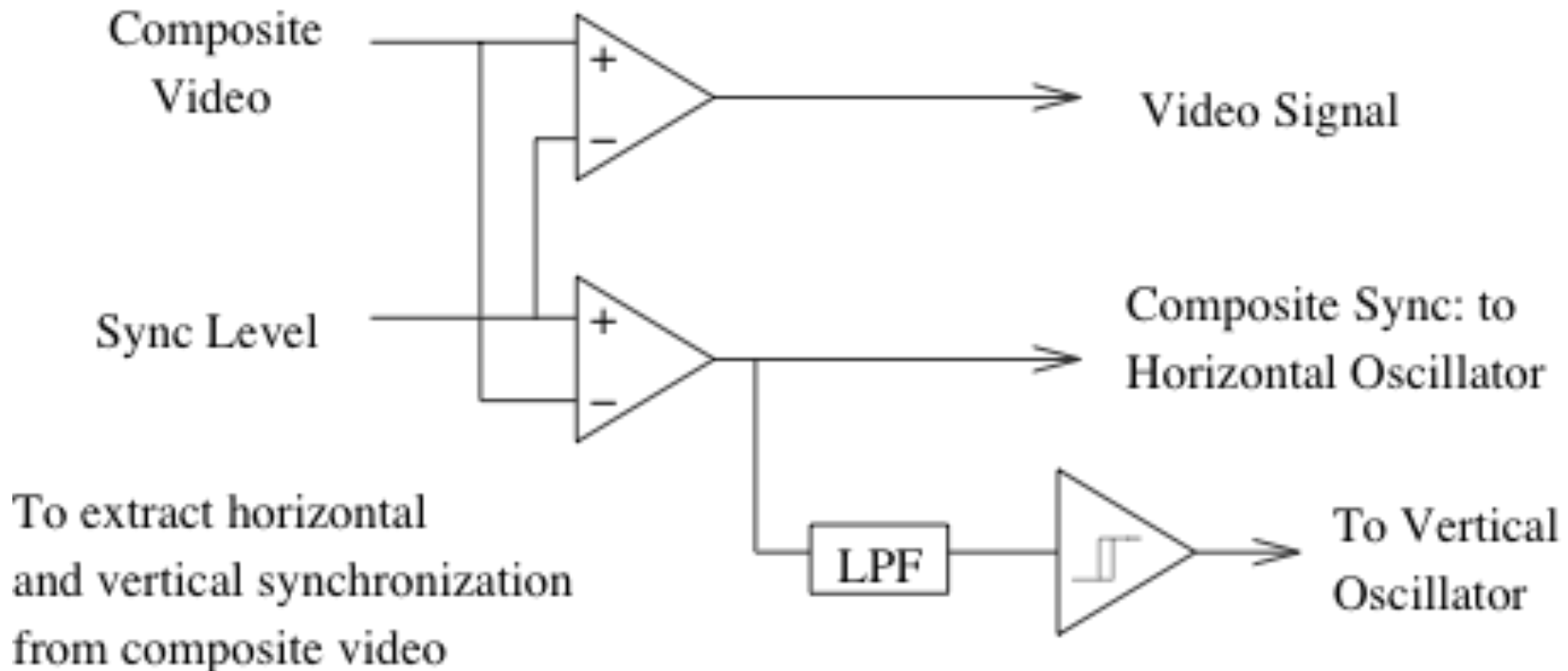
\*National Television System Committee: 1940

