ECE271, IR Remote Design Project, Group 05

Shea Mcneely, Christopher Qualls, Brian Castellon Rosales

June 9, 2023

Contents

1 Project Description

Our project is to create a game by using sprites, an FPGA board, VGA, and an IR receiver. The IR receiver receives input from infrared light(IR). Infrared light is non-visible light waves used in remote controls and other electronic devices to receive and send information for various functions. We are using the control used in auraLED light strips to send information to the IR receiver. The IR receiver is connected to the FPGA board which will decode the information the IR receiver reads from the control. The FPGA decodes the information for a few different buttons on the control. This makes the sprite move up, down, left, or right. The sprite we are using is the following Sonic sprite:

Our game consists of this sprite simply moving within the screen using the IR remote as a control. If you press the designated "right" button on the remote, the sprite responds and moves accordingly. The other directions follow the same principle.

2 Project Design

2.1 Brainstorming

As the VGA and sprite driver were designed for labs 6 and 7 respectively, no extra brainstorming was required for their design. The IR sensor, however, needed to be completely designed from scratch. The first initial ideas came in the form of figuring out how to recognize a new code coming into the FPGA. This led to us resortin g to looking at the NEC IR protocol. This protocol consists of 108ms blocks with 67.5ms pertaining to relevant input values. Remotes will output a 9ms start block followed by another 4.5ms block. Then the 8-bit address and its logic inverse follows in a 27ms time slot. The final 27ms are reserved for data and its logical inverse. This data section is the only section we care about for this project.

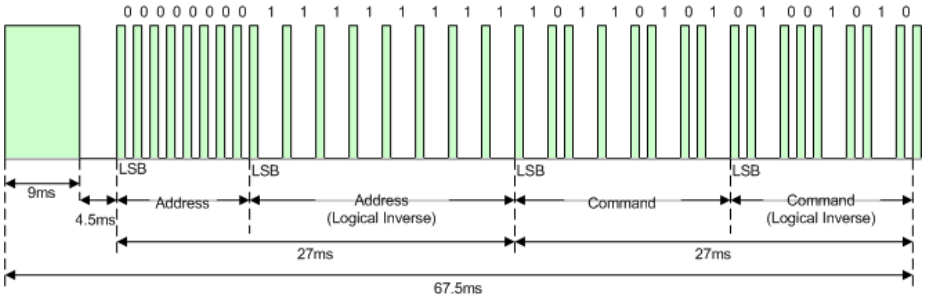


Figure 1: NEC IR Protocol initial block timing diagram

After the initial 108ms block is sent, the remote sends what is called a "repeat code". This can lead to issues as there is no address or a data block in these repeat codes. This special case will need to be dealt with in the design.

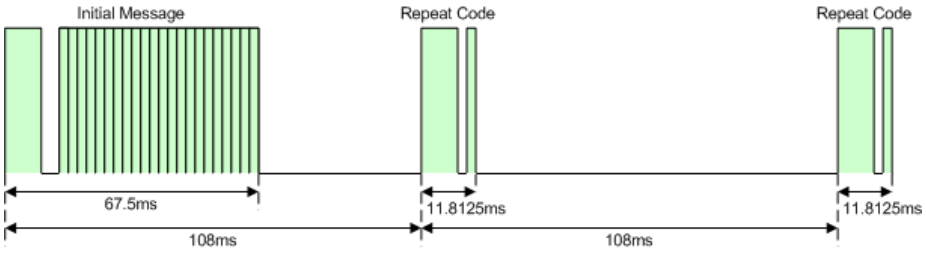


Figure 2: NEC IR Protocol inital message and repeat message

The idea of using an overall state machine with smaller blocks that work within the control of the higher state machine block came from lecture; this is

what we will be implementing in this report. There are four states, an idle state which detects a new code coming in, another start block that reads a valid start code and makes sure it is not a repeat code, a data storing state, and an interpret code or wait state.

To begin figuring out how these blocks will work, we first sketched out ideas of how each block interacts with each other and the state control. The following diagram was designed:

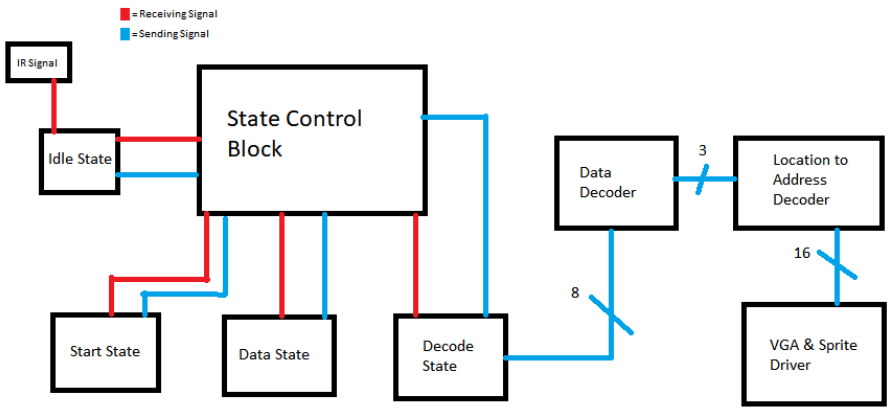


Figure 3: High-level block diagram

Each state block has an enable line, marked with blue, that allows the clock to enter the block and a nextstate line, in red, to tell the state control when the block completes its timed task.

After the final state is complete, an 8-bit data block will be in the decode state's register. This is output to a decoder which interprets the data and sends a decoded bit to our start position decoder for the sprite. After this sprite location is established, it is sent to the VGA driver which puts the sprite onto the screen.

2.2 Oscilloscope

The next step in the design process was to figure out the codes we needed from the remote. To find these, we hook up our IR sensor to an oscilloscope and manually read the data from the sensor. An example code is in Figure 4. This helped us to establish the IR sensor is an active low device. A large low pulse indicates the start of a new packet and the values, 0 or 1, within these packets are determined by the length the pulses hold high after a uniform low pulse length.

After establishing the system in which the IR sensor runs, we found all four codes we would need to decode and move the sprite. In our design, we use the logical inverse of the data section for the codes. They are up, denoted by a data packet of 01101111, down, or 01110111, left as 11001111, and right as 10001111. These are found in Figures 5, 6, 7, and 8 respectively.

