EE 552 High Level Digital ASIC Design Using CAD

Where's Waldo Image Finder

Final Report

Instructor: Dr. Elliott

By:
James Walton (james3412@hotmail.com)
Jacob Slobodov (slobodov@ee.ualberta.ca)
Han-Jen Yang (hanjen@ualberta.ca)
David Pittis (pittis@freenet.edmonton.ab.ca)

November 27, 2001

Declaration of Original Content

The design elements of this project are entirely the original work of the authors and have not been submitted for credit in any other course except as follows:

1. USB architectural control ideas from reference [3];
2. DPLL state diagram from [7];
3. USB test bench (modified) from [3]

_____	_____	_____	_____
David Pittis	Jacob Slobodov	James Walton	Han-Jen Yang

Abstract

This document describes the design, implementation, and verification of EE552 “Where is Waldo” project. The project is to construct image recognition device built on an Altera Flex10k FPGA which is mounted on an Alerta UP1 broad. The image recognition device utilizes the external input and output devices and the self-developed image recognition algorithm is configured into FPGA. The image recognition device will take the target and candidate images from the Logitech Quickcam Express camera and transform the image from 16-bit colour into 3-bit colour. Then the image will stored into external SRAM and being processed by the recognition algorithm. After the image searching is done, the result will be output onto the Monitor for found image and LED for percentage match.

Table of Contents:

Achievements

Originally, this project was supposed to be an image recognition system working in real time. However, due to significant obstacles such as proprietary (and unavailable) information about the transfer protocols for our USB camera, we were forced to abandon the real time approach. Instead of using real time processing, we did an offline technique.

Also, our original goal was to use some addressing tricks to simulate a zoom in and out, as well as rotation so that we can gain a more robust image recognition algorithm. This was abandoned due to lack of space on FPGA and amount of processing time it would take.

We had originally planned to use black and white images only (one bit per pixel) in our image recognition. We have increased this to Bayer RGB colour (two bits per pixel). This improvement should refine our ability to recognize the target image.

Also we originally planed to implement a complete USB host controller, but instead a USB receiver was designed. Using a wire tapping, we managed to extract required video information from USB cable. This part of the project was successfully implemented and tested in hardware.

Functional modules:

1. The USB receiver – worked in hardware;
2. Data filter – worked in hardware;
3. Image recognition – worked in simulation;
4. VGA output – functionality partially tested in hardware (displayed static images, did not try to output image stream yet).

Description of operation

Where's Waldo? The "Where's Waldo?" project uses a USB camera to gather image data and store it into memory. "Where's Waldo?" then takes data out of memory and performs a recognition algorithm on it.

The user operates the device by pushing one of two buttons. There is a "Capture Target" button, and a "Capture Candidate" button. When the Capture Target button is pushed, the system is reset and the device captures a 48 pixel by 48-pixel target image. After this is complete, the system will wait for the user to push another button. If the user pushes the Capture Target button again, another target will be captured. If the user pushes the Capture Candidate button, a 240 by 240 image will be captured. The system will then perform its image recognition algorithm and try to find the Target Image somewhere in the Candidate Image. The system keeps track of the best match. When the image recognition algorithm is complete, the system outputs the percentage of the best match to the LEDs on the board, and the entire 240 by 240 Candidate Image is displayed to a VGA screen, and shows a crosshairs where the best match is. The top-level hierarchy diagram is shown below in Fig. 1.

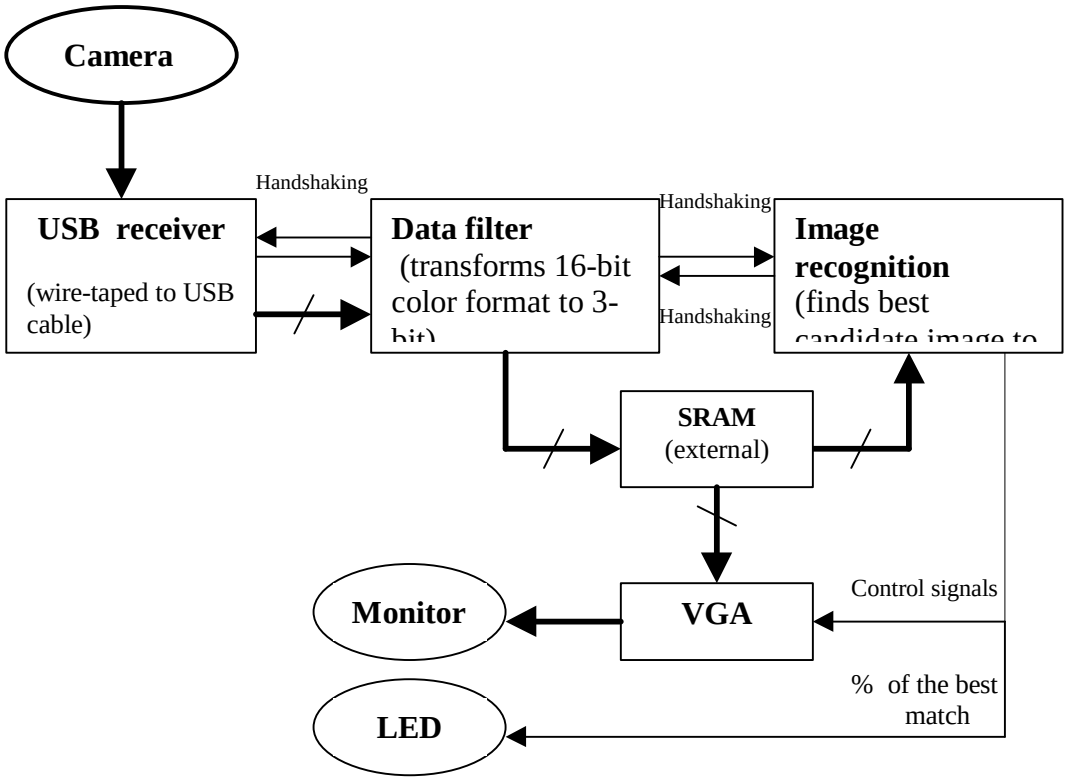


Fig1."Where's Waldo" image finder. Top-level block diagram.

The USB camera is plugged into a laptop computer, where an image capturing program is running. Since the laptop is sending the appropriate signals to the USB camera, the camera is sending image data back to the laptop. The USB receiver, shown in Fig. 1, listens to the signals sent from the camera via a wiretap. The USB receiver interprets this data and sends it to the Filter shown in Fig. 1. The data comes into the USB receiver as packets of image data as well as other extraneous information, which is

normally handled by software on the computer. Since we only want the raw image data for the project, the USB receiver has to select the appropriate packets to send to the Filter.

The data received by the Filter is in a particular format, which is described in detail in the Filter design section, later in the report. The Filter converts this raw data into a format that is understood by the Image Recognition algorithm, and stores it into the SRAM. The data is now ready for the Image Recognition part, and control is passed to that module.

