ECE532H1S Final Report

FLASH ACTIVATED STILL IMAGE CAPTURER

Team Members:	Andrew Michell	990879144
	Enrico Baldovino	990884101
Submitted to:	Professor Paul Chow	
	April 12, 2004	

Contents

1.0 Overview ... 3

2.0 Results and Future Improvements ... 4

3.0 Block Description ... 5

3.1 MicroBlaze Processor ... 5

3.2 OPB External Memory Controller (EMC) ... 5

3.3 OPB General Purpose I/O (GPIO) ... 6

3.4 Video Processor (vidcap) ... 6

3.5 YCrCb Buffer Viewer ... 8

4.0 Design Tree Description ... 10

1.0 Overview

This report describes the implementation of a *Flash Activated Still Image Capturer*, a video processing project proposed by Professor Steve Mann. The basic operation of the overall system is as follows:

1. A camcorder provides a continuous video signal to the Xilinx Multimedia Board through the board's component video input connection.
2. The ADV7185 video decoder in the board samples the incoming video signal and converts it to YCrCb digital format.
3. A video processing unit, which is enabled through the GPIO, measures the luminance of each incoming video frame and compares it to a luminance threshold (also specified at runtime through the GPIO).
4. A bright camera flash in the video picture causes the frame's luminance to exceed the threshold value, prompting the video processing unit to store the subsequent frame into external ZBT RAM.
5. The memory where the picture is stored is read and written to a file by an XMD command line script.
6. This file is read by *YCrCb Buffer Viewer*, a custom viewing application written in Java, which displays the captured picture by sequentially drawing each pixel.

The system was built upon the *zbt_test* design by Lesley Shannon. This design included a MicroBlaze processor and one ZBT RAM bank connected to the OPB bus through an External Memory Controller (EMC). The video input core, *vidcap*, was connected to the OPB bus in master configuration. This core was enabled by the user at runtime through the GPIO. The original vidcap core, written by Monty Nandra, continuously read YCrCb digital video data from the board's video decoder and stored it in 32-bit format on the ZBT RAM module. Modifications were made to the vidcap core to enable measurement of the energy of each frame. The core's operation was changed such that it would only write frames into memory once it has found a frame with an energy measurement that exceeds a certain threshold. A more detailed description of the system's components can be found in section 3.0 of this document.

2.0 Results and Future Improvements

The Flash Activated Still Image Capturer successfully measures the brightness of each incoming frame by summing the values of each luminance sample sent by the video decoder. This sum is compared to a threshold level, which can be set by writing the desired value in the GPIO register. The value written must end with the last two bits set high, as these two bits also function as switches, which enable the video capture module to write frames to memory. Frames are not written to memory until after the module senses a frame above the set threshold level.

Although the system works and achieves its objectives, there are some drawbacks that can potentially be improved upon:

1) Due to time constraints, we were unable to develop a hardware module that displays the captured frame on a VGA or TV monitor once it is stored in memory. This would eliminate the need to transfer the memory contents to a file and open display it using the YCrCb Buffer Viewer Java application.

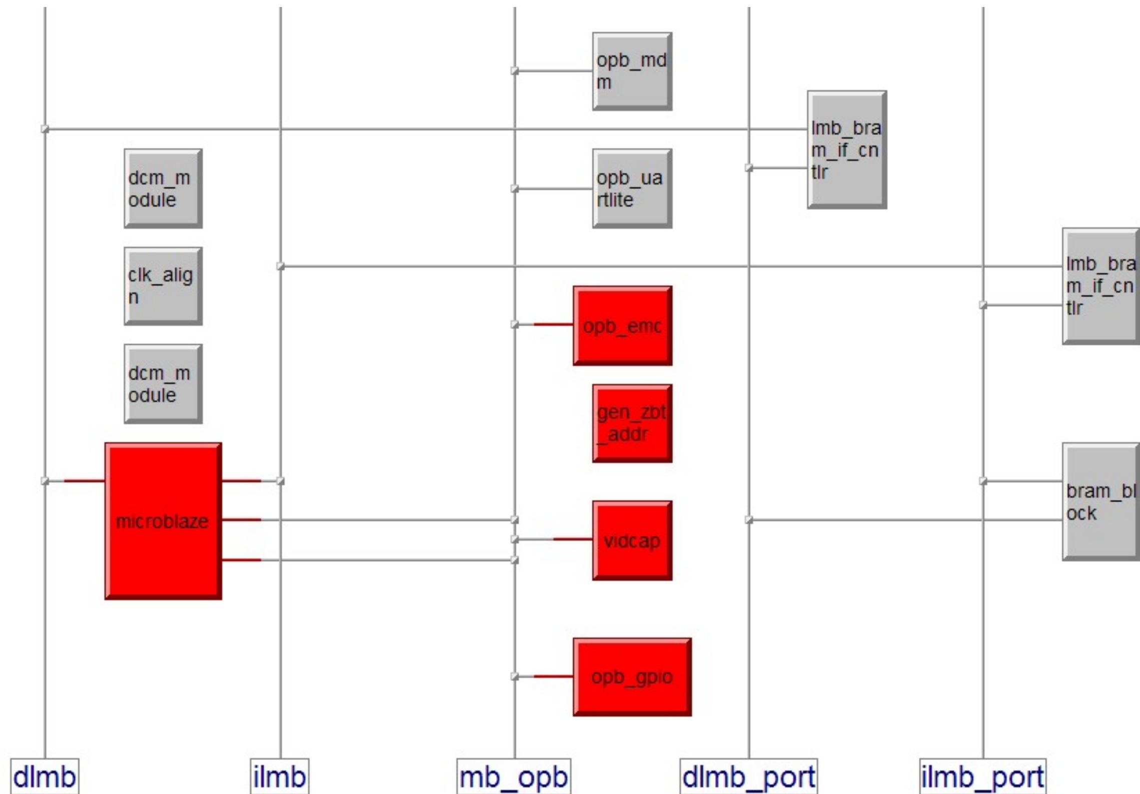
2) It was observed that the system is quite dependent on the light conditions where it is being used. In rooms with bright fluorescent lighting, flash frames become less differentiable from regular frames. This problem may be solved by modifying the method by which the energy of each frame is measured such that there is a larger discrepancy between normal frames and those with a flash.

3) Ideally, only one frame should be captured and written to memory. However, we found that when only one frame is written, blank spots appear in the picture. This may be due to the fact that since the video capture module is connected to the OPB as a master, it must be granted permission to write to the RAM. This process might not be happening fast enough for every sample to be written. (Note that this is only a hypothesis; the system should be simulated and tested further in order to find the exact cause of the problem.) Thus, it was necessary to allow the module to write several frames into the same location on the external memory.

4) One might consider modifying the operation of the video processing core so that frames are continually written to memory until a flash frame is found, at which point writing is halted. In this case, the last frame written to memory would be the flash frame, instead of the subsequent frames. This is essentially the inverse of the system previously described.

3.0 Block Description

The following figure is a schematic diagram of the overall system as it was implemented on the Xilinx Multimedia Development Board. The hardware modules shaded in red will be described in further detail in this section.



3.1 Microblaze Processor

The Microblaze processor does not take part in digital video processing, as this is done by the video processing module (see section 3.4). However, it still plays a relatively important role in the system by running the Microblaze Debug Module (MDM). The user must be able to write values to the GPIO to specify the brightness threshold and enable the video processing module to write to memory. The user must also be able to access the ZBT RAM directly to transfer its contents to a file. Access to the GPIO and the RAM (as far as we know) can only be done by connecting to the MDM stub through XMD.

3.2 OPB External Memory Controller (EMC)

The EMC module is used to control reads and writes to the external memory. The EMC is connected to one bank of external memory. Our project used Lesley Shannon's zbt_test project to implement the external memory. There is an extra module, gen_zbt_addr, which, as stated in the project's README file, "flips the address bus to select the correct bits".

The following changes were made in Add/Edit Cores:

- set the base address to 0x80100000
- set the end address to 0x801fffff

3.3 OPB General Purpose I/O (GPIO)

The GPIO module is used to start the flash detector, stop the flash detector and set the threshold of the energy. The GPIO core was added to the project in Add/Edit Cores using the following steps:

- set the base address of the GPIO core to 0x80000300
- add the GPIO core to the OPB bus as a slave
- connect the GPIO_d_out signal to inflags signal of vidcap
- set the WIDTH parameter to 32

The upper 30 bits of the GPIO_d_out signal are used to set the threshold for the vidcap core and the lower 2 bits are used as start/stop bits. To start the flash detector, write a 0xXXXXXXX3 to address 0x80000300, which is the base address of the GPIO core. The threshold of the vidcap core will be set to 0xXXXXXXX0. To stop or reset the flash detector, write a 0x00000000 to address 0x80000300.