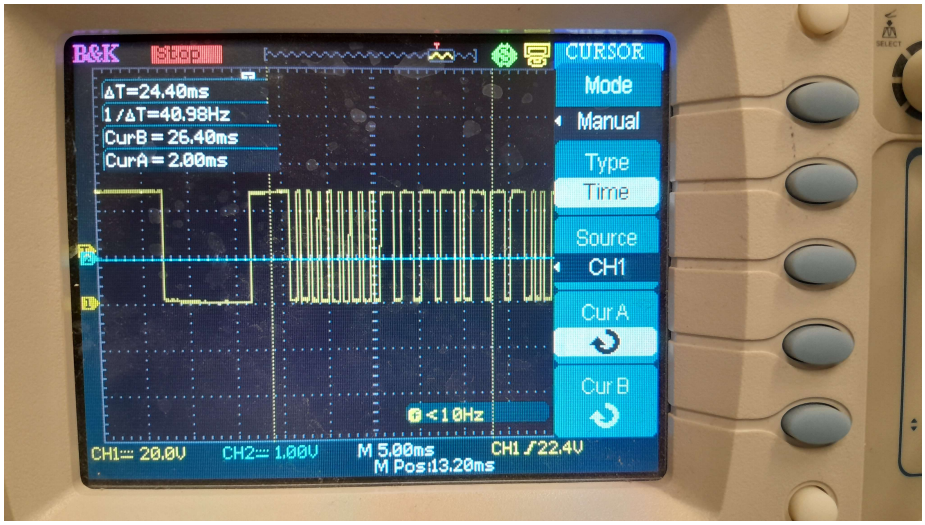


Figure 4: Example IR code

Figure 5: Up IR code

Figure 6: Down IR code

Figure 7: Left IR code

Figure 8: Right IR code

After decoding which binary codes we need, the next step was to design the blocks to correctly get these codes and feed them into the VGA driver.

2.3 Schematics and SystemVerilog

The clock to run all of the timing systems within the state control and state blocks is 6.25MHz. To find the times for the IR system, we multiply the clock frequency by the real times we need from the NEC protocol. This gives a number that we count to using a counter block. If the counter is over a certain value, by using a comparator block, it means we have waited for the time allocated.

The idle state requires a system to sense when the IR signal goes low, but it needs to filter out random noise. This noise is filtered out using a timing system. If the IR signal goes low for 56250 cycles, or 9ms, we know it is a proper start block and not random noise. This sets a buffer flag and means we can move on to the next state.

In figure 9, we have the RTL view of the state control logic block. The block is written in SystemVerilog. A simple system of nextstate and state that are cycled at the clock is implemented in the middle. The top left of the block is the counter to filter noise and the right is to set the enable bits for the state blocks. The PS system is explained more in the start block.

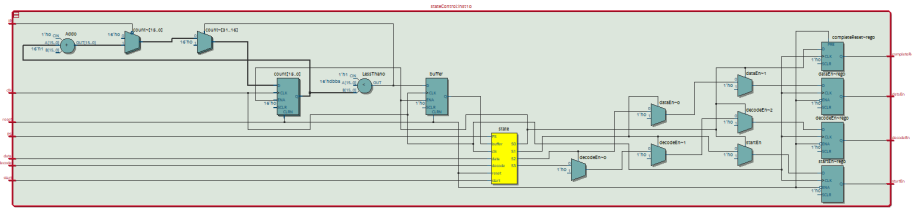


Figure 9: RTL view of statecontrol.sv

The first block that runs after idle is the start block, this block checks that we have a valid high time after the start block and a valid address. If we have a repeat code, as shown in Figure 2, there is no proper address sent in the packet and it's instead a low signal. We use this to our advantage to filter out repeat blocks. There is a clock that resets every time the IR signal goes high. If the value of this counter is over 6ms, the PS signal activates. This sets the state control back to state 0, or idle. This prevents the state machine from trying to shift in empty data codes.

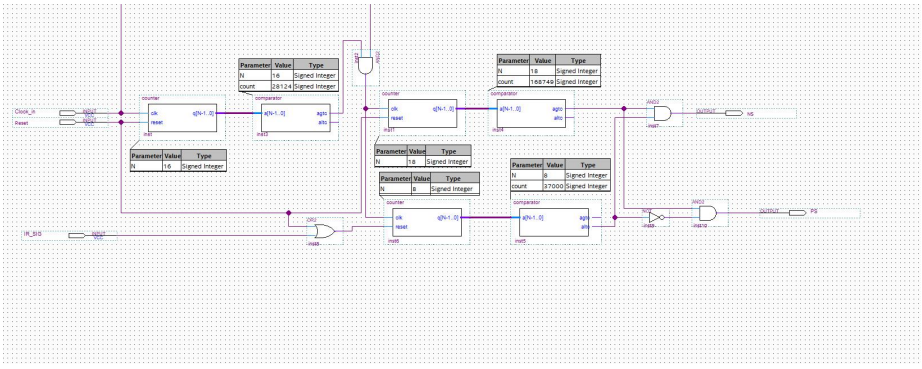


Figure 10: Start state block diagram

After this start state completes, the state moves to the data-shifting state. This state interprets the different lengths of the IR signal as a 0 or 1 and shifts it into the data register. It shifts all 16 bits and as the register is only 8-bits long, it shifts the code out and keeps the logical inverse of this code. After the 27ms that represents this data section of the IR signal, the state moves on and waits until the 108ms time frame. This is the decode state. After this, the state machine moves back into idle automatically and waits for a new input packet.

The data block works by knowing that the only difference between the 0 and 1 codes is the time low. This means we simply count how long the signal is low, and if it's over 3000 clock cycles (5ms), we shift in a 1; otherwise, we know the value is a 0. The counter and shift in signal for these bit interpret blocks are reset every time the IR signal goes high, which is after every sent bit. This system is shown in Figure 10.

The next state logic is simply a counter that counts up to 168750 clock cycles, or 27ms. This means that we have reached the end of the data section of the packet and now need to decode the data. This shifts the state to the decode state, which simply waits and sends a decoded value to the address generator block. This block takes the 3-bit values sent to it by the decoder and outputs a horizontal and vertical position for the VGA signal to interpret. The decoder takes the 4 input codes we have and generates a unique binary value from them for the address block to utilize.