The Image Recognition algorithm takes the data out of data one pixel at a time and performs a comparison. These comparisons are made between every pixel in the target image and the potential match from the candidate image, and constitute one frame. After each frame is complete, the percentage of match is calculated and compared with the best percentage from the previous frames. The Image Recognition algorithm also keeps the location of the centre of the frame that had the highest percentage of match. After all the frames have been analyzed, the best percentage and corresponding centre location will be sent to the output. The percentage is displayed directly on the LEDs on the board, and the centre location coordinates are sent to the VGA output stage. Control is passed to the VGA output module.

The VGA output module takes the candidate image data from the SRAM and displays it on the screen. The VGA output module also displays a crosshairs on the frame, which most closely matches the target image.

Implementation

USB data receiver (extractor)

Since the current project utilizes a USB video camera, it is necessary to design a device that is able to receive (extract) and decode video data from USB communication line. For that purpose the USB data receiver was designed. Tapping the USB cable, which goes from camera to PC's USB port, does the physical connection. The designed USB receiver does not send any signals to the USB cable, it just "listens" to the communication between camera and PC, analyzes USB packets and extracts only video data, ignoring everything else. The block diagram of the USB data receiver is presented on Fig2.

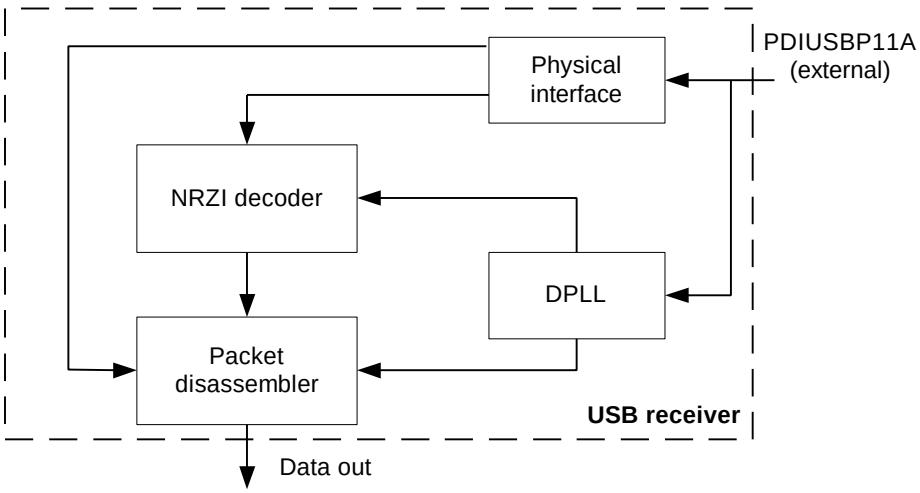


Fig 2. USB data receiver. Top-level block diagram

The top level module has entity *USB* which composed from 4 functional blocks: physical interface (PHY), digital phase locked loop (DPLL), NRZI decoder and bit-unstuffer, packet disassembler. Their entities are *pdiusb*, dpll, *nrzi_decode* and *pd* respectively. Fig3 represents entity of the USB data receiver.

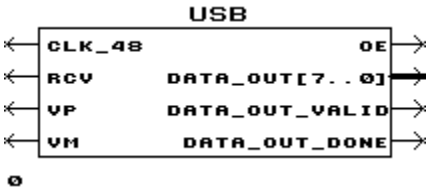


Fig 3. USB data receiver. Entity block diagram.

The entity has four inputs and outputs. External crystal oscillator supplies 48MHz-clock frequency (*clk_48*). Signals *rcv*, *vp*, *vm* and *oe* are connected to the external IC (USB transceiver). Output signal *data_out* is 8-bit wide buss outputs extracted video data, *data_out_valid* (handshaking) indicates that output data is valid and can be latched-in by the next functional unit. Output *data_out_done* is used to signal last data in a packet been received. The extracted from USB cable data is sent out from receiver in 8-bit (byte) wide chunks with distance of 8 clock cycles of *clk_dpll* (12MHz).

The top-level functional block (USB) also has POR (power-on reset) module, used to preset all signals and registers to required initial values. It forces *po_rst* signal to be high for one clock cycle shortly (3 clock cycles) after FPGA is configured.

The following parts of this chapter describe each functional module in details.

? ***Physical interface (entity: pdiusb)***

According to [1] USB employs a differential driver to drive the USB data signal onto the USB cable, as well as to receive data signal from USB cable. For that purpose the USB transceiver PDIUSBP11A is used. This IC converts differential signal levels (D+, D-) into CMOS levels. Fig4 shows a corresponding circuit diagram for the USB transceiver part.

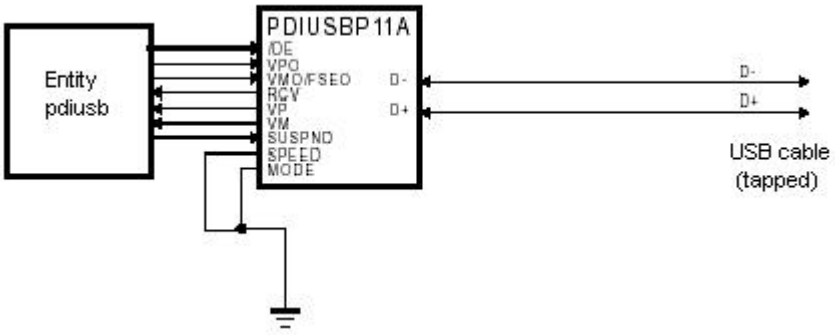


Fig 4. USB transceiver schematic diagram (modified from [3])

The entity *pdiusb* works as glue logic between the USB transceiver chip and rest of the functional units. It is responsible EOP (end of packet) recognition, SE0 (single ended zero), host initiated reset and USB line error detection. The corresponding entity diagram of physical interface module is shown on Fig5.

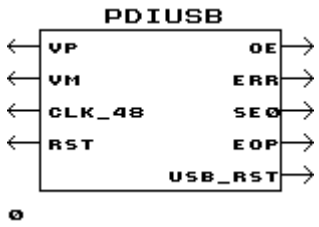


Fig 5. Physical interface. Entity block diagram.

As stated in [1], the EOP is initiated by pulling differential lines to sate SE0 (D+ and D- lines are both low) for two bit times (~160ns) followed by *J* state for one bit time. These are detected by physical interface. It check time for with SE0 was applied ad if was long enough that device generates EOP signal.

Host initiated reset on USB line is done by driving an extended SE0 at the port. If device sees an SE0 on its upstream port for more then *2.5us* it may treat that signal as a reset. The reset sensor is implemented by using counter and flip-flop. Once SE0 is detected counter starts counting and when host drives SE0 up to 120 cycles of clk_48 the *usb_rst* signal is forced to be high for one clock cycle.

