Project Report

Digital Network Camera

by
Gao Haifeng
Li Le
Farid Rosli
Marco Taubmann

Applied Embedded Systems Project WS 2012/13
Architektur eingebetteter Systeme
Institut für Informatik und Mikroelektoronik
Fak.IV-Elektrotechnik und Informatik
Technische Universität Berlin

March 5, 2013

Abstract

The Digital Network Camera Project was implemented during the course Applied Embedded Systems Project WS 2012/13 at TU Berlin. The target of our project was to capture a live image stream with an embedded digital camera device (client) and send them via network to a server which shows this live stream in a GUI. The client is implemented on a FPGA-Development board while the server is implemented in software on a personal computer. This report describes our system design, which components we used and how we implemented the functionality.

Contents

1 Introduction

The goal of this project is to develop a digital camera that should have some basic functionalities. Figure 1 below shows the system overview that has to be implemented. It shows three basic components that are connected to each other via cable and connector to realize the data transfer between them. In this case, the data bytes will originate from the camera module and propagate via the Microblaze client to a PC, which acts as a server. The PC has the capability to display the received data bytes. The camera module which is attached to the FPGA board can be configured by the Microblaze via the I2C bus so that the camera will provide the desired image format.

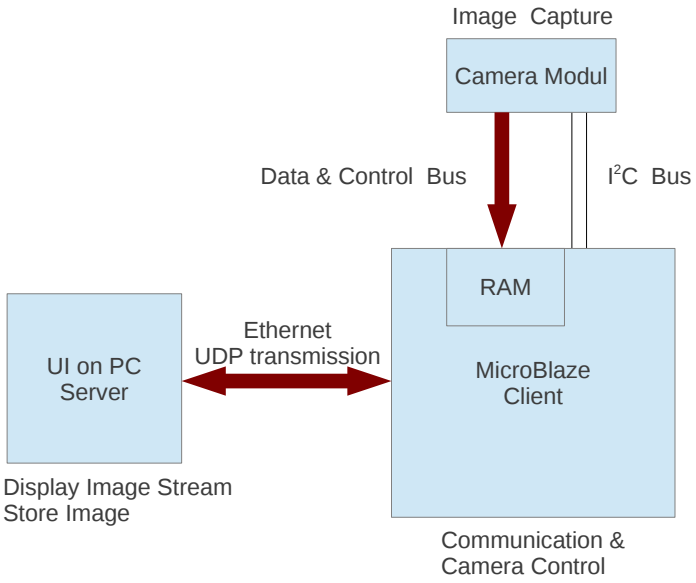

Figure 1: Overview of our system.

2 Background

2.1 Project description

In this project, a digital camera should be implemented by using the provided hardware and development board. The digital camera has to be capable of capturing and storing images. The first task is to connect a sensor board to an FPGA development board. By implementing a self-developed algorithm, the FPGA board must then be able to stream images from the camera sensor board and send them to a computer to be displayed. Therefore a small program has to be developed in Java which runs on a computer to receive images from an FPGA board and display them.

2.2 Development environments

Listed below are the hardware and software development environment used for the project:

1. **Xilinx Spartan 3E Starter Development Board**
   The Spartan-3E FPGA Starter Kit is a complete development board solution. The board has Microblaze processor that controls all other peripherals.
2. **VMODCAM MT9D112 2-Megapixel CMOS Digital Sensors**
   The module is equipped with two camera sensors. It provides data and control interface to the host system through VHDC connector. The data bus is 8-bits wide and the data transfer is controlled using three signals, pixel-clock signal, end-of-line signal and end-of-frame signal. A new clock pulse is introduced with each pixel. Each camera each sensor is controlled and configured via a two-wire serial interface called IIC, which is a generic master/slave based interface that enables low rate data communication between a master device and a peripheral.
3. **Xilinx ISE, EDK and SDK**
   These tools are used to realize the designed architecture that should be implemented on the board. The EDK is important to select peripherals that are necessary for this project such as DRAM and I2C controller. The address and connection between the peripherals via PLB bus can also be defined in EDK. Custom IP cores are programmed in VHDL using the ISE. The SDK is used to program the Microblaze processor.
4. **Eclipse for Java**
   Eclipse is used in this project to develop a Java program that runs on a computer to receive and display images from the FPGA development board.