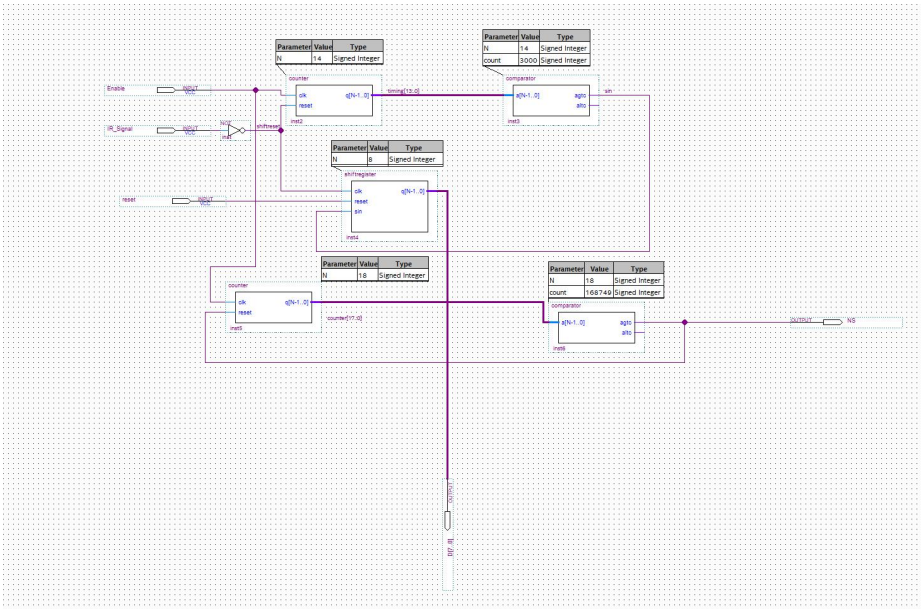


Figure 11: Block diagram of data state block

The decode block is made up of two smaller blocks: a shift interpreter which reads the codes and sends a next state signal, and a decoder which takes the value from the interpreter and converts it to horizontal and vertical positions on the screen. The shift interpreter has a simple look-up table for the 4 values and outputs the respective 3-bit outputs and a counter which checks if the next state signal should be active. This occurs after 67.5ms. The decoder has two values for the horizontal and vertical position (initialized to 0) and updates those based on the input 3-bit. They update twice every second by a counter dividing the 6.25MHz clock down 18 times (2^{18}).

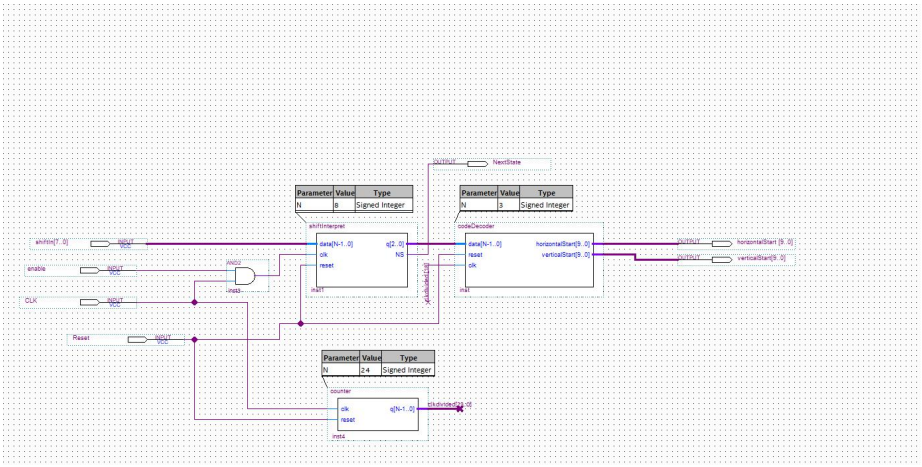


Figure 12: IR decode state block

After completing the path through the IR decode block. The start values for the sprite are sent to the address decoder block. This block takes the horizontal and vertical values on it's input and utilizes them, along with the vertical and horizontal count from the VGA timing block, to generate an address for the memory block. This address then outputs the correct pixel colors to the RGB 12-bit output. The address is calculated using an x and y value derived from the current location vertically or horizontally subtracted by the values from the decode block. These values are then used as a multiplier and counter for the index of the memory.

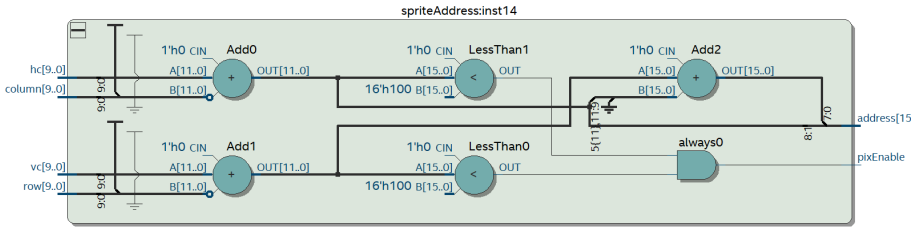


Figure 13: spriteAddress.sv RTL view

The VGA timing is generated using horizontal and vertical counters that activate comparator blocks at given values. This generates the Vsync, Hsync, and blanking signals. The blanking signal forces the output MUXs to output 0s during times when the screen is not active.

