ECE532

Multipoint Hardware Point Detection

Group Report

March 31, 2008
Prof. P. Chow

Phil Lam	994060449
Daniel Ly	994068682
Yi Yao	994085583

Table of Contents

1 Overview 3
2 Outcome 5
3 Block Descriptions 6
3.1 Processor Local Bus (plb_v34_v1_02_a) 6
3.2 On-Chip Peripheral Bus (opb_v20_v1_02_a) 6
3.3 PLB to OPB Bridge (plb2obp_v1_01_a) 6
3.4 PowerPC (ppc405_v2_00_c) 7
3.5 OPB Uartlite (opb_uartlite_v1_00_b) 7
3.6 OPB GPIO (opb_gpio_v3_01_a) 7
3.7 PLB DDR Memory Controller (plb_ddr_v1_11_a) 8
3.8 Block RAMs (bram_block_v1_00_a) 8
3.9 BRAM IF Controller (plb_bram_if_cntlr_v1_00_b) 9
3.10 VGA Controller (plb_tft_cntlr_ref_1_00_d) 9
3.11 VGA Capture Core (vidcap_v1_00_a) 10
3.12 PLB IF (bplb_acces_v1_01_a) 10
3.13 PLB Master (plb_read_test_v1_00_a) 11
3.14 PLB Master (plb_m_v1_00_a) 11
3.15 Point Detection (vidcap_v1_00_a) 11
3.15.1 BRAM Image Buffer (bram_buffer.v) 12
3.15.2 Laplacian Calculation (del_squared3.v) 13
3.15.3 Local Minimum Determination (islocal_min2.v, BRAM_shift.v) 15
3.15.4 BRAM-based Detection Memory (list_mem.v) 17
3.15.5 Agglomerative Clustering (list_cluster_FSM.v) 17
3.15.6 Output Interface (list_cluster_FSM.v) 17
3.16 Software Pong Game (main.c, main.h) 18
3.16.1 2D_types (2d_types.c, 2d_types.h) 18
3.16.2 ball (ball.c, ball.h) 18
3.16.3 collidable (collidable.c, collidable.h) 18
3.16.4 draw (draw.c, draw.h) 19
3.16.5 vga (vga.c, vga.h) 19
4 Design Tree 20
5 References 22

1 Overview

Blob detection is a practical subfield of computer vision which focuses on detecting points or regions of a different intensity than the surrounding image. It is a mature field of interest with the majority of research and breakthroughs occurring in the middle to late 1990s[1]. Although the initial techniques have since been improved by various contributors, the essential algorithm remains the same.

Traditionally, this technology was applied in fields such as robotic manufacturing, but is now increasingly used in consumer products. In particular, many products have explored a wide variety of novel human-computer interfaces, and blob detection is often the driving technology behind it. For example, Apple's new line of portable MP3 players (iPod Touch) and cellular phones (iPhone) emphasize their multi-touch interfaces which detects a variety of fingers on an interactive screen [2] [3]. There is also academic interest in this field [4].

Because blob detection is computationally intensive but requires relatively simple arithmetic operations, it is an ideal candidate for parallelization in hardware. The main goal of this project was to develop a hardware implementation for blob detection that is able to detect an arbitrary number of blobs in a video image, and present the data to a standard, sequential processor in a simple manner. Implementations could potentially find use in consumer applications; possible applications include integration as an on-chip peripheral for processors commonly used in cell phones and other consumer electronics for which multi-touch interfaces are being developed. Others include interfaces such as the Nintendo Wii-mote, which operates by using an infrared-sensitive charge-coupled devices (CCD) to track a set of physically fixed infrared light emitting devices (LEDs).

To showcase the capabilities of the hardware blob detection core, an interactive game demo was designed. The game was an interactive version of the classic video game, Pong. Using a video camera set to record infrared light, an arbitrary number of players would be given paddles with infrared LEDs to play in front of a field generated in front of a VGA projector. Using blob detection, the hardware core would then resolve all the locations of the paddles and transmit this to a processor that in responsible for running the game and controlling the video output.

The project was implemented on a Xilinx XUP Virtex-II Pro Development System [5] and used the VDEC1 video decoder board. In addition, a standard NTSC video camcorder was used in "nightshot" mode to remove the packaged infrared filter. In addition, a visible light filter, composed of floppy disks, was used to filter visible light. Custom game paddles were produced with infrared LEDs purchased at the local electronics store.

The following is a block diagram of the project design:

