Christian L QUALE

MEng Hons Phase Two Project Report HSP 2386

Prototype Design of a System on Chip for Visible Light Communication

January 2013

Original Mission Statement

MEng Project Mission Statement

Prototype Design of a System on Chip for Visible Light Communication

Student: Christian Leonard Quale

Supervisor: Dr. Björn Franke

Background

My project will be done with the PASTA project [1] in the University of Edinburgh School of Informatics. The PASTA project initially began as a research-project, addressing topics within compiler synthesis and microprocessor architecture, and through its research it has developed a family of customisable embedded processors, the EnCore family.

A developing technology within communication engineering is Optical Wireless, or Visible Light Communication. [2] This is being worked on by Professor Haas et al. within the school of Engineering at the University of Edinburgh.

One of the goals of the PASTA project is to build a custom system on chip, comprising of the EnCore processor, that will enable it to communicate with other devices that facilitate Optical Wireless Communication. My project will be to design the hardware that converts, buffers and scales the signals between the processor and the optical input and output.

Aims

The EnCore processors are able to communicate with external devices using the standard on-chip AXI protocol. The signal received from and sent to the optical input and output must therefore be converted to and from the AXI protocol. The hardware required to do this will be described in Verilog, and will initially be synthesised on a FPGA which will have the EnCore processor running on it. My project will be to design the AXI-converters which will

allow the processor to communicate with the outside world through optical wireless, with a possible scope for extension being to also create an AXI-converter allowing the processor to transmit and receive data over a network.

In order to do this there are a number of interim tasks I will have to carry out.

- Become familiar with the AXI protocol and how it is used as well as the Analogue To Digital, and Digital to Analogue converters that will be used to convert the optical signals.
- Gain an understanding of the need to buffer and up-sample or down-sample the optical signals.
- Program the AXI converters themselves.
- Attempt to optimise the hardware with regards to speed, die-area required, power consumed, etc.

The supervisor and student are satisfied that this project is suitable for performance and assessment in accordance with the guidelines of the course documentation.

Christian Leonard Quale	Björn Franke

Abstract

In this report I present the work done for Phase 2 of my thesis. I have used Verilog HDL to describe a module which will facilitate the use of Visible Light Communication to receive and transmit data by a System on Chip. As part of the the signal reception, the module also carries out preliminary signal processing by detecting incoming packets, aiming to limit the amount of noise that is written to the system.

The module was designed in compliance with the open AMBA AXI protocol, and is intended to be used with the System on Chip of the PASTA group in order to facilitate further research on the practical aspects of VLC. The work has been successful, and through a number of simulations it is shown that the designed module works according to the specification. The report also outlines the main challenges faced in designing the system, and discusses the solutions which were implemented to overcome these. The result is compared to and contrasted with related work.

I declare that this thesis is my original work except where stated.

Christian Leonard Quale

Table of Contents

List of Acronyms

ADC	Analogue to Digital Converter
AXI	Advanced eXtensible Interface
BER	Bit Error-Ratio
DAC	Digital to Analogue Converter
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
LED	Light Emitting Diode
OFDM	Orthogonal Frequency-Division Multiplexing
QAM	Quadrature Amplitude Modulation
SNR	Signal to Noise Ratio
SoC	System on Chip
VLC	Visible Light Communication
WLAN	Wireless Local Area Network

Chapter 1

Introduction

For my Masters project I have designed and created a comprehensive prototype for facilitating the reception and transmission of data over Visible Light Communication. This chapter gives the reader an introduction to the project, focusing primarily on my main goals and challenges in the course of working on it.

1.1 Introduction

This thesis presents the work I have carried out in the course of working on my masters project. My project consisted of making a prototype design of a System on Chip for Visible Light Communication, using the AXI interface to interact with the system's interconnect. This chapter introduces the reader to the nature of this work, providing an outline of what I hoped to achieve, and an overview of the work carried out. Important goals and challenges are presented, giving an impression of what aspects of the work required more thought and effort than others. The nature of these goals is largely specific to the niche for which my hardware is being designed, sending and receiving data over VLC. At the end of this chapter a short overview of the content contained in this report is given.

1.2 Background and Motivation

The PASTA group [1] are working on developing the Encore Processor, and use this processor in a system implemented on an FPGA. The aim of my project has been to design and describe hardware which will enable Visible Light Communication to be used as a means of transmitting and receiving data by the PASTA system. The hardware functions would be

implemented through a Verilog module that can connect to this system as a peripheral, interacting with it through the AMBA AXI protocol [3]. The immediate motivation for this effort is to extend the potential research that can be carried out on the system used by the PASTA group to include interaction with VLC. The hardware described in this project will be tailored around the SoC used to research the performance of the EnCore processor.

However, in the process of describing the hardware I have also had a more general goal in mind. The EnCore processor and the SoC used by the PASTA group both use the AXI protocol as an interconnect standard. As previously mentioned, the module that has been developed for my project will also conform to this AXI protocol. This opens up opportunities beyond the specific application for which this module is initially being built. As the AXI protocol is widely used in a range of systems, an AXI compatible module has the potential to be used in many other applications, as it would be relatively straightforward to include as a peripheral to any system utilising an AXI interconnect.

In working on the project I have kept this in mind, attempting to limit application-specific design decisions wherever possible, and making a range of options customisable through parameters. This adds to the motivation of carrying out the work. While my primary aim has remained the development of a module for use with the experimental SoC used by the PASTA project, I have at the same time aimed to develop a basis for something that could be used in similar SoC's in the future.

1.2.1 The AMBA AXI Protocol

My project revolves around creating an AXI interface, and it is therefore important that the reader understands why such an interface is needed. This section, briefly explaining the nature of, and the reason for using the AXI interface, is heavily based on Section 3.3.3 of Phase One of my thesis [4].

The practical work in my project will to a large extent be to describe a Verilog-module that defines an AXI Interface for sending and receiving data through the EnCore processor using Visible Light Communication. Any system on chip will have a number of components that are all expected to work seamlessly with each other. These components will probably have been created by a range of different manufacturers. Therefore it is important that all the components on the system conform to a standard specification for interconnecting, so the components will be able to work together in the system as a whole.

One of the most popular standards, and the one used by the PASTA project for the EnCore processor, is the AMBA AXI[1] protocol [3]. The AXI protocol is *"targeted at high-performance, high-frequency system designs"* and is a high-speed submicron interconnect. It

[1] **A**dvanced **M**icrocontroller **B**us **A**rchitecture, **A**dvanced e**X**tensible **I**nterface

is an open standard, and describes rules and procedures that components should conform to when passing data to the address and data buses. In practice, the AXI protocol is a burst-based way of reading and writing data between a component and a memory location. It is based on components acting as masters and/or slaves where a master instructs the slave either to accept data from it, or return data from a given address back to it. Unless a component is natively AXI-compatible, it is normal to use AXI modules as a translator between the component and the rest of the system. The interaction done by my module with the AXI interconnect conforms to version 1.0 of the AXI standard, while the AXI4-stream protocol is used internally within the module.

1.3 Goals and Challenges

The work involved in designing and making my module is conveniently split into two main parts, namely the writing of incoming data into memory, and the reading of data from memory and the transmission of it. The core work involved designing the modules such that they would work properly with the AXI interconnect, fulfilling the requirements of an AXI master and slave as discussed in Section 2.2. As an addition to the original scope of my project, I was required to make a mechanism for detecting incoming signals, and adding a preamble on to outgoing signals to enable their detection. Here I describe, on a high level, what problems I had to solve, and the main challenges that had to be overcome in solving them.

1.3.1 Packet Detection

The first stage of my system should detect when the incoming signal represents the start of a data packet, as opposed to being just noise. The module should be able, in real time, to process the incoming signal and detect whether it represents the preamble which precedes incoming data. Some challenges faced in doing this are:

- Speed

  A new sample of data will be available to the detection module at a rate ranging from 50 to 75 MHz. Using samples of 16 bits, this is equivalent to a data-rate of between 800 and 1200 mbps, or 100 to 150 megabytes per second. It is essential that the system is able to handle this amount of data reliably.

- Calculation efficiency

  In order to be able to handle the required speed, the system needs to be able to perform certain, potentially heavy, calculations very quickly. The main expected computational

challenge was the cross-correlation of the incoming signal with itself, as discussed in Section 4.4.1. However, another unexpected computational challenge arose in the calculation of the signal energy. This involved adding together the magnitude of a large number of values, something which turned out to be much more difficult to do efficiently than I initially anticipated it would be.

- Accuracy

  For the packet detection to be useful, it needs to be accurate. Its purpose is to act as a "first line" interpretation of the incoming signal, dropping irrelevant samples completely, preventing the relatively slow memory from having to deal with them. The packet detection decides what should, and what should not be forwarded on to the memory, to be processed by software. Importantly, signals that actually contain data should not be lost, hence, eliminating false negatives is essential. False positives are less critical; garbage data will be treated as garbage data, and noise in the system will not cause problems beyond wasting some computation cycles. Regardless, the packet detection should be as close to perfect as possible, and the large majority of data written into memory should be actual data packets.

- Hardware utilisation

  When having to process data quickly, the problem can often be made easier by using more hardware. While spatial optimisation in this case was secondary to the time constraints, this was still an issue that had to be taken into account when designing the system.

1.3.2 Signal Reception

Receiving signals is one of the basic functions that needs to be carried out by the system. These are the challenges that were faced, and the problems that had to be solved, in enabling the system to do this.

- Buffering

  The buffering requirements when receiving the signal were straightforward, but tricky to execute in practice. Any incoming signal is written to memory through the AXI interconnect, but due to the nature of the AXI protocol, the speed at which transfer of data is carried out is unreliable. Therefore, buffering has to be done to minimise the risk of any AXI related problems impeding on the data handling in the system.

