The Caricatron

6.111 Final Project Report, May 13, 2004

Punya Biswal, Finale Doshi, Javier Velez

1 Introduction

We designed and implemented a system to capture a digital frame and output a caricaturized version of the image. The user could take multiple snapshots until he or she was pleased with the image. Next, edges were found using a Laplacian of Gaussian filter and lines extracted lines from the edges. Designed but not fully implemented was a plan to exaggerate the curvatures of the lines to give the output a cartoon-like look and to print the image to a standard post-script printer through a parallel port.

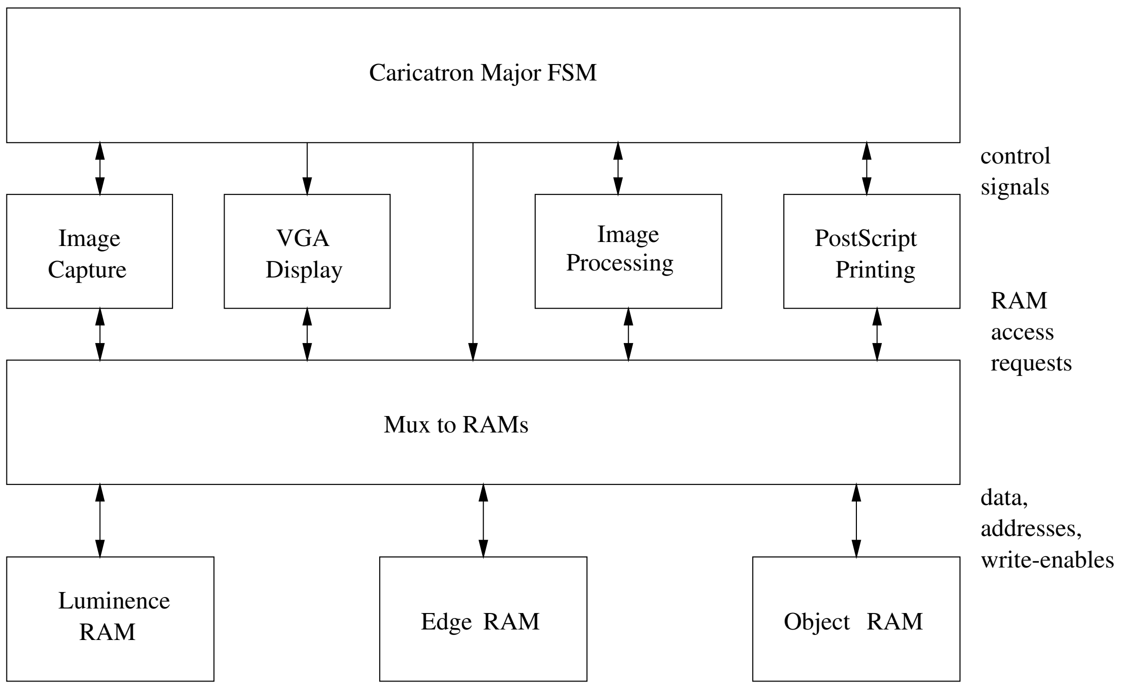


Figure 1: Overall Block Diagram of System

2 Overview

Figure 1 provides a general overview of how we implemented our design. The Caricatron FSM sent begin and received end signals from each of the project modules that captured the image, displayed an image, detected edges, extracted lines, or printed the image. It also sent control signals to a multiplexer that acted as a bus arbiter for all the modules requesting access to our three RAMS. In cases where only one module needed access to part of a RAM—for example, the only the video capture needed to write data to the luminence RAM—those signals were sent directly to the RAMs.

The first RAM stored 8-bit luminence values from the video capture of a 320 by 240 image. The next stored 16-bit edge detect values for the same image. The final RAM contained space for up to 2048 68-bit objects. Each object consisted of four points, two endpoints and two control points, to describe a curve. The two image RAMs were addressed by an {x,y} value and provided the appropriate clock for the requesting module (all 27MHz except for the VGA which required a 31.5MHz clock). All RAMs were registerd and clocked on the negative clock edge.

Figure 2 shows the state transition diagram for our the Caricatron FSM. In addition to the global reset, the Caricatron FSM received synchronized and debounced continue and reject signals from the user. It also received completion signals from the project modules. The following sections will describe the operation of each of the project modules in greater detail.

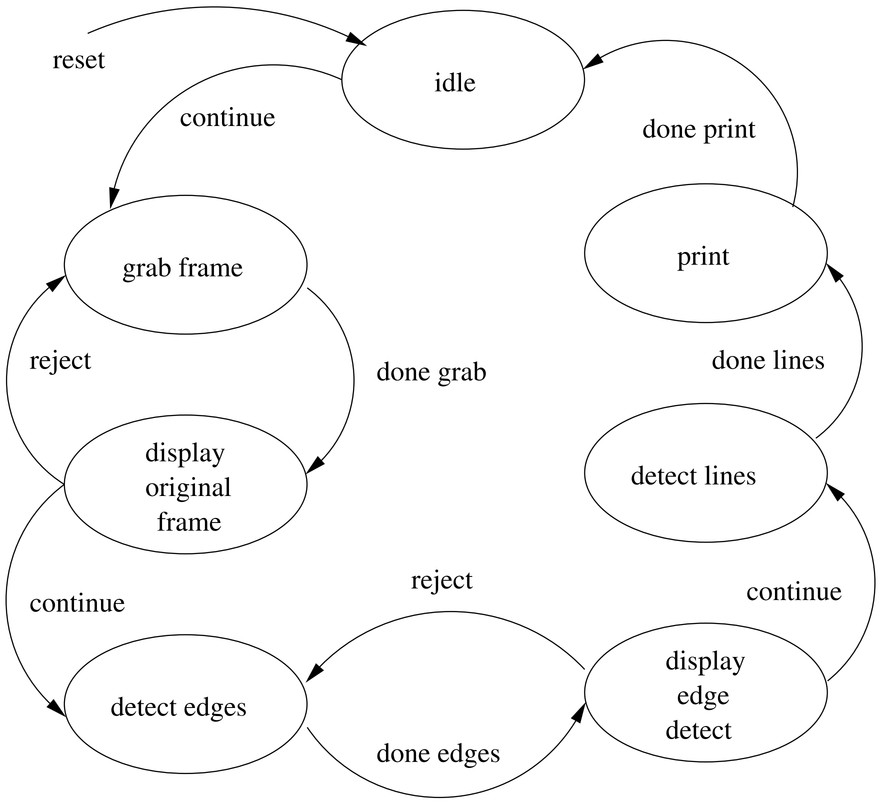


Figure 2: Caricatron FSM

3 Design Methodology

3.1 Video Capture

The first step of the process was to grab a frame from the video camera. An ADV7185 digitized the NTSC video data into a YCrCb format and inserted tags to indicate the start and end of active video information. Each video frame consisted of two interlaced fields that were 780 samples across and 243 lines down. Each pair of samples contained one relevant luminance value, making 390 luminence values in a line. Since we desired a 320 by 240 pixel image, only every other luminence value (every fourth data value) was stored in a line until 320 samples were collected. Also, only data from first 240 lines from one field were collected.

