Augmented Reality in a Virtual Reality Setup

Video transfer and processing

by

Niels Versluis
Chy Lau

To obtain the degree of Bachelor of Science at the Delft University of Technology, to be defended on Thursday June 30, 2016 at 15:30.

Students: Niels Versluis 4227646
           Chy Lau 4155955

Thesis committee: Prof. dr. E. Eisemann, TU Delft, supervisor
                  Dr. ir. M. A. P. Pertijs, TU Delft
                  Prof. dr. ir. A. J. van der Veen, TU Delft

Project duration: April 18, 2016 – June 30, 2016
Abstract

This thesis covers part of the development of the project "Augmented Reality in a Virtual Reality Setup". The covered part is about the communication between a Personal Computer (PC), and an FPGA (Field-Programmable Gate Array) and vice-versa. This communication is crucial for transfer of a camera stream from the FPGA to the PC and applying video processing there, and for getting the processed stream back to the FPGA in the correct format.

What was achieved is the following datastream: The camera stream from the FPGA is sent to the computer using USB, at an average 14 frames per second in full colour. From there, the augmented reality overlay from the PC is created by rendering a frame and then replacing "transparent" pixels, pixels that should be replaced by the camera image, by pure green (RGB 0,255,0), which is a format the FPGA expects. The resulting video output can be sent back to the FPGA via HDMI, where further processing takes place. For the data transfer USB 2.0 is used, which is not ideal for the transfer, but is workable within the scope of the project our system is used in. For future projects, USB 3.0 or 3.1 is preferable.
## Contents

1 Introduction 1

2 Programme of Requirements 2
   2.1 Functional requirements 2
   2.2 System requirements 3
   2.3 Development requirements 4

3 Design Process 5
   3.1 Data Transfer Interfaces 6
      3.1.1 Digilent Serial Peripheral Interface (DSPI) 6
      3.1.2 Digilent Parallel Transfer Interface (DPTI) 7
   3.2 Design Choices 7

4 Implementation 8
   4.1 Data Transfer from FPGA to PC 8
      4.1.1 DPTI API 8
      4.1.2 Unity3D 8
      4.1.3 Syncing Transmitter and Receiver 10
      4.1.4 Transfer Speed 12
   4.2 Video Processing 13
      4.2.1 Hybrid-Infrared Vision 13
      4.2.2 AR Overlay 14
   4.3 Implementation FPGA 15
      4.3.1 DPTI Controller 15
      4.3.2 Byte Splitter 16

5 Results 18
   5.1 FPGA to PC transfer 18
   5.2 PC to FPGA transfer 18

6 Conclusion 19
   6.1 Conclusion 19
   6.2 Future works 19

A Appendix 20
   A.1 DPTI DLL 20
   A.2 DPTI Controller 23
   A.3 Byte Splitter 28

Bibliography 30
Introduction

Currently, the Virtual Reality (VR) and Augmented Reality (AR) markets are booming, with an expected market value of $150 billion by 2022.[1] Especially VR seems to be benefiting from the gaming market, which has allowed the technology to be developed quite well over the past few years, with advanced consumer-grade VR technology becoming available to the masses such as the Oculus Rift and the HTC Vive.[2]

While the VR market seems to have clear big players (Oculus, HTC, Sony), the AR market is still much more infantile, with no clear leaders[3]. The reason for this is that all currently available AR systems either expensive, such as the Microsoft Hololens[4] or have limited field-of-view, such as the Meta 1[5], although we have to admit the upcoming Meta 2[6] seems like a formidable device. However, it has not yet been released, so it remains to be seen if they can deliver what they promise.

The problem is that there currently is no AR system that is both affordable and technically advanced enough to be attractive to potential customers. Our proposed solution is as follows: Combine the best of both worlds: Use a good VR Head Mounted Display (HMD) and a pair of good cameras to turn that VR HMD into an affordable, great-quality AR HMD. Although other attempts have been made by hobbyists, ex.[7], these attempts all used a PC for all of the processing, causing the camera image to lag noticeably behind the actual movements of the head.

Our plan is to eliminate most of the input lag of the camera image, by letting a Field Programmable Gate Array (FPGA) do the mixing of the camera signal and the augmented layer that is transferred by a PC. An FPGA consists of programmable hardware.[8]

This means that, instead of having the camera image be processed by software, it will be processed by our own hardware logic, specifically designed to output the image as fast as possible to the HMD. This approach may still cause lag on the augmented layer, since that data still has to pass through the PC, but the camera stream will be directly input into the mixing hardware, causing minimal lag, and therefore hopefully allowing the user to look at the real world through the HMD as if they were looking with their own eyes.

This thesis is about the software on the PC that we use to control the FPGA, and the data transfer from the FPGA to the PC. The FPGA expects a certain image format in order to be able to correctly process it in a fast way. This means that we have to write PC software that can create an output signal on the PC’s HDMI output that is supported by the FPGA. Modifications include distorting the image and marking pixels for replacement by the camera image.

In order to process the real world around the device, we also need to get the camera image onto the PC. This is where the FPGA to PC transfer part comes in. We have to find a way to get useful information about the image to the PC as quickly as possible, while also keeping our bandwidth high, because uncompressed video signals generally use a lot of bandwidth, and we do not have the resources on the FPGA to compress video as these algorithms are usually very complex.[9]

This document consists of this introduction, followed by the programme of requirements, in which the requirements for success of this project are described. Next is the explanation of the design process, where we elaborate on our design and choices. The next chapter, we describe how we implemented our design. After that come the results, where we summarize our achievements and compare them to what we wanted to achieve, and finally we draw our conclusions. Appended to this document are c# and VHDL code files and the bibliography.
2 Programme of Requirements

2.1. Functional requirements

1. The PC should send all pixels that need to be replaced by camera image as RGB value [0,255,0].

   The FPGA part that blends the computer image with the camera is implemented by another subgroup of the project, which can be found in the sub-thesis on the Implementations on an FPGA [10]. They designed the hardware so that whenever a pixel has RGB value [0,255,0], which is full green, this pixel should be replaced by a pixel from the camera image. One could say all the green pixels act as if they are a greenscreen. This means that the computer generated image should have all transparent parts marked as pure green, so the FPGA knows which pixels to replace.

2. The PC should send an image that is already barrel distorted correctly to match the pincushion distortion of the Oculus Rift DK1.

   Because the Oculus Rift DK1, which is used as the HMD, has lenses that cause a pincushion distortion, this distortion needs to be counteracted by applying a barrel distortion to the generated image that is the exact opposite of the lens distortion, so that the image that goes into the user’s eyes is undistorted.

![Barrel and Pincushion Distortion](image)

Figure 2.1: Barrel- and Pincushion distortion

3. The FPGA should be able to transfer images from the cameras to the PC with at least 10 frames per second (fps) at 1280x800 pixels resolution.

