Laser Pointer Mouse

Xinpeng Huang and William Putnam

TA: Javier Castro
6.111: Introductory Digital Systems Laboratory
18 May 2006

Abstract: The purpose of this project is to design and implement a laser pointer mouse. The laser pointer mouse allows lecturers and presenters to point at a projected screen, and, with the press of a button, move the mouse cursor to the location of the laser, without ever touching the computer or mouse. A few more buttons allow the user to perform left, right, and double clicks. The system is implemented in Verilog, and realized on the FPGA on the 6.111 labkit.

Overview

During a PowerPoint or any other kind of presentation, it would be convenient for the lecturer to be able to not only point and direct with his laser pointer, but also to use it as a mouse. For example, he or she may want to open up a demo in another window, and it would be convenient to do so without having to walk over to his or her computer. Hence our project idea, the laser pointer mouse, is born.

The laser pointer mouse works as follows. A camera is directed at the projected screen. The camera inputs its video data into the FPGA, where it is converted to a more useful format and then processed to extract the location of the laser pointer on the screen, along with the corner coordinates of the screen. The projected screen is then mapped to the computer screen – the projection does not fill up the entire field of view of the camera, so some rendering will be done to make the necessary conversion. After the laser pointer location is rendered to map to the user's computer screen, its coordinates are outputted through PS/2 to the computer to move the mouse in the appropriate fashion.

In addition to this basic functionality, a user interface with a variety of operating modes is incorporated into the system. The system has three functional modes: idle, find, and adjust. Additionally, the user interface provides the user with a variety of options in customizing the image processing to best suit the current lighting conditions. The user can choose between three possible methods of finding the laser pointer and can adjust various parameters that define how the laser is found. Figure 1 presents an overall block diagram of the complete laser pointer mouse system. The rest of this paper will go through each of the blocks in detail.

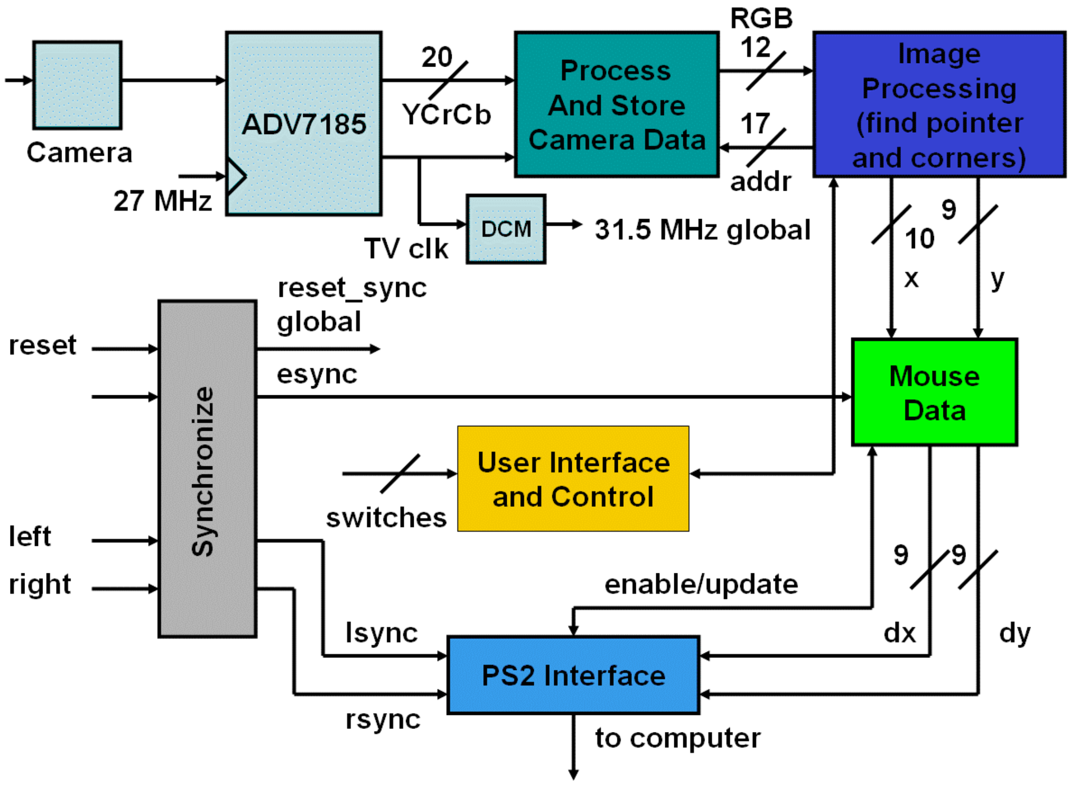

Figure 1: Overall system block diagram.

Video Input Buffering

Xinpeng Huang

Our system requires a camera to take in data from the screen where the laser pointer will be shined (hereafter the *projection* or *projected screen*) and store it into memory, so that the image processing can take in a video frame and find the location of the pointer. A high-level block diagram of the video input processing is shown in Figure 2.