Source: <http://www.ntsc-tv.com>



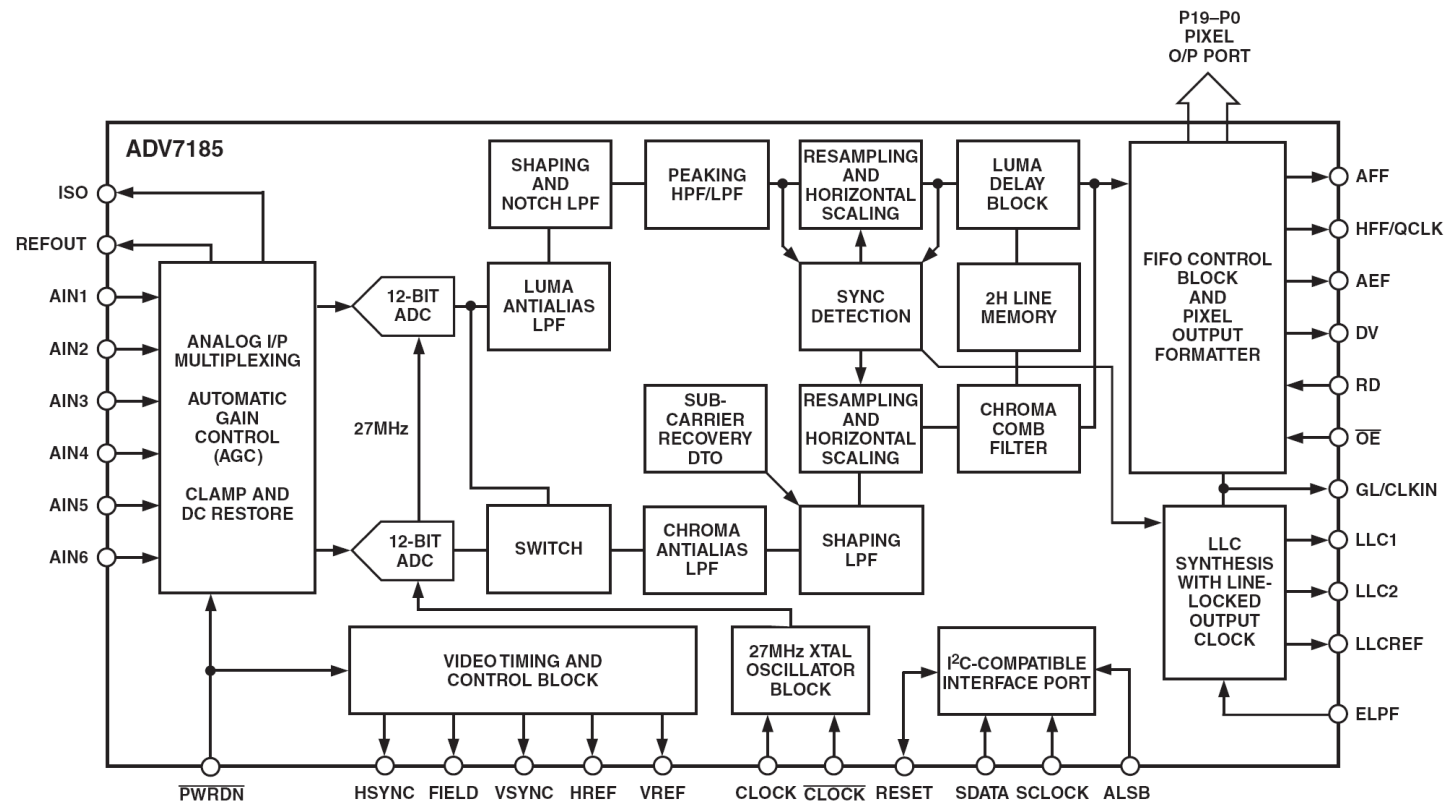
# Video Capture: Signal Recovery

- Composite video has picture data and both syncs.
  - Picture data (video) is above the sync level.
  - Simple comparators extract video and composite sync.
- Composite sync is fed directly to the horizontal oscillator.
- A low-pass filter is used to separate the vertical sync.
  - The edges of the low-passed vertical sync are squared up by a Schmidt trigger.



# Labkit: ADV7185 NTSC Decoder

- Decodes NTSC and PAL video (composite or S-video)
- Produces CCIR656 (10-bit) or CCIR601 (8-bit) digital data



# Labkit: ADV7185 NTSC Decoder

- Decodes NTSC and PAL video (composite or S-video)
- Produces CCIR656 (10-bit) or CCIR601 (8-bit) digital data

BLANKING PERIOD			TIMING REFERENCE CODE				720 PIXELS YUV 4 : 2 : 2 DATA										TIMING REFERENCE CODE				BLANKING PERIOD		
...	80	10	FF	00	00	SAV	C <sub>B</sub> 0	Y0	C <sub>R</sub> 0	Y1	C <sub>B</sub> 2	Y2	...	C <sub>R</sub> 718	Y719	FF	00	00	EAV	80	10	...	

Pixel 1: Y1, C<sub>B</sub>0, C<sub>R</sub>0

Pixel 0: Y0, C<sub>B</sub>0, C<sub>R</sub>0

8-bit SAV/EAV code: 1FVHabcd  
 10-bit SAV/EAV code: 1FVHabcd00  
 F = field (0: field 1/odd, 1: field 2/even)  
 V = vsync (0 for SAV)  
 H = hsync (0 for SAV)  
 a = V<sup>^</sup>H  
 b = F<sup>^</sup>H  
 c = F<sup>^</sup>V  
 d = F<sup>^</sup>V<sup>^</sup>H  
 8h'80, 10'h200 = start of even field  
 8h'C7, 10'h31C = start of odd field

8-bit data:

Y in range 16-235;  
 C<sub>R</sub>, C<sub>B</sub> in range 16-240  
 (offset by 128)



10-bit data:

Y in range 64-943;  
 C<sub>R</sub>, C<sub>B</sub> in range 64-963  
 (offset by 512)

