A Box Addressable Black and White VGA Monitor Driver

Final Project Report
December 9, 1999
E157

Michael Cope and Philip Johnson

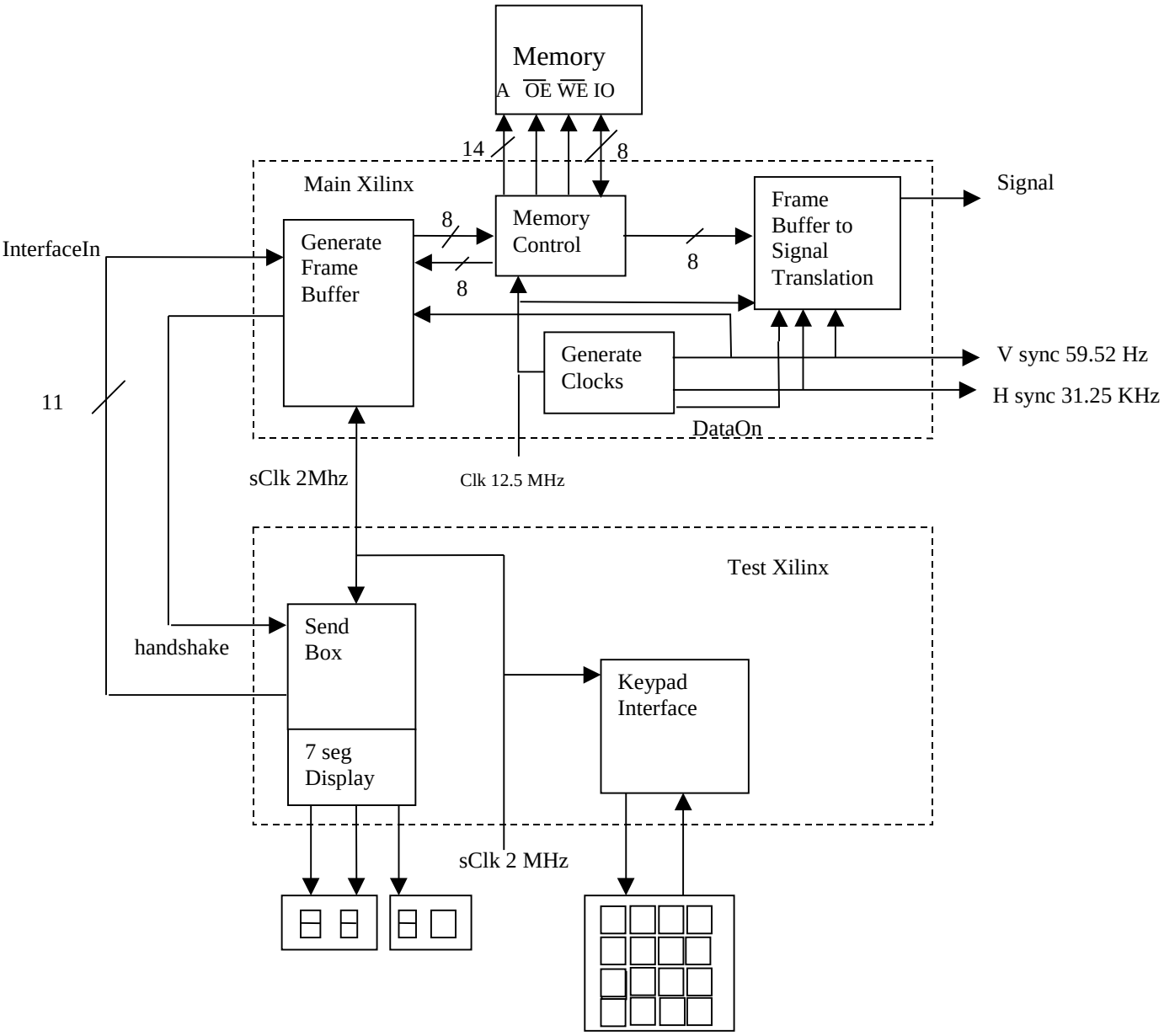
Abstract:

We implemented a black and white driver for a standard VGA monitor. An FPGA running at 12.5 MHz reads from an external 32K SRAM and produces the timing signals to display the contents of the SRAM on the monitor. Between vertical refreshes, the FPGA accepts coordinates and color of a new box to draw on the screen over a parallel interface. The contents of this box are written to the SRAM at the appropriate addresses. A second FPGA is used for accepting keypad input and testing the parallel interface.

All images are persistent until they are explictly erased. The effective resolution of this system is 312x240 with a refresh rate of 60 boxes/sec.

Introduction

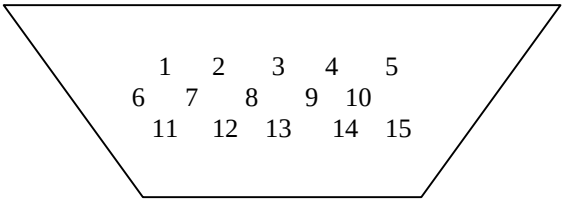
We wanted to make a CRT driver that someone could use to implement Pong or some such simple block based game. To that end we implemented a driver on an FPGA board that accepts block coordinate inputs, and a color bit indicating black or white, then draws that box to a frame buffer stored in memory. The memory is read to produce the image on the screen. In order to test this driver we implemented a test board which accepts keypad input and sends the box data over the interface.



