Advanced Digital Design 1 (EEET2162) Laboratory Project

Oliver Patterson (S3723206)
Kaizhe Huang (S3686152)

Laboratory Demonstrator:
Genevieve Fahey

Date:
07/06/2022

RMIT School of
Engineering

Executive Summary

This project uses the DE-10 Nano development board from Terasic Inc and uses the FPGA component to configure the integrated ADV7513 HDMI controller to repeatedly display two frames of the iconic Never Gonna Give You Up music video by Rick Astley to a monitor via an HDMI to DVI cable.

Contents

1 Introduction

In this project, an FPGA design is created for the DE-10 Nano development board to be capable of displaying custom images to an HDMI monitor. The design is developed with Quartus Prime and written in Verilog HDL.

The initial goal of the project was to allow the FPGA to receive video data from a controller software running on the hard processor, and forwards such video to an HDMI monitor. However, due to time constraint and technical difficulty, the goal of the project is reduced to only displaying images hardcoded within the FPGA.

2 Literature Search

2.1 DE-10 Nano

The DE-10 Nano development board is based a Cyclone V SoC FPGA. Relavant components include:

- A Field-Programmable Gate Array (FPGA), this is the main focus of this project.
- A Hard Processor System (HPS) based on ARM Cortex-A9 architecture.
- An ADV7513 HDMI transmittor IC.
- A built-in USB-Blaster programmer for the FPGA.
- A Micro-SD card slot, allowing an OS to be loaded to the HPS.

2.2 ADV7513

This HDMI transmittor IC simplifies the logic required to output a valid HDMI video signal. The IC, configurable via I^{2}C, supports vairous HDMI resolutions and colour formats. When correctly configured the ADV7513 receives a video signal from the FPGA and outputs it through an HDMI connector, where it is interpreted by an HDMI monitor.

The input signals are:

- Colour data (D[23:0])
- Data enable (DE)
- Hsync
- VSync
- Clock

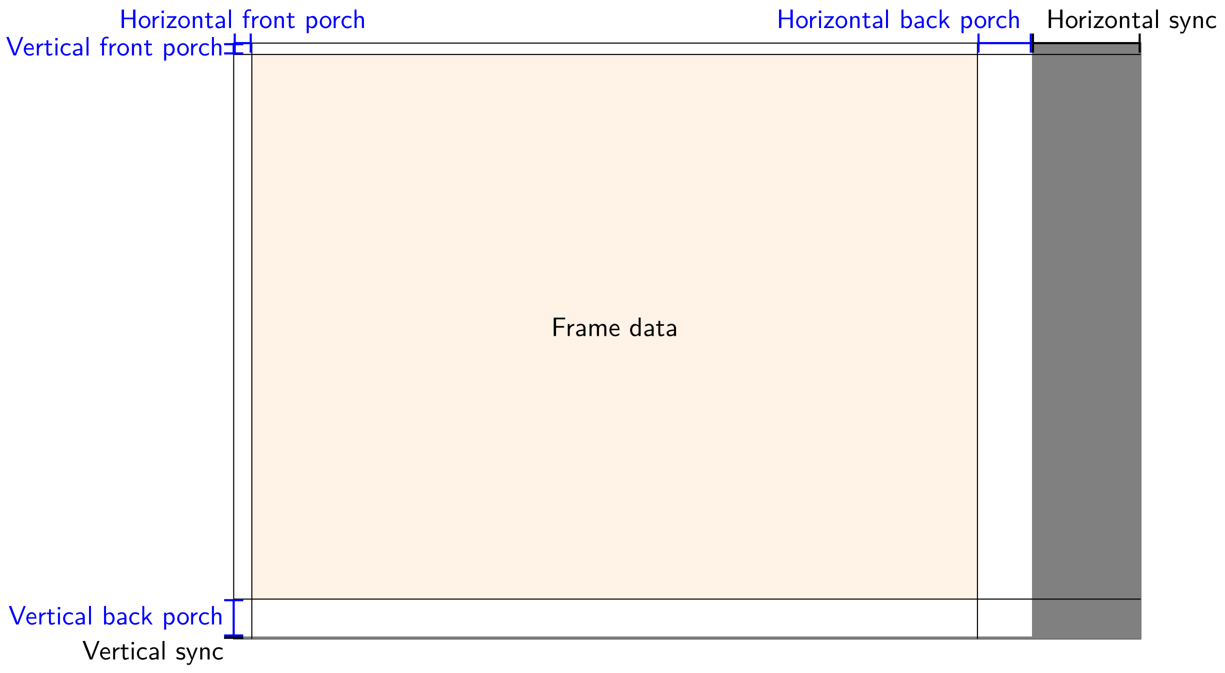


Figure 1: Video signal timing graph

Each pixel is sent left-to-right, by scanlines, top-to-bottom, in each clock cycle. Figure 1 shows the timings required for one frame. Each frame starts in a vertical front porch for the first several scanlines, where no data is sent. Then scalines containing frame data is sent. Followed by a vertical back porch, where no data is sent, and a vertical sync period, where a Vsync signal is asserted low. Each scanline starts in a horizotal front porch, where no data is sent, then data corresponding to this scanline is sent. Followed by a horizontal back porch, where no data is sent, and a horizontal sync period, where the Hsync signal is asserted low.

Outside of the front and back porches and syncing periods, DE is asserted high, and D[23:0] will contain the colour of the current pixel. D[23:0] may be in many colour formats, depending on the configurations made via I^{2}C, but the simpliest format is RGB, where each of the red, blue and green colour components are encoded in 8-bit unsigned integers.

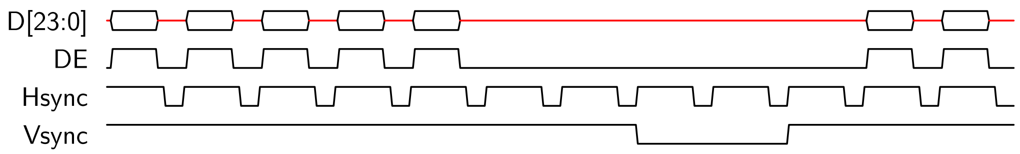


Figure 2: Video signal timing waveform

Figure 2 shows an example of a small portion of the video signal. The example contains 5 scanlines, in between which are the horizontal back porches, horizontal sync periods, and horizontal front porches. The signal then enters a vertical back porch, a vertical sync period, and a vertical front porch, before start sending the next frame.

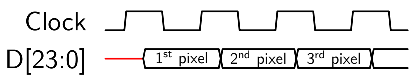


Figure 3: Video clock and data signal [1, p. 26]

The video clock signal marks the time at which the pixel values are sampled, its required relationship with data is shown on Figure 3, i.e. There needs to be one clock cycle per pixel, and data needs to be held stable while clock transitions high-to-low-to-high.

For this project, a video mode of 640×480 @ 60 Hz is used, as it is simple and supported by all monitors that support 60Hz video sync [2]. This mode requires the video signal timing periods described in Table 1. The frequency of the video clock can also be inferred from this information, as the total number of pixels for each frame is $800 \times 525 = 420000$, therefore $420000 \times 60 = 25200000$ pixels per second, translating to a 25.2 MHz clock.

Horizontal	
Front porch	16
Sync width	96
Back porch	48
Active pixels	640
Total pixels	800

Vertical	
Front porch	10
Sync width	2
Back porch	33
Active pixels	480
Total pixels	525

Table 1: Display timings for 640×480 @ 60 Hz

Analog Devices [3] describes various registers that can be set to achieve any desired configuration. However, due to its length and complexity, it was deemed easier to reference an example design provided by Terasic [4]. Their HDMI_TX example uses the same colour format as this project, and was confirmed to be functional. Table 2 lists the registers the example code sets, along with their comments. These registers also include configuration of sound for the ADV7513, which remained unused in this project, but the configuration steps were kept in for easier project re-use in any further work.

