The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

ECE532 Digital Hardware

Group Report

Title: **Virtual Xylophone (Idioscope)**

Team members:	Name:	Student Number
	Phil Messina	995395406
	Mike McLean	995515161
	Alex Wong	994617982

Submission Date:	Monday, April 5, 2010

Table of Contents

1. Overview

1.1. Project Goal

The goal of the project was to create a system allowing users to play a ‘virtual xylophone’ by giving the user visual and auditory feedback. With this project, the user would strike hand drawn squares on a sheet of paper using a baton (microphone), and an overhead mounted camera would actively track the baton (microphone). The microphone is connected to the Xilinx XUP-V2P board which would detect strikes, ‘hits’, of the baton on the playing surface, thus playing a specific tone for each virtual xylophone key ‘hit’.

1.2. Project Components

The various components required to develop the project consisted of the following:

- Xilinx XUP-V2P development board
- Camera to be mounted overhead playing surface with composite output feed
- Microphone with a red tip acting as the baton
- Speakers
- Paper indicating the playing field

1.3. System Block Diagram

Below is the system block diagram for the project. The diagram is intended to present how the individual blocks are connected to each other, as well as the direction of data flow.

The diagram consists of the key custom developed IP Cores, existing IP Cores used from Xilinx and existing projects, and some directly attached hardware components on the development board.

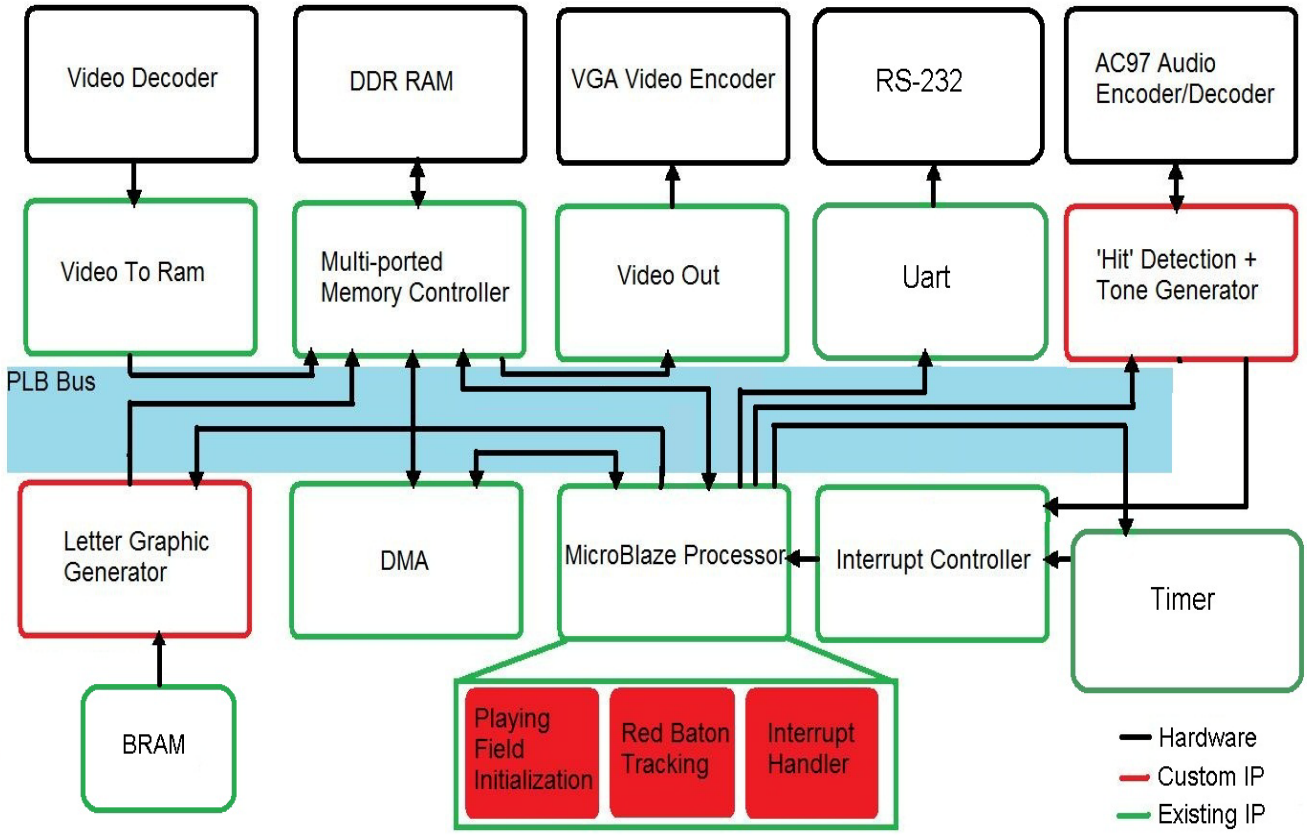

Figure 1: System Block Diagram

1.4. Brief Description of IP

The following table gives a brief description of the blocks found in the system block diagram above.

Block	Functionality	Origin
Video to Ram	• Takes video from video decoder, and stores into RAM according to Video Out specification (see appendix A)	Existing IP - Jeff Goeders Custom Pcore
Multi-Ported Memory Controller	• 5 ported memory controller • Offset 0x40000000: active video region • Offset 0x200000: Red baton tracking data processing • Offset 0x400000: Playing field initialization data processing	Xilinx IP
Video Out	• Displays video from RAM onto VGA attached monitor	Xilinx IP
Uart	• Used to output debugging text to a terminal	Xlinx IP

Hit detection/ Tone Generator	• Generates an interrupt if a baton hit is detected audibly • Generates audio tones for sound output, depending on which region was hit	Mike McLean/ Phil Messina Custom PCore – started from Xilinx AC97 audio processing module
Letter Graphic Generator	• Generates an overlaid letter of which note is being played over top of the live video feed	Phil Messina Custom PCore
BRAM	• Block RAM in the form of initialized ROM • Used to store letter information for printing notes to the screen	Xilinx IP
DMA	• Quickly copies live video frame from 0x40000000 of RAM into an offset Red baton tracking data processing location 0x200000	Xilinx IP
MicroBlaze Processor	• Software processor	Xilinx IP
Interrupt Controller	• Forwards the interrupt generated by the Hit detection module to the MicroBlaze processor	Xilinx IP
Timer	• Used to time ½ second for when the tone of the note, and graphic of the note is active	Xilinx IP
Playing Field Initialization	• On startup, this function scans the video and sets up virtual boundaries where hand-drawn lines are located on the playing field	Alex Wong Custom software
Red Baton Tracking	• Continually calculates the centroid of the first detected red-coloured object on the playing field	Mike McLean Custom software
Interrupt Handler	• On detection of a hit, this handler detects which boundaries the red objects lies within, and sends the coordinates to the Letter Graphic Generator, and the Tone Generator and activates them • A ½ second timer is also started to disable the above modules later	Mike McLean Custom software