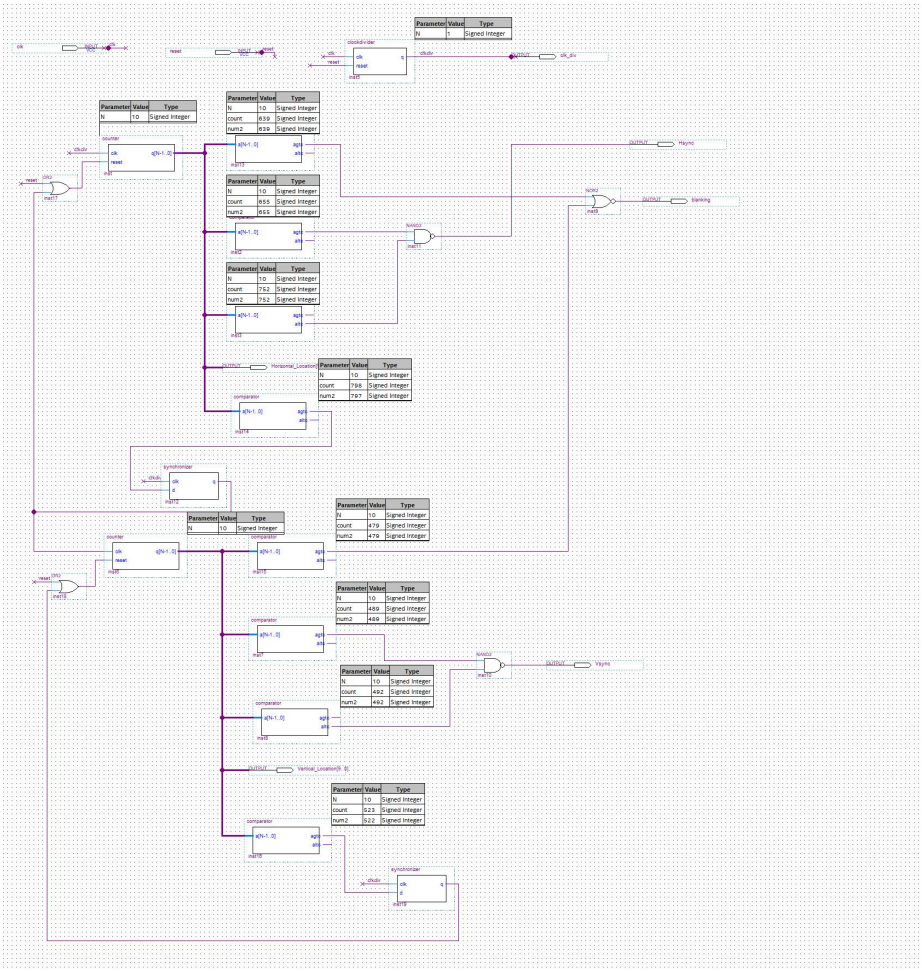


Figure 14: VGA Timing block diagram

The final block on the design is the sprite block. This block is generated using a .mif file. This file is imported into Quartus and uses 65536 words that are 8-bits wide. The input is simply an address and an enable bit. This enable bit is devised by the spriteAddress block. This signal is high whenever the horizontal and vertical counters are within 256 pixels of the horizontal and vertical start values. This makes sure to only output a single instance of the sprite to the screen.

2.4 Interface Definition Table:

The interface of the sprite and the user is through two inputs. A switch that resets the device and an IR input that is controlled by a remote. This remote

feeds data that the IR Signal sends into the FPGA through the GPIO. After processing, the FPGA outputs the contents of the data shift register to the LEDs and then outputs the sprite onto the screen using the Hsync, Vsync, Red, Green, and Blue pins.

Interface Name	Interface Purpose	Input or Output	Number of Pins	Active High or Low	Board Label	FPGA Pin
CLK	Circuit Clock	Input	1	Active High (Rising Edge)	CLK 50 MHz	P11
Reset	Register and State Machine Reset	Input	1	Active High	Switch 0	C10
Hsync	VGA Timing	Output	1	Active Low	Horizontal sync	N3
Vsync	VGA Timing	Output	1	Active Low	Vertical sync	N1
Red[3..0]	VGA Color	Output	4	Active High	VGA Red	AA1, V1, Y2, Y1
Green[3..0]	VGA Color	Output	4	Active High	VGA Green	W1, T2, R2, R1
Blue[3..0]	VGA Color	Output	4	Active High	VGA Blue	P1, T1, P4, N2
Data[7..0]	Shift Register Value	Output	8	Active High	LED	A8, A9, A10, B10, D13, C13, E14, D14
IR Signal	IR Input	Input	1	Active Low	2x20 GPIO	AA15

Table 1: Interface Definition

3 Results

3.1 ModelSim and Testing

This design required many testing sessions to get working properly. This is due to continually finding bugs. After every test session, we would find a small mistake and have to fix it, only to find it was concealing other issues. Eventually, we arrived at a version that works as intended with as little issues as possible. This version is shown working in the following video: https://www.youtube.com/watch?v=$_0qnckJ_GZg$

All the below figures are the Modelsim tests we performed during the design and verification process. Simulations helped us find where bugs in the design may be occuring and how to fix them.