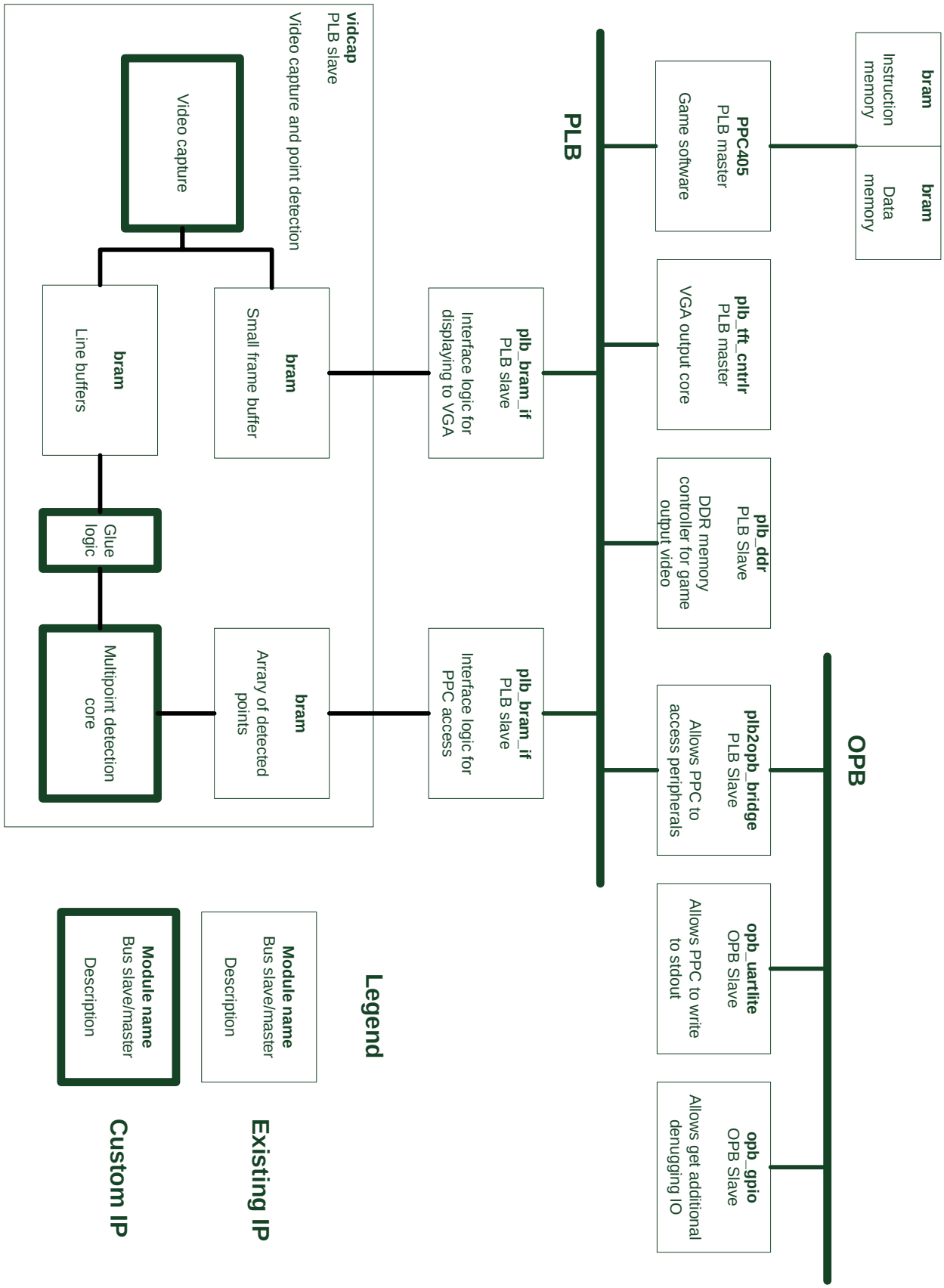

Figure 1 Block diagram of the project

2 Outcome

The final result of the project fell short of the goal set in the proposal. The initial goal of the project was to create a game in which user interaction is primarily based on point tracking of infrared emitters. However, at the time at which the document was submitted, only components of the project were complete and full system integration has not yet occurred. The first completed component consisted of the video capture core modified to store seven lines of video into a buffer for the hardware detection core. The hardware blob detection was the second component and the third component was the processor simulating the game and outputting this to VGA.

Although the project goals were not fulfilled, a significant amount of work was completed and a wide variety of cores were generated. The primary difficulty was to find a common interface between the video capture and the hardware processor. The original design required entire frames to be buffered and thus used the off-board DDR memory and its PLB controller. As a result, we invested enormous effort and time into developing a PLB master interface which could be used in our project.

We generated a wide variety of PLB masters with varying success. We attempted solutions using the PLB_IF component as well as our own custom hardware based on the Xilinx and IBM datasheets. Our design revolved around our full system simulation of a PLB bus and BRAM interface. One of the primary issues which prevented us from creating functional IP was the discrepancy between behavioural simulation and post place-and-route functionality of the written HDL. Behavioural simulation performed on our PLB masters failed to point out the deficiencies in our design.

Two custom cores were designed for the purpose of this project, the video capture core and multipoint detection core. Most cores were simulated extensively in a test environment similar to that used for synthesis. The custom cores exhibit properties of their intended functionality but due to the lack of extensive testing, the functionality was not fully verified.

3 Block Descriptions

3.1 Processor Local Bus (plb_v34_v1_02_a)

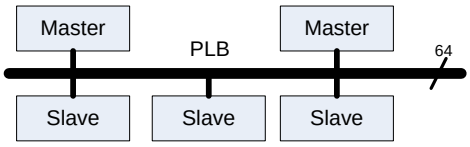

Figure 2 Block Diagram of PLB

The PLB (processor local bus) is the PPC's native bus. It is 64 bit wide and runs at a core frequency of 100MHz in our designs. This bus supports multiple masters, multiple slaves and the arbiter provides support for masters of different priorities. All peripherals on this bus are memory mapped. This bus supports burst transactions and timeouts.

In our design, this bus was favoured other the OPB as the primary data bus due to its higher band width. However, creating a bus master and bus slave proved to be a challenge.