New Hardware

VGA Adapter

1 Signal (R)
2 Signal (G)
3 Signal (B)
4 No pin
5 N/C
6 Gnd (R)
7 Gnd (G)
8 Gnd (B)
9 No pin
10 N/C
11 N/C
12 N/C
13 HSync
14 VSync
15 N/C



All Signals are tied
All Gnds are tied

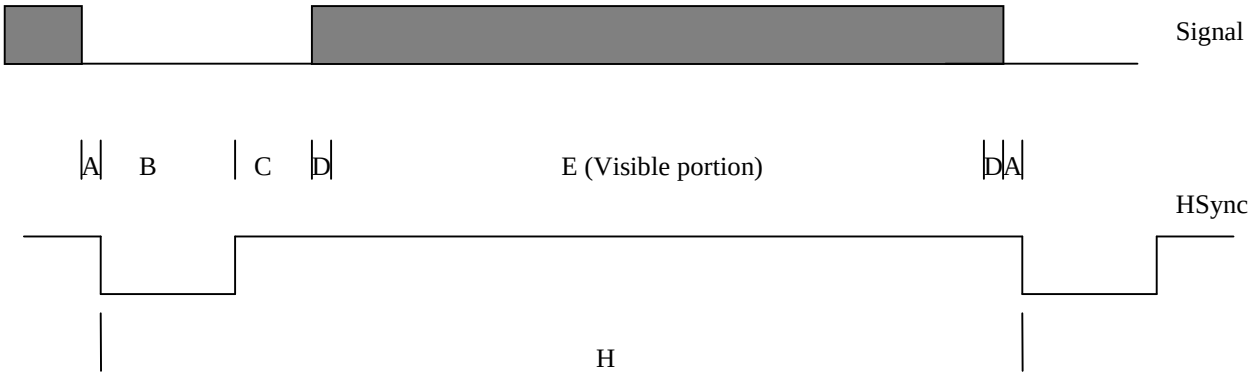
VGA Monitor

VGA signals are fairly simple. The critical components are Signals R, G, and B which are analog voltages cooresponding to the intensity of Red, Green, and Blue in the display. In our design, these are tied to a single Signal line and has the value of 0 or 2 V. For standard 640x480 VGA, Hsync and Vsync are negative polarity TTL pulse trains running at 31 KHz and 60 Hz respectively. Figures 2 and 3 illustrate the timing relationships between the signal components. Note the presence of "porches" around the Hsync and Vsync pulses during which Signal must be zero. Table 1 gives the timing information for these scans. Note that horizontal times are in terms of a clock cycle and vertical times are in terms of horizontal periods

Table 1: Comparative timing (cf. [1])

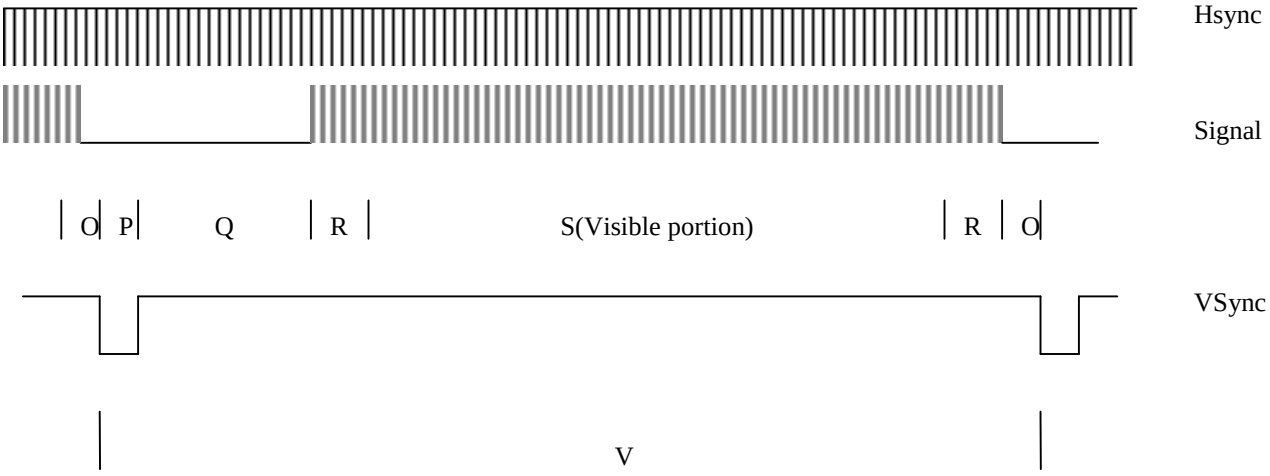
	25.175 MHz clock (VGA Standard)	12.5 MHz clock (implemented)
[clock cycles] time		
Hsync period	[800] 31.778 us	[400] 32 us
H front porch	[8] 317.775 ns	[18] 1.44 us
Hsync pulse length	[96] 3.813 us	[48] 3.84 us
H back porch	[40] 1.589 us	[22] 1.76 us
H Border	[8] 317.775 ns	[0] 0 ns
H active video	[640] 25.422 us	[312] 24.96 us
[scans] time		
Vsync period	[525] 16.683 ms	[525] 16.8 ms
V front porch	[2] 63.555 us	[6] 64 us
Vsync pulse length	[2] 63.555 us	[2] 64 us
V back porch	[25] 794.439 us	[29] 800 us
V Border	[8] 254.220 us	[0] 256 us
V active video	[480] 15.253 ms	[480] 15.4 ms
Vsync, Hsync polarity	-,-	-,-
H frequency	31.47 KHz	31.25 KHz
V frequency	59.94 Hz	59.52 Hz

Figure 1: Horizontal Scan Timing Diagram



A: Horizontal front porch
B: Hsync pulse length
C: Horizontal back porch
D: Horizontal border (video signals which are not displayed, may be combined with porch)
E. Active horizontal video time
H. Hsync period

Figure 2: Vertical Frame Timing Diagram



O: Vertical front porch
P: Vsync pulse length
Q: Vertical back porch
R: Vertical border (video signals which are not displayed, may be combined with porch)
S. Active vertical video time
V. Vsync period