3.4 Video Processor (vidcap)

The vidcap module is used to capture incoming video data from the off chip decoder and write the video data. A state machine and two counters were added to Monty's vidcap to measure the energy of each video frame and write a configurable number of frames following the high energy frame to memory. The vidcap core was added to the project by following these steps:

In Add/Edit Cores,

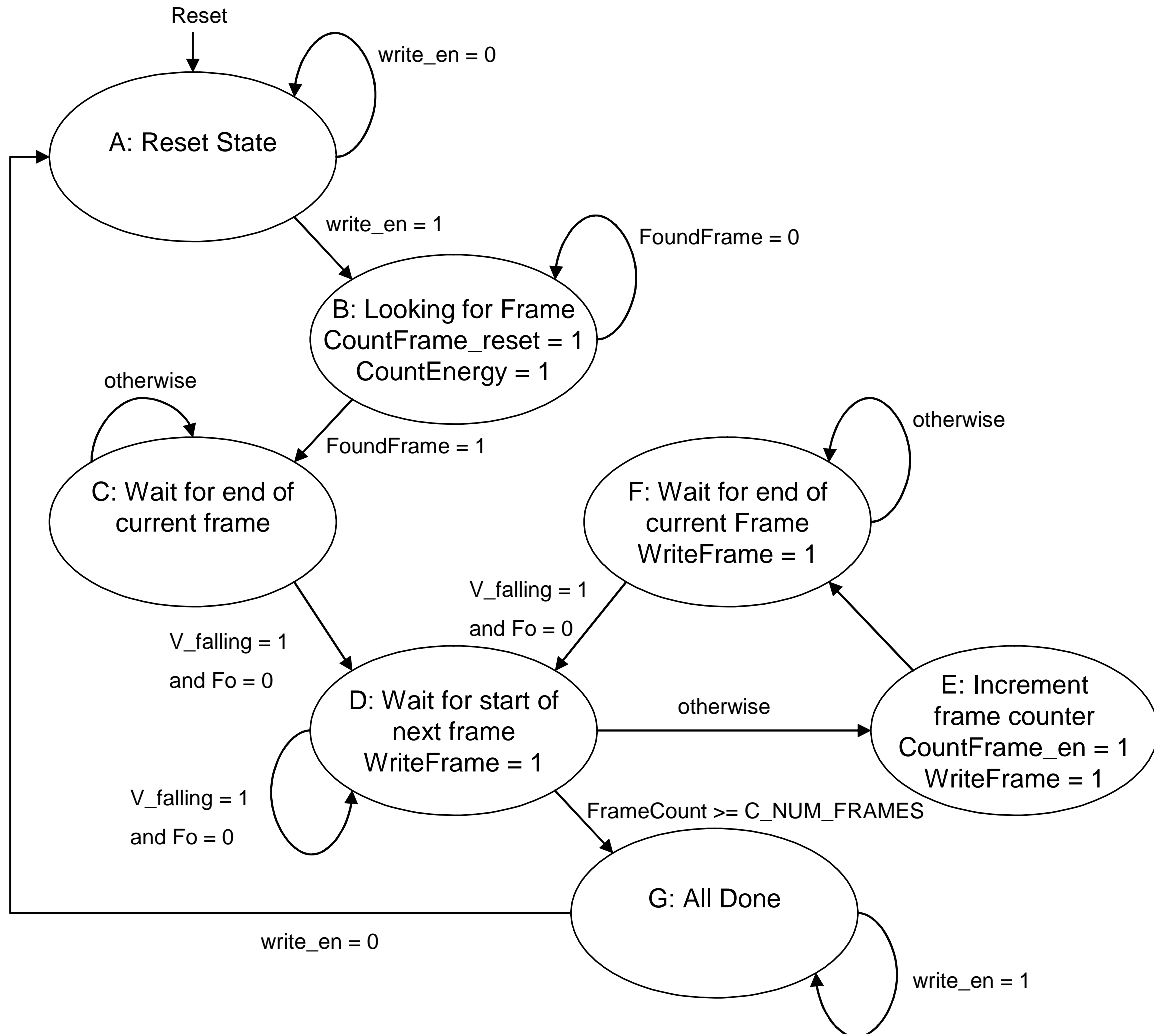
- create a new vidcap instance
- connect the vidcap core to the OPB as a master
- add the following signals:
    - led1
    - led2
    - inflags
    - vid_clk
    - YCrCb_in
- connect the inflags signal to the GPIO_d_out signal of the GPIO core
- set the C_FBADDR parameter to the frame buffer address
- set the C_NUM_FRAMES to the number of frames that will be written to memory (maximum of 15)

In the system.ucf file,

- connect vidcap_0_led1 to the pin for user_led0
- connect vidcap_0_led2 to the pin for user_led1
- connect vidcap_0_vid_clk to the 27 Mhz video signal (CHAN1_LINE_LOCK_CLOCK1)
- connect vidcap_0_YCrCb_in to the video data (CHAN1_VIDEO_DATAn)

Description of Changes Made

The original vidcap core was modified by adding two counters and a state machine to control the two counters. One counter is used for summing the luminance of each pixel and the other is used for counting the frames that have been written to memory. The state machine is used to control the counters and the states are labeled A through G. The following figure is the state diagram of the state machine.



Description of each state:

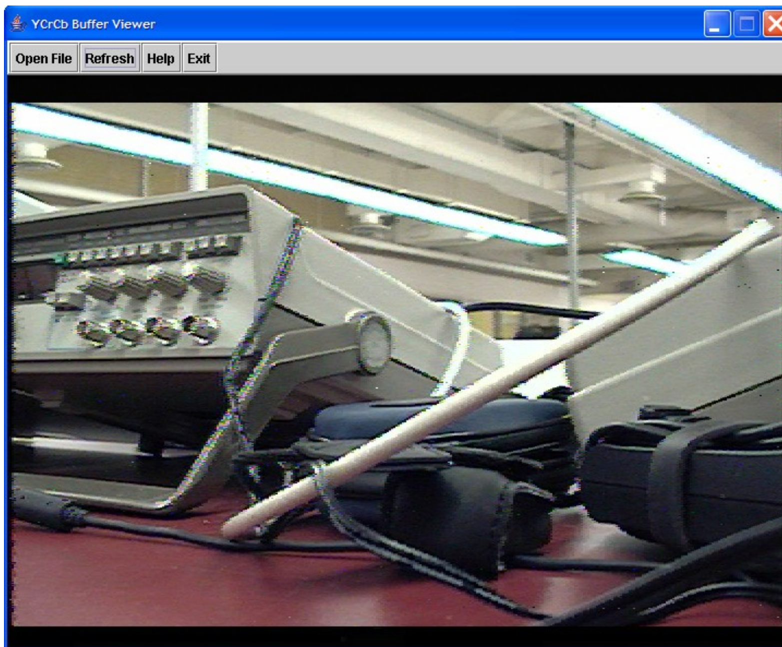
A. In this state the user has not written a number ending in 3 to the GPIO. The state machine will transition to state B when the user starts the flash detector.
B. In this state the luminance of each pixel is being added to the energy counter, and the frame counter is reset. If the value in the energy counter is greater than the threshold set by the GPIO then a FoundFrame signal will go high and the state will change to state C.
C. In this state the frame with high energy has been detected and we are waiting for the high energy frame to end so that we can write the following frames to memory. At the end of this frame V_falling will be high and Fo will be low.

When this happens the state will chage to state D. V_falling is high and Fo is low for a few clock cycles between each frame.

D. In this state we are waiting for the beginning of a frame we wish to write. If the frame counter is equal to C_NUM_FRAMES then the state will change to state G. Otherwise, we wait for the beginning. This happens when V_falling is not high and/or Fo is not low. At the beginning of the next frame the state will change to E.
E. In this state the frame counter is incremented. At the next clock cycle the state will change to F.
F. In the state a frame is written. At the end of a frame V_falling will be high, Fo will be low and the state will change to state D.
G. In this state all of the frames have been written to memory. The state machine will stay in this state until the user resets the flash detector, at which point the state will change to state A.

The vidcap core was simulated using Quartus to ensure that the state machine functioned properly and that video data was written to memory. For more information on the data transmitted by the video decoder, please refer to the document "XAPP286: Line Field Decoder", which can be found at the end of this report.

3.5 YCrCb Buffer Viewer



Since there was not enough time to develop a video output hardware module in the Multimedia Board, a Java application was written to display the picture that was captured and stored into memory. The viewer opens a file named *cap*, which is created by the *sst* script, which can be found in the project directory (to run the script from the project directory in XMD, type "source sst" at the command prompt). The *cap* file simply lists the contents of the ZBT RAM, four bytes per line. Each line contains the hexadecimal values for two luminance (Y) samples and two chrominance (Cr & Cb) samples in the following format: 0xY1CrY2Cb. It is important to note that the values are stored in

interlaced format, meaning all the odd video lines are written first, followed by the even lines. The following code segment, extracted from Viewer.java, performs the conversion of the file to RGB values for each pixel.

```
all: for( y=0; y<525; y++ ){
    // iterate over all samples per line
    for( x=0; x<720; x+=2 ){
        // read a line from the cap file
        data = in.readLine();
        if(data == null) break all;
        if(data != ""){
            try{
                // isolate each value and convert to decimals
                sY1 = data.substring(2,4);
                sCr = data.substring(4,6);
                sY0 = data.substring(6,8);
                sCb = data.substring(8,10);

                Y1 = Integer.parseInt(sY1,16);
                Cr = Integer.parseInt(sCr,16);
                Y0 = Integer.parseInt(sY0,16);
                Cb = Integer.parseInt(sCb,16);
            } catch (Exception e){}

            // convert YCrCb values to RGB values
            Y = (Y1 + Y0)/2;
            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);

            // limit values to {0,255}
            if(R>255)   R = 255;
            if(G>255)   G = 255;
            if(B>255)   B = 255;
            if(R<0)     R = 0;
            if(G<0)     G = 0;
            if(B<0)     B = 0;

            c = new Color((int)R, (int)G, (int)B);
            g.setColor(c);
            if(y<263)
                // draw odd lines
                g.drawLine(x,y*2,x+1,y*2);
            else
                // draw even lines
                g.drawLine(x,(y-263)*2+1,x+1,(y-263)*2+1);
        }
    }
}
```

For more information on the conversion from YCrCb to RGB, and how this may be achieved in hardware, please refer to the document XAPP283: Color Space Converter", which can be found at the end of this document.