Register Address	Register Value	Comment
'h98	'h03	Must be set to 0x03 for proper operation
'h01	'h00	Set 'N' value at 6144
'h02	'h18	Set 'N' value at 6144
'h03	'h00	Set 'N' value at 6144
'h14	'h70	Set Ch count in the channel status to 8
'h15	'h20	Input 444 (RGB or YCrCb) with Separate Syncs, 48kHz fs
'h16	'h30	Output format 444, 24-bit input
'h18	'h46	Disable CSC
'h40	'h80	General control packet enable
'h41	'h10	Power down control
'h49	'ha8	Set dither mode - 12-to-10 bit
'h55	'h10	Set RGB in AVI infoframe
'h56	'h08	Set active format aspect
'h96	'hf6	Set interrup
'h73	'h07	Info frame Ch count to 8
'h76	'h1f	Set speaker allocation for 8 channels
'h98	'h03	Must be set to 0x03 for proper operation
'h99	'h02	Must be set to Default Value
'h9a	'he0	Must be set to 0b1110000
'h9c	'h30	PLL filter R1 value
'h9d	'h61	Set clock divide
'ha2	'ha4	Must be set to 0xA4 for proper operation
'ha3	'ha4	Must be set to 0xA4 for proper operation
'ha5	'h04	Must be set to Default Value
'hab	'h40	Must be set to Default Value
'haf	'h14	Select HDMI mode
'hba	'h60	No clock delay
'hd1	'hff	Must be set to Default Value
'hde	'h10	Must be set to Default for proper operation
'he4	'h60	Must be set to Default Value
'hfa	'h7d	Nbr of times to look for good phase

Table 2: ADV7513 registers needed to be set

2.3 Running Linux on the DE-10 Nano

Intel [5] provides an image that can be flashed onto a Micro-SD card. Powering-on the DE-10 Nano while the Micro-SD card is insert boots a Linux distribution, Angström Linux, on the HPS.

The boot process is facilitated by U-Boot. U-Boot may configure the FPGA, where it can later be accessed by device drivers on Linux, and then loads the Linux kernel from the SD card. By default, U-Boot configures the FPGA to be a graphical interface for Linux. This can be modified so that the FPGA is loaded with a custom firmware.

2.4 Never Gonna Give You Up

Never Gonna Give You Up is a popular music created by British musician Rick Astley [6]. Its music video has received increasing popularity as the practice of "RickRoll-ing", playing the music video in front of unaware individuals, has become common-place. Therefore it was selected as the video to play for this project. Due to time constrains, however, the scope was reduced to only display a selection of iconic images within the music video.

3 Technical Work and Results

3.1 Overview

Appendices 6.1 to 6.8 list all Verilog code used to implement the FPGA design. Among them, the top-level entity (TLE) contains plumbing for the other modules, depicted in Figure 4.

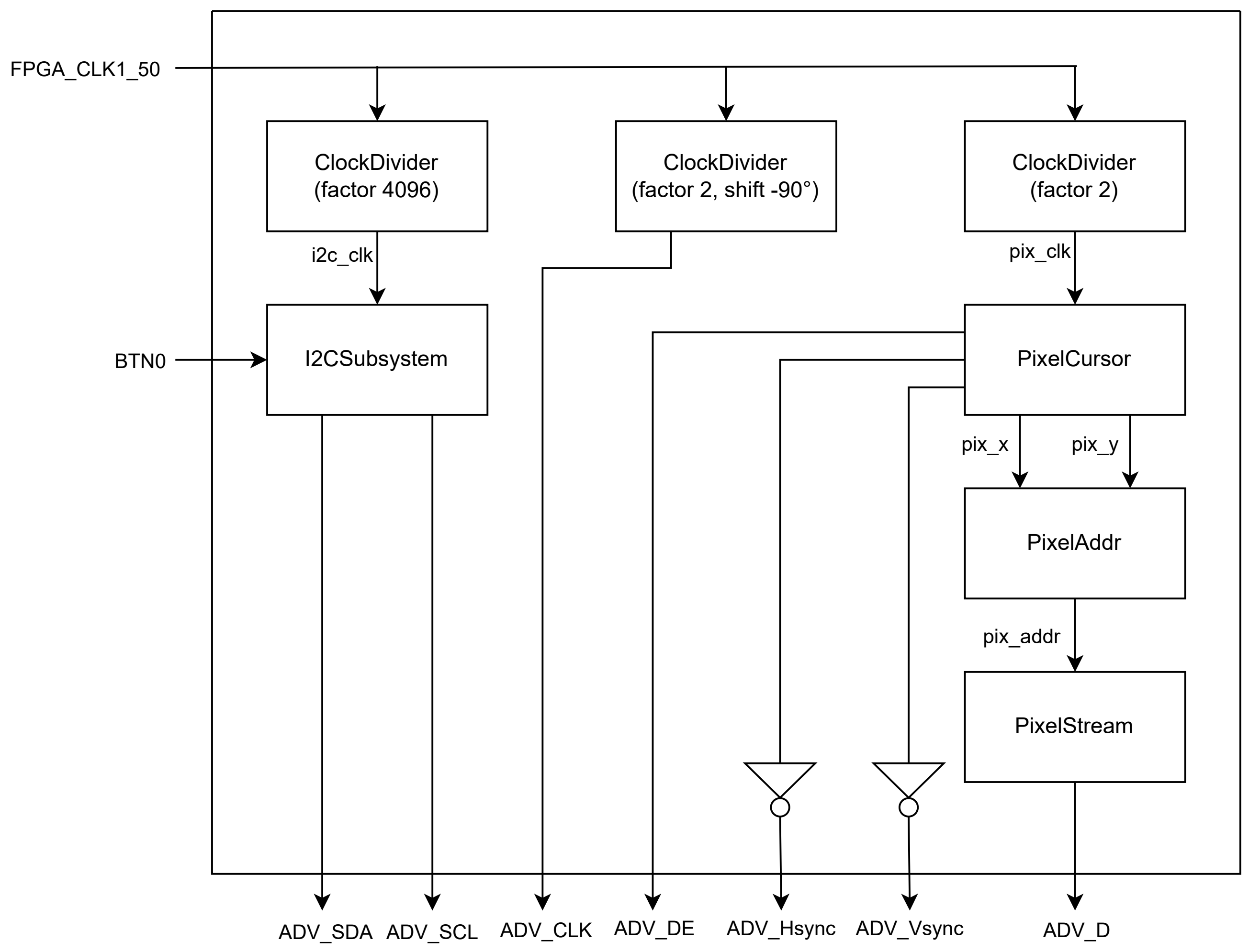


Figure 4: Overview of all modules

The input FPGA_CLK1_50 is connected to the built-in 50 MHz clock of the FPGA; BTN0 is connected to 1 of the 2 tactile buttons on the DE-10 Nano, it is debounced by external circuits on the board. The outputs named ADV_* all connect to the ADV7513 transmitter IC. Internal signals are denoted in lower case.

3.2 ClockDividerPow2

Clock dividers are needed to reduce the frequency of the 50 MHz built-in FPGA clock. Specifically, there are 3 slower clocks needed:

i2c_clk The I^2Cinterface of ADV7513 does not function with a high frequency, therefore the clock is divided by 4096 before being used by the I^2Ccontroller.