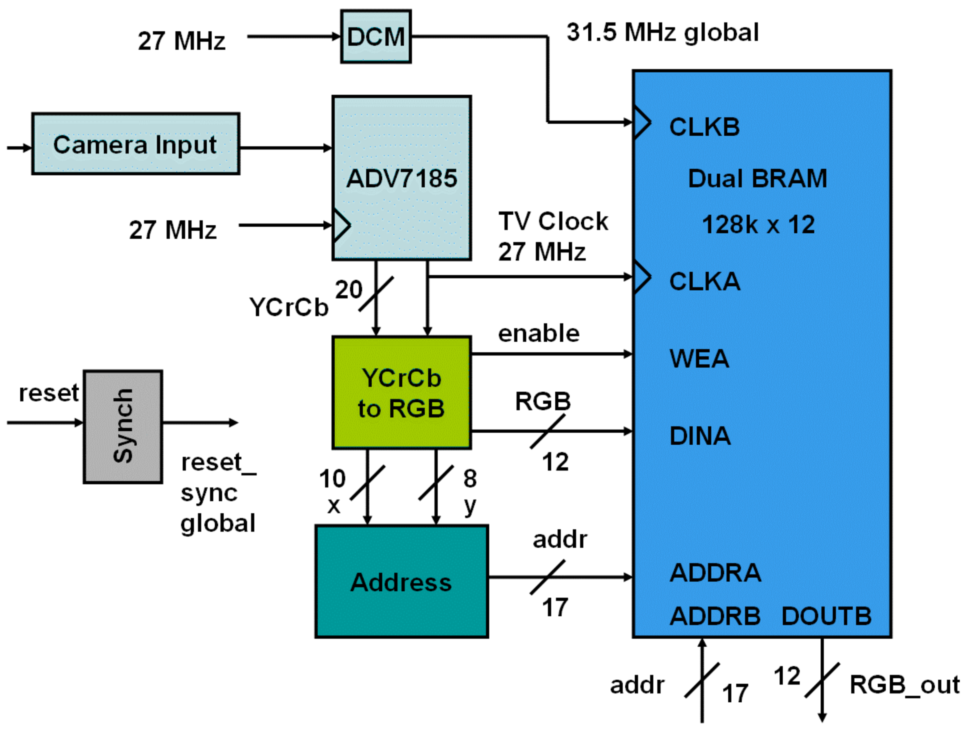

Figure 2: Video block diagram.

The light from the projection goes through a series of darkening filters before passing into the camera. The filters are needed to filter ambient noise from the camera by darkening the incoming light intensity, and as a result make image processing easier later.

The video data coming out of the camera is an analog TV composite signal compatible with NTSC standards. It is transmitted through the composite video port in the labkit and passes through the ADV7185 video decoder before entering the FPGA. The ADV7185 is an analog-to-digital converter (ADC) that takes in the analog TV data and outputs digital YCrCb signals to the FPGA. The output data is sampled at a 27 MHz TV clock generated by the decoder.

YCrCb is a family of color spaces used for composite video data. Y is the luminance component, and Cr and Cb are red and blue chrominance components, respectively. YCrCb comes out from the ADV7185 as a 20-bit signal, the ten most significant bits of which actually contain all the color data. This 10-bit YCrCb data can be converted into RGB using the following transformations[1]:

[1] Reproduced from XAPP283 Xilinx Color Space Converter, http://www-mtl.mit.edu/Courses/6.111/labkit/appnotes/xapp283.pdf.

R = 1.164 (Y – 64) + 1.596 (Cr – 512),
G = 1.164 (Y – 64) – 0.813 (Cr – 512) – 0.392 (Cb – 512),
B = 1.164 (Y – 64) + 2.017 (Cb – 512).

In addition, YCrCb obeys a set of timing and blanking standards. The three relevant signals are *F* (field), *V* (vertical blank), and *H* (horizontal blank). The video data is actually interlaced into odd and even lines, and the field signal indicates which set of lines is being processed, with *F* = 0 odd and *F* = 1 even. *H* and *V* are blanking signals, and video data is only being sent from the camera when both *H* and *V* are zero. The data comes in with 720 active pixels per line and a total of 525 lines.

Generally, *F*, *V*, and *H* only change at the beginning and end of every line of video data written. The data from the DAC communicates the *F*, *V*, and *H* signals by preceding each transition with a series of 10-bit *timing reference* words in the order 3FF, 000, 000, *XY*. The last word, *XY*, takes the form *XY* = {1, *F*, *V*, *H*, *P3*, *P2*, *P1*, *P0*, 0, 0}, where *P3*, *P2*, *P1*, and *P0* are protection bits intended for error detection.

We made a conscious design decision to store a frame of video data into block memory. This eliminates the problems caused by the instability of the TV clock as well as the need to devise a scheme to overcome the problem of clocking the entire system at two different frequencies. Due to limited memory space, we are only able to store a quarter of the information, so the resolution in both the x- and y-directions is cut in half. This conveniently allows us to ignore the even lines and store only the odd lines, thus simplifying the decoding process.

The *YCrCb-to-RGB* module in Figure X1 is a finite state machine (FSM) that is clocked on the 27 MHz clock from the ADC. It reads in YCrCb data on the rising edge of the TV clock and looks for the sequence of words 3FF, 000, 000, *XY* = 200, corresponding to *F* = *V* = *H* = 0. This signals that a new line is about to come from the decoder. The FSM reads data until coming across the sequence 3FF, 000, 000, *XY* = 27C corresponding to *F* = *V* = 0, *H* = 1, signaling the end of the line. Every line, a line counter is incremented to keep track of the current (odd) line being sampled. At the end of a frame, the FSM detects 3FF, 000, 000, *XY* = 2D8 corresponding to *F* = 0, *V* = *H* = 1, and resets the line counter to prepare for the next frame.

During active video, data is received in the pattern Cb, Y, Cr, Y, Cb, Y, Cr, and so on. The FSM stores the current Cb, Cr, and Y values in registers and waits for the next packet of data. The Cr and Cb values are shared between adjacent pixels, so for every two data words, we have enough information to determine the YCrCb color values of a single given pixel. A combinational logic block calculates RGB from YCrCb according to the transformations described earlier, and this color data is outputted to RAM, along with a memory write enable signal every two clock cycles.

The memory element used is a Xilinx CoreGen dual-port block RAM. In order to infer a mapping between pixel coordinates and memory addresses, a simple *address* module is created that simply concatenates the most significant 9 bits of the 10-bit x-coordinate with the y-coordinate (which has already been halved). The result is a 17-bit address that is sent to the RAM. The first set of ports on the RAM is clocked on the TV clock, and samples the most significant 12 bits of RGB data on every enable pulse. We then have a memory size of 128k x 12, which can be comfortably generated using available block RAMs on the FPGA. On the other set of ports, the image processing module, clocked at 31.5 MHz, can access the color of any pixel in the current frame and perform the calculations it needs.

Image Processing and Control

William Putnam

In this section the design and implementation of the system's image processing and control units will be discussed. The image processing portion of the system uses several algorithms to extract pertinent information from the buffered video input, including the location of the red laser pointer on the screen and the corner coordinates of the projection. The control element is a small system of major and minor Moore-type FSMs that organize and coordinate the operation of various parts of the image processing, as well as the taking and passing of user input and camera data to the this part of the system.

Image Processing

The image processing portion of the system performs two main tasks. It determines the laser pointer's coordinates and locates the corners of the projected screen. In this subsection we will first discuss various challenges involved in the image processing, and then outline the algorithms implemented to face these challenges and extract the pointer location and corner coordinates of the screen.