- Writing to Memory

  A limited slice of the memory is available for incoming data to be written to. The module receiving the signal must therefore keep track of which addresses in memory have already been written to, and which of these addresses have been processed by the system, and are therefore free to be reused.

- Reporting Back

  Uniquely among the modules in the system, the module writing to memory should preferably have a way of talking back to the SoC. The SoC needs to be kept up to date about various pieces of information, such as whether the module is ready to receive data, or whether the allocated memory is running low on free capacity.

1.3.3 Signal Transmission

One of the basic specifications of the system is for it to transmit data using visible light. In practice this is done through reading 16-bit Q15[2] formatted samples from memory, and transmitting them through a DAC. A few problems need to be overcome to do this.

- Buffering

  Any outgoing data has to be read from the memory through the AXI protocol. The AXI protocol is fast, but it works in an unreliable, best-effort manner with regards to throughput. Considering the separate clocks driving the AXI interconnect and the actual signal transmission, as described in Section 3.2.2, the output of data from memory should in theory be fast enough for the system to be able to transmit in real time. However, the reading of memory through AXI is not reliable. As a packet needs to be transmitted continuously and without gaps, a buffer has to be in place to minimise the chance of data not being read from memory before it is due to be transmitted.

- Transmission of Preamble

  Prior to any outgoing signal, a 160-sample preamble needs to be transmitted. The module transmitting the signal should therefore be set up to transmit this signal, prior to transmitting any data.

[2]The Q15 format is a 16-bit number where the 15 least significant bits are fractional bits, and the most significant bit represents the integer -1. If the most significant bit is set, the fractional bits are added on to this. If the most significant bit is not set, the result is a positive number represented by the fractional bits.

1.3.4 Simulation and Testing

The hardware description will be done according to a specification, and test procedures will have to be defined and available to ensure proper functionality of the system.

- Generating sample signals

  To test the system properly, data samples have to be generated which resemble the actual data the system will have to be dealing with. Due to the large number of samples present in a packet of data, a method of automatically generating these samples must be designed.

- Simulating "real life" operation of the system

  In addition to testing the system with data, other factors have to be taken into account. Most importantly, tests must be carried out to see how well the system handles and deals with varying levels of noise. Edge cases also have to be kept in mind, and the tests must be designed with these uncommon situations in mind.

1.3.5 Overall Goals

In addition to the specific challenges faced in the making of individual parts of the module, other goals related to the system as a whole also need to be considered.

- Abstracting parameters out of the code

  In carrying out the work, some design decisions and tradeoffs will have to be made along the way. As many as possible of these design decisions should be easily configurable, for the benefit of modifying these decisions through parameters if different priorities are desirable. This could be done through a file included alongside the Verilog code, which would contain variables that can be used throughout the code. In designing the modules, a goal should be to make them as parametrisable as possible.

- Creating a robust AXI implementation

  While this is a basic and crucial part of my project, it is also a challenging one. The system should conform to all the AXI standards, and be able to work reliably in any AXI-based system.

1.4 The Nature of this Report

The work required to develop a system such as the one for my project consists of two main parts. Firstly there is the system design, and secondly there is the writing of code based on this design, including testing and debugging. While these two stages are not entirely separate, e.g. a design may change as a consequence of errors found in the code, they are of a very different nature.

This report discusses almost exclusively the system design aspect of my work. The process of writing the code is only mentioned in high level terms describing how it was achieved, rather than the inclusion of the code itself. The code itself, and the process involved in writing it, would be of limited or no interest to anyone reading this report. The exact code used to implement ideas is much less interesting than the ideas themselves, and does not contribute to enhancing the understanding of these. This is especially the case for a hardware description language such as Verilog, where very different code, mainly because of coding style, can result in exactly the same hardware. While this report contains little actual code, I have endeavoured to include sufficient information about my design so that an interested reader would be able to create their own code using these ideas.

Parts of the Verilog code written for the project are included in Appendix A. The code in its entirety can be found in the PASTA SVN repository.

1.5 Overview

Following this introduction, Chapter 2 describes the background material, and the tools used to carry out the work for my project. Then, in Chapter 3, comes an overview of the system as a whole, giving a high level description of the hardware I have designed for my project. Chapter 4 describes what I did to detect incoming packets, a feature added to the project scope after I had started the work. Chapters 5 and 6 contain detailed descriptions of the work and simulations carried out for the two main parts of my module, the parts of the system that interact with the memory, reading and writing data. Finally the report concludes with a chapter on related work, followed by a conclusion.

Chapter 2

Tools and Background

This chapter covers the background and theory required to put my project into a context. The AXI protocol is discussed, along with the experimental setup around which my module is designed. It also outlines the tools used to simulate and to carry out synthesis on my work.

2.1 Introduction

In my project, a number of tools have been used to facilitate the work. This chapter gives an overview over these tools, and describes how they were used. The system has been built around an experimental setup, and is based heavily on the AXI protocol. Given the important roles that both the setup and the protocol play in my work, these are discussed in some depth. This should provide the reader with sufficient background information to fully appreciate the work which is described in the rest of this report. Some background is also given on aspects of the external system. This environment is outside the scope of my project, but will eventually be used in conjunction with what I have created.

2.2 The AXI Protocol

My project is heavily based around the AMBA AXI protocol. The purpose of the AXI protocol is to act as an interconnect between different components in a system, allowing both for interoperability within a system, ensuring that all components of a system are compatible, and the re-usability of any component that is designed to adhere to the AXI standard. The rationale behind the use of the AXI protocol, and general background information relating to it, can be found in Section 1.2.1. Below is a simplified overview of the more practical

aspects surrounding the AXI protocol, describing how it will be used in the system that is being designed.

Seen from a high level, the AXI protocol facilitates the reading and writing of data between components in a system. A common use-case for AXI would be to read data from memory prior to carrying out operations, and writing data back once these have operations have been carried out.

Data is transferred through "bursts", which can consist of a number of data transfers. A single transfer of data can contain varying numbers of bits. In the PASTA SoC, and in my module, a single transfer consists of 32 bits. A burst of data can be made up of as little as one data transfer, up to as many as sixteen data transfers. A burst of data is contained within a "transaction", which denotes the process used for reading and writing data. For every transaction a procedure is followed by the components to declare the address required, as well as signalling the successful reception of data. Transferring data in bursts makes the transfer of data faster, as time is saved which would otherwise be required to carry out the transaction procedure for each individual 32-bit segment of data.

This is all done through an AXI interconnect which in effect carries out arbitration between transactions. The protocol operates over five channels which are described in the AXI protocol [3] and summarised below.

Write address channel

This channel handles control signals relating to the writing of data from a Master to a Slave, including the address to be written to, as well as the length of the burst, and the size of each transfer in a burst. These signals are asserted by the Master.

Write data channel

Handles the actual data to be written by a Master to a Slave. These signals are asserted by the Master.

Write response channel

Handles the completion of the write transaction, including the ID tag of the transaction. These signals are asserted by the Slave that is being written to.

Read address channel

In this channel the desired address to be read from is asserted, as well as the size of the bursts in a read transaction, and the number of bursts. These signals are asserted by the Master.

Read data channel

This channel contains the data that is being read by the Master from the Slave. These signals are asserted by the Slave.

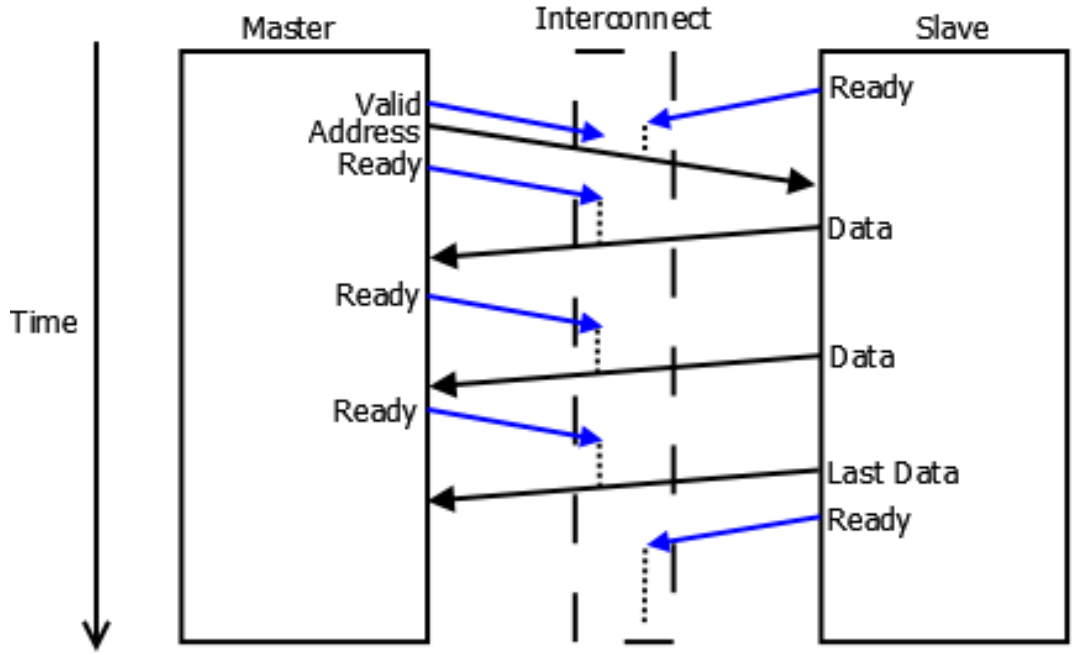


Figure 2.1: A simplified overview of the interaction between an AXI Slave, Master and interconnect when the Master requests to read data from the Slave, in this case in form of a burst containing three data transfers. Blue arrows represent signals used by the interconnect, while black arrows represent data transmitted over the data channel. Signals from the interconnect to the modules are omitted.