ADV_CLK As established in Section 2.2, the ADV7513 transmitter IC requires a 25.2 MHz clock signal. This can be approximated by dividing FPGA_CLK1_50 by 2 to obtain a 25 MHz clock.

pix_clk As shown in Figure 3, ADV_CLK must not be the same as the clock that updates D[23:0], instead, ADV_CLK needs to be 90° earliar in phase than this clock.

The module ClockDividerPow2 divides an input clock by a factor of 2^n, where n is a parameter. This is implemented by an n-bit counter that increments by 1 at each rising clock edge. The MSB of the counter is treated as the output clock, whose frequency will then be

$$f_{\text{out}} = \frac{f_{\text{in}}}{2^n}$$

The phase shift between ADV_CLK and pix_clk are created by inverting the FPGA_CLK1_50 before the ADV_CLK clock dividerm but not inverting the other. The result is illustrated in Figure 5.

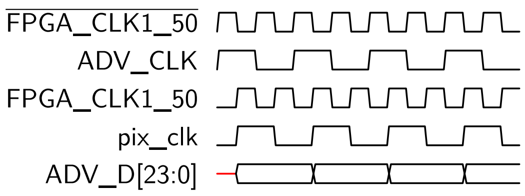
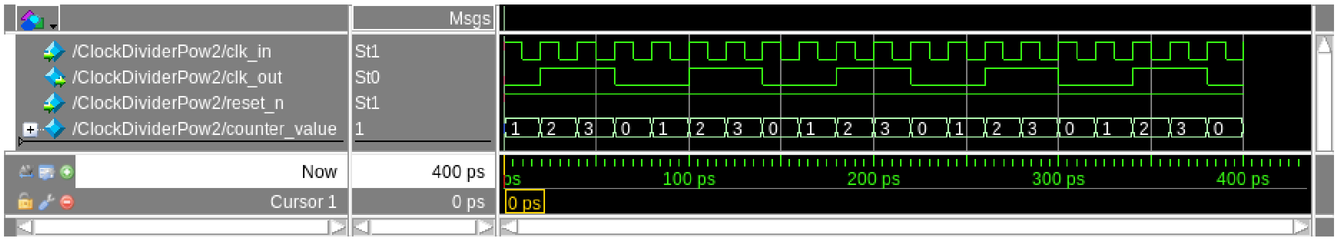


Figure 5: Creating phase shift between divided clocks

Figure 6 shows a ModelSim simulation of this module with a power of 2, thus division by 4.



Msgs
/ClockDividerPow2/clk_in St1
/ClockDividerPow2/clk_out St0
/ClockDividerPow2/reset_n St1
/ClockDividerPow2/counter_value 1
1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0
Now 400 ps
Cursor 1 0 ps
ps 100 ps 200 ps 300 ps 400 ps
0 ps

Figure 6: ClockDividerPow2 simulated in ModelSim, $(n = 2)$

The clock divider originally divided clocks by an arbitrary factor, where the output is produced by comparing the counter content to half of the dividing factor. However, doing so caused strange behaviour in the resultant signals when driven at a high clockrate, thus the design is simplified to only divide by a power of 2, and renamed from ClockDivider to ClockDividerPow2.

ADV_CLK and pix_clk were originally designed to run at the specified frequency in the ADV7513 Datasheet of 25.2 MHz, which was created by a PLL IP Block. This caused timing issues with the HDMI monitor when connected. A simplier 25 MHz clock appears to be functional.

3.3 PixelCursor

Two counters are used to generated the video control Signals of DE, Vsync and Hsync, as specified in Figure 1 and Table 1. The counter hcount stores the horizontal coordinate of the pixel being drawn, increasing on every rising edge of pix_clk, and its possible values can be defined by $c_h \in \mathbb{Z} \cap [0, 799]$. The counter vcount stores the vertical coordinate of the pixel being drawn, increasing whenever hcount overflows from 799 to 0, and its possible values can be defined by $c_v \in \mathbb{Z} \cap [0, 524]$. For simplicity, the $(0, 0)$ position is defined to be at the top-left corner of active frame data, after the horizontal and vertical front porches (Figure 7). This effectively moves the front porches to be positioned after the sync periods, from the perspective of the signal receiver, no difference is made, but from within the FPGA design, this decision has the advantage of hcount and vcount directly corresponding to the coordinates of the actual pixel being displayed, when video is active ($c_h < 640$ and $c_v < 480$). Therefore, their values can be directly output as pix_x and pix_y, which are received by PixelAddr and PixelStream to decide what pixel colours to draw.

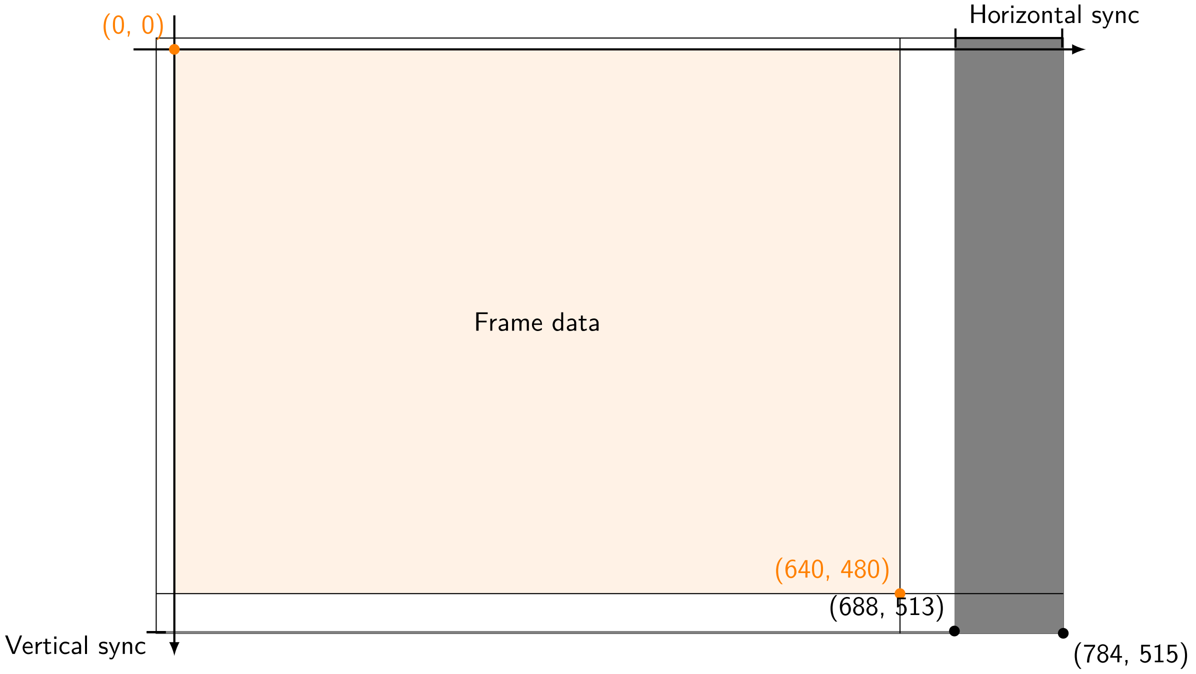


Figure 7: Coordinate system established by PixelCursor

The control signals can then be generated by comparing the counters to some constants:

$$\text{DE} = \begin{cases} 1 & c_h < 640,\ c_v < 480 \\ 0 & \text{otherwise} \end{cases} \quad \text{Hsync} = \begin{cases} 1 & 688 \leq c_h < 784 \\ 0 & \text{otherwise} \end{cases} \quad \text{Vsync} = \begin{cases} 1 & 513 \leq c_v < 515 \\ 0 & \text{otherwise} \end{cases}$$

Figure 8 shows a ModelSim simulation of this module in 2 scales, to showcase Hsync and Vsync.