Pointer Finding—Challenges

Finding the laser pointer presents several challenges. The first problem is over-saturation of the camera—the camera takes in too much light and everything appears very white. Most PowerPoint or other presentations are projected onto a white screen. Hence, the projection is set up as a large white piece of cardboard attached to the wall. The reflective nature of the white surface coupled with the close proximity of the lab's lighting leads to a large amount of ambient light being captured by the camera. When the video data is converted to RGB and displayed on the VGA monitor, everything appears very bright with poor contrast. Consequently, it is very difficult to distinguish between the red of the laser pointer and the white of the screen. To minimize these detrimental effects of over-saturation, we use two neutral density filters with the camera. These filters simply reduce the intensity of the light entering the camera, thereby decreasing the overall brightness of the image and forming a clearer distinction between the red of the laser and the white of the screen.

Another formidable challenge associated with finding the laser pointer involves the actual size of the pointer. In the buffered video data and on the VGA display, the actual laser pointer is made up of a very small number of pixels (with the color range we are using, this number seems to range from around 2 to 10). The pointer finding algorithms are based on a color matching scheme—the algorithms essentially find a collection of pixels in a certain color range and average their coordinates to find the pointer's center. Since the laser pointer comprises of very few pixels, these algorithms can only wait for a small number of "good" pixels (pixels in the appropriate color range) before assigning the pointer location. The laser pointer tracking algorithms are therefore more susceptible to random white noise present in the video and lack smooth behavior when following a fast-moving laser pointer. The latter follows from the fact that when the laser pointer moves rapidly across the screen, its pixels seem to spread somewhat and, being that they are farther apart, the algorithm is more tempted to discard such pixels as noise.

Pointer Finding—Algorithms

In order to overcome these two major pointer finding challenges, two different finding algorithms are implemented in parallel. Each algorithm is based on a color-matching scheme. This essentially means that each algorithm looks at each pixel's RGB data and compares it to some reference values to determine if the pixel could be part of laser pointer. These values are called the *floor* and *roof* values. If a pixel's RGB data is greater than the *floor* value and less than the *roof* value, then the pixel is flagged as a potential laser pointer pixel.

The first and primary algorithm was implemented in a module called *pointerfind*. This algorithm seeks to minimize the destructive effects of noise. Color-matching is performed, and after a pixel is flagged as a potential laser pointer pixel, the algorithm will wait to see if another pixel in the appropriate color range (call them red pixels) appears within a specified distance of this pixel (this distance is specified by a parameter called *BOX* = 2). If a pixel does appear within this distance, the coordinates of the two pixels are averaged—this is done by adding their coordinates and then bit-shifting right by one—and the result is outputted as the laser pointer's center coordinate. If a pixel does not appear within this range, then the process is restarted when the next red pixel is found.

This algorithm relies on the fact that it is improbable that two very red pixels will randomly appear as noise in very close proximity with one another. Although this algorithm performs well with noise, it is ineffective with a rapidly moving pointer. This is because a rapidly moving pointer will cause the few laser pointer pixels to spread and, if the pointer is moving fast enough, this spread will be greater than the *BOX* parameter and the pointer will not be found until it stops its rapid movements.

One more additional feature of this algorithm is to limit jumpiness when the laser pointer is stationary. When the pointer is stationary, its red pixels may still be shifting around slightly because the person holding the pointer may not be able to stabilize his hand. In order to limit the error associated with these vibrations, when the algorithm is ready to update the pointer's center coordinate, it checks if the new center coordinate is within a certain distance (specified by the parameter *STILL*) of the old center coordinate. If it is within this range then the pointer's position will not update, so it will not jump. Otherwise, the pointer will move as it does normally.

The other pointer finding algorithm was implemented in a module called *otherfind.* This algorithm also runs on a color-matching scheme, but is somewhat simpler than the preceding one. For each pixel that is flagged, a counter is incremented and the coordinates of the flagged pixel are added to two running sums, one for each of the x- and y-coordinates. When the counter reaches four, the running sums are divided by four by shifting two bits to the right, and this result is outputted as the laser pointer's center coordinate. This procedure of adding and bit-shifting is equivalent to averaging. Clearly this algorithm will be less robust to noise than the preceding one, but it will perform slightly better against a rapidly moving pointer.

Corner Finding

The second half of the image processing block involves finding the corners of the projected screen. The corner finding algorithm is similar to the pointer finding algorithms, although somewhat simpler, because the problem of locating a very small set of pixels is no longer an issue.

The corner finding algorithm proceeds as follows. First, on a reset, temporary registers that hold the coordinates of the top-left and the bottom-right corners are initialized to the maximum and minimum pixel coordinates, respectively. Next, the RGB value of each pixel is compared to a certain threshold value. Because the projection is predominantly white, if a pixel exceeds a certain threshold value, we can call it white and say fairly confidently that it belongs to the projection. Now we can compare each pixel's coordinates to our temporary top-left and bottom-right corner coordinates. If the coordinates exceed that of the current value we have registered for the bottom-right corner, then this new pixel is registered as the new bottom right corner. If the coordinates are less than that of the current value in the top-left register, we assign the coordinates of this new pixel to the top-left register. This process repeats until the coordinates of the bottom- right and-top left corners have remained the same for 32 frames. At this point, the algorithm outputs these values as the corners of the screen, and these values are not updated again unless the system is reset.

The corners are permanently set after 32 frames because as we pass this 32 frame limit, the probability that a random noise signal will pull one of the corners astray becomes non-negligible. Through repeated testing, this 32 frame limit has been determined to be very close to an optimal value to guarantee that the corner finding system will find the corners and hold them there well before any noise signal could affect them.

Figure 3 summarizes the image processing design in block diagram form. The connections between the initialization FSM to the finding modules are the *roof* and *floor* parameters and are 24 bits each. The connections between the Cornerfind module and the finders are the coordinates of the top-left and bottom-right corners, each of which is 19 bits. The combinational logic block on *find_en* connects to *choose*, a signal that decides which finder to enable. The combinational logic block on the output chooses whether or not to pass the pointer coordinates through the rendering block and which finder's coordinates to output.

