Advanced Digital Design 1 (EEET2162) Laboratory Project

Oliver Patterson (S3723206)
Kaizhe Huang (S3686152)

Laboratory Demonstrator:
Genevieve Fahey

Date:
21/05/2022

RMIT School of
Engineering

Executive Summary

Contents

1 Introdution

2 Literature Search

3 Technical Work and Results

4 References

[1] J. D. Daniels, *Digital Design from Zero to One*. Wiley, 1996.

[2] G. Mathews, *Laboratory project, topic list and assessment guidelines - 2022*, https://rmit.instructure.com/courses/90952/files/23324580, Accessed: 2022-05-02, 2022.

[3] M. G. Corporation, *Modelsim® command reference manual*, 2015.

[4] *The Verilog® Golden Reference Guide*, Doulos, 1996.

[5] *Verilog® HDL Quick Reference Guide*, Sutherland HDL, Inc, 2001.

[6] Terasic, *De10-nano user manual*, Terasic Inc, 2017.

[7] John. "Writing reusable verilog code using generate and parameters." (Nov. 2020), [Online]. Available: https://fpgatutorial.com/verilog-generate/ (visited on 05/10/2022).

[8] DalonW. "Angstrom on socfpga." (Jun. 2019), [Online]. Available: https://rocketboards.org/foswiki/view/Documentation/AngstromOnSoCFPGA_1 (visited on 05/10/2022).

[9] M. Nunez. "Preloader and u-boot customization - v13.1." (Sep. 2017), [Online]. Available: https://rocketboards.org/foswiki/view/Documentation/PreloaderUbootCustomization131 (visited on 05/10/2022).

[10] 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.

5 Appendices

5.1 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
```

5.2 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
```

5.3 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
```

5.4 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
```

5.5 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
```

5.6 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
```

5.7 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
```

5.8 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
```