Table 1: Description of IP Blocks

2. Project Outcome

2.1. Results

The end result of the project was very successful. As intended, the project can successfully detect the hand-drawn vertical lines on a sheet of paper over the playing field, creating up to a maximum of 7 different regions which represent the notes to be played with the baton. After the learning phase, an infinite loop is used to detect the centroid of a red object found in the playing field.

As the user plays the virtual xylophone, the auditory impacts of the baton to the ground are detected, which creates interrupts for the software system. The software then uses the most recently calculated red-object centroid to detect which region the baton is located in for that "hit", and activates the Tone Generator and Letter Graphics Generator for ½ second – giving auditory and visual feedback for each note played by playing a sound and displaying the note.

The only problems occasionally encountered are when the lighting is not ideal. In this case, both the playing field initialization code and red baton tracking code may fail to locate the lines and baton. To avoid this situation, good lighting is required where the contrast between lines and contrast between the baton and background are optimum. Also, it is ideal to have a background not containing any red, so that the red baton can be seen clearly and run our system in natural lighting conditions.

2.2. Future Improvements

To address the issues noted above, more robust algorithms could be proposed which better process the video so that a more accurate detection could be made of the boundaries and red baton. Since more sophisticated algorithms would imply greater processing times, it may also be beneficial to implement them in hardware instead of software to decrease lag. Since the existing detecting algorithms are relatively simple, the slowness attributed to them being implemented in software is not apparent.

A more ambitious improvement to the project would be to have active boundary detection. Currently, the playing field initialization is done only at the reset of the board, and so the boundaries are only calculated once. If the boundaries were continuously detected, then the camera would not need to be fixed, and would be allowed to move. Furthermore, this can be expanded even further to allow the rotation of the camera, so that the camera is completely free to move around in any direction. As stated earlier, this would require faster processing then what can be done in software, so this would most likely need to be implemented in hardware.

3. Block Descriptions

3.1. Hardware Modules

3.1.1. MicroBlaze

MicroBlaze, version 7.10.d was utilized to interface the hardware and software of the project.

- The microBlaze processor handles custom c code that: initializes the playing field, tracks in real time a red object, and handles interrupts.
- Interfaces with the Letter Graphics Generator to determine the correct graphic (musical note) that should be drawn to the VGA monitor.
- Interfaces with the 'Hit' Detection & Tone Generator to ensure the tone played corresponds to the note being played.
- Interfaces with DMA controller to transfer a single frame to a location in memory so the software can process red color detection.

3.1.2. PLB Bus

Plb_v46, version 1.03.a was utilized to provide communication amongst all IP Cores. To avoid putting too much stress on a single bus we used five PLB Busses in our system to interact with the Memory.

1. Used for the microBlaze to communicate with the DDR_SDRAM (memory), the 'Hit' Detection & Tone Generator module and the Letter Graphics Generator module.
2. Used for the XPS Central DMA controller to communicate with memory.
3. Used by the Letter Graphics Generator to write the correct graphic to memory.
4. Used by the TFT to get video data from memory.
5. Used by the Video to RAM Core to put data into memory.

3.1.3. Multi-Port Memory Controller (MPMC)

MPMC, version 4.03.a was utilized for this project. Four ports were utilized as inputs from various IP Cores, and one port was utilized as an output to the Video Decoder (as seen in Figure 1 above). Having the ability to utilize multiple ports allowed for easier implementation of interfacing many different modules especially since we were working with a live video feed and in some instances drawing a graphic overtop the live feed.

3.1.4. Video Out (XPS Thin Film Transistor (TFT) Controller)

The XPS Thin Film Transistor Controller, version 1.00.a is a hardware core found in the Xilinx EDK library. The XPS TFT controller reads video data in RGB from RAM

and feeds it to a VGA monitor. The controller is connected a 32-bit wide PLB bus which is connected to the MPMC. The core starts reading at address 0x40000000, the location of the first pixel stored in RAM, and continues reading the next pixel at the next address location until the last pixel is read.

3.1.5. Video to RAM (Custom Logic)

This is a custom IP Core developed by one of our peers: Jeffrey Goeders. This core takes in an active video feed from the video decoder of the VDEC1 Video Decoder Daughter Card supplied with the Xilinx board and stores frames in a specific RAM location (memory address 0x40000000). These frames get converted from YCbCr format to RBG so that a VGA monitor can properly display the active video feed. Once the frame has been properly encoded to RGB it is displayed on a VGA monitor.

3.1.6. Block RAM (BRAM)

Bram_block, version 1.00.a was utilized to store ROM blocks consisting of COE files containing hexadecimal representations of a letter that is being drawn to the screen. Mike McLean had created a parser in c++ that takes as input a monochrome bitmap image consisting of a single letter seen in Figure 2 below.