Figure 4 is a screenshot of the image processing in action. Here, the green squares mark the coordinates the corner finder finds for the corners, and the top-left corner of the red square represents where the pointer finders have judged the laser pointer to be (it is slightly off because the pointer was moving).

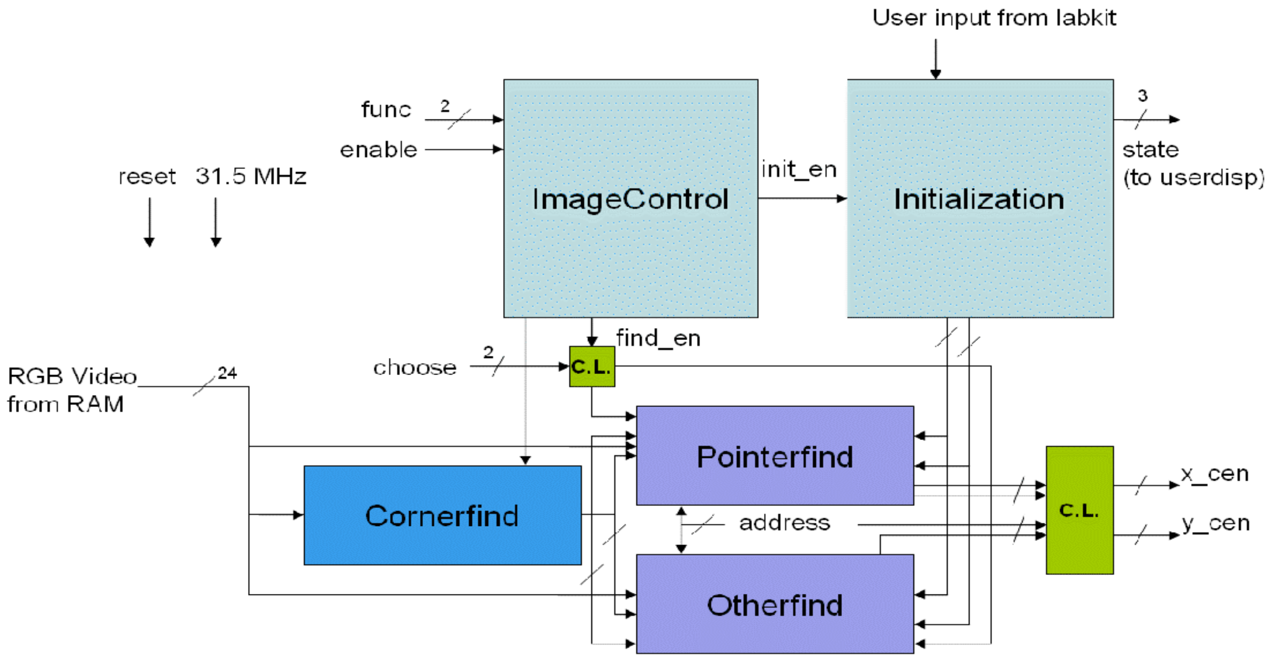

Figure 3: Image processing block diagram.

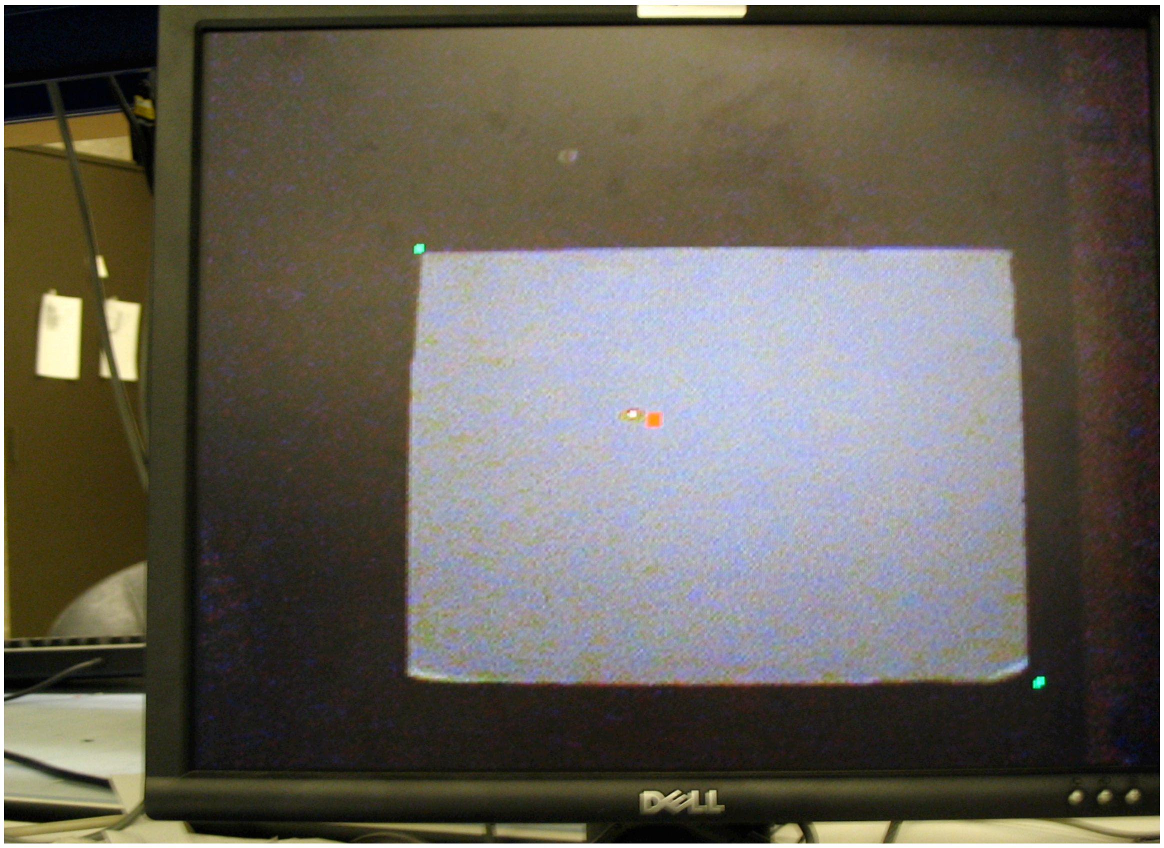

Figure 4: Screenshot of image processing on the VGA display.