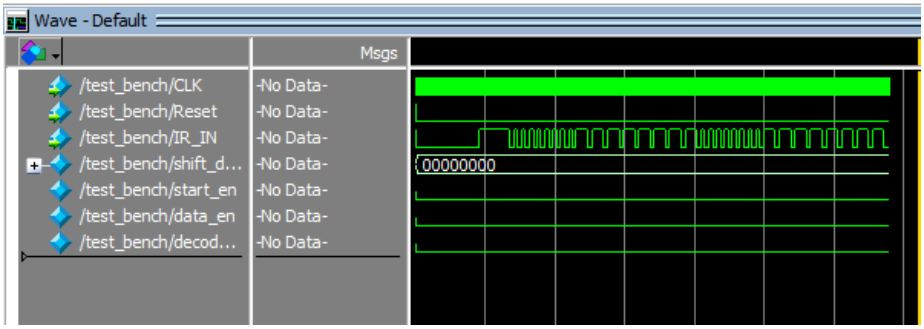


Figure 15: IR Signal test in ModelSim

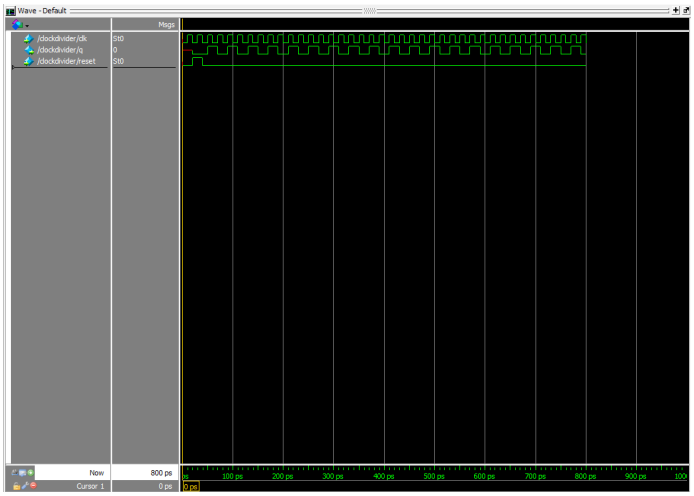


Figure 16: Clock Divider block

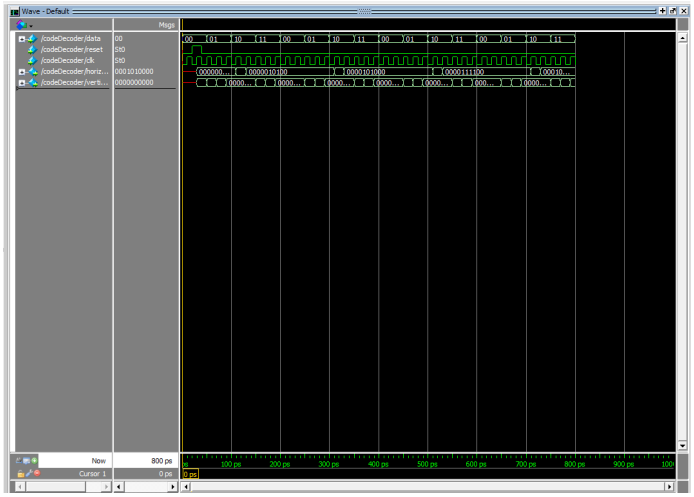


Figure 17: Code decoder that outputs values for the address generator

Figure 18: Shift Register Code Interpreter

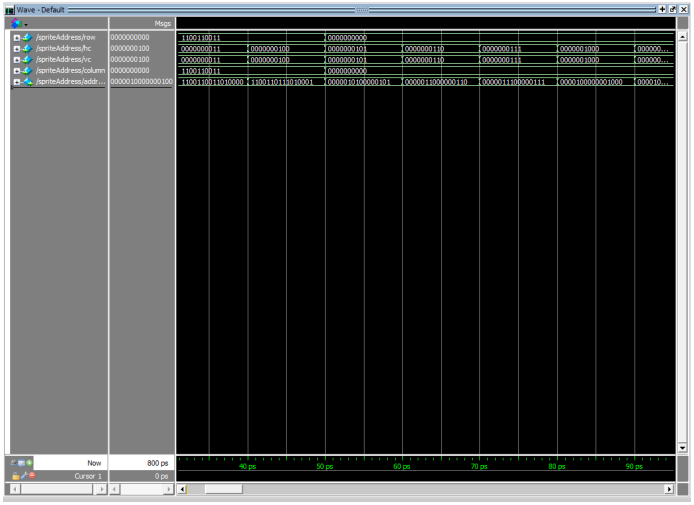


Figure 19: Address generator from input x y locations

3.2 Experiment Notes

The experiment went far from smoothly. A major issue we continually ran into was the repeat codes disrupting the data portion of the packet and shifting our wanted data values out of the register. After realizing we could filter out these errors with some state machine and start block tweaking, everything started to fall into place. The completion of the IR block finally allowed us to hook up the VGA and test whether the implementation worked at full course. It eventually did after adding an additional counter for sprite update frame-rate.

4 FPGA Usage

4.1 Introduction

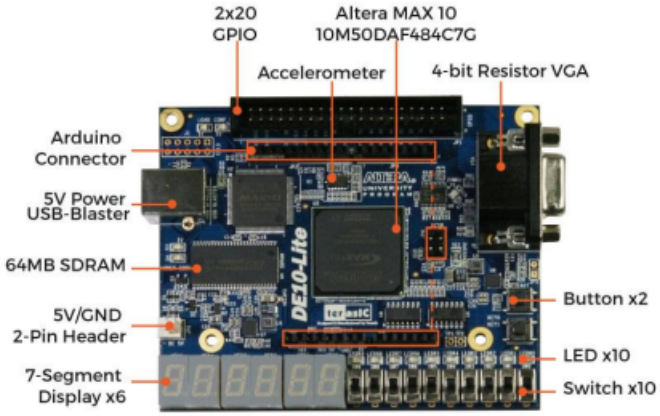


Figure 20: DE-10 Lite Layout

We used the DE-10 Lite FPGA board. This is the board that was provided to us by OSU for our ECE272 Lab. The DE-10 Lite has a lot of different components available for use. This list includes an accelerometer, ten switches, two buttons, six 7-Segment displays, 2x20 GPIO, two 50mHz clock inputs, and much more. For our project, we used one of the 50mHz clock inputs (we did have to slow it down for it to work with our project), one switch, the VGA output, and a few pins from the 2x20 GPIO for our IR receiver.

4.2 Top-Level Block Diagram

The block diagram shows the high level of our design. There are two sections consisting of an IR state machine at the top and the VGA driver at the bottom section. A clock divider for the entire system sits in the bottom left. These two sections interact with each other through two horizontal and vertical position busses.

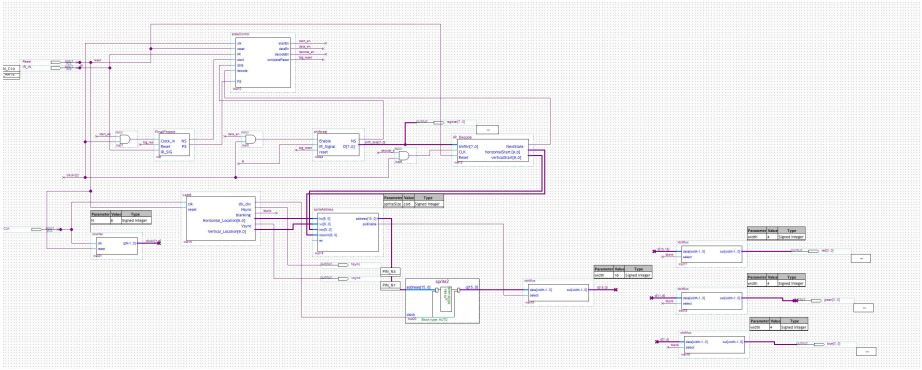


Figure 21: Overall block diagram with state machine and VGA driver

The physical diagram is very simple, with the only external device being the IR sensor. This sensor is connected to the GPIO of the DE10 and is controlled externally by the IR remote.

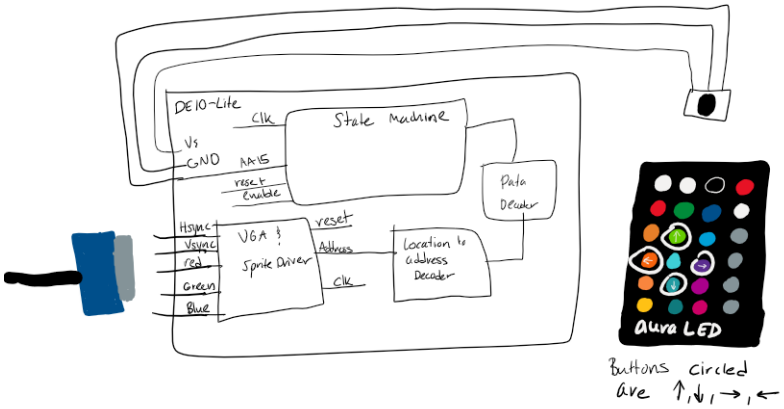


Figure 22: DE-10 Lite Diagram

4.3 Block Usage

The design only uses a small portion of the available Logic Elements on the DE10-Lite. A total of 324 are used, with only 151 of the available registers (D Flip Flops) created in these logic elements. This is mostly due to all of the counters and latches needed for the design to work as intended. A total of less than a percentage is a good efficiency for the design.