4.0 Design Tree Description

The following items have been included in the submission:

zbt_test: the project main directory.

vidcap_v1_00_b: the modified vidcap core; this can be found in the pcores directory. The modified file is vidcap.vhd, which can be found in pcores/vidcap_v1_00_b/hdl/vhdl/.

sst: the script which writes the memory contents to a file named *cap*.

viewer: the directory containing the YCrCb Buffer Viewer application.

Instructions to run the system:

1) Connect a video camera to the Xilinx Multimedia Board through the component video-in connection.
2) Download the hardware configuration onto the board.
3) Open XMD and connect to the stub.
4) Write a value to the GPIO to set the luminance threshold using the following command: "mwr 0x80000300 0x02000003". This command sets the threshold to 0x02000000. The 3 on the end enables the vidcap core to write frames to memory. At this time, led1 should turn on. This means that the vidcap core is searching for a bright frame.
5) Use a camera or some other light source to generate a bright frame.
6) Once a bright frame is found and subsequent frames are written to memory, led1 should turn off and led2 should turn on.
7) In XMD, type "source sst" to create the cap file.
8) Open the cap file using the YCrCb Buffer Viewer application.
9) Reset the system by typing in the command "mwr 0x80000300 0".

XAPP286 (1.0) December 13, 2001

Line Field Decoder

Author: Gregg Hawkes

Summary

The video standard ITU-R.656 uses the sample definition defined in ITU-R BT.601 and SMPTE 125M. The standards describe how video field and line timing are embedded in the bit-parallel data through the use of reserved data words known as timing reference signals (TRS)[1] [2]. The MicroBlaze™ and Multimedia development board uses and decodes this information to regain the timing of the incoming video stream. This timing is then passed to other algorithms inside the device.

The reference design available with this application note decodes TRS information and supplies timing control signals to the rest of the video algorithms. The design is a modification of design files associated with **XAPP248: Digital Video Test Pattern Generators**.

Component Video Voltages

The Associated Digital Video Data Values in Each Video Line

The MicroBlaze and Multimedia development board uses an Analog Device decoder, ADV7185, to sample (up to four times over-sampling) incoming analog video and convert it to digital values. Figure 1 shows how the voltages relate to the digital values sent to the Virtex™-II or Spartan™-II device. Notice the 8-bit data values of `FFh` (decimal 256) and `00h` (decimal 0) in Figure 1 do not occur in the normal stream of video. Therefore, `FFh` or `00h` can be inserted in non-picture parts of a line to mark timing information. These inserted symbols are the timing reference signals (TRS),

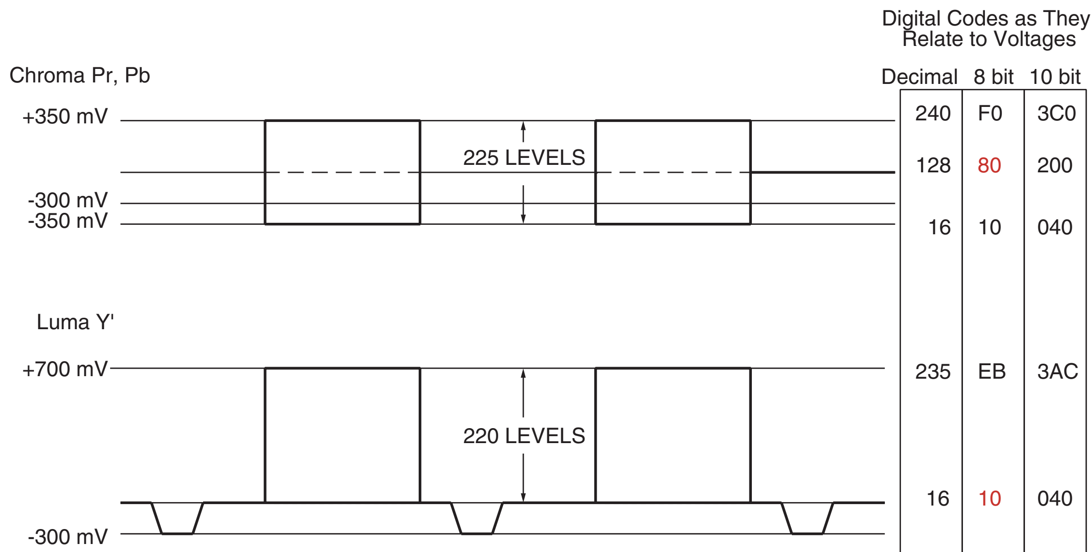


Figure 1: **Analog Component Video Voltage Levels and Associated Digital Values**

© 2001 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and disclaimers are as listed at http://www.xilinx.com/legal.htm.
All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

The MicroBlaze and Multimedia development board supports 10-bit video data coming from the decoder device. Notice in Figure 1 how this compares to 8-bit data values. The two extra bits beyond the normal 8-bit data are appended to the rightmost part of the 8-bit data and are assumed to be fractional. When the fractional parts are zero, the specifications just "leave them off".

For example, the bit pattern `10010001` would be expressed as `145d` or `91h`, whereas the pattern `1001000101` is expressed as `145.25d` or `91.4h`. In fact, the data paths on the development board and inside the Virtex-II device were designed specifically for future MPEG investigation and are 12-bits wide. The extra two data bits beyond the10-bit data are appended to the MSBs, leaving more "headroom" for calculations.

Figure 2 and Figure 3 summarize the embedded timing format described fully in the video standard ITU-R BT.656. The significant components of a single horizontal line are shown; "front porch", "horizontal sync", "back porch", and "active video". The number of samples allocated to each horizontal line for the NTSC and PAL standards are also shown.