Rendering (*Xinpeng Huang*)
The rendering module takes the location of the laser pointer as detected in camera coordinates and transforms it into screen pixel coordinates. Provided the laser pointer is within the boundaries of the wall projection, a linear transformation is used to calculate the coordinates of the laser pointer as if the projection formed the boundaries of the computer screen. Initially, the corner finder sends in corner locations of the projection, and the rendering takes the difference of the unrendered laser pointer location and the upper left corner of the projection, and multiplies it by a scaling factor in each dimension. This scaling factor in the x- and y-directions is the quotient of the actual computer screen width and height and the projected screen dimension width and height, respectively.

Control

This section deals with how the video data is retrieved for image processing, how the user interacts with the system, and how the output eventually gets to the PS/2 interface in a usable format. This section will discuss the main control FSMs and some of the more important helper modules that allow this data flow and user interaction to occur.

Control FSMs and User Interfacing
The system is controlled through a simple major-minor structure consisting of two Moore-type FSMs. The major FSM is called the *ImageControl*. The main purpose of *ImageControl* is to take in the operating mode input from the user and activate and coordinate the operation of the various parts of the

image processing block. The only minor FSM in the structure is *Initialization*. This state machine handles user input when the system goes into *adjust* mode.

The major FSM *ImageControl* is responsible for handling the user input from the labkit's switches and determining which mode to go into by setting the appropriate enable signals to the image processing block. The system has three modes of operation: *idle*, *find*, and *adjust*.

The *idle* mode is entered immediately on reset. In this mode, the system finds the corners of the projected screen—the pointer finding modules are disabled and the corner finding module is active. Switches one and two on the labkit allow the user to change modes of operation. If switch one is set high with two set low, the system will go into *find* mode.

In *find* mode, corner finding is also enabled, but because the corners are likely not changing at this point, the finder is continually outputting the same corner coordinates. The pointer finding modules are also enabled, and the user can select which algorithm to run using switches four and five. If switch four is low, the algorithm *pointerfind* is active, and if switch four is high and switch five is low, *otherfind* is active. If both switches are high, then the average of the results of the two algorithms is used to calculate the laser pointer's position.

In *adjust* mode, the user has the ability to reprogram the *roof* and *floor* parameters we discussed when we addressed color-matching. In, adjust mode the right pushbutton can be used to select the part of the *floor* or *roof* values corresponding to R, G, or B, and then the up and down buttons are used to increment or decrement the value. Additionally, the left button allows the user to increment or decrement the most significant byte of each of these values. The new values are automatically saved and used when the user switches back into *find* mode. In *adjust* mode, the minor FSM *initialization* is also enabled. This FSM handles the user input to adjust the *roof* and *floor* parameters of the image processing block.

Interface Modules
We have implemented several important interface modules that allow the user to interface with the system and that package the laser pointer location data in an appropriate format to be sent to the PS/2 output block.

The alphanumeric display on the labkit is used for user interfacing. The display shows the mode the user is in—*idle*, *find*, or *adjust*—and when the user presses the right button in *adjust* mode to look at the value of a different *floor*/*roof* parameter, the display will show the name of the parameter and the value. For the alphanumeric display we used the modified lab3 code (modified by Chris Buenrostro); this module was called *alphanumeric_display*. To display character strings, a module written by Isaac Rosmarin was used; this module was called *alpha_display*. The *alpha_display* module takes in either ASCII string data or a hex number and creates the appropriate dots signal to send to the alphanumeric display. We created a module called *Userdisp* which takes in the state of the *initialization* FSM as well as the *func* input from the user to choose what string to display—either *idle*, *find*, *adjust*, or one of the *roof*/*floor* parameter names and the associated value. The *Userdisp* module then sends the appropriate ASCII/hex data to the *alpha_display* module, which, in conjunction with the *alphanumeric_display* module, writes the data to the display. Figure 5 is a snapshot of the alphanumeric display in *adjust* mode.

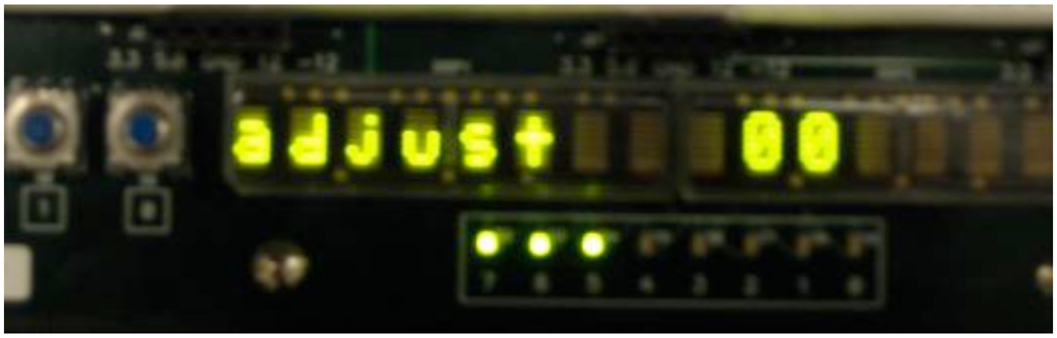

Figure 5: Alphanumeric display in adjust mode.

A module called *mousedata* is used to package the laser pointer location data into a format the PS/2 can send to the computer. The function of *mousedata* is to send the PS/2 interface change in *x* (*dx*) and change in *y* (*dy*) information as required by the PS/2. The PS/2 interface sends data to the computer at 13.5 kHz, while the laser pointer location data is coming in from the image processing block at a much more rapid rate. Accordingly, the PS/2 block sends an *update* signal when it is ready for new mouse data. When this *update* signal goes high, the *mousedata* module takes the current laser pointer location coordinates from the image processing block and subtracts these coordinates from the previous update to obtain *dx* and *dy*. The module then outputs these values along with a data enable signal to the PS/2 system. One nuance to this described behavior is that on reset, the PS/2 interface will not request an update unless something has been written. Therefore, there is a level to pulse module included in the *mousedata* module to detect a reset, so that it can set a data enable signal and write zeros for *dx* and *dy* to the PS/2 block as an initialization packet.

Figure 6 summarizes the control block with an overall block diagram. Note that the *userdisp* module actually has 16 outputs—one for each character on the alphanumeric display. These outputs are named *af*-*qf*, and each one passes to an *alpha_display* module. All 16 of these modules are omitted for clarity.

