ELEX 7660: Digital System Design
Composite Video Encoder with SPI Interface

Matthew Knight, Cole Harkness

2017–04–17

Contents

Introduction

Letter from Matt Knight

At a party a couple years ago, a friend who worked in film approached me and told me about a niche market in film that required the syncing of frame rates of monitors to shutter rates of cameras. One of the most less controllable of monitors were CRT televisions as their frame rate could not be changed, unlike many digital monitors. While there were old converter devices that handled this, these were made by "weird dudes in their basements"i. These induviduals would now charge large sums of money for new devices as they did not want to make any more.

This conversation got me researching composite video signals and the NTSC standard. This research changed my goal to generating composite video digitally. The idea was that one could use an old analog television as a simple display for whatever project they chose. I first attempted to generate my own video source from a microcontroller, however I easily ran out of processing power for a black and white pixel display, let alone something with greyscale output or even colour.

And instead of turning to a bigger, faster processor, it made sense to move towards a hardware solution, and that is where FPGA's come in.

- Matt Knight

Composite Video Driver for ELEX 7660

This project is a video card that takes a 9-bit parallel interface to transfer video data, and outputs the analog signal required for composite video. The purpose of this device is to provide an easy to use, and efficient method to display an image or video on a CRT television. There is little practicality in this project since it is based on such an old technology, it was chosen more for a challenging goal that suited the premise for using FPGA's: software is not fast enough.

Our project is hosted on our github repository at **?**.

Background

The Composite Video Signal

Originally video was broadcast in black and white, only requiring one channel. In terms of driving the television itself, one needs only change DC voltage in the active video region of the signal to change luminosity levels.

Once colour televisions became a reality, video needed to be broadcast in colour as well. In order to also be backwards compatible with black and white televisions still in common use, colour was added to the video signal by adding an amplitude and phase modulated sinusoid. This allowed for only requiring one additional radio band for broadcast, keeping the now "composite" signal on a single wire, and black and white televisions would discard the colour components by passing the signal through a low-pass filter.

In the CRT television is an electron beam. In order to generate images, this beam sweeps from left to right and changes intensity. The brightness of the point at which the beam is directed to is proportional to the intensity of the electron beam. One sweep of the electron beam creates a single line, and it will do this several hundred times in order to generate a single frame on the television. In North America, televisions will have frame rates of 30 Hz **?**, **?**, **?**, **?**.

Input Output Machine

The simplest way to view this project is as an input output machine. The input being any device that could drive a 9-bit input and a clock. This input was basically, an image broken down into pixels and colours. Our machine would store the input signal and covert it to a composite video output. The composite video output is complicated and requires precise timing, and precise repetition at a high frequency. The purpose of this was to be able to take any device as an input we could convert it to a component video output.

Objectives

Main Objective

The main objective of this project is to generate a black and white composite video signal that will drive a Cathode Ray Tube (CRT) television. To accomplish this, we will develop a video card with a serial interface that can generate a composite video signal. We will be able to achieve a resolution of 240 x 320 pixels on the screen.

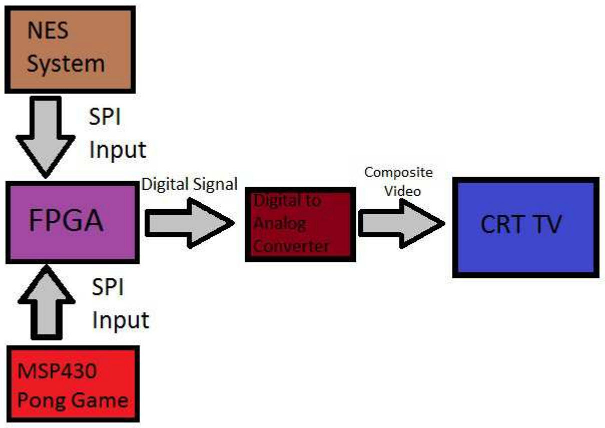

Figure 1: System Overview

Secondary Objective

Colour

The colour component of a composite signal is transmitted through the phase and angle modulation of a 3.58MHz carrier. In order to generate this signal we would need to employ a parallel DAC instead of our cheap, lopsided, but inexpensive resistor divider DAC. Another project within our class is to create a Ninstendo Entertainment System on an FPGA, and so we will base our colour output on the NES.