   The native resolution of the Oculus Rift DK1 is 1280x800 pixels. Since the image to be transferred is in 24-bit colour, the estimated required bandwidth can be calculated using formula 2.1.
2.2. System requirements

\[ B = W \cdot H \cdot C \]  

(2.1)

\( B \) is the bandwidth in [bits/s], \( W \) and \( H \) are respectively the screen width and height in [pixels] and \( C \) is the color complexity in [bits].

Substituting with \( W = 1280, H = 800, C = 24 \) gives us a \( B \) of 24 Mbit/s.

Since the maximum bandwidth of USB 2.0 is 480 Mbit/s, 20 fps should be achievable. However, the maximum theoretical transfer speed of USB 2.0 is rarely achieved because of factors such as overhead, we set the requirement to half of that: 10 fps. This should still allow for decent image processing, since 10 fps is the bare minimum update rate required for the animation to still seem interactive to the human eye[13], while being easily transferable over USB 2.0, even with overhead taken into account.

4. The FPGA should be able to synchronize the start of an RGB sequence with the PC.

The FPGA should not just randomly start sending bytes. The available transfer protocols allows a maximum transfer of 8 bits per clock cycle (more on this in chapter 3), which is exactly one colour value of one pixel. This means one pixel will be sent over 3 clock cycles, so the PC needs to know which byte contains which colour. Furthermore, the PC needs to know which pixel is the first one of a frame. In other words, the transfer protocol needs to be synchronized.

2.2. System requirements

In order to achieve the functional requirements, this platform should fulfill the following requirements:

1. The PC should be equipped with decently powerful hardware.

In order to achieve the aforementioned barrel distortion, and of course to be able to create a decently-looking augmented layer, a relatively powerful PC is needed, such that an output of at least 60 fps to the Oculus Rift can be achieved. The most important parts of the PC for this calculation are the GPU (Graphics Processing Unit) and the CPU (Central Processing Unit). The GPU is needed for parallel graphically intensive calculations, such as the barrel distortion and the rendering of 3D objects, while the CPU is needed for fast serial calculations, such as building a frame from the received FPGA data and running the framework scripts.

2. The PC should be able to run Unity3D[14]-software.

All of the programming is made to run in Unity3D, so the PC needs to be able to run that software. This is not really that much of a problem, since almost all PC's support this software.

3. The PC should have a USB 2.0 or higher port.

4. The PC should be able to map the FPGA datastream to a virtual COM-port.

An important requirement is that the FPGA datastream should be accessible from C#, Unity3D's main programming language. The easiest way to get data there is to emulate a COM-port and pull data from that port, since C# has excellent support for COM-ports.

5. The PC should have an HDMI 1.0 or higher output port.

6. The FPGA should have a USB 2.0 compatible host controller and a USB port.

7. The FPGA should have and HDMI 1.0 or higher in- and output.

8. The FPGA should have an internal clock speed of at least 60MHz.

Since, as mentioned before, is is possible to transfer 8 bits, or 1 byte, per clock cycle, we need at least a 60 MHz clock to drive the USB controller or we will not be able to achieve the theoretical transfer speed of 480 Mbit/s.
2.3. Development requirements

1. The PC-side code should be written so that the data processing can be done in C#, which is the main programming language in Unity3D.

The final code that will run on the PC will be programmed in C#, extended with the Unity3D API. However, intermittent languages are of course acceptable. In chapter 4.1.1, for example, a script written in C/C++ is explained, which is also necessary. That, however, is acceptable because a wrapper of that code can be made and imported into C#.

2. The FPGA should be programmed in VHDL, because that is the HDL (Hardware Development Language) the team working on the entire system agreed upon because the team is already experienced in this language.

VHDL is used by all other subgroups participating in this project, and therefore our modules should also be written in VHDL so that the other groups can verify and study our project code, so they will not have to regard it as a black box with operation instructions.
The goal is to design a system which takes a video stream and applies some video processing to it and outputs the resulting video stream. The block design of this system can be seen in figure 3.1.

Starting from the HDMI receiver block on the FPGA side:

- The HDMI receiver receives the video stream from the PC via Unity3D
- Unity3D makes a request to the DLL, using C#, to obtain data
- Inside the DLL the API starts and performs the data transfer
- The camera block on the FPGA side prepares the data for transfer

![Figure 3.1: FPGA - PC block design](image)

The camera video stream from the FPGA has to be fed to the PC such that video processing can be applied. In order to achieve this, there has to be some communication between the FPGA and the PC for there to be data transfer. There are several ways to transfer data between the FPGA and PC, such as: Ethernet, UART and DSPI/DPTI.

A requirement of this system is to receive a video stream with a resolution of 1280 x 800 on the PC, ideally with a frame rate of 60 fps. Each pixel consists of three component intensities, namely red, green and blue.
(RGB). Additionally, each intensity ranges between the values 0 and 255, or 8 bits. The total bit rate required for such transfer is obtained using,

\[ R = W \cdot H \cdot f \cdot C \]  

(3.1)

Where \( W \) and \( H \) is the width and height, respectively, in pixels. \( f \) is the frame rate in fps. \( C \) is the colour depth in bits, in this case 24. Substituting the values into eq. 3.1 yields a bit rate of 1.47 Gbits/s.

### 3.1. Data Transfer Interfaces

The Nexys Video Artix-7 provides two interfaces in order to transfer data from and to the PC. Using the API provided by the Adept 2 SDK [15] data can be presented or requested from the FPGA. On the FPGA the incoming and outgoing data is passed to the on-board dual port FT2232H USB controller. The connection between the USB controller and FPGA can be seen in figure 3.2.

#### 3.1.1. Digilent Serial Peripheral Interface (DSPI)

The serial peripheral interface uses four data lines for serial full-duplex communication. The master is the FT2232H USB controller and the slave is the FPGA. The serial clock (SCK) data line provided by the PC determines the transfer speed of the data. The Master Out/Slave in (MOSI) data line transmits data from the PC to the FPGA via the USB controller. The Master In/Slave Out data line takes the data provided by the FPGA and transmits it to the PC via the USB controller. Lastly, the Slave Select (SS) data line enables the data transfer and is set by the PC.

An overview of the four data lines (signals) can be seen in table 3.1.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction (FPGA)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCK</td>
<td>Input</td>
<td>Serial clock</td>
</tr>
<tr>
<td>MOSI</td>
<td>Output</td>
<td>Data to the FPGA</td>
</tr>
<tr>
<td>MISO</td>
<td>Input</td>
<td>Data to the PC</td>
</tr>
<tr>
<td>SS</td>
<td>Input</td>
<td>Enabling data transfer</td>
</tr>
</tbody>
</table>

Table 3.1: DSPI signals
3.1.2. Digilent Parallel Transfer Interface (DPTI)
The Digilent Parallel Transfer Interface offers a higher bandwidth than DSPI. This is because it provides an 8-bit wide parallel FIFO-style data interface. For an in-depth explanation of DPTI refer to chapter 4.3.1.