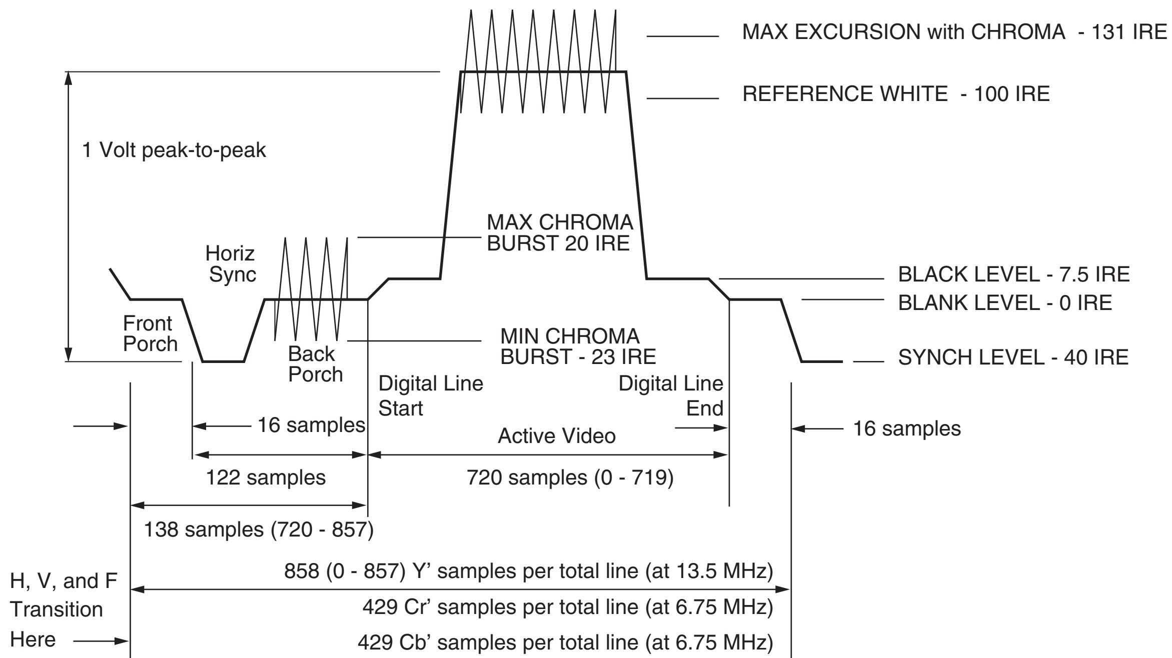


Figure 2: **Composite NTSC (525 Line With Set-up) Horizontal Scan Line Detail**

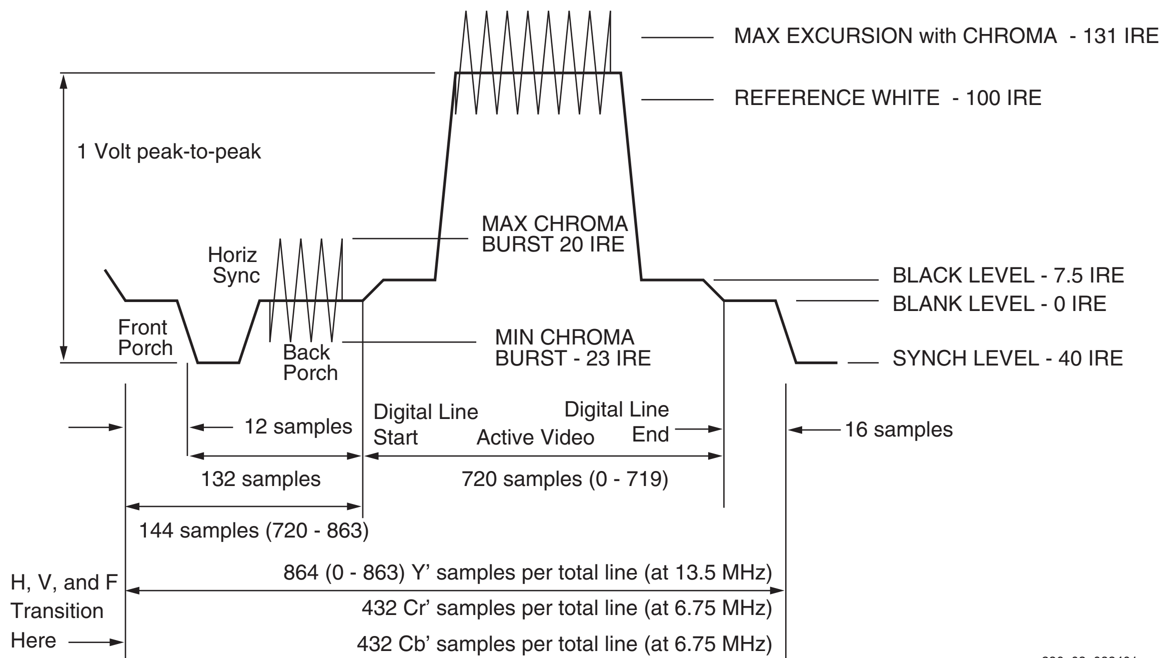


Figure 3: **Composite PAL (625 Line With Set-up) Horizontal Scan Line Detail**

Video Timing Information Embedded in Each Video Line

As shown in the previous figures, the video data words are conveyed as a 27 MHz data stream, in the following order:

Cb0, Y'0, Cr0, Y'1, Cb1, Y'2, Cr1, Y'3, Cb2, Y'4, Cr2, Y'5...

All the data values are sampled on the rising edge of the 27 MHz clock. In an 8-bits-per-word implementation, the data values `FFh` and `00h` are used to form the TRS preamble. A TRS preamble consists of three words, `FFh`, followed by `00h`, followed by `00h`. In 10-bit implementations, the data values `3FFh` and `000h` are used for the TRS preamble. The TRS preamble and following XY word are decoded and combined with various counts, such as line count, to completely specify the NTSC or PAL timing to the rest of the video algorithms in a system design.

Field and frame timing is actually embedded in the data stream by the word following the TRS. This word, known as XY can be decoded for different timing events.

The terms SAV and EAV are abbreviations for "Start of Active Video" and "End of Active Video", respectively. SAV is identified with the timing reference code (`FF 00 00`) followed by the XY word where bit four of the XY word is a logic Low. EAV is identified as a logic High in bit four of the XY word. SAV signals that active video pixels will follow. EAV signals that horizontal blanking follows. Figure 4 shows this detailed horizontal pixel information for a horizontal NTSC 525 line and Figure 5 shows the same information for a PAL 625 line.

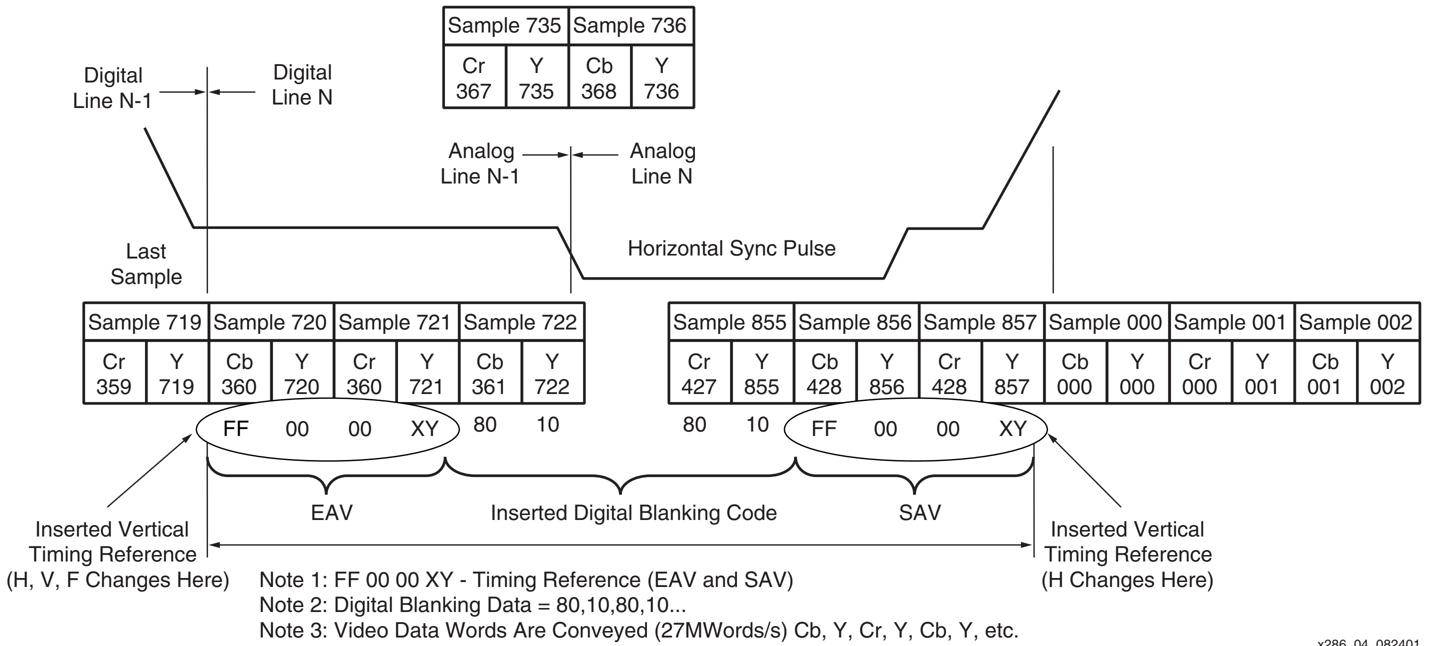


Figure 4: **NTSC (525 Line) Horizontal Scan Line Detail, Inserted Codes**

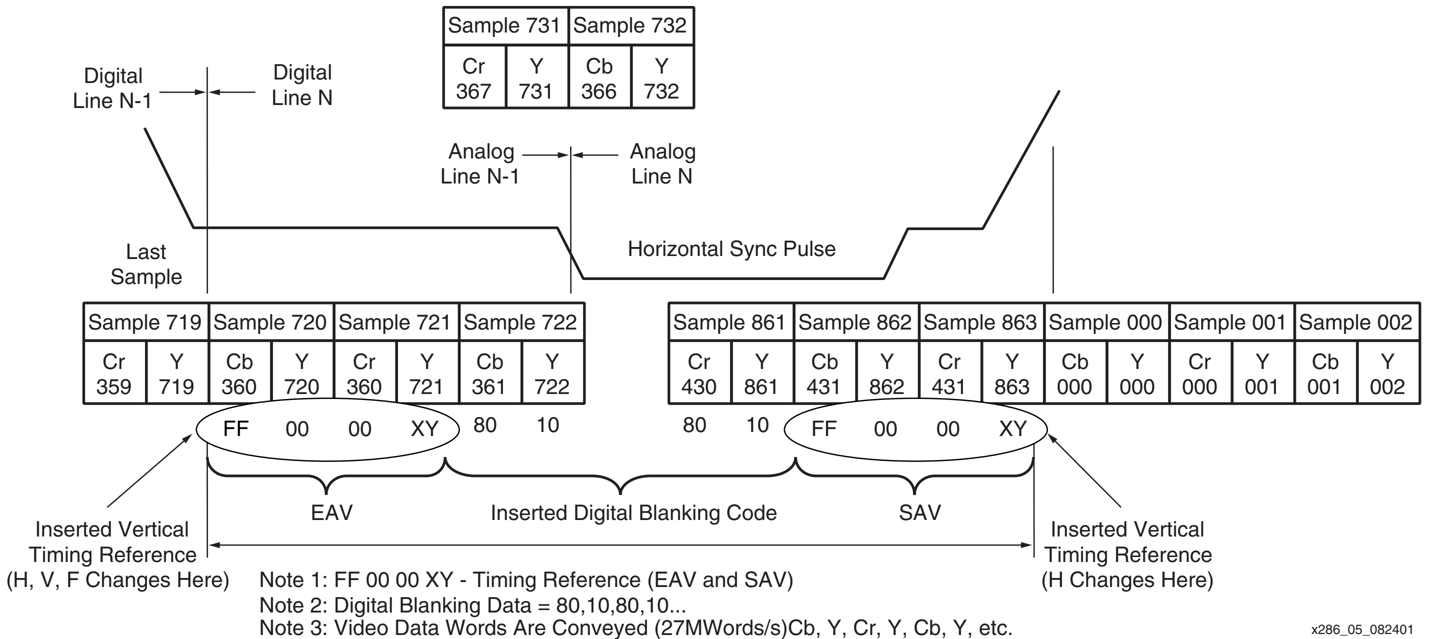


Figure 5: **PAL (625 Line) Horizontal Scan Line Detail, Inserted Codes**

Field number and vertical blanking are also conveyed by XY, following a field ID. The "F bit" or bit-position six and the "V bit" or bit-position five are decoded as follows:

F = 0, denotes field 1

F = 1, denotes field 2

V = 0, denotes no vertical blanking

V = 1, denotes vertical blanking

Video Line and Field Timing

There are more pixels in the blanking period for PAL than NTSC (144 vs. 138), but the number of active pixels, in a line, are the same (720). While there are more lines in PAL (625 per frame for PAL vs. 525 for NTSC), there are less frames per second (25 for PAL vs. 30 for NTSC). In fact, the pixel frequencies are approximately equal.