Hardware

To be able to complete this project we will need more hardware than what we currently have. We have a MSP430 microcontroller, the FPGA board, and a mini black and white TV. We need an 8-bit parallel, single channel DAC. We will use the AD9748 from Analog Devices Inc. which can be ordered from Digikey, and we will also need a breakout board for the 32-QFN package which can also be picked up from Digikey. Any hardware we may need, such as parts for filtering will be provided by us.

Implementation

8-bit DAC Hardware

A relatively high quality DAC is required in order to synthesize composite video. AD9748 was selected as it had a fast rise and fall time, parallel input, and up to 210 MSPS. The only challenge was that no breakout boards existed for this IC, so a general breakout board was ordered for the QFN-32 package, and with some help from Ed Casas we managed to reflow solder the IC successfully:

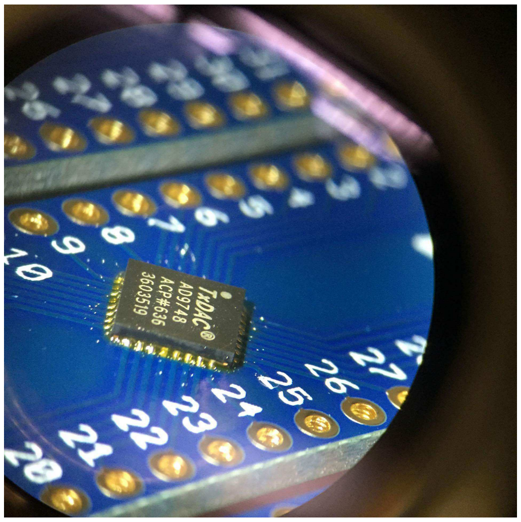

Figure 2: Surface Mounted DAC IC

As this DAC was going to be operated at 50 MSPS, the clock was put in differential mode, and a shielded twisted pair was used as the connection between the FPGA and the DAC board. This was to reduce any possible noise. additionally the analog power was given a separate regulator that recieved power from the external dual power supply. The clock and digital power was taken from the FPGA.

The dual supply is also used to power the video amplifier. This takes the differential current output of the DAC and converts it into the appropriate range for composite video (-1V to 1V). The output of the amplifier is connected to an RCA jack so that it can be easily connected to standard televisions. Standard value resistors were used for the video amplifier while a potentiometer was used to adjust the full scale range of the current output of the DAC.

In application we would not be using an external dual power supply but a switching regulator. This keeps the package size down while only requiring a single power supply. Additionally the switching power supply would provide some noise immunity from harmonics created by the 50 MHz clock.

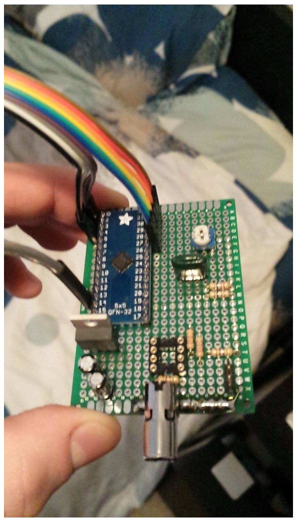
Figure 3: Prototype DAC Board

Composite Driver

Timing Control

The challenge with generating composite video is that it is a real-time signal. Timing of specific synchronizations is important to create an undistorted image. The timer module controls these timings using a lookup table, a counter, and some pointers.

Once a vertical blanking signal is needed, the timer is reset to the beginning. The basic workings of the timer module is that is will have a specific point in the table, or state. Every clock cycle the counter increments, and once the counter reaches the value for the next entry in the table, the state increments. For each entry in the table, there is the count value and the DAC value.