3.2 On-Chip Peripheral Bus (opb_v20_v1_02_a)

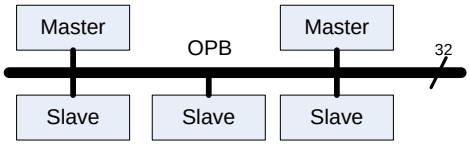

Figure 3 Block Diagram of OPB

The OPB (On-Chip Peripheral Bus) is the MicroBlaze's native bus. It is 32 bits wide in our designs and runs at various frequencies. This bus also supports multiple masters, multiple slaves. Slaves on the OPB are also memory mapped.

In our project, most low bandwidth peripherals such as GPIOs and UART were attached to the OPB.

3.3 PLB to OPB Bridge (plb2obp_v1_01_a)

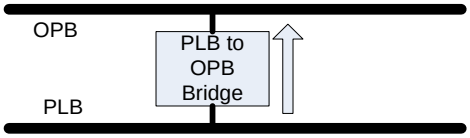

Figure 4 Block Diagram of PLB to OPB Bridge

This bridge allows the PPC to access peripherals attached to the OPB. The PLB to OPB Bridge acts as a slave on the PLB and master on the OPB. There is a FIFO in this bridge and attempting to make too many bus transactions through this bridge can cause the FIFO to overflow and cause system instability.

3.4 PowerPC (ppc405_v2_00_c)

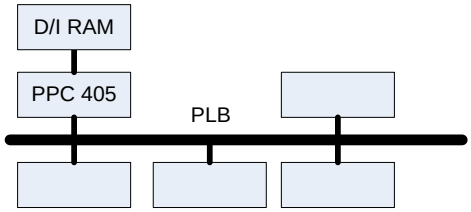

Figure 5 Block Diagram of PPC

The PowerPC (also known as the 405) is a hard processor within the FPGA fabric of Virtex Pro FPGAs. Due to their hard nature, they can run at much higher frequencies than soft processors such as MicroBlazes. The PPC is a PLB master.

The PPC lacks floating point support, but software emulation proved to be sufficient in our project.

3.5 OPB Uartlite (opb_uartlite_v1_00_b)

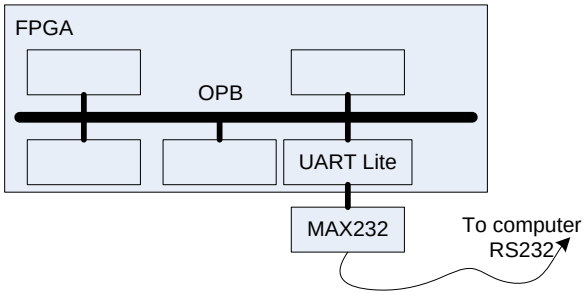

Figure 6 Block Diagram of OPB Uartlite

This peripheral provided our hardware design with communication with a computer at a baud rate fixed during before synthesis. All stdio function calls are routed through this device. The UART has an output buffer of variable length and can perform input and output asynchronously.

3.6 OPB GPIO (opb_gpio_v3_01_a)

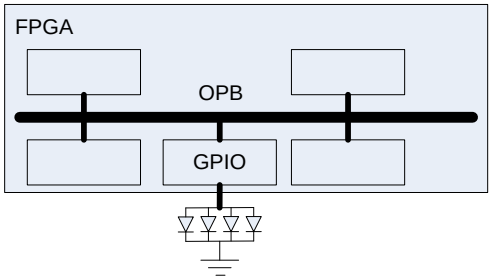

Figure 7 Block Diagram of OPB GPIO

General purpose IO (GPIO) allows for parallel access to external ports. These parallel ports allow for output such as driving LEDs and input such switches and pushbuttons.

3.7 PLB DDR Memory Controller (plb_ddr_v1_11_a)

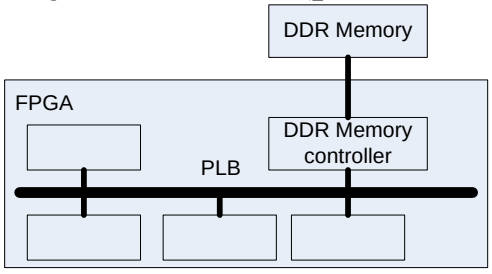

Figure 8 Block Diagram of PLB DDR

This core is a PLB master which allows for memory mapped access to off chip DDR memory. Different DDR memory controllers exist for memories of different sizes and they are not exchangeable. For our project, we used a single rank 256MB DDR memory controller. The primary use of the memory controller in our design is to provide screen buffers for the VGA output core. Secondary use of this memory is to provide a heap for functions requiring malloc.

It has been noticed that the OPB DDR memory controller hangs the system bus after a few consecutive bus transfers. Due to the instability of the OPB version of this core, the architecture of our initial design was modified. Initially, we choose to use a MicroBlaze as the primary processor for its floating point and FSL support. However, due to the instability of the OPB DDR memory controller, the design was moved to a PPC with software emulation for floating point instructions.

3.8 Block RAMs (bram_block_v1_00_a)

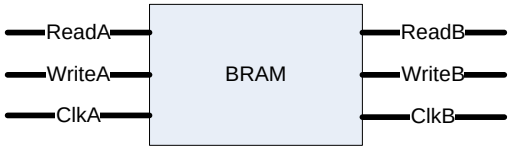

Figure 9 Block Diagram of BRAM

Block RAMs (BRAM) are comprised of static memory which exist within the FPGA fabric. Multiple memory primitives are stitched together to provide larger memories which can be used to hold instruction or data. Each BRAM has 2 ports which can be used simultaneously for reading and writing. This functionality makes BRAMs useful for passing large volumes of data across clock domains and thus, are utilized in FIFOs.