Figure 2: 128x128 Pixel Image

The parser processes the image files and converts the images to COE files (BRAM initialization files) in which black is depicted as a binary 0 and white is depicted as a binary 1 which can be seen in Figure 3 below. The COE files consist of a read depth size of 16 bits, and a read width size of 1024 bits. The Figure below only depicts the first 384 bits wide.

```
;COE file generated from monochrome BMP by Mike McLean
;Block memory of depth=128, and width=128
memory_initialization_radix=16;
memory_initialization_vector=
00000000000000000000000000000000000000000000000000000000000000000000000000000003ffffff8000000000000...,
000000000000fffffffe00000000000000000000000001fffffffe00000000000000000000000001fffffffe800000000000...,
0000000000007fffffffe00000000000000000000000007fffffffe00000000000000000000000007fffffffe00000000000...,
0000000000fffffffff8000000000000000000fffffffffffc0000000000000000000fffffffffffc0000000000...,
000000003fffffeffffff000000000000000003fffffe3fffff000000000000000000007fffffe3fffffc000000000...,
000000001ffffff01ffffff000000000000000001ffffff01ffffff000000000000000001fffffe00ffffff800000000...,
00000000ffffff0007fffffe00000000000000000ffffff0003fffffe000000000000003ffffff0003ffffff00000000...,
00000007fffffc00007fffffc000000000000007fffffc00007ffffff000000000000007fffffc00003ffffff0000000...,
0000003fffffe0000007fffffc0000000000007fffffe0000007fffffc0000000000007fffffc0000007fffffe000000...,
000007fffffe00000000fffffffc00000000007fffffe00000000fffffffc000000007fffffe000000007fffffc00000...,
00003fffffffffffffffffffff8000000003fffffffffffffffffffc000000003fffffffffffffffffffc0000...,
0000ffffffffffffffffffffffff80000001ffffffffffffffffffffff80000001ffffffffffffffffffffff8000...,
0007ffffffffffffffffffffffe0000007ffffffffffffffffffffffe000000ffffffc0000000000007fffffe000...,
001ffffff000000000000000ffffffc00007ffffff000000000000000ffffffc00007ffffff000000000000007fffffc00...,
01ffffff000000000000000003ffffff8001ffffff000000000000001ffffff8007ffffff000000000000000001fffffc0...,
0ffffff800000000000000003ffffff01fffffff800000000000000003ffffff83ffffff800000000000000003ffffff8...;
```

Figure 3: COE File of the letter 'A'

The Xilinx IP (CORE Generator & Architecture Wizard), more specifically the Distributed Memory Generator v3.4 provided by the ISE tool was utilized to create a ROM block to store the COE file so that the hexadecimal contents could be accessed by the Letter graphics Generator Core (as described in section 3.1.7 below). The image seen in Figure 2 above had a maximum size of 128x128 pixels. The reason for this is because the ROM block used to store the letter via COE file allows a maximum read depth of only 16bits. This is seen in Figure 4 below where the radix corresponds to the ROM Read Depth.

Specifying Memory Contents Using a COE File

The initial contents of the memory can be defined using the COE file, which consists of two parameters: memory_initialization_radix and memory_initialization_vector. Each line is terminated by a semicolon.

- **memory_initialization_radix**. The radix of the initialization value is specified here, with the choices being 2, 10, or 16.
- **memory_initialization_vector**. Each row of memory elements are defined with a binary, decimal, or hexadecimal number having an equivalent binary value that represents whether an individual memory element along the width of the row is set to '1' or '0.' Each row of memory initialization is separated by a comma or white space, up to the depth of the memory. Negative values are not allowed.

An example COE file:

```
; Sample Initialization file for a 16x32 distributed ROM
memory_initialization_radix = 16;
memory_initialization_vector =
23f4 0721 11ff ABe1 0001 1 0A 0
23f4 0721 11ff ABe1 0001 1 0A 0
23f4 721 11ff ABe1 0001 1 A 0
23f4 721 11ff ABe1 0001 1 A 0;
```

Figure 4: Distributed Memory Generator v3.4 Product Specification

So to achieve a letter that appears square on the screen, it was decided to have a ROM read width of 1024bits.

16bits*1024bits = 16384pixels = 128*128pixel image

7 binary COE files as well as 7 ROM blocks were created to store each individual letter (A through G) that can be drawn to the screen.

3.1.7. Letter Graphic Generator Core (Custom Logic)

This is a custom core that was developed to draw a letter to a screen overtop a live video feed. As previously stated, the letters are stored in BRAM ROM blocks allowing for a 128x128 pixel resolution letter to be drawn to the VGA. This core includes PLB slave registers which are controlled by the software system determining where on screen the letter is to be drawn, what letter to draw, the colour of the letter displayed as well as for how long the letter should be drawn. In addition, this core implements Master Service Configuration to obtain Master User Logic control so that this core can directly interact with and write to the PLB Bus. The entire state diagram can be found in Appendix B.

The design of the Finite State Machine is broken down into 3 discrete stages;

1. Start & Determine which Letter to print

   PLB slv_reg7 is a software controlled register that starts the FSM (letter generation process) once this bit has been changed from a 0 to a 1. Once a 1 is asserted, slv_reg8, which contains the region number 0 through 6, is examined to determine which letter A through G gets printed to the screen and data from the corresponding ROM block at ROM_Address 0x0 is stored in a register called DATA. Figure 5 below depicts the physical placement of each letter.