# YCrCb to RGB (for display)

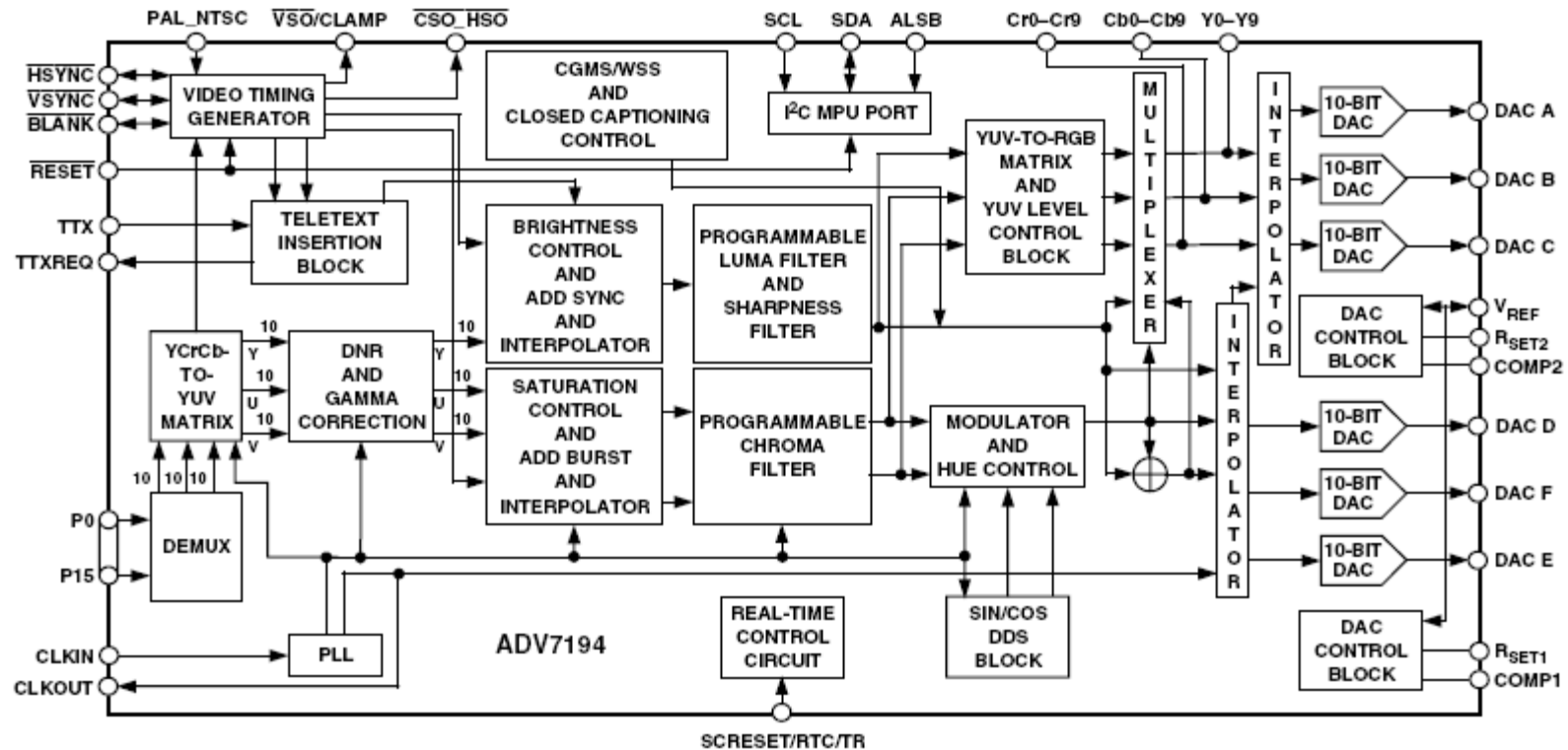
- 8-bit data
  - $R = 1.164(Y - 16) + 1.596(Cr - 128)$
  - $G = 1.164(Y - 16) - 0.813(Cr - 128) - 0.392(Cb - 128)$
  - $B = 1.164(Y - 16) + 2.017(Cb - 128)$
- 10-bit data
  - $R = 1.164(Y - 64) + 1.596(Cr - 512)$
  - $G = 1.164(Y - 64) - 0.813(Cr - 512) - 0.392(Cb - 512)$
  - $B = 1.164(Y - 64) + 2.017(Cb - 512)$
- Implement using
  - Integer arithmetic operators (scale constants/answer by  $2^{11}$ )
  - 5 BRAMs (1024x16) as lookup tables for multiplications

# Video Feature Extraction

- A common technique for finding features in a real-time video stream is to locate the center-of-mass for pixels of a given color
  - Using RGB can be a pain since a color (eg, red) will be represented by a wide range of RGB values depending on the type and intensity of light used to illuminate the scene. Tedious and finicky calibration process required.
- Consider using a HSL/HSV color space
  - H = hue (see diagram)
  - S = saturation, the degree by which color differs from neutral gray (0% to 100%)  

  - L = lightness, illumination of the color (0% to 100%)  

- Filter pixels by hue!