Like our primary system clock, the video data stream was clocked at 27 MHz. However, since the two clocks were not synchronized, data from AD7185 was put into an eight-element fifo buffer. On every rising edge of the video clock, data was written to the next address in the buffer. The video capture module read from the buffer on the edge of every system clock. Upon reset, the read address was set four places behind the write address. This ensured that the data was not changing as we tried to read from the video decoder.

When the Caricatron FSM sent the video capture module a grab frame signal, the module first began searching for the end-active-video sequence of a blanking interval. Next it searched for a begin-active-video of field one sequence. This ensured that we would find the top of the image. The first data value following the tag was the red chrominence value; we skipped this and stored every fourth following to the luminence RAM. Once 320 values were stored, we waited for the next start-active-video sequence. The image capture was complete when 240 lines were stored. This method ensured that our luminence RAM was a small as possible; we further reduced the size of our RAM by noting that although 17 address bits were required to store the data in an {x,y} format, the largest possible x-value was only 320 (instead of 512). Thus our RAM required only 82,000 8-bit locations instead of 131,072.

3.2 VGA Display

The VGA module ran on on a 31.5 MHz clock because at 7/6 of the system clock, it was the simplest 640 by 480 VGA monitor clock to produce using the digital clock manager. Control inputs from other modules were synchronized to the 31.5 MHz clock; the RAM modules that the VGA required were clocked at 31.5 MHz while the VGA module was using them. Whenever it received a run signal from the Caricatron, it began scrolling through x and y values. The x values repeated every 832 counts, while the y values repeated every 520 counts.

On every clock edge, data for that address (from either the luminence or edge RAM) was sent to the red, green, and blue inputs of a ADV7125 8-bit video DAC to produce a monochromatic output. Address values were determined by shifting the x and y positions to the right to divide by two (since a 320 by 240 image was being displayed on a 640 by 480 screen). Blanking and sync information were also set based on the x and y positions. The horizontal and vertical sync signals that went directly to the VGA output were delayed by two clock cycles to account for the delay in the DAC.

3.3 Edge and Line Detection

We divided the image processing section of the system into two major sections: filtering the image to get edge pixels and extracting curves from the edge pixel data. The first section was implemented using a simple 2D convolution. Figure 3 shows the processing system design.

The edge detection system takes data from the YRAM (8-bit values) and stores the resulting filter data into the ERAM (16-bit values) using the ConvolveFSM_Slow, MAC_Slow, and ForceZero modules. The filtering adds zeros for values that are out-of-range in the image (essentially framing the image in layer of zeros seven deep and filtering normally). Once the filter values are in the edge ram, then the second section starts. We designed and implemented an algorithm to extract curve objects from edge pixel data. The algorithm extracts a curve and represents it as four (x,y) points: a start pixel, an end pixel, and two control points. Postscript constructs a standard cubic Bezier curve from this information.

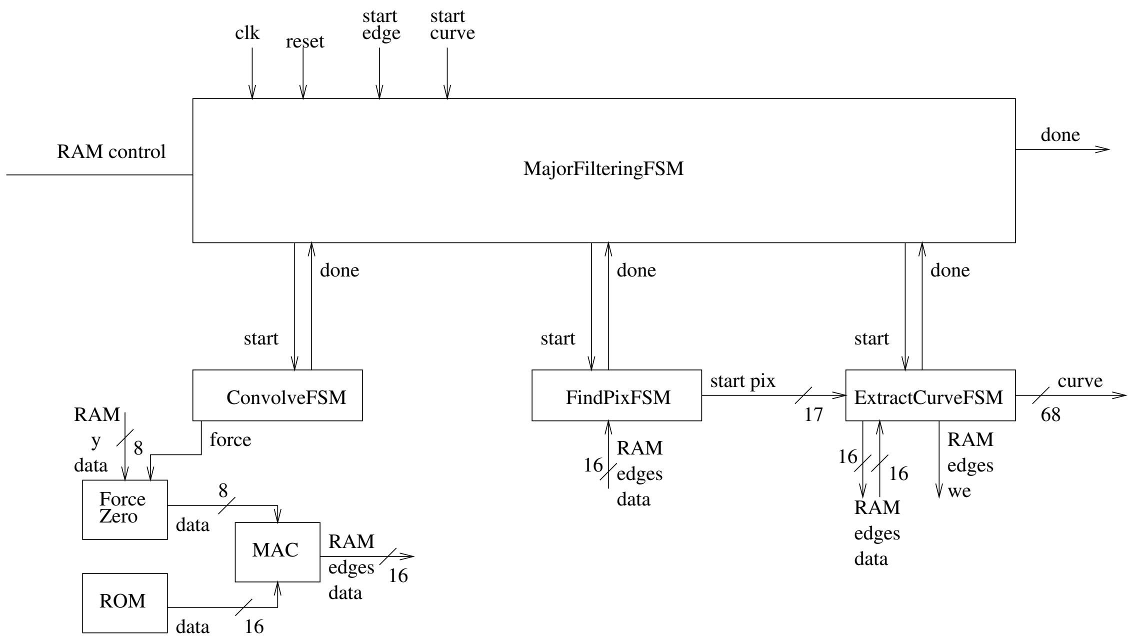


Figure 3: Block Diagram of Edge and Line Detection

We created our own algorithm for extracting curves from edge pixels. The pseudocode below shows the steps for the algorithm. First, we find a pixel which is an edge and which has a neighbor that is also an edge. This pixel will be the start of a curve. The neighbors of a pixel include the eight pixel position surrounding the pixel. Next, we calculate the gradients between the pixel and its neighbors; select the minimum such gradient. Based on the pixel whose gradient is the smallest (pixel Pg) , we compute which possible pixel position to look at next. The possible pixel positions are the three neighbor pixels of Pg closest to the direction of the gradient found for Pg. For example, if the Pg was found to be the pixel right on top, then the possible position would be the three top neighbor pixels of Pg. In this way, the previous gradient direction is used to guide the next locations to extract the curve from. The algorithm keeps walking down pixels in the above manner, keeping track of the maximum x and y position achieved, until Pg is a pixel which is not an edge. Every time a new Pg is computed, we remove the pixel data from the edge ram so as not to loop back on a curve and follow it forever.

```
For each pixel in image, P
    If P is edge AND (any neighbor(P) is edge) then
        StartPixel = P;
// now we have a start pixel

For each neighbor of StartPixel, N
    compute gradient from StartPixel to N
InitialGradient = min(gradient found above)

// set up the current pixel and loop
CurPix = pixel with min gradient
Grad = InitialGradient
while CurPix is edge
    PN = possible neighbors of CurPix given Grad
    Set CurPix to pixel with min gradient within PN
    Set Grad to the gradient found for CurPix

// have a end point
return StartPixel and CurPix as begin and end of curve
```

ConvolveFSM_Slow The ConvolveFSM_Slow module takes care of directing the YRAM and ROM addresses for the MAC_Slow module. The module simple keeps track of the current pixel in the edge ram we are writing to, starting at (0,0). It then iterates through the 15x15 neighbors of this pixel in the YRAM addresses, which pass the data over to the MAC. The ROM addresses are iterated from (0,0) to (14,0) to (14,14). Also, the module must drive the mac_clear, force_zero, and ram_edges_write signals. The mac_clear signal forces the MAC_Slow module to clear it's accumulator and become ready for the next 225 values of the filter. The force_zero signal forces a 0 into the MAC_Slow module when the addresses for the YRAM are out-of-range in terms of the image size. Lastly, the ram_edges_write signal is the write enable for the edge RAM.