Flow Status	Successful - Fri Jun 09 13:22:00 2023
Quartus Prime Version	18.0.0 Build 614 04/24/2018 SJ Lite Edition
Revision Name	test_bench
Top-level Entity Name	test_bench
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	324 / 49,760 (< 1 %)
Total registers	151
Total pins	25 / 360 (7 %)
Total virtual pins	0
Total memory bits	786,432 / 1,677,312 (47 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 23: Total FPGA usage report

The block map shows a similar story, where only a small amount of Logic Elements are used within a small area. They all line a RAM block and this allows short connections between busses. This means the design can efficiently run and has small propagation delays between LEs.

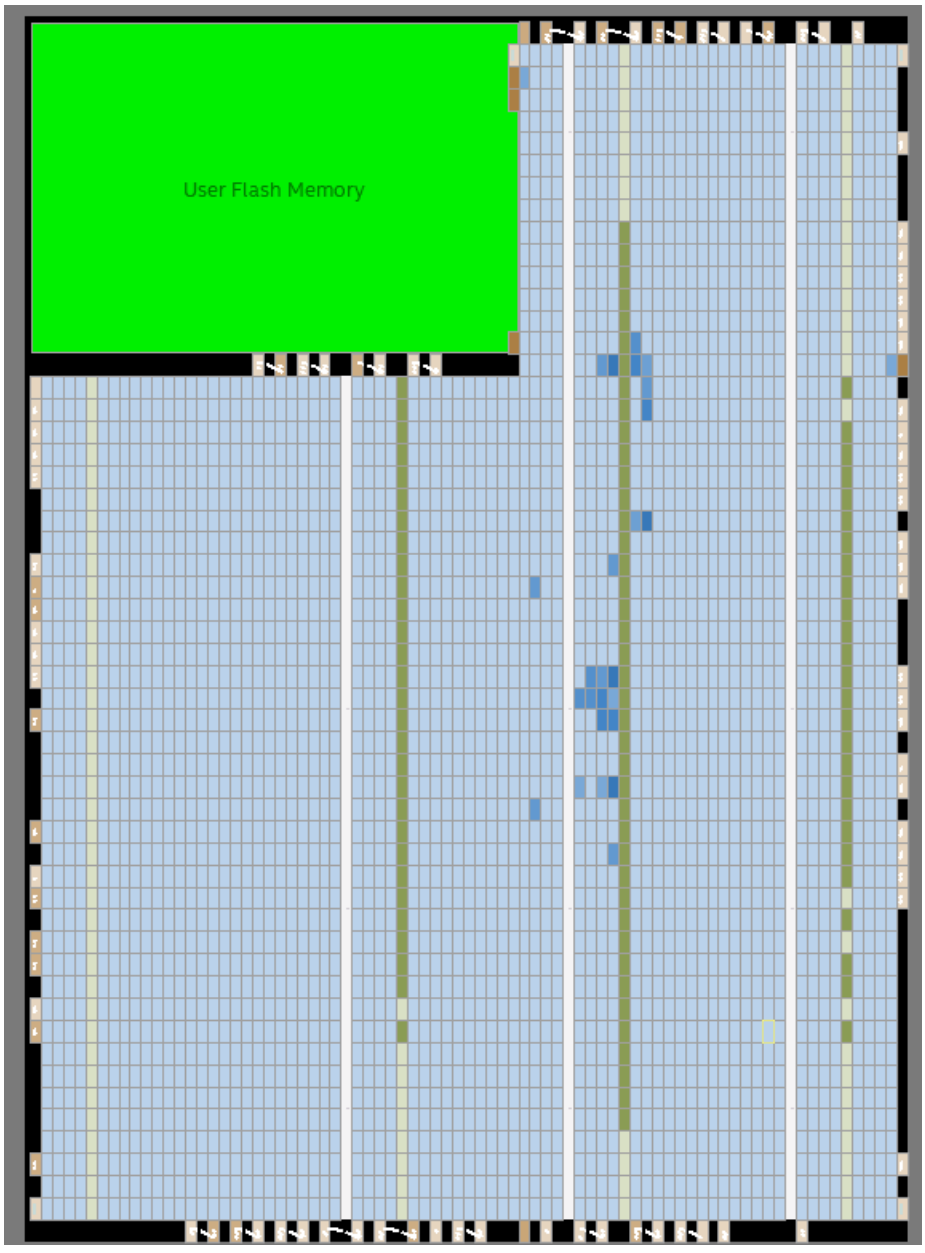


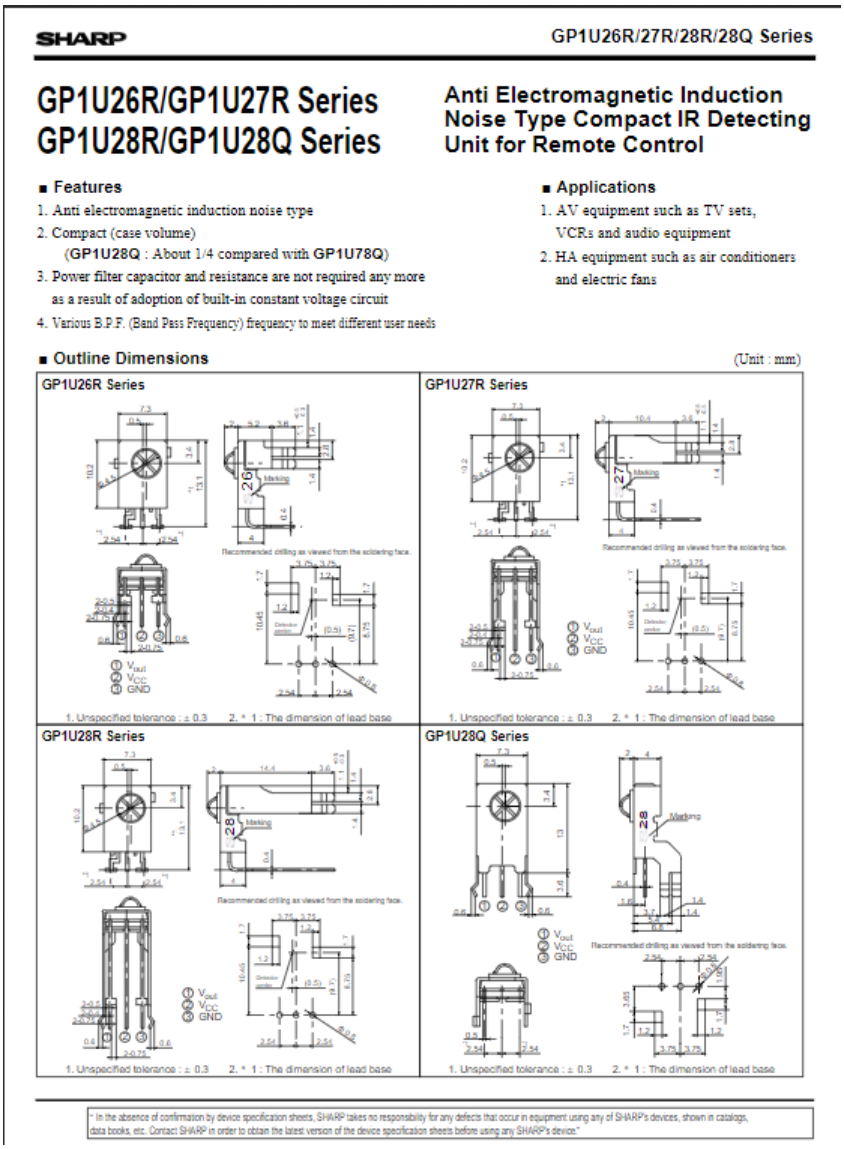
Figure 24: FPGA usage block map

4.4 IR Sensor and VGA

4.4.1 IR Receiver Sharp GPU1U28Q:

Figure 25: IR Receiver GP1U28Q

This was the IR Receiver that we used for our project. This receiver is from Sharp and is usually used to receive signals from remote controls. We purchased this IR Receiver from the Resistore on the OSU Corvallis Campus. We connected this receiver into the FPGA board by using a breadboard, and three wires which are connected to the 2x20 GPIO pins.



SHARP GP1U26R/27R/28R/28Q Series

GP1U26R/GP1U27R Series
GP1U28R/GP1U28Q Series

Anti Electromagnetic Induction Noise Type Compact IR Detecting Unit for Remote Control

■ Features

1. Anti electromagnetic induction noise type
2. Compact (case volume)
   (**GP1U28Q** : About 1/4 compared with **GP1U78Q**)
3. Power filter capacitor and resistance are not required any more as a result of adoption of built-in constant voltage circuit
4. Various B.P.F. (Band Pass Frequency) frequency to meet different user needs

■ Applications

1. AV equipment such as TV sets, VCRs and audio equipment
2. HA equipment such as air conditioners and electric fans

■ Outline Dimensions (Unit : mm)

GP1U26R Series

1. Unspecified tolerance : ± 0.3 2. * 1 : The dimension of lead base

GP1U27R Series

