Color Tracking Robot

Dawit Bekele

Han Liang

Alison Leonard

Edward Mung

Embedded Systems 4840

Stephen Edwards

May 10, 2006

OVERVIEW

Computer vision is a hot research topic. One imagines a future with smart computers everywhere processing data about their environment. There are thousands of vision algorithms, many of which are quite advanced. Here, we implement one of the simplest schemes: color tracking. Vision with color tracking is concerned with color data only. No complicated analysis of the image is necessary. The algorithm does not differentiate between shades of color or separate objects with the same color. It simply reports which parts of the image are above a color threshold and which parts are not.
We show the output of our color data by directing the motion of a Lego Mindstorm robot. The robot moves forward when the color is concentrated in the center of the screen, or covers the screen. It moves to the left when color is concentrated in the left half of the screen and moves to the right when color is concentrated on the right. One could certainly do more detailed motion controls, however we want to do the simplest possible thing.

We use an old video camera to take in image data. The video camera feeds into the Phillips SAA7114H chip on the XSB300 board. The Philips chip translates the pixel data into two 8bit streams which it passes to the FPGA. We perform color thresholding and pixel "on" counting on the FPGA and pass the results to a C progam on the microblaze. The C program uses the pixel "on" counts to make a decision about which direction to move in. The C program sends this directional cue to the serial port. The serial port feeds to a linux machine which uses the directional cues to send commands to an IR tower connected to the USB port. The IR tower communicates with the Lego Mindstorm and the robot moves. Here is the large scale block diagram:

In what follows, we will describe each segment of the project in more detail. All code used is available as an attachment.

VISION ALGORITHM

We divide the screen into a 4X4 grid. As pixels are read in we keep a running count of the "on" pixels in each grid. For this discussion, we will call the 4x4 matrix of pixel counts 'M'.

The software part of the system takes this matrix M, and yields a decision. Let D be the set of decisions the system makes, then the software part is just an implementation of the function A, where $A : A\_M \_\_ d \_ D$

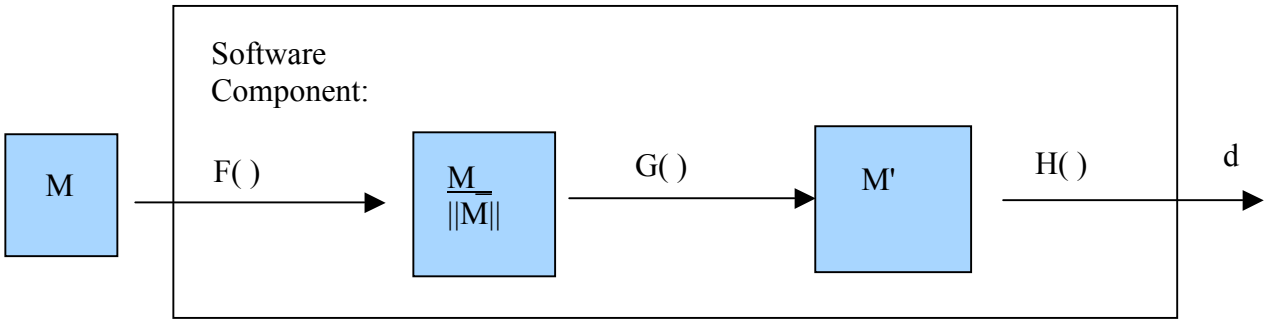
In our project, D is {"left", "right", "straight", "hold"}.

Each entry in M, represents the frequency of "on" pixels for the corresponding block of the image. The greater the number, the more of the target is spotted in that associated block. We wish to decide which of the large blocks are "on". We find the average of the pixel counts, and then turn all blocks above the average to "on". We also set a lowest number threshold such that all block counts above 300 are set to "on". This threshold combats the problem of counts that are all high, or are clustered around the average. In this case, we want all counts that are close to the average register as "on", whether they are above or below. We found that the system leaned to the side of low sensitivity, so we set the threshold quite low. We also check to make sure the average is above 20 pixels. If a only few pixels are on in a block, it is unlikely that the block actually contains an object. If the average is below 20, all blocks are marked 'off'. The result or this processing is a matrix M', whose entries are binary.