ForceZero The ForceZero module takes in the data from the YRAM and a force_zero signal, it outputs an 8-bit value to the MAC_Slow. The module allows the ConvolveFSM_Slow to force the MAC_Slow data for a certain address to be 0 regardless of what the YRAM data actually is. It is used when the YRAM address is out-of-range in terms of the image size.

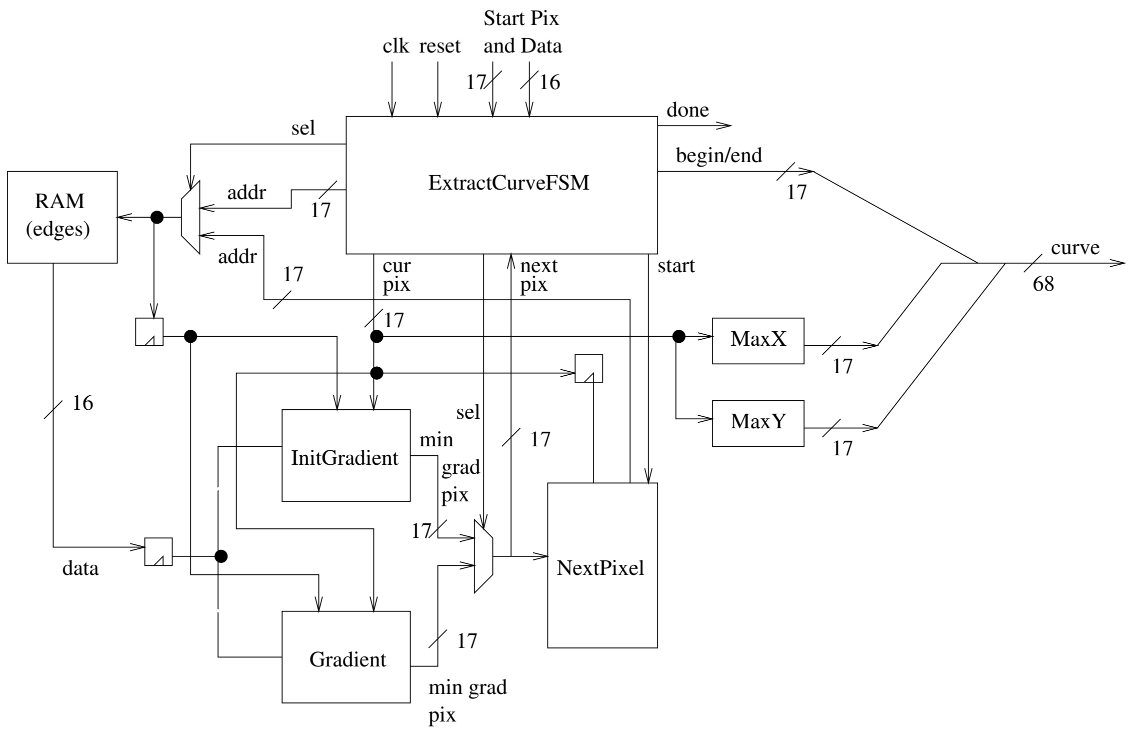


Figure 4: Overall Block Diagram of Line Detection System

MAC_Slow The MAC_Slow module takes in an 8-bit value and a 16-bit value. The 8-bit value comes from the ForceZero module, the 16-bit from the ROM. The 8-bit value is in simple binary format, whereas the 16-bit value is in sign-magnitude format. The output of the MAC_Slow module is a 16-bit value and is in sign-magnitude format. The MAC_Slow module has an internal 32-bit accumulator, the top 16-bits of which become the output. The MAC_Slow module also has a clear signal which forces the accumulator to 0 to reset the value.

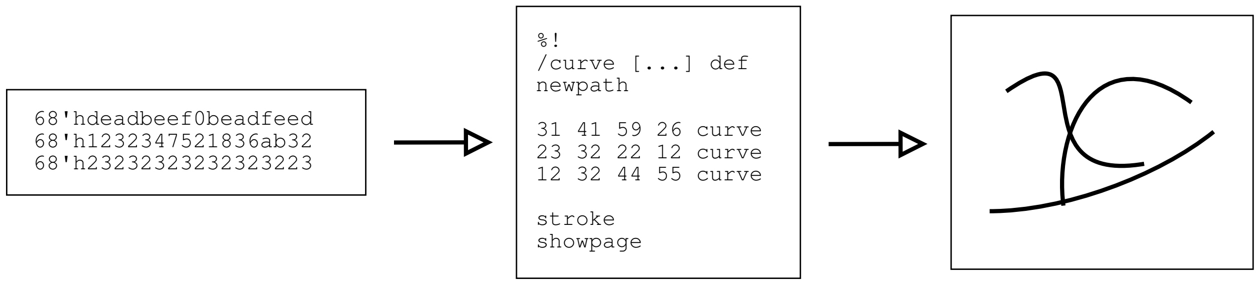
FindPixFSM The FindPixFSM iterates through the edge ram to find a pixel correct for the start pixel of the algorithm described above. It starts at the top left corner of the image and looks for and edge pixel in the edge RAM. Once it finds a single edge pixel, it checks its neighbors to see if any of them are edges. If one neighbor is, then we have found a pixel for extracting curves. Once it finds such a pixel, it stores the value and allows the rest of the system to know what the start pixel is.

ExtractCurveFSM The ExtractCurveFSM module is in chrage of implementing the logic sequencing behind the algorithm for extracting a curve. It takes in a start pixel from the FindPixFSM module as a starting point. The module constantly outputs the current pixel it is looking at and has decided is part of a curve. The module begins by sending the neighbors of the start pixel to the InitGradient module. It does this by setting up the edges RAM addresses, sending all nine pixel address (8 neighbors plus 1 pixel). It then tells then NextPixel module to compute the next pixels to look at, latching the output of the pixel from InitGradient. If this pixel latched is an edge, it becomes the current pixel, else the module finishes and returns the extracted begin and end positions of the curve found. The control points are computed by the MaxY and MaxY modules. The ExtractCurveFSM also takes care of selecting whether to use the InitGradient or Gradient modules (the InitGradient is only used for the first pixel). Figure 4 shows the setup of the ExtractCurveFSM as well as helper modules.

InitGradient The InitGradient takes the last nine inputs given to it and computes the smallest gradient, outputting both the gradient and the pixel for it. It assumes that the nine last inputs form a 3x3 square with the reference pixel being in the middle. This pixel is used to compute the gradient against the other eight pixels. The found signal is not used within the system.

Gradient The gradient module takes the last three pixels and their values. It returns the smallest gradient given the reference pixel that is sent to it. It works similarly to the InitGradient, but must be given the reference pixel and value separately.

NextPixel The NextPixel module takes the output from the Gradient or InitGradient modules. It gets the current pixel from the ExtractCurveFSM and will output three addresses for the next possible pixels to look for given the gradient information. These pixel addresses



```
68'hdeadbeef0beadfeed
68'h1232347521836ab32
68'h2323232323232323223
```

```
%!
/curve [...] def
newpath

31 41 59 26 curve
23 32 22 12 curve
12 32 44 55 curve

stroke
showpage
```

Figure 5: Printing

loop back to the Gradient module which then goes back to the NextPixel. This forms the loop for the algorithm, broken by the ExtractCurveFSM when a non-edge pixel is chosen as the current pixel by the Gradient module.