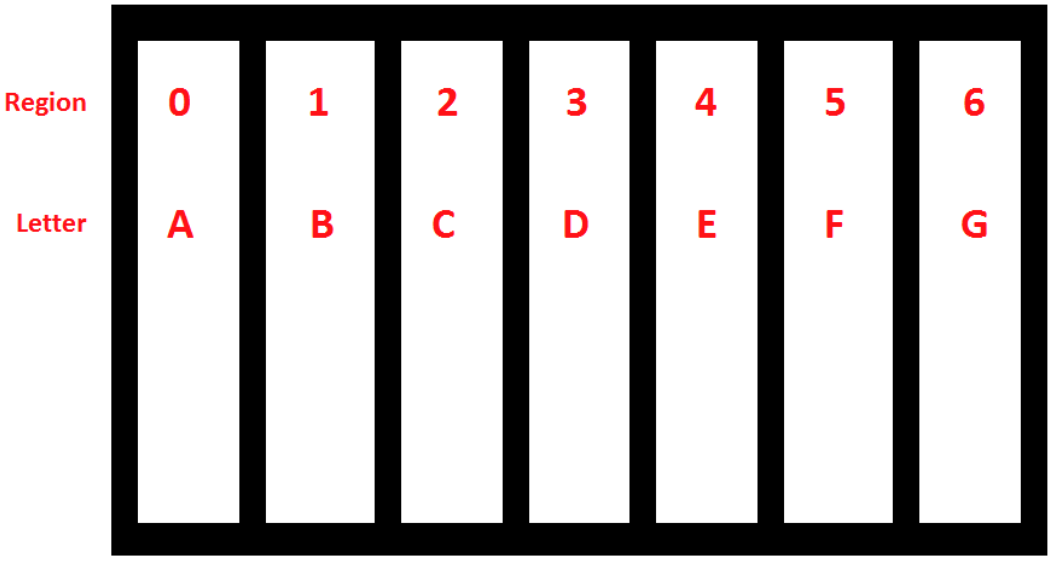

Figure 5: Physical Keyboard Layout

2. Parse DATA register

   Each row in the COE is individually considered at this stage. Two 128 bit counters are implemented in order to draw a 128x128 pixel letter to the screen. The first row of the COE file, all 1023 bits of the DATA register is examined by left shifting the data. Whenever a 0 bit in the DATA register is seen, col_count is incremented and the next bit in the DATA register is examined. When a 1 bit is seen in the data register, it means that a single pixel must now be drawn to the VGA (See stage 3 below). When the column counter reaches 127, it gets reset and a row counter (row_count) increments and now we start to

see the 128x128 pixel block begin to form. When all 1024 bits of the first row of the COE file have been examined most counters are re-set (except for row_count), and the ROM_address is incremented by 1 to obtain the next set of 1024 bit row data to be placed in placed in the DATA register. This process repeats until the ROM_Address has incremented 16 times, meaning that all data in the COE file has been exhausted. At this point all counters are then re-set and the process repeats. This process stops after 0.5 seconds (which we determined is ideal for a note to be displayed) when the software causes an interrupt which sets slv_reg7 to 0. The state diagram found in Appendix B depicts this design.

3. Draw Pixel to VGA

In order to draw a pixel to the correct address location, both col_count and row_count are added to the TFT base frame memory address 0x40000000. This computation is done as follows;

IP2Bus_Mst_Addr <= 0x40000000 + (4096*row_count) + (4*col_count);

Where IP2Bus_Mst_Addr is the address where we want to write pixel data to. This calculation was taken from the TFT data sheet which can be found in Appendix A. Once the address has been computed, the IP2Bus_MstWr_Req (BUS Write Request) is set high in order to draw a pixel at this specific location. Once the bus acknowledges that the pixel data has been written, which is done by examining Bus2IP_Mst_Cmplt (BUS Master Complete signal), the Master Write Request goes low and stage 2 above continues. Figure 6 below shows a letter being printed overtop a live video feed.

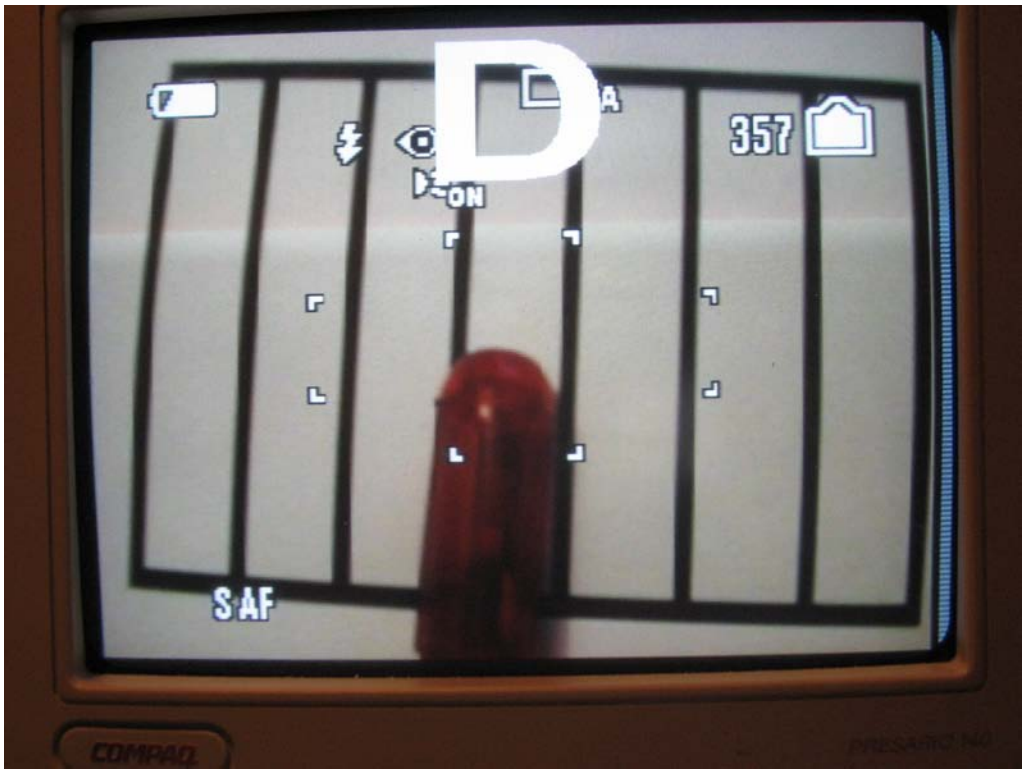

Figure 6: Implementation of Letter Graphics Generation Core

3.1.8. Hit Detection/ Tone Generator Core

This core was not provided in the IP Catalog. A hardware implementation utilizing the AC97 Audio Codec Controller that takes in input from a microphone and outputs a single tone was found online by our peer Nancy Chong; http://embedded.olin.edu/xilinx_docs/projects/audio-v2p.php [1].

In order to integrate this core with the system we had to create a custom Verilog audio core to interface with the interrupt controller which interacts with the software causing an interrupt in the system whenever a 'hit' was detected by the AC97 Audio Controller. Within this custom pcore, we instantiated the ac97_audio and processing modules in order to correctly process audio only when an interrupt occurred within the system.

Using a microphone connected into the Xilinx Board, the Audio Controller actively detects a sound above a threshold of 0x4 and generates an interrupt on the PLB to the microblaze processor from the Verilog audio core. The software then interacts with the Letter Graphic Generator to draw a specific note to the screen as well as play a specific tone corresponding to the note being played by interacting with the audio cores tone generator. Figure 7 below depicts this process. The processing.v module that was included in the package from the website listed above was modified to generate 7 different tones as output for the Audio Controller. The processing.v module already had Verilog code to generate one square wave tone. This code was then expanded to produce 7 different tones starting at a low frequency noise and continually increasing to a higher frequency tone.