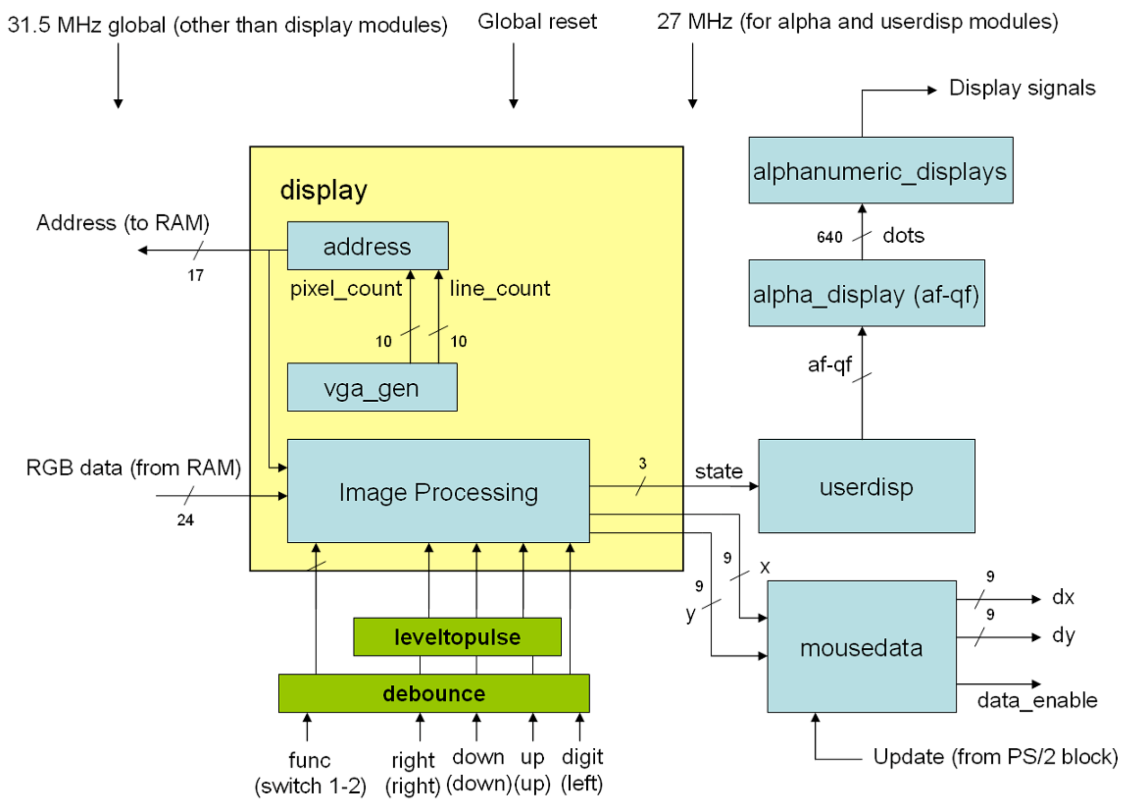

Figure 6: Control block diagram.

PS/2 Mouse Interface

Xinpeng Huang

After the image processing finds the location of the laser pointer and translates it into a pixel location on the screen, the system needs to move the mouse pointer there. This is done through PS/2 interface with the computer. We choose PS/2 because it is specialized for only two things, keyboards and mice, making it far less complex than USB.

PS/2 is a bidirectional serial protocol that allows communication between the mouse or *device* and the computer or *host*. The PS/2 interface carries four lines between the device and the host: a *data* line, a *clock* line, a *power* line, and a *ground* line. In normal operation, the mouse is able to send movement data to the host in a three-byte packet. Table 1 shows the movement data format:

Table 1: PS/2 mouse movement data format.
Reproduced from http://www.computer-engineering.org/ps2mouse/.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y over	X over	Y sign	X sign	1	Middle	Right	Left
Byte 2				X movement				
Byte 3				Y movement				

The X and Y overflow bits are zero by design, because we will never have to move the mouse pointer enough such that the data overflows. Since we do not support a middle button, that bit is also zero. The movement data in the x- and y-directions are 9-bit 2's complement numbers, but the most significant bit of each is packaged in the first byte.

The PS/2 is subject to a set of rigid timing constraints. Timing diagrams for device-to-host and host-to-device communication are illustrated in Figure 7.

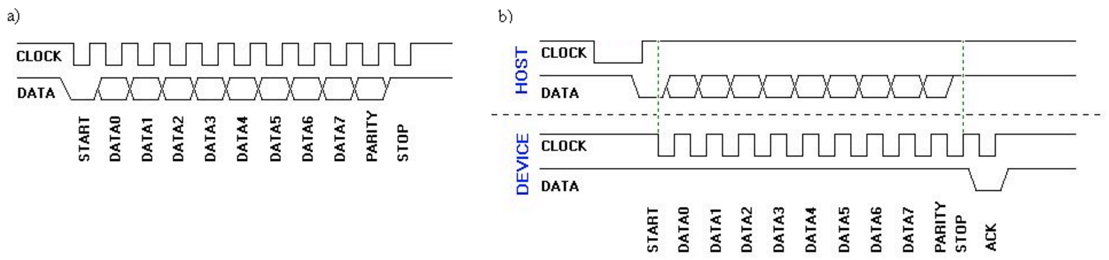

Figure 7: PS/2 communication timing diagrams.
a) Device to host communication. b) Host to device communication.
Reproduced from http://www.computer-engineering.org/ps2protocol/.

The default idle state of the PS/2 occurs when both data and clock are high. When the device is ready to send data, it generates a series of clock signals with frequency range 10-16.7 kHz and dumps bits onto the data line at every positive clock edge in the following order: 0 (start bit), 8-bit data (least significant bit first), parity bit, and 1 (stop bit), for a total of 11 bits. The parity bit is 1 if there are an even number of 1's in the data and 0 otherwise, and is used by the computer for error detection.