MaxX and MaxY The MaxX module takes the current pixel from the Gradient module and sotres the maximal x values of the values seen so far. The MaxY is similar to the MaxX module, just for the y variable of pixels.

MajorFilteringFSM The MajorFIlteringFSM acts like a bus arbiter for the YRAM addresses and data as well as the edges RAM. It gives control over to the separate modules then they need it. The module uses the signals sent by the other modules to know what state it is in and change the control.

3.4 PostScript Printing

After finding curves in the image using the line-detect module, the Caricatron FSM transfers control to the printing module. This module interfaces to a PostScript-based laser printer using the Centronics Parallel Port standard (part of IEEE 1284). It accepts 68-bit curve objects extracted from the image by the line detector *via* the Object RAM, and converts them into a sequence of PostScript `curveto` commands. The printer interprets these commands to produce smooth Bézier curves with the given end and control points (see Figure 5).

Figure 6 shows the organization of the module.

Parallel Port Protocol The Centronics (SPP) protocol is the simplest and least feature-rich standard for parallel port communication. While it is outdated and has been replaced by faster, bidirectional protocols like ECP and EPP in practice, printers and computers still support it for backward compatibility. SPP is an asynchronous, half-duplex protocol that either transfers 8 data bits from the host to the peripheral, or 4 control bits from the peripheral to the host. The timing requirements are summarized in Figure 7.

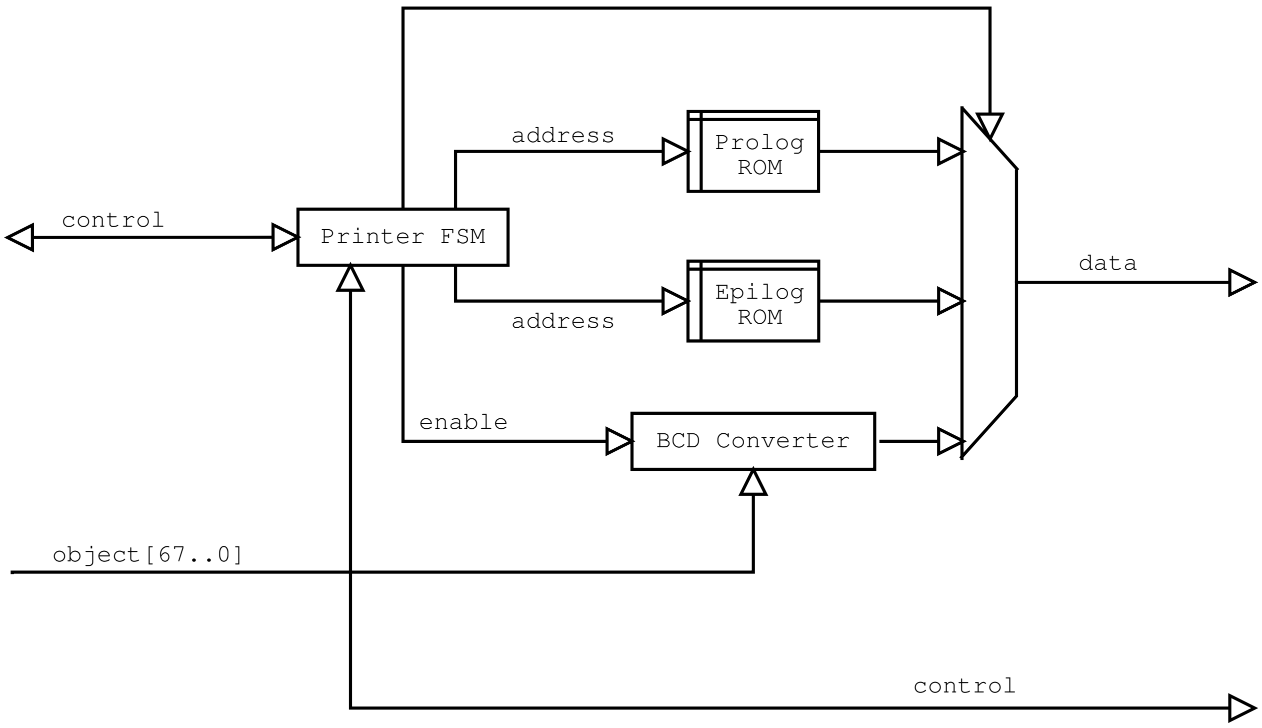


Figure 6: SPP Timing Requirements

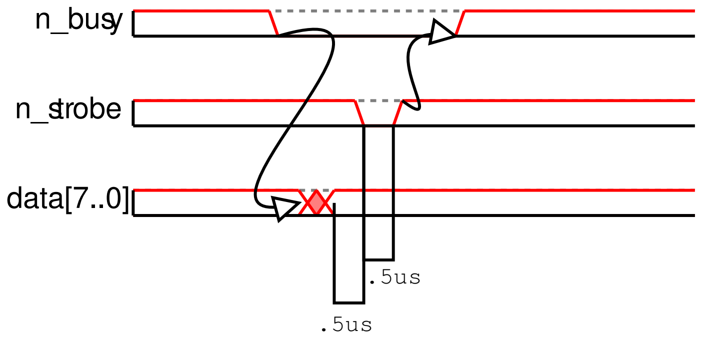


Figure 7: SPP Timing Requirements

PostScript Language Conversion Adobe PostScript is an interpreted, stack-based programming language used to transfer text and graphics information to most laser printers. The caricatures resulting from processing an image consist of a large number of short Bézier curves, so the PostScript stream sent to the printer starts with a brief *prolog* that sets up commands to draw these curves. Then, each object translates into one `curve` command, which specifies the eight coordinate values needed to represent a curve.

4 Testing

4.1 Caricatron FSM

The Caricatron FSM was tested with dummy modules that sent automatic completion signals when they received a start signal. We monitored the state changes on the logic analyzer to make sure that all control outputs were sent at the correct times. Also, the proper operation of the user reset, continue, and reject inputs were observed on the Caricatron FSM's state changes.

Before the Caricatron FSM could use the user inputs, they had to be synchronized and debounced. A sync-debounce module first passed the pushbutton inputs through a pair of registers to synchronize them to the system clock. Next, whenever a pushbutton became active, a one-clock-cycle pulse was sent to the Caricatron FSM (and all other relevant modules). An approximately one-second counter was activated, and all future user inputs were ignored until the counter had completed counting. The module's operation was verified by observing the (bouncy) switch input and the one-cycle pulse on the logic analyzer.

4.2 Video Capture

The video capture module was first observed on the logic analyzer. The video-capture states and write enable signals were compared to the video data stream to make sure that 320 luminence values were being captured after the start-active-video and that the video capture started at the beginning of the field. We also verified the addresses passed to the rams on the logic analyzer.

The creation and use of the RAM to store the video data proved to be a more difficult challenge. When synthesizing from the shared directory, the process would hang on the creation of even small RAMs; this was the primary reason that the code was initially changed to record only the relevant luminence values. (Originally, even the 320x240x8 RAM had to be synthesized as eight 320x240x1 RAMs.) Creating the project in a lcoal directory allowed for the creation of larger RAMs, allowing us to use the register-array method instead of CoreGen tools or the ZBT.