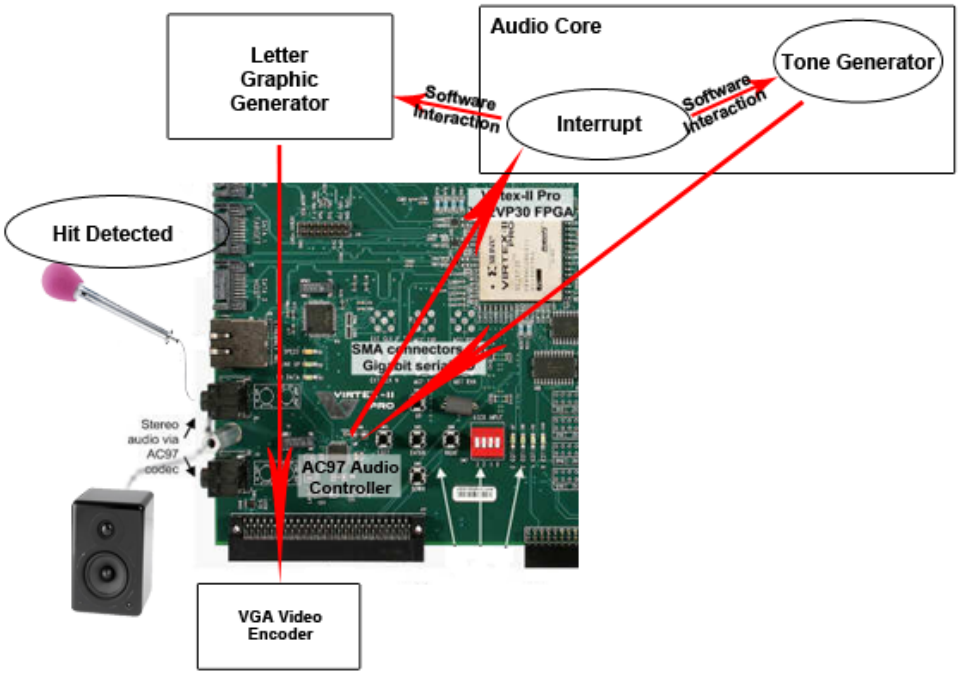

Figure 7: Audio Core Implementation

3.1.9. DMA (XPS Central DMA Controller)

The XPS Central DMA Controller, version 2.00.b is an IP in the Xilinx EDK library that allows the copying of data from one memory location to another without the need of a processor. The controller is connected to a 32-bit wide PLB bus, which is connected to the MPMC. The DMA Controller is used to copy a frame from memory location 0x40000000 to 0x40200000 such that the Red Color Detection Module is able to operate.

3.1.10. XPS Interrupt Controller

The XPS Interrupt Controller, version 1.00.a is an IP in the Xilinx EDK library that takes in inputs from multiple peripheral to handle multiple interrupts and produce a single output through a PLB bus to a MicroBlaze processor. The input of this block is connected to a XPS GPIO module, which will be used to for our “hit” detection.

3.1.11. Timer (XPS Timer/Counter)

The XPS Timer/Counter, version 1.00.a is an IP in the Xilinx EDK library, which is connected to a 32-bit PLB bus as a slave. The counter will increment and generate an interrupt through the PLB bus when a value is reached. The timer is used to control the duration of the tone and graphic that will appear once a “hit” has been detected in our system.

3.1.12. XPS UART Lite

The XPS UART, version 1.00.a is an IP in the Xilinx EDK library that provides an interface for serial data transfer. The core is able to take serial data and transmit it in parallel through the connected 8-bit PLB bus. Similarly, the controller can transmit serial data from incoming parallel data.

3.2. Software Modules

3.2.1. Red Colour Detection

This software module is used to compute the centroid of a single red object located on the screen. This routine in the software is run as a continuous loop, which constantly recalculates the centroid. The centroid is then used (by the interrupt handler) to calculate which region the centroid is within – or which ‘note’ the baton is playing

The live video is stored as single continuously updated frame at the beginning of the DDR SDRAM location 0x40000000. The active portion of the frame is 640x480

pixels, but is stored in memory in a block of 1024x512 words, where the extra 384 words of each row is ignored. Also, the last 32 rows are ignored, since they do not hold any information. Each word holds a single pixel in the RGB colour space in the following format.

Pixel Address	Bits	Description
TFT Base Address + (4096 * row) + (4 * column)	[0:7]	Undefined
	[8:13]	Red Pixel Data: 000000 = darkest → 111111 = brightest
	[14:15]	Undefined
	[16:21]	Green Pixel Data: 000000 = darkest → 111111 = brightest
	[22:23]	Undefined
	[24:29]	Blue Pixel Data: 000000 = darkest → 111111 = brightest
	[30:31]	Undefined

Figure 8: Image Data, As Stored in Memory

Note that since each colour channel is only 6 bits, they have a minimum value of 0x00, and a maximum value of 0x3F (see appendix A for more details).

Because the video frame is refreshed at 30 frames per second, the software would not be fast enough to process a single frame before it is refreshed. To solve this problem, the algorithm first starts by initiating a DMA memory transfer from 0x40000000-0x401FFFFF to 0x40200000-0x403FFFFF. Then, the software processing is done on location 0x40200000 which resembles a still frame from the instant the DMA was called.

The red colour detection module is essentially two nested for loops that scan through the entire frame looking for red. It does this by first (1) going row by row, and scanning across looking for the beginning and ending x coordinate of a region deemed as ‘red’. Once this is done, the x centroid is the midpoint of these two x coordinates. Next (2), the algorithm goes through each column in between the beginning and ending x coordinate, and scans downward, looking for the beginning and ending y coordinate of a region deemed as ‘red’. The y centroid is then the midpoint between these two y coordinates. Figure 9 below demonstrates the described algorithm.