3 Design and Implementation

The system consists of a server which receives images from the client and presents them in a user interface. The server is represented by a PC. The embedded digital camera device represents the client and is implemented on the FPGA-board. The client captures images with the camera and sends them to the server by using the ITP Protocol we designed for that purpose. Figure 2 shows this concept.

Section 3.1 describes the ITP Protocol which we use for the communication between client and server. Section 3.2 shows how we implemented the client which produces the images and how he utilizes the ITP Protocol to send the image stream over network. With this knowledge section 3.3 describes the implementation of the server that receives this stream and creates the output to the GUI.

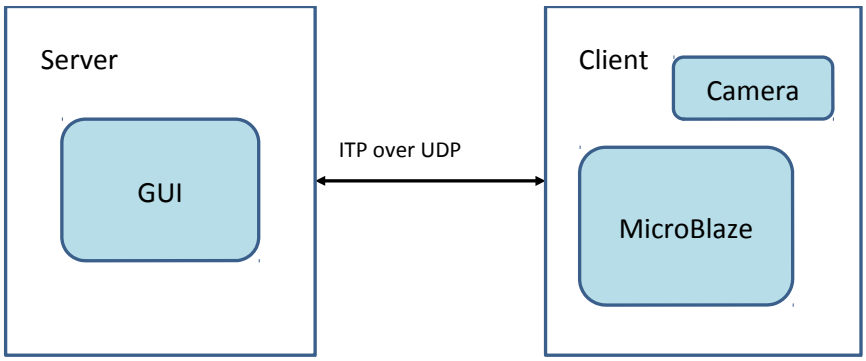

Figure 2: Client Server Structure.

3.1 Image Transfer Protocol (ITP)

To send an image stream from the client, which produces the images, to the server, which shows them in a user interface, we designed the Image Transfer Protocol (ITP). This protocol is simple structured and runs on top of UDP, which itself runs on top of the Internet Protocol, which in our application uses MAC to deliver the packets. That means we use single ITP packages as the UDP Payload.

The maximum payload for MAC is set to 1500 Byte. That means for our application we can use 1480 Byte maximum for one ITP packet, because the IP header and UDP header require 20 Byte together. The maximum lengths and the structure of on MAC frame is given in figure 3.

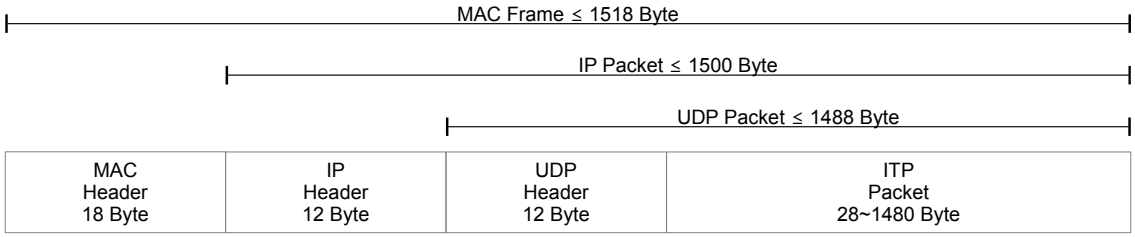

Figure 3: MAC frame structure.

The images in our stream have the dimensions of width $w = 640$ and height $h = 480$ pixel. Each pixel is represented in the *RGB565 format* (see 3.2.3 Camera Interface) which requires $s_p = 2\text{Byte}$ per pixel. This means the size of an image is given by:

$$\begin{aligned} s_{image} &= w \cdot h \cdot s_p \\ &= 640 \cdot 480 \cdot 2\text{Byte} \\ &= 614400\text{Byte} \\ &\approx 614k\text{Byte} \end{aligned} \tag{1}$$

Therefore it is not possible to send one image in one ITP packet over the network. One image must be split into several parts that each fit into an ITP packet. The ITP protocol therefore must be able to signal which parts belong to the same image. Furthermore we want the protocol to be able to adapt to different image sizes and keep it simple. These considerations result in the ITP packet structure given in figure 4.