The device generates the clock signal, but the host can inhibit communication at any time by forcing the clock line low. If the host then pulls the data line low, it is telling the device to prepare to receive data. When the host then releases the clock line, the device must begin generating clock signals and sampling incoming data on the rising edge of its clock. The data is received in the same bit order as it is sent. After the entire 11-byte packet has been transferred, the device needs to reply with an acknowledge bit to indicate receipt. The host needs to communicate with the mouse in this way on startup in order to determine if it is indeed connected to a mouse which is functioning correctly.

Our implementation of the PS/2 communication system follows the above specifications. The goal is to take in *dx* and *dy* movement data from image processing and send the appropriate signals to move the mouse cursor on the screen accordingly. Figure 8 illustrates a block diagram of our design.

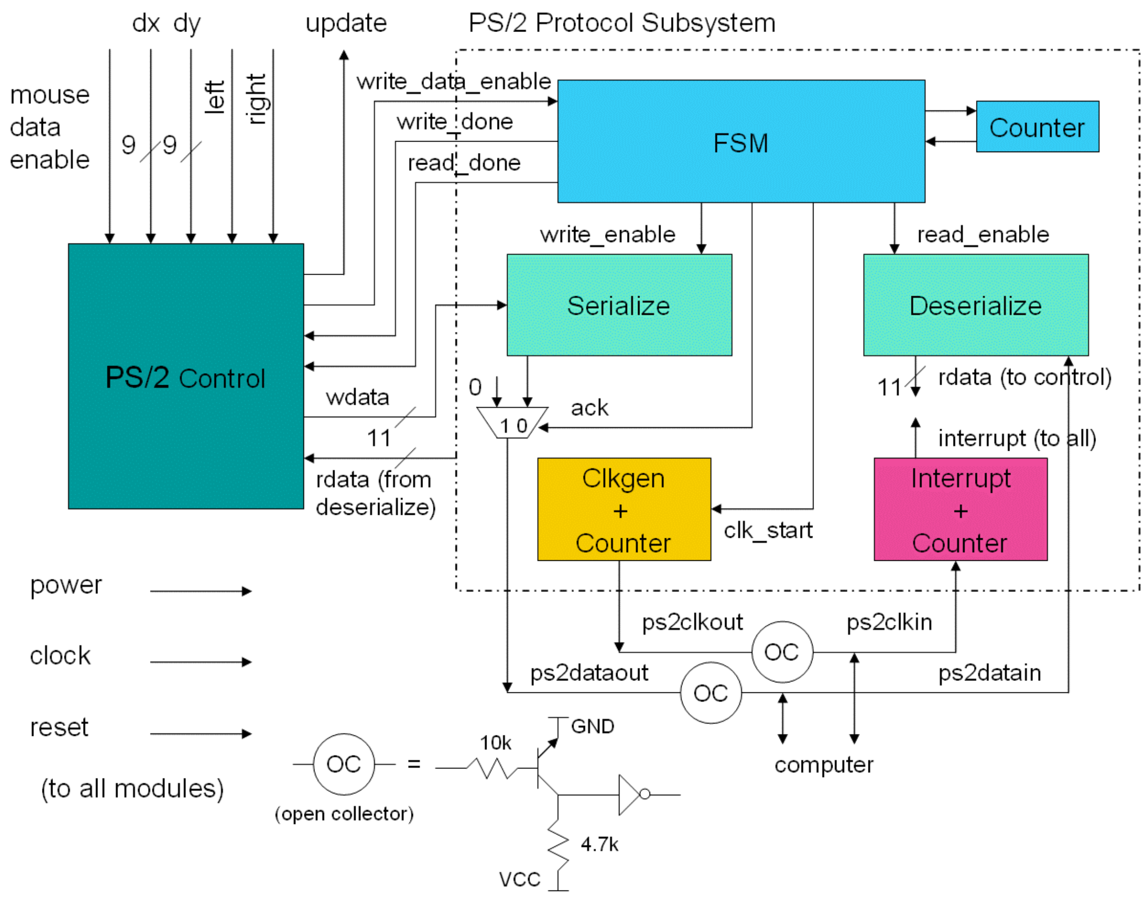

Figure 8: PS/2 mouse system block diagram.

The design is very modular and is broken up into two main parts: the controller and the protocol. Below is a short description of each component and how it interacts with the system to successfully emulate a mouse.

Controller

The controller is an FSM that takes in mouse data and an enable signal from the image processing block and communicates the appropriate data packet to be sent to the protocol. On startup, it looks for power from the computer, and once powered, sends a sequence of self-test (0xAA) and mouse ID (0x00) signals to the host to identify itself.

The host at the appropriate time will request the device to resend or repeat information by sending commands. The controller responds by telling the protocol to write an acknowledge byte (0xFA), and then sends a response to the host's command if prompted. The recognized host commands in our implementation are[2]:

0xFF – Reset – respond acknowledge (FA), self-test (AA), mouse ID (00)
0xF2 – Get ID – respond acknowledge (FA), mouse ID (00)
0xF5 – Disable Data – respond acknowledge (FA), mouse cannot send movement data
0xF4 – Enable Data – respond acknowledge (FA), mouse can then send movement data

[2] Reproduced from http://www.computer-engineering.org/ps2mouse/.

When enabled, the controller is free to send movement data packets to the protocol to be written. Upon completion of all three bytes of movement, the controller pulses an update signal back to image processing, signaling it to update the position of the mouse cursor and send in new movement data.

Protocol

The protocol handles the clock generation and the reading and writing of data onto the data line when appropriate. There is an overarching FSM that controls the current state of the protocol and sends signals that activate each of the other modules.

At the beginning of its cycle, the FSM uses a counter to determine how long the clock line remains high. If this reaches 150 us and the host has not interrupted, then the FSM enters the *write* state, where it tells the clock generator to begin generating PS/2 clock signals and commands a *serializer* to begin dumping bits onto the data line on a rising PS/2-generated clock. If the host does not do anything after the data is sent, the FSM returns to the *idle* state until it receives notification from the controller to send another byte of data.

If the host pulls the clock line low at any time, the current state of the FSM is stored into a *prev* register, and the entire protocol hangs. If the clock line goes up, the FSM is restored to its previous state and continues operation. If, however, the host pulls the data line low and releases the clock line, the FSM will enter the *read* state, where it tells the *clock generator* to generate clock pulses and commands a *deserialize*r to read bits from the data line. After the read operation, the FSM enters two acknowledge states, where it sends out an *ack* signal. This *ack* signal operates a multiplexer that will pull the data line low for an additional clock cycle, telling the host that it has successfully received the data.