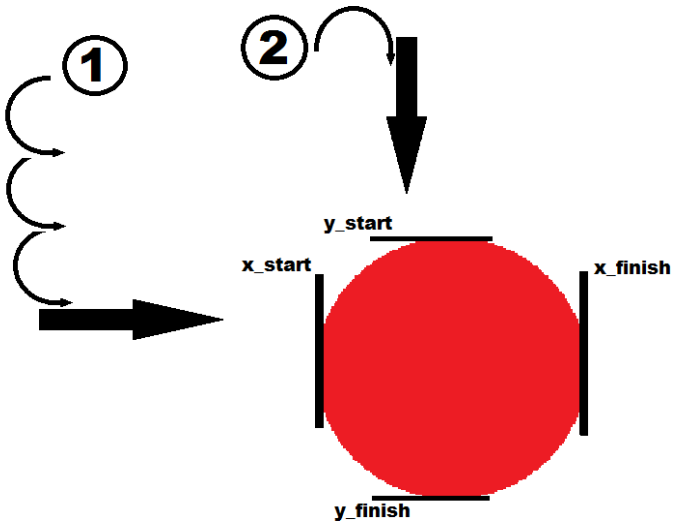

Figure 9: Red Colour Detection Algorithm

To make the process of defining 'red' less unambiguous, the algorithm (as it scans) performs a thresholding operation. This operation looks at each channel of the RGB colour space for each pixel, and if the value is above a certain threshold, it considers it as maximum (0x3F), otherwise it considers it as a minimum (0x00). To demonstrate this better, the following Figures 10 and 11 show what the algorithm sees before and after thresholding as it is looking for a 'red' object.

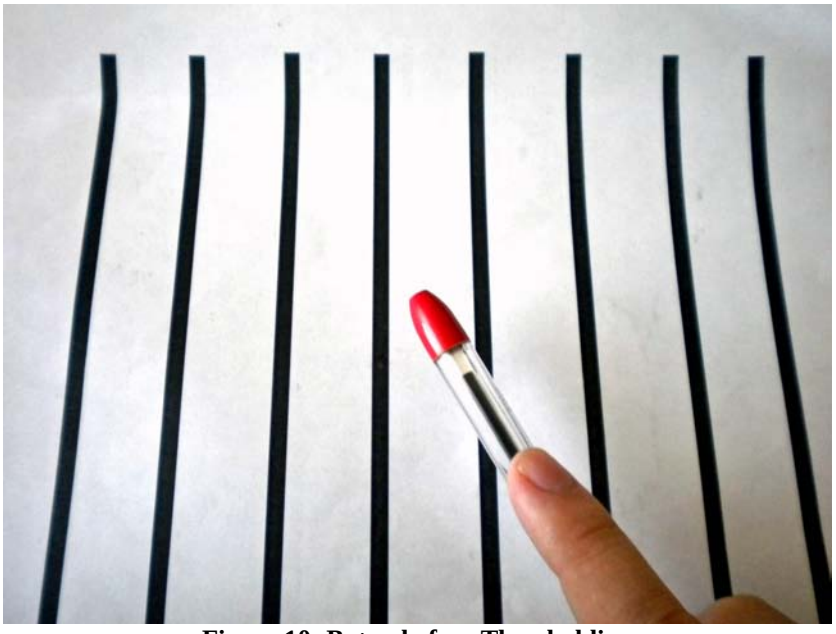

Figure 10: Baton before Thresholding

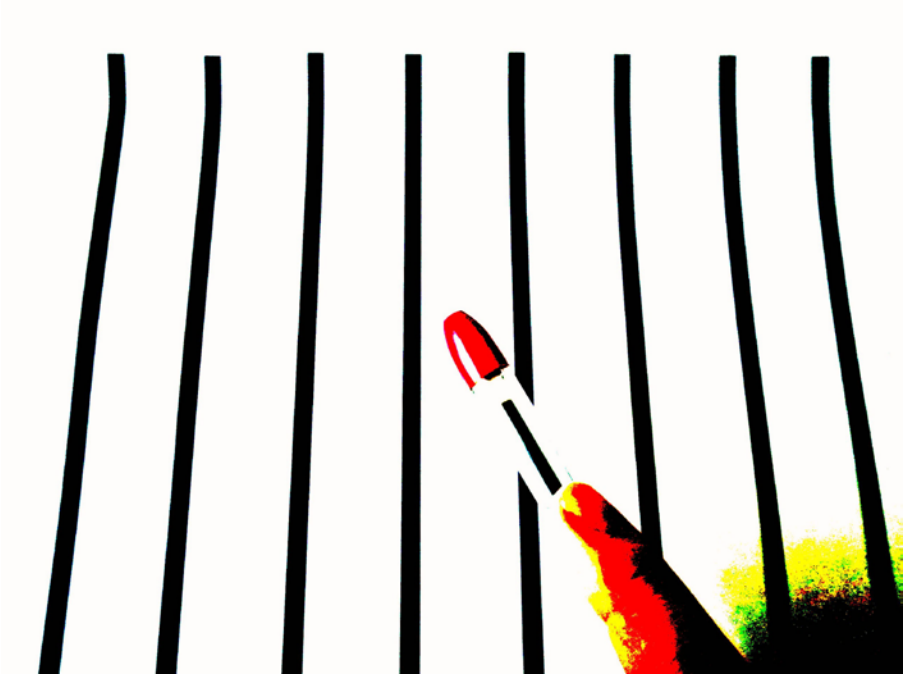

Figure 11: Baton after Thresholding

As can be seen from the Figures above, what is considered ‘red’ after thresholding is no longer unambiguous; it is simply when the R channel is the maximum (0x3F), and the G and B channels are the minimums (0x00).

As a result of the way the algorithm progresses, only a single red object will be detected in the frame, and the red objects will have a top-left priority for being detected.

3.2.2. Interrupt Handler

The primary interrupt handler is called whenever the hit detection/tone generation module detects a sound above a certain threshold. This purpose of the interrupt handler is to decide which note was hit, and activate the graphics generator and tone generator appropriately. To accomplish this, the interrupt handler performs several consecutive tasks:

1. Disable interrupts.
2. Compare the most recent centroid coordinates with the array of region boundaries to determine which region it lies within (if it’s out of bounds, then return).
3. Write the region number (0-6) to the graphics generator and tone generator registers, and activate them (by writing ‘1’ to their ‘start’ registers).
4. Start the timer to interrupt after 0.5 seconds.
5. Re-enable interrupts.

After the above steps are complete, the tone generator will begin to sound a specific note for the region hit, and a letter will display on screen for the region hit (A for region 0, B for region 1, ect.).