3.2. Design Choices
Using the Nexys Video there are, as stated before, several ways to transfer data from the FPGA to the PC. One of them is via Ethernet, which has a maximum bandwidth of 1000 Mbit/s. It is, however, inherently unreliable; when the receiver receives packets it will either accept or discard them, without sending an acknowledgment to the sender notifying the status of the packets [17]. How unreliable is Ethernet? According to the Ethernet 802.3 standard, the raw bit error rate is about $10^{-10}$, which means a packet drop every 10 seconds is within the standard [18].

The next option is using a UART. This, however, poses an immediate problem - the maximum bandwidth of a UART is 9.6 kbit/s, which is well below the desired transfer rate.

Ultimately DSPI/DPTI is chosen for data transfer even though Ethernet has a higher bandwidth, because DSPI/DPTI is easier to implement. Since Digilent has released an SDK for the implementation of the two interfaces, it is a better choice, time-wise, to use DSPI/DPTI. One could argue that performance-wise it is lacking, however it is still enough to perform video processing. Since both interfaces, DSPI and DPTI, make use of USB 2.0 with a maximum bandwidth of 480 Mbit/s, transferring a video stream with a resolution of 1280 x 800 is only possible with a maximum frame rate of 20 fps. Using DPTI, however, the maximum transfer speed is about 40 MB/s or 320 Mbit/s. This yields a frame rate of approximately 13 fps.

Due to the unavailability of DPTI during the first half of the project (it was slated to be released at the beginning of June), DSPI was used to test the communication between the FPGA and PC.
4 Implementation

4.1. Data Transfer from FPGA to PC

4.1.1. DPTI API

The core of the data transfer is the Adept 2 SDK, which provides the API for performing data transfer. Data transfer from the FPGA to the PC is performed using the following functions, in order:

1. Open the device (FPGA) using the `DmgrOpen(HIF * phif, char * szSel)` function
2. Enable the specific port using the `DptiEnableEx(HIF hif, INT32 prtReq)` function
3. Get data from the FPGA using the `DptiIO(HIF hif, BYTE * pbOut, DWORD cbOut, BYTE * pbIn, DWORD cbIn, BOOL fOverlap)` function
4. Disable the specific port using the `DptiDisable(HIF hif)` function
5. Close the device (FPGA) using the `DmgrClose(HIF hif)` function

The preliminary testing of the data transfer is done in the command-line interface (CLI), which simply prints the values of the received bytes. Additionally, the transfer speed can be determined by keeping track of how much time has elapsed between requesting the data, an x amount of bytes, and actually receiving it. The resulting transfer speed is about 40 MB/s or 320 Mbit/s.

The API functions mentioned above are written in C/C++, which is fine when only using the CLI. Unity3D, however, uses a different language for scripting, namely C#. The C# code written for Unity3D cannot directly interact with C/C++ functions. In order to access these function from Unity3D, the API functions, including additional error/status checking functions, need to be compiled to a Dynamically Linked Library (DLL). Then, simply including the compiled DLL in the C# script allows the use of the desired API functions. The code of the DLL can be found in Appendix A.1.

4.1.2. Unity3D

In Unity3D the function which gets the data from the FPGA, `DptiIO(...)`, gets called once per frame and is inside the `Update()` function. After some additional checks the data, an array of bytes, is passed to the function `LoadRawTextureData(byte[] data)` and it transforms the array of bytes into an image. The first three indexes of the array correspond to the first pixel of the image, located at one of the corners of the image. Which corner it corresponds to depends on the orientation of the image. The second three indexes of the array correspond to the second pixel of the image, and so on. A simple example of the implementation of the `LoadRawTextureData(byte[] data)` function can be seen below.
Listing 4.1: Simple implementation of transforming a byte array to an image

```csharp
using UnityEngine;
using System.Collections;

public class simple_test : MonoBehaviour {
    public int WIDTH = 2;
    public int HEIGHT = 2;
    public Texture2D tex;

    void Start () {
        tex = new Texture2D(WIDTH, HEIGHT, TextureFormat.RGB24, false);

        byte[] data = new byte[] {0xFF, 0x00, 0x00, 0x00, 0xFF, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0x0FF};

        tex.LoadRawTextureData(data);
        tex.Apply();
        GetComponent<Renderer>().material.mainTexture = tex;
    }

    void Update () {
    }
}
```

The array of bytes (line 26-27) contains four colours, each made up of three bytes: red, green, blue and white. This array is then passed to the function (line 30) and is displayed as a 2 x 2 image. The resulting image can be seen in figure 4.1.

![Figure 4.1: A 2 x 2 image containing four colours](image)

From the image four colours can be observed, the first pixel of the image is located at the top-right. At
each new horizontal line the colours start from the right; this is due to the orientation of the plane the image is placed on.

### 4.1.3. Syncing Transmitter and Receiver

So far, we have described the basic principle of receiving the array of bytes and processing it to an image. One crucial step, however, has to be taken in order for the data transfer to work effectively, namely syncing. As explained before, the `LoadRawTextureData(byte[] data)` function expects the first byte to hold the value of R, second byte of B, third byte G, and so on. If, for example, the FPGA transmits one byte too late or too early, the rendered image will turn out to be much different than expected.

To solve this problem, there needs to be some way to sync the transmitter and receiver. The transmitter (FPGA) sends a byte sequence to the receiver (PC) and when recognized, the next byte sent is the value of R. The sequence being sent should be as unique as possible, such that the chance of the occurrence of the sequence is low. Based on chrome keying, blue (#0000FF) or green (#00FF00) are colours which not present in human flesh pigments [19]. The sequence chosen for testing is a 9-byte sequence: {0x00, 0xFF, 0x00, 0x00, 0xFF, 0x00, 0x00, 0xFF, 0x00}.

One way to implement the data transfer with syncing is to first receive one byte at a time, in order to find the sequence. After the sequence is found a frame array is fetched and transformed to an image, after which the sequence finder is ran again, and so on. This yields in a simple algorithm, therefore also a simpler code.

A problem, however, is the time it takes to find the first sequence. Since the sequence always is at the beginning of the frame, missing a sequence yields a maximum delay of one frame. In our case, the very first sequence cannot be found, this in turn means that the sequence finder checks a maximum of one frame worth of bytes. A frame equals to 1280·800·3 = 3072000 bytes. Since one byte is being received at a time when the sequence is being found, the first frame is obtained after approximately 78 seconds.

This, of course, is not desirable. Therefore, a more complex algorithm as described below is used to combat this problem.