Function Generator

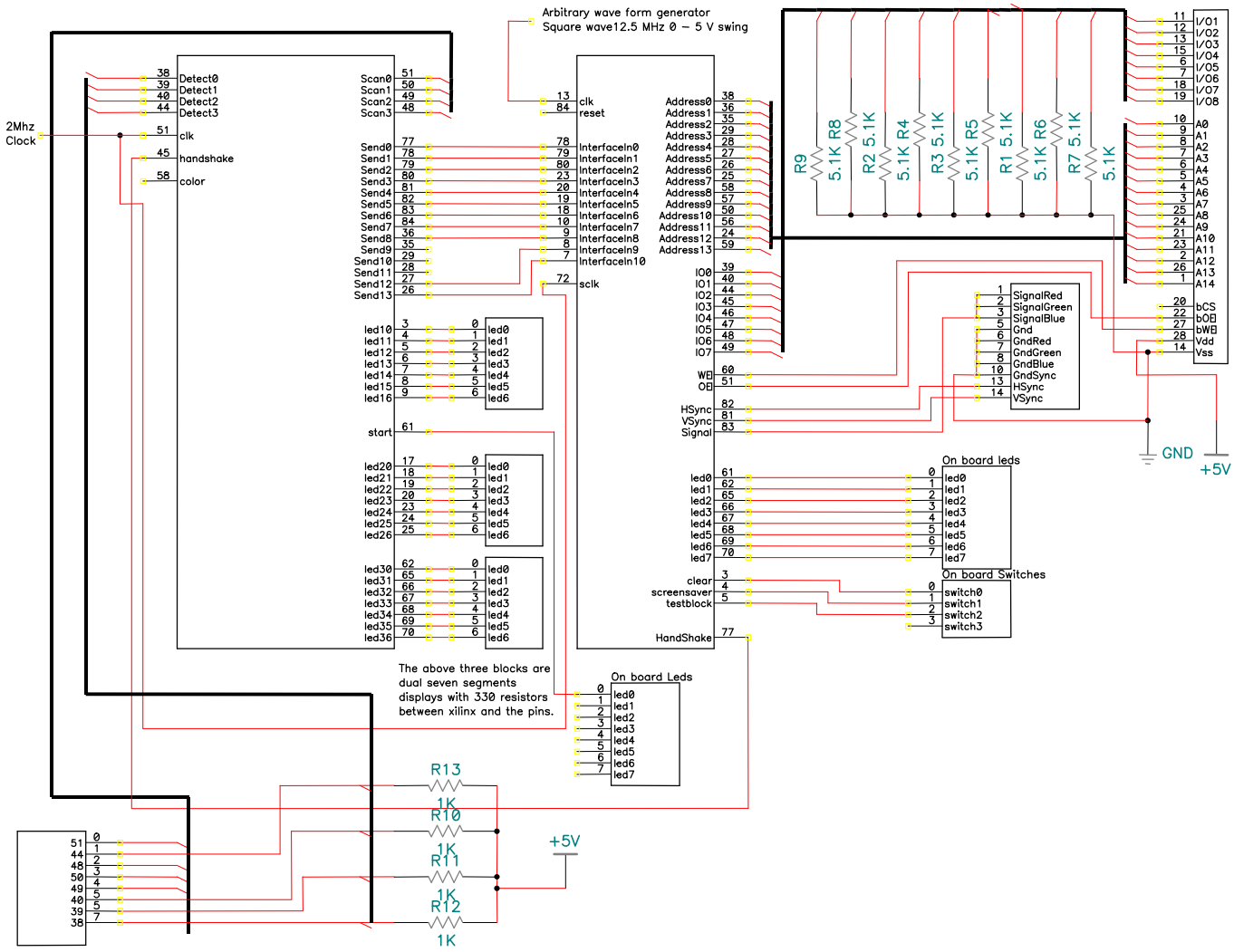
We were unable to find a 12.5 MHz clock, so we used an HP function generator to produce a 0-2.5V 12.5 MHz square wave and input the clock to the MCLK pin (13) of the FPGA board with the jumper off. This is not a clean solution. It would be much better to find a 12.5 or 25.175 MHz clock.

SRAM

We chose a 32Kx8 asynchronous SRAM for a frame buffer (see reference [2] for datasheet). It has 15 address lines, 8 bidirectional data lines and 3 control lines, CS_bar, OE_bar, and WE_bar. CS selects the chip, and hence should be tied low. OE_bar enables output, hence it should be low before a read and high before a write. WE_bar enables writes and hence should be low before a write and high before a read (see truth table on page 2 of datasheet).

Complete timing information is available on the datasheet, but the time for a complete read or write cycle is 15ns or less including enabling, addressing and hold times. This is much smaller than the 80ns clock cycle so one can do a read or a write in a single cycle if you set all your control lines and address on the rising edge and read or write on the trailing edge. You need to be careful, however, since there are variable delays in the bus due to varying lengths of wires to and varying impedance of the paths. You also need to be careful about the capacitance of the protoboard which lengthens the rise and fall times of the signals. For these reasons, and because we couldn't use a data stream any faster, our actual design takes 3 cycles to read, 3 cycles to write, with 1 cycle in between where both WE_bar and OE_bar are high.

Schematics


Arbitrary wave form generator
Square wave12.5 MHz 0 – 5 V swing
2Mhz
Clock
Detect0
Detect1
Detect2
Detect3
clk
handshake
color
Scan0
Scan1
Scan2
Scan3
Send0
Send1
Send2
Send3
Send4
Send5
Send6
Send7
Send8
Send9
Send10
Send11
Send12
Send13
clk
reset
InterfaceIn0
InterfaceIn1
InterfaceIn2
InterfaceIn3
InterfaceIn4
InterfaceIn5
InterfaceIn6
InterfaceIn7
InterfaceIn8
InterfaceIn9
InterfaceIn10
sclk
HandShake
Address0
Address1
Address2
Address3
Address4
Address5
Address6
Address7
Address8
Address9
Address10
Address11
Address12
Address13
IO0
IO1
IO2
IO3
IO4
IO5
IO6
IO7
WE
OE
HSync
VSync
Signal
R9
5.1K
R8
5.1K
R2
5.1K
R4
5.1K
R3
5.1K
R5
5.1K
R1
5.1K
R6
5.1K
R7
5.1K
5.1K
I/O1
I/O2
I/O3
I/O4
I/O5
I/O6
I/O7
I/O8
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
bCS
bOE
bWE
Vdd
Vss
SignalRed
SignalGreen
SignalBlue
Gnd
GndRed
GndGreen
GndBlue
GndSync
HSync
VSync
GND
+5V
led10
led11
led12
led13
led14
led15
led16
start
led20
led21
led22
led23
led24
led25
led26
led30
led31
led32
led33
led34
led35
led36
led0
led1
led2
led3
led4
led5
led6
led7
On board leds
clear
screensaver
testblock
On board Switches
switch0
switch1
switch2
switch3
The above three blocks are dual seven segments displays with 330 resistors between xilinx and the pins.
On board Leds
R13
1K
R10
1K
R11
1K
R12
1K
+5V