Once created, we still had trouble writing to the RAMs—even a simple module to write the same grey value to all the relevant addresses, or put four quadrants of grey values to the screen, would produce messy results. After double checking the timing on addresses and write enable, we determined that the problem occurred only when the more significant bits of the address were changing. The simplest solution appeared to be registering the address inputs inside the RAM, but we were not able to make this method work. Instead, we decided to have the RAM latch data on the negative clock edge so that all bits would have had time to settle. This solved our image capture problem.

4.3 VGA Display

The duration and frequency of all sync and blanking signals, both to the DAC and directly to the VGA were measured on the logic analyzer and verified against the timing parameters given on the class website. The module was tested using a dummy input module that displayed a separate gray for each quadrant and vertical gradient. Setting the signals based on the given timing parameters was relatively straightforward and easy to implement.

4.4 Edge and Line Detection

Many lessons were learned through the implementation of this system. The first and foremost lesson learned was that synthesis is a truly chaotic process. Any slight change will account for unimaginable repercussion throughout the entire system synthesis, placement, routing, and overall functionality. A single wire being assigned in module B can, and many times did, cause the entire module A to stop working. Even when module B and module A had nothing in common.

Secondly, we found that timing constraint can make or break a system. Several times, adding a single extra bit to a counter broke the entire system. While this is to be expected, timing being key, the nature of our inability to predict or even deal with the timing constraints was unexpected. Most, if not all, of the time for the processing unit was spent debugging timing constraints where the unit broke the rest of the system (not even breaking itself sometimes!).

Last of all, we found that the extremely long compilation time coupled with the chaotic nature of the system greatly crippled our efficiency to debug and even implement a working system. The expected number of compilations for our system before timing would be met was around 4. Thins means that only one out of every four compilations were we actually getting new data on the system. As such, we could not debug properly. Some signal even refused to be changed from a set specific pattern that magically made the system work. The original code written for the processing module changed less than 10 lines in terms of the logic it was performing. However, it took many many iterations, and somewhere around 41 hours of actual lab time, to correctly make it work within the system because of timing issues.

4.5 PostScript Printing

Unfortunately, we began testing the printing functionality very late in the design process. An isolated module that attempted to handshake with the printer and send it a fixed image did not produce any results whatsoever. Because the FPGA was sensitive to small changes in code, this non-functional code was not added to the circuit.

5 Conclusions and Future Work

We were extremely excited and grateful for the opportunity to use the new VirtexII labkits. While we had many diffulties in creating memory blocks and addressing associated Xilinx timing issues, the built in video and VGA chips were quite useful in our project. A better understanding of the fundamentals behind the timing and memory creation issues would have allowed us to complete the project; once we grasped those concepts the obvious next step would be to complete the printing portion of our project and reintroduce more efficent edge and line detection algorithms.

A Top File (Abbreviated Version)

```
module caricatron (clk, clk31, clktv,
   resetA, contA, rejectA, thresh, topButtonA,
   parallel,
   red, green, blue,
   blank, sync, vsync, hsync,
   videoIn,
   debug,
   toLeds);

   //EXTERNAL
   //clocks
   input clk, clk31, clktv;
   //user controls
   input resetA, contA, rejectA;
   input topButtonA;
   wire  reset, cont, reject;
   wire  topButton;
   input [7:0] thresh;
   //for printing
   inout [31:0] parallel;
   //for display
   output [7:0] red, green, blue;
   output  blank, sync, hsync, vsync;
   //for image capture
   input [9:0]  videoIn;

   //INTERNAL
   //with rams
   wire  yclk, eclk, oclk;
   wire [16:0]  lumAddr, edgeAddr, vgaAddr, vidAddr, procAddrY, procAddrE;
   wire [10:0]  procAddrO, printAddr, objAddr;
   wire [7:0]  origOut, origIn, vgaOut;
   wire [15:0]  edgeOut, edgeIn;
   wire [67:0]  procIn, printIn, objIn, objOut;
   wire  ywe, ewe/*baaa!*/, owe, printOwe, procOwe;
   //with majorFSM
   wire [2:0]  muxcntl;
   wire  startGrab, doneGrab, startEdge, doneEdge, startLine;
   wire  doneLine, startPrint, donePrint, runVGA;
   //with image capture
   wire [9:0]  vidQdata;
   //for debuggering
   wire [15:0]  capdebug;
   wire [15:0]  vgadebug;
   output [15:0] debug;
   output [7:0] toLeds;

   //wire done_t;

   /*assign toLeds[0] = (muxcntl == 0) ? 0 : 1;
   assign toLeds[1] = (muxcntl == 1) ? 0 : 1;
   assign toLeds[2] = (muxcntl == 2) ? 0 : 1;
   assign toLeds[3] = (muxcntl == 3) ? 0 : 1;
   assign toLeds[4] = (muxcntl == 4) ? 0 : 1;
   assign toLeds[5] = (muxcntl == 5) ? 0 : 1;
   assign toLeds[6] = (muxcntl == 6) ? 0 : 1;
   assign toLeds[7] = 0;*/

wire [4:0] dbleds;

   assign toLeds = ~{muxcntl, dbleds};

   //assign debug = {clk, hsync, vsync, sync, blank, red[2:0], thresh};
   //assign debug = capdebug;

   //for user signals
   sync myReset (.reset(resetA), .sreset(reset), .clk(clk));

   sync myCont (.reset(contA), .sreset(cont), .clk(clk));

   sync myReject (.reset(rejectA), .sreset(reject), .clk(clk));

   //major fsm and major mux
   caricatronFSM rightBrain (.clk(clk), .reset(reset),
     .continue(cont), .reject(reject),
     .startGrab(startGrab), .doneGrab(doneGrab),
     .startEdge(startEdge), .doneEdge(doneEdge),
     .startLine(startLine), .doneLine(doneLine),
//.doneTemp(done_t),
     .startPrint(startPrint), .donePrint(donePrint),
     .runVGA(runVGA),
     .state(muxcntl));

   topMux theMuxinator (.state(muxcntl),
.vgaAddr(vgaAddr), .vidAddr(vidAddr),
.procAddrY(procAddrY), .procAddrE(procAddrE),
.lumAddr(lumAddr), .edgeAddr(edgeAddr),
.orig(origOut), .edges(edgeOut), .vgaout(vgaOut),
.procAddrO(procAddrO), .printAddr(printAddr),
.objAddr(objAddr),
.procWe(procOwe), .printWe(printOwe), .objWe(owe),
.procIn(procIn), .printIn(printIn), .objIn(objIn),
.clk27(clk), .clk31(clk31), .objclk(oclk),
.lumclk(yclk), .edgeclk(eclk));

   //rams
   Yram origImage (.clk(yclk), .we(ywe), .addr(lumAddr),
.di(origIn), .do(origOut));

   Eram edgedImage (.clk(eclk), .we(ewe/*baaa!*/), .addr(edgeAddr),
.di(edgeIn), .do(edgeOut));

   Oram objects (.clk(oclk), .we(owe), .addr(objAddr),
.di(objIn), .do(objOut));

   //image capture
   videoFifo myFifo (.reset(reset), .tvclk(clktv), .clk(clk),
     .dataIn(videoIn), .dataOut(vidQdata));

   captureVideoSmart framegrabber (.clk(clk), .reset(reset),
.capture(startGrab), .done(doneGrab), .addr(vidAddr),
.writeEn(ywe), .RAMdata(origIn),
.vidData(vidQdata), .debug(capdebug));

   //image display
   VGAout vgadisplay (.pix_clk(clk31), .sreset(reset), .srun(runVGA),
      .imData(vgaOut), .imAddr(vgaAddr),
      .red(red), .green(green), .blue(blue),
      .blank(blank), .sync(sync),
      .hsync(hsync), .vsync(vsync), .debug(vgadebug));

   /*Javier: your module should have the following IO stuff:
    control: inputs: startEdge, startLine; outputs: doneEdge, doneLine;
    user: thresh (8)
    luminence ram: procAddrY(17), origOut(8);
    edge ram: ewe (baaa...), procAddrE(17), edgeOut(16), edgeIn (16);
    object ram: procOwe, procAddrO (17), procIn (68), objOut (68);
    */

   wire [15:0]   dbdebug;

   sync myTopButton(.clk(clk), .reset(topButtonA), .sreset(topButton));



   DonkeyBalls db(.clk(clk),
  .reset(~reset),
  .start_convolution(startEdge || topButton),
  .start_curves(startLine),
  .threshold(thresh),
  .ram_y_out(origOut),
  .ram_edges_out(edgeOut),
  .ram_y_address(procAddrY),
  .ram_edges_address(procAddrE),
  .ram_edges_data(edgeIn),
  .ram_edges_write(ewe),
  .ram_objects_address(procAddrO),
  .ram_objects_data(procIn),
  .ram_objects_write(procOwe),
  .convolution_done(doneEdge),
  .curves_done(doneLine),
  //.done_temp(done_t),
  .debug(dbdebug),
  .leds(dbleds));

   dummy printDummy (.clk(clk), .reset(reset),
//.start(startPrint), .done(donePrint));


endmodule
```