![Figure 4.2: Byte array containing the initialization (purple), start sequence (red) and frames (green)](image)

In this implementation a large array is fetched and is examined for the presence of the synchronization sequence. When the sequence is found the next x bytes are copied to a separate frame array, which contains data for Unity3D to transform into an image. To illustrate this, the algorithm for obtaining the frame arrays using the array in figure 4.2 is as follows,

1. Fetch a large array of bytes: in this case an array of 20 bytes
2. Find the end of the sequence in the array, starting from index 0: end of sequence is at index 3

3. When found, check if there is enough bytes left for a full frame: 19 - 3 = 16 bytes left (enough)

4. Enough bytes left? The next \(x\) bytes are copied to a frame array: bytes from index 4 to 15 are copied to a frame array

5. Check if there is enough bytes left for a sequence: 19 - 15 > 3 (enough)

6. Enough bytes left? Find the end of the sequence: end of sequence is at index 18

7. Check if there are enough bytes left for a full frame: 19 - 18 = 1 byte left (not enough)

8. Copy the remaining byte(s) to the frame array: the byte at index 19 is copied to the frame array

9. Fetch a new large array of bytes

10. Fill the frame array with data from the new array until it's full

11. Back to step 2 and continue

If at any point there are not enough bytes left in the fetched array for a sequence or frame, an additional array of bytes is fetched.

Figure 4.3: Transfer speed against the chunk size on a logarithmic scale
4.1.4. Transfer Speed
Fetching a large array of bytes and then searching for the sequence instead of receiving one byte at a time to find the sequence poses a question. *What size does the large array need to be to obtain the fastest data transfer?* In figure 4.3 the transfer speed is plot against the size of the array, chunk size. From the logarithmic plot can be observed that the maximum transfer speed is for a chunk size of about $10^6$ bytes. To determine the optimal chunk size the measurements are performed for chunk sizes around $10^6$ bytes. The results can be seen in figure 4.4. From the plot can be observed that maximum transfer speed, which is 44 MB/s, is obtained when the chunk size is set to $1.1 \times 10^6$ bytes.

For clarification, in the example in chapter 4.1.3 the check if there is enough bytes left for a full frame (see step 3 of the example) can be omitted. Since a frame is 3072000 bytes, which is larger than the received array of $1.1 \times 10^6$ bytes; which means that a received array can never contain a full frame.

![Figure 4.4: Transfer speed against the chunk size on a linear scale](image)
4.2. Video Processing

In Unity3D the received array of bytes is turned into an image every update cycle. This results in a video, where processing can be applied to.

4.2.1. Hybrid-Infrared Vision

The video input from the cameras are fed into Unity3D where the brightness level is analyzed. Areas of the video which have a brightness below a certain threshold are masked. In figure 4.5 this process is illustrated. The mask needs to be recognizable for the FPGA such that it can be replaced by the image of the IR cameras, more information can be found in the sub-thesis on the Implementations on an FPGA [10]. In figure 4.6 the green mask can be seen applied to a video stream. Light coming from the left windows creates a shadow on the PC (left), Oculus Rift (middle) and a person sitting (behind). Additionally, some other objects are masked with a green overlay. The colour chosen for the mask is green #00FF00, due to reasons stated in subsection 4.1.3.
4.2.2. AR Overlay

In figure 4.7 the process of using an AR overlay is illustrated. First the barrel distorted video input is transformed to a normal video, by applying pin cushion distortion. Using the normal video as a reference, the overlay can be positioned. Using image processing a position on the video input can be tracked to, for example, keeping a AR cube fixed to a table when the user’s head is turned. This can also be done by reading the sensor data of the gyroscope, accelerometer, and magnetometer using the Oculus SDK [20]. Using these sensors the head movements can be tracked, which can be used to hold an AR object in fixed in a place. This, however, is beyond the scope of the project and belongs to the future works.

The desired object to be placed as an AR overlay needs to be isolated, meaning parts that do not belong to the object are coloured green #00FF00. An example of an AR overlay can be seen in figure 4.8. The 3D cube is isolated from the background by colouring the background green. Additionally, it can be seen that the cube from the left and right vision is barrel distorted. This is achieved by using a barrel distortion shader [21] in Unity3D.
4.3. Implementation FPGA

On the FPGA side, a VHDL design is needed that drives the onboard FT2232H USB controller of our FPGA. Furthermore, the RGB signal, which is 24 bits wide, needs to be split into 8-bit parts, so it can be sent per byte, which is what DPTI requires and also what makes it faster than DSPI.

The system consists of two major blocks: the "DPTI controller" and the "byte splitter". The DPTI controller is a design that provides the onboard USB controller with the signals it needs. The byte splitter splits the 24-bit RGB signal that should be sent for each pixel up into 3 bytes, so it fits through the 8-bit DPTI bus in 3 clock cycles and then sends it to the DPTI controller with the correct timing.

4.3.1. DPTI Controller

Overview

The DPTI controller is the block that drives the FT2232H USB controller, and its design is provided by Digilent Inc., the producer of our FPGA, as part of the Adept 2 SDK[15].

Only the ports of the controller that are used to output data to the PC will be used, since the PC will output its data to the FPGA via HDMI instead of USB. This means that its block diagram can be reduced to the following, if only the ports that will be used are shown:

The figure has been divided into two subsections: write control and USB control. Write control is the part that the byte splitter is attached to, while USB control is attached to the USB controller.
4. Implementation

Write Control
The *wr_clk* signal is used to provide a clock to the controller to which the write operation is synchronized. Each rising edge on the clock signal, a byte is written from the *data_in* bus to an internal FIFO (First-In First-Out) register inside the DPTI controller if the signal on the write enable (*wr_en*) port is high.

*wr_full* is a signal that indicates when the DPTI controller’s internal FIFO register is full, and no further data can be written until the USB controller reads it out. This signal is connected to the byte splitter so it knows when to pause and wait until the data it is outputting can be used again.

The reset signal is used to reset the device, and usually connected to a manual reset button on the FPGA. The FPGA uses active-low reset, which means that whenever the reset signal is low, it will reset. At startup, the FPGA also automatically sends a reset pulse to this signal so that the device is correctly initialized.

USB Control
The *clk_o* signal is the clock from the USB controller, used to synchronize behaviour with that controller. Each rising edge, a byte is sent from the internal FIFO to the USB controller via the *data_out* bus.

*rx_en* and *tx_en* are used to indicate whether data can be respectively read from or written to the USB controller. The DPTI controller only sends data when *tx_en* is high. *rx_en* is not used, but still attached so that the USB controller does not have any floating outputs.

*spi_en* is a reset signal, but this one is provided by the USB controller instead of manually pressing the reset button.

*rdn* and *wrn* are used to trigger the USB controller to respectively read out and write to the internal FIFO register of the DPTI controller. These are used in asynchronous DPTI, but we use the synchronous version, so these ports are connected but not used. The *oen* signal tells the USB controller whether the data bus is used for reading or writing. If it is low, the bus is used to read to the FPGA, when high it is used to write to the PC.