3.9 BRAM IF Controller (plb_bram_if_cntlr_v1_00_b)

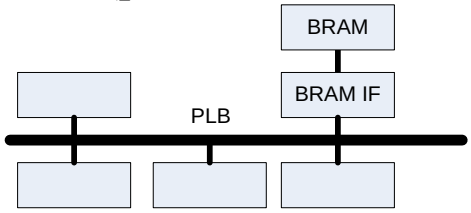

Figure 10 Block Ram Controller

BRAM have a very simple interface for read and write operations. This interface is insufficient to interface a complex bus such as the PLB and OPB. The purpose of the BRAM interface controller is to provide wrapper logic to map BRAM contents to a bus.

This functionality is exploited in our final design where one port of a dual port BRAM is used for a custom IP and the other port is used for accessing the bus. This feature allowed for debugging and easy IP interfacing.

3.10 VGA Controller (plb_tft_cntlr_ref_1_00_d)

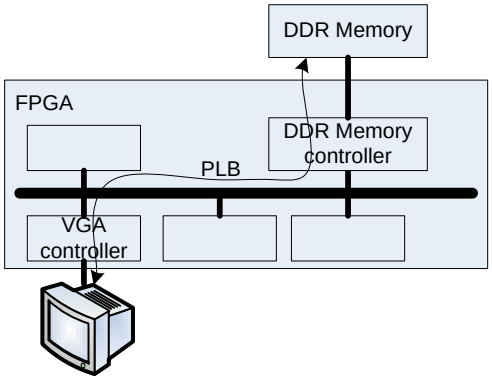

Figure 11 Block Diagram of VGA Controller

The video capture core was taken from the reference slideshow design available from the Digilent website. This core acts as a PLB master and displays a frame of data on through a VGA interface. This core reads data from the PLB using burst reads. The output resolution is 640 by 480. Each frame occupies 2MB of memory and must be aligned to a 2MB boundary. The location of the frame and the monitor's state can be modified through a register on the DCR bus. Each pixel takes up 4 bytes (unused byte, red, green and blue bytes). Each line is 1024 pixels long even though only the first 640 pixels are displayed. Each frame is 512 lines long even though only the first 480 lines are displayed. Due to this organization, each frame takes up 2MB of memory.

3.11 VGA Capture Core (vidcap_v1_00_a)

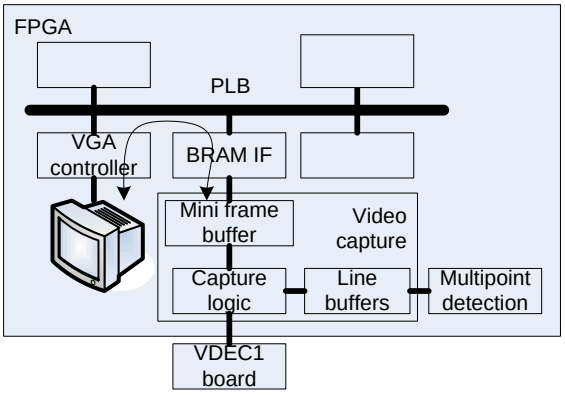

Figure 12 Block Diagram of VGA Capture Core

This core is adapted from the video_capture project available from Digilent's website. The purpose of this core was twofold:

1. Capture selective lines of video for display on a VGA terminal. This is done primarily for debugging purposes
2. Buffer lines of video for further processing by the multipoint detection core.

The mini frame buff currently consists of only 8 lines due to the resource restriction on BRAMs in the Virtex 2P fabric. By placing the capture buffer on the PLB, it also potentially allows software to access the captured video for other types of processing. Currently, due to the small size of the mini frame buffer, the VGA controller gets bus timouts when trying to access nonexistent lines form the mini frame buffer. As a result, the VGA controller is unable to make timings to perform horizontal and vertical sync and hence the output display is unstable in the current configuration. However, it has been shown that the output does match the input by demonstrating colour correctness and horizontal resolution.

3.12 PLB IF (bplb_acces_v1_01_a)

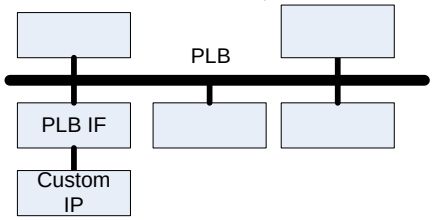

Figure 13 Block Diagram of PLB IF

In our initial designs, a large effort was placed in creating a PLB master. This was motivated by the need to perform high bandwidth memory access required for video capture and processing. Many attempts were made to achieve this, but due to the lack of proper documentation, this proved difficult and unfruitful. Due to the difficulty in design, this approach was abandoned and an alternative design which did not require PLB access was devised and adopted in the end.

Even when behavioural simulation showed proper bus transactions, the synthesized design rarely does. Even when bus reads and writes were performed in hardware, common issues that we ran into were:

- Endianess mismatching
- Incorrect address (ie memory fetched or written to the wrong address)
- Memory access restricted to address 0x00000000

3.13 PLB Master (plb_read_test_v1_00_a)

An attempt was made to interface to the PLB directly by constructing a Master bus interface. Custom logic was built to mimic PLB transactions as described in the PLB documentation. The slideshow reference design [6] provided by Digilent in conjunction with the XUP-V2P board was used as a starting point.