FPGA Design

Refer to System block diagram on page 1 and the code in the Appendix

Generation of the HSync and VSync signals: (gensync.v)

HSync is based off of the 12.5Mhz clock. The module counts from zero to 397 for a frequency of 33kHz. To get the duty cycle desired for HSync an if statement checks if the value of the counter is 0 to 47 during which it will set HSync low, otherwise it is high.

To account for the porches and border time Hdata is set high during the time a signal can be sent in regards to HSync. The signal is low between 394 and 71 for a total low time of 2.35μsec. This includes front and back porches, HSync time, and the border.

The line data is set high when it is time to send a signal. This signal is a combination of the Hdata signal and the VSync counter. In regards to the VSync line, this signal is low for two cycles of HSync before the VSync pulse, for 2 HSync cycles during the VSync pulse, and 33 HSync cycles for the back porch and border. This period is equal to 37 HSync cycles.

The next always block sets VSync. VSync is based off of HSync, since HSync generally is high and pulses low the triggering is off of the negative edge of HSync. The counter counts up to 525 and then resets. VSync is low only when all bits are zero except for the LSB, which can have a value of one or zero.

Memory Arbitration and display: (gensignalswitch.v)

The heart of this module is an 8 state FSM that increments on each clock during DataOn. The states are
`read_addr, read_data0, read_data1, idle_1, write_addr, write_data0, write_data1, idle_2.`
The memory address changes during read_addr. Memory is read during read_data1. Memory is written during write_data1. OE_bar is low during the read_* states enabling output from the memory. WE_bar is low during the write_* states enabling writing to the memory. When WE_bar is high, the FPGA sets the memory bus to high Z. Since we have extra bits to play with on the memory address, the memory address is simply {vertical coordinate, horizontal coordinate/8} which simplifies the blockdecoder's job.

Once we have read memory in, we store it to a second buffer and display it to the screen by indexing the 8 bit memory buffer with a counter which contains the horizontal position.

When DataOn is off, indicating that we should not display to the screen, the buffers are reset, and the memory address is pointed to an empty byte at 7FFF.

Block decoding: (blockdecoder.v)

Each bit of memory is checked as it is read to determine if it is inside the block that is in the block buffer. If so, the bit is set to the color of the block, if not it is written back unchanged. This process requires 18 comparisons and uses a lot of area on the chip for the adders. It also means that only one block can be processed per Vsync for a total of 60 boxes per second, which is not a lot. A better solution might be to generate different addresses for writes than reads and have the block decoder process data as fast as it can. This would mean that small blocks would be faster and large blocks would take no additional time. This would not be too difficult to implement. The write_addr state in gensignalswitch anticipates this need.

Parallel Interface In: (blockdecoder.v)

This is a combination of two FSM. The first is responsible for the generation of the handshake signal. The handshake goes high when VSync goes low and goes low again if either data is being received or the video board starts sending signals to the screen again. If it received data, it is necessary to wait until VSync goes high again before returning to the initial condition so that handshake does not go high twice in one VSync period.

The second FSM is basically a counter that starts when it detects the box corner bit in the parallel interface and the handshake signal is high. The first state sets the parallel interface information to the first location in the Xilinx onboard memory, the second parallel interface word to the second locations and so

on. The fifth state of the state machine is a wait state, which is not necessary any more (the board was demonstrated with it in, so it is documented). The state machine then resets. The memory is ready before the first signal period of HSync.

Parallel interface for the second Xilinx

A fundamental building block for this unit is the lab 4 design of a debounce FSM. It is slightly modified from original form. The two major changes are a third digit and a signal that is high when 3 digits have been rotated out and 3 new digits have come in. This is the module GetKeyPress that is called at the top level.

In order to interface with the parallel interface on the video board, the test board needs to generate 4 14bit words (12 bit is enough but the input is hex + 2 bits for the box corner bit and the color bit) and store this until the video board is ready to accept. When the video board is ready to accept, it sets handshake signal high and the test board sends the 4 words, at one word per 2Mhz clock tick.

This design requires a finite state machine and two counters. The first counter, which is in the first always block, collects the 3 hex characters and appends the color bit and the box corner bit every time the ready signal comes from the lab four building block. When this has four new 14bit words and the system detects a handshake signal, the finite state machine moves to the next value. Since the handshake signal comes much more frequently than it is possible to enter even one digit, a separate state for handshake received is not required in this particular design. With the new value of the FSM, the Send counter starts and sends the data with the 2Mhz clock. When this is complete, the FSM moves to a new state. In this state the FSM is waiting for the system to start get new data. This involves waiting for the data collection counter to go to zero and handshake to go low (the handshake condition is not truly necessary because handshake shuts off as soon as data incoming is detected).

This description covers the major functionality of the block. In addition, start is supposed to light the first led when the system is ready to read in the first x coordinate. This does not seem to work. The SevenSegDis module calls are used to display the data is the memory; this also does not seem to be working properly.

There is a flaw in the design: the data collection only seems to work intermediately. As of yet the fault has not been tracked down. This means that this interface only sometimes produces the results that would be expected on the screen.