As ideas and inventions are introduced, video evolves in many different ways creating enormous variations in formats. For example, the introduction of computer systems generated a desire to mix video broadcast systems and computers further complicating formats with the notion of the square pixel. When looking at a table of information about pixels per line, lines per frame, and frames per second, a basic understanding of the specific format is necessary.

Figure 6 and Figure 7 show the detail associated with NTSC and PAL vertical information conveyed by the standard.

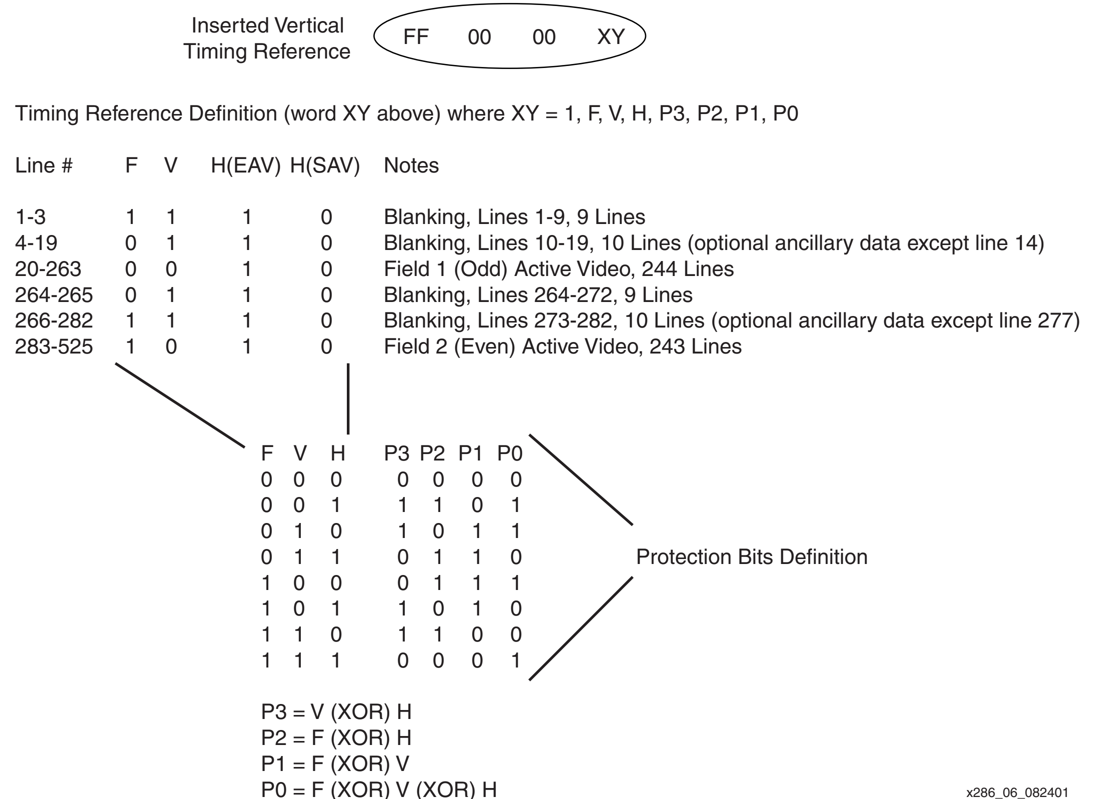


Figure 6: **NTSC (525 Line) Vertical Timing Reference (8-bit Implementation)**

Inserted Vertical Timing Reference: FF 00 00 XY

Timing Reference Definition (word XY above) where XY = 1, F, V, H, P3, P2, P1, P0

Line #	F	V	H(EAV)	H(SAV)	Notes
1-22	0	1	1	0	Blanking, Lines 1-22, 22 Lines
23-310	0	0	1	0	Field 1 (Odd) Active Video, 288 Lines
311-312	0	1	1	0	Blanking, Lines 311-312, 2 Lines
313-335	1	1	1	0	Blanking, Lines 313-335, 23 Lines
336-623	1	0	1	0	Field 2 (Even) Active Video, 243 Lines
624-625	1	1	1	0	Blanking, Lines 624-625, 2 Lines

Protection Bits Definition

F	V	H	P3	P2	P1	P0
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1

P3 = V (XOR) H
P2 = F (XOR) H
P1 = F (XOR) V
P0 = F (XOR) V (XOR) H

x286_07_082401

Figure 7: **PAL (625 Line) Vertical Timing Reference (8-bit Implementation)**

Figure 8 and Figure 9 shows the vertical detail scope waveforms as if H, V, F, HSYNC, BLANK, and VSYNC were decoded from the digital component video and presented. This is essentially what the line field decoder state machine does.