*siwun* is a signal with two functions: function one is *Send Immediately*. If the computer is operating normally, the USB controller sends a data packet to the PC immediately, even if the FIFO does not contain an entire packet yet. If the computer is in sleep mode, this signal can wake the computer. Both actions are triggered by a low pulse. These signals are all generated within the DPTI controller.

4.3.2. Byte Splitter
The byte splitter is a simpler block that provides the DPTI controller with the correct data to send when provided with pixel data in 24-bit RGB form. Its block diagram is as follows:

```
+-----------------+            +-----------------+
| RGB [23:0]      |            | byte_out [7:0] |
+-----------------+            +-----------------+
| clk             |            | wr_en           |
+-----------------+            +-----------------+
| reset           |            | next_px         |
+-----------------+            +-----------------+
| wr_full         |            |
```

Figure 4.11: Block diagram of the byte splitter

The behaviour of the byte splitter is as follows: Every rising edge of the clock (*clk*), the byte splitter puts a different section of the *RGB* bus on the *byte_out* bus. It starts with bits 0 to 7, then bits 8 to 15, and finally bits 16 to 23. When the third section is being put onto the *byte_out* bus, the byte splitter also sends a high pulse on the *next_px* signal, telling the block that provides the byte splitter with the pixel data (which is made by another subgroup) to put the next pixel on the *RGB* bus.

The byte splitter also checks if the FIFO register of the DPTI controller is not full, by only running while *wr_full* is low, meaning there is still space it can write the bytes to. Only then will the design cycle through the *RGB* sections, and only then will the *write_en* signal be high, which is connected to the DPTI controller’s *wr_en* signal.

The byte splitter also has another function: sending the synchronization sequence that indicates the beginning of a new frame. Since the only resolution used is 1280x800, the design can count until it has sent that
amount of pixels and then know that it has to send the sync sequence again, since the protocol is to synchronize every frame. The sync sequence consists of 3 [0,255,0] pixels, and after transferring that sequence, the byte splitter will resume its normal operation.

The reset signal resets the design whenever it is low and is also connected to the manual reset button. It receives a reset pulse at startup as well.
5.1. FPGA to PC transfer

The maximum transfer speed that was achieved, as can be seen in subsection 4.1.3, is 44 MB/s.
The Oculus Rift’s native resolution is 1280x800 pixels, and that in 24-bits colour. The possible framerate of the raw RGB video that is sent from the FPGA to the PC, can be calculated using formula 5.1, with \( S = 44 \times 10^6 \), \( W = 1280 \), \( H = 800 \), \( C = 24 \).

\[
F = \frac{S}{W \cdot H \cdot \frac{C}{8}}
\]  
(5.1)

Where \( F \) is the framerate in [frames/s], \( S \) the transfer speed in [bytes/s], \( W \) the screen width in [pixels], \( H \) the screen height in [pixels] and \( C \) the colour depth in [bits].

Putting the data that was just mentioned into this formula gives a framerate of

\[
F = \frac{44 \times 10^6}{1280 \cdot 800 \cdot \frac{24}{8}} = 14.32 \text{ fps}
\]  
(5.2)

Rounding this down gives a framerate of 14 fps.

If needed, the choice can be made to sacrifice some colour complexity, since the data is only used for processing anyway, and achieve even higher framerate. For example, if we shifted to 16-bits colour (\( C = 16 \)), we would be able to achieve 21 fps.

The most extreme version would be switching to greyscale, allowing for transfer of all data of one pixel in one byte (\( C = 8 \)), thus resulting in 42 fps.

Synchronization works well, and by searching chunks when looking for the synchronization sequence instead of sending individual bytes, we were able to significantly reduce the startup process to somewhere between 1 and 3 seconds, depending on how long it takes to find the first sync sequence.

5.2. PC to FPGA transfer

Using Unity3D, it is possible to replace individual pixels that need to be replaced by the camera image with a pure green color (RGB \([0,255,0]\)), as seen in figure 4.6.

Since the FPGA already sends an identification signal to the GPU, we do not need to force the PC to output the correct resolution; it does so by itself, and in the strange case that it will not automatically adjust the resolution, we can just set it to the correct one in the configuration panel of the operating system.

As seen in section 4.2, our program can undistort, process, mask, redistort and then send a video stream from the cameras to the FPGA, and indirectly the Oculus Rift, using the correct format, which is exactly what we needed, so we are satisfied with this result as well.
6.1. Conclusion

Our transfer system works well, and is able to synchronize correctly with the PC, to guarantee correct data transfer at speeds around 44MB/s. This is enough to provide the PC with a decent amount of frames per second that it can then use for calculation. We also have the option of sacrificing colour depth for faster frame rates.

This all is under the condition that the used protocol is DPTI, of course. The alternative, DSPI, was much slower, and even this faster protocol is just on the edge of usable, but that is as expected since our USB port is limited to version 2.0.

The PC itself can transfer the required image in the correct format to the FPGA over HDMI, using Unity3D to process the images.

All in all we are happy with our results, since they met all of our requirements stated in the programme of requirements.

6.2. Future works

The Oculus Rift DK1 we are currently using for this project is outdated technology. The amount of pixels is too low to make a decent product ready for the consumer market; the current product is merely a proof-of-concept.

If we want to move to VR HMD’s with higher resolutions and screen refresh rates, we will require much more bandwidth to transfer the video signal from FPGA to PC. This will force us to abandon USB 2.0 and move on to USB 3.0 (5 GB/s) or, preferably, USB 3.1, which is able to achieve transfer rates of 10GB/s.

Since USB 3.1 is currently quickly becoming mainstream in the PC world, and VR applications require a high-end PC anyway, we suspect that if we develop a newer version of our product using the USB 3.1-bus, the standard will be widespread enough among PC enthusiasts, the main consumers of VR, that we can use it for our system.

Additionally, on the PC side the software could be improved for image recognition to track an object. As mentioned in chapter 4.2.2, an AR object could interact with the environment, for example, keeping an AR cube fixed on a table when the user’s head is turned.
A.1. DPTI DLL

```c
#include "dptilib.h"

extern "C" {

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>
#include "dpcdecl.h"
#include "dmgr.h"
#include "dpti.h"

HIF hif = hifInvalid;
BYTE* pbOut = NULL;
BYTE* pbIn = NULL;
BYTE* pbInSeq = NULL;
INT32 prtReq;
DPRP dprpPti;
// DWORD cbRcv;
BYTE *pbInReturn = NULL;
BYTE pbInReturn2;
const DWORD cbTrans = 0;
const DWORD cbSeq = 1;
int seq_index = 0;

/* Attempt to open device */
int dpti_open () {
    if (!DmgrOpen(&hif, "NexysVideo")) {
        return 1;
    }
    else {
        return 0;
    }
}

/* Enable the specified DPTI port */
```