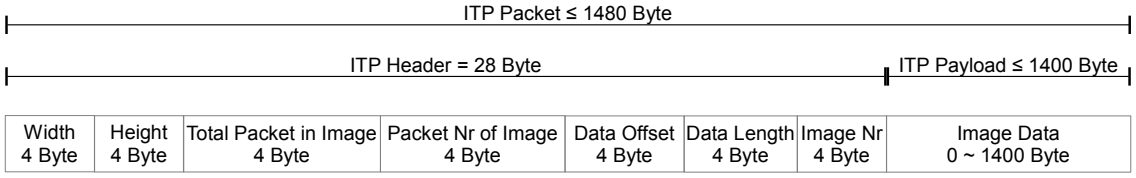

Figure 4: ITP packet structure.

The header always contains the *Width* and the *Height* of the image the current part belongs to. It also tells the number of *Total Packets in Image* as well as the part that the current packet contains in *Packet Nr of Image*. The *Data Offset* field tells the address offset of the current data in the image. *Data Length* contains how long the data is. This field is especially useful for the last packet of one image. Since the image stream is continuous, the *Image Nr* identifies the image in the stream this packet belongs to. All fields in the header have a size of 4 Byte and are in *unsigned integer* format. Behind the header the ITP payload follows. It contains the the image data of the specified part and is limited to a length of 1400 Byte.

By choosing this Packet format, there is no need for negotiating parameters over this protocol. The client continuously sends Images to the server in the specified format by using several ITP packets for one Image.

3.2 Client Implementation

The client partly runs in the FPGA of the Spartan board. The FPGA beside others contains a Microblaze processor, an ethernet controller, an I2C controller, an RAM controller, and custom core we created on our own: camera interface module. We used the Xilinx XPS to implement the client hardware (see figure 5).

The hardware architecture that is used for the implementation of the client is shown in Figure 6. A PC is connected to the system board via the Ethernet interface, which is bound to a PLB bus. The PLB bus is the medium used by the Microblaze and other peripherals to exchange data with each other. As the PLB master, Microblaze has to initiate the communication via this PLB bus. In order to configure the external camera, Microblaze will send the configuration data via I2C controller. The camera interface module is a custom IP core which is designed to fetch image data from the camera. Additionally there is another PLB bus that is used by the camera interface module to

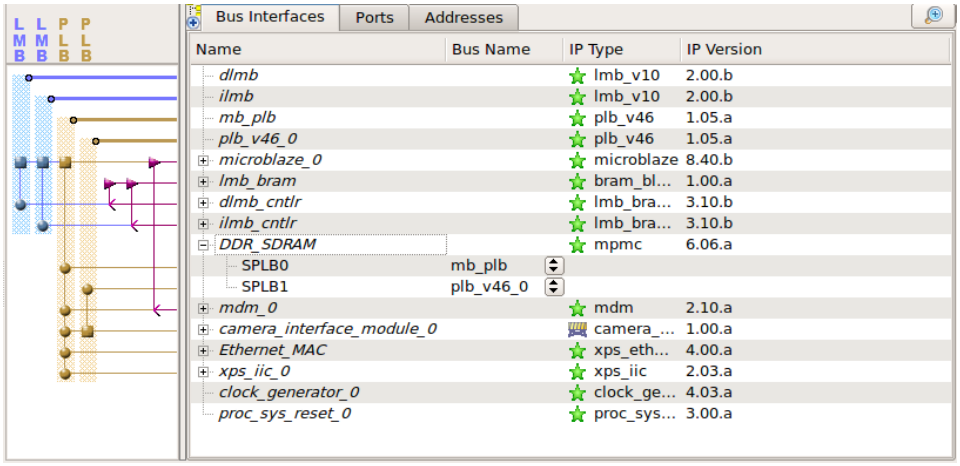

Figure 5: XPS implementation.

store the image bytes in RAM. The RAM has two ports so that the Microblaze can access the image data in parallel and send the data over network.

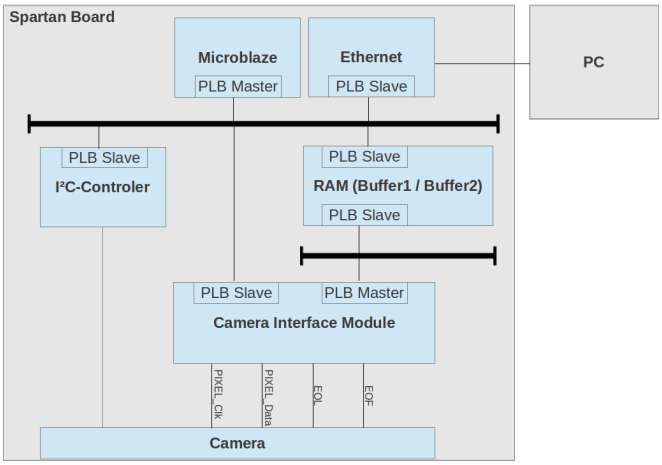

Figure 6: Client Implementation Overview.