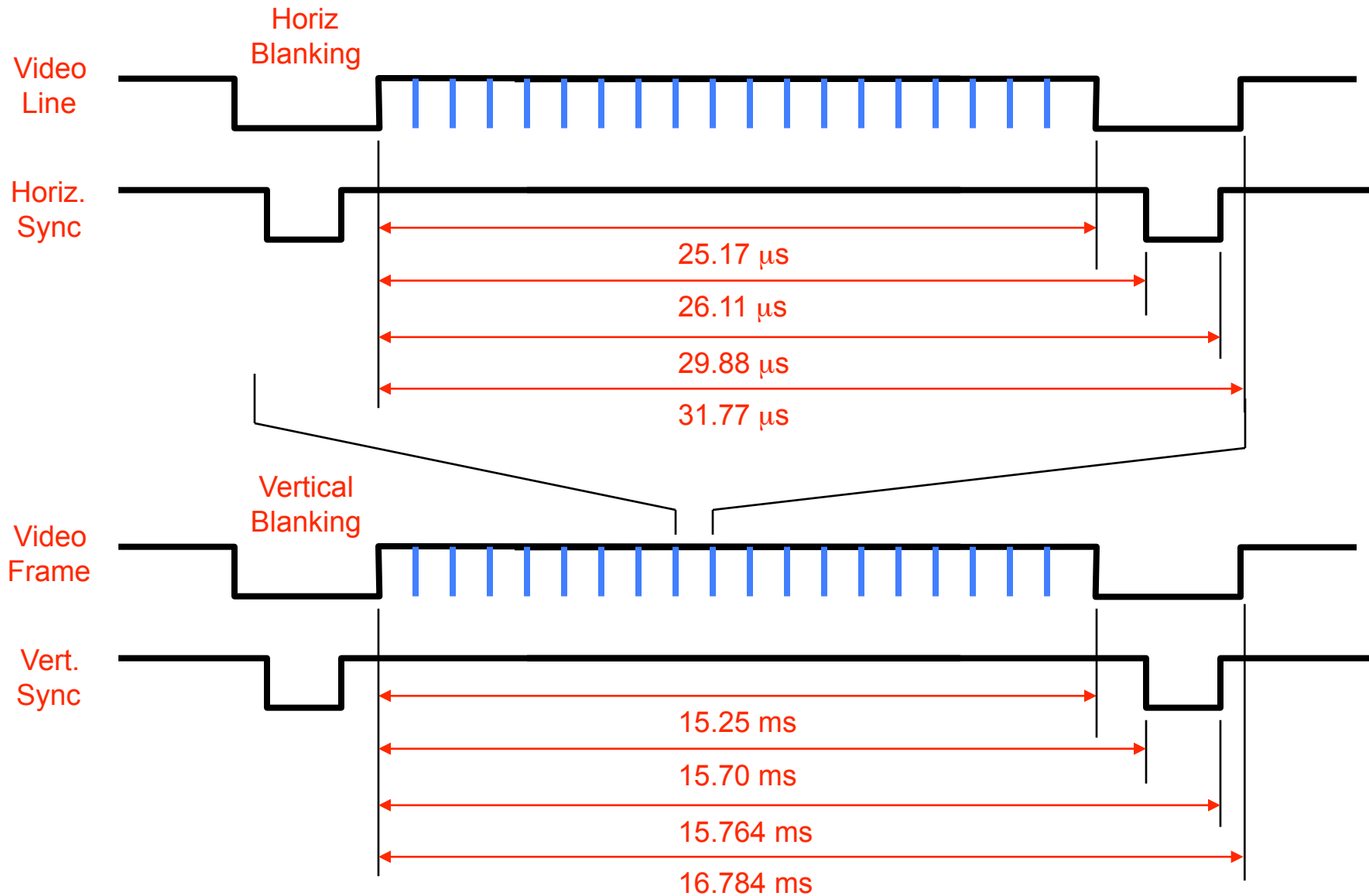


# Labkit: AD7194 Digital Video Encoder



CCIR 601/656 4:2:2 digital video data → analog baseband TV signal

# VGA (640x480) Video

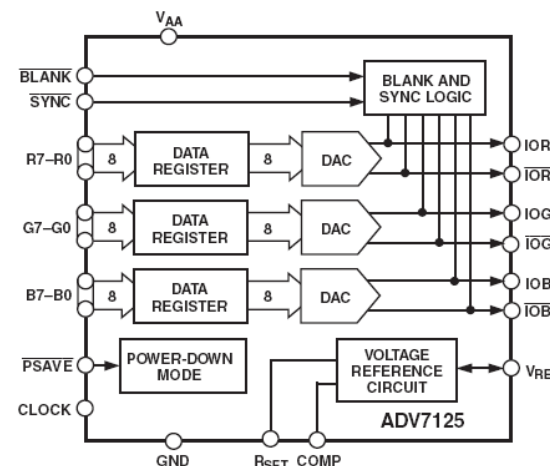


# Labkit: ADV7125 Triple DAC (VGA)

- Two Challenges:
  - (1) Generate Sync Signals
    - Sync signal generation requires precise timing
    - Labkit comes with 27 MHz clock
    - Use phase-locked-loops (PLL) to create higher frequencies
    - Xilinx FPGA's have a "Digital Clock Manager" (DCM)