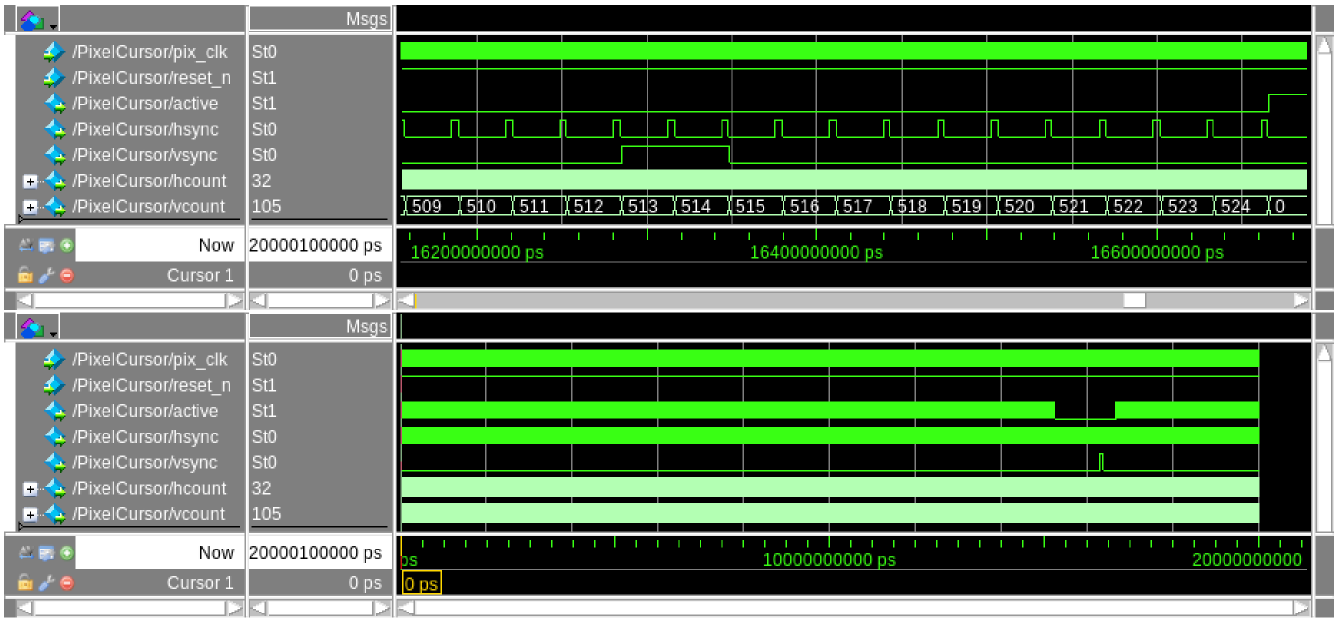


Figure 8: PixelCursotr simulated in ModelSim

3.4 PixelAddr and PixelStream

Astley [6] is cropped and scaled to a resolution of 160×120, each frame of the video is extracted as 19200 raw pixel values, in 24-bit RGB format which can be directly output to the ADV7513. Initially, a raw binary file containing all frames in 10 frames per second was planned to be stored onto a Micro-SD card. The controller software would copy each frame of the video into RAM consecutively every 0.1 s, where it would be read by the FPGA and sent to the ADV7513. However, as the project scope was reduced, video data needed to be stored on the FPGA itself. Since the memory available on the FPGA is severely limited, only two frames fit into the memory.

PixelAddr works as an address decoder for the pixel data based on the `xpos` and `ypos` inputs. This is done with the formula

$$\text{Address} = \frac{x_{\text{pos}}}{4} \cdot \frac{y_{\text{pos}}}{4} + \text{frame} \cdot 19200$$

with frame being the frame number starting at zero, as we only have two frames, and two frames switching at 60 Hz would be a horrible sight to behold the frames were only incremented if a counter was above 32, making the frames switch at just under 2 Hz. This creates an address which references the pixels by number from left to right in each scanline. This is then fed into the PixelStream module to extract the pixel data.

The 2 frames of data is encoded into `.mem` format, a plain text format where each line specifies 1 word of data in hexadecimal, in this instance, 24 bits. PixelStream takes a memory address calculated by PixelAddr, and outputs the corresponding pixel value from this block of memory, within the same clock cycle.

The downsides to this approach are severely increased compilation time and logical element usage. An attempt at mitigating these issues was made, by using a ROM IP block. An additional advantage to that approach is more available memory space, which can accomodate more than 5 frames. However, a ROM IP block requires 3 clock cycles for its output to respond to the input address, making it unfit for the current design.

3.5 I2CSubsystem

This module is composed of 2 submodules, I2CController and I2CDataFeed.

I2CController receives an Op[1:0] signal specifying what to do to the bus (Table 3), and a byte of data (Data[7:0]) to be sent. The module maintains a state machine with 35 states, with each state corresponding to some output SDA and SCL (Figure 9). State transitions occur on every rising clock edge, some of which depends on the value of Op[1:0] (Figure 10).

Op[1:0]	Name	Description
00	STOP	Stop the I^2Ctransaction, release the bus, or keep it released.
01	START	Start a I^2Ctransaction, or repeat start the transaction without releasing the bus.
10	CONTINUE	Continue sending the the next byte within the ongoing transaction.
11	RESTART	Stop and start the I^2Ctransaction, releasing the bus in the process.

Table 3: Meanings of the Op[1:0] signal in I2CSubsystem

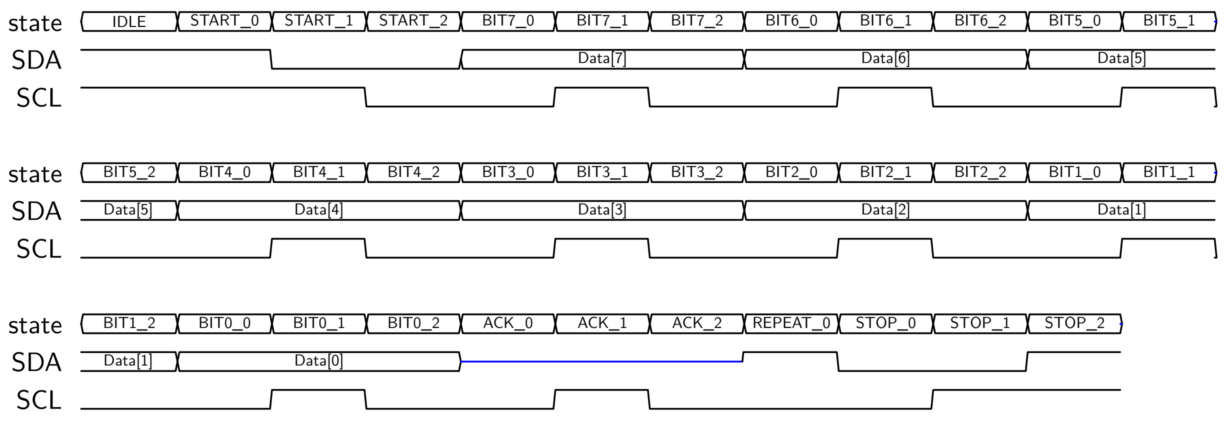


Figure 9: Output of I2CController corresponding to each state

Additionally, I2CController outputs an output Completed, which is high only when the state is ACK_1, indicating the transfer of a byte is completed.