1. Unspecified tolerance : ± 0.3 2. * 1 : The dimension of lead base

GP1U28R Series

1. Unspecified tolerance : ± 0.3 2. * 1 : The dimension of lead base

GP1U28Q Series

1. Unspecified tolerance : ± 0.3 2. * 1 : The dimension of lead base

Figure 26: Page 1 of the IR Receiver Datasheet

This is the main page of the datasheet for the IR Receiver. It gives which pin does which, the dimensions of the IR Receiver, and a description. It also gives some applications and features of the sensor.

4.4.2 VGA

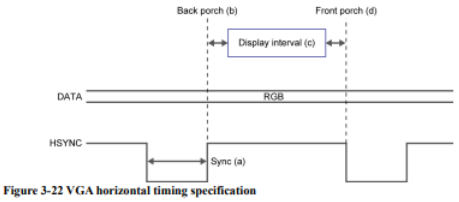


Figure 3-22 VGA horizontal timing specification

Table 3-9 VGA Horizontal Timing Specification

VGA mode		*Horizontal Timing Spec*				
Configuration	*Resolution(HxV)*	*a(pixel clock cycle)*	*b(pixel clock cycle)*	*c(pixel clock cycle)*	*d(pixel clock cycle)*	*Pixel clock(MHz)*
VGA(60Hz)	640x480	96	48	640	16	25

Table 3-10 VGA Vertical Timing Specification

VGA mode		*Vertical Timing Spec*				
Configuration	*Resolution(HxV)*	*a(lines)*	*b(lines)*	*c(lines)*	*d(lines)*	*Pixel clock(MHz)*
VGA(60Hz)	640x480	2	33	480	10	25

Table 3-11 Pin Assignment of VGA

Signal Name	FPGA Pin No.	Description	I/O Standard
VGA_R0	PIN_AA1	VGA Red[0]	3.3-V LVTTL
VGA_R1	PIN_V1	VGA Red[1]	3.3-V LVTTL
VGA_R2	PIN_Y2	VGA Red[2]	3.3-V LVTTL
VGA_R3	PIN_Y1	VGA Red[3]	3.3-V LVTTL
VGA_G0	PIN_W1	VGA Green[0]	3.3-V LVTTL
VGA_G1	PIN_T2	VGA Green[1]	3.3-V LVTTL
VGA_G2	PIN_R2	VGA Green[2]	3.3-V LVTTL
VGA_G3	PIN_R1	VGA Green[3]	3.3-V LVTTL
VGA_B0	PIN_P1	VGA Blue[0]	3.3-V LVTTL
VGA_B1	PIN_T1	VGA Blue[1]	3.3-V LVTTL
VGA_B2	PIN_P4	VGA Blue[2]	3.3-V LVTTL
VGA_B3	PIN_N2	VGA Blue[3]	3.3-V LVTTL
VGA_HS	PIN_N3	VGA Horizontal sync	3.3-V LVTTL
VGA_VS	PIN_N1	VGA Vertical sync	3.3-V LVTTL

Figure 27: VGA Datasheet

This is the datasheet for the VGA for the DE-10 Lite FPGA board. It provides the horizontal and vertical timing specs. This is used to know sync, back poarch, display interval, and front porch of the VGA output. It also provides all the pin information which we use to output the sprite through the VGA and Sprite driver.