? ***Digital Phase Locked Loop (entity: DPLL)***

The 12Mhz clock in the host and the USB device are asynchronous since they are derived from different oscillators. Thus, bitwise synchronization is needed, which can be achieved with the help of PLL. According to the [7], a typical implementation would use a DPLL with 4x oversampling to derive the received clock. A typical DPLL state machine, proposed in [7] is shown in Fig. 6 and runs on the 48Mhz clock. In this diagram 'a' and 'b' are the differential receiver (PDIUSBP11A) output, synchronized by a stage (latches) of the 48Mhz. Also, 'a' is latched on the rising edge and 'b' is latched on the falling edge. In this state machine, states C and D are used to lock to the incoming bitstream using the initial transitions. After lock is achieved, the DPLL circulates in either the right half of diagram (when incoming data is '1') or in the left half (when incoming data is '0'). The nominal loop is trough the four states in vertical line (5-7-6-4 or 1-3-2-0). In this loop, the inner two states (7-6 or 3-2) generate the clock high period and the outer states (4-5 or 0-1) generate the clock low period of the extracted 12Mhz clock. These four states are chosen to ensure adequate setup and hold time for clocking the incoming data. Transitions in the bitstream result in switching between the loops. If there is no change in bit width, the transition will be from states 0 and 4. If the bit width is shorter (host is running at ht e higher frequency), the transition will be from states 2 and 6 (so the low period of the derived clock is shortened by one period of 48Mhz). In case of a long bit width, the loop transition will go from states 3 and 7, through states F and B, adding an extra 48Mhz high period to the derived clock.

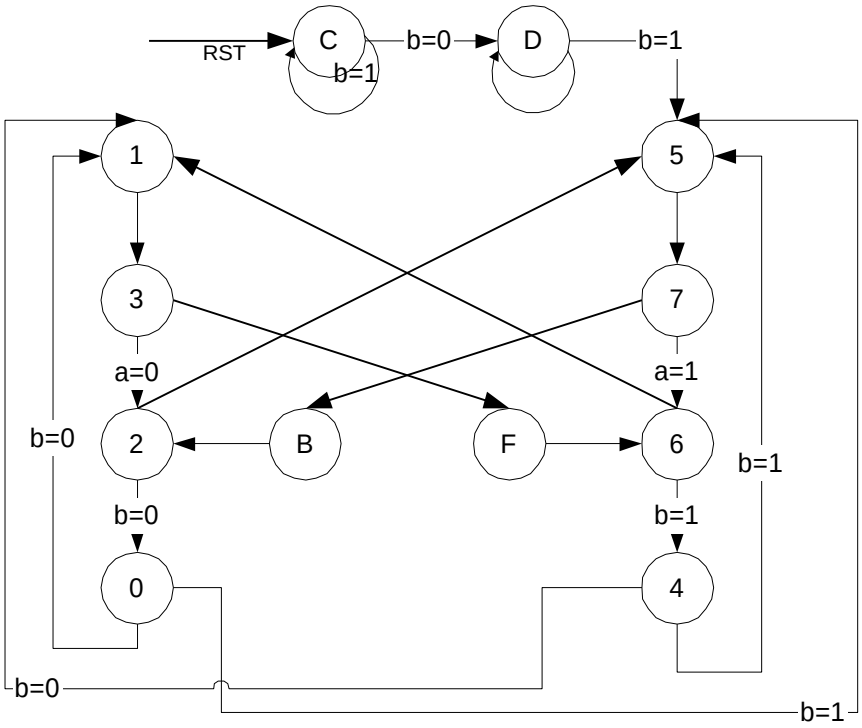


Fig 6. Block diagram of DPLL state machine implementation

The VHDL code implementation of the described DPLL is presented on Fig3 in Appendix A. The first two processes are use to synchronize (latch-in) the incoming data with 48Mhz clock. The third process is DPLL state machine implementation. Fig7 presents entity diagram of DPLL.

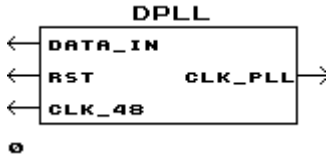


Fig 7. Digital phase locked loop. Entity block diagram.

The guaranteed DPLL clock lock during USB data transmission is ensured by employing a bit stuffing. Also, each packet has Sync pattern prefix (seven consecutive ‘0’, represented as 7 level transitions in NRZI format) used for synchronization (for more details see NRZI decoding/bit unstuffing section).

? ***NRZI decoding/ Bit unstuffing (entity: NRZI_decode)***

The USB employs *NRZI* (Non Return to Zero Invert) when transmitting data. In NRZI ‘1’ is represented by no change in level and ‘0’ is represented by a change in level. A string of zeros causes the NRZI data toggle each bit time, while a string of ones causes long period with no transition in the data. Thus, after data was received it has to be decoded (converted from NRZI to the regular bit stream). Also, in order to ensure adequate signal transitions, *bit stuffing* is employed. A zero is inserted after six consecutive ones in the data stream before the data is NRZI encoded, which forces a transition in NRZI data stream. This gives a receiver (DPLL) a data transition at least once every seven bit times to guarantee the clock lock. The receiver decodes the NRZI data, recognizes the stuffed bits and ignores them. If receiver gets seven consecutive ones anywhere in the packet, it means that bit-stuffing error has occurred and the packet should be ignored. Figure 8 shows entity diagram of NRZI decoder.

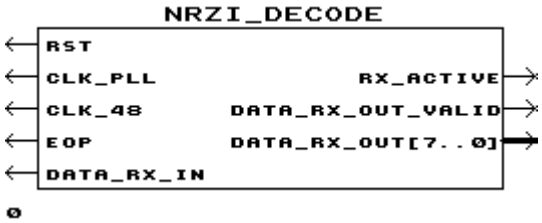


Fig 8. NRZI decoder. Entity block diagram.

The VHDL implementation of NRZI decoder is presented on Fig4 in Appendix A.
As was mentioned before, beginning of each packet has “sync” pattern (seven ‘0’ followed by ‘1’), used to synchronize internal clock with incoming data-stream. This “sync” pattern does not carry any meaningful information and has to be ignored by NRZI decoder. After SOP was detected (by *pdiusb*), the first SM (process: sync_rem) senses sync pattern and forces *sync_end* signal to ‘1’. At that time DPLL has to be locked already and it is used to latch-in incoming data. NRZI decoding is done by comparing current data bit on the input line with previously latched one. If they are the same then decoder outputs ‘1’ if not then ‘0’. As declared in [1], the last ‘1’ of the “sync” pattern has to be count as a first ‘1’ in bit stuffing sequence, but should not be shifted to the output. Thus, two enable signals (*shift_en* and *stuff_count_en*) are used to enable bit-unstaffing and shifting incoming data into output register. There is also *rx_active* signal that indicates that data can be received and that it is valid data (no errors occur). In case of bit-stuffing error (7 consecutive ‘1’) *rx_active* forced to be low until EOP is received, which prevents corrupted packets from further processing. Once incoming data been converted from NRZI format, it has to be deserialized. Deserialization is done by using 8-bit shift register, controlled by *shift_en* signal. If six consecutive ‘1’s were received *shift_en* goes low, disabling

shift register, and thus ignoring stuffed bit. After 8 bits of incoming data are in the output shift register, the *data_rx_out_valid* goes high indicating that data at the output is valid. A 3-bit counter (variable *count*) is used to count number of shifted bits and after all 8 bits are shifted it forces *data_rx_out_valid* to be '1' for one clock cycle, which is enough to let next module shift-in the output data.

? ***Packet disassembler (entity: pd)***

In USB all data travels trough physical wires in a packet format. Each packet has its 8-bit wide ID part (PID). Some packets, such as handshake packet, don't have any other field accept PID. In order to extract required information from the data stream, each packet has to be received and analyzed. Since we are only interested in image data from camera, thus data only from data packets should be extracted. The designed packet disassembler, which entity block diagram is presented on Fig9, performs such functions.