I2CDataFeed provides the Op[1:0] and Data[7:0] signals that I2CController needs, they are also determined by a state machine. On reset, its state is 0, where Op[1:0] is STOP, thus keeping the I2CController in idle. When an Update signal is received, it progresses to the next state, looping back to state 0 after the last state. This state machine contains 94 states, with all states except 0 containing some non-STOP Op[1:0] and a byte of Data[7:0], they list the sequence of data that that needs to be sent in order to configure the ADV7513 IC.

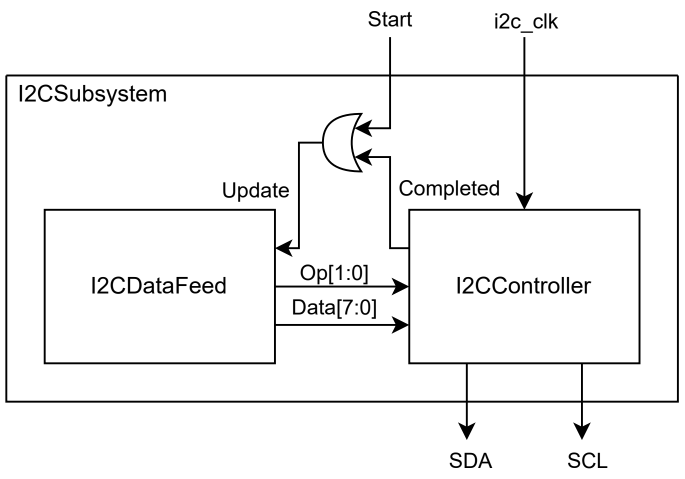


Figure 11: Block diagram of I2CSubsystem

The Update input of I2CDataFeed is connected to the Completed output, forming a feedback loop (Figure 11): When a byte transfer is completed, I2CController notifies I2CDataFeed to update its Op[1:0] and Data[7:0], then I2CController carries out the next operation. When I2CDataFeed reaches state 0, the loop stops. This also means that both state machines will stay idle upon reset. An external stimulus from a button is also connected to Update. When I2CDataFeed is in state 0 and BTN0 on the DE-10 Nano is pressed, it enters state 1, instructing I2CController to start the feedback loop.

For each register that needs to be set within the ADV7513, 5 steps are involved:

1. Start an I^{2}C transaction.
2. Send the slave address of the ADV7513, which is 'h72 on the DE-10 Nano.
3. Send the register address.
4. Send the value to be written to the register.
5. Stop the I^{2}C transaction.

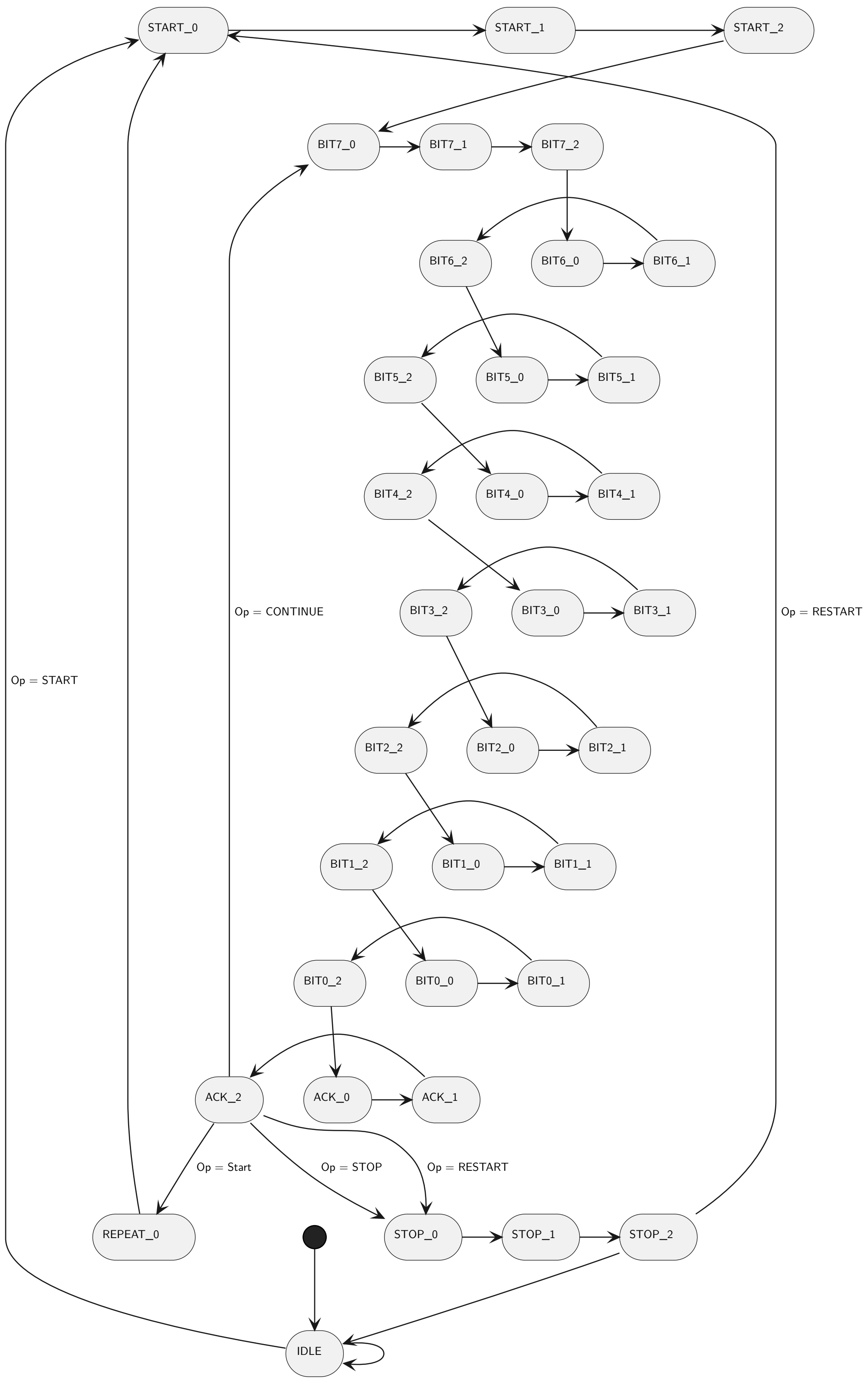


Figure 10: State diagram of I2CController

To set the registers listed in Table 2, I2CDataFeed contains the states partially listed in Table 4, which produces Figure 12, the omitted states can be easily inferred by referencing Table 2.

State	Op[1:0]	Data[7:0]
0	STOP	'h00
1	START	'h72 (slave address)
2	CONTINUE	'h98 (1st register address)
3	CONTINUE	'h03 (1st register value)
4	RESTART	'h72 (slave address)
5	CONTINUE	'h01 (2nd register address)
6	CONTINUE	'h00 (2nd register value)
7	RESTART	'h72 (slave address)
		⋮
93	CONTINUE	'h7d (last register value)