For the vertical blanking state, the DAC output value will be the Hsync and Blanking levels which get sent straight to the DAC. When in the horizontal line state, some of the outputs are Hsync and blanking, however there is the active video state which switches the DAC output to be driven by the line buffer. Additional to the clock counter is a pixel counter which increments every 8 clocks so that each pixel in a line is timed perfectly and correct pixel can be extracted from the line buffer.

Line Buffer/Block FIFO

In order to deal with different rates of writing to the DAC and incoming video data, a buffer that contains all the pixel information for one line was instantiated in the video generator. A "block" FIFO was used for this buffer. The idea behind this type of FIFO is that once it is full, it will not accept any more data until it is reset. Upon a reset, the old data is still available to the read side until it is overwritten. The video generator would also tell the SPI module which line it wanted, and so the buffer would get a burst of writes that would be the line that the video generator was requesting.

Colour Modulation

The colour modulation, which did not get to be tested used lookup tables to decode the colour numbers. A numerically controlled oscillator generated the phase-time signal for the 3.58 MHz carrier. Phase modulation was done at this stage since one only needed the addition operator in order to modulate this signal. Then the phase went into an 8-bit sine lookup table. This was done in two's compliment so that the dc offset was referenced to the middle of the sine wave and not the crest. This made generating the lookup tables far simpler. The sine wave was then amplitude modulated by a value that was assertained from the colour number using another lookup table. Finally the dc value representing luminosity was added to the sine wave and the signal was sent to the DAC to be converted.

SPI Module

The SPI module was designed to take a 9-bit input and a clock input. 8 of the 9 bits was to indicate colour. The last bit was for internal communication. Internal communication included setting individual pixels, resetting the frame and any other special features the input device could use. Our SPI module was designed this way so any device, slow or fast, could be used as an input.

Once the SPI module obtained an input, the module would stack two input pixels and store the input on an array 320 elements long. Each element on the array would store 16-bits or two pixels. This make for easy conversion to the SDRAM. The SDRAM stores 16 bits worth of data at a time.

RAM Interface

The RAM interface was not completed. Although to get this working we would needed to use NIOS 2 software to give us a file to be able to access the SDRAM. Once this Verilog file would be created we would have to modify it to fit our needs and properly implement it in our system.

The SDRAM was needed for our project to reach its max potential because the FPGA could not store enough data alone for a full frame. This is because we needed the FPGA to store 640x480 8-bit pixels. At first, we tried to quarter the resolution, 320x240, but this still required the use of the SDRAM. When we figured we didnt have enough time to complete the SDRAM module, we lowered the resolution again to 160x120. With this resolution, we were finally able to be stored an entire frame on the FPGA without using the SDRAM. We stored the data on the FPGA in the form if a 160x120 matrix with elements of 8-bit pixels.

Results

In order to get an operable demonstration of working composite video synthesis some of the modules had to be bypassed. The digital interface was complete and verified in simulation, however we did not have time to finish our video source, Pong on the MSP430, so the digital interface was replaced by a simple verilog module that fed the video generator a line that was white for the first half and black for the second half. When displayed, the entire screen would be half white and half black. Although not incredibly impressive, it did demonstrate that we met our objective of generating a composite video signal.