While I have described the channels above as being asserted by either the Master or the Slave, each channel also has a signal which is asserted by the other party: the "READY" signal. This signal forms the basis of the READY/VALID handshake mechanism on which every channel is based. Transfer of data is only carried out when both components involved in the transaction are ready.

Figure 2.1 shows an example of a simple read transaction between two components. Note that both modules assert a signal addressed to the interconnect, signalling whether they are ready to operate, rather than to each other. The interconnect handles these signals, and decides when these modules are to interact. This allows for multiple modules to operate on the same data and address bus, provided the interconnect keeps track of which components are currently interacting.

When the interconnect has received a signal from the Master, requesting to read data from a certain address, it will keep track of this request. When the system is ready to allow this transaction to happen, the interconnect will pass this address on to the Slave. The Slave will then transmit the appropriate data when the interconnect gives a READY-signal to the Slave saying that the Master is ready to accept the data. This happens when the interconnect knows that the system is free to transmit data and also, of course, when the Master has signalled that it is ready to receive the data.

Along with the address asserted on the address channels, each transaction is also given an ID. This ID allows the interconnect to perform out of order data transfer, if necessary, as well as allowing a component to initiate new read or write requests before the transactions associated with the previous request have concluded. The full details behind how the AXI protocol works can be found in the AMBA AXI specification [3]. The specific implementation of the AXI interconnect in the SoC used for my project was developed by Oscar Almer during the work on his PhD, and has been described in detail in Section 3.2.2 of his thesis [5].

Most of the work I have carried out during this project concerns designing and developing AXI Masters which either write data to memory, or read data from it. However, my top module also acts as an AXI Slave, enabling the system to write data to it, in order to configure the module. Hence I have, during the course of my project, implemented the AXI protocol for the purposes of writing data to a Slave, reading data from a Slave, and a Slave being written to by a Master.

2.3 The SoC

My project is centred around the description of hardware to be used along with the experimental setup used by the PASTA group for their development and research. This section introduces the EnCore processor, and describes the experimental setup.

2.3.1 The EnCore Processor

Originally, EnCore was created to implement and test the results of PASTA's work, and to prove the concepts which were developed [6]. It is based on the ARCompact instruction-set architecture, and aims to be compact and easily extendible. The processor is used as part of the experimental setup, as it is very efficient in terms of both power and size, as well as having been proven to be functionally sound [5]. It can also be implemented, efficiently, on an FPGA, making it convenient as a processor to be used for research purposes. The EnCore processor plays an important part in the experimental setup around which my modules are created, and the ultimate goal of my modules are to be an extension used by the EnCore processor. Four processors are included in the experimental setup for which I shall be designing my modules, as shown in Figure 2.2.

2.3.2 The Experimental Setup

Throughout the my project, my code has been compiled using the Xilinx compilation tools for the Virtex-6, ML605, family of FPGAs [7]. This has been done in a setup developed for

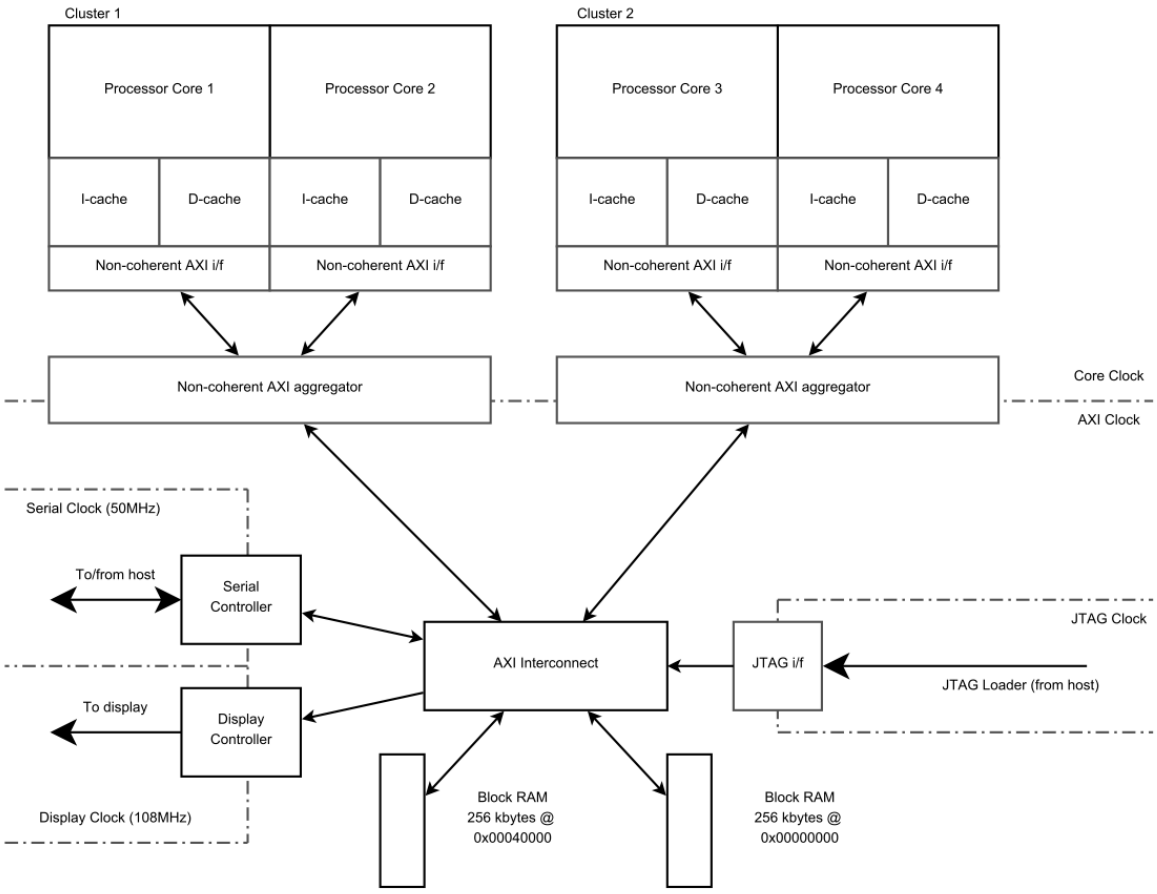


Figure 2.2: Schematic overview of the ML605 system developed by Oscar Almer for his PhD thesis [5], for which the modules presented in this project are developed. This figure is a copy of Figure 3.2 from [5].

a PhD thesis [5], a schematic of which is shown in Figure 2.2. Here I will briefly outline the practical way in which my modules are incorporated in the rest of the system for the purpose of simulating and testing the functionality of my code. During a lower level synthesis, a different incorporation method is used, as described further in Section 2.8.

For simulating the functionality of my code when used with the AXI interconnect, the Verilog files are automatically processed and placed in a directory to be run by Modelsim. Most of this is done through a series of Makefiles and configuration files which have been developed along with the experimental system as a whole, and which are therefore beyond the scope of this report. I will only be outlining how I interact with them.

I instruct the configuration files how my module should be treated by using a *.sdsc* file, located in my home directory on the *pasta2*[1] system. In this file I give my module a name, in my case *ads0*, and specify that it will be acting as an AXI Master, as well as an AXI

[1] "pasta2" is the hostname on the informatics network of the machine I have been writing and running my code on.

Slave. This is also the file in which a Slave address for my module is defined, used by the rest of the system to interact with it. As well as information about my module, this file also sets up the other modules that make up the system, including the memory, and it specifies parameters such as which clock should be used for the AXI interconnect, what depth should be used for the FIFO, and the amount of RAM in the system. While none of these settings directly impacts on my module, I left most of these settings unchanged, so that they would correspond to the settings widely used by the system with which my module is ultimately going to interact. However, one setting I did change was the program memory to be loaded into the system, for the purposes of the simulation. This is described in Section 2.5.

Based on the *.sdsc* file, the configuration file then processes the necessary Verilog-files, copies them into a separate directory, and loads these files in Modelsim. The simulation carried out in Modelsim is a simulation of the entire system, and shows its behaviour given the input of the program memory which is set in the configuration file. Through Modelsim's interface it can be observed not only how my module behaves, but also what happens in all the other relevant parts of the system, including the memory and the FIFO.

Further details on the experimental system can be found in [5], which also describes the development of it.

2.3.3 The FIFO

In my final design the asynchronous FIFO, which is part of the Virtex-6 FPGA, plays an important role. The FIFO used has a size of 36 Kbits and can be used as a block in any code to be run on the FPGA. The FIFO has the basic functionality one would expect from such a block: data is written into the memory, and can only be read out in the same order as which it was written. The more interesting, and for the purposes of my project, useful, property of the FIFO is the fact that it can be used asynchronously. This allows for data to be written and read at separate clock rates. As the clock used for receiving and transmitting data is different from the clock under which the AXI interconnect operates, this allows for using the FIFO as a buffer between the two. As long as care is taken to ensure that the FIFO never becomes full, and that data is never expected from it when it is empty, the FIFO can be used as a practical and safe way of transferring data from one clock domain to another.