Table 4: Partial list of states within I2CDataFeed

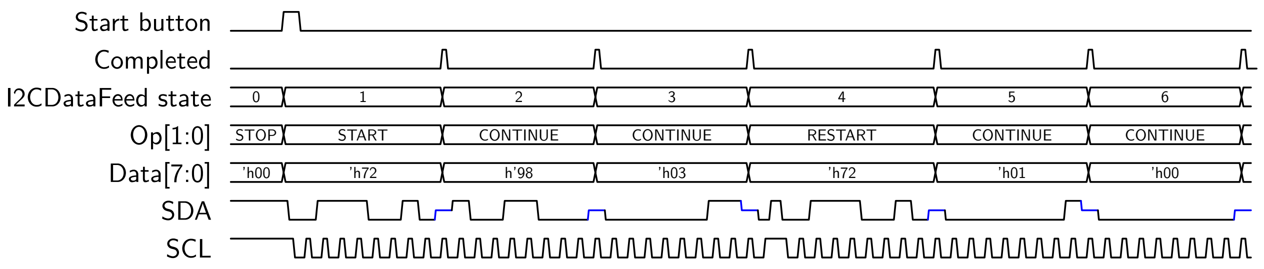


Figure 12: Operation of I2CSubsystem

4 Conclusion

The project was successful in the reduced scope of two alternating frames being displayed on the monitor using the ADV7513 through the FPGA on the DE-10 Nano. One of the lessons learned during testing was that the signals should be tested using equipment that reaches sample rates of at least three times the frequency of the component signal, this was found when low sampling rates produced artifacts in the waveform analysis performed on the 25 MHz clock signals. Another lesson learned was the amount of time initialising memory in the FPGA fabric takes, the compilation time increases to over minutes when one frame is imported into memory when using the method discribed in 3.4. Future work for this project could include moving the memory storage to a ROM IP block, incorporating the HPS to load image data from the SD card into RAM to display more of the video and increasing the video quality to the full 640×480 resolution currently set as the display resolution.

5 References

[1] Analog Devices, *Adv7513 low-power hdmi 1.4a transmitter – hardware user's guide*, Nov. 2011.
[2] *High-definition multimedia interface specification*, version 1.3a, Hitachi Ltd. et al., Nov. 2006.
[3] Analog Devices, *Adv7513 low-power hdmi 1.4a compatible transmitter – programming guide*, Oct. 2011.
[4] Terasic, *De10-nano user manual*, Terasic Inc, 2017.
[5] Intel. 'Terasic de10-nano kit.' (), [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/hardware/fpga-de10-nano.html (visited on 10/05/2022).
[6] R. Astley, *Never gonna give you up*, 27th Jul. 1987.
[7] J. D. Daniels, *Digital Design from Zero to One*. Wiley, 1996.
[8] G. Mathews, *Laboratory project, topic list and assessment guidelines - 2022*, https://rmit.instructure.com/courses/90952/files/23324580, Accessed: 2022-05-02, 2022.
[9] Mentor Graphics Corporation, *Modelsim® command reference manual*, 2015.
[10] *The Verilog® Golden Reference Guide*, Doulos, 1996.
[11] *Verilog® HDL Quick Reference Guide*, Sutherland HDL, Inc, 2001.
[12] John. 'Writing reusable verilog code using generate and parameters.' (Nov. 2020), [Online]. Available: https://fpgatutorial.com/verilog-generate/ (visited on 10/05/2022).
[13] DalonW. 'Angstrom on socfpga.' (Jun. 2019), [Online]. Available: https://rocketboards.org/foswiki/view/Documentation/AngstromOnSoCFPGA_1 (visited on 10/05/2022).
[14] Mario Nunez. 'Preloader and u-boot customization - v13.1.' (Sep. 2017), [Online]. Available: https://rocketboards.org/foswiki/view/Documentation/PreloaderUbootCustomization131 (visited on 10/05/2022).
[15] B. Li and W. Kaczurba. 'Hdmi made easy: Hdmi-to-vga and vga-to-hdmi converters,' Analog Devices. (2022), [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/hdmi-made-easy.html.

6 Appendices

6.1 Top Level Entity

```
module Project(
    // Buttons
    input BTN0,                     // Button I2C Trigger
    input RST_N,                    // Button Reset

    // Clocks
    input FPGA_CLK1_50,             // Clock 50MHz FPGA

    // From ADV7513
    // To ADV7513
    output          ADV_DE,         // ADV Data Enable
    output          ADV_CLK,        // ADV Video Clock
    output  [23:0]  ADV_D,          // ADV Video Data
    output          ADV_Hsync,      // ADV Horizontal Sync
    output          ADV_Vsync,      // ADV Vertical Sync
    inout           ADV_SDA,        // ADV Serial Port Data
    output          ADV_SCL,        // ADV Serial Port Data Clock

    // GPIO for debugging
    output [19:0] GPIO_1
);

    // Define Wires
    wire        pix_clk;
    wire        frame;
    wire        i2c_clk;
    wire        hsync;
    wire        vsync;
    wire    [9:0]   pix_x;
    wire    [9:0]   pix_y;
    wire    [15:0]  pix_addr;

    // Define Registers
    reg     i2c_start = 0;

    // Direct Assignments
    assign ADV_Hsync = ~hsync;
    assign ADV_Vsync = ~vsync;

    // Configure Clocks, ADV leads pix_clk by 90 degrees
    ClockDividerPow2 #(1) div_pix_clk(FPGA_CLK1_50,RST_N,pix_clk);
    ClockDividerPow2 #(1) div_adv_clk(~FPGA_CLK1_50,RST_N,ADV_CLK);
    ClockDividerPow2 #(12) i2c_clkdiv(FPGA_CLK1_50,RST_N,i2c_clk);

    // Configure Hsync, Vsync, ADV_DE, And Pixel x and y values
    PixelCursor pixel_cursor(
        pix_clk,
        RST_N,
        pix_x,
        pix_y,
        ADV_DE,
        hsync,
        vsync
    );

    // Get Pixel Address from x and y values
    PixelAddr PixAddr(
        pix_x,
        pix_y,
        pix_addr,
        frame
    );

    // Get Pixel Value from Pixel Address
    PixelStream PixStream(pix_addr,ADV_D);

    // Synchronous button press trigger for I2C
    always @(posedge(i2c_clk))
        i2c_start <= ~BTN0;

    // I2C Configuration
    I2CSubsystem i2c(
        .Start(i2c_start),
        .Clock(i2c_clk),
        .Reset_n(RST_N),
        .SDA(ADV_SDA),
        .SCL(ADV_SCL)
    );
endmodule
```