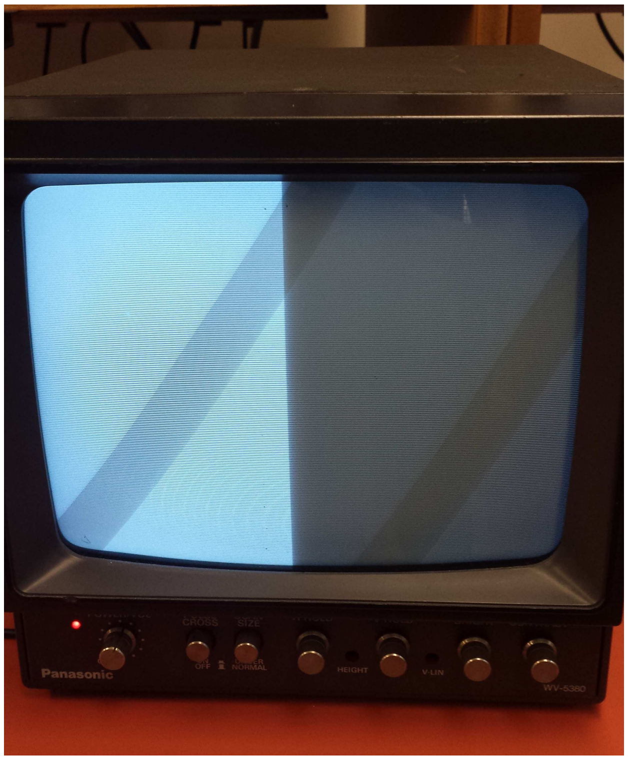

Figure 4: Demonstration Setup. Note: Grey diagonal bars are caused due to differences in the frame rate of the display and the shutter rate of the camera. The display is in fact half white and black.

Additionally, since we only had a black and white television, we were not able to test our colour generation capability, however simulations of the module output were successful.

Conclusion

This was a challenging project that was enjoyable and helped us understand HDL in greater detail. Not only did this project require understanding and implementing an already existing video transmission method, it required us to deal with high frequency components, precise timing, multiple device communication, and temporary data buffers (SDRAM or Large arrays).

Although we did not reach all our initial goals, this project was a success. I can say this because we were able to control the image displayed on the black and white TV. We would also be able to display this in 8-bit colour. Lastly, with just a day longer I would confidently say we would be able display an image from an input source.

If we were to change anything about our project we would have wanted to implement the SDRAM because then we wouldn't be limited by resolution. This would also let us easily expand our project to other video outputs.

Future Directions

Given more time, colour could easily be generated with this device. The colour palette could be improved so that 256 colour could be implemented as there was room in the colour encoding word size, Only the look up tables would need to be regenerated.

Also, later in the project it became clear that it would be advantageous for the composite video driver to use some sort of communication bus for use with a processor core. Using the NIOS II would expand the capabilities of the system so that it could multitask while dealing with the data coming into the module.

Additionally, the resolution could be increased to the standard 480x640 pixels, this would require a faster and still reliable clock.

If we used NIOS II we could have also implemented the SDRAM. This would give us the ability to store an 640x480 image and possible multiple images. This could be used to upload a few images to the devices then disconnecting the source or for creating simple animations.