/* A DLL containing the DPTI API functions from the Digilent Adept 2 SDK
 * Website: https://reference.digilentinc.com/digilent_adept_2#software_downloads
 */
int dpti_enable(INT32 prtReq) {
    if (!DptiEnableEx(hif, prtReq)) {
        return 1;
    } else {
        return 0;
    }
}

/* Check if DPTI is synchronous or asynchronous */
int dpti_checktrans(INT32 prtReq) {
    /* Obtain the port properties associated with the specified DPTI port */
    DptiGetPortProperties(hif, prtReq, &dprpPti);
    if (dprpPtiAsynchronous & dprpPti) {
        return 1; //asynchronous DPTI port enabled
    } else {
        return 0; //synchronous DPTI port enabled
    }
}

int dpti_match(BYTE* sequence, int size) {
    pbInSeq = (BYTE*)calloc(cbSeq, sizeof(BYTE));
    if (pbInSeq == NULL) {
        return NULL;
    }
    int seq_index = 0;
    while (1) {
        if (!DptiIO(hif, pbOut, cbTrans, pbInSeq, cbSeq, fFalse)) {
            free(pbInSeq);
            return 1;
        } else {
            /* Match fetched byte with index seq_index of the sequence */
            if (*pbInSeq == *(sequence + seq_index)) {
                if (seq_index < size) {
                    seq_index += 1;
                } else {
                    free(pbInSeq);
                    int seq_index = 0;
                    return 0;
                }
            } else {
                seq_index = 0;
            }
        }
    }
}

/* Perform data fetching and put into array*/
BYTE *dpti_fetch(unsigned long cbRcv) {
    /* Allocate memory */
    pbIn = (BYTE*)calloc(cbRcv, sizeof(BYTE));
    if (pbIn == NULL) {
        return NULL;
    }
    if (!DptiIO(hif, pbOut, cbTrans, pbIn, cbRcv, fFalse)) {
        free(pbIn);
    }
return NULL;
}
else {
    /* Allocate memory for returning data */
    pbInReturn = (BYTE*) CoTaskMemAlloc(cbRcv);
    if (pbInReturn == NULL) {
        free(pbIn);
        return NULL;
    }
    else {
        memcpy(pbInReturn, pbIn, cbRcv);
        free(pbIn);
        return pbInReturn;
    }
}
}

/* Perform data fetching */
BYTE dpti_fetch2(unsigned long cbRcv) {
    /* Allocate memory */
    pbIn = (BYTE*) calloc(cbRcv, sizeof(BYTE));
    if (pbIn == NULL) {
        return NULL;
    }
    if (! DptiIO(hif, pbOut, cbTrans, pbIn, cbRcv, fFalse)) {
        free(pbIn);
        return NULL;
    } else {
        pbInReturn2 = *pbIn;
        free(pbIn);
        return pbInReturn2;
    }
}

/* Disable the DPTI port */
int dpti_disable() {
    if (! DptiDisable(hif)) {
        return 1;
    } else {
        return 0;
    }
}

/* Close the device handle */
int dpti_close() {
    if (! DmgrClose(hif)) {
        return 1;
    } else {
        return 0;
    }
}
A.2. DPTI Controller

-- Company: Digilent Inc.
-- Engineer:
--
-- Create Date: 16:16:24 10/26/2011
-- Design Name: dpti_ctrl - Behavioral
-- Module Name: dpti_ctrl - Behavioral
-- Project Name: 
-- Target Devices: 
-- Tool versions: 
-- Description: This module implements a synchronous DPTI interface. It includes 
-- two local FIFOs that allow the data to be moved from the DPTI clock domain 
-- to another clock domain in the FPGA.
--
-- Dependencies: 
--
-- Revision: 
-- Revision 0.01 - File Created
-- Additional Comments: 
-- 5/30/2016(sjb) --- Prepared for public release

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using 
-- arithmetic functions with Signed or Unsigned values 
-- use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating 
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

Library UNIMACRO;
use UNIMACRO.vcomponents.all;

entity dpti_ctrl is
  Port ( 
    --User Write FIFO signals 
    wr_clk : in std_logic;
    wr_en : in std_logic;
    wr_full : out std_logic;
    wr_afull : out std_logic;
    wr_err : out std_logic;
    wr_count : out std_logic_vector(11 downto 0);
    wr_di : in std_logic_vector(7 downto 0);
    --User Read FIFO signals 
    rd_clk : in std_logic;
    rd_en : in std_logic;
    rd_empty : out std_logic;
    rd_aempty : out std_logic;
    rd_err : out std_logic;
    rd_count : out std_logic_vector(11 downto 0);
    rd_do : out std_logic_vector(7 downto 0);
    --misc. signals 
    rst : in std_logic; 
    --Asynchronously resets the entire component. Must be held high for at least 100ns, or 6 clock cycles of the slowest fifo clock
    --DPTI Port signals
    prog_clko : in STD_LOGIC;
    prog_txen : in STD_LOGIC;
    prog_spien : in STD_LOGIC; 
    --called jtagen on some platforms
    prog_rdn : out STD_LOGIC;
    prog_wr : out STD_LOGIC;
    prog_oen : out STD_LOGIC;
    prog_siwun : out STD_LOGIC;
    prog_d : inout STD_LOGIC_VECTOR(7 downto 0));
end dpti_ctrl;

architecture Behavioral of dpti_ctrl is

-- Component Declarations

component clk_wiz_0 is
  port ( 
    CLK_IN1 : in std_logic;
    
  )

end component;
CLK_OUT1 : out std_logic;
CLK_OUT2 : out std_logic;
LOCKED : out std_logic;
end component;

-- Local Type Declarations

-- Constant Declarations

-- The following constants define state codes for the Synchronous PTI port state machine. The high order bits of the state number provide a unique state identifier for each state. The low order bits are the state machine outputs for that state. This type of state machine implementation uses no combinational logic to generate the outputs, which should result in glitch-free outputs.

constant stPtiRdy : std_logic_vector(6 downto 0) := "0" & "00" & "1111";
constant stPtiInOut0 : std_logic_vector(6 downto 0) := "1" & "00" & "1111";
constant stPtiInOut1 : std_logic_vector(6 downto 0) := "1" & "00" & "1011";
constant stPtiInOut2 : std_logic_vector(6 downto 0) := "1" & "01" & "1010";
constant stPtiInOut3 : std_logic_vector(6 downto 0) := "1" & "10" & "0101";

-- Signal Declarations

signal stPtiCur : std_logic_vector(6 downto 0) := stPtiRdy;
signal stPtiNext : std_logic_vector(6 downto 0);
signal clkPti : std_logic;
signal clkIOB : std_logic;
signal locked : std_logic;