Matrix M' is examined, and if the left side has more on blocks, the command we issue is "left". If the right side has more on blocks, the command we issue is "right". If they are

equal, the command is “straight”. If there are no on blocks the command is “hold” , ie stop.

The following diagram illustrates the processing in the software component:



F( ): Normalization

$$let\ \bar{x} = \frac{(\sum_{i,j} x_{i,j})}{(n*m)},\ where\ M\ is\ n\ by\ m,\ output: \frac{M}{\bar{x}}$$

G( ): Abstraction to Binary

$$output:\ \ x'_{i,j} = 1\ \ if\ x_{i,j} > 1;\ \ 0\ \ otherwise$$

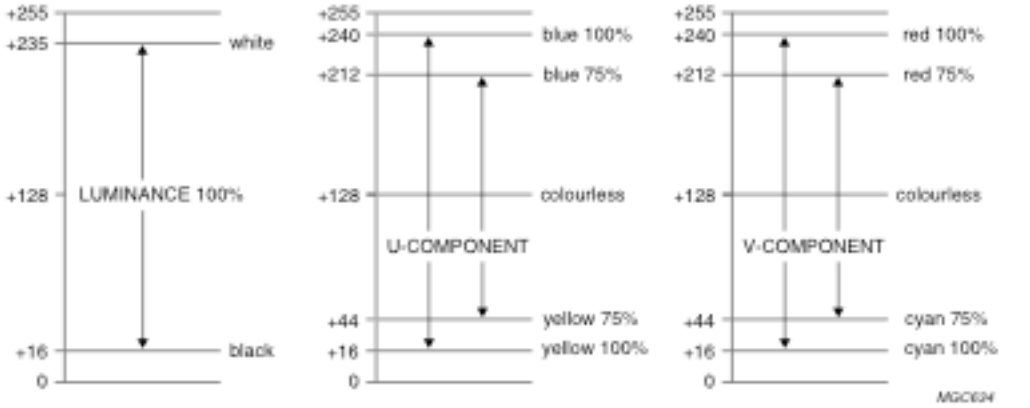
($x'_{i,j}$ is entry in row i and column j in M')

H( ): Direction Decision Making

$$let\ k = floor(\frac{m}{2});$$
$$let\ p = \sum_{i,j \le k} x_{i,j};$$
$$let\ q = \sum_{i,j > k} x_{i,j};$$
$$output:\ \ d = \text{'left'}\ \ if\ p > q,\ \ d = \text{'right'}\ \ if\ p < q,\ \ d = \text{'straight'}\ \ if\ p = q$$

THRESHOLDING DECISION

To decide on the appropriate threshold, we ran images with a red object against different background in MATLAB to test the threshold boundary. We decided to use the value 0xA0 in the chrominance V as the lower bound in order to consider a red object. Chrominance V ranges from cyan to red. See the diagram below. In addition, we limit chrominance U to the region between 0x70 and 0x90, this is because 0x80 is the colorless region for chrominance U. Chrominance U ranges from yellow to blue. We do not want to select pixels with combinations of red and other colors, eg purple. In our scenario, luminance value is not of concern because the brightness of red will not affect our color detection.



a. Y output range. b. U output range (C_B). c. V output range (C_R).