Appendix

Module Listings

```
// Copyright (C) 1991-2016 Altera Corporation. All rights reserved.
// Your use of Altera Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Altera Program License
// Subscription Agreement, the Altera Quartus Prime License Agreement,
// the Altera MegaCore Function License Agreement, or other
// applicable license agreement, including, without limitation,
// that your use is for the sole purpose of programming logic
// devices manufactured by Altera and sold by Altera or its
// authorized distributors.  Please refer to the applicable
// agreement for further details.

module compositeDriver
(
        // System Clock
        input CLOCK_50,

        // DAC Interface
        output [7:0] dac,                                                   // DAC Data
        output dacClk, dacClk_n,                                          // DAC Diff Clock
        input [8:0] sdata,
        input sclk


);




        reg reset = 1'b0;
        wire write;
        wire [7:0] data_i;
        wire [7:0] line;
        wire ready;
        reg [8:0] i;
        reg [8:0] pixel;

        // Differential Clock signal to DAC
        assign dacClk = ~CLOCK_50;
        assign dacClk_n = ~dacClk;


        // Video Signal Generator Module
        videoGen v_0(
                .clk(CLOCK_50),
                .reset(reset),
                .write(write),
                .data_i(data_i),
                .line(line),
                .ready(ready),
                .dac(dac)
        );

        // SPI Controller
        SPI spi_0(
                .CLK(CLOCK_50),
                .reset(reset),
                .Next_Line(ready),
                .clk_SPI(sclk),
                .Row_Select(line),
                .Data(sdata),
                .Data_O(data_i),
                .Data_Valid(write)
        );

endmodule

// This Module Takes 8 bit input from SPI Signal and stores the data on temporary buffer
// Called Buffer. It will then send 16 bit data to the RAM.
// By William Harkness


module SPI (     input wire [8:0] Data,
                                input wire [7:0] Row_Select,
                                input wire clk_SPI, Next_Line, CLK, reset,
                                output reg Data_Valid,
                                output reg [7:0] Data_O);

        parameter PTR_Max_Row =  119;
        parameter PTR_Max_Col =  159;


        reg [7:0] Buffer [PTR_Max_Row:0] [PTR_Max_Col:0] ;              // 320 pixal = one lin


        reg [8:0] PTR_Read_Col, PTR_Write_Row, PTR_Write_Col;
        reg Full, Line_Flag;
        reg clk_SPI_next, Doubler;
        reg [8:0] Data_next;

                                                                                          // Po

        initial begin
                PTR_Read_Col <= 8'd0;
                PTR_Write_Col <= 8'd0;
                PTR_Write_Row <= 8'd0;
                Data_O <= 8'd0;
                Data_next <= 8'd0;
                Full = 1'b0;
                Line_Flag = 1'b0;
                Data_Valid = 1'b1;
                Doubler = 1'b0;
        end


        always @(posedge clk_SPI_next) begin

                Buffer[PTR_Write_Row][PTR_Write_Col] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd1][PTR_Write_Col] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd2][PTR_Write_Col] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd3][PTR_Write_Col] <= Data_next[7:0];

                Buffer[PTR_Write_Row][PTR_Write_Col + 8'd1] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd1][PTR_Write_Col + 8'd1] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd2][PTR_Write_Col + 8'd1] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd3][PTR_Write_Col + 8'd1] <= Data_next[7:0];

                Buffer[PTR_Write_Row][PTR_Write_Col  + 8'd2] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd1][PTR_Write_Col + 8'd2] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd2][PTR_Write_Col + 8'd2] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd3][PTR_Write_Col + 8'd2] <= Data_next[7:0];

                Buffer[PTR_Write_Row][PTR_Write_Col + 8'd3] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd1][PTR_Write_Col + 8'd3] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd2][PTR_Write_Col + 8'd3] <= Data_next[7:0];
                Buffer[PTR_Write_Row + 8'd3][PTR_Write_Col + 8'd3] <= Data_next[7:0];



                if (((PTR_Write_Col == PTR_Max_Col - 8'd3 ) && (PTR_Write_Row == PTR_Max_Row -