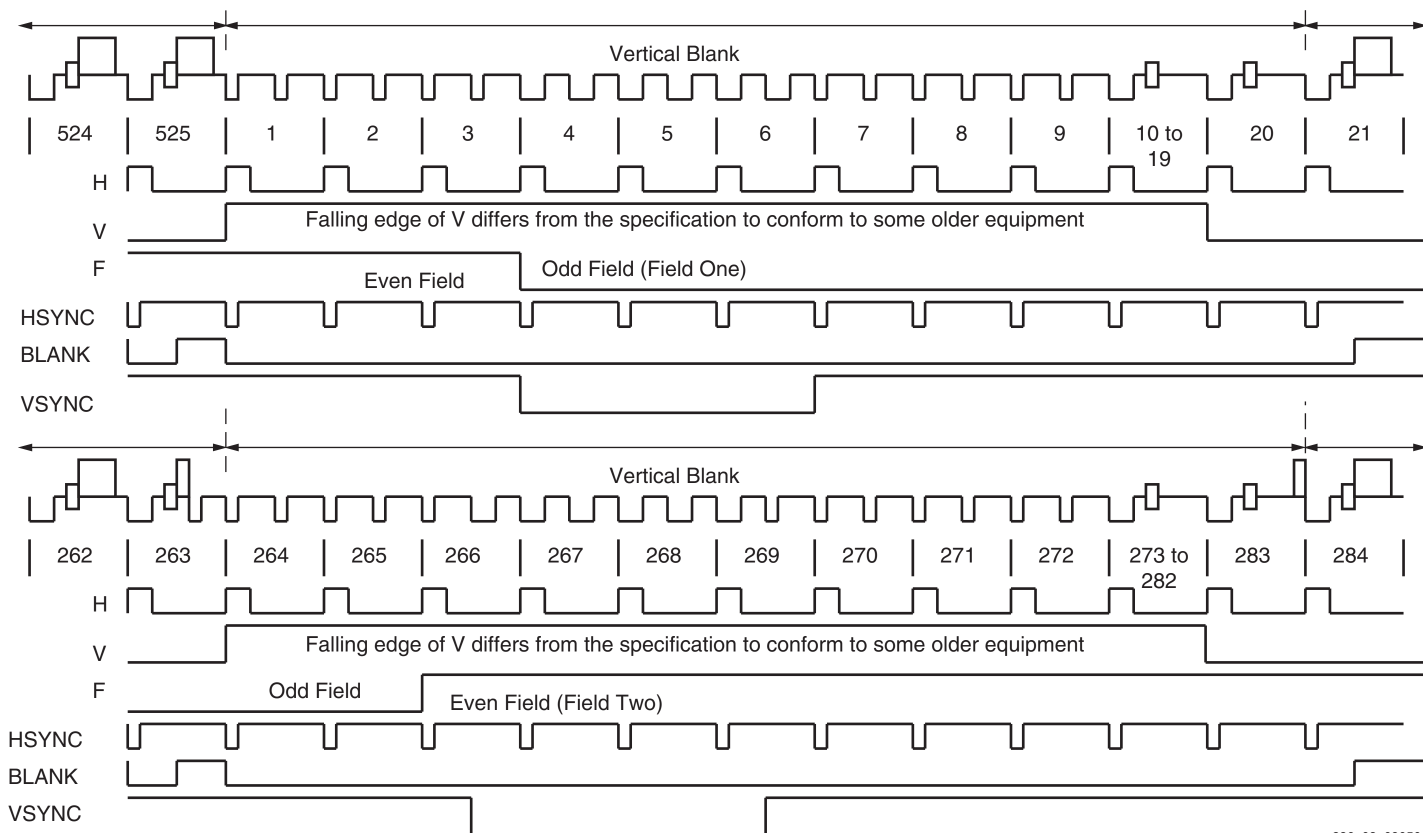


Figure 8: **Composite NTSC 525 Vertical Timing Detail**

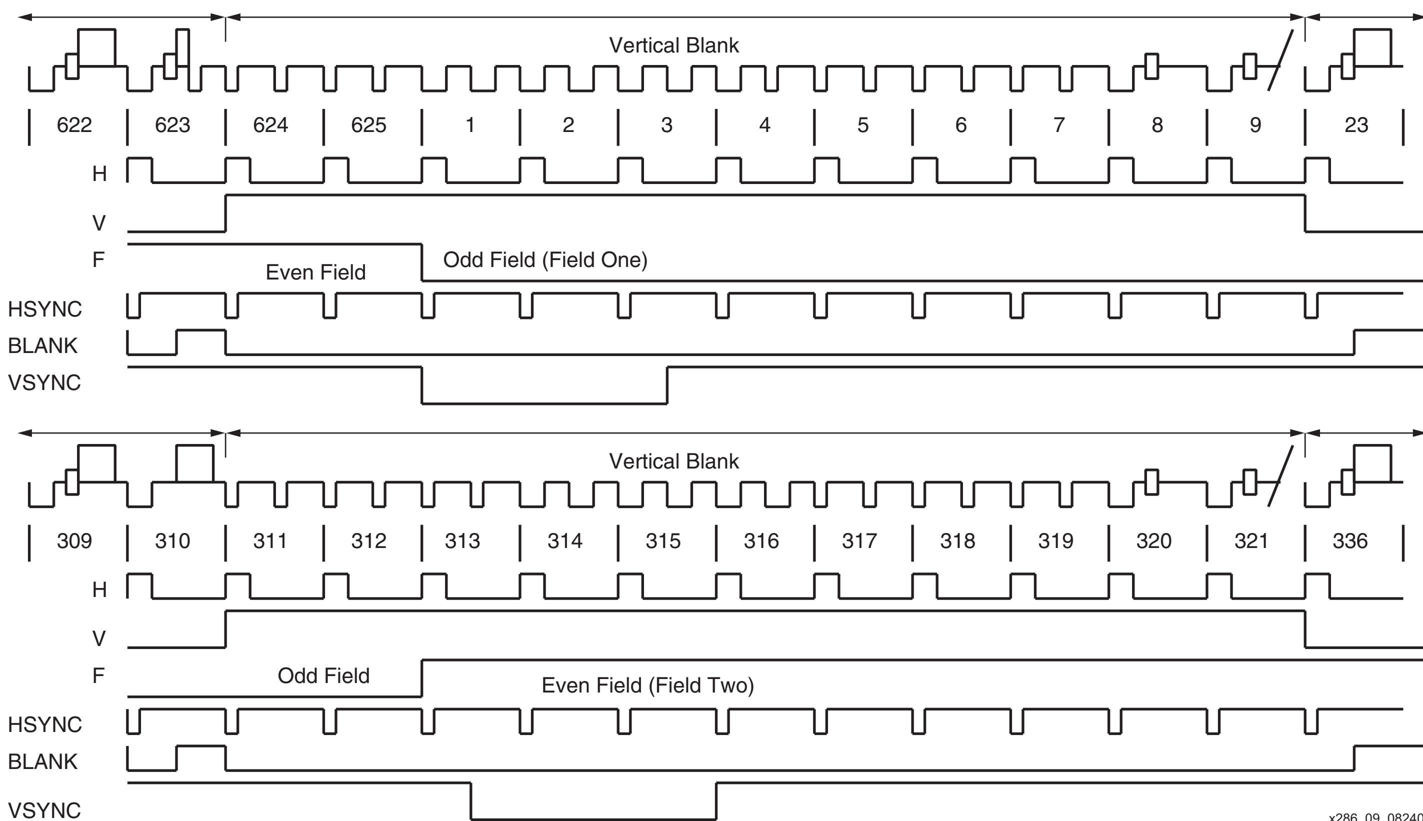


Figure 9: **Composite PAL 625 Vertical Timing Detail**

Reference Design

The reference design for this application note, in both VHDL and Verilog code, is available on the Xilinx FTP site at: **ftp://ftp.xilinx.com/pub/applications/xapp/xapp286.zip**. A simple description of the video line and field decoder module state machine function is:

1. Auto detect the format by counting clocks between EAV and SAV
2. Determine and output the horizontal sync or H bit
3. Determine and output the Field bit
4. Determine and output the line and pixel count

This "Line Field Decoder" reference design is a modification of the auto-detect module found in **XAPP248**. It uses the autodetect code to track the TRS symbols and report the format. Steps 2, 3, and 4 are added to support other modules in the demonstration board.

The auto-detect module in XAPP248 examines a digital video stream to determine the matching video standard. The supported video standards are listed in Table 1.

Table 1: **Supported Video Standards**

Video Format	Corresponding Standards
NTSC 4:2:2 component video	SMPTE 125M, ITU-R BT.601, ITU-R BT.656
NTSC composite video	SMPTE 244M, SMPTE 259M
NTSC 4:4:4 component 13.5 MHz sample	SMPTE RP174
PAL 4:2:2 component video	ITU-R BT.656
PAL composite video	EBU 3280-E
PAL 4:2:2 16 x 9 component video	ITU-R BT.601
PAL 4:4:4 component 13.5 MHz sample	ITU-R BT.799

Since the Microblaze and Multimedia demonstration board only supports NTSC 4:2:2 component video and PAL 4:2:2 component video, the design could be made smaller by eliminating the other standards. XAPP248 has details of a finite state machine used to track timing reference symbols.

Once TRS symbols are being tracked accurately, the horizontal sync and field bits are decoded by looking at the SAV XY word. The line count must be determined. When the F-bit transitions from Low to High, a line counter can be loaded with the correct value based on the format. If NTSC then load 266, otherwise use the PAL value of 313.

Conclusion

The design receives a pixel clock (27 MHz) used to clock in each Y', Cr, and Cb value. Currently pixels pass through the module with a delay. Future additions to the module might zero them or duplicate them during blanked portions of a line. This will produce different effects in the 422 to 444 module. The outputs of this module are a signal that suggests the incoming video is NTSC or PAL (PAL_NTSC_out), a line count (lcnt), the H, V, and F signals and the three sync signals, hsync_out, vsync_out, and blank_out. Figure 8 and Figure 9 the signal behavior.

Downstream modules that receive the ITU656 stream will need this information to further process and store the input pixels.

Each input video stream will use this code to allow other data path elements and control elements examining the input video stream to know what format the stream is (NTSC or PAL), and to know what pixels are available and any given time. The simple control logic easily runs at the pixel rate of 27 MHz in the Virtex-II and Spartan-II families.

The results of the synthesis and implementation are included for this simple controller here:

Table 2: **Size and Performance Results using FPGA Express 3.5, Xilinx 3.3i**

Part Number	Flip-Flops	LUT	Ports	Clock Latency	Speed
XC2V1000-5	117	152	38	8	8 ns (125 MHz)
XC2S200-5	117	152	38	8	11 ns (91 MHz)

References:

1. The video standards beginning with ITU come from the International Telecommunication Union. ITU-R BT.656 and by ITU-R BT.601 standards are available on the International Telecommunication Union's web site, **http://www.itu.int/itudoc/itu-r/rec/bt/** for a small fee. The SMPTE or Society of Motion Picture and Television Engineers standards can be found on **http://www.smpte.org** and will also require membership or a fee.
2. "Video Demystified", by Keith Jack, published by Harris, ISBN 1-878707-23-X, is a good beginners guide to video techniques. It can be read or purchased on line at the following URL; **http://www.video-demystified.com**
3. Analog Devices ADV7194 Data Sheet, "Professional Extended-10 Video Encoder with 54 MHz Over Sampling". URL: **http://www.analog.com**

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/13/01	1.0	Initial Xilinx release.

XAPP283 (v1.1) April 22, 2001

Color Space Converter

Author: Latha Pillai

Summary

This application note describes three ways to implement the Y'CrCb Color Space to R'G'B' Color Space conversion necessary in many video designs. The tick marks on red, green, blue, and Luma, assume the components are in the gamma corrected space. No gamma correction is applied to color difference signals Cr and Cb.

The first implementation shows how to simply write behavioral Verilog to describe the conversion equations, and then synthesize to a silicon target. This technique infers MULT_ANDs for the constant coefficient multiplier.

The second implementation uses the Xilinx feature of embedded RAM functioning as a Look-up Table (LUT), or ROM, to store all possible intermediate results for the terms in the three equations. Since three of the seven total terms are identical, only five ROMs are needed. The depth of the ROM, 1K, is driven by the color component bit width of 10 bits or studio quality video. To target Spartan-II devices, either add more ROMs or use commercial 8-bit video instead of 10-bit studio quality.

The third implementation makes use of the embedded multiplier in the Virtex™-II series of devices to perform the color space conversion. Again, only five multipliers are used. The Verilog model using the embedded multiplier is synthesized, placed, and routed. The design has a clock performance of 185 MHz after place and route, using simple constraints.

Color Space Definition

The human eye has three types of photoreceptor cells called cones. Stimulating the cells causes the human brain to "perceive" color. Colors can be specified, created, and visualized using different color formats or "color spaces."

Different color spaces have historically evolved for different applications. In each case, a color space was chosen for reasons that may no longer be applicable. Maybe a choice was made on a particular color space because the math elements needed to process were simpler or faster. Maybe a certain choice was better because it required less storage and bandwidth on digital buses.

Whatever historical reasons caused color space choices in the past, the convergence of computers, the Internet, and a wide variety of video devices, all using different color representations, is forcing the digital designer today to convert between them. The objective is to have a common color space that all inputs are converted to before algorithms and processes are executed. The converters are useful for a number of markets, such as image processing and filtering. Their basic function is to convert from one color space to another. This application note describes one such conversion.

© 2002 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at http://www.xilinx.com/legal.htm. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

Three Color Space Examples

RGB Color Space

RGB color space is a simple and robust color definition used in computer systems and the Internet to help ensure that a color is correctly mapped from one platform to another without significant loss of color information. RGB uses three numerical components to represent a color. This color space can be thought of as a three-dimensional coordinate system whose axes correspond to the three components, R or red, G or green, and B or Blue. RGB is the color space that computer displays use. RGB corresponds most closely to the behavior of the human eye.