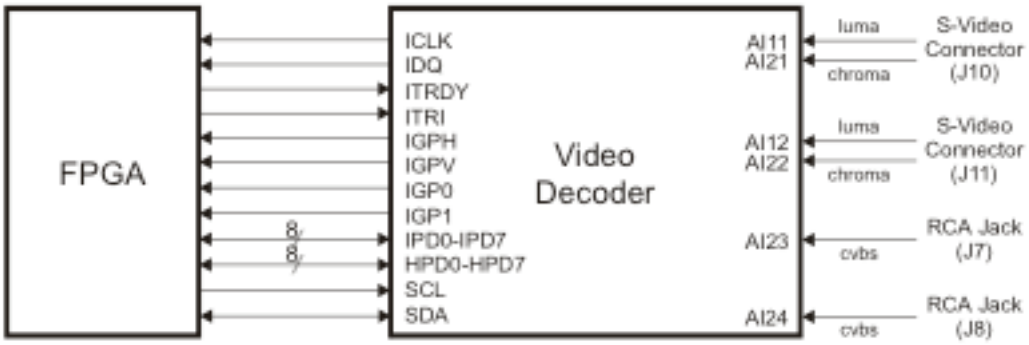
SET REGISTERS WITH I2C PROTOCOL

The Phillips video decoder chip has many possible input and output formats. The chip can do preprocessing of the signal such as trimming the image. Registers on the chip control the input/output settings. These registers must be set properly in order to get good data from the video decoder.

We write to the registers using the I2C protocol. This is a 2 pin protocol that is tailored for slow clocked systems. The protocol is widely used due to its simplicity. One pin is a clock pin, and the other sends and receives data. The registers must be set each time the chip is used.

Our group did not write the code to set up the I2C communication and set the registers. We borrowed it from a previous group. (JAYCAM '04 I believe)

COMMUNICATION: VIDEO DECODER TO FPGA

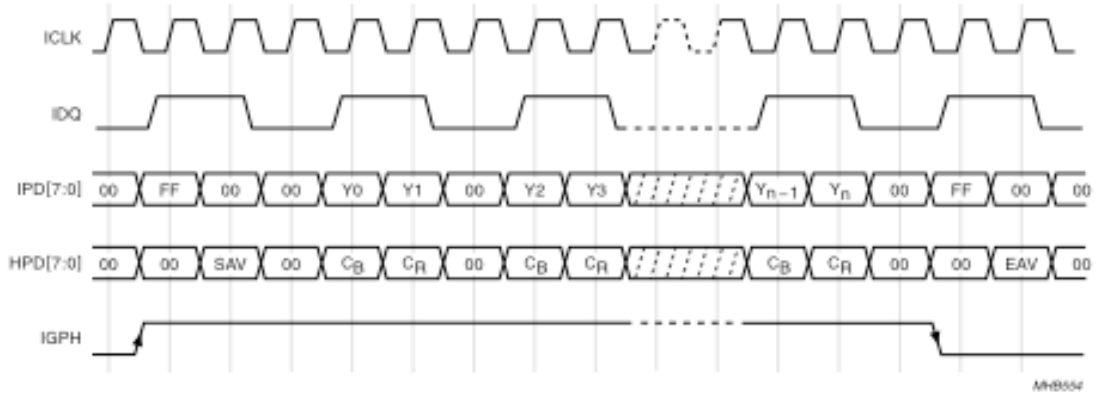


Video arrives at the FPGA over two eight bit buses. The IDP bus is for luminance values (Y) and the HDP bus is for chrominance values (UV).