                else if ((PTR_Write_Col == PTR_Max_Col - 8'd3) || Data_next[8]) begin


                        PTR_Write_Col = 8'd0;

                        if (Data_next[8])
                                PTR_Write_Row = 8'd0;

                        else if (PTR_Write_Row == PTR_Max_Row - 8'd3)
                                PTR_Write_Row = PTR_Max_Row - 8'd3;

                        else
                                PTR_Write_Row = PTR_Write_Row + 8'd4;

                end
                else
                        PTR_Write_Col = PTR_Write_Col + 8'd4;

        end



        always @(*)  begin
```

```
            if  ((PTR_Write_Col == PTR_Max_Col - 8'd3) && (PTR_Write_Row == PTR_Max_Row - 8
                    Full = '1;

            if(Full)
                    Data_O <= Buffer[Row_Select[7:1]]  [PTR_Read_Col];

            else
                    Data_O <= 8'd0;

    end


    always @(posedge CLK) begin

            clk_SPI_next <= clk_SPI;
            Data_next <= Data;


            if(Next_Line && ~reset)
                    Line_Flag = 1'b1;

            if(Next_Line || Line_Flag || reset) begin                    // Video want

                    if( ~Doubler && ~reset)
                            Doubler = 1'b1;

                    else if ((PTR_Read_Col == PTR_Max_Col) || reset) begin
                            PTR_Read_Col = 8'd0;
                            Line_Flag = 0'b0;
                            Doubler = 1'b0;
                    end

                    else begin
                            PTR_Read_Col = PTR_Read_Col + 8'd1;
                            Doubler = 1'b0;
                    end
            end
    end

endmodule

// Video Signal Generator Module

// Author: Matthew Knight
// File: videoGen.v
// Date: 2017-04-05

module videoGen(
    input clk, reset, write,
    input [7:0] data_i,
    output ready,
    output [7:0] line,
    output [7:0] dac
);
```

```
        // wires to connect modules
        wire [8:0] pixel;
        wire [7:0] lum;
        wire phaseReset;
        reg bufReset;


        always @(negedge clk) begin
                if (pixel == 9'd330)
                        bufReset <= 1'b1;
                else
                        bufReset <= 1'b0;
        end

        // Timer Module for controlling sequences in the signal
        timer t_0(
                .clk(clk),
                .reset(reset),
                .phaseReset(phaseReset),
                .pixel(pixel),
                .video(lum),
                .dac(dac),
                .line(line)
        );

        // Line Buffer to store the next raster line in
        blockfifo lineBuffer(
                .clk(clk),
                .reset(reset | bufReset),
                .write(write),
                .ready(ready),
                .data_i(data_i),
                .readPtr(pixel),
                .data_o(lum)
        );



endmodule

// Timer Module

// Author: Matthew Knight
// File: timer.v
// Date: 2017-04-05

// This module is to control the timing of the syncing pulses in the composite
// video standard.

// Defines
`define CW 16                          // Counter Width
`define SYNC 8'd90
`define BLANK 8'd127
`define VIDEO 8'hFF
```

```
`define WHITE 8'd219
`define BLACK 8'd134

module timer(
        input clk, reset,
        input [7:0] video,
        output phaseReset,
        output [8:0] pixel,
        output reg [7:0] dac,
        output reg [7:0] line
);

    reg [(`CW-1):0] clockCount;          // counter variable
    reg [(`CW+7):0] vertical [0:57];
    reg [(`CW+7):0] horizontal [0:5];
    reg [6:0] stateCount;                // Counter for moving up states
    reg [(`CW+7):0] next;
        reg [7:0] videoState;
        reg state;
        wire [(`CW-1):0] pixelCount;

        assign pixelCount = clockCount - 'b111010110;
        assign pixel = pixelCount[11:3];
        assign phaseReset = 'b0;

        // Switching between horizontal scanning and vertical blanking
    always @(*) begin
            if (state) begin
                    next = horizontal[stateCount+1'b1];
                    videoState = horizontal[stateCount];
            end else begin
                    next = vertical[stateCount+1'b1];
                    videoState = vertical[stateCount];
            end

            // Mux for the dac output
            case(videoState)
                    `SYNC : dac = `SYNC;
                    `BLANK : dac = `BLANK;