RGB is an additive color system. The three primary colors red, green, and blue are added to form the desired color. Each component has a range of 0 to 255, with all three 0s producing black and all three 255s producing white.

Y'CbCr Color Space

Y'CbCr Color Space was developed as part of the Recommendation ITU-R BT.601 for worldwide digital component video standard and is used in television transmissions. Y'CbCr is a scaled and offset version of the YUV color space where Y represents luminance (or brightness), U represents color, and V represents the saturation value. Here the RGB color space is separated into a luminance part (Y') and two chrominance parts (Cb and Cr).

As mentioned earlier, the historical reasons for this choice, over R'G'B', were to reduce storage and bandwidth. Since the eye is more sensitive to change in brightness than change in color, the reduction in bandwidth requirement seemed a valid trade for little or no visual difference.

Engineers found that 60 to 70 percent of luminance or brightness is found in the "green color." In the chrominance part Cb and Cr, the brightness information can be removed from the blue and red colors.

To generate the same color in the RGB format, all three color components should be of equal bandwidth. This requires more storage space and bandwidth. Also, processing an image in the RGB space is more complex since any change in the color of any pixel requires all the three RGB values to be read, calculations performed, and then stored. If the color information is stored in the intensity and color format, some of the processing steps can be made faster.

The result is that Cb and Cr provide the hue and saturation information of the color and Y' provides the brightness information of the color. Y' is defined to have a range of 16 to 235 and Cb and Cr have a range of 16 to 240 with 128 equal to zero. Because the eye is less sensitive to Cb and Cr, engineers did not need to transmit Cr and Cb at the same rate as Y'. Less storage and bandwidth was needed, resulting in design costs being reduced.

Converting from Y'CrCb to R'G'B'

A color in the Y'CrCb color space is converted to the RGB color space using the following equations:

$$R' = 1.164(Y' - 16) + 1.596(Cr - 128)$$
$$G' = 1.164(Y' - 16) - (0.813)(Cr - 128) - 0.392(Cb - 128)$$
$$B' = 1.164(Y' - 16) + 2.017(Cb - 128)$$

Where R'G'B' are gamma-corrected RGB values and Y', Cr, and Cb are 8-bit inputs.

For 10-bit inputs, the equations are:

$$R' = 1.164(Y' - 64) + 1.596(Cr - 512)$$
$$G' = 1.164(Y' - 64) - (0.813)(Cr - 512) - 0.392(Cb - 512)$$
$$B' = 1.164(Y' - 64) + 2.017(Cb - 512)$$

Figure 1 shows a direct mapping of the above three equations. Notice that three of the seven terms are duplicates. This term is computed once and fed to the output adders for the Y', Cr, and Cb results.

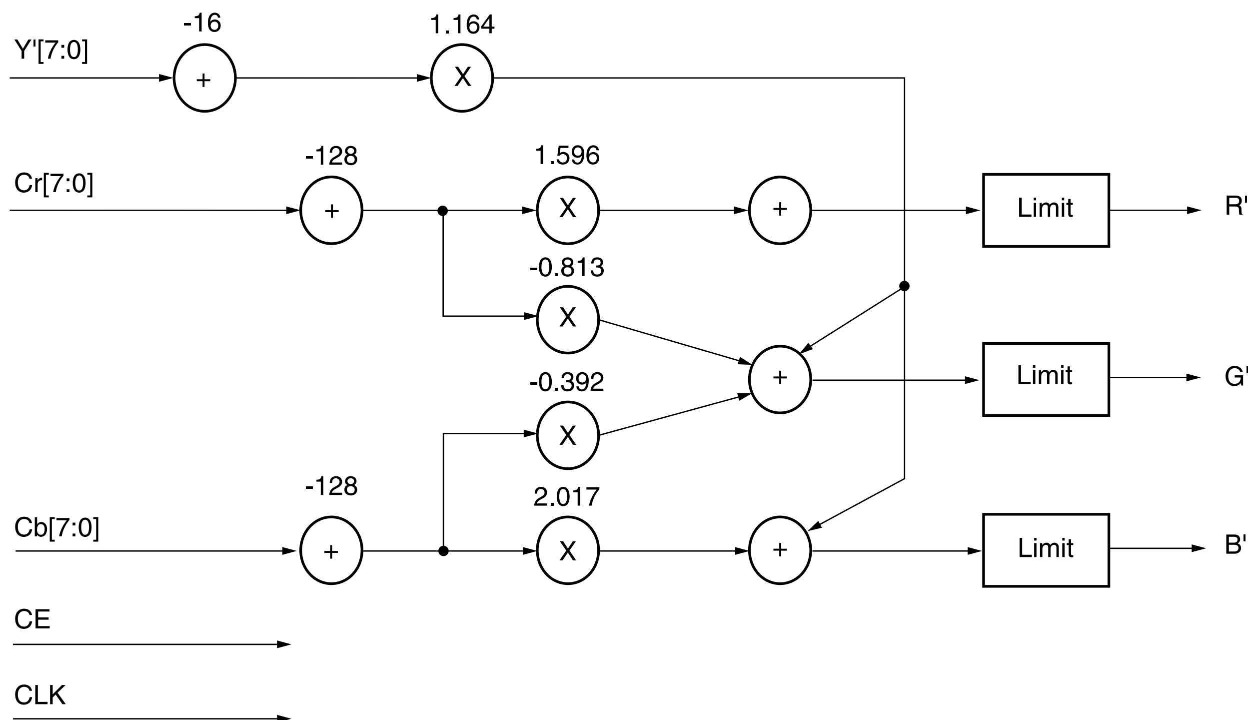


Figure 1: **Block Diagram Showing Math Elements**

Virtex-II Implementation Examples

The high density, on-chip memory in the Virtex-II designs increase overall system bandwidth by providing fast and resource-efficient FIFO buffers, shift registers, and CAMs. With embedded multipliers and improved arithmetic functions, Virtex-II solutions deliver over 600 billion MACs/s of Xtreme DSP performance.

There are up to 192 18 x 18 signed multipliers in a single device, supporting up to 36-bit signed multiplications. Cascading these multipliers supports even larger numbers. The multipliers can be combinatorial or pipelined, running between 140 MHz and 250 MHz depending on bit width. These features make Virtex-II devices the ideal choice for implementing the color space converter.

Verilog Examples

As mentioned at the start of this application note, there are three different implementation examples. The following are the results of synthesizing and implementing each example.

Three different implementation examples are detailed in this application note. A fourth example is a CoreGen distributed arithmetic approach. The CoreGen approach is not implemented, but estimated results are given. The following sections show the results of synthesizing and implementing each example.

Implementation Using Behavioral Verilog (gen_model.*)

In this implementation, the basic Y'CrCb2R'G'B' conversion equations are synthesized using Synplicity. All the signals are registered at the input and at the output. The synthesized EDIF file is then placed and routed using Design Manager. A timing constraint of 10 ns was given to the place and route tool. The implementation results are listed in the following tables.

Notes:

1. See Verilog file, `gen_model.v`.

Design Summary

Table 1: **Behavioral Implementation Design Summary**

Device	LUTs	FFs	Ports	Performance
XC2V500-5 (slowest speed grade)	258	52	68	14 ns / 71 MHz (inputs and outputs registered
XC2V500-5 (slowest speed grade)	260	85	68	9.4 ns / 106 MHz (one intermediate pipe stage)

Implementation Using Block RAM as Look-Up ROM (ram_model.*)

Y’, Cb, and Cr are 10-bits wide and so have a range of 0 to 1023. This would give the following values for each of the terms in the R’,G’, and B’ equations:

$$1.164(Y' - 16) = 1.164[(0 - 16) to (1023 - 16)] = 1.164(-16 to 1007)$$
$$1.596(Cr - 128) = 1.596[(0 - 128) to (1023 - 128)] = 1.596(-128 to 895)$$
$$0.813(Cr - 128) = 0.813[(0 - 128) to (1023 - 128)] = 0.813(-128 to 895)$$
$$0.392(Cb - 128) = 0.392[(0 - 128) to (1023 - 128)] = 0.392(-128 to 895)$$
$$2.017(Cb - 128) = 2.017[(0 - 128) to (1023 - 128)] = 2.017(-128 to 895)$$

Each of these terms is calculated for all the possible input values. The results can then be stored in a 16-bit wide, 1024-deep RAM. Five RAMs are used for the five terms. The address lines to the RAMs are the respective input signals that are used in each of the terms. The output of the RAM is the data stored in the location addressed by the input signals, Y’, Cr, and Cb. The output of the RAMs are added using an adder. The block diagram and the implementation results for this method are shown in Figure 2.