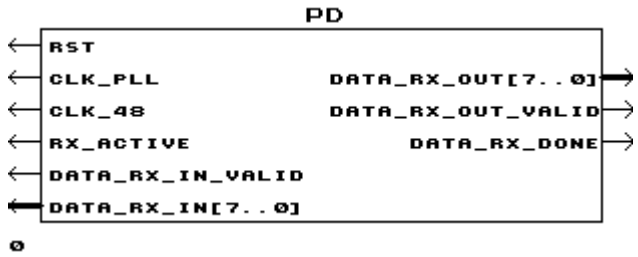


Fig9. Packet disassembler. Entity block diagram.

After incoming data been translated in NRZI decoder, it goes directly to PD. At the beginning each packet has its PID. The actual packet type is coded by four LSBs. The MSB part is LSB's bitwise complement, used for PID error checking. We are interested only in data packets, which have PID "0011" for even and "1011" for odd data packets (the data PID toggling is used for additional synchronization). Thus, only data packets will be filtered and sent out. The core of PD is a control state machine that has three states. If in *idle* state SM sees that input data is valid (*data_rx_in_valid='1'*) and system in RX active mode, it lathes input data in and goes to *active* state. Here it checks what type of packet was received and if its data packet then it goes to state *data*, otherwise SM goes back to *idle* in order to wait for the next packet. In *data* state all input data is conveying directly to the output. If end of packet was detected (*rx_active* goes low in this case), then SM returns back to *idle*.

Each received PID is checked for errors by comparing its MSB and LSB. If error is detected then current packet will be ignored. This error checking helps to prevent USB receiver from getting false (when other packets due to an error become a data packet) data packets.

Data Filter

The filter transforms data from the data stream out of the camera to the unique format used by the image recognition algorithm and stores it to memory.

The data stream, as listed in [10], from the camera consists of image header sequences, chunk size sequences and image data in the bayer RGB format. The image header is a VSYNC patter (80 02 00 00 80 01 00 00) , the chunk size is 02 00 HH LL where HH LL is the number of bytes of image data that follow. The image data is in a two byte per pixel format: green then red for even rows and blue then green for odd rows. The image is set by the attached computer to be 320 by 240 pixels.

The VSYNC and chunk header portions of the data stream are used by the process image_data_out (Fig. 8 in Appendix A) in the filter portion of the VHDL code. This process reads in all image related data from the usb interface and writes out only the color data, a valid flag and indications of a new image starting.

The data format used by the image recognition algorithm is a unique format using one byte for two pixels. The byte is divided into two with the upper four bits representing the first pixel and the lower four the second pixel. Within each half of the byte the most significant bit is the binary representation of the first color followed by the second color. The binary representation of the colors is obtained by thresholding the color byte at 0x80 and setting a 1 for greater than and 0 for less than. The third bit indicates if the pixel is on an odd or even line which affects which colors the preceding two bits represent. Finally the fourth bit is meaningless since two colors were stored in one bit for ease of implementation. This byte format is illustrated in Fig 10.

Color 1 pixel 1	Color 2 pixel 1	Odd / Even	Don't Care	Color 1 pixel 2	Color 2 pixel 2	Odd / Even	Don't Care

Fig. 10: Image recognition byte format

Data transformation is handled by the transform process (Fig. 8 in Appendix A). Using the size of each image, the threshold values for the colors and the outputs of the previous process, the data is transformed from the bayer RGB pattern to the one described above.

The process memory_set (Fig. 8 in Appendix A) waits until the new byte has been completely formed and then sets a flag indicting the byte is ready to be written to memory and sets the value to be written (to_memory) to the byte value.

Finally, the process memory_write (Fig. 8 in Appendix A) actually stores the image in memory. It uses flags set by the image recognition state machine to determine if the image being stored is the target image or a candidate image. Accordingly it stores the image in the appropriate place in memory discussed in the image recognition section. The filter entity is diagrammed in Fig. 11.

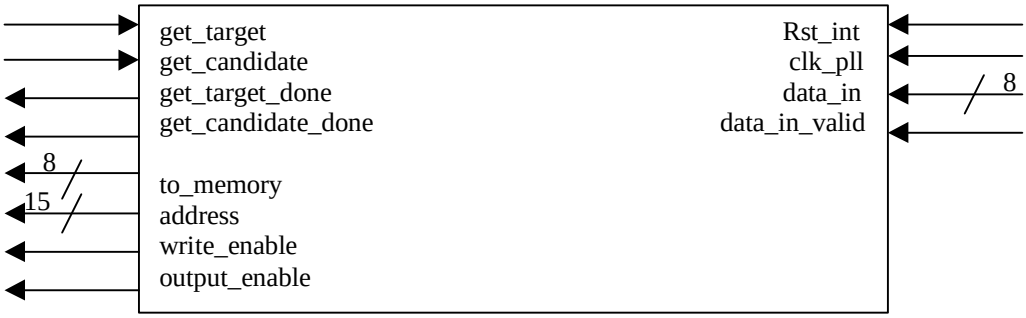


Fig 11. Filter Entity Description

Image Recognition

? **General Description of Algorithm**:

The image recognition portion of the project will take the target image stored in memory and compare it with frames of the same size from the candidate image. The target image is 48 by 48 pixels,

and the candidate image will be 240 by 240 pixels, where each pixel is represented by two bits. The camera selected for the project uses the Bayes RGB method of colour compression so only two colours, either red and green, or blue and green are preserved for each pixel.

The image recognition system works by systematically and exhaustively comparing each possible candidate image with the target image. I.e. the 240 by 240 stored image has a total of $(240 ? 48) * (240 ? 48) ? 36864$ possible 48 by 48 candidate images in it. This is shown graphically in fig. 12.

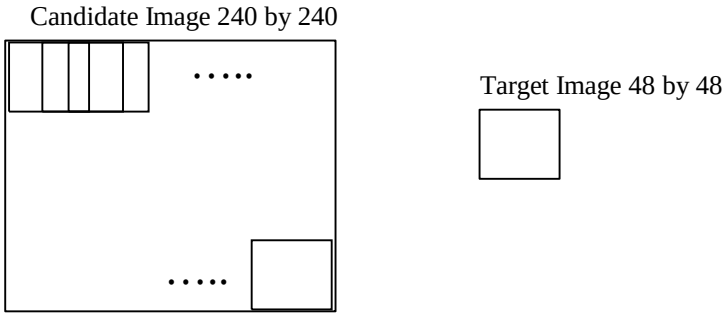


Fig 12. Image recognition.

The possible matches are represented by the dashed squares. The actual comparisons are done on a pixel by pixel level, and we have devised a kind of "fuzzy" matching algorithm. A complete match between two pixels of the same type is a +1, a partial match between two pixels of the same type is a +0, and a mismatch is a –1. These +1's , +0's and –1's are accumulated and the result can be interpreted as a percentage of a complete match. I.e. a complete match will have accumulated $48 * 48 ? 2304$ +1's (one for each pixel in the target image). The general formula then, to calculate the percentage of the match between the target and candidate image is:

$$percent ? counter * \frac{100}{2304}$$

However, since it is inconvenient to represent negative numbers, we shifted the values so that a total match will add 2 to the counter, a partial match will add 1 to the counter, and a mismatch will add 0 to the counter. We also adjusted the denominator so that it is a factor of 2 to aid in computational efficiency. The new formula is:

$$percent ? counter * \frac{100}{4608} ? \frac{?counter * 89?}{4096} ? \frac{?counter * 89?}{2^{12}}$$

This percentage is then represented on the LED outputs on the board and will show the percentage of match between the target image and the best candidate image in the 240 by 240 field of possible candidate images.