6.2 I2C Controller Module

```
module I2CController (
    input [7:0] Data,
    input [1:0] Op,
    input Clock,
    input Reset_n,

    // Indicate one byte transfer has completed, upstream should update Data and Op
    // Op and Data are only expected to update when state is IDLE or when Completed rises
    output Completed,

    inout SDA,
    output SCL
);

localparam OP_STOP          = 2'd0; // Stop indefinately
localparam OP_START         = 2'd1; // Send start or repeat start signal
localparam OP_CONTINUE      = 2'd2; // Continue sending data
localparam OP_RESTART       = 2'd3; // Stop the transaction and start a new one
localparam STATE_IDLE       = 6'd00;
localparam STATE_START_0    = 6'd01;
localparam STATE_START_1    = 6'd02;
localparam STATE_START_2    = 6'd03;
localparam STATE_BIT7_0     = 6'd04;
localparam STATE_BIT7_1     = 6'd05;
localparam STATE_BIT7_2     = 6'd06;
localparam STATE_BIT6_0     = 6'd07;
localparam STATE_BIT6_1     = 6'd08;
localparam STATE_BIT6_2     = 6'd09;
localparam STATE_BIT5_0     = 6'd10;
localparam STATE_BIT5_1     = 6'd11;
localparam STATE_BIT5_2     = 6'd12;
localparam STATE_BIT4_0     = 6'd13;
localparam STATE_BIT4_1     = 6'd14;
localparam STATE_BIT4_2     = 6'd15;
localparam STATE_BIT3_0     = 6'd16;
localparam STATE_BIT3_1     = 6'd17;
localparam STATE_BIT3_2     = 6'd18;
localparam STATE_BIT2_0     = 6'd19;
localparam STATE_BIT2_1     = 6'd20;
localparam STATE_BIT2_2     = 6'd21;
localparam STATE_BIT1_0     = 6'd22;
localparam STATE_BIT1_1     = 6'd23;
localparam STATE_BIT1_2     = 6'd24;
localparam STATE_BIT0_0     = 6'd25;
localparam STATE_BIT0_1     = 6'd26;
localparam STATE_BIT0_2     = 6'd27;
localparam STATE_ACK_0      = 6'd28;
localparam STATE_ACK_1      = 6'd29;
localparam STATE_ACK_2      = 6'd30;
localparam STATE_REPEAT_0   = 6'd31;
localparam STATE_STOP_0     = 6'd32;
localparam STATE_STOP_1     = 6'd33;
localparam STATE_STOP_2     = 6'd34;

reg [5:0] state = STATE_IDLE;

always @(posedge(Clock), negedge(Reset_n))
begin
    if (~Reset_n)
        state <= STATE_IDLE;
    else begin
        case (state)
            STATE_IDLE:
                state <= Op == OP_START ? STATE_START_0 : STATE_IDLE;
            STATE_ACK_2:
                case (Op)
                    OP_START:
                        state <= STATE_REPEAT_0;
                    OP_CONTINUE:
                        state <= STATE_BIT7_0;
                    default:
                        state <= STATE_STOP_0;
                endcase
            STATE_REPEAT_0:
                state <= STATE_START_0;
            STATE_STOP_2:
                state <= Op == OP_RESTART ? STATE_START_0 : STATE_IDLE;
            default:
                state <= state + 6'b000001;
        endcase
    end
end
```

```
reg sdar = 1; // register for SDA output
reg sda_en = 0; // if SDA should output or be high-Z
reg sclr = 1; // rergister for SCL
assign SDA = sda_en ? sdar : 1'bz; // tristate SDA
assign SCL = sclr;
assign Completed = state == STATE_ACK_1;

always @(state, Data)
begin
    case (state)
        STATE_IDLE: begin
            sdar <= 1;
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_START_0: begin
            sdar <= 1;
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_START_1: begin
            sdar <= 0;
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_START_2: begin
            sdar <= 0;
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT7_0: begin
            sdar <= Data[7];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT7_1: begin
            sdar <= Data[7];
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_BIT7_2: begin
            sdar <= Data[7];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT6_0: begin
            sdar <= Data[6];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT6_1: begin
            sdar <= Data[6];
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_BIT6_2: begin
            sdar <= Data[6];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT5_0: begin
            sdar <= Data[5];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT5_1: begin
            sdar <= Data[5];
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_BIT5_2: begin
            sdar <= Data[5];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT4_0: begin
            sdar <= Data[4];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT4_1: begin
            sdar <= Data[4];
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_BIT4_2: begin
            sdar <= Data[4];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT3_0: begin
            sdar <= Data[3];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT3_1: begin
            sdar <= Data[3];
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_BIT3_2: begin
            sdar <= Data[3];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT2_0: begin
            sdar <= Data[2];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT2_1: begin
            sdar <= Data[2];
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_BIT2_2: begin
            sdar <= Data[2];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT1_0: begin
            sdar <= Data[1];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT1_1: begin
            sdar <= Data[1];
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_BIT1_2: begin
            sdar <= Data[1];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT0_0: begin
            sdar <= Data[0];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_BIT0_1: begin
            sdar <= Data[0];
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_BIT0_2: begin
            sdar <= Data[0];
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_ACK_0: begin
            sdar <= 0;
            sda_en <= 0;
            sclr <= 0;
        end
        STATE_ACK_1: begin
            sdar <= 0;
            sda_en <= 0;
            sclr <= 1;
        end
        STATE_ACK_2: begin
            sdar <= 0;
            sda_en <= 0;
            sclr <= 0;
        end
        STATE_REPEAT_0: begin
            sdar <= 1;
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_STOP_0: begin
            sdar <= 0;
            sda_en <= 1;
            sclr <= 0;
        end
        STATE_STOP_1: begin
            sdar <= 0;
            sda_en <= 1;
            sclr <= 1;
        end
        STATE_STOP_2: begin
            sdar <= 1;
            sda_en <= 1;
            sclr <= 1;
        end
        default: begin
            sdar <= 1;
            sda_en <= 1;
            sclr <= 1;
        end
    endcase
end

endmodule
```