After 0.5 seconds, the timer will cause another interrupt which will call the secondary interrupt handler. All this handler does is de-activates the tone generator and graphics generator by writing a '0' to their 'start' registers. This causes the letter to disappear (because it is written over within 1/30 of a second by live video data), and the tone to stop.

3.2.3. Playing Field Initialization

The Playing Field Detection Module is a software module using the MicroBlaze processor, designed to detect the different regions on the playing field when our system first starts up. Each region will output a different note when a red baton strikes it.

As described above, the video data is stored memory with each pixel located in a single block of memory and 1024 pixels in one line (only 640 pixels displayed) stored sequentially. From this, we can simply calculate the memory address for a specific pixel using the equation:

Address = StartAddress + row * 1024 + column

Every read pixel will be converted from RGB color values to YCbCr using this conversion [2]:

Y = 16 + (65.738 * R)/256 + (129.057 * G)/256 + (25.064 * B)/256 [2]

The Y component represents the luminance. This is important because it allows the detection of black lines. For black, the Y value is expected to be low while white will produce a high Y value.

The module first looks for the piece of paper where our playing field will be by searching for a pixel that has a Y value greater than 0x9FFF. Once found, a search for vertical lines is started. To find the vertical black lines, a scan is performed horizontally at a certain row. If the Y value is greater than our threshold (0x80FF), the location is stored indicating that a black line is found. This will continue until all 8 lines are found (creating 7 boundaries) or the end of the line is reached. Figure 12 below depicts this process.

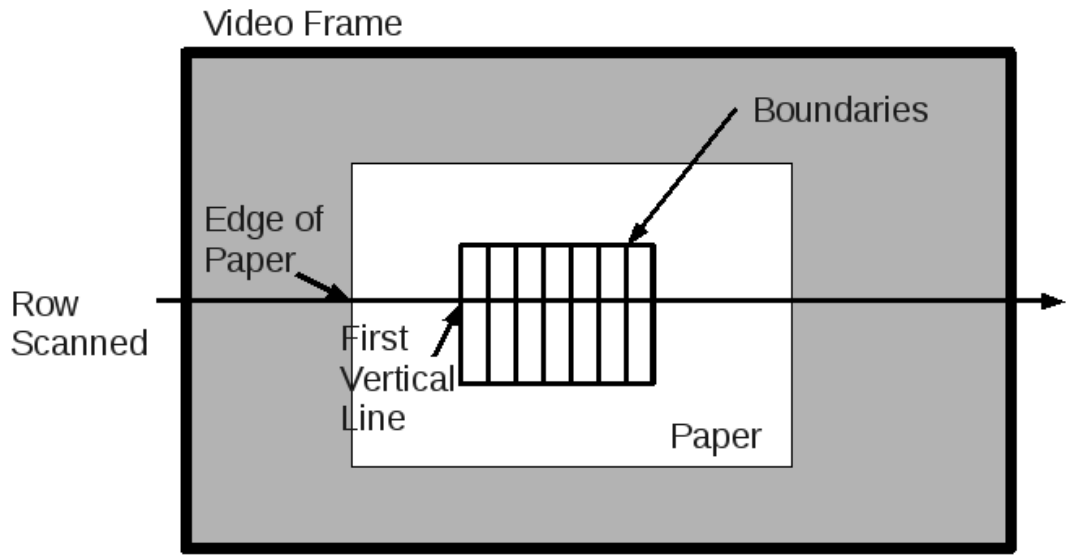

Figure 12: Horizontal Scan for a Row

Once the vertical lines are found, the upper and lower bounds must be found using the same strategy as above. A vertical sweep will be performed until a pixel that meets our threshold is found indicating that a line is found at a certain coordinate. The coordinates will be stored in memory where the Interrupt Handler Module can access it.

Algorithm

1. Find row to sample (row searched are predefined)
2. Read pixels along that row (sweeping horizontally)
3. Locate the piece of paper that our playing field will be on. This requires luminance value to be high of the paper
4. Once playing field is found, continue to read the pixels in that row to locate the first vertical line. A line is located when a pixel meets a threshold for its Y value
5. Continue scanning for the remaining vertical lines
6. Sample other rows to ensure vertical lines are consistent
7. Once vertical lines are located, read pixels vertically (in the same column) to find the upper and lower lines
8. Store boundaries into memory for Interrupt Handler module to process

4. Description of Our Design Tree

The following table illustrates the content of the folders that we have submitted with regards to our project. A README file has also been included at the top level directory for the same purpose.

Folder Name	Description
System	This folder contains the XPS project
System\pcores	This folder contains the code for existing as well as our own custom pcores. Key files include: 1. user_logic.v (for Letter Graphic Generator) System\pcores\draw_note_v1_00_b\hdl\verilog\user_logic.v 2. A.v to G.v (modules instantiating each individual ROM block) System\pcores\draw_note_v1_00_b\hdl\verilog\A.v – G.v 3. user_logic.v (for Hit Detection) System\pcores\audio_v1_00_a\hdl\verilog\user_logic.v 4. processing.v (for Tone Generation) System\pcores\audio_v1_00_a\hdl\verilog\processing.v 5. ac97_audio.v (for input and output audio processing) System\pcores\audio_v1_00_a\hdl\verilog\ac97_audio.v These files are commented for further review.
System\pcores\ video_to_ram_v1_00_a	This folder contains a custom IP Core developed by one of our peers; Jeffrey Goeders.
System\pcores\ led_debug_mux_v1_00_a	This folder contains a custom IP Core developed by one of our peers; Jeffrey Goeders. (Was not used in this project)
System\sw	This folder contains the software code used for this project. Key file includes: 1. video_setup.c (used for Red Colour Detection, Interrupt Handler, Playing Field Initialization) System\sw\video_setup.c
Documentation	This folder includes all of the documentation files for our project. Project Proposal: Idioscope Project Proposal Group Report: Group Report PowerPoint of our Presentation: Presentation Playing Field: Playing Field

Table 2: Description of Design Tree

5. References

[1] EmbeddedComputing, “AC’97 Audio Codec Controller for Digilent XUP-V2P,” 2007. [Online] Available: http://embedded.olin.edu/xilinx_docs/projects/audio-v2p.php. [Accessed: March 17, 2010].