2.4 VLC Processing

My work is done in order to facilitate the reception and transmission of VLC signals. Prior to transmitting, and once it has received signals, the system has to carry out encoding and decoding of these. Figure 2.3 shows an overview of the process used by the system in order

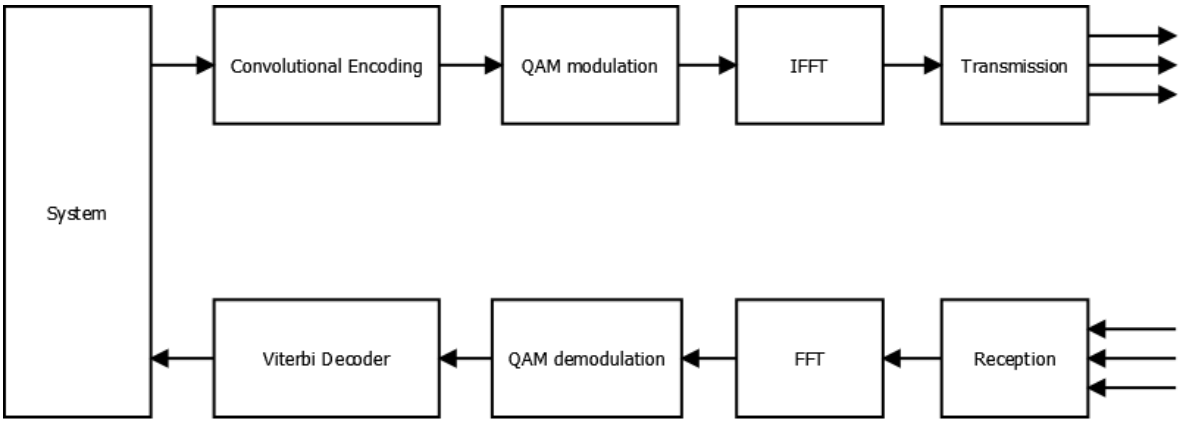


Figure 2.3: Block diagram of the process involved in encoding VLC signals before they are transmitted, as well as the process of decoding data once it has been received. The figure has been adapted from a figure created by Ewan Leaver for his Masters project [8]. The "Transmission" and "Reception" blocks represent the hardware I have described for my project.

to do this.

Outgoing data first goes through convolutional encoding and QAM modulation to ready the data for transmission. An Inverse Fast Fourier Transform is then performed on it to convert the data to samples in the time domain, and make it suitable for transmission.

Similarly, a Fast Fourier Transform is performed on the incoming data to decode it. QAM demodulation is then carried out to demodulate the modulated data. Finally a Viterbi decoder uses the Viterbi algorithm to decode the data, which has been encoded using a convolutional code.

These are all operations which are outside the scope of my project. For the purposes of my project, I will be treating this entire system as a black box which provides my system with samples to transmit, and accepts the samples I provide for it.

2.5 Creating C-tests

Most of my intermediate testing, to verify basic functionality, was done using standard Verilog test harnesses, which were created by myself and run from within the working directory using an appropriate Makefile. However, testing the system parts that interact with the AXI interconnect requires the whole system to be simulated, as outlined in Section 2.3.2. Simulating the entire system using regular test harnesses is impossible, as they cannot simulate the instructions which will be issued by the processor through the AXI interconnect. To facilitate simulation of instructions issued by the processor, programs can be created and loaded into the program memory.

This is done using C-code which is compiled to a binary file. The location of this file is

specified in the *.sdsc* file, and the contents are copied into the program memory along with the processed Verilog files being copied to the build directory.

My C-tests were all rather simple. The main purpose of the tests was to initialise the system in ways which allowed me to test my modules from within Verilog files, as well as issuing transmit instructions. A more in-depth description of how I used the C-files to test the individual modules can be found in Chapter 6.

2.6 ModelSim

ModelSim 6.4c is the simulator used by the PASTA group to simulate the EnCore processor, and is also the simulator I will be using to simulate and debug my Verilog code. ModelSim is owned and sold by Mentor Graphics, and is regarded as one of the three industry leaders for simulation of Verilog code. It is able to simulate Verilog code very quickly, and has a wide range of features making it a very useful tool for testing and debugging systems.

2.7 The .vpp File Format

Rather than using standard *.v* files, I wrote my Verilog in *.vpp* files which added an extra layer of commands which could be used for operations such as generating code, importing code, and defining variables. "VPP" (Verilog Pre-Processor) is similar to the pre-processor used in the C programming language. This allowed me to define variables in a separate file, which was imported once the module was synthesised. Throughout the project I have used a file called *ads.vh* to hold global variables such as the length of a data packet, the frequency at which I wanted to write status information to the system, and the number of significant digits to use in intermediate calculations.

In addition, code can be generated using an adaptation of C, which allows repetitive code, a common feature of Verilog, to be scripted in the *.vpp* file itself. This makes the written code more concise, and more readable. When the file is processed prior to simulation or synthesis, this code is generated and becomes part of the *.v* file which is used for these purposes.

2.8 Synthesis

Once the functionality of my code had been verified, the next step was to run it through a device-specific synthesis process. While basic syntax checks are carried out prior to the simulation of the system, many of the optimisations performed by the Xilinx synthesis tools are not carried out.

The device-specific synthesis is done from within the working directory, using a Makefile and a *.xst* file. These launch the Xilinx Design Suite and run the synthesis process with the appropriate settings, as set in the configuration files. This shows errors and warnings of a more practical nature, such as unclosed if-statements which could cause problems in the hardware implementation of the system.

The output is a *.ngc* file which gives an extensive report on how the system would be implemented on an FPGA, including reports on what hardware is used and timing information.

2.9 Conclusion

This chapter has given the reader some background information relating to my project. The AXI protocol has been covered, including a description of how it facilitates the reading and writing of data over an AXI interconnect. An understanding of this protocol is important, as it is used comprehensively in my work, and is, indeed, the justification and foundation for my project as such. The experimental setup, created in [5], which has been used for my project is described. In a sense, this is the harness in which my work has been tested, both in terms of functionality and in terms of hardware requirements. The asynchronous FIFO is covered, being the only external module I use in my project. It serves an important purpose, both in acting as a buffer for the transmission and the reception modules, and in enabling the exchange of data between different clock domains.

Finally the tools and methods used for simulation and synthesis have been described, explaining the project workflow.

Chapter 3

System Overview

This chapter gives the reader an overview of the entire system which has been created for my project. The work has been split into parts, each representing a module performing a specific task which is necessary for the system as a whole to work. This chapter summarises each of these modules, serving as reference to which function is performed by which part of the system. General information about the system, and development of it, is also covered.

3.1 Introduction

The work I have carried out for my project has been with the aim of creating a system which fulfils a specific purpose. This chapter gives the reader an overview of the system and the functionality it is intended to have, as well as defining its specification.

The purpose of the system I have designed is to act as a gateway between an ADC/DAC and the rest of the SoC. This is done by doing some preliminary processing on the incoming signal, then storing it to memory, as well as reading data from the memory, processing it, and transmitting it through the DAC. The data-handling tasks required of the system can be split into two parts, the handling of incoming data, and handling of outgoing data:

Incoming data:

- Detect when the incoming signal contains a packet
  - Calculate the delayed autocorrelation[1] of the signal

[1]The term "delayed autocorrelation" is used here to describe the required operation, and would not make sense to a reader attempting to understand it in terms of a regular autocorrelation. The operation described here is discussed in Section 4.4.1.

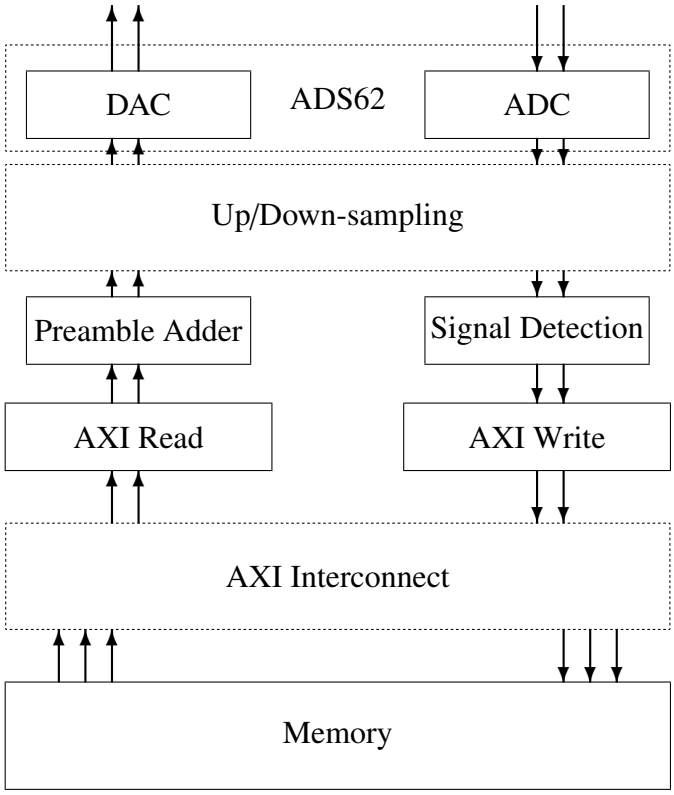


Figure 3.1: Simplified block diagram outlining the desired functionality of the system. The modules I am working on are represented in the diagram by: Preamble Adder, AXI Read, Signal Detection and AXI write.

  - Find the energy of the correlation
  - Detect peaks in this energy

- Buffer incoming signals
- Write data samples represented by the incoming signal to memory

Outgoing data:

- Read data from memory into a buffer
- Transmit a preamble
- Transmit the packet directly after the preamble

A high level overview of the system can be seen in Figure 3.1.

3.2 Specification

The purpose of the module is to enable the SoC to send and receive data through Visible Light Communication. For this to happen, the incoming signal, in the form of light with varying intensity, needs to be picked up, interpreted and passed on to the SoC. The light will be picked up by a photosensor which is connected to an Analogue to Digital converter (ADC), which in turn will convert the intensity of the light to a 16-bit two's complement input.

The system will have to be designed to handle 16-bit incoming data in two's complement, and will be required to handle one input of 16-bits for every clock cycle. The system will be required to carry out preliminary packet detection on this signal, and write data to memory only when it believes the incoming data is part of a data packet. As the data will be incoming at a high rate, potentially faster than it can be written to the system memory, this data will need to be buffered and written to the memory as quickly as possible. The writing of the data to memory will be done using the AMBA AXI protocol [3] as an interconnect.

The system as a whole will also have to act as an AXI Slave to enable it to receive instructions from the SoC. The system has been given its own address space in the system, the base of which is `hFF050000`, and it picks up data which is sent to these addresses on the common bus. The settings, or parameters, set in this way are used primarily to specify which addresses in memory the module is to interact with.