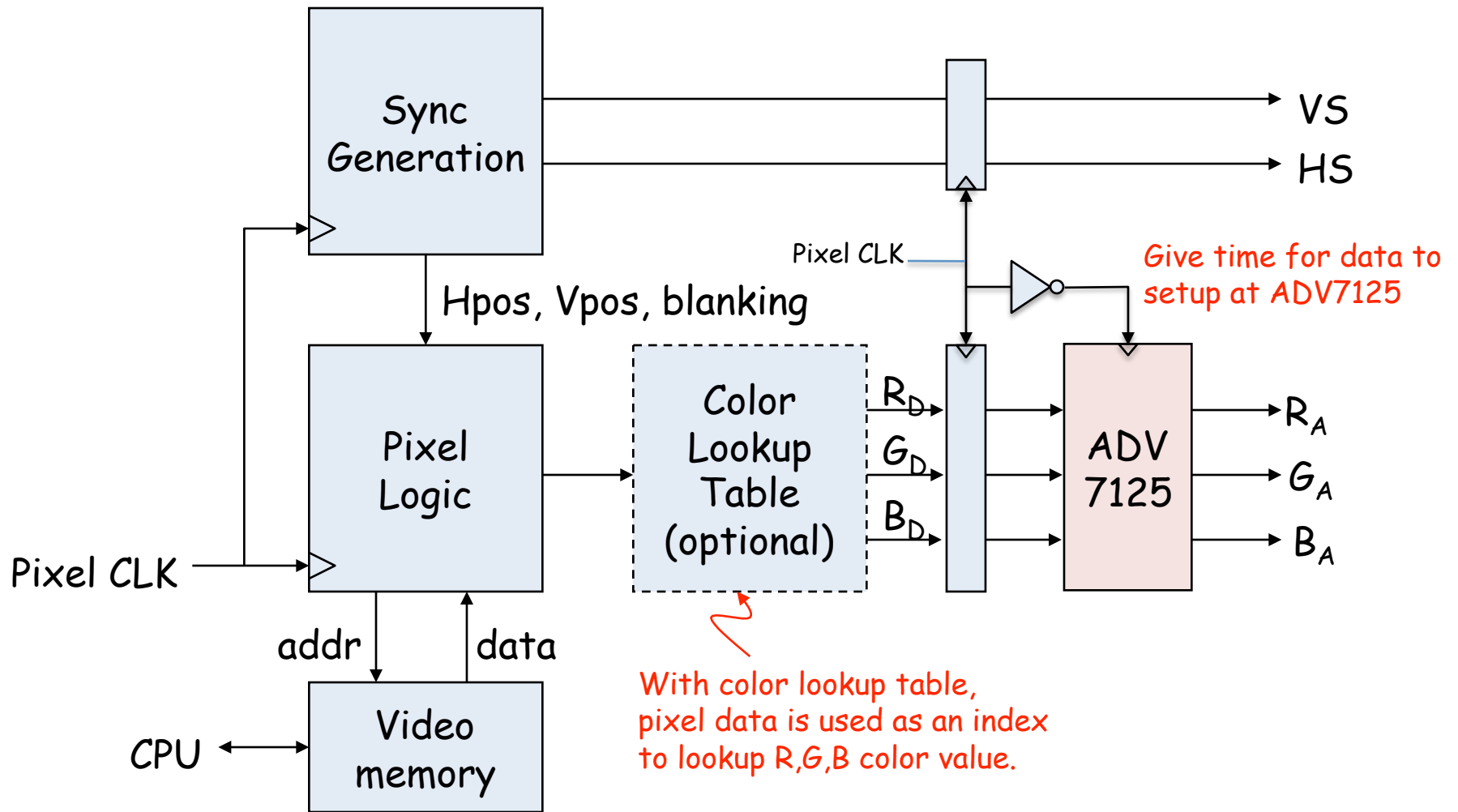
```
DCM pixel_clock(.CLKIN(clock_27mhz), .CLKFX(pixel_clock));  
// synthesis attribute CLKFX_DIVIDE of pixel_clock is 10  
// synthesis attribute CLKFX_MULTIPLY of pixel_clock is 24  
// 27MHz * (24/10) = 64.8MHz
```

- (2) Generate Video Pixel Data (RGB)
  - Use ADV7125 Triple DAC
  - Send 24 bits of R,G,B data at pixel clock rate to chip
  - Create pixels either in real time
  - Or using dual port RAM
  - Or from character maps
  - Or ...?