-- user fifo signals
signal usrWrClk : std_logic;
signal usrWrEn : std_logic;
signal usrFull : std_logic;
signal usrAFull : std_logic;
signal usrWrErr : std_logic;
signal usrRdClk : std_logic;
signal usrRdEn : std_logic;
signal usrEmpty : std_logic;
signal usrAEmpty : std_logic;
signal usrRdErr : std_logic;
signal usrRdCnt : std_logic_vector(11 downto 0);
signal usrDO : std_logic_vector(7 downto 0);
signal usrWrCnt : std_logic_vector(11 downto 0);
signal usrDI : std_logic_vector(7 downto 0);

-- Internal control signals
signal ctlRd : std_logic;
signal ctlWr : std_logic;
signal ctlOe : std_logic;
signal ctlDir : std_logic;
signal ctlRxf : std_logic;
signal ctlTxe : std_logic;
signal ctlRst : std_logic;
signal ctlFull : std_logic;
signal ctlAFull : std_logic;
signal ctlEmpty : std_logic;
signal ctlFwr : std_logic;
signal ctlFrd : std_logic;
signal dummyWrCnt : std_logic_vector(11 downto 0);
signal dummyRdCnt : std_logic_vector(11 downto 0);
signal busPtIN : std_logic_vector(7 downto 0);
signal busPtINReg : std_logic_vector(7 downto 0);
signal busPtOUT : std_logic_vector(7 downto 0);

-- Module Implementation
A.2. DPTI Controller

begin
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

This clocking wizard is used to synchronize the clock used throughout this design
−with the proclkio input. It reduces delays introduced by the global clocking
−network (~4ns in post-route simulation on Artix 200t). It is also used to generate
−a phase shifted clock for latching input data/flags

clkwiz.inst : clk_wiz_0
port map
( CLK_IN1 => proclkio,
  CLK_OUT1 => clkIOB,
  CLK_OUT2 => clkPti,
  LOCKED => locked);
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

This process latches the inputs from the FTDI part at IOB flip-flops. They are latched
−using a phaseshifted clock about ~11ns after the rising edge of the input clock. This
−makes meeting the input delay timing constraints possible. Additional constraints for
−traveling from the clkIOB domain to clkPti are automatically inferred by the tools
−because they have a known phase relationship (signals will need to settle in ~<5ns).
−If the design fails interclock timing analysis with clkIOB, first ensure that IOB flip
−flops are being instantiated for the registers in this process. Then you can try reducing
−the phase shift by a few degrees (too much will cause the input delay constraints to fail). Last
−resort is to modify the design so that these flags are all immediately latched to the clkPti
−domain.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

@sjb: I recently came across this code and began questioning if it is necessary. At first thought, it seems
−that latching the input signals would add to needed setup times vs. just letting them thru
−asynchronously. It took me awhile, but I believe I understand why this is needed, so I have decided
−to document my reasoning further for the next time someone comes across this.
−I believe I inserted these because the FTDI part only holds the data on the bus valid for 1 ns
−after a proclk edge, and I was concerned that the databus/ctrl signals might
−beat the clock due to delays introduced by the locking infrastructure. The IOBs treat this situation
−by ensuring the data/ctrl gets latched internally the moment it is guaranteed to be valid. This seems like
−the most architecture independent way to treat the problem because it only requires three things, all of which
−are very likely to be present in future FPGA architectures/devices:
− 1) Ability to generate a clock with minimum input delay (~<3ns)
− 2) IOB flip-flops with minimum setup requirement (~<3ns)
− 3) Ability to generate a clock with a 230 degree phase shift relative to the generated clock with minimum input delay.
−Last note, I think the input_delay constraints with the --min option in timing xdc describe this requirement
−to the tools. I'm pretty sure removing these flip-flops will cause the design to fail meeting those constraints.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

begin
if (ctlRst = '1') then
  busPtiInReg <= (others => '0');
  ctlRxf <= '0';
  ctlTxe <= '0';
elsif (rising_edge(clkIOB)) then
  busPtiInReg <= busPtiIn;
  ctlRxf <= prog_rxen;
  ctlTxe <= prog_txen;
end if;
end process;
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

−− Map basic status and control signals
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
--- PTI State Machine

--- Map glitch-less control signals from the current state
ctlRd <= stPtiCur(0); -- keep in mind: active low
ctlWr <= stPtiCur(1) or ctlEmpty; -- keep in mind: active low
ctlSe <= stPtiCur(2);
ctlDir <= stPtiCur(3);

--- Map additional control signals from the current state
ctlFwr <= stPtiCur(4) and not ctlRxf;
ctlFrd <= stPtiCur(5) and ( (not ctlTxe) and (not ctlEmpty));

--- This process moves the state machine to the next state on each clock.
process (clkPti, ctlRst)
begin
  if ctlRst = '1' then
    stPtiCur <= stPtiRdy;
  elsif clkPti = '1' and clkPti' Event then
    stPtiCur <= stPtiNext;
  end if;
end process;

--- This process determines the next state based on the current state and the
--- state machine inputs.
--- TODO: It appears the bandwidth is very limited when the FTDI->FPGA FPGA side FIFO
--- gets almost filled (4080 out of 4096). Almost full seems to take a very long time
--- to deassert, regardless of how fast the user is reading data. This causes the state machine
--- to continuously receive a single byte at a time (1/3 bandwidth) until the FIFO is full, or until
--- the FIFO has been read by the user past the almost full point and no data has been available on
--- the DPTI bus for a long time (~50 us). Need to fix this because it will lower bandwidth of designs that involve
--- continuously sending large amounts of data to the FPGA.
process (stPtiCur, stPtiNext, ctlRxf, ctlTxe, ctlFull, ctIAFul, ctlEmpty)
begin
  case stPtiCur is
    when stPtiRdy =>
      stPtiNext <= stPtiInOut0;
    when stPtiInOut0 =>
      -- Start any pending transactions
      if ctlRxf = '0' and ctlFull = '0' then --Receive data from FTDI (gets priority)
        stPtiNext <= stPtiIn0;
      elsif ctlTxe = '0' and ctlEmpty = '0' then --send data to FTDI
        stPtiNext <= stPtiOut3;
      else --no transaction yet
        stPtiNext <= stPtiIn0;
      end if;
    when stPtiInOut1 =>
      -- Initiate read transaction (FTDI->FPGA)
      stPtiNext <= stPtiIn0;
    when stPtiInOut2 =>
      -- Continue Read transaction
      if ctlRxf = '0' and ctIAFul = '0' then
        stPtiNext <= stPtiIn0;
      else --if no data available, or FPGA-side FIFO is almost full, stop transaction
        stPtiNext <= stPtiOut3;
      end if;
    when stPtiInOut3 =>
      -- Start/Continue write transaction (FPGA->FTDI)
      if ctlTxe = '0' and ctlEmpty = '0' then --note almost empty is used because the FIFO is first-word fall thru
        stPtiNext <= stPtiIn0;
      else --if FTDI FIFO is full, or we are on the last byte being sent, stop transaction
        stPtiNext <= stPtiOut3;
      end if;
    when others =>
      stPtiNext <= stPtiRdy;
  end case;
end process;