3.2.1 Final Test

The whole system will be tested using a combination of a binary file written in C, and a Verilog test bench. The system is tasked with both writing data in a given format, and reading data in that same format. The system can therefore be tested by having it read out data, before transmitting the same data back to itself and checking if it correctly detects and reads the data it has sent to itself.

The test will consist of a program which first writes data to memory, then gives my module instructions to read this data out. Instead of transmitting the data through the DAC, the data will be treated as if it were an output from the ADC. The module should then detect it as being a data packet, and write it back to a different address-range in memory.

The functionality of this will be monitored and verified in Modelsim.

3.2.2 Working with Two Clock Domains

The system I built for my project had to be designed to use two separate clocks which I will be referring to in this report as the "sample clock" and the "AXI clock".

The sample clock is the clock used to receive incoming samples, and to transmit outgoing samples. This clock is assumed to operate in the range of 50 MHz to 75 MHz. The rate of this clock is determined by the downsampling and upsampling that is performed on the incoming and outgoing signals respectively. This is handled by a module not made by myself, shown in Figure 3.1 as "Up/Down-sampling".

The AXI clock can be considered the main clock in the system, and is the clock the AXI interconnect operates at. This clock is assumed to operate at about 100 MHz.

Working with these two clocks has the effect of complicating some of my work, while making other aspects of it simpler. The obvious downside to using two clocks is that the accidental crossing of clock domains can lead to unfortunate errors in a system, errors which might not be picked up by the synthesis process. Also, transferring data between the two clock domains can be tricky, as asynchronous logic has to be used.

Fortunately the transfer of data was simplified by the availability of an asynchronous FIFO, discussed in Section 2.3.3. I prevented the crossing of clock domains by dividing my Verilog code into separate sub-modules, depending upon which clock the code was designed to work with. The components that handle the incoming and outgoing samples, as listed below, are therefore designed as top files for three sub-components; the part that interacts with the AXI interconnect using the AXI clock, the part that transmits or receives samples, using the sample clock, and the asynchronous FIFO which uses both clocks as inputs.

In addition to making the system "safer", this also contributes to making the code more re-usable, as each of these modules could be used individually in other systems to carry out the tasks they were designed for.

3.2.3 Components

Multiple sub-modules will work together in order to fulfil this specification. This is a short summary of the main components, giving a brief overview of the role they play in the overall system. Note that this is not an exhaustive list of the components contained in the system, as some of those listed here have further sub-components of their own. However, the functionality of these sub-components serves to facilitate the functionality of the components described here, and is therefore treated as being part of them.

In addition to the components listed, I have also made use of a FIFO, included in the Virtex-6 FPGA hardware. This is described in Section 2.3.3.

The Top Module

The top module acts as the block containing all the other modules in the system. It facilitates all incoming and outgoing signals, and is the only block which is directly connected to the

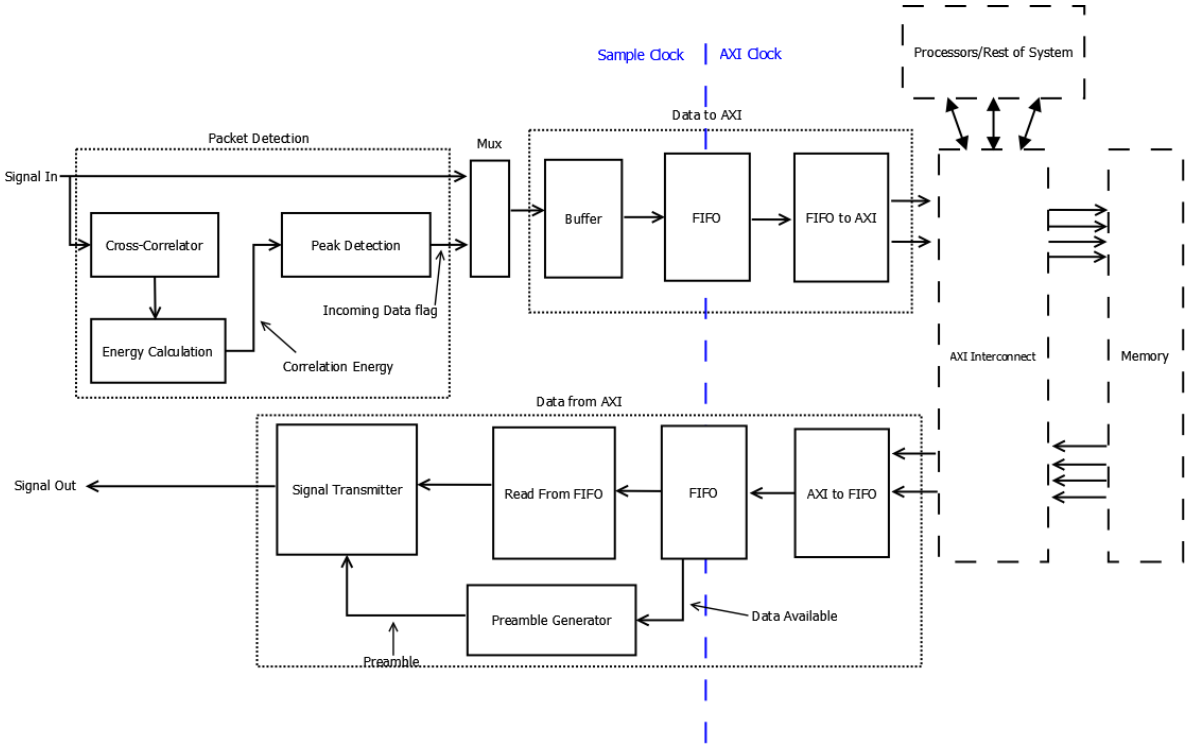


Figure 3.2: Block diagram showing the functionality of my system. Note that, for the sake of readability, most signals have been omitted. Unless otherwise indicated, arrows represent the flow of data. A more detailed description of each individual module can be found in the relevant chapters of this report. As discussed in Section 3.2.2, two different clock domains are used, separated by the FIFO blocks. All blocks shown in this schematic are created by myself, apart from those shown with dashed lines: The AXI Interconnect, the Memory and the Processors/Rest of the System.

AXI interconnect. This module also holds registers containing all configurable values, such as memory addresses, and handles the interactions required to pass these values to the parts of the system where they are required. In order to receive these values from the system, this module also has to act as an AXI Slave. More information on how the module acts as a Slave is given in Section 3.4 in this chapter. The top module is also the module that ties all the other modules together, connecting any inputs and outputs between sub-modules together in the appropriate ways.

Signal Detection

Here the incoming data is being processed continuously with the aim of detecting a preamble. All incoming samples are processed using a cross-correlation algorithm, along with a signal energy calculator and a peak detector. The purpose of this module is to prevent the rest of the system from having to write to memory samples which only constitute noise. More on this module can be found in Chapter 4.

Data to AXI

This module handles signal reception, and is responsible for receiving the incoming signal and writing it to memory once the start of a packet has been detected. The module keeps track of which memory addresses have, and have not been written to, and which addresses have been freed up by the system for further writing. If the signal which symbolises that a packet has been detected is high, the incoming data is written into a buffer. Whenever there is unwritten data in this buffer, this data is written to memory, provided that it fits into the available space. A full description of this module can be found in Chapter 5.

Data from AXI

This module reads data out from the memory, through the AXI protocol, and transmits this to the DAC. It awaits an instruction telling it to read, then reads a packet-worth of data and stores it in a FIFO. Once a given amount of data has been written to the FIFO, a preamble is transmitted, followed by the packet itself. More on this module can be found in Chapter 6.

3.3 Implementing the AXI Protocol

In this report I have largely omitted the discussions of the specific efforts involved in implementing the AXI protocol in my system, often referring to the implementation using words such as “simple” or “straightforward”. This should not be taken to mean that the implementation of AXI in the system was trivial. The majority of the time spent on developing the system described in this report was spent on the implementation, debugging and testing of the logic interacting with the AXI protocol.

The reason this has been downplayed in the report is the “uninteresting” nature of the system design that I had to carry out, as the system is already designed and described in the AXI protocol. Words such as “simple” and “basic” refer therefore to the nature in which the AXI protocol was seen as a black box in terms of functionality. The development of the actual AXI-related code has been challenging, but neither novel nor especially innovative. Details of the general AXI implementation are therefore not included in depth, and have rather been summed up in a few sentences at the appropriate places. The interested reader can find the specifics of the AXI protocol in the specification [3], which I rigorously conformed to when designing my system. Examples of the Verilog code used to implement the AXI protocol can be found in Appendix A.

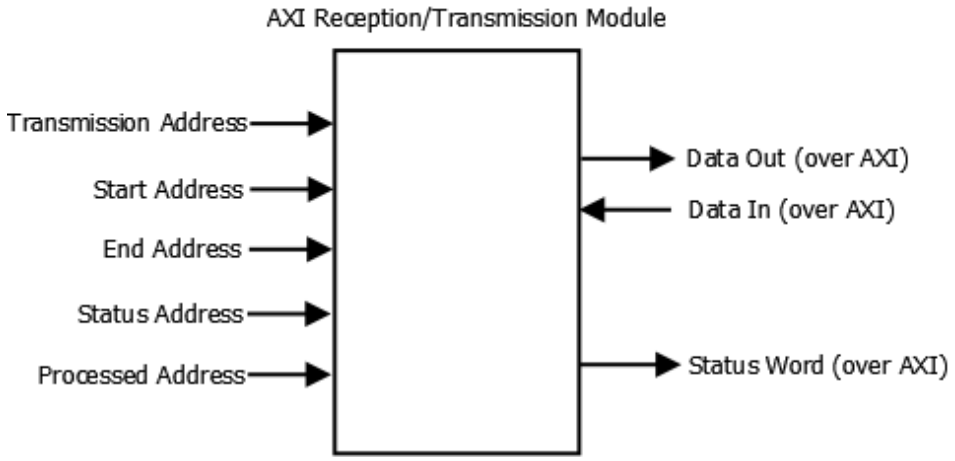


Figure 3.3: Schematic showing the overall module, and what inputs and outputs it contains. AXI signals, as well as signals representing the interaction with the ADC/DAC have been omitted. Table 3.1 lists the address of the input signals, as well as describing the function of these.