Results

The basic goal of creating a video interface for future E157 students was achieved. The parallel interface was tested and while the sending unit did not function correctly, in the limited testing of the interface on the video handling Xilinx no problems were found. While the general goal was achieved several modifications were made along the way. Most of these were from gained insight into the working of standard VGA.

The clock rate was moved from 2Mhz to 12.5Mhz. This was necessary in order to get the resolutions of interest. Basically, there were not 256 clock ticks between the HSync pulses with a 2Mhz clock, which means that a resolution of 256 is not possible at 2Mhz.

Another change is the resolution. The final design uses 312 x 240, this makes the pixels square as compared to the original design that would have stretched pixels. This is approximately half of the standard 640 x 480. The difference from half is because in the present form, memory reading and generation of the signal start at the same time for a give horizontal line. This makes it impossible to read the data for the first 8 bits. This would be something to correct in latter versions of the board.

Because of this change the parallel interface had to be change to accommodate the extra bit needed for the horizontal line. On the Xilinx handling video this changed required 1 extra bit, on the test board this change caused enlargement by 4 bits because the input to the test board is hexadecimal.

In addition, a screen saver and clear switches were added to the video board. The clear switch assisted with debugging and probably will be useful to future E157 students that might pursue this work. The screen saver gives a graphic demonstration that the board is working and the speed at which it can draw new boxes into the memory.

References

[1] http://www.hut.fi/Misc/Electronics/docs/pc/vga_timing.html
[2] http://www.winbond.com/sheet/257a-12.pdf

Parts List

Part	Source	Vendor Part #	Price
Winbond W24257A 32Kx8-12 Cache 5 Volt 28 Pin static memory DIP package	Fry's	?Need to look on receipt	$2.59
DHS-15 S 15 Pin female high density solder cup d-sub	Fry's	1424214	$1.99
VGA Monitor	Stock Room		
HP Function Generator 33120A	Stock Room		$1500

Appendices

Main Driver

Toplevel.v

```
module TopLevel (clk, HSync, VSync, Signal, reset, OE, WE, address, IO, InterfaceIn, handshake,
                sclk,led, clear,screensaver,testblock) ;

input          clk ;          //12.5 MHz clock
input          reset;
input   [10:0] InterfaceIn;   //block coordinate input
input          sclk;          //block input clock

output         handshake;     //set high indicates ready to receive data
output         HSync ;
output         VSync ;
output         Signal ;
output         OE, WE;        //OE_bar and WE_bar to memory
output  [13:0] address;       //Current memory address

inout   [7:0]  IO;            //8 bit memory bus
output  [7:0]  led;           //debugging leds

input          clear;         //set to 1 to clear memory (switch 1)
input          screensaver;   //set to 1 to draw random blocks to screen (switch 2)
input          testblock;     //set to 1 to draw standard block to screen (switch 3)

wire           H,V;
wire           DataOn;
wire           pixel_clk;
wire    [7:0]  data_out;
wire    [2:0]  scan_state;
wire    [7:0]  mem_data;

GenSyncs GenSyncs1(clk, H, V, reset, DataOn); //generate monitor timing signals

//receive and decode block
BlockDecoder BD1(clk, InterfaceIn, address, mem_data, data_out,
                reset,  DataOn, VSync, sclk, handshake,led,clear, screensaver,testblock);

//generate Signal to monitor and arbitrate memory
GenSignalSwitch GenSignal1(V, DataOn, Signal, clk, address, IO,
                           OE, WE, mem_data, data_out);
assign HSync = H;
assign VSync = V;

endmodule
```

GenSync.v

```
module GenSyncs (clk, HSync, VSync, reset, Data) ;

input  clk ;
input  reset ;
output HSync ;
output VSync ;
output Data ;

//12.5 Mhz clk period = 0.00000008

//31468.31 Hz
reg    [8:0]  slowdownforHsync;
reg    [9:0]  slowdownforVsync;
reg           HSync;
reg           HData;  //High when according to Hsync data is ready to flow
reg           VSync;
reg           Data;

always @ (posedge clk)
       begin
              slowdownforHsync = slowdownforHsync + 1;
              if ((slowdownforHsync == 9'b110001101) || (reset == 1'b1))
slowdownforHsync = 0;

              if ((slowdownforHsync >= 0) && (slowdownforHsync <= 9'b000101111)) HSync =
0;
              else HSync = 1;
              if ((slowdownforHsync <= 9'b001000111) || (slowdownforHsync >=
9'b110000111)) HData = 0;
              else HData = 1;

              if (((slowdownforVsync < 10'b1000001011) && (slowdownforVsync >
10'b0000100100)) && HData) Data = 1;
              else Data = 0;  //
       end

always @ (negedge HSync)
       begin
              slowdownforVsync = slowdownforVsync + 1;
              if ((slowdownforVsync == 10'b1000001101) || (reset == 1'b1))
slowdownforVsync = 0;

              VSync = |slowdownforVsync[9:1];
       //     if (((slowdownforVsync <= 10'b1000001011) && (slowdownforVsync >=
10'b0000100101)) && HData) Data = 1;
       //     else Data = 0;
       end



endmodule
```