3.14 PLB Master (plb_m_v1_00_a)

A second attempt was made to interface to the PLB directly by constructing a Master bus interface. Rather than using the reference design, this core was based from the IBM PLB specification sheet and followed its conventions for timing and signal analysis.

Although behavioural simulation of this core with a BRAM memory showed proper bus transactions, the synthesized design did not have similar success with the DDR memory. Despite only finding this datasheet one day before the demonstration, successful writes to memory were achieved. Memory reads, however, proved more difficult and due to time constraints, was not further investigated.

3.15 Point Detection (vidcap_v1_00_a)

The point detection core is the heart of the project, and was developed in several parts. The design interfaces with dual-port block RAMs at the input and output; while the design is synchronous between operational blocks, it can be operated at an arbitrary frequency relative to the rest of the design, provided that it runs fast enough to process the incoming data. The operation of the block is sequential, data passed from each operational block to the next. The diagram below shows the flow of data through the core:

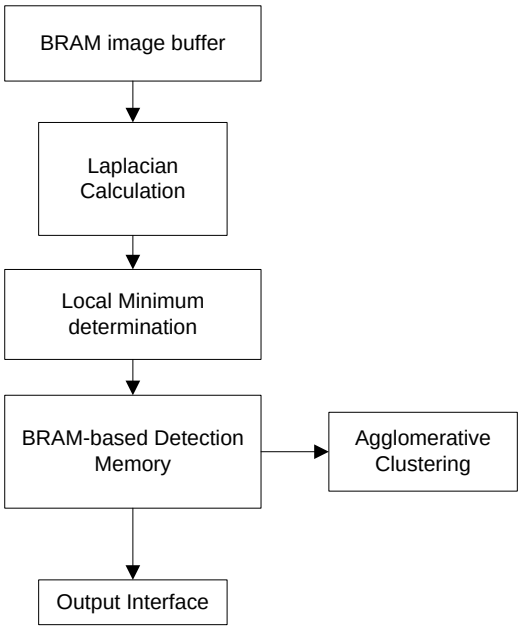

Figure 14 Black Diagram of Point Detection Core Flow

3.15.1 BRAM Image Buffer (bram_buffer.v)

Overview

The image buffer contains a matrix of 42 BRAMs. Each BRAM is organized as a 32b x 512 memory, for a total of 2KB of storage per BRAM, or 84KB total. Each BRAM is a true dual-port memory, and the memory is constructed in such a fashion that it can be written to by a video source while simultaneously being read by the detection core

.

Memory Organization

The incoming image data is greyscale, represented by 8-bit integers. This means that each BRAM address can represent four pixels. Since the Local Minimum Determination module requires a matrix of at least 3x3 Laplacian values to generate a determination for a single pixel, the Laplacian Calculation module requires a grid of 7x7 raw pixels to provide this information. Thus, the BRAM must be able to present a block of at least 49 pixels at its output for the Laplacian calculation to proceed without additional registering of data.

Since some of the Laplacian calculations are reused when determining local minima between adjacent pixels, it is efficient to make the block size larger than 7x7, to analyze more than one pixel at a time. This can be done up to the limit of BRAMs or logic in the FPGA. A block size of 12 x 10 pixels was chosen, as a trade-off between efficiency of calculation and the amount of logic required on the FPGA. This structure requires a matrix of at least 3 x 10 BRAMs, since each BRAM can represent 4 pixels: (3x4) x 10. The pixel matrix is outputted on a wide (960-bit) output bus, so that the calculation can be done combinatorially from this output, without additional registers.

To support true simultaneous buffering and processing, the actual BRAM array consists of 3 x 14 BRAMs, for a total array size of 12x14 pixels. This allows for guaranteed availability

of four lines, while the remaining ten are being processed. Logic around the memory is used to arbitrate which lines are being written to and read from.

Memory Access

The input port is designed to be as opaque as possible, requiring only a simple clock-and-data interface. The video source must simply provide data line by line, and a clock signal upon which to read the data in. Internal logic routes the data to the appropriate BRAMs and addresses, using counters and the like. Line and frame sync pulses are also required from the interface, to ensure that data does not go out of sync.

The memory organization is shown in the diagram below:

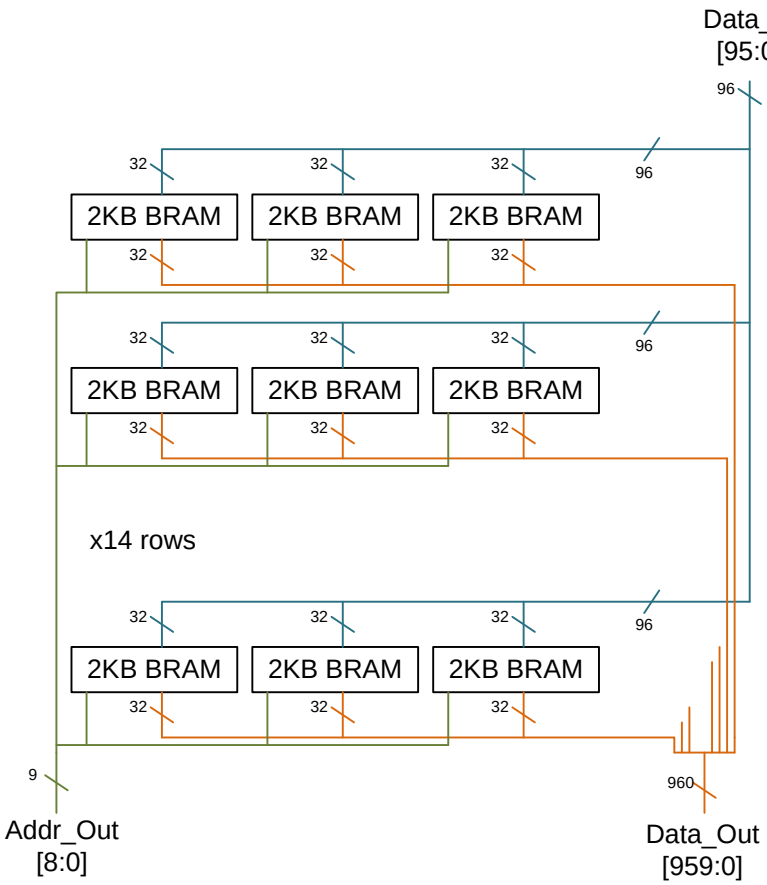

Figure 15 Memory Organization

3.15.2 Laplacian Calculation (del_squared3.v)

Overview

The application of differentiation to discrete data is most easily done by using the traditional, continuous definition of the derivative, but forgoing taking the limit as the differential distance Δx goes to zero. Additionally, by using the width between pixels as a natural unit of distance, the division by Δx is eliminated, and the only remaining operation is a subtraction. In this simplified case, the calculation of the Laplacian of a given image, which is essence the sum of the partial differentials in the two image axes becomes a matter of a large number of subtractions.

Mathematics

A note on conventions:

When developing this core, several conventions were followed when dealing with matrix arithmetic and array representation.

- The m and n axes were used, to represent the column (horizontal) and row (vertical) axes, respectively.
- All array and image indexing was done using the standard matrix notation, whereby element indexes are increasing downwards and to the right.
- Indexing for any arrays or matrices starts at zero, and continued to N-1 in the vertical direction and M-1 in the horizontal, given that the array size was (N,M). This means that the top left element has index (0,0)
- All ordered pair notation was done in "Row-Column" form, or (m,n.) An easy way to remember this is to note that in the ordered pair, m and n appear in reverse alphabetical order.
- The following naming conventions were used:
    - The original image matrix is denoted $M(n,m)$
    - A partial derivative is denoted by a subscript. For example, a partial derivative in the vertical direction of the original source image is $M_n(n,m)$
    - A second partial derivative is denoted by the addition of a second subscript, indicating the dimension of the second derivative. Example: $M_{nn}(n,m)$ is equivalent to $\frac{\partial^2 M(n,m)}{\partial n^2}$.
    - The Laplacian Matrix is denoted by $L(n,m)$

Laplacian

The Laplacian of image M is defined as

$$\nabla^2 M(n,m) = \frac{\partial^2 M(n,m)}{\partial n^2} + \frac{\partial^2 M(n,m)}{\partial m^2}$$

This can be equivalently represented as

$$\nabla^2 M(n,m) = \frac{\partial}{\partial \mathrm{n}}\left(\frac{\partial M(n,m)}{\partial \mathrm{n}}\right) + \frac{\partial}{\partial \mathrm{m}}\left(\frac{\partial M(n,m)}{\partial \mathrm{m}}\right)$$
$$\nabla^2 M(n,m) = \frac{\partial}{\partial \mathrm{n}}\left(\mathrm{M_n}(n,m)\right) + \frac{\partial}{\partial \mathrm{m}}\left(\mathrm{M_m}(n,m)\right)$$
$$\nabla^2 M(n,m) = \mathrm{M_{nn}}(n,m) + \mathrm{M_{mm}}(n,m)$$

To do this in a discrete fashion, we must first define $\mathrm{M_n}(n,m)$ and $\mathrm{M_m}(n,m)$ using the standard definition of a limit. Δx in this case is 2 pixels, to make the derivative symmetric about the pixel (m,n). Division by Δx (in this case, 2) is ignored.

$$\mathrm{M_n}(n,m) = M(n+1,m) - M(n-1,m)$$
$$\mathrm{M_m}(n,m) = M(n,m+1) - M(n,m-1)$$

Now, we re-use this definition to obtain the second partial derivatives.

$$\mathrm{M_{nn}}(n,m) = \mathrm{M_n}(n+1,m) - \mathrm{M_n}(n-1,m)$$
$$\mathrm{M_{mm}}(n,m) = \mathrm{M_m}(n,m+1) - \mathrm{M_m}(n,m-1)$$

Expanding this provides the following definition for the Laplacian of a given matrix:

$$\nabla^2 M(n,m) = [M(n,m+2) - M(n,m)] - [M(n,m) - M(n,m-2)] + [M(n,m) - M(n-2,m)] - [M(n,m) - M(n-2,m)]$$

Note that a 5x5 matrix of pixels is required to compute the Laplacian of the centre pixel. The lighter area in the figure below illustrates the area required to calculate the Laplacian for pixel (m,n). The darker area illustrates the area required to calculate the single partial derivatives in the m and n directions for pixel (m-1,n-1).

M(n,m)	m-3	m-2	m-1	m	m+1	m+2	m+3
n-3							
n-2							
n-1							
n							
n+1							
n+2							
n+3							

Figure 16 A 7x7 block of the original image matrix

Implementation

The Laplacian Calculation block is implanted purely combinatorially. The expression shown above indicates that a series of arithmetic operations are all that is required to generate a matrix representing the Laplacian of the source image. Because of the large amount of data involved, it is inefficient to register the output data of the BRAMs; instead, it is possible to compute the result directly from the outputs. Thus, the Laplacian block is designed to be purely combinatorial, and consists almost exclusively of adder trees designed to produce the Laplacian in the least possible time.