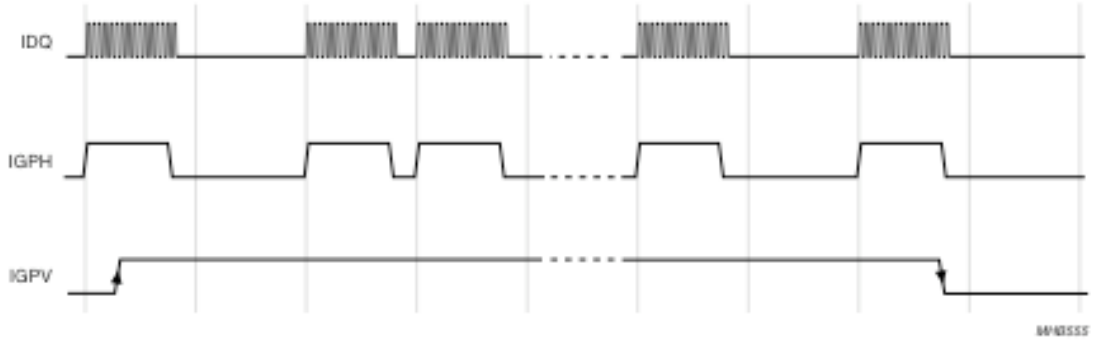
The ICLK signal clocks received pixels from the video decoder. The FPGA gets 16 bits every clock cycle. The IDQ signal indicates whether the data on the IDP and HDP buses is valid. The pixel data is transmitted with varying size gaps, so the IDQ is crucial to operation. The IGPH (horizontal reference) and IGPV (vertical reference) signals tell when we are at the end of a frame and the beginning or end of a line. These signals are crucial for processing. IPGH is on whenever the horizontal line is in an active region. Active meaning within the range of image pixels rather than video protocol regions such as front porch, back porch, etc. In other words, IPGH turns on at the beginning of a line and off at the end of a line. IPGV is on when the *frame* is in an active region. IPGV turns on at the beginning of a frame and off at the end of a frame.

The ITRI and ITRDY signals are kept at "1" at all times.

Timing Diagram for Image Data coming to FPGA



This timing diagram shows the beginning and end of a line. SAV means Start Active Video and EAV means End Active Video. The diagram shows the role played by IDQ and IPGH. IPGH indicates the beginning and end of each active line.



This timing diagram shows the role of the IPGV signal. It indicates the beginning and end of each frame. Also, we see that IDQ is constantly going on and off in the middle of active video lines.

VIDEO PROCESSING ON THE FPGA

All signals from the video decoder are buffered and latched on the FPGA. This increases the stability of the design.

VIDEO DECODER INTERFACE

The latched signals are sent to a block called the Video Decoder Interface. We did not write this block. We believe it was written by the TA for the '04 class. The Interface provides useful data for further image processing. It counts pixels and lines, distinguishes between adjacent frames and reports a line filling level, which is the amount of the current line that has been read. We use the pixel and line counts for our project.

VIDEO PROCESSING BLOCK

Our video processing block takes in the latched data signals, and the relevant counts from the Video Decoder Interface. We process two pixels at a time. As explained above, there are two chrominance values which are each sent every other pixel. We wished to process all chrominance information at the same time. We store the first pixel in a register and then when the second pixel arrives, we send it with the first pixel to the thresholding block. This requires dividing the clock by two. The first pixel storage register is clocked every two ICLK cycles. All other blocks on the video processing unit are clocked on the opposing every two ICLK cycles. We can think of the new clocking system as having a 1 2 1 2 1 2 1 2 … pattern instead of 1 1 1 1 1 1 1… .

The thresholding device uses a threshold that is sent from the user via the OPB bus. The threshold can be sent at the beginning of operation. It is stored in a register and input to the video processing unit at every clock cycle. The thresholding unit compares the 32 bits of pixel data to the given threshold. In our case, we had no luminance threshold but had 3 chrominance conditions, as explained above. The pixels are then marked "on" or "off". This one bit of information is output to the pixel counting block.