                    `VIDEO : dac = video + `BLACK ;
                    default : dac = `BLANK;
            endcase
    end

    always @(posedge clk) begin
            if (reset) begin
                    clockCount <= 'b0;
                    stateCount <= 'b0;
                    state <= 'b0;
            end else begin
                    if (state) begin
                            // Horizontal
                            if (clockCount >= 'hC6B) begin
                                    clockCount <= 'b0;
```

```
                                        stateCount <= 'b0;
                                        line <= line + 1'b1;

                                        if (line >= 'd242)
                                                state <= 1'b0;

                                end else begin
                                        clockCount <= clockCount + 1'b1;
                                end
                                if (clockCount >= next[(`CW+7):8]) begin
                                        stateCount <= stateCount + 1'b1;
                                end
                        end else begin
                                // Vertical
                                if (clockCount >= 'hF86F) begin
                                        clockCount <= 'b0;
                                        stateCount <= 'b0;
                                        state <=1'b1;
                                        line <= 1'b0;
                                end else begin
                                        clockCount <= clockCount + 1'b1;
                                end
                                if (clockCount >= next[(`CW+7):8]) begin
                                        stateCount <= stateCount + 1'b1;
                                end
                        end
                end
end

    // Values for signal generation
initial begin
                state = 'b0;

                // Pre-Equalizing Pulses
                vertical[0] = 'h0000 + `SYNC;
                vertical[1] = 'h7200 + `BLANK;
                vertical[2] = 'h63500 + `SYNC;
                vertical[3] = 'h6A800 + `BLANK;

                vertical[4] = 'hC6C00 + `SYNC;
                vertical[5] = 'hCDE00 + `BLANK;
                vertical[6] = 'h12A100 + `SYNC;
                vertical[7] = 'h131400 + `BLANK;

                vertical[8] = 'h18D800 + `SYNC;
                vertical[9] = 'h194A00 + `BLANK;
                vertical[10] = 'h1F0D00 + `SYNC;
                vertical[11] = 'h1F8000 + `BLANK;

                // Vertical Sync Pulses
                vertical[12] = 'h254400 + `SYNC;
                vertical[13] = 'h2A8F00 + `BLANK;
                vertical[14] = 'h2B7900 + `SYNC;
                vertical[15] = 'h30C400 + `BLANK;
```

```
vertical[16] = 'h31B000 + `SYNC;
vertical[17] = 'h36FB00 + `BLANK;
vertical[18] = 'h37E500 + `SYNC;
vertical[19] = 'h3D3000 + `BLANK;

vertical[20] = 'h3E1C00 + `SYNC;
vertical[21] = 'h436700 + `BLANK;
vertical[22] = 'h445100 + `SYNC;
vertical[23] = 'h499C00 + `BLANK;

// Posy-Equalizing Pulses
vertical[24] = 'h4A8800 + `SYNC;
vertical[25] = 'h4AFA00 + `BLANK;
vertical[26] = 'h50BD00 + `SYNC;
vertical[27] = 'h512F00 + `BLANK;

vertical[28] = 'h56F400 + `SYNC;
vertical[29] = 'h576600 + `BLANK;
vertical[30] = 'h5D2900 + `SYNC;
vertical[31] = 'h5D9B00 + `BLANK;

vertical[32] = 'h636000 + `SYNC;
vertical[33] = 'h63D200 + `BLANK;
vertical[34] = 'h699500 + `SYNC;
vertical[35] = 'h6A0700 + `BLANK;

// Blank lines
vertical[36] = 'h6FCC00 + `SYNC;
vertical[37] = 'h70B700 + `BLANK;
vertical[38] = 'h7C3800 + `SYNC;
vertical[39] = 'h7D2300 + `BLANK;
vertical[40] = 'h88A400 + `SYNC;
vertical[41] = 'h898F00 + `BLANK;
vertical[42] = 'h951000 + `SYNC;
vertical[43] = 'h95FB00 + `BLANK;
vertical[44] = 'hA17C00 + `SYNC;
vertical[45] = 'hA26700 + `BLANK;
vertical[46] = 'hADE800 + `SYNC;
vertical[47] = 'hAED300 + `BLANK;
vertical[48] = 'hBA5400 + `SYNC;
vertical[49] = 'hBB3F00 + `BLANK;
vertical[50] = 'hC6C000 + `SYNC;
vertical[51] = 'hC7AB00 + `BLANK;
vertical[52] = 'hD32C00 + `SYNC;
vertical[53] = 'hD41700 + `BLANK;
vertical[54] = 'hDF9800 + `SYNC;
vertical[55] = 'hE08300 + `BLANK;
vertical[56] = 'hEC0400 + `SYNC;
vertical[57] = 'hECEF00 + `BLANK;

// Horizintal line timing
horizontal[0] = 'h0000 + `SYNC;
horizontal[1] = 'hEA00 + `BLANK;
```

```
                horizontal[2] = 'h10800 + `BLANK;
                horizontal[3] = 'h18500 + `BLANK;