B Video Capture

```
//goal of this module is to capture video data that contains
//atleast one complete field

module captureVideoSmart (clk, reset, capture, done,
addr, writeEn, RAMdata,
vidData,
debug); //for the debuggering
output [15:0] debug;

//system inputs
input clk, reset;
input capture;
output done;
reg done, doneReg;

//video input
input[9:0] vidData;
reg [9:0] vidsync1, vidsync2;
reg [9:0] curr, prev1, prev2, prev3;

//video data sram
output [7:0] RAMdata;
//reg [7:0] RAMdata;
output [16:0] addr;
//reg [16:0] addr, addrReg;
output writeEn;
reg writeEn, writeEnReg;

//state parameters
reg [2:0] state, next;
parameter idle = 0;
parameter lookForBlankingOne = 1;
parameter lookForLineStart = 2;   //of field one only
parameter grabData = 3;
parameter skipData1 = 4;
parameter skipData2 = 5;
parameter skipData3 = 6;

//code parameters
parameter start = 10'b1111111111;
parameter mid = 10'b0000000011;
parameter f2bstart = 10'b1110110011;
parameter f2bend = 10'b1111000111;
parameter f1bstart = 10'b1010101111;
parameter f1bend = 10'b1011011011;
parameter f1start = 10'b1000000011;
parameter f1end = 10'b1001110111;
parameter f2start = 10'b1100110011;
parameter f2end = 10'b1101101011;

//internal addresses
reg [8:0] xpos, xposReg;
reg [7:0] ypos, yposReg;
parameter xmax = 320;
parameter ymax = 240;

//internal data
reg [7:0] writeData;

//for that debuggering
assign debug = {state, ypos, capture, done, clk, reset, writeEn};


//keep track of the last four values (registered)
always @ (posedge clk) begin
//vidsync1 <= vidData;
//vidsync2 <= vidsync1;
//curr <= vidsync2;
curr <= vidData;
prev1 <= curr;
prev2 <= prev1;
prev3 <= prev2;
end




//advance the state
always @ (posedge clk) begin
if(reset)begin
state <= idle;
writeEn <= 0;
xpos <= 0;
ypos <= 0;
done <= 0;
end
else begin
state <= next;
```

```
writeEn <= writeEnReg;
xpos <= xposReg;
ypos <= yposReg;
done <= doneReg;
end
end

//set the next state
always @ (curr or state or xpos or ypos) begin
writeEnReg = 0;
doneReg = 0;

case(state)
idle: begin
writeData = writeData;
if(capture)begin
next = lookForBlankingOne;
xposReg = 0;
yposReg = 0;
end
else begin
next = idle;
xposReg = 0;
yposReg = 0;
end
end
lookForBlankingOne: begin
writeData = writeData;
if((curr == f1bend) &&
   (prev1 == mid) &&
   (prev2 == mid) &&
   (prev3 == start)) begin
    next = lookForLineStart;
xposReg = 0;
yposReg = 0;
    end
else begin
next = lookForBlankingOne;
xposReg = 0;
yposReg = 0;
end
end
lookForLineStart: begin
writeData = writeData;
if((curr == f1start) &&
   (prev1 == mid) &&
   (prev2 == mid) &&
   (prev3 == start)) begin
    next = skipData3;
xposReg = xpos;
yposReg = ypos;
    end
          else begin
             next = lookForLineStart;
xposReg = xpos;
yposReg = ypos;
           end
          end
//old stuff here

skipData1: begin
next = skipData2;
xposReg = xpos;
yposReg = ypos;
end
skipData2: begin
next = skipData3;
xposReg = xpos;
yposReg = ypos;
end
skipData3: begin
next = grabData;
xposReg = xpos;
yposReg = ypos;
writeEnReg = 1;
end
grabData: begin
if((xpos == xmax) && (ypos == ymax)) begin
next = idle;
xposReg = 0;
yposReg = 0;
doneReg = 1;
end
else if ((xpos == xmax) && (ypos < ymax)) begin
next = lookForLineStart;
xposReg = 0;
yposReg = ypos + 1;
end
else begin
next = skipData1;
```

```
//next = skipData3;   //a change
xposReg = xpos + 1;
yposReg = ypos;
end
end
 //end of old stuff
//new stuff so addr, data valid over En
/*
skipData1: begin
writeData = writeData;
next = skipData2;
xposReg = xpos;
yposReg = ypos;
end
skipData2: begin
writeData = writeData;
if((xpos == xmax) && (ypos == ymax)) begin
next = idle;
xposReg = 0;
yposReg = 0;
doneReg = 1;
end
else if ((xpos == xmax) && (ypos < ymax)) begin
next = lookForLineStart;
xposReg = 0;
yposReg = ypos + 1;
end
else begin
next = skipData3;
xposReg = xpos + 1;
yposReg = ypos;
end
end
skipData3: begin
writeData = writeData;
next = grabData;
xposReg = xpos;
yposReg = ypos;
end
grabData: begin
writeData = curr[9:2];
next = skipData1;
xposReg = xpos;
yposReg = ypos;
writeEnReg = 1;
```

```
end
*/ //end of new stuff

endcase
end

//always set RAMdata, addr to current value
//assign RAMdata = writeData;  //new
assign RAMdata = curr[9:2];  //old
assign addr = {xpos, ypos};


endmodule
```