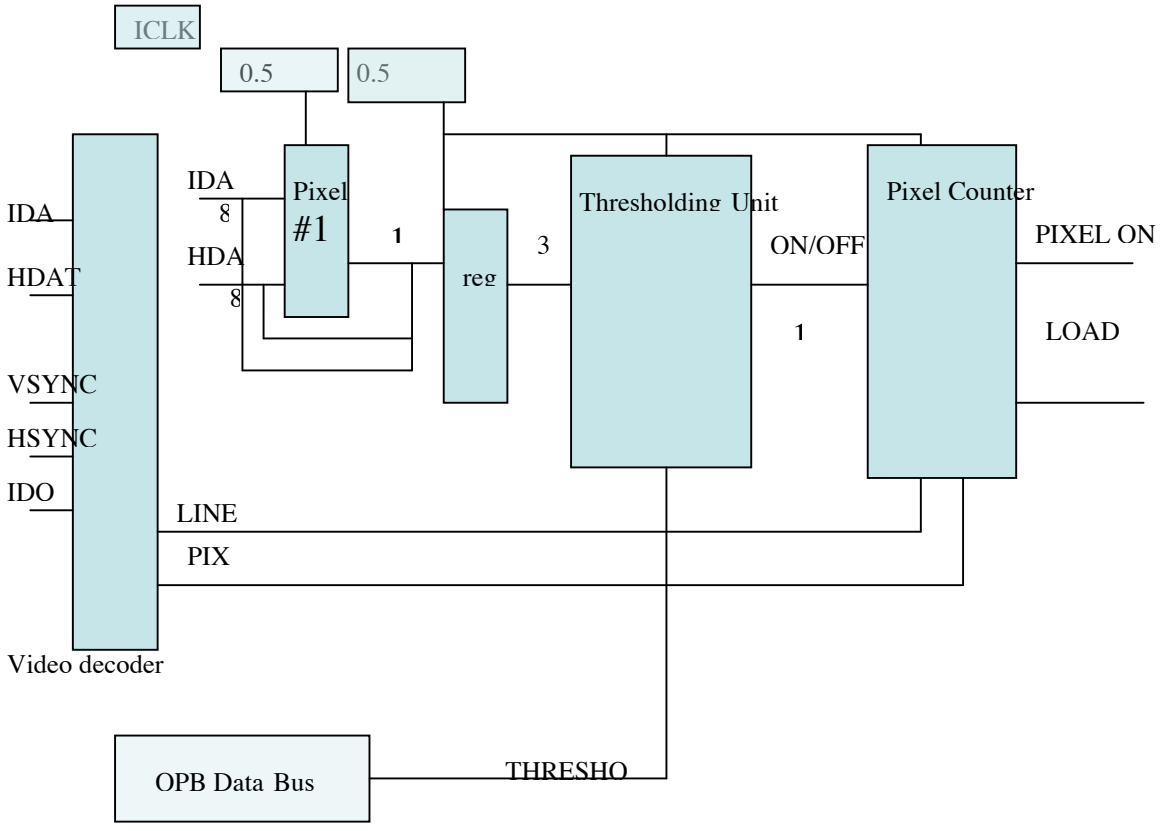
The pixel counting block counts pixels on a 4X4 grid as explained above. We use the pixel count and line count data to know which grid we are in. At the end of each grid, we send the total count for that grid to a register outside the video processing unit. Since the video data reads line by line, we only need to keep track of 4 grid counts at a time. We reuse the grid signals in order to keep our code more compact. The grid count output signal sends the count from each grid in turn. We create a signal that indicates which set of grids we are in (1-4). This signal allows us to send correct register load signals without writing extraneous conditional statements.

There are 16 registers on the FPGA to hold the pixel count data. All 16 have as input the grid count coming from the video processing unit. The are controlled by individual load signals. A demux makes a decides which load signal to turn on. The load ouput signal from the video processor is the demux select signal.