The *serializer* and *deserializer* are simply shift registers that shift on every rising edge of the generated PS/2 clock. The *serializer* dumps a bit on the data line and shifts the remaining packet to the right, while the *deserializer* shifts the data line value left by the appropriate number of bits and adds it to its partial data packet, every time making sure the new incoming data the most significant bit of the current packet.

The clock generator takes an enable signal from the FSM and interfaces with a counter in order to output a clock signal that is within the range of 10-16.7 kHz (equivalent to 30-50 us high, followed by 30-50 us low). For our implementation we choose a middle value of 13.5 kHz. Finally, there is an *interrupt* module, which is basically just another counter that activates if the clock line is low for more than 100 us. This *interrupt* signal freezes every other module in the protocol until the host either releases the clock line or puts in a request to send data by pulling the data line low.

In order to simulate the bi-directionality of the clock and data lines, there are actually two different signals of each. The clock generator's output goes through an *open collector* circuit before making its way both to the computer and back into the protocol into the interrupt module. Similarly, the *serializer*'s data output goes through another open collector and then dumps into the computer and the *deserializer*. The open collectors are necessary so that the host can safely pull down both lines without contention on the outputs of the clock generator and the *serializer*, whereas the *interrupt* and *deserializer* modules will still correctly read what is being generated, unless the host is interfering.

Testing and Debugging

Testing and debugging is a crucial part of any project. In this section we will briefly describe our experience testing and debugging the various parts of our system.

Video Input Buffering

Xinpeng Huang

The video input buffering was the first part of the project implemented, because the image processing depends heavily upon it. When I first hooked up the camera and tried to get streaming video to display on the computer, it was scrolling vertically and had horizontal shifting noise. The vertical scrolling I fixed eventually by looking for the XY = 27C to reset the line count, instead of counting the total number of lines and looping around when I thought it was done. The horizontal noise was eventually determined to be due to a malfunctioning camera, and a different camera solved the problem.

Image Processing and Control

William Putnam

Testing and debugging played a major role in the development of the image processing component of the system. The image processing portion of the project was less documented than the other aspects of the system. Accordingly, the process of developing my pointer finding algorithms was an incremental one. I would come up with an idea implement it, test it, and then try to improve it. I went through many iterations of this process and tried many different ideas and combinations of ideas in my algorithms. Finally, I arrived at the fairly simple, yet effective solution presented in the image processing section.

It should be clear from the image processing section that the algorithms depend on a variety of parameters—the most obvious being the color ranges, *roof* and *floor*. To determine adequate values for these parameters I took the RGB color data and compared it to values assigned by the labkit's switches. If the color data was in the range specified by my switches then the screen would display red otherwise the screen would display black. I repeated this process with different settings for the switches until I saw as much of the laser pointer and as little of everything else as possible. Testing was also crucial to determine appropriate values for other parameters mentioned in the image processing section in particular the 32 frame wait time before the corners are set was determined through numerous tests with a variety of different values.

While testing and debugging was absolutely integral to the development of the image processing part of the system, it did not play a major role in the development of the control unit. The control unit actually served more as a tool for testing and debugging the image processing. The capabilities of the *adjust* mode allowed for easy testing of different parameter combinations without having to go through the time consuming process of reprogramming the entire system.

PS/2 Mouse Interface

Xinpeng Huang

Even though PS/2 is well documented, debugging was still a long and painful process. The first few days consisted of writing modules and simulating them extensively to make sure they obey exactly the correct timing constraints. After soldering the wires inside a mouse cord to a header and building the open collector circuitry, the computer would freeze on startup when I tried to connect the PS/2. The logic analyzer was used extensively as an invaluable debugging tool. The first problem was that I had been ignoring the power line from the computer and forgetting that the device can only send data when it is turned on. Fixing this problem resolved the freezing computer, but it still was not working.

The problem was that I had misinterpreted the parity bit completely, on multiple levels, and was continuously sending the host invalid signals. I had the parity bit backwards, and after flipping it on every signal I sent, I was at last able to move the mouse pointer using buttons on the labkit. However, the mouse was not moving in the correct directions, and seemed to be behaving randomly. Eventually I found the nasty bug that when I add the all the bits in the data byte to be sent and stored it in a one-bit parity register, the parity bit was not taking the least significant bit of the addition as I had expected. I changed all the additions to XORs and it fixed the problem.

Integration

Xinpeng Huang and William Putnam

When we integrated the PS/2 system with the image processing, we initially could not get the mouse to follow the laser pointer, even though the coordinates as viewed through the logic analyzer were correct. We found eventually that we had mistakenly clocked the PS/2 at 27 MHz and the image processing at 31.5 MHz. Clocking everything at 31.5 MHz fixed the problem. Incidentally, this mistake was also related to a jumping problem William experienced in his alphanumeric display, and the correction fixed both problems.

Conclusion

Overall, the project was a great success. It is amazing that we were able to create a system that detects a laser pointer on a screen and moves the mouse cursor when the laser pointer moves. This project and the development of this general idea can have important consequences in the future for controlling computers remotely, especially during presentations. Completing this project felt like a great accomplishment.

In retrospect, there were a number of things we could have improved upon given more time and resources that would result in a more robust and versatile project. We had considered for a time to interface to ZBT RAM instead of block RAM because it would be faster and we could store a lot more information in both resolution and color, but ultimately gave up on it because time was short. This could potentially be a great improvement to the image processing, because it may be possible to have far more accurate laser pointer finding and more accurate thresholding.

In addition, there were two extensions we had set upon at the beginning of project design that never had a chance to be implemented. Currently, left and right clicking is done through buttons on the labkit. It would have been nice to be able to click from a wireless handheld device, such as a remote control, to fit more with the control-at-a-distance theme of this project. In addition, the idea of being able to intercept data feeding into the projector in order to draw over certain pixels on a projected screen was a very cool idea but did not have time to be implemented, but would be seriously considered in any future work on this idea.

We would like to thank Gim for the mouse cord and his resources in helping us debug multiple parts of this project. We would also like to thank our TAs Javier, Theodoros, and Jae for their help, time, consideration, and for keeping the lab open for extra hours the weekend before the due date. Without them this project would have been a lot harder!