# Generating VGA-style Video

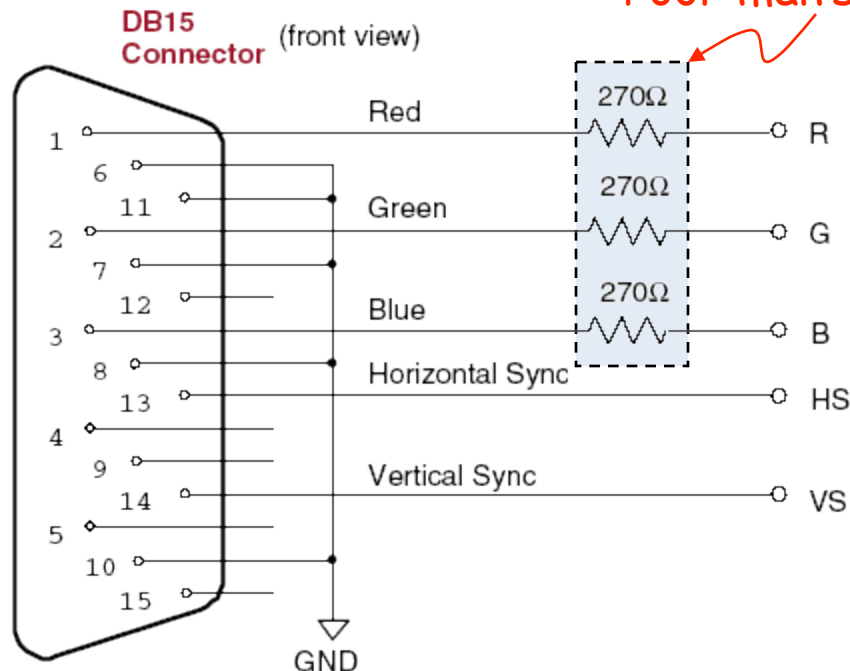


With color lookup table,  
pixel data is used as an index  
to lookup R,G,B color value.

Without color lookup table,  
pixel data is used directly as  
R,G,B value (aka "true color")

# Simple VGA Interface for FPGA

## Poor man's Video DAC



Your circuitry should produce TTL-level signals (3.3V high level)

HS, VS are active-low signals.

R, G, B are active-high.

Shown: a simple "8-color" scheme

The R, G and B signals are terminated with 75 Ohms to ground inside of the VGA monitor. So when you drive your 3.3V signal through the 270 Ohm series resistor, it shows up at the monitor as 0.7V - exactly what the VGA spec calls for.

$$0.7V = \left(\frac{75}{75 + 270}\right)(3.3V)$$

## Verilog: XVGA Display (1024x768)

```
module xvga(clk,hcount,vcount,hsync,vsync);  
    input clk;                // 64.8 Mhz  
    output [10:0] hcount;  
    output [9:0] vcount;  
    output hsync, vsync;  
    output [2:0] rgb;  
  
    reg hsync,vsync,hblank,vblank,blank;  
    reg [10:0] hcount;        // pixel number on current line  
    reg [9:0] vcount;         // line number  
  
    wire hsynccon,hsyncoff,hreset,hblankon; // next slide for generation  
    wire vsyncon,vsyncoff,vreset,vblankon; // of timing signals  
  
    wire next_hb = hreset ? 0 : hblankon ? 1 : hblank; // sync & blank  
    wire next_vb = vreset ? 0 : vblankon ? 1 : vblank;  
  
    always @(posedge clk) begin  
        hcount <= hreset ? 0 : hcount + 1;  
        hblank <= next_hb;  
        hsync <= hsynccon ? 0 : hsyncoff ? 1 : hsync; // active low  
  
        vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;  
        vblank <= next_vb;  
        vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync; // active low  
    end
```

# XVGA (1024x768) Sync Timing

```
// assume 65 Mhz pixel clock

// horizontal: 1344 pixels total
// display 1024 pixels per line
assign hblankon = (hcount == 1023); // turn on blanking
assign hsyncon = (hcount == 1047); // turn on sync pulse
assign hsyncoff = (hcount == 1183); // turn off sync pulse
assign hreset = (hcount == 1343); // end of line (reset counter)

// vertical: 806 lines total
// display 768 lines
assign vblankon = hreset & (vcount == 767); // turn on blanking
assign vsyncon = hreset & (vcount == 776); // turn on sync pulse
assign vsyncoff = hreset & (vcount == 782); // turn off sync pulse
assign vreset = hreset & (vcount == 805); // end of frame
```

# Video Test Patterns

- Big white rectangle (good for “auto adjust” on monitor)

```
always @(posedge clk) begin
    if (vblank | (hblank & ~hreset)) rgb <= 0;
    else
        rgb <= 7;
end
```