6.3 I2C Data Feed Module

```
module I2CDataFeed (
    input Update,
    input Reset_n,
    output reg [1:0] Op = 0,
    output reg [7:0] Data = 0
);

localparam OP_STOP          = 2'd0; // Stop indefinately
localparam OP_START         = 2'd1; // Send start or repeat start signal
localparam OP_CONTINUE      = 2'd2; // Continue sending data
localparam OP_RESTART       = 2'd3; // Stop the transaction and start a new one

localparam SLAVE_ADDR       = 8'h72; // or 'h7A ?

reg [6:0] state = 0;

always @(posedge(Update), negedge(Reset_n))
begin
    if (~Reset_n)
        state <= 0;
    else begin
        case (state)
            93: // The last state
                state <= 0;
            default:
                state <= state + 1'b1;
        endcase
    end
end

always @(state)
begin
    case (state)
        0: begin
            Op <= OP_STOP;
            Data <= 0;
        end
        1: begin
            Op <= OP_START;
            Data <= SLAVE_ADDR;
        end
        2: begin
            Op <= OP_CONTINUE;
            Data <= 'h98;
        end
        3: begin
            Op <= OP_CONTINUE;
            Data <= 'h03;
        end
        4: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        5: begin
            Op <= OP_CONTINUE;
            Data <= 'h01;
        end
        6: begin
            Op <= OP_CONTINUE;
            Data <= 'h00;
        end
        7: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        8: begin
            Op <= OP_CONTINUE;
            Data <= 'h02;
        end
        9: begin
            Op <= OP_CONTINUE;
            Data <= 'h18;
        end
        10: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        11: begin
            Op <= OP_CONTINUE;
            Data <= 'h03;
        end
        12: begin
```

```
            Op <= OP_CONTINUE;
            Data <= 'h00;
        end
        13: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        14: begin
            Op <= OP_CONTINUE;
            Data <= 'h14;
        end
        15: begin
            Op <= OP_CONTINUE;
            Data <= 'h70;
        end
        16: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        17: begin
            Op <= OP_CONTINUE;
            Data <= 'h15;
        end
        18: begin
            Op <= OP_CONTINUE;
            Data <= 'h20;
        end
        19: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        20: begin
            Op <= OP_CONTINUE;
            Data <= 'h16;
        end
        21: begin
            Op <= OP_CONTINUE;
            Data <= 'h30;
        end
        22: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        23: begin
            Op <= OP_CONTINUE;
            Data <= 'h18;
        end
        24: begin
            Op <= OP_CONTINUE;
            Data <= 'h46;
        end
        25: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        26: begin
            Op <= OP_CONTINUE;
            Data <= 'h40;
        end
        27: begin
            Op <= OP_CONTINUE;
            Data <= 'h80;
        end
        28: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        29: begin
            Op <= OP_CONTINUE;
            Data <= 'h41;
        end
        30: begin
            Op <= OP_CONTINUE;
            Data <= 'h10;
        end
        31: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        32: begin
            Op <= OP_CONTINUE;
            Data <= 'h49;
        end
        33: begin
```

```
            Op <= OP_CONTINUE;
            Data <= 'hA8;
        end
        34: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        35: begin
            Op <= OP_CONTINUE;
            Data <= 'h55;
        end
        36: begin
            Op <= OP_CONTINUE;
            Data <= 'h10;
        end
        37: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        38: begin
            Op <= OP_CONTINUE;
            Data <= 'h56;
        end
        39: begin
            Op <= OP_CONTINUE;
            Data <= 'h08;
        end
        40: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        41: begin
            Op <= OP_CONTINUE;
            Data <= 'h96;
        end
        42: begin
            Op <= OP_CONTINUE;
            Data <= 'hF6;
        end
        43: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        44: begin
            Op <= OP_CONTINUE;
            Data <= 'h73;
        end
        45: begin
            Op <= OP_CONTINUE;
            Data <= 'h07;
        end
        46: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        47: begin
            Op <= OP_CONTINUE;
            Data <= 'h76;
        end
        48: begin
            Op <= OP_CONTINUE;
            Data <= 'h1f;
        end
        49: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        50: begin
            Op <= OP_CONTINUE;
            Data <= 'h98;
        end
        51: begin
            Op <= OP_CONTINUE;
            Data <= 'h03;
        end
        52: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        53: begin
            Op <= OP_CONTINUE;
            Data <= 'h99;
        end
        54: begin
```

```
            Op <= OP_CONTINUE;
            Data <= 'h02;
        end
        55: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        56: begin
            Op <= OP_CONTINUE;
            Data <= 'h9a;
        end
        57: begin
            Op <= OP_CONTINUE;
            Data <= 'he0;
        end
        58: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        59: begin
            Op <= OP_CONTINUE;
            Data <= 'h9c;
        end
        60: begin
            Op <= OP_CONTINUE;
            Data <= 'h30;
        end
        61: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        62: begin
            Op <= OP_CONTINUE;
            Data <= 'h9d;
        end
        63: begin
            Op <= OP_CONTINUE;
            Data <= 'h61;
        end
        64: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        65: begin
            Op <= OP_CONTINUE;
            Data <= 'ha2;
        end
        66: begin
            Op <= OP_CONTINUE;
            Data <= 'ha4;
        end
        67: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        68: begin
            Op <= OP_CONTINUE;
            Data <= 'ha3;
        end
        69: begin
            Op <= OP_CONTINUE;
            Data <= 'ha4;
        end
        70: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        71: begin
            Op <= OP_CONTINUE;
            Data <= 'ha5;
        end
        72: begin
            Op <= OP_CONTINUE;
            Data <= 'h04;
        end
        73: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        74: begin
            Op <= OP_CONTINUE;
            Data <= 'hab;
        end
        75: begin
```

```
            Op <= OP_CONTINUE;
            Data <= 'h40;
        end
        76: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        77: begin
            Op <= OP_CONTINUE;
            Data <= 'haf;
        end
        78: begin
            Op <= OP_CONTINUE;
            Data <= 'h14;
        end
        79: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        80: begin
            Op <= OP_CONTINUE;
            Data <= 'hba;
        end
        81: begin
            Op <= OP_CONTINUE;
            Data <= 'h60;
        end
        82: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        83: begin
            Op <= OP_CONTINUE;
            Data <= 'hd1;
        end
        84: begin
            Op <= OP_CONTINUE;
            Data <= 'hff;
        end
        85: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        86: begin
            Op <= OP_CONTINUE;
            Data <= 'hde;
        end
        87: begin
            Op <= OP_CONTINUE;
            Data <= 'h10;
        end
        88: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        89: begin
            Op <= OP_CONTINUE;
            Data <= 'he4;
        end
        90: begin
            Op <= OP_CONTINUE;
            Data <= 'h60;
        end
        91: begin
            Op <= OP_RESTART;
            Data <= SLAVE_ADDR;
        end
        92: begin
            Op <= OP_CONTINUE;
            Data <= 'hfa;
        end
        93: begin
            Op <= OP_CONTINUE;
            Data <= 'h7d;
        end
        default: begin
            Op <= OP_STOP;
            Data <= 0;
        end
    endcase
end

endmodule
```

6.4 I2C Subsystem Module

```
module I2CSubsystem (
    // Per Hardware User Guide:
    // > The user should wait 200ms for the address to be decided
    // Therefore I2CSubsystem does not start running upon reset, but instead
    // waits for a Start signal, which may come from software or button.
    input Start,
    input Clock, // Slow clock for I2C
    input Reset_n,

    inout SDA,
    output SCL
);

wire controller_completed;

wire update_data = Start | controller_completed;
//reg update_data = 0;
//always @(posedge(Clock), negedge(Reset_n))
//begin
//  if (~Reset_n)
//      update_data <= 0;
//  else
//      update_data <= Start | controller_completed;
//end

wire [7:0] data;
wire [1:0] op;

I2CController controller (
    .Data(data),
    .Op(op),
    .Clock(Clock),
    .Reset_n(Reset_n),
    .Completed(controller_completed),
    .SDA(SDA),
    .SCL(SCL)
);

I2CDataFeed data_feed (
    .Update(update_data),
    .Reset_n(Reset_n),
    .Op(op),
    .Data(data)
);

endmodule
```

6.5 Clock Divider Module

```
module ClockDividerPow2 #(parameter POWER)(

    input clk_in,
    input reset_n,
    output clk_out
);

reg [POWER-1:0] counter_value = 0;

always @(posedge(clk_in), negedge(reset_n))
begin
    if (~reset_n)
        counter_value <= 0;
    else
        counter_value <= counter_value + 1'b1;
end

assign clk_out = counter_value[POWER-1];

endmodule
```

6.6 Pixel Address Module

```
module PixelAddr(

    input   [9:0]   xpos,
    input   [9:0]   ypos,
    output  [15:0]  address,
    output  [7:0]   frameOut

);

    reg     [15:0]  addr    = 16'b0;
    reg     [15:0]  offset  = 16'b0;
    reg     [5:0]   frame   = 05'b0;

    always @(*)
    begin
        if (xpos < 10'd640 && ypos < 10'd480)
        begin
            addr <= (ypos/10'd4) * 10'd160 + (xpos/10'd4) + offset;
        end
        else if (ypos==10'd524)
        begin
            if (frame < 5'h10)
            begin
                offset <=  16'h4b00;
                frame <= frame + 5'h01;
            end
            else
            begin
                offset <= 16'b0;
                frame <= frame + 5'h1;
            end
        end
        else
        begin
            addr <= 16'b1;
        end
    end

    assign frameOut = (frame < 5'h10);
    assign address = addr;

endmodule
```

6.7 Pixel Cursor Module

```
module PixelCursor (
    input pix_clk,  // 25 MHz
    input reset_n,
    output reg [9:0] hcount = 0,
    output reg [9:0] vcount = 0,
    output active,
    output hsync,
    output vsync
);

always @(posedge(pix_clk), negedge(reset_n)) begin
    if (~reset_n) begin
        hcount <= 0;
        vcount <= 0;
    end else begin
        if (hcount == 799) begin
            hcount <= 0;
            if (vcount == 524)
                vcount <= 0;
            else
                vcount <= vcount + 1'b1;
        end else
            hcount <= hcount + 1'b1;
    end
end

assign active = hcount < 640 && vcount < 480;
assign hsync = hcount >= 688 && hcount < 784;
assign vsync = vcount >= 513 && vcount < 515;

endmodule
```

6.8 Pixel Stream (Memory Stream) Module

```
module PixelStream(

    input   [15:0]  address,
    output  [23:0]  pixel

);
    // Define Registers
    reg [23:0] memory [38399:0];
    reg [23:0] value;

    // Initialise all registers
    initial
    begin
        value = 24'b0;
        $readmemh("PixStream.mem",memory);
    end

    always @(*)
    begin
        value <= memory[address];
    end
    assign pixel = value;
endmodule
```