To initiate the data transfer from the camera to the system board, Microblaze will send a start signal to the camera interface module. When the interface recognizes the signal, it will start receiving bytes from the camera. 4 bytes of pixel data will then be stored in the RAM at one time. When a buffer is fully written, the interface modul will signalize the microblaze by writing a ready value in its slave register. Microblaze will recognize by reading this slave register. It will fetch the pixel bytes written in the buffer and send them to the PC via Ethernet. At the same time, the interface will begin to write in another buffer. This procedure will be repeated when the other buffer is fully written. The camera interface module will be explained in more detail in the next section.

3.2.1 Camera Interface Module

The user logic file of the camera interface module is customized so that it will start collecting bytes from the camera upon receiving a so-called "go" signal from the Microblaze. Another VHDL module called data deserializer within the user logic is responsible for collecting pixel data. Its state machine is shown in figure 10. This module receives 8 bits of pixel data in each pixel clock cycle. Since the width of the PLB bus is 4 bytes or 32 bits, the module needs to accumulate the pixel bytes internally until it has all 32 bit data. This module will send those data bit to the RAM according to the base address received from the Microblaze. At this time, the data deserializer signalizes the user logic state machine by setting the valid signal. This state machine (figure 8) is responsible for writing all the data bits on the PLB bus to the target address of the RAM. The target address will be updated during the next writing process. Data deserializer will also signalize the user logic state machine if a frame is completed by setting frame finished signal to 1. It does this when EOF from the camera is set to zero. A clock generator is necessary to drive the camera.

For development and testing purposes we added an image generator module to replace the inputs from the camera.

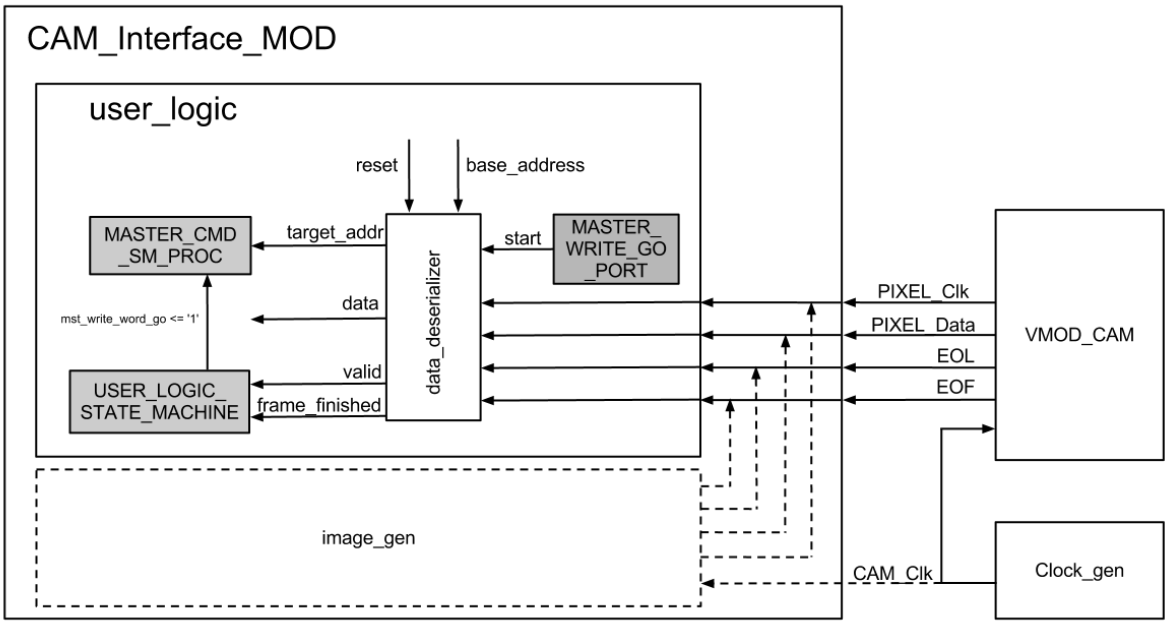

Figure 7: Camera Interface module details.
in

The user logic state machine, the data deserializer state machine and their interaction as well as the camera interface are described in more detail in the next parts.