5 Appendix

5.1 References

IR Receiver Sharp GP1U28Q Data Sheet:

https://pdf1.alldatasheet.com/datasheet-pdf/view/42826/SHARP/GP1U28Q.html

Mif Maker Python Program by p-bodson:

https://github.com/p-bodson/mifMaker

DE-10 Lite FPGA Board Manual

https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/Boards/DE10-Lite/DE10_Lite_User_Manual.pdf

5.2 Drive Link with Project Files:

https://drive.google.com/drive/folders/1-tXW9OSukzKnK6J86546anoA3nmd7ZZ6?usp=sharing

5.3 SystemVerilog Files

```
1   module comparator #(parameter N=8, parameter count=6)
2
3           (input logic [N-1:0] a,
4           output logic agtc,
5           output logic altc);
6
7       assign agtc = (a>=count);
8       assign altc = (a<=count);
9
10  endmodule
```

```
module codeDecoder #(parameter N = 2) (input logic[N-1:0] data, input logic reset, input logic clk,
                                       output logic[9:0] horizontalStart, output logic[9:0] verticalStart);
    always_ff @(posedge clk)
    begin
        if(reset) begin
            horizontalStart = 0;
            verticalStart = 0;
        end
        case(data)

                0: begin horizontalStart = horizontalStart; verticalStart = verticalStart; end
                1: verticalStart = verticalStart - 1;
                2: horizontalStart = horizontalStart + 1;
                3: verticalStart = verticalStart + 1;
                4: horizontalStart = horizontalStart - 1;
                default: begin verticalStart = verticalStart; horizontalStart = horizontalStart; end
        endcase

        if(horizontalStart > (384)) horizontalStart = 384;
        if(verticalStart > (224)) verticalStart = 224;
        if(horizontalStart < 0) horizontalStart = 0;
        if(verticalStart < 0 ) verticalStart = 0;
    end
endmodule
```

```
module clockdivider #(parameter N = 1)
                     (input logic clk,
                     input logic reset,
                     output logic q);
    always_ff@(posedge clk, posedge reset)
        if(reset) q <=0;
        else      q <=q+1;
endmodule
```

```
1   module counter #(parameter N=8)
2
3       (input logic clk,
4       input logic reset,
5       output logic [N-1:0] q);
6
7
8   always_ff@(posedge clk, posedge reset)
9       if (reset)
10          begin
11              q <= 0;
12          end
13      else
14          begin
15              q <= q+1;
16          end
17  endmodule
18
```

```
1   module horizontalclock #(parameter lowtime = 95, parameter hightime = 799) (input logic clk,
    input logic reset, output logic horizontalSync);
2
3       logic [9:0] counter;
4
5       always_ff @(posedge clk, posedge reset)
6           if(reset)
7               begin
8                   counter <= 0;
9                   horizontalSync <= 1;
10              end
11          else
12              begin
13              counter <= counter + 1;
14              if (counter < lowtime)
15                  horizontalSync <= 0;
16              else if (counter < hightime)
17                  horizontalSync <= 1;
18              if (counter > (hightime-1))
19                  begin
20                      counter <= 10'd0;
21                      horizontalSync <= 0;
22                  end
23              end
24  endmodule
```

```
module shiftInterpret #(parameter N = 8) (input logic[N-1:0] data, input logic clk, input
logic reset, output logic[2:0] q, output logic NS);

    logic[19:0] count = 0;

    always_comb
        begin
            case(data)

                8'b01101111: q=1; //up
                8'b10001111: q=2; //right
                8'b01110111: q=3; //down
                8'b11001111: q=4; //left
                default: q=0;
            endcase
        end

    always_ff @(posedge clk) begin
            if(count < 253124)
                begin
                    NS <= 0;
                    count <= count + 1;
                end
            else
                NS <= 1;
                count <= 0;
        end


endmodule
```

```
module shiftregister  #(parameter N = 8)
                (input logic clk,
                 input logic reset,
                 input logic sin,
                 output logic [N-1:0]q);

always_ff@(posedge clk, posedge reset)
    if(reset)
        q<=0;
    else
        q<= {q[N-2:0], sin};

endmodule
```

```
module sprite2 (
    address,
    clock,
    q);

    input   [15:0]  address;
    input     clock;
    output  [15:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [15:0] sub_wire0;
    wire [15:0] q = sub_wire0[15:0];

    altsyncram  altsyncram_component (
                .address_a (address),
                .clock0 (clock),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_a ({16{1'b1}}),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_a (1'b0),
                .wren_b (1'b0));
    defparam
        altsyncram_component.address_aclr_a = "NONE",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "../../../../../mifMaker/mifData.mif",
        altsyncram_component.intended_device_family = "MAX 10",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 65536,
        altsyncram_component.operation_mode = "ROM",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "CLOCK0",
        altsyncram_component.widthad_a = 16,
        altsyncram_component.width_a = 16,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

```
1   module synchronizer(input logic clk,
2                       input logic d,
3                       output logic q);
4
5       logic n1;
6
7       always_ff@(posedge clk)
8           begin
9               n1 <= d;
10              q <= n1;
11          end
12  endmodule
```

```
1   module xbitMux #(parameter width = 3) (input logic [width-1:0] data, input logic select,
    output logic [width-1:0] out);
2
3       assign out = select ? data : 0;
4
5   endmodule
```

```
module stateControl (input logic clk, input logic reset, input logic IR, input logic PS, input logic start, input logic data,
                     input logic decode, output logic startEn, output logic dataEn, output logic decodeEn, output logic completeReset);
    enum logic[1:0] {S0, S1, S2, S3} state, nextstate;
    logic[15:0] count;
    logic buffer;

    always_ff @(posedge clk, posedge reset)
    begin
       if (reset)
       begin
          state <= S0;
          completeReset <= 1;
          buffer <= 0;
          count <= 0;
       end
       else
       begin
          state <= nextstate;


          if(state == S0)
          begin
             if(IR == 0) count <= count + 1;
             else count <= 0;

             if(count >= 56250) begin buffer <= 1; count <= 0; end
             else buffer <= 0;

             completeReset <= 1;
             decodeEn <= 0;
             startEn <= 0;
             dataEn <= 0;
          end
          else if(state == S1)
          begin
             completeReset <= 0;
             startEn <= 1;

             decodeEn <= 0;
             dataEn <= 0;
          end
          else if(state == S2)
          begin
                completeReset <= 0;
                startEn <= 0;
                dataEn <= 1;
             decodeEn <= 0;
          end
          else if(state == S3)
          begin
                completeReset <= 0;
                startEn <= 0;
              decodeEn <= 1;
                dataEn <= 0;
          end
          else
          begin
             completeReset <= 0;
             decodeEn <= 0;
             startEn <= 0;
             dataEn <= 0;
          end

       end
    end

    always_comb
    begin
       case(state)
          S0: nextstate <= (buffer == 1) ? S1: S0;
          S1: begin nextstate <= (PS == 1) ? S0: S1; nextstate <= ((start == 1) & (PS == 0)) ? S2: S1;  end
          S2: nextstate <= (data == 1) ? S3: S2;
          S3: nextstate <= (decode == 1) ? S0: S3;
          default: nextstate <= S0;
       endcase


    end

endmodule
```