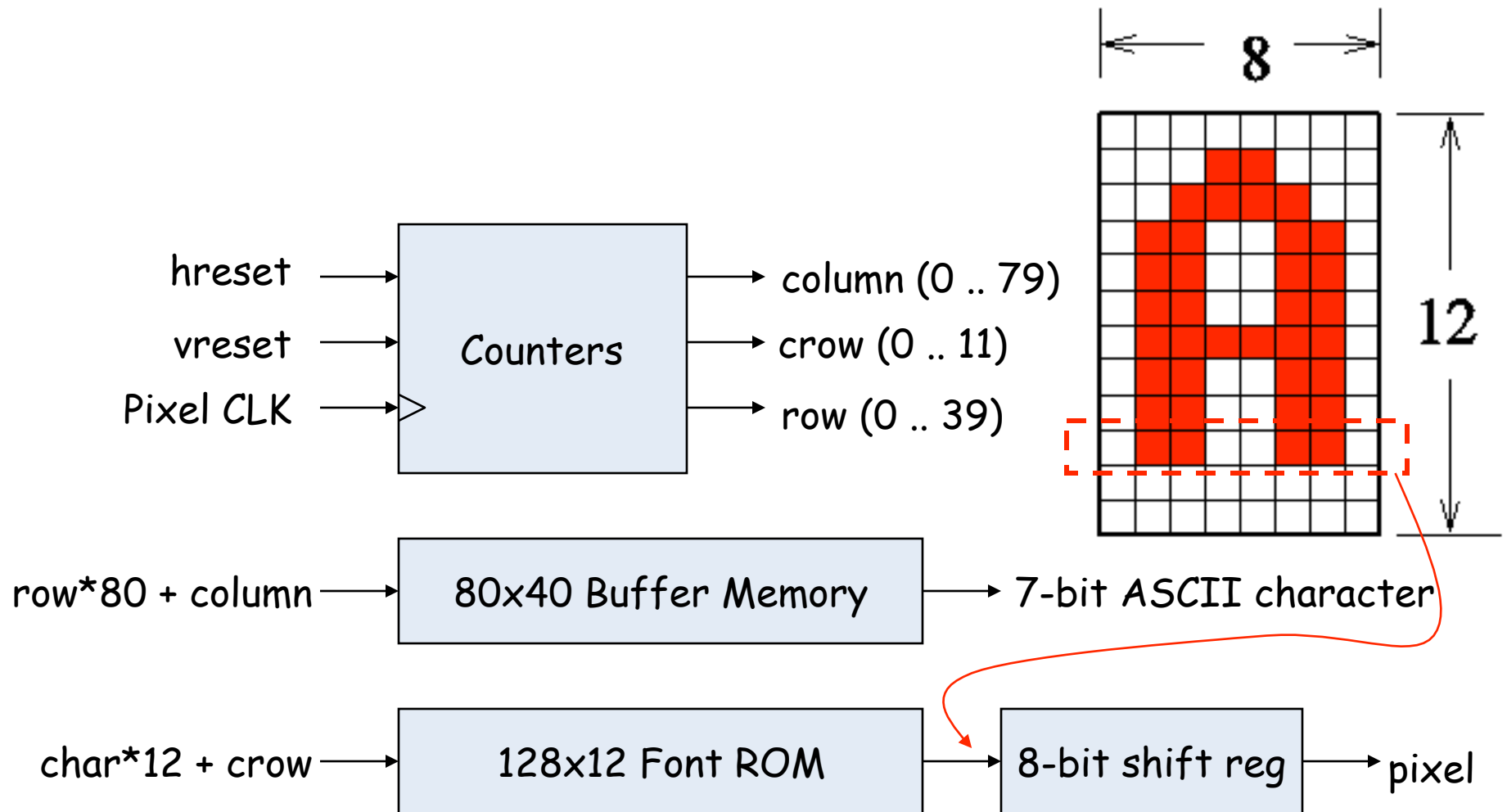
- Color bars

```
always @(posedge clk) begin
    if (vblank | (hblank & ~hreset)) rgb <= 0;
    else
        rgb <= hcount[8:6];
end
```

<i>RGB</i>	<i>Color</i>
000	black
001	blue
010	green
011	cyan
100	red
101	magenta
110	yellow
111	white

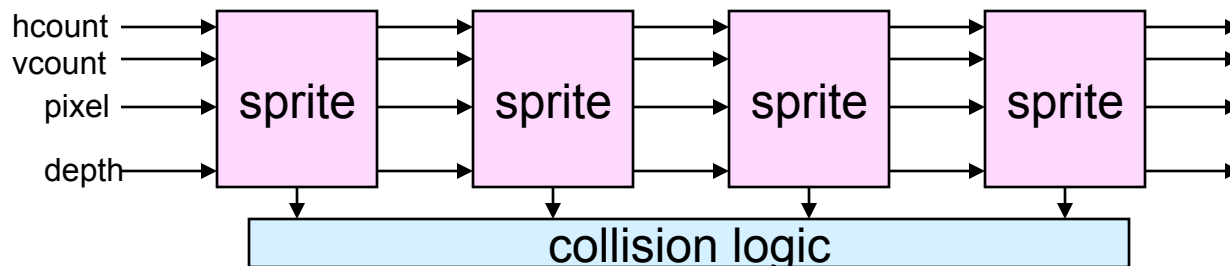
# Character Display

(80 columns x 40 rows, 8x12 glyph)