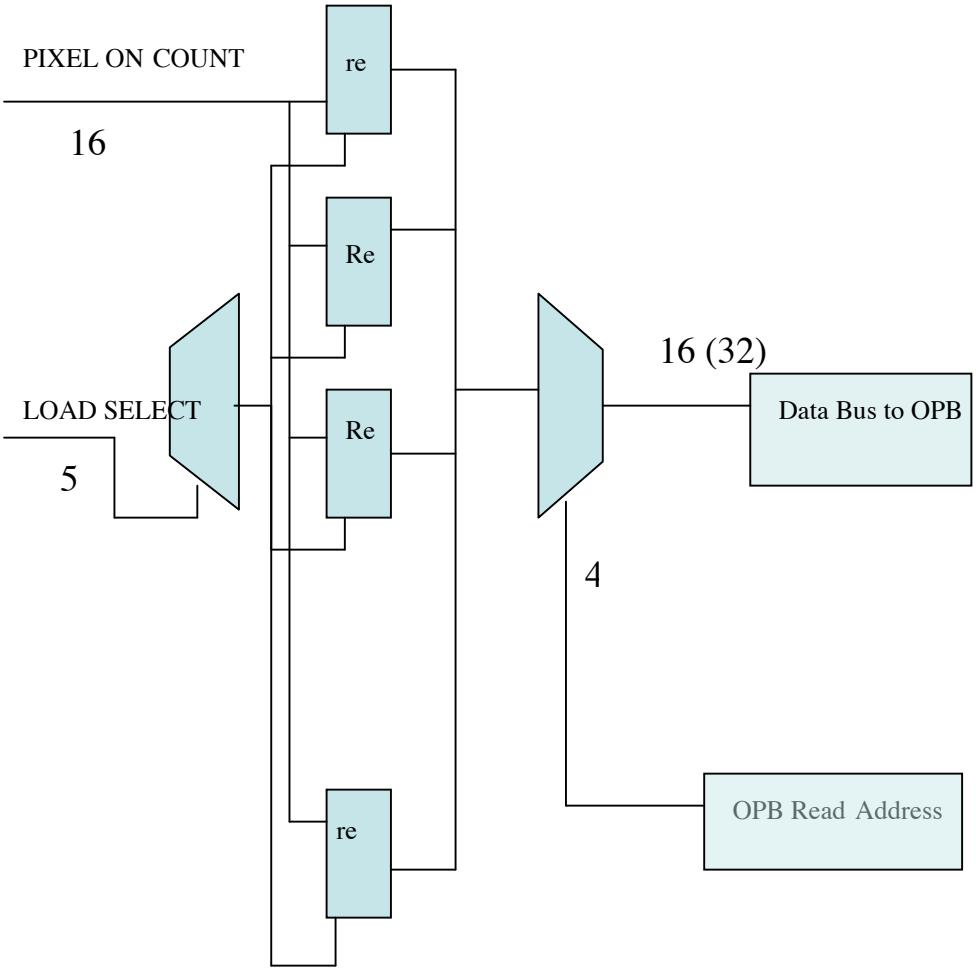
OPB VIDEO DECODER

OPB video decoder is like the main() file for the VHDL video processing. It talks to the OPB bus and links the sub-units together. The frame for this program was written by the TA from '04 (I believe). However, we made significant changes. The program latches and buffers the signals coming from the video decoder. It also holds the registers for the threshold and the pixel count data. The OPB bus takes data from the pixel count registers and sends it to our C program for further processing. All register outputs are connected to a multiplexer. 4 bits of the OPB read address are the select signal for this multiplexer.