3.2.2 User Logic State Machine

The user logic state machine controls the deserializer and the MASTER_CMD_SM_PROC state machine which implements the data transfer from the user logic to the RAM over the PLB bus. Figure 8 shows the three states *wait_go*, *wait_valid* and *wait_invalid* of the user logic state machine.

The *wait_go* state is active until the Microblaze sets the "go" signal via the PLB-Bus. In this state our module keeps the value "1" in its own slave register 0, meaning that the module is ready to start transfer the next image. The Micoblaze can read this slave register in a busy waiting loop. In the other two states the slave register 0 is set to the default value "0" meaning that the module is busy and not ready. Changing the state from *wait_go* to *wait_valid* sets the mst_go_clr signal which else is not set. This signal unsets the value of the mst_go signal, meaning that we recognized the "go" signal. We also set the start signal which starts the deserializer state machine to collect data from the camera. Then in the *wait_valid* state the module waits for for valid data from the deserializer as long as the frame_finished signal and the valid signal from the deserializer are both unset. No output is generated. When the deserializer signals, that it collected 4 Byte of data, it sets the valid signal. This means the 32 Bit are ready to transfer to the ram. We start the MASTER_CMD_SM_PROC state machine by setting the mst_write_word_go signal. This state machine was generated by the custom user logic template and we modified it to listen on that new signal. It transfers the the 32 Bit to the address in the RAM that the deserializer generated and then returns to its initial state. The user logic state machine operates on the bus clock which is at least five time faster than the pixel clock which controls the deserializer state machine. That means the valid signal from the deserializer will stay set for some bus clock periods. Therefore we have to wait until the valid signal goes down and up again before we can transfer the next 32 Bit to the ram. We achieve this by introducing the *wait_invalid* state. While this state is active, the mst_write_word_go signal is kept low in order to not start the MASTER_CMD_SM_PROC state machine. When we wait for new valid data in the *wait_valid* state, we return to the initial *wait_go* state if the frame_finished signal goes up.

3.2.3 Camera Interface

The camera itself has different output signals. The End of Frame signal (EOF) is active low, meaning it is high when the camera is transferring image data. The same holds for the End of Line signal (EOL). All transferred data during a high EOL signal belong to one image line and all transferred data during a high EOF signal belong to one image. The data is transferred over the Pixel Data signal with 8 Bit width. A new byte of image data is available one every new Pixel Clock cycle and when EOL as well as EOF are high. Some shortened example signals for EOF, EOL and the Pixel Clock are given in figure 9. These signals from the camera are used in the deserializer.

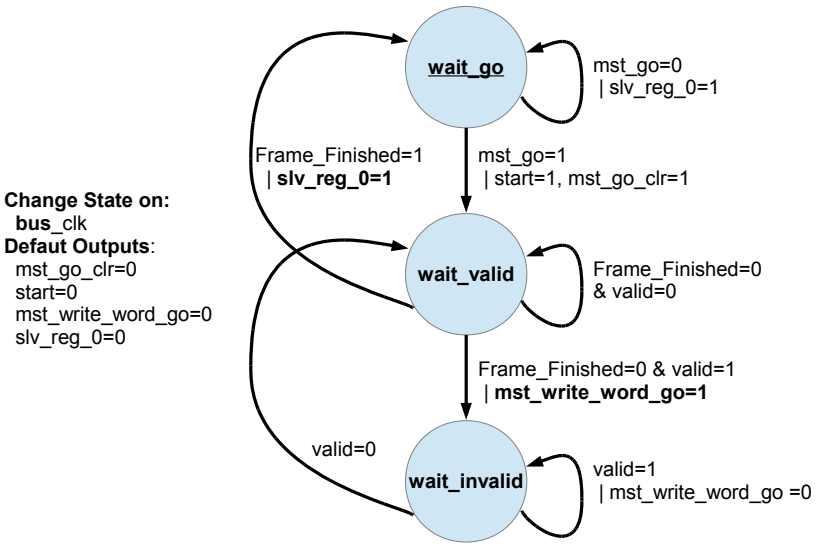

Figure 8: User logic state machine.