The ASIC performs this algorithm using four components. These components are: Image Address, Pixel Recognition, Percent, and LED Output. The hierarchy diagram is shown below in Fig. 13.

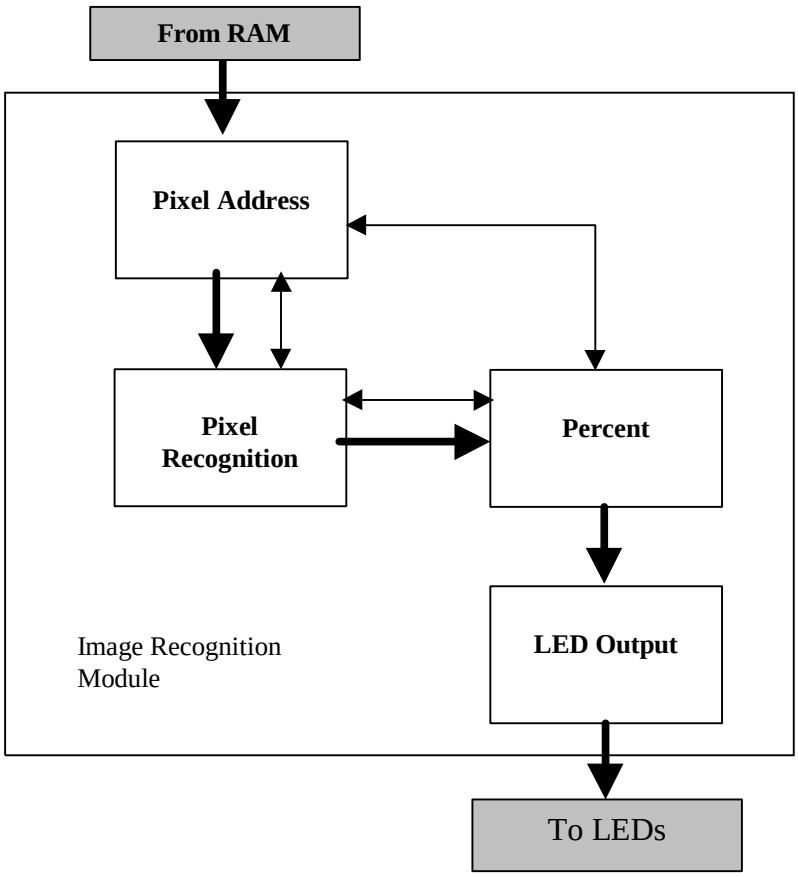


Fig 13. Image recognition module. Top-level diagram.

The data flows through the modules following the heavy arrows. The double ended arrows represent the handshaking between the modules.

? **Pixel Address:**

The Pixel Address module works out the coordinates of the pixels to be compared on the target image and fetches the data from the memory. Then, the Pixel Address module does the same thing for the candidate image pixel. Once both pixels have been fetched from memory, they are held in the Pixel Address's buffer until the Pixel Recognition module requests them. The pixels are sent simultaneously by the Pixel Address module to the Pixel Recognition module. The Pixel Address module is shown below in Fig. 14.

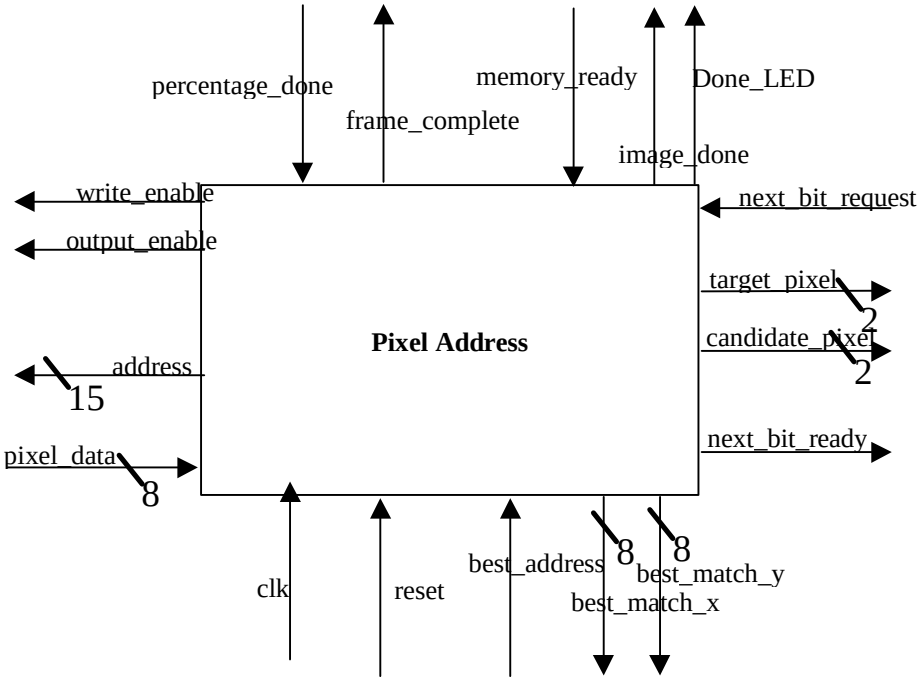


Figure 14. Pixel Address block diagram

? **Pixel Recognition:**

The Pixel Recognition module checks to see if the pixels are a complete match, partial match, or a mismatch. This information is buffered until the Percent module requests it. Once the Percent module requests the match status information, the Pixel Recognition module requests another pixel from the Pixel Address module. The Pixel Recognition module is shown below in Fig. 15.

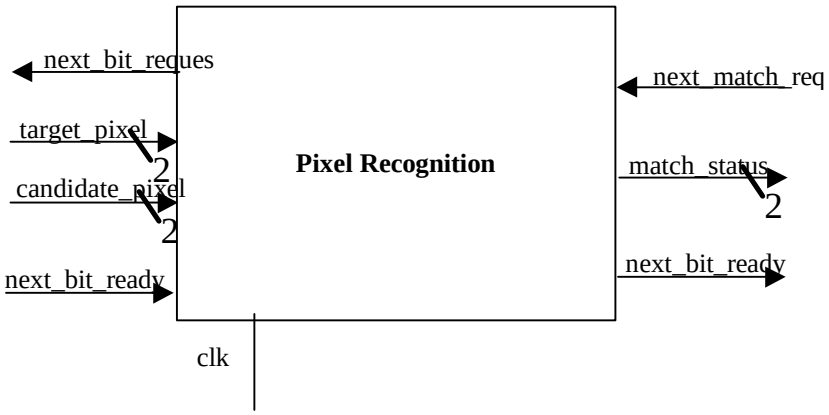


Fig 15. Pixel Recognition module block diagram

? **Percent:**

The Percent module receives the match status information from the Pixel Recognition module and accumulates it with the previous data. The Percent module also checks to see whether a complete frame has been analyzed. If it has, the Percent module tells the Pixel Address module to wait while it calculates the percentage of the match. After this has been completed, the Percent module checks to see if the latest percent is higher than the best percent so far. If it is, the new best percent is updated and sent to the LED Output module. When a new best percent is saved, the save_output signal is sent to the Address module so that the location of the best match can be saved. When this is done, the Percent module tells the Pixel Address module to start going again. If a frame is not complete, then the Percent module simply accumulates the numbers for future calculation. The Percent module is shown below in Fig. 16.