VIDEO PROCESSING BLOCK DIAGRAM



REGISTER STORAGE SCHEME FOR PIXEL COUNT DATA



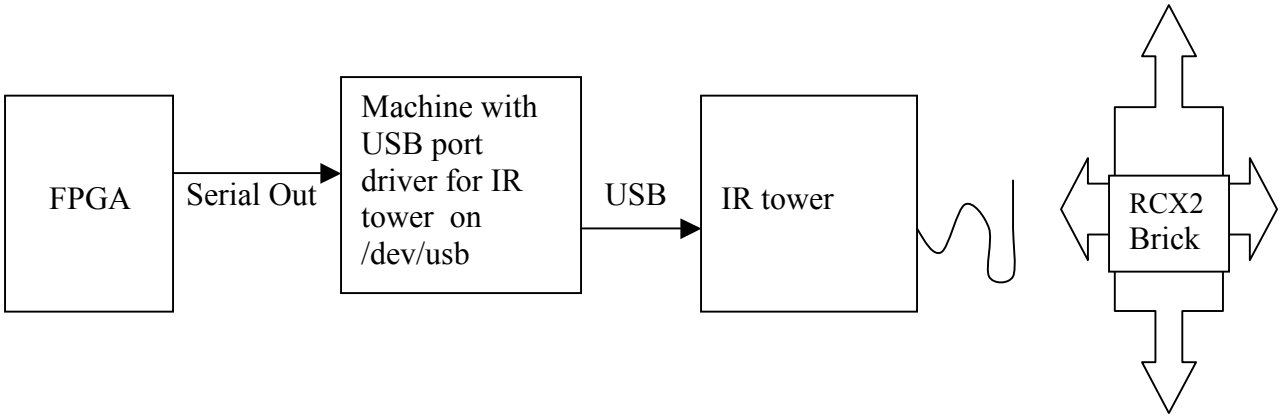
CONSIDERATIONS FOR PROCESSING IN C

When working with C code, we found that our system did not have enough memory to execute all of the commands. The original set up allocated 4k of BRAM memory to the microblaze. The program to set the registers holds a large array and takes up a majority of this memory. To solve this problem we expanded the microblaze memory allotment from 4k to 8k. This issue motivated our switching the pixel count storage from BRAM to registers. This change actually made our system much cleaner. We had been using a BRAM apparatus left over from another project and it was not optimized for our needs.

In order for the C program to operate properly, it must be able to read the register data at the correct times. This timing is accomplished using the vertical sync cue. When the program wishes to read it first waits until the vertical sync turns off, so that it knows it is

will not start in the middle of a frame. It sends the okay for the read command when the vertical sync turns on again. While executing the read, the program must continually check that the vertical sync remains on. If it turns off, then the data may be corrupted as it may be a mix of two different frames. If the vert sync turns off during read execution, the read exits.

COMMUNICATION WITH LEGO MINDSTORM



We use the Lego mindstorm robot to implement our directional cues. The computer for the mindstorm is called the RCX2, or the brick. The block diagram above shows the main components in the robot communication system.

The brick receives commands from a remote machine via an infrared tower. Theoretically, we could have an infrared tower with a serial port and we would be able to send commands from the XSB300 board directly to the robot via the serial output. Unfortunately, we could not get our hands on a serial port IR tower. Our IR tower works with USB. Again theoretically, we would be able to connect the IR tower to the usb port on the linux machine that is running Xilinx. However, this machine did not have the proper driver. So we made a new linux machine to act as a wire: taking serial data input and outputting USB data. The difference in structure between having this new machine and using the linux machine that is running Xilinx is just a matter of extra hardware. The process would be the same for either "computer as a wire".

In order for the computer to act as a serial to USB wire, we had to write a small script to read data from the serial port and write it to the USB port. This program is written in

perl. Although it was not necessary, we found it simpler to place all of the robot directional cues in this program. The FPGA sent out simply 'L', 'R', 'H', 'S'. These letters indicate left, right, hold and straight. The perl script reads this output and implements the appropriate stored directional command.
In order to issue directional commands to the brick, we needed to install a program that runs NQC. NQC stands for Not Quite C and is a protocol for communicating with the brick. We installed a linux system that allowed us to write command line arguments that would be sent to the USB port. We omit the exact command we sent to the robot. These commands can be viewed in our perl script.

WHAT WE LEARNED

If a future group wants to do a project with a robot, we suggest figuring out how it communicates with a remote machine as a first step. Our project would have been simpler had we gotten our hands on a serial port IR tower.

We thought that the lego movement commands would be easy to implement, but actually there is a bit of fine tuning needed to get the thing working properly. Proper time and seriousness should be allotted to this task.

Getting the VHDL to compile was very difficult at first. We were missing key ingredients to the structure files such as MHS, MPD and UCF. It DOES NOT work to just try to copy these files from the labs. It is important to know what these files do in order to get them to compile properly. We tried to write a lot of new code all at once. This was a bad idea. It works better to implement things one at a time. Also, we tried starting with a previous groups project and changing things one at a time from there. The best decision we made was to run simulations of our code. This allowed us to debug small errors without recompiling the VHDL over and over. Recompiling VHDL can take 20-30 minutes.