The camera is configured initially by the Microblaze over the I2C bus. It sends pixels in RGB565 format. This means a pixel has a size of 16 bits or 2 Bytes: 5, 6 and 5 bit for the red, green and blue planes respectively.

3.2.4 Deserializer

The deserializer converts the 8 Bit wide data stream from the camera to a 32 Bit wide data stream that can be used on the PLB bus. It receives the start signal with an target address from the user logic and the EOL, EOF, Pixel Data, Pixel Clock signals from the camera. It generates a frame finished signal as well as a valid signal to give feedback to the user logic state machine. Furthermore the address offset of the generated data in the current image is an output of this deserializer.

The deserializer is defined by the state machine given in figure 10.

The deserializer waits for the start signal (generated by the user logic state machine) in the *wait_start* state. The start signal can occur when the camera is in the middle of an image. Therefore is depends on the EOF signal if the deserializer has to wait in the *wait_no_frame* state until the current image is finished or if it can wait in the *wait_first_data* state for the first pixel data of the new image. When waiting in the *wait_no_frame* state it goes to *wait_first_data* state when the current frame finishes indicated by a low EOF signal. The first data then arrives with the high EOL and the high EOF signals. The address offset is reset to 0 and the first 8 Bit of the output data take the value of the Pixel Data input from the camera. Then the deserializer waits for the second byte in state *wait_data_2*. It will arrive with the next cycle of the Pixel Clock, so that the following state of *wait_data_2* is always *wait_data_3* when doing this

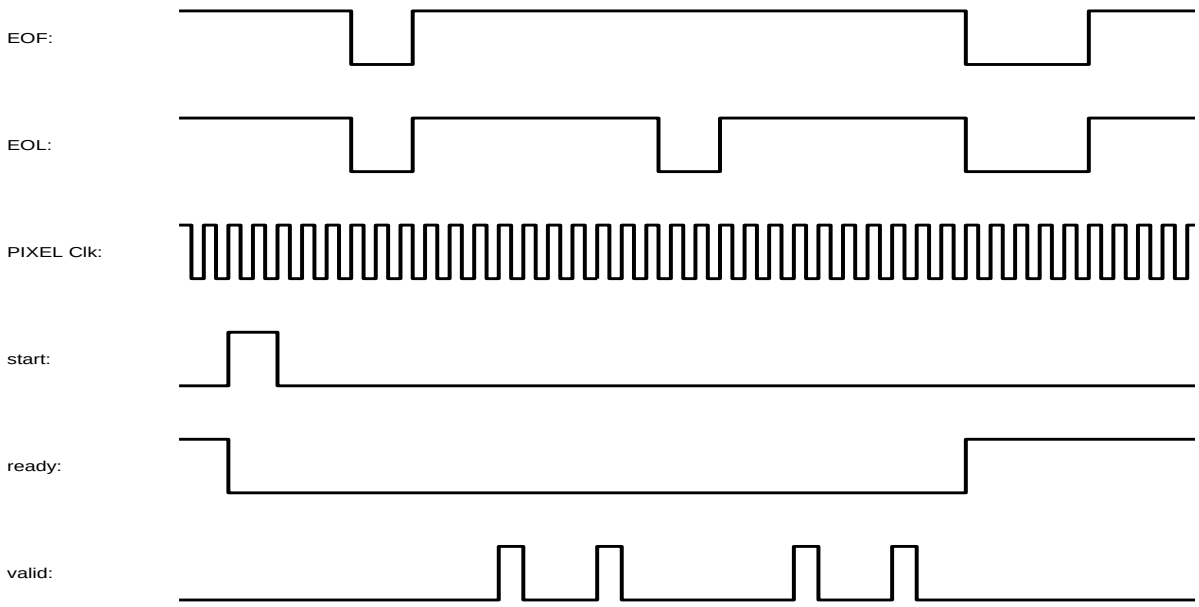

Figure 9: Example signals for the camera interface module.

transition the second byte of the output data takes the value of the Pixel Data input. The same holds for the third and fourth data byte. Additionally the valid signal is set when the fourth byte arrives. This makes the user logic state machine start the transfer of the complete 32 Bit word over the PLB bus to the defined RAM address. The RAM address itself is the sum of the input target address and the address offset generated by this state machine. After this transfer is started, the deserializer waits in the *wait_data_1* state for the next four byte of data as long as the EOF is set and EOL is not set. This tracks all bytes send from the camera in the image, because we only choose an even number of pixels for the image width. When the EOF and EOL both are set four new bytes are available: the offset is increased by 4, the first byte of the output is taken from the current input and the circle for the states *wait_data_2*, *wait_data_3*, *wait_data_4* and *wait_data_1*. When in state *wait_data_1* it is also possible that the image has finished. This is detected by the EOF signal going down. We show this to the user logic state machine by setting the Frame Finished signal and return to the initial state *wait_start*.