3.15.3 Local Minimum Determination (islocal_min2.v, BRAM_shift.v)

Overview

Mathematically expressed, local minima in the Laplacian of the source image indicate blobs of a certain scale in the image. This is implemented practically by using a simple logical test to find minima. Additionally, this block constructs coordinate data for any local minima it finds, storing these in the Detection Memory. See the source files for a detailed explanation of the coordinate storage format.

Algorithm

To determine local minima for a given pixel (m,n), a comparison test is made for all 8 adjacent pixels. If the pixel (m,n) is less then all adjacent pixels, and is below a certain minimum threshold, the point is labeled as a local minimum. The figure below illustrates this:

Laplacian Matrix L(m,n)	m-1	m	m+1
n-1			
n			
n+1			

Figure 17 A 3x3 grid of the Laplacian matrix

M(m,n) is a local minimum if:

$L(n-1, m-1) \geq L(n,m)$ and
$L(n-1, m) \geq L(n,m)$ and
$L(n-1, m+1) \geq L(n,m)$ and
$L(n, m-1) \geq L(n,m)$ and
$L(n, m+1) \geq L(n,m)$ and
$L(n+1, m-1) \geq L(n,m)$ and
$L(n+1, m) \geq L(n,m)$ and
$L(n+1, m+1) \geq L(n,m)$ and
$L(n,m) \leq LOW\_DET$

where LOW_DET is a constant.

Implementation

Local Minimum determination was straightforward comparison. See the source file islocal_min2.v after reading the Algorithm section for this block. This sub-block generates a signal of 24 islocal_min[n] signals, which corresponds to a 6x4 block of pixels centered at the original block of pixels that was analyzed (recall, a 7x7 grid is required to calculate the local minimum signals, and so the outer pixels of the original image block are only used to in intermediate calculations.)

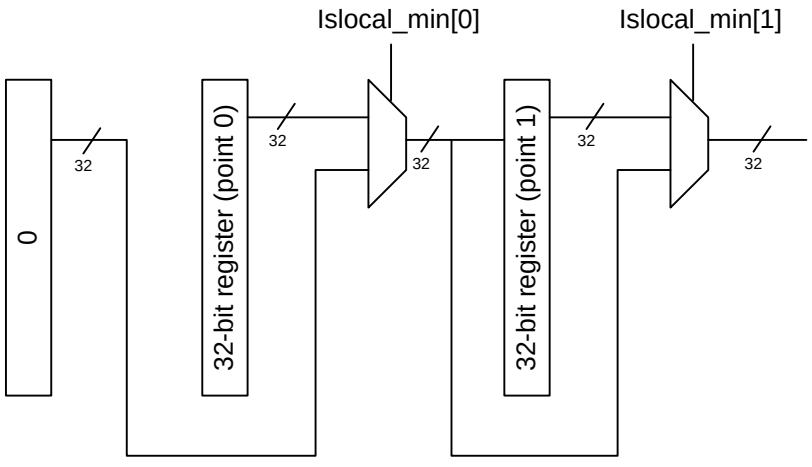

Figure 18 Local Minimum Shift Registers

The mechanism to create local minimum coordinate lists is more complex. For each block of the original image processed, this module sets up a shift register of variable length containing the coordinates of every local minimum found. The SR varies from length zero to 24, and shifts 32-bit data directly into the Detection Memory block. This is accomplished by using multiplexers to choose individual registers in the SR to bypass. See the diagram below:

The data in each register is initialized to be the coordinate for whichever pixel the islocal_min[n] signal corresponds to. This is determined using the coordinate of the particular block in conjunction with the pixel number.

3.15.4 BRAM-based Detection Memory (list_mem.v)

This is a standard BRAM to hold a list of up to 512 points described as local minima. The BRAM actually contains mirrored copies of the contents, which are written simultaneously from one port, but can be modified individually from the other. This allows the Local Minimum Determination block to write two copies to the memory simultaneously, and allows the Agglomerative Clustering block to manipulate each copy separately. It is worth noting, though, that although the memory is mirrored; only data written to the source port is synchronized. Changes made using the list1 and list2 ports need to be synchronized by an external source.

3.15.5 Agglomerative Clustering (list_cluster_FSM.v)

This block performs agglomerative clustering on the points collected in the Detection Memory. This block runs only once, at the end of each frame, after all blocks in the image have been analyzed. Because of the way the blocks are searched, it is assumed that the points arrive ordered, so that locational proximity in the image corresponds to proximity in the list in which they are enumerated. The pseudocode for the algorithm is as follows:

```
for each local minimum found
  for the next DIST_THRESH local minima
    calculate the distance to the neighbour
      if the distance < CLUSTER_THRESH,
        1) calculate the midpoint of the two points
        2) copy midpoint coordinates to the neighbour's entry in the list
        3) set the original entry to zero.
```

This algorithm is implemented sequentially in hardware, making use of duplicate memories in the Detection Memory to increase the speed and decrease complexity.

3.15.6 Output Interface (list_cluster_FSM.v)

As part of the clustering algorithm state logic, outputs are provided to interface to either a BRAM or FSL. The data is outputted sequentially at the end of a frame, allowing for easy retrieval. Outputs are standard BRAM compatible, see Virtex-IIP BRAM datasheets for details.