GenSignalSwitch.v

```
module GenSignalSwitch(VSync, DataOn, Signal, clk, address_reg, IO, OE, WE,
byte_buffer,data_out);

input          VSync ;
input          DataOn ;
output         Signal ;
input          clk ;
output  [13:0] address_reg;
reg     [13:0] address_reg;          //contains current address being sent to memory
inout   [7:0]  IO;                   //bidirectional data bus to memory
output  [7:0]  byte_buffer;          //the most recent byte read from memory
reg     [7:0]  current_buffer;       //the next to most recent byte read from memory
reg     [8:0]  pixel;                //the horizontal coordinate of the current pixel
reg     [8:0]  h_count;              //the veritcal coordinate of the current pixel
reg     [7:0]  byte_buffer;
reg     [2:0]  scan_state;
reg     [2:0]  next_scan_state;
wire           pixel_clk;
wire    [7:0]   data_in;
wire    [13:0]  address;
input   [7:0]   data_out;
output OE;                           //OE_bar really, low enables memory output
output WE;                           //WE_bar really, low enables memory writes

//bookkeeping of memory access states
parameter read_addr  = 3'b000;
parameter read_data0 = 3'b001;
parameter read_data1 = 3'b011;
parameter idle_1     = 3'b010;
parameter write_addr = 3'b110;
parameter write_data0= 3'b111;
parameter write_data1= 3'b101;
parameter idle_2     = 3'b100;



always @ (posedge clk)
        begin
                if (DataOn) begin
                        pixel <= pixel + 1;
                        scan_state <= next_scan_state;
                        //change address and read memory into buffer in appropriate states
                        case (next_scan_state)
                                read_data1:    byte_buffer <= data_in;
                                read_addr:     begin
                                                       current_buffer <= byte_buffer;
                                                       address_reg <= address;
                                               end
                        endcase
                end

                //clear buffers between scans
                else   begin
                        scan_state <= next_scan_state;
                        byte_buffer <= 8'b0;
                        current_buffer <= 8'b0;
                        pixel <= 0;
                        address_reg <= {14'b1};
                end
        end
```

```
always @ ( posedge DataOn or negedge VSync)
	begin
		if (!VSync)
			h_count = 0;
		else
			h_count = h_count +1 ;
	end


always @ (scan_state or DataOn)	begin
	if (DataOn) begin
		case (scan_state)
			read_addr:	next_scan_state <= read_data0;
			read_data0:	next_scan_state <= read_data1;
			read_data1:	next_scan_state <= idle_1;
			idle_1:		next_scan_state <= write_addr;
			write_addr:	next_scan_state <= write_data0;
			write_data0:	next_scan_state <= write_data1;
			write_data1:	next_scan_state <= idle_2;
			idle_2:		next_scan_state <= read_addr;
			default:	next_scan_state <= read_addr;
		endcase
	end
	else
		next_scan_state <= read_addr;
end



assign address = {h_count[8:1],pixel[8:3]};

assign Signal  =  current_buffer[pixel[2:0]] && DataOn;     //draw to screen if DataOn


//Enable output if we're reading or about to read, enable writes if we're writing or
about to write
assign OE	=
!((scan_state==read_addr)||(scan_state==read_data0)||(scan_state==read_data1));
assign WE	= !((scan_state==write_addr)||(scan_state == write_data0) || (scan_state
== write_data1));

assign data_in = (!OE) ? IO  : 8'bz;
assign IO	= (!WE) ? data_out : 8'bz; //tristate output if we're not writing



endmodule
```