3.4 AXI Slave Functionality

The top module, which mainly works as a web which connects the rest of the modules, also carries out an important task in being the databank containing parameters required by the other modules. This is done by serving as an AXI Slave, enabling the rest of the system to write data to the registers where these parameters are contained. In effect, the module is seen by the SoC as a very small write-only memory where certain memory addresses correspond to certain parameters used by the system. The implementation of this Slave protocol is relatively simple, based on the AXI protocol. By default the module sets the AWREADY signal to high, signalling that it is ready to receive any incoming data. Once an AWVALID signal is seen along with an address belonging to the module, meaning that data is incoming, the module enters into a state machine which receives data, and responds appropriately as described in the AXI protocol. The incoming data is written to the relevant register, as determined by the address to which the incoming data has been written. More detailed information on AXI in general, and on how Masters and Slaves interact, is given in Section 2.2.

Table 3.1 shows the inputs accepted by the top module, and what addresses have to be used by the system to access them.

3.5 Conclusion

Ultimately, the goal of my project was to design a system which could send and receive data through the medium of Visible Light Communication. While the overall system has been designed to carry out this specific task, the system as a whole is the sum of a number of smaller parts which all work together to deliver the desired functionality. This chapter has presented

a high level overview of these parts, outlining their purpose, and creating some context with regard to how these parts interact. This interaction requires a module which can both link the various parts and hold the parameters used by them. In my system, this is done through implementing a top module which acts as an AXI Slave, facilitating all inputs and outputs for the other modules, regardless of whether the inputs are outputs from other modules, outputs from the system as a whole, or parameters. This allows the individual modules to work independently as black boxes, provided their inputs and outputs are connected appropriately.

The next chapter goes on to discuss the process carried out to detect data packets in the incoming signal.

Parameter	Address (hex)	Description
Transmission Address	hFF050000	The address in memory at which a data packet is waiting, ready to be transmitted. Receiving this signal acts as an instruction to transmit a packet of data.
Start Address	hFF050008	The address in memory to which the module should start writing incoming data.
End Address	hFF05000C	The upper-bound memory address to which the module is permitted to write data.
Status Address	hFF050010	The address to which my module should write its status-word, information from the module which the rest of the system needs to interact with it, as described in Section 5.3.4.
Processed Address	hFF050014	The processed address, which has to be located between the start address and end address, is the last address which has been read and processed by the system. This tells the module which addresses in the memory can be re-used.

Table 3.1: Table showing the inputs accepted by the system, and what these inputs affect.

Chapter 4

Signal Detection

One key function of the system to be designed is to write the received data to memory. For system efficiency, the noise needs to be separated from the data so that most of the signal written to memory represents actual data packets. Detecting this data is a complicated process, worthy of its own chapter. This chapter will describe the theory behind frame detection, why it is done, how my module is designed to achieve it, and how I implemented this in the system.

4.1 Introduction

One prime purpose of Visible Light Communication is high-speed data transfer. However, there are bottlenecks in a system with regard to how quickly data can be written to memory, and once the data is in memory, how quickly the system can extract actual data from the encoded data packets. One solution to this problem is to limit the reception to input that is known to contain data packets, ignoring as much of the noise as possible.

This problem has been approached previously in the design of IEEE802.lla [9], with the transmission of data using OFDM, and it is solved by prepending a preamble on to all outgoing packets. The preamble consists of two parts. One part consists of two training-symbols, enabling fine-grained timing synchronisation of the incoming signal by a system. The second part is a preamble, allowing for less exact, but easier, detection of incoming signals. This part consists of a 160-sample segment, made up of 16 samples repeated ten times. Detection of this signal can be done efficiently using electronics, and for my module I adapted the method presented in *Frame Detection For Synchronization In OFDM* [10].

In terms of the logical construction of the system as a whole, this component is regarded

as part of the reception module covered in Chapter 5. However, this module is used as a black box in the signal reception module, and is complex enough to warrant a separate chapter to describe it. This chapter presents the work I did to enable signal detection, and the challenges I had to deal with.

4.2 Specification

The signal detection module performs the first-line analysis and processing of the continuous incoming signal. Crucially, it needs to be able to operate quickly enough to process the incoming signal in real time. The module needs to detect the preamble, and set a signal high which alerts the system that an incoming packet has been detected. False negatives are more damaging than false positives; at worst, a false positive will cause the system to process a packet worth of garbage data, while a false negative would cause actual data to be dropped.

4.2.1 Test to fulfil

The module will be tested in Modelsim. I will use a sample set consisting of 10 000 samples of data generated in Matlab that will contain packets at various signal-strengths imposed on to various levels of noise. The verification of the module will be to see whether it will pick up these packets without erroneously detecting too many packets.

4.3 Why a preamble?

Due to the real time nature of the signal being processed, the module must be able to detect packets with a process that is not too time-consuming. To address this problem, the existence of a periodic preamble as part of the signal, prior to all transmitted data packets, is assumed. The idea of using a preamble, and the principles of its design, is an idea adapted from the

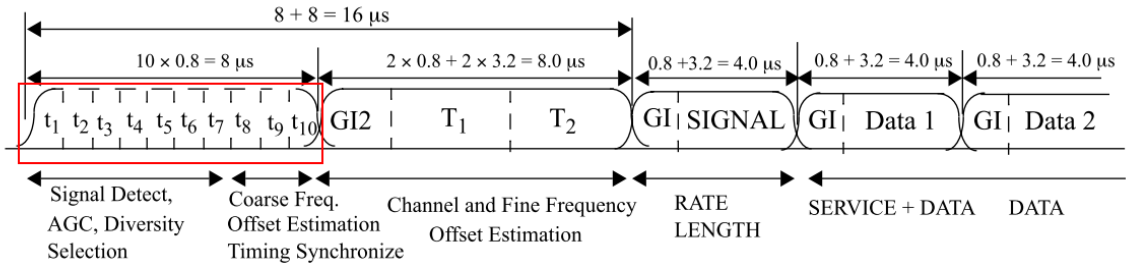


Figure 4.1: The training structure of data packets transmitted using OFDM. The red rectangle shows the 10 segments that are included, in the form of ten 16-sample segments, as a preamble to VLC data packets. The original figure is a copy of Figure 110 from the IEEE 802.11a standard [9].

specification of OFDM in the IEEE 802.11a standard [9], shown in Figure 4.1. However, the way in which this preamble is used for VLC differs from the OFDM standard, which generates it on the fly based on sub-carriers, and decodes it using FFTs.

Instead, the periodic nature of the signal is exploited, an idea adapted from the method used in *Frame Detection For Synchronization In OFDM* by Thakur and Khare [10]. Rather than using FFTs, which are computationally heavy, the periodic signal can be detected using methods similar to a cross-correlation. The signal resulting from a cross-correlation on noise will be random, hence have a very low signal energy. However, the resulting signal from a cross-correlation on two signals resembling each other will contain peaks, making the resulting signal energy much higher. This can be used as a method to detect the beginning of data packets, eliminating the need for performing FFTs, in real time, on all received signals. The incoming signals can be ignored by the hardware writing to memory until the presence of a preamble is detected, at which point data is passed on to, and processed by, the rest of the system.

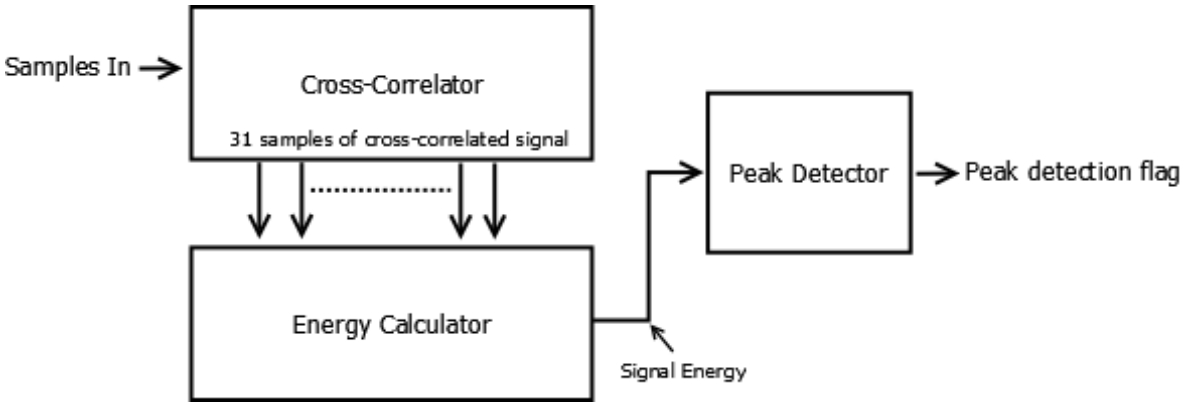


Figure 4.2: Schematic of how the signal detection is done, from left to right: the signal comes into a cross-correlator, 31 samples representing the cross-correlation is sent to an energy calculator which calculates the energy of the signal. Then the energy goes through a peak detector which detects peaks in the preamble, representing the start of an incoming data packet.

4.4 Modules

A number of modules work together to perform the signal detection. Firstly a cross-correlation module takes the incoming signal, performs the appropriate cross-correlation on the signal, calculates the energy of the cross-correlation and outputs this signal energy. In this process the cross-correlation module uses two other modules; the multiplication module for multiplying numbers in the Q15 format, and the *energy* module for calculating the signal energy of the cross-correlation result. Figure 4.2 shows a block diagram illustrating the functionality of this module.