C Laplacian of Gaussian Values in Edge ROM

```
// *fudge-factor* == \infty
// 15x15 rom found using MatLab
//all values not specified here are 16'd27

 8'h00: rom_data <= 16'd27;
 8'h01: rom_data <= 16'd27;
 8'h02: rom_data <= 16'd27;
 8'h03: rom_data <= 16'd27;
 8'h04: rom_data <= 16'd27;
 8'h05: rom_data <= 16'd27;
 8'h06: rom_data <= 16'd27;
 8'h07: rom_data <= 16'd27;
 8'h08: rom_data <= 16'd27;
 8'h09: rom_data <= 16'd27;
 8'h0A: rom_data <= 16'd27;
 8'h0B: rom_data <= 16'd27;
 8'h0C: rom_data <= 16'd27;
 8'h0D: rom_data <= 16'd27;
 8'h0E: rom_data <= 16'd27;

    ...

 8'h50: rom_data <= 16'd27;
 8'h51: rom_data <= 16'd27;
 8'h52: rom_data <= 16'd27;
 8'h53: rom_data <= 16'd27;
 8'h54: rom_data <= 16'd27;
```

```
8'h55: rom_data <= 16'd27;
8'h56: rom_data <= 16'd28;
8'h57: rom_data <= 16'd28;
8'h58: rom_data <= 16'd28;
8'h59: rom_data <= 16'd27;
8'h5A: rom_data <= 16'd27;
8'h5B: rom_data <= 16'd27;
8'h5C: rom_data <= 16'd27;
8'h5D: rom_data <= 16'd27;
8'h5E: rom_data <= 16'd27;

8'h60: rom_data <= 16'd27;
8'h61: rom_data <= 16'd27;
8'h62: rom_data <= 16'd27;
8'h63: rom_data <= 16'd27;
8'h64: rom_data <= 16'd27;
8'h65: rom_data <= 16'd28;
8'h66: rom_data <= 16'd135;
8'h67: rom_data <= 16'd1014;
8'h68: rom_data <= 16'd135;
8'h69: rom_data <= 16'd28;
8'h6A: rom_data <= 16'd27;
8'h6B: rom_data <= 16'd27;
8'h6C: rom_data <= 16'd27;
8'h6D: rom_data <= 16'd27;
8'h6E: rom_data <= 16'd27;

8'h70: rom_data <= 16'd27;
8'h71: rom_data <= 16'd27;
8'h72: rom_data <= 16'd27;
8'h73: rom_data <= 16'd27;
8'h74: rom_data <= 16'd27;
8'h75: rom_data <= 16'd28;
8'h76: rom_data <= 16'd1014;
8'h77: rom_data <= 16'hA927; //d-10535;
8'h78: rom_data <= 16'd1014;
8'h79: rom_data <= 16'd28;
8'h7A: rom_data <= 16'd27;
8'h7B: rom_data <= 16'd27;
8'h7C: rom_data <= 16'd27;
8'h7D: rom_data <= 16'd27;
8'h7E: rom_data <= 16'd27;

8'h80: rom_data <= 16'd27;
8'h81: rom_data <= 16'd27;
```

```
8'h82: rom_data <= 16'd27;
8'h83: rom_data <= 16'd27;
8'h84: rom_data <= 16'd27;
8'h85: rom_data <= 16'd28;
8'h86: rom_data <= 16'd135;
8'h87: rom_data <= 16'd1014;
8'h88: rom_data <= 16'd135;
8'h89: rom_data <= 16'd28;
8'h8A: rom_data <= 16'd27;
8'h8B: rom_data <= 16'd27;
8'h8C: rom_data <= 16'd27;
8'h8D: rom_data <= 16'd27;
8'h8E: rom_data <= 16'd27;

8'h90: rom_data <= 16'd27;
8'h91: rom_data <= 16'd27;
8'h92: rom_data <= 16'd27;
8'h93: rom_data <= 16'd27;
8'h94: rom_data <= 16'd27;
8'h95: rom_data <= 16'd27;
8'h96: rom_data <= 16'd28;
8'h97: rom_data <= 16'd28;
8'h98: rom_data <= 16'd28;
8'h99: rom_data <= 16'd27;
8'h9A: rom_data <= 16'd27;
8'h9B: rom_data <= 16'd27;
8'h9C: rom_data <= 16'd27;
8'h9D: rom_data <= 16'd27;
8'h9E: rom_data <= 16'd27;

   ...

8'hE0: rom_data <= 16'd27;
8'hE1: rom_data <= 16'd27;
8'hE2: rom_data <= 16'd27;
8'hE3: rom_data <= 16'd27;
8'hE4: rom_data <= 16'd27;
8'hE5: rom_data <= 16'd27;
8'hE6: rom_data <= 16'd27;
8'hE7: rom_data <= 16'd27;
8'hE8: rom_data <= 16'd27;
8'hE9: rom_data <= 16'd27;
8'hEA: rom_data <= 16'd27;
8'hEB: rom_data <= 16'd27;
8'hEC: rom_data <= 16'd27;
```

```
 8'hED: rom_data <= 16'd27;
 8'hEE: rom_data <= 16'd27;

 default: rom_data <= 16'h0;
```