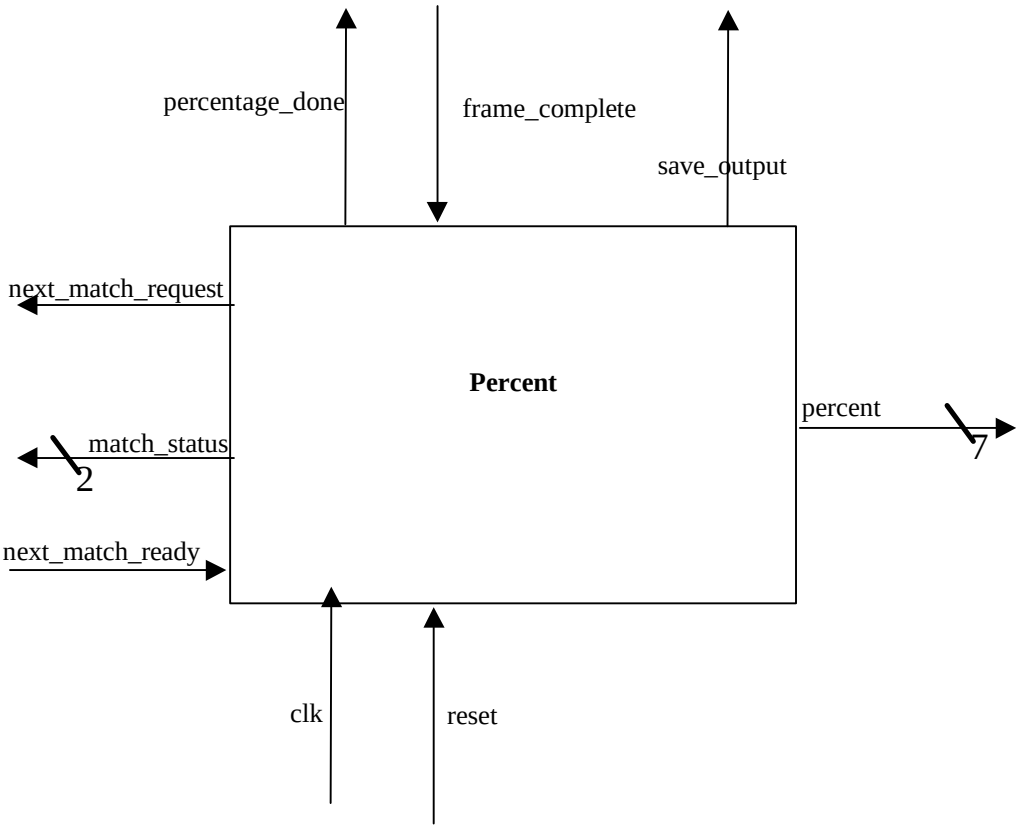


Fig 16. Percent module block diagram

? **LED Output:**

Once the frame is complete and the percentage has been sent to the LED Output module, the LED Output module decodes the 7 bit binary number and sends it to the LED pins on the board. The LED Output module is shown below in Fig 17.

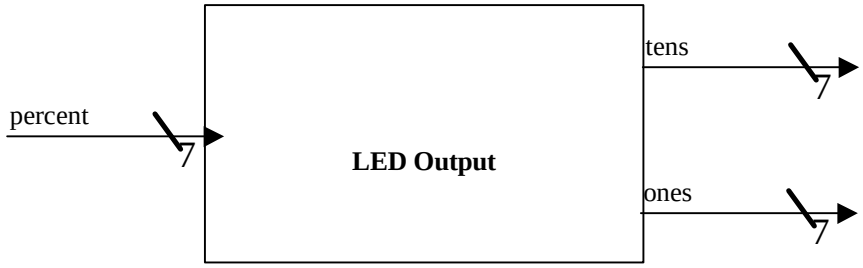


Fig 17. LED Output module block diagram

Memory module

To make sense of the data stored into the memory, we have constructed a memory map showing the location of the target and candidate images in the SRAM. We used a 32k by 8 SRAM module. This RAM chip has 15 address pins (a0 to a14) pointing to 2^{15} ? 32,768 unique memory locations. Each memory location stores 8 bits, or one byte. Since the camera selected for the project uses the Bayes RGB method of colour compression (i.e. only two colours, either red and green, or blue and green are preserved for each pixel), we have chosen to store the pixels in memory with two pixels per byte. The Pixel Type Indicator distinguishes between the two types of pixel. Two bits in each byte are unused.

We organized the bytes of memory into a grid of 128 by 256. Address pins a0 through a6 give the x-coordinate of the map, and address pins a7 to a14 give the y-coordinate. This map is shown below in Fig 18.

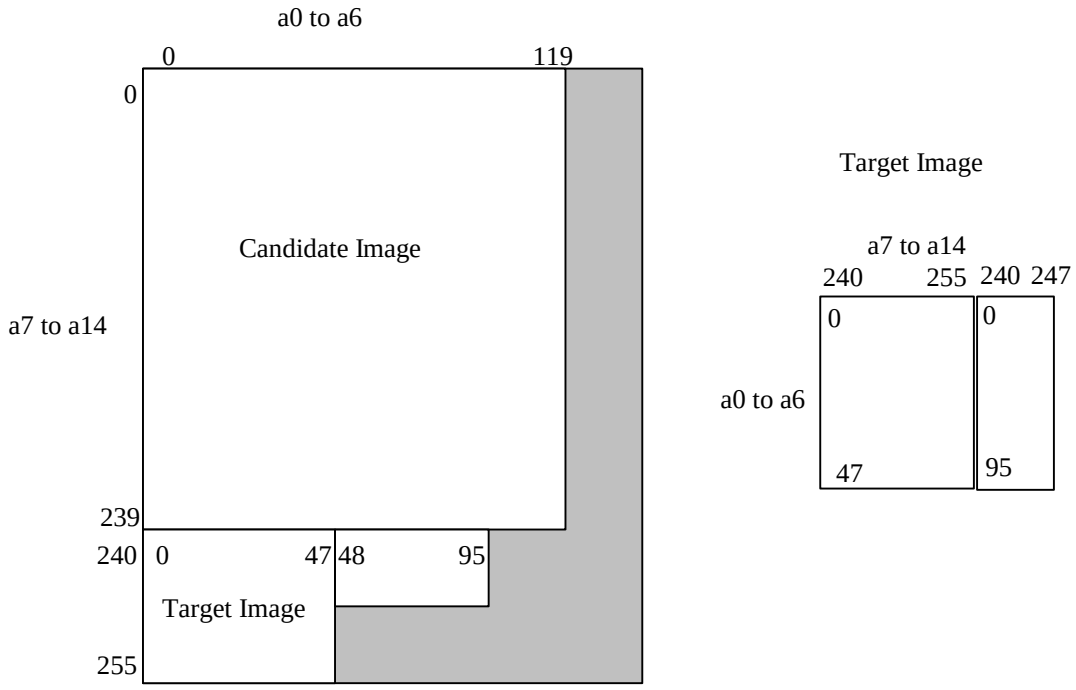


Fig 18. Memory Map.