4.4.1 Cross-correlation

My module performs a special case of a cross-correlation, bearing similarities to an autocorrelation. While a cross-correlation is generally a way of determining the delay which has to be applied to a signal to make it similar to another one, in this case the interest lies in finding out when this similarity occurs with a fixed delay, and a varying input signal.

The definition of a cross-correlation of two 16-sample signals for a given n, representing an offset in time, is as follows, where f^* and g are signals to be compared:

$$\sum_{m=-16}^{16} f^*[m]\ g[n+m]$$

Note that f^* denotes the complex conjugate of f, but as I am dealing with real numbers, this is equal to f. Using this definition, which consists of a number of multiplications and additions, the method shown in Figure 4.3a can be used. Each of the resulting bins is then treated as samples making up a signal, of which the energy is then found. On the face of it, carrying out this calculation would be very inefficient. If all these multiplications were to be done for every input, the module would be required to perform $16 \times 16 = 256$ multiplications for every new sample to enter the module. Rather than doing this, I am exploiting the fact that many of the multiplications are repeated, as illustrated in Figure 4.3 b) and c).

In principle, this means that only 31 new calculations will have to be carried out for every new sample to enter the system, and the results of the appropriate multiplications are to be added to the values which had already been acquired. My first implementation therefore included registers for all the positions in which calculations are performed, as shown in Figure 4.3a. Rather than carrying out calculations to establish the number for each register, it would move the results from $X_i Y_j$ to $X_{i+1} Y_{j+1}$ for every new input, exploiting the symmetry, only having to perform 31 multiplications to calculate the new numbers. It would then add together the registers for each bin, resulting in the complete output once this had been carried out for every bin. I implemented the module with this method, and it was functionally sound. However, there was a problem with this implementation which presented itself when synthesis was carried out.

For every bin, up to 16 numbers had to be added together in every clock cycle. In total, 256 additions had to be carried out. All these numbers were also stored in separate registers, requiring a very large number of flip-flops. This led to a very large amount of adders and flip-flops being used by the module. It would be desirable to avoid this.

Consulting with Oscar on the matter, I was advised that the FPGA contained many multipliers, and that these worked relatively quickly. I had previously designed my system using the general rule of thumb that adders are preferable to multipliers. After further consideration,

I realised that I could perform the cross-correlation using far fewer adders, at the expense of adding just a few more multipliers. As can be seen in Figure 4.3, the numbers added to make up the value of a given bin are mostly the same. It would therefore be sufficient to keep only one register for each of these bins, and for every new sample to enter the system, only two additional values have to be calculated: the new number to be added to a bin, and the value of the number that is leaving the bin. The latter has to be subtracted. This results in a need to perform 60[1] multiplications in every clock cycle rather than 31, but this turned out to be an acceptable tradeoff considering the much larger savings in terms of adders and flip-flops. Examples of the code I used to implement this can be found in Section A.5.1 of Appendix A.

Configurability

When performing the cross-correlation, and required multiplications, extensive arithmetic is involved. By default the inputs are Q15 formatted numbers, and have 16 significant bits. Due to the preservation of values that has to be done in the module, the cross-correlation module alone has 57 registers and 57 wires which are synthesised down to being flip-flops. As one flip-flop is needed for every bit of these numbers which is being stored, cutting as few as three significant bits off each number could therefore save as many as $3 \times (57 + 57) = 342$ flip-flops. Since signal detection only relies on detection of the difference in signal energy, the exact value of these numbers is not important, and a lower resolution could be used when processing them. I therefore made the module configurable, allowing the number of significant bits to be set as a parameter. The module will only make the registers and wire-buses wide enough to fit as many bits as what has been specified. When testing, I could then try reducing the number of significant bits used in these calculations until I observed the module no longer functioning as expected.

4.4.2 Theory

The main purpose of a cross-correlation is that of comparing one signal to another, finding the correspondence between them. This can be thought of as convolving two signals with each other using a constant offset.

Ways of doing the correlation

I had the choice between two different ways of detecting the preamble of incoming data packets. One way is closer to the traditional method for performing cross-correlations. Since I know the exact nature of the preamble that is expected to precede any incoming packet, I

[1] Due to the first and last bin being equal to the product of only one multiplication, the number of multipliers required is a little less than double of the previous requirement.

	X_1	X_2	X_3	X_4	X_5				
	Y_1	Y_2	Y_3	Y_4	Y_5				
X_5Y_n:	X_5Y_1	X_5Y_2	X_5Y_3	X_5Y_4	X_5Y_5				
X_4Y_n:		X_4Y_1	X_4Y_2	X_4Y_3	X_4Y_4	X_4Y_5			
X_3Y_n:			X_3Y_1	X_3Y_2	X_3Y_3	X_3Y_4	X_3Y_5		
X_2Y_n:				X_2Y_1	X_2Y_2	X_2Y_3	X_2Y_4	X_2Y_5	
X_1Y_n:					X_1Y_1	X_1Y_2	X_1Y_3	X_1Y_4	X_1Y_5
	bin1	bin2	bin3	bin4	bin5	bin6	bin7	bin8	bin9

(a) The general method I use to calculate the output signal from a cross-correlation. The bins in the bottom line are sums of the numbers above them, and each of these bins represent a sample of the output signal from the cross-correlation. This figure shows the calculation using 5-sample inputs. In the actual system these inputs are 16 samples each, and there are 31 output bins.

	1	2	3	4	5				
	6	7	8	9	10				
	30	35	40	45	50				
		24	28	32	36	40			
			18	21	24	27	30		
				12	14	16	18	20	
					6	7	8	9	10
	30	59	86	110	130	90	56	29	10

(b) An example calculation. The numbers used for the calculation, 1 through 10, represent consecutive input samples. 1 is the most recent sample received, while 10 is the least recent sample.

	11	1	2	3	4				
	5	6	7	8	9				
	20	24	28	32	36				
		15	18	21	24	27			
			10	12	14	16	18		
				5	6	7	8	9	
					55	66	77	88	99
	20	39	56	70	135	116	103	97	99

(c) The calculation following the one shown in Figure 4.3b. *11* has arrived as the most recent input, shifting all inputs along to the right. Note the symmetry in the calculation, all numbers shift upwards to the row above. Examples of this are the numbers in red and blue, which can be seen to have shifted in relation to the same numbers in the previous calculation, in Figure 4.3b.

Figure 4.3: Examples of the cross-correlation calculations.

could take the cross-correlation between the known 16-sample preamble segment, and the incoming signal. I ran some tests using this method of correlation, as described in Section 4.5.3.

The other way of detecting the signal, which I opted to use, was to do the cross-correlation of the signal and a 16-sample delayed version of the same signal.

On the face of it, there are upsides and downsides to each of these methods. One upside of comparing the signal to a known preamble is that it would take less hardware to compute the cross-correlation, as many of the values used to perform the calculation would remain constant. Another apparent advantage of this method is that it would prevent the system from detecting unrelated signals of a periodic nature, as only the known preamble would have the desired effect. The downsides are that this would be less flexible, as the expected preamble would have to be changed on a hardware level if it were to change. Also, correlating with a known preamble makes the system more vulnerable to noise. If the correlation is carried out between a set, ideal, signal and an actual signal, the resulting peak in the correlation signal's energy will reduce dramatically if the actual signal is distorted by noise. If the signal is compared with itself, given a 16-sample offset, these two segments are more likely to be similar, as they will both have been affected by similar noise.

Ultimately, I decided to correlate the signal with itself, rather than with a known set of samples. This was done due to a significant difference in the performance of these two methods when they were tested against each other, as described in Section 4.5.3.

Multiplication

Due to the nature of the calculations involved, being able to do multiplications quickly becomes an important part of the cross-correlation. For every incoming word of data, 60 multiplications are performed. I therefore wanted to make sure I enabled multiplication to be done as efficiently as possible.

My initial approach was one of assuming that multipliers would be slow, and that the multiplication should, preferably, be done using adders. I therefore described a custom Q15 multiplication module. The module performed the standard shift-and-add method of multiplication. The reason I wanted to describe this module myself was to have control of the number of significant bits used in the calculation. For the purposes of signal detection only rough calculations are needed. Through making my own multiplier with a configurable amount of significant bits, I could drastically reduce the amount of adder operations that had to be done, compared to the alternative, with calculations synthesised to be performed with maximum accuracy.

I successfully designed such a multiplication module, however, the number of flip-flops and adders required to perform all the necessary calculations was still undesirably high. I

was then told by Oscar from the PASTA group that the FPGA which my modules would be running on had the capability to perform this number of multiplications very quickly. While perhaps taking a little longer, the use of these built-in multipliers would save a significant amount of hardware in terms of flip-flops and adders. This made the tradeoff an obvious one to make. I therefore adapted my Q15 multiplier to simply multiply the two inputs, treating them as regular two's complement numbers, then right shift the result by 15 places to truncate it back into a Q15-formatted number.

4.4.3 Energy

As shown in Figure 4.2, the energy module takes as an input the 31 bins calculated by the cross-correlation module. It calculates the signal energy of these bins, treating the value in each bin as a sample of a signal. Using these samples, it calculates the energy of the signal averaging out the absolute values of the signal samples.

Due to efficiency concerns, and corner cases, my module does a few additional things which go beyond regular energy calculation. Rather than taking the average value of the input signals, which would involve dividing by 31, I divide by 32 instead, as this is done using a simple 5-bit shift. Also, to avoid arithmetic overflows when an average of eight of these results are calculated in the peak detection module, the value is shifted by an additional 3 bits. This can be done, as the module is only looking for relative differences between the energy values. Therefore it does not matter how much a value is being shifted, as long as all values are being shifted by the same number of bits.