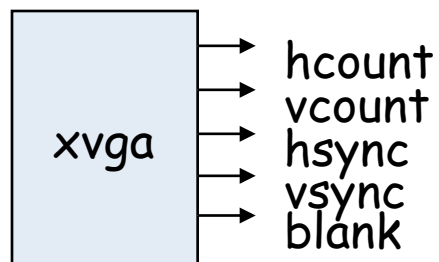


# Game Graphics using Sprites

- Sprite = game object occupying a rectangular region of the screen (it's bounding box).
  - Usually it contains both opaque and transparent pixels.
  - Given (H,V), sprite returns pixel (0=transparent) and depth
  - Pseudo 3D: look at current pixel from all sprites, display the opaque one that's in front (min depth): see sprite pipeline below
  - Collision detection: look for opaque pixels from other sprites
  - Motion: smoothly change coords of upper left-hand corner
- Pixels can be generated by logic or fetched from a bitmap (memory holding array of pixels).
  - Bitmap may have multiple images that can be displayed in rapid succession to achieve animation.
  - Mirroring and 90° rotation by fooling with bitmap address, crude scaling by pixel replication, or resizing filter.

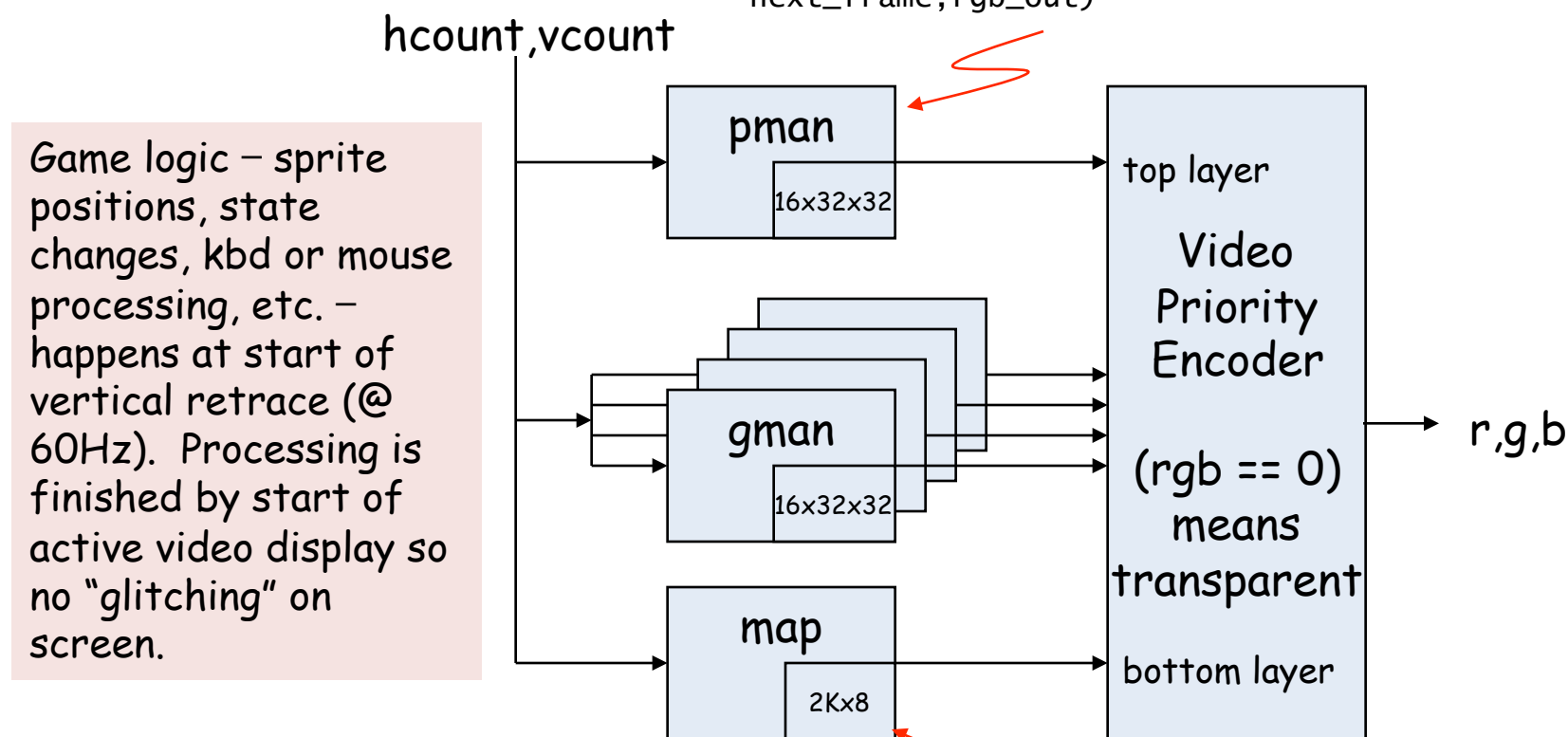


# Pacman



Sprite: rectangular region of pixels, position and color set by game logic. 32x32 pixel mono image from BRAM, up to 16 frames displayed in loop for animation:

`sprite(clk, reset, hcount, vcount, xpos, ypos, color, next_frame, rgb_out)`



4 board maps, each 512x8  
 each map is 16x24 tiles (376 tiles)  
 Each tile has 8 bits: 4 for move direction (==0 for a wall), pills