We found that subtleties in the C code were a difficult block to getting the code working. The C code needed to read data in at the right times. This is accomplished using the vertical sync function. We tried to use other cues, but had trouble getting them to work.

It was important to understand the OPB operation. We never fully understood its function from the labs, but using it is actually quite simple once one sees what is going on. The OPB is useful as a conduit from custom hardware on the FPGA to the C code. There are built in commands to send and receive data using the OPB.

PROJECT TASK BREAKDOWN

Idea for Vision Algorithm: Han
VHDL: Alison, Edward
C code: Alison Edward
Robot Stucture: Edward, group
Finding robot OPCODES: Dawit,Han
Perl script for serial to USB: Han, Dawit
Debugging Robot : group
Power Point: Han, group
Report: Alison, group

PERL SCRIPT

```
#!/usr/bin/perl

print "Launching commander\n";

$BIN_FILE = "/home/anl2103/nqc-3.1.r4/bin/nqc";
$command_prefix = $BIN_FILE." -Susb -Trcx2 -far -raw ";
#state 0: off; state 1:straight; state 2:left; state 3: right
$state = 0;

$speed_control = "130502";
$fast = $speed_control."07";
$mid = $speed_control."04";
$slow = $speed_control."01";
$slow_right = "13010205";
$slow_left = "13040205";
$stop = "2945";
$command_lefton = $command_prefix."2984";
$command_righton = $command_prefix."2981";
$command_bothon = $command_prefix."2985";

$command_leftoff = $command_prefix."2944";
$command_rightoff = $command_prefix."2941";
$command_bothoff = $command_prefix."2945";
system($slow);
open(SERIAL, "/dev/ttyS0");
#Speed Control construct, specify in command line
if($ARGV[0] eq "fast") {system($fast); print "Car will run $ARGV[0]\n"; }
elsif($ARGV[0] eq "slow") { system($slow); print "Car will run $ARGV[0]\n"; }
elsif($ARGV[0] eq "") {    system($mid); print "Car will run Default speed\n"; }
else        { system($mid); print "Car will run $ARGV[0]"; }


while($c =<SERIAL>){
            print "$c";
            if($c =~ /S/){
                        &issue(1);
            }
            if($c =~ /L/){
                        &issue(2);
            }
            elsif($c =~ /R/){
                        &issue(3);
            }
            elsif($c =~ /H/){
                        &issue(0);
            }
            else{
            }
}

sub issue{
            my $newState = $_[0];
            if($state == 0){
                        if($newState == 1){
```

```
                system($slow);
                system($command_bothon);
        }
        elsif($newState == 2){
                system($slow_right);
                system($command_righton);
        }
        elsif($newState == 0){
                # if on hold twice stop
                system($stop);
        }
        else{
                system($slow_left);
                system($command_lefton);
        }
}
elsif($state == 1){

        if($newState == 1){
                system($slow);
                # do nothing
        }
        elsif($newState == 2){
                system($command_leftoff);
        }
        elsif($newState == 0){
                system($command_bothoff);
        }
        else{
                system($command_rightoff);
        }
}
elsif($state == 2){

        if($newState == 1){
                system($slow_left);
                system($command_lefton);
        }
        elsif($newState == 2){
                # do nothing;
        }
        elsif($newState == 0){
                system($command_rightoff);
        }
        else{
                system($command_rightoff);
                system($slow_left);
                system($command_lefton);
        }
}
elsif($state == 3){

        if($newState == 1){
                system($slow_right);
                system($command_righton);
        }
```

```
                elsif($newState == 2){
                        system($command_leftoff);
                        system($slow_right);
                        system($command_righton);
                }
                elsif($newState == 0){
                        system($command_leftoff);
                }
                else{
                        # do nothing
                }
        }
        print("state transition: $state -> $newState\n");
        $state = $newState;
}
print "Commander finished\n";
```