The deserializer state machine runs with the Pixel Clock which is at least five times slower than the Bus Clock used by the user logic state machine. That means the 32 bit word is transferred to the RAM during one cycle in the deserializer state machine.

3.2.5 Client Firmware

To initiate an image capture, the microblaze will send some signals to the camera interface module via its slave interface. The function *start_get_frame()* in the code is responsible for this task. It will write 0x40 to the control register and the target address to the address register. The target address indicate to where in the RAM the pixel

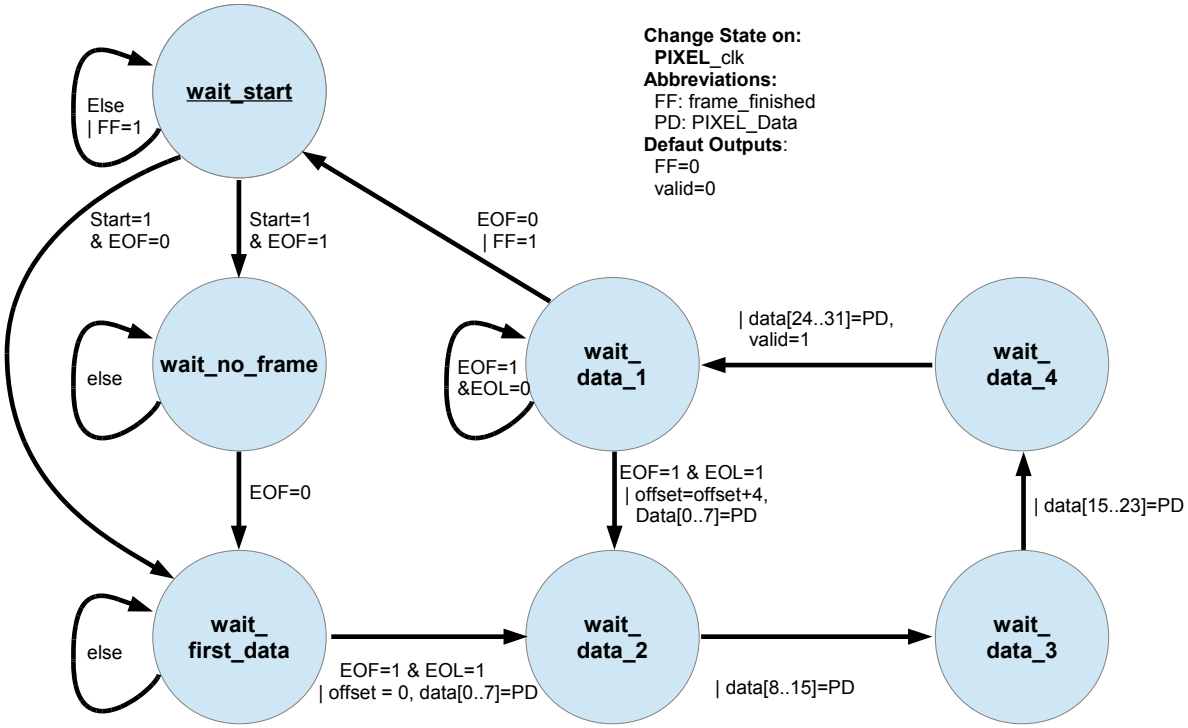

Figure 10: Deserializer state machine.

byte should be stored. Additionally the byte enable and length register of the camera interface module should also be written with some values. Finally the Microblaze sends a so-called "go" signal to the camera interface module to initiate the image capture. It does this by writing 0x0a to the go register.