blockdecoder.v

```
module BlockDecoder (clk, InterfaceIn, address, mem_data, data_out, reset, DataOn, VSync,
sclk,
                     handshake, led, clear, screensaver,testblock);

input          clk ;          //12.5 MHz
input  [10:0]  InterfaceIn;   //interface bus for block coordinates
input  [13:0]  address;       //current memory location
output [7:0]   data_out;      //data to be written to memory
input  [7:0]   mem_data;      //data read from memory
input          reset;
input          VSync;
input          DataOn;
input          sclk;          //slow (2 MHz) clock
output         handshake;     //set to indicate ready to receive data
output [7:0]   led;
input          clear;         //set to 1 to clear memory
input          screensaver;   //set to 1 to show random blocks
input          testblock;     //set to 1 to show a test block

wire   [5:0]   h_pos;         //horizontal coordinate of current memory byte divided by 8
wire   [7:0]   v_pos;         //vertical     "       "       "       "

wire   [7:0]   data1;
wire   [7:0]   data_out;

wire   [8:0]   ux1;           //upper left x coordinate of box
wire   [7:0]   uy1;           //upper left y coordinate of box
wire   [8:0]   lx1;           //lower right x coordintate of box
wire   [7:0]   ly1;           //lower right y coordinate of box


wire   [8:0]   h_pos_7;       //horizontal coordinates of pixels in current memory byte
wire   [8:0]   h_pos_6;
wire   [8:0]   h_pos_5;
wire   [8:0]   h_pos_4;
wire   [8:0]   h_pos_3;
wire   [8:0]   h_pos_2;
wire   [8:0]   h_pos_1;
wire   [8:0]   h_pos_0;


wire           v_pos_1;
wire           valid_1;
wire           color_1;

reg    [2:0]   block_input_state;
reg    [2:0]   next_block_input_state;
reg    [43:0]  block;
reg    [7:0]   data_out_reg;
reg            vsync_reg;
reg    [15:0]  random;        //random number register
wire           new_random_bit;


reg [1:0] PS;
reg [1:0] NS;
reg [2:0] prevstate;
reg [2:0] nextstate;

//minimum and maximum coordinates, could be used for error checking.  Currently not in
use
parameter h_min = 0;
parameter h_max = 312;
parameter v_min = 0;
parameter v_max = 240;
```

```
always @ (VSync or DataOn or PS or prevstate)
	case (PS)
		0 :	if (~VSync) NS = 1;
			else NS = 0;
		1 :	 if (DataOn || (prevstate == 3'b001)) NS = 2;
			else NS = 1;
		2 :	if (~VSync) NS = 2;
			else NS = 0;
		3 :	NS = 0;
	endcase

always @ (posedge sclk)
	begin
		prevstate <= nextstate;
		PS <= NS;
		if(screensaver) begin
			//make a new random block each Vsync
			if (vsync_reg != VSync) begin
				block[10]    <= 1;
				block[9:0]   <= random[9:0];
				block[18:11] <= random[15:9];
				block[30:22] <= random[9:0] + {random[2:0], random[12:9]};
				block[40:33] <= random[15:9] + {random[12:9], random[2:0]};
				vsync_reg = VSync;
			end
		end
		else
			//show a standard test block
			if (testblock) begin
				block[10:0]  <= 11'b11000000100;
				block[18:11] <=     8'b00000100;
				block[30:22] <=    9'b101010011;
				block[38:33] <=     8'b00010000;
			end

		else
			//receive data
			if (~prevstate[2] )
				case (prevstate[1:0])
				0 : if (InterfaceIn[10]) block[10:0] <= InterfaceIn[10:0];
				1 : block[21:11] <= InterfaceIn[10:0];
				2 : block[32:22] <= InterfaceIn[10:0];
				3 : block[43:33] <= InterfaceIn[10:0];
				endcase

	end

assign handshake = NS[0];

always @ (prevstate or InterfaceIn or handshake)
	case (prevstate)
	3'b000 :	if (InterfaceIn[10] && handshake) nextstate = 3'b001;
			else nextstate = 3'b000;
	3'b001 :	nextstate = 3'b010;
	3'b010 :	nextstate = 3'b011;
	3'b011 :	nextstate = 3'b101;
	3'b101 :	nextstate = 3'b000;
	default :	nextstate = 3'b000;
	endcase

//test whether pixels are in block
assign data1 = {((h_pos_7 >= ux1) && (h_pos_7 <= lx1) && v_pos_1),
		((h_pos_6 >= ux1) && (h_pos_6 <= lx1) && v_pos_1),
		((h_pos_5 >= ux1) && (h_pos_5 <= lx1) && v_pos_1),
		((h_pos_4 >= ux1) && (h_pos_4 <= lx1) && v_pos_1),
		((h_pos_3 >= ux1) && (h_pos_3 <= lx1) && v_pos_1),
		((h_pos_2 >= ux1) && (h_pos_2 <= lx1) && v_pos_1),
		((h_pos_1 >= ux1) && (h_pos_1 <= lx1) && v_pos_1),
		((h_pos_0 >= ux1) && (h_pos_0 <= lx1) && v_pos_1)};

//clear memory if clear is set
```

```
//Otherwise set the pixel to color_1 if the pixel is in the block
//Otherwise keep the pixel at the value stored in memory
assign data_out = (clear) ?   8'b0 :
                              {((data1[7]) ? color_1: mem_data[7] ),
                               ((data1[6]) ? color_1: mem_data[6] ),
                               ((data1[5]) ? color_1: mem_data[5] ),
                               ((data1[4]) ? color_1: mem_data[4] ),
                               ((data1[3]) ? color_1: mem_data[3] ),
                               ((data1[2]) ? color_1: mem_data[2] ),
                               ((data1[1]) ? color_1: mem_data[1] ),
                               ((data1[0]) ? color_1: mem_data[0] )};

//test whether pixels are between top and bottom of block.
assign v_pos_1 = (v_pos >= uy1) && (v_pos <= ly1) && valid_1;

//assign addresses for the position of the individual pixels
assign h_pos_7 = {h_pos, 3'b111};
assign h_pos_6 = {h_pos, 3'b110};
assign h_pos_5 = {h_pos, 3'b101};
assign h_pos_4 = {h_pos, 3'b100};
assign h_pos_3 = {h_pos, 3'b011};
assign h_pos_2 = {h_pos, 3'b010};
assign h_pos_1 = {h_pos, 3'b001};
assign h_pos_0 = {h_pos, 3'b000};

//get the vertical and horizontal positions from the address
assign {v_pos,h_pos} = address;

//assign names to block[] data to keep track of coordinates
assign ux1 = block[8:0];
assign uy1 = block[18:11];
assign lx1 = block[30:22];
assign ly1 = block[40:33];
assign color_1 = block[9];

//check if upper left coordinate indicator bit is set
assign valid_1 = block[10];

//debug by showing valid_1 and color bit followed by lower 6 bits of ux1
assign led     = {block[10:9],block[5:0]};

//15 bit maximal length linear feedback shift register psuedo random number generator
//repeats every 2^15-1 bits.  Taken from CDMA standard
always @ (posedge clk) begin
       random[15:0] <= {random[14:0], new_random_bit};
end

assign new_random_bit = (| random[15:0]) ? random[1] + random[4] + random[5] +
                                           random[6] + random[7] + random[11] +
                                            random[12] + random[13]
                                         : 1;

endmodule
```

Second Xilinx

Toplevel.v

```
module toplevel (Scan, Detect, Send, color, clk, led1, led2, led3, start, handshake) ;

output [3:0] Scan ;    //used for keypad detection
input [3:0] Detect ;   // "
output [13:0] Send ;   //parallel data lines
input color ;

input clk;
output [6:0] led1 ;
output [6:0] led2 ;
output [6:0] led3 ;
input          handshake;
output start;


wire   [11:0]  s; //8bit bit number received from keypad
wire           ready;
reg    [55:0]  box;
reg    [55:0]  readytogo;
reg    [1:0]   cornercounter;
reg    [2:0]   Sendcounter;
reg    [13:0]  Send;
reg    [13:0]  resetholder;

reg  [1:0]     prevstate;
reg  [1:0]     nextState;
reg    [1:0]   address;



wire   resetcounter;
GetKeyPress GetKeyPress1(Detect, Scan, s, ready, clk, slowclk);

SevenSegDis sevensegdis1(box[3:0],led1);
SevenSegDis sevensegdis2(box[7:4],led2);
SevenSegDis sevensegdis3(box[11:8],led3);


assign start = |cornercounter;

assign resetcounter = (prevstate ==1);

always @ (posedge ready or posedge resetcounter)
       if (resetcounter)
              begin
                     cornercounter = 2'b00;
                     box[13:0] = 14'b0;
                     box[27:14] = 14'b0;
                     box[41:28] = 14'b0;
                     box[55:42] = 14'b0;
              end
       else
              begin
                     case (cornercounter)
                            0 : box[13:0] = {1'b1,color,s};
                            1 : box[27:14] = {0,color,s};
                            2 : box[41:28] = {0,color,s};
                            3 : box[55:42] = {0,color,s};
                     endcase

                     if (cornercounter != 2'b11) cornercounter[1:0] = cornercounter[1:0]
+ 1;
                     else readytogo = box;
              end

always @ (posedge clk)
```

```
        begin
                prevstate <= nextState;
                if (nextState == 1)    // changed prevstate to nextState
                        case (Sendcounter[1:0])
                        0 : Send <= readytogo[13:0];
                        1 : Send <= readytogo[27:14];
                        2 : Send <= readytogo[41:28];
                        3 : Send <= readytogo[55:42];
                        default : Send <= 14'b0;
                        endcase
                else Send <= 14'b0;
                if (nextState == 2) Sendcounter <= 0;
        end

always @ (prevstate or Sendcounter or handshake or cornercounter or readytogo or box)
        case (prevstate)
                0 : if (cornercounter == 2'b11 && (readytogo == box) && handshake)
nextState = 1;
                    else nextState = 0;
                1 : if (Sendcounter == 3'b100) nextState = 2;
                    else nextState = 1;
                2 : if (~handshake && cornercounter == 2'b00) nextState = 2'b00;
                    else nextState = 2'b10;
                default : nextState = 0;
        endcase

endmodule
```