                horizontal[4] = 'h1D500 + `VIDEO;
                horizontal[5] = 'hC1B00 + `BLANK;
    end

endmodule

// Block FIFO Verilog Module

// Author: Matthew Knight
// Date: 2017-03-21

// The Block FIFO has an output signal to signal when full. When full the module
// does not accept writes to memory, but allows for reads. The FIFO only becomes
// empty when it is reset.

module blockfifo #(
    parameter len = 320,
    parameter wid = 8,
parameter addrWid = 9
)
(
    input clk, reset, write,
    output reg ready,

    input [(wid-1):0] data_i,
    input [(addrWid-1):0] readPtr,
    output reg [(wid-1):0] data_o
);

    reg [(addrWid-1):0] writePtr;

    reg [(wid-1):0] ram [0:(len-1)];

    // Combinational
    always @(*) begin

        // Determine if buffer is full
        if (writePtr == len)
            ready = 'b0;
        else
            ready = 1'b1;

        if (readPtr < len)
            data_o = ram[readPtr];
        else
            data_o = 'b0;

    end

    // Sequential logic
    always @(posedge clk) begin
        if (reset)
            writePtr <= 'b0;
        else begin
            if (write & ready) begin
                ram[writePtr] <= data_i;
                writePtr <= writePtr + 1'b1;
            end
        end
    end

endmodule

// Colour Decoder Verilog Module

// Author: Matthew Knight
// Date: 2017-02-27

// This module takes a colour code adhering to the NES colour palette (64
// colours), a clk, and asynchronous reset, and outputs video data to be sent to
// the DAC for the colour burst or active video.

`include "../source/nco.v"
`include "../source/synthesizer.v"

module colourDecoder(
    input clk, reset,
    input [5:0] colourNum,
    output [7:0] video
);

    wire [7:0] phase;

    // Frequency control word for a 3579545 Hz output and a 50Mhz clock
    reg [23:0] fcw = 24'd1201096;

    // Instantiation of NCO
    nco #(24) osc(
        .clk(clk),
        .reset(reset),
        .fcw(fcw), .out(phase)
    );

    // Instantiation of colour synthesizer
    synthesizer s_0(
        .clk(clk),
        .reset(reset),
        .colourNum(colourNum),
        .phase(phase),
        .video(video)
    );

endmodule

// Numerically Controlled Oscillator Verilog Module

// Author: Matthew Knight
// Date: 2017-02-21
// N is the width of the frequency control word and
// M is the width of the truncated output

module nco
#(parameter N = 16, M = 8)
(
    input clk, reset,
    input [(N-1):0] fcw,
    output [(M-1):0] out
);

    // count registers
    reg [(N-1):0] count, count_next;

    assign out = count[N:(N-M)];

    always @(*)
        count_next = count + fcw;

    always @(posedge clk or posedge reset)
        if ( reset) begin
            count <= 1'b0;
        end else begin
            count <= count_next;
        end

endmodule

// Look-up Table Verilog Module

// Author: Matthew Knight
// Date: 2017-02-25

// This is a parameterized model for creating Look-up Tables.

module lut #(
    parameter L = 256,                  // Length of the table
    parameter W = 8,                    // Width of each element
    parameter file = ""                 // File to source rom data
)
(
    input clk, reset,                   // Clock input
    input [(addrWid-1):0] addr,         // Address for element selection
    output reg [(W-1):0] data           // Table output data
);

    // Address width
    parameter addrWid = $clog2(L);

    // Instantiation of ROM
    reg [(W-1):0] rom [0:(L-1)];

    // Data for lookup table
    initial
```

```
        $readmemb(file, rom);

    // data is updated on the positive edge
    always @(posedge clk or posedge reset) begin
        if (reset)
            data = 'b0;
        else begin
            if (addr < L)
                data <= rom[addr];
            else
                data <= 'b0;
        end
    end

endmodule

// Delay Verilog Module

// Author: Matthew Knight
// Date: 2017-02-26

// This module is simply a D Flip-flop for the purpose of delaying a signal for
// a single clock cycle.

module delay #(
    parameter W = 8                     // Width of the bus
)
(
    input clk, reset,                   // Clock input
    input [(W-1):0] D,                  // Data input
    output reg [(W-1):0] Q              // Delayed output
);

    always @(posedge clk or posedge reset)
        if (reset)
            Q <= 'b0;
        else
            Q <= D;

endmodule
```