3.16 Software Pong Game (main.c, main.h)

The primary component to the software Pong game was a discrete time, state space physics simulator based on Newtonian mechanics. The ball was modelled using a state space representation with its position and velocity was the state variables. A simple Euler's method ordinary differential equation (ODE) solver was used to determine the ball's next state. This was then followed by a thorough check of all the collidable objects to determine whether a collision has occurred. If a collision occurred, then a simple elastic collision was modelled following the conservation of energy.

3.16.1 2D_types (2d_types.c, 2d_types.h)

This is a library which defines the basic types required for two-dimensional descriptions on a Cartesian plane. There are two structures, point2 and vector2, which consists of an x and y coordinate. Both of these coordinates are stored as floating point values. In addition, this library provides functions for the basic operations for these structures, which include:

- point-vector addition
- vector through point-point subtraction
- vector-vector addition
- vector scaling
- vector dot products
- finding the magnitude for vectors
- finding unit vectors
- finding both left and right normal vectors

3.16.2 ball (ball.c, ball.h)

This library was constructed to maintain the state space model of the ball. Each state consisted of three data types representing the position, velocity and radius of the ball. Also, there was a function to determine the direction vector from the ball to a given wall. This provides an essential role in ensuring that the radius of the ball is accounted for when simulating the physical interactions between the ball and its surroundings.

3.16.3 collidable (collidable.c, collidable.h)

This library was constructed to define all objects that the ball could collide with. At the moment, it consists of a single structure called wall. There are four data types representing the origin, unit direction vector, length and normal vector of the wall. In addition to the wall definition, there are a number of functions which provide the functionality for the physics:

- checkCrossing – this function provides a quick check to determine whether the ball will pass from one side of the wall to the other, which will result in a collision
- checkParallel – this function provides a quick check to determine whether the ball is travelling in a direction that is parallel to the wall

- findCollisionPoint – given the ball's initial position and its unrestricted position at the next step, this function will determine at the point of collision between the ball and the wall
- reflectVector – this function will calculate the reflected vector given the incident vector of velocity upon an ball-to-wall collision

3.16.4 draw (draw.c, draw.h)

This library is responsible for drawing sprites to the frame buffer in memory. It consists of four functions:

- an initialization function which clears the screen
- drawing rectangles which defined borders to draw the outline of the game
- drawing ball sprites with a 5 pixel radius
- erasing ball sprites by drawing over them with the background color in order to minimize bus transactions since frame buffers will not require redraws every time

3.16.5 vga (vga.c, vga.h)

This library was included with the slideshow reference project and provides a high level interface for setting and clearing frames. We added some minor modifications, such as RGB constant definitions, to make the library more suitable for our project.

4 Design Tree

Description of notation: Directories are in Normal Text while descriptions are *italicized*

- project – *root directory*
  - hw – *directory containing all the hardware designs of the project*
    - dead – *pcores with limited or unproven functionality*
      - bplbaccess_v1_01_a – *an attempt to generate a PLB master through the plb_if core; the PLB slave was confirm to read and write but the master could only read and write from DDR memory address 0x0*
      - plb_m_v1_00_a – *an attempt to generate a PLB master by following the convention in the IBM PLB datasheet; the read and write access was confirmed for a hardware synthesis with BRAM but only a read access was successful on DDR memory*
      - plb_read_test_v1_00_a – *an attempt to generate a PLB master by editing the VGA hardware core; only burst reads and writes from DDR2 memory address 0x0 and subsequent addresses were successful*
      - vidcap_v1_00_b – *an attempt to synthesize the point detection core with the video capture core; this core was not simulated since the video input signals could not be generated – as a result, we do not have a clear idea where the fault lies*
    - living
      - vidcap_v1_00_a – *a modified core from the VDEC1 reference design which stores 12 horizontal lines of video into local BRAMs*
  - ref_designs – *fully functional XPS projects*
    - game_n_vga_output – *a demo project showcasing the VGA output, a number of test frames, and the physics simulator; use the 5 button inputs to scroll through the test frames and press the top and bottom button simultaneously to jump to the physics simulator*
    - video_capture – *a demo project showcasing the video capture storing 12 horizontal lines of video; the data is subsequently fed into the VGA output; since the BRAM only accounts for a small fraction of the frame buffer, the VGA core times out on subsequent memory accesses and thus misses the sync signals resulting a flickering display of the capture buffer*
  - sim – *fully simulated projects*
    - plb_if_bram_xps – *an XPS project which simulates the PLB master core plb_m_v1_00_a with a BRAM to test for memory read and write access*
    - plb_master_bram_xps – *an XPS project which simulates the PLB IF master core pplbaccess_v1_01_a with a BRAM to test for memory read and write access*

        - point_detection_sim_ise – *the complete simulation setup for the hardware detection core in ISE*
- sw – *directory containing all the hardware designs of the project*
    - drivers – *drivers for XPS projects*
        - dead
            - plb_access_v1_01_a – *drivers for software access to the slave registers*
        - living
    - game_code – *a repository of the physics simulator and game code*

5 References

[1] http://www.nada.kth.se/~tony/earlyvision.html

[2] http://www.microsoft.com/surface/index.html

[3] http://www.phonemag.com/pmatrix-multitouch-display-by-stantum-is-iphone-beating-touchscreen-02981.php

[4] http://cs.nyu.edu/~jhan/ftirtouch/

[5] http://www.xilinx.com/univ/xupv2p.html

[6] http://www.digilentinc.com/Data/Products/XUPV2P/slideshow_256mb.zip