The camera interface module should write a value in its slave register indicating a buffer is completely written, and the Microblaze keeps polling the value in the register at the same time. The function *camera_interface_module_wait_ready()* does this. It will return a value of 1 when a buffer is completely written and a value is written in the slave register. Then the Microblaze will send target address of another buffer to the camera interface module so that it will begin to write pixel data in this buffer. It does this by swapping the variable *store_buffer* and *send_buffer*. At the same time, the microblaze reads the first buffer and sends the pixel data over the network via a self-designed protocol ITP. The function *send_image_over_network()* will complete this task. It will determine the total packets per image based on the image size and maximum ITP packet size. Each packet is wrapped with ITP packet header and will be sent over the network via UDP.

Alternative to the double buffering concept explained above, single buffering procedure can also be used. The difference is that single buffering is slower since reading and writing buffer occur sequentially. In double buffering, the process of writing and reading buffer are independent to each other and can happen simultaneously.

3.3 Server Implementation

This server run as a application on PC and is implemented with Java. Compare to the application developed with C# or C++ can not easily be ported on different platforms, this server is designed to run on different platforms without modification. The server application can be used to preview the images send by client and have options for user to store image into disk. This application consists of four modules: Graphic User Interface, UDP Server, FIFO Buffer for incoming images and Function Realization. The Function Realization module can be extended in the further if more functions needed.

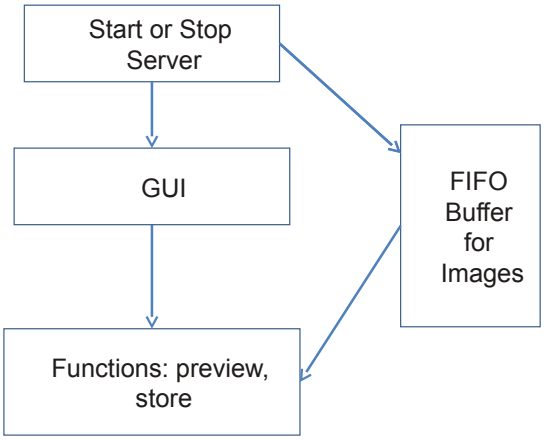

Figure 11: Overview Server Modules.

3.3.1 Graphic User Interface

The GUI is designed with Swing of Java and is divided into two main parts. the main area is used to show the recently received images. And the second area consists with several buttons to control the application to do correlate actions. When the application starts, the udp server is not really running on the background, it means that the application is not able to the user should use the "Start/Stop" button to start the udp server, and later use this button to stop or restart the server. See figure 12.

3.3.2 UDP Server

The UDP Server run as a sub thread of the application. It is designed to listen on the specific port as the same as the port used by client, then receive the date send by client. It also parses the udp packet in order to recogonize the ITP protocal and decoding the image data from UDP packet, furthmore reconstructs the image or drops the wrong packet by using the information from ITP header. As there is no ready made class in java to parsing the UDP packets, the data structure of ITP is implemented and provides

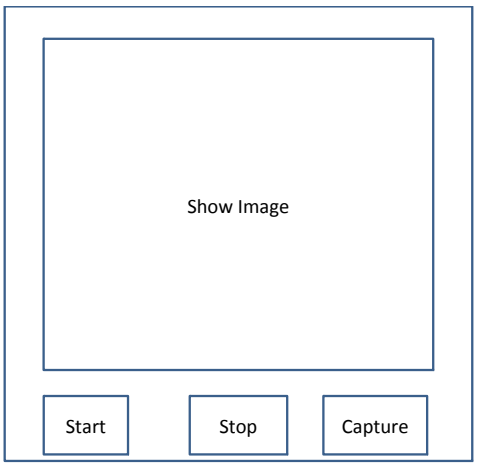

Figure 12: Graphical User Interface.

the function to anylyse the UDP and ITP packets. After decoding the ITP packet, the server compares the current image number with the image number of last packet, if they are different, then drops the old packets and start to receive a new image with current packet. Otherwise, the server checks the packet number of current packet, if this is the last packet of recent image, then rebuild the image and store that into the buffer.

3.3.3 FIFO Buffer

After udp server received image, it should first put that on the tailer of the FIFO Buffer. This buffer is necessory because of the latency of I/O to store the image or show that on the screan.

3.3.4 Functional Thread

Two functional threads will be used to realize the functions for previewing and storing image. Such as the thread for previewing, it always snoops the FIFO-Buffer.If the buffer is empty, the thread will do nothing, if there is images in the buffer, the thread will show the image on the area for image. A thread will be created if the "Capture" button is clicked. It will then get the image one the head of FIFO Buffer and store that into disk.