--- Input/Output FIFOS for data

--- Output FIFOS
usr2dpti_fifo : FIFO_DUALCLOCK_MACRO
generic map |
DEVICE => "7SERIES", -- Target Device: "VIRTEx5", "VIRTEx6", "7SERIES"
ALMOST_FULL_OFFSET => X"0FF0", -- Sets almost full threshold
ALMOST_EMPTY_OFFSET => X"000F", -- Sets the almost empty threshold
DATA_WIDTH => 8, -- Valid values are 1-72 (37-72 only valid when FIFO_SIZE="36Kb")
FIFO_SIZE => "36Kb", -- Target BRAM, "18Kb" or "36Kb"
FIRST_WORD_FALL_THROUGH => TRUE -- Sets the FIFO FWFT to TRUE or FALSE
port map |
ALMOSTEMPTY => open, -- 1-bit output almost empty
A.2. DPTI Controller

ALMOSTFULL => usrAFull, -- 1-bit output almost full
DO => busPtiOut, -- Output data, width defined by DATA_WIDTH parameter
EMPTY => ctlEmpty, -- 1-bit output empty
FULL => usrFull, -- 1-bit output full
RD_COUNT => dummyRdCnt, -- Output read count, width determined by FIFO depth
RDERR => open, -- 1-bit output read error
WR_COUNT => dummyWrCnt, -- Output write count, width determined by FIFO depth
WEERR => usrWrErr, -- 1-bit output write error
DI => usrDI, -- Input data, width defined by DATA_WIDTH parameter
RDCLK => clkPti, -- 1-bit input read clock
RDEN => ctlFrd, -- 1-bit input read enable
RST => ctlRst, -- 1-bit input reset
WRCLK => usrWrClk, -- 1-bit input write clock
WHEN => usrWrEn, -- 1-bit input write enable
);

usrWrClk <= wr_clk;
usrWrEn <= wr_en;
wr_full <= usrFull;
wr_afull <= usrAFull;
wr_err <= usrWrErr;
wr_count <= usrWrCnt;
usrDI <= wr_di;

-- Input Fifo
dpti2usr_fifo : FIFO_DUALCLOCK_MACRO
generic map ()
  DEVICE => "7SERIES", -- Target Device: "VIRTEX5", "VIRTEX6", "7SERIES"
  ALMOSTFULL_OFFSET => X"0FF0", -- Sets almost full threshold
  ALMOSTEMPTY_OFFSET => X"000F", -- Sets the almost empty threshold
  DATA_WIDTH => 8, -- Valid values are 1-72 (37-72 only valid when FIFO_SIZE="36Kb"
  FIFO_SIZE => "36Kb", -- Target BRAM, "18Kb" or "36Kb"
  FIRST_WORD_FALL_THROUGH => TRUE) -- Sets the FIFO PAVT to TRUE or FALSE
port map ()

ALMOSTEMPTY => usrAEmpty, -- 1-bit output almost empty
ALMOSTFULL => ctlAFull, -- 1-bit output almost full
DO => usrDO, -- Output data, width defined by DATA_WIDTH parameter
EMPTY => usrEmpty, -- 1-bit output empty
FULL => ctlFull, -- 1-bit output full
RD_COUNT => usrRdCnt, -- Output read count, width determined by FIFO depth
RDERR => usrRdErr, -- 1-bit output read error
WR_COUNT => dummyWrCnt, -- Output write count, width determined by FIFO depth
WEERR => open, -- 1-bit output write error
DI => busPtiInReg, -- Input data, width defined by DATA_WIDTH parameter
RDCLK => usrRdClk, -- 1-bit input read clock
RDEN => usrRdEn, -- 1-bit input read enable
RST => ctlRst, -- 1-bit input reset
WRCLK => clkPti, -- 1-bit input write clock
WHEN => ctlWr, -- 1-bit input write enable
);

usrRdClk <= rd_clk;
usrRdEn <= rd_en;
rd_empty <= usrEmpty;
rd_aempty <= usrAEmpty;
rd_err <= usrRdErr;
rd_count <= usrRdCnt;
rd_do <= usrDO;
end Behavioral;
A.3. Byte Splitter

---
company: [company]
engineer: [engineer]
create date: [08.06.2016 11:43:30]
design name: byte_sender - Behavioral
module name: byte_sender
project name: [project]
target devices: [target devices]
tool versions: [tool versions]
description: [description]
dependencies: [dependencies]
revision: [revision 0.01]
revision 0.01 - file created
additional comments: [additional comments]
---

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity byte_sender is
  Port ( clk : in STD_LOGIC;
         reset : in STD_LOGIC;
         rgb : in STD_LOGIC_VECTOR (23 downto 0);
         wr_full : in STD_LOGIC;
         next_px : out STD_LOGIC;
         write_en : out STD_LOGIC;
         byte_o : out STD_LOGIC_VECTOR (7 downto 0)) ;
end byte_sender;

architecture Behavioral of byte_sender is

  signal count : integer range 0 to 3 := 0;
  signal init_count : integer range 0 to 10 := 1;
  signal px_count : integer range 0 to 1280 := 0;
  signal h_count : integer range 0 to 720 := 0;

begin

  process (clk)
  begin

    if (rising_edge(clk)) then
      if (reset = '0') then
        count <= 0;
        init_count <= 1;
        px_count <= 0;
        h_count <= 0;
        byte_o <= (others => '0');
        write_en <= '0';
        next_px <= '0';
      else
        if (init_count <= 9) then
          if ((init_count mod 3) = 2) then
            byte_o <= "11111111";
          else
            byte_o <= "00000000";
          end if;
          init_count <= init_count + 1;
        next_px <= '0';
        write_en <= '1';
        count <= 0;
      elsif (wr_full = '0') then
        case count is
          when 0 => -- send R
            byte_o <= rgb(23 downto 16);
          next_px <= '0';
        end case;
      end if;
    end if;
  end process;

end Behavioral;
when 1 => — send G
  byte_o <= rgb(15 downto 8);
  next_px <= '0';
  count <= 2;
when 2 => — send H
  byte_o <= rgb(7 downto 0);
  next_px <= '0';
  if (px_count = 1279) then — HRes
    if (h_count = 719) then — VRes
      init_count <= 1;
      h_count <= 0;
      next_px <= '1';
    else
      h_count <= h_count + 1;
      next_px <= '0';
    end if;
    px_count <= 0;
  else
    px_count <= px_count + 1;
    next_px <= '0';
  end if;
count <= 0;
when others => — This should never happen
  byte_o <= (others => '0');
  count <= 0;
  next_px <= '0';
end case:
write_en <= '1';
else
  write_en <= '0';
  byte_o <= (others => '0');
  next_px <= '0';
end if;
end process;
end Behavioral;