slowdownclock.v

```
module slowdownclk (clk, slowclk);

input clk;
output slowclk;

reg [8:0] counter;

always @ (posedge clk)
        counter = counter + 1;
assign slowclk = counter[8];

endmodule
```

sevensegdis.v

```
module SevenSegDis (s, led) ;

input [3:0] s ;
output [6:0] led ;

reg [6:0] led;

//I assume s = [Detect,Scan]
//scan are the cols
always @ (s)     //7 bits for case when led lights on:
    case (s)     //low        //high
         0: led = 7'b0000001 ; //7'b1111110 ; 0
         1: led = 7'b1001111 ; //7'b0110000 ;1
         2: led = 7'b0010010 ; //7'b1101101 ;2
         3: led = 7'b0000110 ; //7'b1111001 ;3
         4: led = 7'b1001100 ; //7'b0110011 ;4
         5: led = 7'b0100100 ; //7'b1011011 ;5
         6: led = 7'b0100000 ; //7'b1011111 ;6
         7: led = 7'b0001111 ; //7'b1110000 ;7
         8: led = 7'b0000000 ; //7'b1111111 ;8
         9: led = 7'b0000100 ; //7'b1111011 ;9
        10: led = 7'b0001000 ; //7'b1110111 ;a
        11: led = 7'b1100000 ; //7'b0011111 ;b
        12: led = 7'b1110010 ; //7'b0001101 ;c
        13: led = 7'b1000010 ; //7'b0111101 ;d
        14: led = 7'b0110000 ; //7'b1001111 ;e
        15: led = 7'b0111000 ; //7'b1000111 ;f
        default: led = 7'b0000001;
    endcase
```

```
endmodule
keypad
module KeypadTo4bit (s, number) ;

input [7:0] s ;
output [3:0] number ;

reg [3:0] number;

//I assume s = [Detect,Scan]
//scan are the cols

always @ (s)
    case (s)
        8'b10111110: number = 4'b0000;
        8'b01110111: number = 4'b0001;
        8'b10110111: number = 4'b0010;
        8'b11010111: number = 4'b0011;
        8'b01111011: number = 4'b0100;
        8'b10111011: number = 4'b0101;
        8'b11011011: number = 4'b0110;
        8'b01111101: number = 4'b0111;
        8'b10111101: number = 4'b1000;
        8'b11011101: number = 4'b1001;
       8'b01111110: number = 4'b1010;
       8'b11011110: number = 4'b1011;
       8'b11100111: number = 4'b1100;
       8'b11101011: number = 4'b1101;
       8'b11101101: number = 4'b1110;
       8'b11101110: number = 4'b1111;
       default: number = 4'b0000;
    endcase

endmodule
```

getkeypress.v

```
module GetKeyPress (Detect, Scan, s, ready, clk , slowclk);

input   [3:0]   Detect;
output  [3:0]   Scan;
output  [11:0]  s;
output          ready;
input           clk;
//output              slowclk;

reg     [1:0]   PrevState;
reg     [1:0]   NextState;
reg     [11:0]  s;
reg     [1:0]   readycounter;
reg     [1:0]   Channel;
reg     [3:0]   Scan;

wire    [3:0]   first;
output          slowclk;
wire    [3:0]   CScan;

slowdownclk slowclk1 (clk, slowclk);

always @ (posedge slowclk)
       begin
              PrevState = NextState;
              if (NextState == 2'b11)
                     begin
                            s[11:8] = s[7:4];
                            s[7:4] = s[3:0];
                            s[3:0] = CScan;
                            readycounter = readycounter + 1;
                            if (readycounter == 2'b11) readycounter = 2'b00;
                     end
```

```
                if (NextState == 2'b00)
                        begin
                                Channel = Channel + 1;
                        end
        end

KeypadTo4bit K24 ({Detect,Scan},CScan);

assign first = s[3:0];
assign ready = readycounter[1];

always @ (Detect or CScan or first or PrevState)
        case (PrevState)
                0: if (Detect == 4'b1111) NextState = 2'b00;
                   else NextState = 2'b11;
                1: NextState = 2'b00; //Should never reach this state
                2: if (Detect == 4'b1111) NextState = 2'b00;
                   else if (CScan == first) NextState = 2'b10;
                        else NextState = 2'b11;
                3: if (Detect == 4'b1111) NextState = 2'b00;
                   else if (CScan == first) NextState = 2'b10;
                        else NextState = 2'b11;
                default : NextState = 2'b00;
        endcase

always @ (Channel)
        case (Channel)
        0 : Scan = 4'b1110;
        1 : Scan = 4'b1101;
        2 : Scan = 4'b1011;
        3 : Scan = 4'b0111;
        default : Scan = 4'b1110;
        endcase

endmodule
```