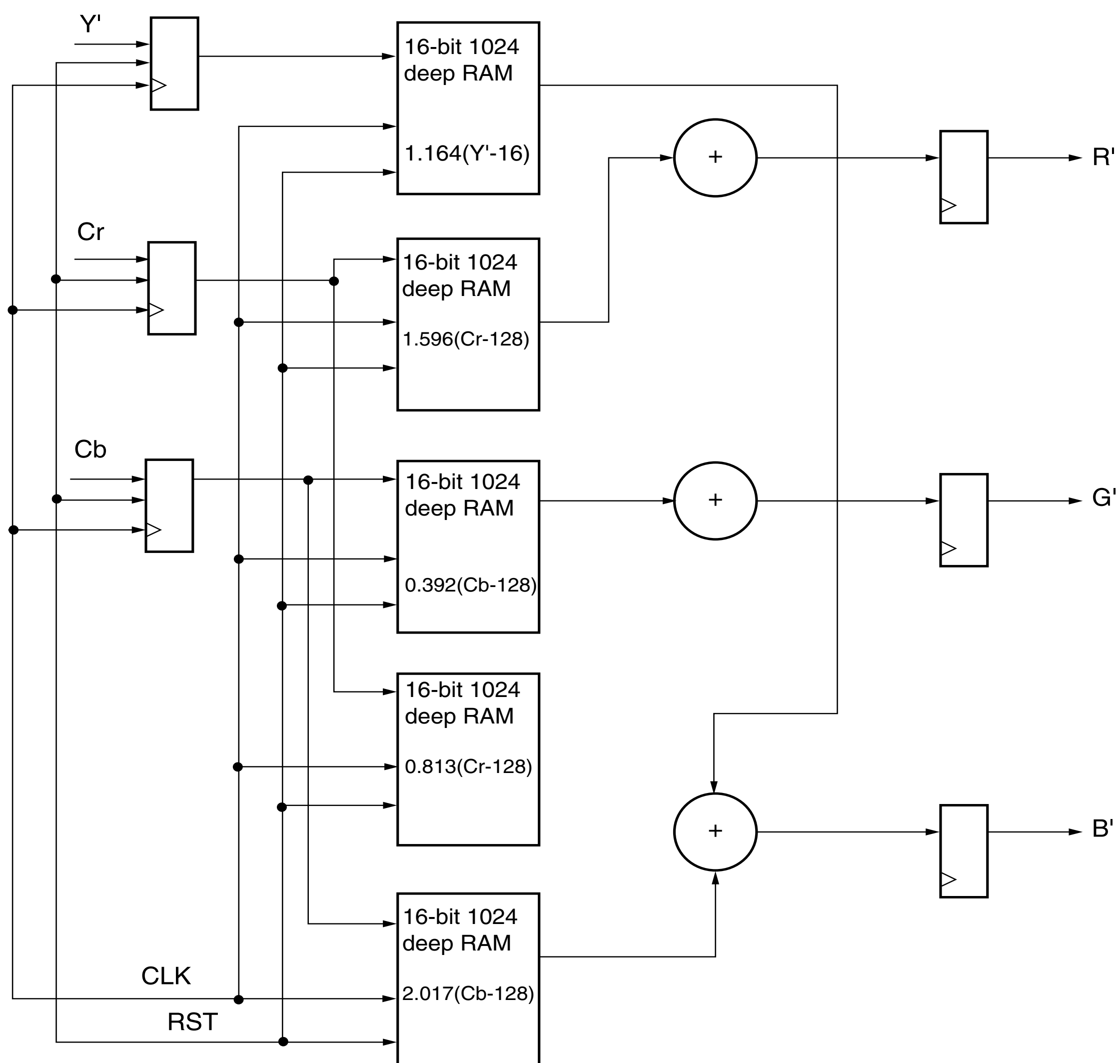


Figure 2: **Implementation Using RAM**

Implementation Results Using Embedded Multiplier in Virtex-II Device

The model with the instantiated block RAM was synthesized using Synplicity and the resulting EDIF file was placed and routed using Design Manager. A timing constraint of 5 ns was given to the place and route tool. The implementation results (push button) for the color space converter using the instantiated block RAM are as follows:

Notes:

1. See Verilog file, `ram_model.v.`

Table 2: **Block RAM Implementation Design Summary**

Device	LUTs	FFs	RAM	Ports	Performance
XC2V500-5 (slowest speed grade)	60	10	5	68	9 ns / 103 MHz (inputs and outputs registered

Implementation Using Embedded Multiplier (mult_model.*)

The block diagram for the implementation using embedded multiplier is as shown in Figure 3. A two's complement circuit is provided to take care of the negative results for (Y'–16), (Cr –128),and (Cb –128) values. The two's complement circuit can be omitted if the inputs are assumed to be in two's complement format.

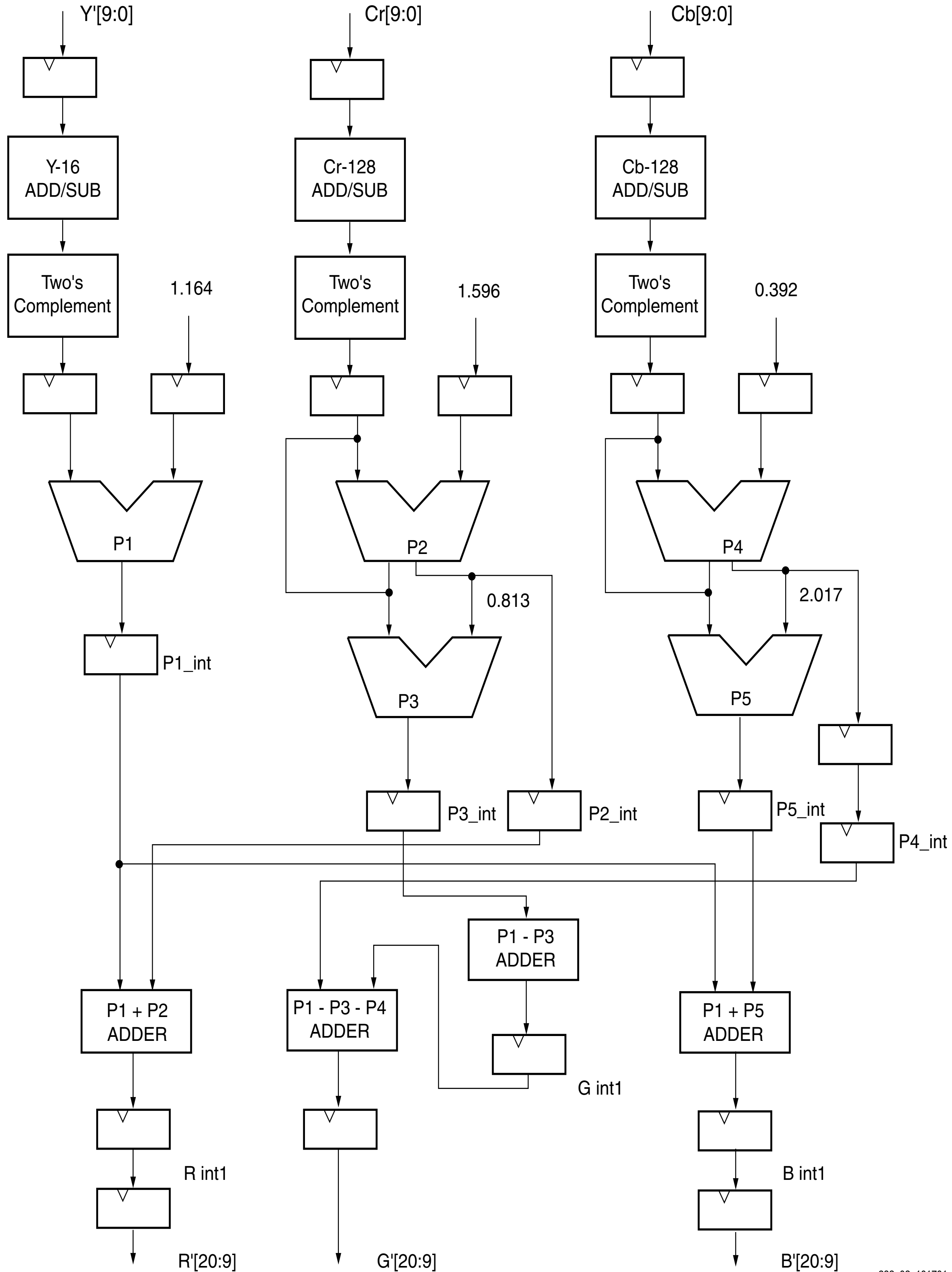


Figure 3: **Implementation Using Instantiated Multiplier**

Implementation Results Using Embedded Multiplier in Virtex-II Device

The model with the instantiated multiplier was synthesized using Synplicity and the resulting EDIF file was placed and routed using Design Manager. A timing constraint of 5 ns was given to the place and route tool. The implementation result (push button) for the color space converter using the instantiated multiplier is as follows:

Notes:

1. See Verilog file, `mult_model.v.`

Design Summary

Table 3: **Embedded Multiplier Implementation Design Summary**

Device	LUTs	FFs	Mult 18 x 18	Ports	Performance
XC2V500-5 (slowest speed grade)	131	177	5	68	8.9 ns / 111 MHz

Reference Design

The VHDL and Verilog reference designs for this application note are available on the Xilinx web site in a .zip file:

ftp://ftp.xilinx.com/pub/applications/xapp/xapp283.zip

Conclusion

The results of the synthesis and implementations demonstrate how the three examples trade off one math resource for another. The behavioral Verilog describing the conversion equations uses a resource available in Virtex, Virtex-E, and Virtex-II devices, known as “MULT_AND” to form the basis of the multiplies in the equations. No block RAM or embedded multipliers are consumed. In the second example, the math resource used is block RAM/ROM, again available in all Virtex families. Finally, the Virtex-II family now provides the most flexible math resource for DSP in the form of an embedded, high-speed, two’s complement multiplier.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/11/01	1.0	Initial Xilinx release
04/22/02	1.1	Updated Figure 1 and Figure 2. Changed implementation summaries with newer data. Updated to include Virtex-II Pro devices. Modified the 10-bit equation on page 2.