The candidate image is straightforward. There are 120 horizontal addresses which are addressed by a0 to a6. Since each byte contains two pixels, this makes 240 horizontal pixels for the candidate image. There are also 240 vertical addresses which are addressed by a7 to a14 that make up the 240 vertical pixels.

The target image is more complex. Since there wasn't room on the chip to store the 48 by 48 image in a single easy-to-understand square shape, it had to be divided into two sections. The map for the memory locations, as it would be seen if all of the pixels were aligned properly, is shown on the right in Fig 18.

The equations to transform the (x,y) coordinates on the candidate image are given below.

$$\text{a0 to a6 address} = \frac{x \,?\, coordinate}{2}$$

The remainder is truncated, but preserved. The remainder will be used to determine which pixel in the byte is being addressed.

$$\text{a7 to a14 address} = y \,?\, coordinate$$

The equations to transform the (x,y) coordinates on the target image are give below.

$$\text{a0 to a6 address} = y \,?\, coordinate \text{ if } x \,?\, coordinate \,?\, 32$$

$$\text{a0 to a6 address} = y \,?\, coordinate \,?\, 48 \text{ if } x \,?\, coordinate \,?\, 31$$

$$\text{a7 to a14 address} = 240 \,?\, \frac{x \,?\, coordinate}{2} \text{ if } x \,?\, coordinate \,?\, 32$$

$$\text{a7 to a14 address} = 240 \,?\, \frac{x \,?\, coordinate \,?\, 32}{2} \text{ if } x \,?\, coordinate \,?\, 31$$

The remainder is preserved to determine the location of the pixel within the byte.

VGA interface

? **Introduction**

The VGA interface of the project is a standard 3-bit 8-color display with 640 by 480 resolution. The major part of VGA interface code is taken from past student application notes. The direct reference to the application notes can be found in the reference section in the report.

? Interface capability

The modification that we made is to accommodate the memory data layout in our design and output the target, candidate, and found cursor accordingly. Because of the simplified processing data, the VGA is designed to be output 8-color which is the maximum color depth that will be stored in the

memory. Also, due to the usage of Bayer RGB pattern in optical sensor of Logitech Quickcam Express, the color violent (RGB = “101”) will never show up anywhere in the memory. This will make the violent color as an excellent choice of the found cursor color.

? **Design**

The VGA interface contains three parts: vga.vhd, count_xy.vhd, and syncgen.vhd. The files count_xy.vhd and syncgen.vhd are taken from the past student application notes and they are unmodified. The vga.vhd is the top level control over the count_cy.vhd and syncgen.vhd with built-in output component. The output component is taking the pixel color information stored in the memory and output it accordingly. The actual address map can be referred to the memory data lay-out diagram in the image recognition part. The count_xy.vhd is to count the pixel position and syncgen.vhd is to generate the vertical synchronous and horizontal synchronous signals for 680 by 480 resolution. Both files has retain the same functionality like the original code except the new added reset function.

The vga.vhd will selectively check the value of the current pixel position counter, and it will only output the signal if the pixel position are valid. The vga.vhd will also check the pixel position with the center position of the matched image. It will also check the image match type which can tell us the type of the match (normal size match, zoom in match, zoom out match, and tilt match). By knowing the center position of the match image, vga.vhd will calculate the corresponding crosshair pixel position and override those output value to be violent color.

? **Testing**

The basic VGA output functionality is tested and has no known bug existed. The VGA output can be configured into whatever format we want by specifying the individual pixel location. The grabbing correct pixel data from the memory is not tested to this date due the time constraint and other codes in the project.

Data Sheet

Features:

? Receives data (video stream) from USB cable using wire tapping.
Image data filtering and compression on streaming input data. Due to limitations of space on the FPGA and in memory, 16 bit colour (8 bits per colour) was transformed into 2 bits (one bit per colour)
Images storage using 32k by 8 bit SRAM memory
? All possible candidate frames compared to the target image to find the closest match. The percentage of this closest match is displayed on the FPGA LED bank, and the location of the closest match frame is shown on the VGA output.
? The same target image can be used for multiple candidate images.
? VGA output of the complete 240 by 240 pixel candidate image plus a crosshairs locating the best match frame within the candidate image

I/O pin requirements

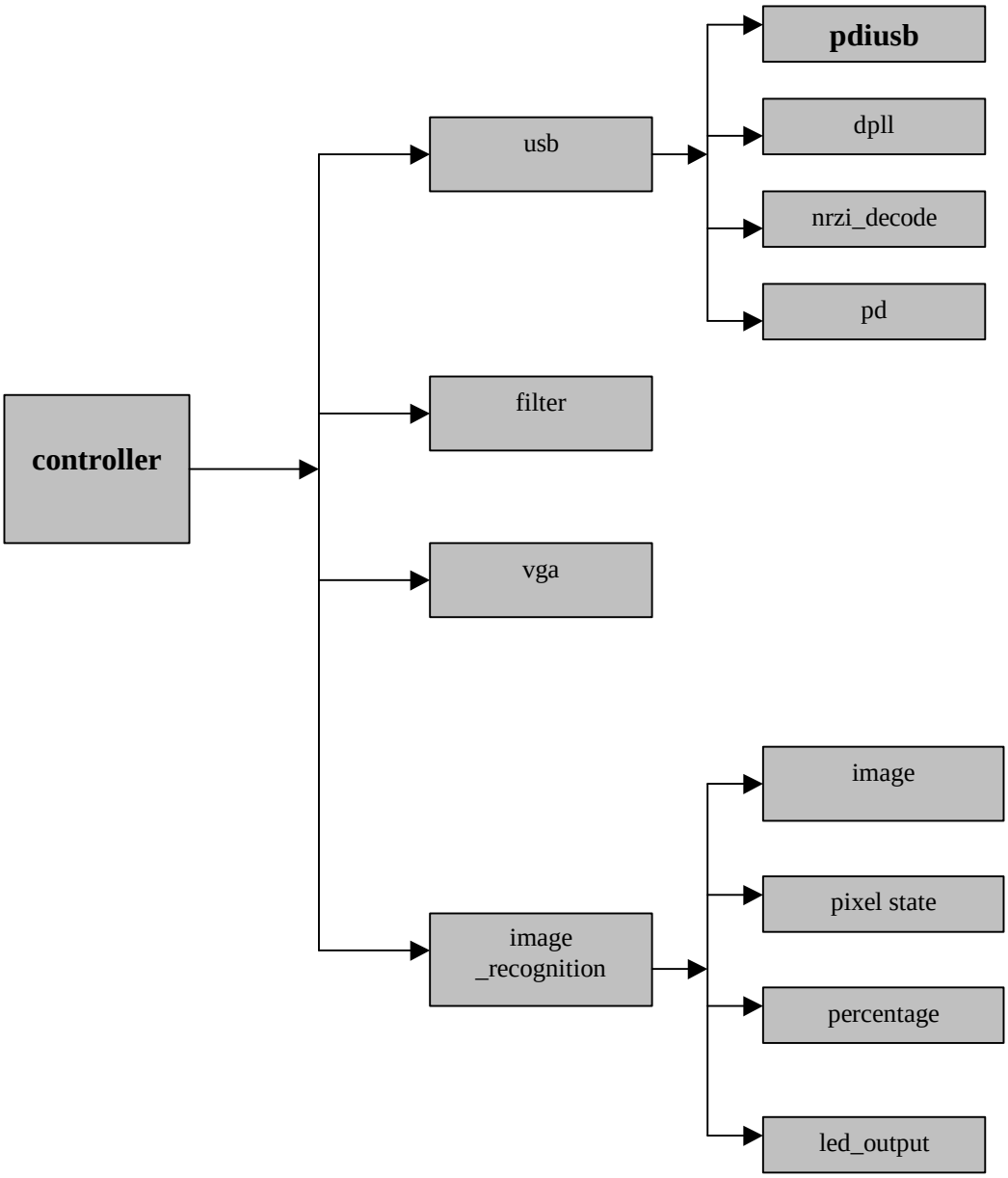
FPGA's I/O pins:

Signal name	Type	# of pins	**Description**
USB receiver			
rcv	**In**	1	Datain input from PDIUSBP11A to FPGA
vm	In	1	USB differential signal (CMOS level) input from PDIUSBP11A to FPGA
vp	In	1	USB differential signal (CMOS level) input from PDIUSBP11A to FPGA
oe	Out	1	PDIUSBP11A operational mode.
address	Out	15	SRAM address bus
to_memory	Out	8	SRAM data bus
write_enable	Out	1	Write anable (WE) signal for SRAM chip
output_enable	Out	1	Output (read) enable signal for SRAM
push_button1	In	1	On board pushbutton (captures candidate image)
push_button2	In	1	On board pushbutton (captures target image)
vga_red	Out	1	Red signal to VGA monitor
vga_green	Out	1	Green signal to VGA monitor
vga_blue	Out	1	Blue signal to VGA monitor
vga_h_sync	Out	1	Horizontal sync signal to VGA monitor
vga_v_sync	Out	1	Vertical sync signal to VGA monitor
tens	Out	7	Ten's part of match percentage. On board LED
ones	Out	7	One's part of match percentage. On board LED

Estimation of required FPGA Logic Cells

Entity		**# of LCs**
USB	*pdiusb*	101
	dpll	20
	nrzi_decode	42
	pd	32
Total *USB*:		**195**
Data filter		**484**
Image_recognition	*Pixel_state*	11
	Percentage	152
	Ledoutput	64
	image	317
Total *Image_recognition*		**544**
VGA		**366**
Total:		**1589**

Design Hierarchy



Index to VHDL code (see Appendix A)

USB receiver

usb*:* top-level entity of USB receiver (encapsulates all other parts) . Extracts video data from USB cable.

- implemented in hardware and tested, no known bugs;

pdiusb: physical interface. Detects EOP and forced reset conditions.

- implemented, no known bugs;

nrzi_decode: NRZI decoding/ bit-unstaffing.

- implemented, no known bugs;

dpll: Digital phase locked loop. Extracts clock from incoming data.

- implemented, no known bugs;

pd: Packet disassembler. Analyze packets and filters out data.

- implemented, no known bugs;

Data filter

filter: Transforms 16-bit color format to 3-bit (8 colors).

- simulated and partially implemented, no known bugs;

Image recognition

image_recognition: top-level entity of image recognition unit

- simulated, no known bugs;

image: works out coordinates of pixels to be compared

- simulated, no known bugs;

pixel_state: compares images

- simulated, no known bugs;

percentage: calculates percent of matches

- simulated, no known bugs;

ledout: outputs number of matches

- simulated, no known bugs;

VGA controller

vga : top-level entity of VGA controller;

- compiled and tested in hardware, no known bugs;

syncgen: H and V sync signal generator;

- compiled and tested in hardware, no known bugs;

count_xy: pixel position counter;

- compiled and tested in hardware, no known bugs;

Test bench index.

USB receiver test bench

usb_test: Top-level entity of USB receiver test bench. Used to simulate behavior of USB receiver. It includes test bench package and UUT.

usb_test_pack: Test bench package file. Comprises all components (procedures), required to generate appropriate USB signals.

Index of test cases and simulations

For detailed description of test cases and simulation waveforms see Appendix B.

Functional module **(entity)**	*Waveform figure #* **(Appendix B)**	Test cases
pdiusb	Fig1	Verified if physical interface can detect EOP (end of packet)
	Fig2	Verified forced (by host) reset detection.
dpll	Fig3	Tests correct functionality of Digital Phase Locked Loop (clock recovery)
nrzi_decode	Fig4	Verified "sync" pattern detection
	Fig5	Verified correctness of data recovering from NRZI format
	Fig6	Verified correct functionality of bit-unstuffing mechanism.
	Fig7	Verification of bit stuffing error detection
pd	Fig8	Checks whether PD can detect "data" packets and ignore "non-data" packets
	Fig9	Verified whether PD can detect corrupted PID and then ignore the whole packet
usb	Fig10	Top-level USB entity simulation, using test bench. Verified full functionality of USB receiver.
filter	Fig 11	Verified ability to write multiple lines of pixels and verified writes to memory and appropriate memory mapping.
	Fig 12	Verified ability to write longer lines of pixels
	Fig 13	Verified ability to detect new VSYNC patterns in data

image	Fig 14 Fig 15	Verify correct addresses assigned during beginning of frame Verify correct addresses assigned during end of frame / beginning of new frame
pixel state	Fig 16	Verify that the correct match status signals are being sent
percentage	Fig 17	Verify that the percentage counter is working correctly
LED_output	Fig 18	Verify that the LED_output is sending the correct signals to the ones and tens LEDs
image_recognition	Fig 19 Fig 20	Verify that the entire algorithm works at the beginning of a frame Verify that the entire algorithm works at the end of a frame

References

1. USB 1.1 Specification documents: http://www.usb.org/developers/docs.html
2. Logitech website: http://developer.logitech.com/
3. Full-Speed USB 1.1 Function Controller: http://www.trenz-electronic.de
4. Source code for Logitech QuickCam Express driver on Linux OS system: http://qce-ga.sourceforge.net/
5. EE552 student application notes: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/2001_w/misc/Apps_Links.html
6. Image processing concept: http://www.sciam.com/0697issue/0697villabox4.html
7. Design a robust USB serial interface engine (SIE): http://www.usb.org/developers/whitpapr.html
8. Cyclic redundancy check in USB: http://www.usb.org/developers/whitpapr.html
9. Logitech Quickcam data format: http://wwwbode.cs.tum.edu/~acher/quickcam/quickcam.html
10. Camera data stream format from Georg Acher's Homepage http://wwwbode.cs.tum.edu/~acher/quickcam/quickcam.html