D Line Extraction

```
module ExtractCurveFSM(clk,reset,start, start_pix, start_pix_data,
       next_pix, next_pix_data, thresh, start_next_pixel,
       ram_edges_write, ram_edges_address, ram_edges_data,
       ram_edges_sel, cur_pix, cur_pix_data, grad_sel,
       return_data, done);

   input clk, reset, start;
   input [16:0] start_pix;
   input [15:0] start_pix_data;
   input [16:0] next_pix;
   input [15:0] next_pix_data;
   input [7:0]  thresh;
   output  start_next_pixel;
   output  ram_edges_write;
   output [16:0] ram_edges_address;
   output [15:0] ram_edges_data;
   output   ram_edges_sel;
   output [16:0] cur_pix;
   output [15:0] cur_pix_data;
   output   grad_sel;
   output [33:0] return_data;
   output   done;

   reg   start_next_pixel, start_next_pixel_int;
   reg   ram_edges_write, ram_edges_write_int;
   reg [16:0]   ram_edges_address, ram_edges_address_int,
 cur_pix, cur_pix_int;
   reg [15:0]   ram_edges_data, ram_edges_data_int,
 cur_pix_data, cur_pix_data_int;
   reg   ram_edges_sel, ram_edges_sel_int,
 grad_sel, grad_sel_int,
 done, done_int;
   reg [33:0]   return_data, return_data_int;


   reg [3:0]   counter, counter_int, tcounter, tcounter_int;
   /*reg   latch_next_pix, latch_start_pix, load_cur_pix;*/
   reg [16:0]   cur_pix_reg;
   //reg [15:0]   cur_pix_data_reg;
   reg [16:0]   beg_pix_reg /*, end_pix_reg*/;
   /*reg [15:0]   beg_pix_data_reg , end_pix_data_reg;*/

   reg [8:0]   tempx;
   reg [7:0]   tempy;

   reg [4:0]   state, next;

   \\eating one battery
   \\eating five batteries

   parameter   THRESHOLD = 0; // ?!?!? what threshold??
   parameter   IMAGE_WIDTH = 320;
   parameter   IMAGE_HEIGHT = 240;
   parameter   INIT = 0;
   parameter   FIND_INIT_GRAD = 1;
   parameter   GENERATE_NEXT = 2;
   parameter   GENERATE_NEXT_WAIT = 7;
   parameter   FIND_MIN_GRAD = 3;
   parameter   NEW_POINT = 4;
   parameter   REMOVE_POINT = 5;
   parameter   RETURN = 6;
   parameter   RETURN_DATA = 8;

   always @ (posedge clk)
     begin

if(!reset) begin
   state <= INIT;
   ram_edges_write <= 0;
   ram_edges_address <= 0;
   ram_edges_data <= 0;
   cur_pix <= 0;
   cur_pix_data <= 0;
   ram_edges_sel <= 0;
   grad_sel <= 0;
   done <= 0;
   counter <= 0;
   tcounter <= 0;
   cur_pix_reg <= 0;
   //cur_pix_data_reg <= 0;
   beg_pix_reg <= 0;
   //beg_pix_data_reg <= 0;
   //end_pix_reg <= 0;
   //end_pix_data_reg <= 0;
   return_data <= 34'h3FFFF;
   start_next_pixel <= 0;
end
else begin
   state <= next;
   start_next_pixel <= start_next_pixel_int;
   ram_edges_write <= ram_edges_write_int;
   ram_edges_address <= ram_edges_address_int;
   ram_edges_data <= ram_edges_data_int;
   cur_pix <= cur_pix_int;
   cur_pix_data <= cur_pix_data_int;
   ram_edges_sel <= ram_edges_sel_int;
   grad_sel <= grad_sel_int;
   done <= done_int;
   return_data <= return_data_int;
   counter <= counter_int;
   tcounter <= tcounter_int;
   if(start) begin
      cur_pix <= start_pix;
      cur_pix_data <= start_pix_data;
      beg_pix_reg <= start_pix;
      //beg_pix_data_reg <= start_pix_data;
   end
   else begin
      cur_pix <= cur_pix_int;
      cur_pix_data <= cur_pix_data_int;
      beg_pix_reg <= beg_pix_reg;
      //beg_pix_data_reg <= beg_pix_data_reg;
   end
   //end_pix_reg <= end_pix_reg;
   //end_pix_data_reg <= end_pix_data_reg;
end

     end

   always @ (state or reset or start or start_pix or start_pix_data
     or next_pix or next_pix_data or counter
     or cur_pix_reg /*or cur_pix_data_reg*/
or ram_edges_address or ram_edges_data or cur_pix
or cur_pix_data or return_data or tcounter or thresh or beg_pix_reg)
     begin

// defaults of signals
```

```
ram_edges_address_int = ram_edges_address;
ram_edges_data_int = ram_edges_data;
cur_pix_int = cur_pix;
cur_pix_data_int = cur_pix_data;
ram_edges_write_int = 0;
start_next_pixel_int = 0;
ram_edges_sel_int = 0;
grad_sel_int = 0;
done_int = 0;
return_data_int = return_data;
counter_int = counter;
tcounter_int = tcounter;
next = INIT;

if(!reset) begin
   ram_edges_address_int = 0;
   ram_edges_data_int = 0;
   cur_pix_int = 0;
   cur_pix_data_int = 0;
   ram_edges_sel_int = 0;
   grad_sel_int = 0;
   done_int = 0;
   return_data_int = 34'h3FFFF;
   counter_int = 0;
   tcounter_int = 0;
end
else begin

   case(state)

     INIT: begin
if(start) begin
   counter_int = 0;
   tcounter_int = 0;
   grad_sel_int = 1;
   ram_edges_sel_int = 1;
   ram_edges_address_int = cur_pix_reg;
   // start reading from Y ram
   next = FIND_INIT_GRAD;
end
else begin
   next = INIT;
end
     end
     FIND_INIT_GRAD: begin
if(counter == 2) begin
   if(tcounter == 2) begin
      //finally got done reading all data
      next = GENERATE_NEXT_WAIT; // wait 2 cycles
      counter_int = 0;
      tcounter_int = 0;
      //start_next_pixel_int = 1;
   end
   else begin
      counter_int = 0;
      tcounter_int = tcounter + 1;
      next = FIND_INIT_GRAD;
   end
end
else begin
   counter_int = counter + 1;
   next = FIND_INIT_GRAD;
end
tempx = cur_pix[16:8] - 1 + counter;
tempy = cur_pix[7:0] - 1 + tcounter;
ram_edges_address_int = {tempx,
 tempy};
grad_sel_int = 1;
ram_edges_sel_int = 1;
     end

     GENERATE_NEXT_WAIT: begin
if(counter == 3) begin
   // latch the next pix
   cur_pix_int = next_pix;
   cur_pix_data_int = next_pix;
   counter_int = 0;
   //start_next_pixel_int = 1;
   next = GENERATE_NEXT;
end
else begin
   counter_int = counter + 1;
   next = GENERATE_NEXT_WAIT;
end
if(counter == 2) begin
   start_next_pixel_int = 1;
end
if(counter < 2) begin
   grad_sel_int = 1;
   ram_edges_sel_int = 1;
   ram_edges_address_int = cur_pix;
   ram_edges_data_int = 0;
   ram_edges_write_int = 1;
end
     end

     GENERATE_NEXT: begin
counter_int = 0;
next = FIND_MIN_GRAD;
//start_next_pixel_int = 1;
     end

     FIND_MIN_GRAD: begin
// wait for 2 cycles
if(counter == 4) begin
   start_next_pixel_int = 1;
   next = NEW_POINT;
   counter_int = 0;
end
else begin
   counter_int = counter + 1;
   next = FIND_MIN_GRAD;
end
if(counter == 2) begin
   //start_next_pixel_int = 1;
end
     end
     NEW_POINT: begin
if(next_pix_data > thresh && next_pix_data < 16'h8000) begin
   // latch the next pix
   cur_pix_int = next_pix;
   cur_pix_data_int = next_pix_data;

   // remove the point here, so in sync with cur_pix change
   ram_edges_sel_int = 1;
   ram_edges_address_int = next_pix;
   ram_edges_data_int = 0;
   // write to ram!

   counter_int = 0;

   //check if at edges of image
   if(next_pix[16:8] > IMAGE_WIDTH - 1 || next_pix[7:0] > IMAGE_HEIGHT - 1)
     next = RETURN;
   else
     next = REMOVE_POINT;
end
else begin
   next = RETURN;
end
     end

     REMOVE_POINT: begin
// make sure to take enough time to sync with the NEXT PIX block!
// wait 2 cycles
if(counter == 1) begin
   counter_int = 0;
   next = GENERATE_NEXT; // might have to wait more?!?
end
else begin
   counter_int = counter + 1;
   next = REMOVE_POINT;
end
ram_edges_sel_int = 1;
ram_edges_address_int = cur_pix;
ram_edges_data_int = 0;
ram_edges_write_int = 1;
     end

     RETURN: begin
ram_edges_address_int = cur_pix;
ram_edges_sel_int = 1;
ram_edges_data_int = 0;
ram_edges_write_int = 1;
next = RETURN_DATA;
     end
     RETURN_DATA: begin
return_data_int = {beg_pix_reg, cur_pix};
done_int = 1;
next = INIT;
     end
   endcase
end
     end
endmodule
```