[2] Jack, Keith. "YCbCr to RGB Considerations", intersil. [Online] Available: http://www.intersil.com/data/an/AN9717.pdf. [Accessed: March 30, 2010]

[3] Xilinx. (September 16, 2009). Product Specification: Thin Film Transistor (TFT) Controller (v2.00a).

6. Appendix A

The following is an applicable excerpt from the Xilinx product specification for the 'Thin Film Transistor Controller (v2.00a)' [3].

Video Memory

It is important to design the system so that there is a sufficient bandwidth available between the XPS TFT controller and the PLB memory device to meet the video bandwidth requirements of the TFT. Furthermore, there must be enough bandwidth available for rest of the system. If the bandwidth requirement of rest of the system is more, the TFT clock frequency can be reduced. However, reducing the TFT clock frequency also lowers the refresh rate of the screen. This may lead to a noticeable flicker on the screen if the TFT clock is too slow. The PLB master interface logic has the ability to skip reading a line of data if it fails to finish reading data from a previous line because of shortage of PLB bandwidth. This prevents the XPS TFT controller losing synchronization between the PLB and TFT interface logic. Note that extreme shortage of available bandwidth for the XPS TFT controller may cause the screen to appear unstable as stale lines of video data are displayed on the screen.

The video memory is expected to be arranged so that each RGB pixel is represented by a 32-bit word in memory. The video memory should be stored in a 2 MB region of memory consisting of 1024 data words (1 word = 32 bits) per line by 512 lines per frame. Out of this 1024 x 512 memory space, only the first 640 columns and 480 rows are displayed on the screen.

For a given row (0 to 479) and column (0 to 639), the pixel color information is encoded as shown in Table 4.

Table 4: **Pixel Color Encoding**

Pixel Address	Bits	Description
TFT Base Address + (4096 * row) + (4 * column)	[0:7]	Undefined
	[8:13]	Red Pixel Data: 000000 = darkest → 111111 = brightest
	[14:15]	Undefined
	[16:21]	Green Pixel Data: 000000 = darkest → 111111 = brightest
	[22:23]	Undefined
	[24:29]	Blue Pixel Data: 000000 = darkest → 111111 = brightest
	[30:31]	Undefined

Figure 13: Address Calculation Used to Write Pixel Data

7. Appendix B

The following is a state diagram of the Finite State Machine implemented for the Letter Graphic Generator core.

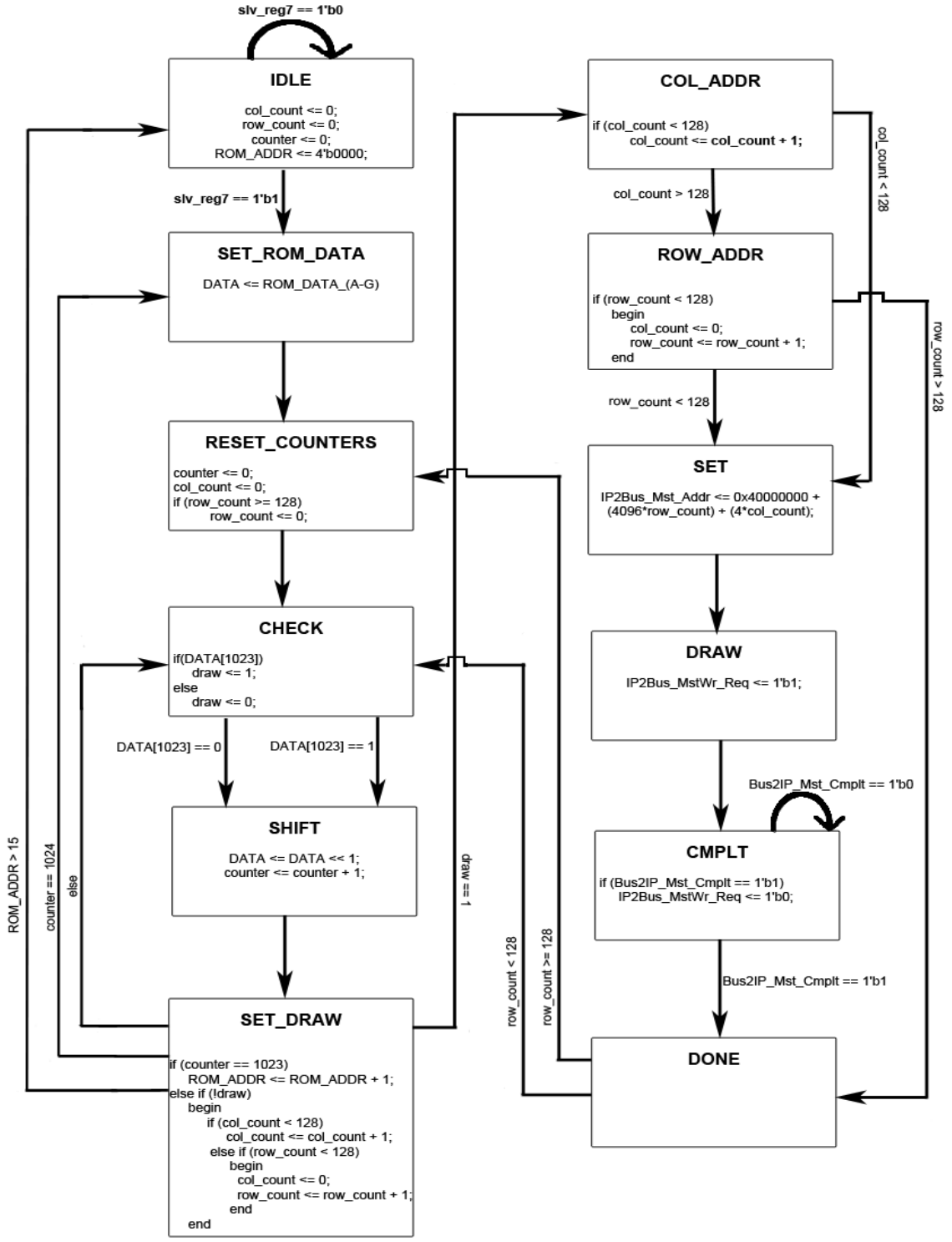

Figure 14: Letter Graphics Generator FSM