Adding Absolute Values

As described above, I utilise the average of the absolute values of the signal samples. The process of adding together the absolute value of multiple Q15 numbers is more complicated than it might appear to be. The nature of a Q15 formatted number is similar to that of two's complement: the leftmost bit represents a negative number, and the bits to the right of this represent positive numbers which are added to the negative number. This leads to a lack of symmetry between positive and negative numbers. As opposed to formats using a sign-bit, in which the absolute value could easily be found by ignoring this bit, the method of inverting the sign of a Q15 number involves flipping the bits and adding 1, as one would for two's complement.

The method for adding together the absolute values of multiple Q15 numbers therefore becomes quite complicated. Either the original value or the "flipped and incremented" value, conditional upon whether the leftmost bit is true or not, has to be added to the corresponding value of all the other numbers. My initial algorithm for adding the numbers was as follows,

for each number:

1. Perform a bitwise AND across all bits of the original value and the inverse of the leftmost bit. Add this to the cumulative result.

2. Flip all the bits of the original value, add 1. Perform a bitwise AND between all bits of this value and the leftmost bit of the original value. Add this to the result.

The bitwise AND with the leftmost bit will determine whether the number should be added at all. If the number is positive, the leftmost bit will be 0. If the number is negative the leftmost bit will be 1. The method described above would therefore add only the absolute value of the original number to the final result.

This method was functionally sound. However, note that step 2 requires two additions: one for incrementing, and another one for adding the result to the total. Hence three additions were carried out for each of the 31 numbers which needed to be added together. This resulted in an adder tree which was too large to complete during the time available in a clock cycle.

In considering how to solve this problem I realised that, as one of the two values would always be zero, I could use a bitwise OR operation on these two values, rather than having to add them together. This brought the adder requirement down to two additions per number, cutting the additions required by one third, resulting in an overall calculation that could be completed within the timing constraints.

I spent some time searching for solutions to calculate signal energy efficiently using hardware, however this still seems to be an unsolved problem. Solutions have been proposed for calculating signal energy efficiently, but these apply algorithms that are unsuitable for implementation in hardware.

4.4.4 Peak detection

In the peak-detection module, an average is calculated using 8 signal energy values. The module buffers the last 16 values to enter the system, and calculates the average of the 8 least recent ones. When a “current” sample is greater than two times this average, and is above a set floor, a peak is detected, and an output is set high, signalling to the rest of the system that the incoming signal represents a data packet.

Determining this floor was done through trial and error during the testing stage. If the floor was set too low, false positives would be detected, as anomalies in the random signal could look similar enough to create a minor peak. If the floor was set too high, potential valid positives would not be picked up at all. In the end, choosing a floor ended up being a relatively simple task, as the valid peaks tended to be much larger than the false ones, as illustrated in Figure 4.4.

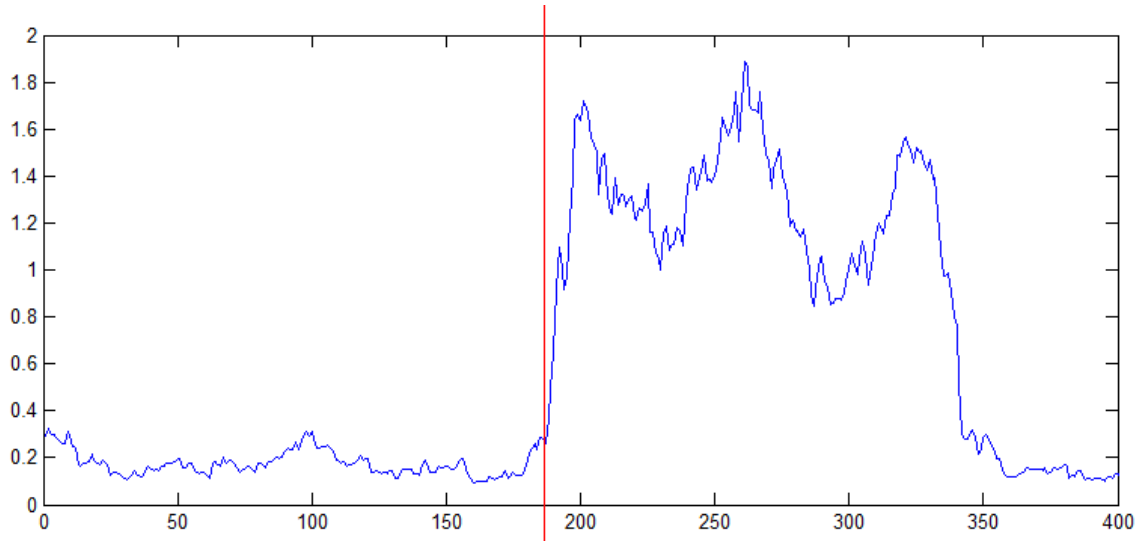


Figure 4.4: A Matlab simulation of the resulting signal energy when the cross-correlation method described in this chapter is carried out on a test packet. The preamble begins at the vertical red line. Note that the method used to generate the data in this figure approximates rather than emulates the method used in hardware. It is included to give the reader an impression of what the signal energy will look like in the system, rather than showing the exact value.

4.5 Testing

There were a few different things about this module that needed to be tested and verified. Primarily, I wanted to make sure it actually worked, and that the module was able to detect data packets under ideal conditions. However, as discussed above I also made this module configurable with respect to how many significant bits were used in intermediate calculations. I therefore wanted to run a range of tests to see how space-efficient I could make my module, without compromising its functionality.

4.5.1 Generating the test data

To carry out the test, I created 10 000 sample inputs, each representing an input from the ADC. The test samples were generated in Matlab, then converted to the Q15 format, again using Matlab. Initially 10 000 samples' worth of random noise was generated, using a normal distribution between 0.999... and -1, which are the upper and lower bounds of the Q15 number representation. To obtain different levels of noise at different places in the signal, I multiplied ranges within the array of samples as follows:
Samples 0 through 2000 were multiplied by 0.3, "Low noise"
Samples 2001 through 6000 were multiplied by 0.5, "Medium noise"
Samples 6001 through 10 000 were multiplied by 0.85, "High noise"

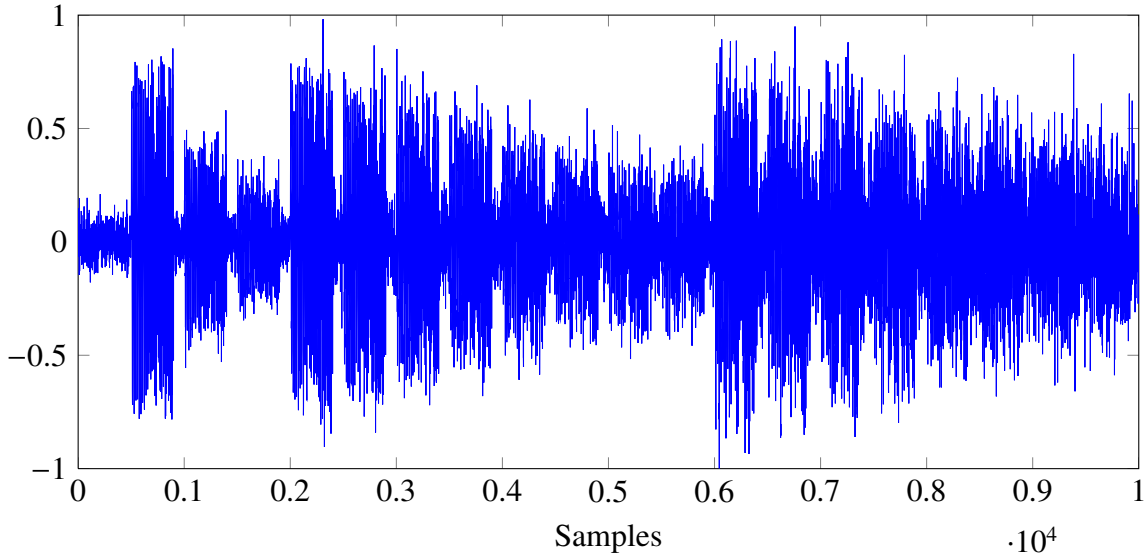


Figure 4.5: This is the 10 000-long set of samples used to test the peak detection, and later to test the signal reception. It can be seen that the magnitude of noise, and of the signal, varies throughout the signal. The process behind creating the signal is described in section 4.5.1.

Note that "low noise" was applied to less samples than the higher noise levels. This was done deliberately, as I expected the low noise to have little effect on the quality of the signal detection.

I then generated a sample data packet with a length of 400 samples. The first 160 of these samples represented the preamble, and the remaining 240 samples were generated randomly using a uniform distribution between 0.999... and -1. Once I had this data packet, I added it on to the noise at intervals of 500 samples. To simulate signals sent at varying signal strengths, this packet was multiplied by different values before adding it to the noise. Finally, the entire signal was normalised so that it would be contained within the limits that can be represented by the Q15 number format. The final 10 000-sample test-signal is shown in Figure 4.5. These samples were converted to the Q15 number format using Matlab, and exported to a text file with each sample starting on a new line.

4.5.2 Using the test data

The generated test data had to be simulated in Verilog. To facilitate this I modified the Verilog top file, which handles all inputs and outputs, including the input of incoming data. I made a counter which, starting from 0, incremented by 1 on every clock cycle. I then wanted to insert my test data so that a new sample was input into the system for every incrementation of this counter. To do this I needed Verilog code which would assign a new value of the sample data to the input for every value of the counter. As I had to deal with 10 000 samples, doing

this manually would have been impractical. Instead I made a short Python script that would make a case-statement that would assign a new value, taken from the generated samples, to the register *numberin* for every value of the counter:

```
f = open('samples.txt', 'r')  # Opens the 10 000 line file of samples
counter = 1                   # Initialises the counter
print "case (counter)"        # Prints the start of the case statement
for line in f:                # Generate line of Verilog for every input
    print "%s: numberin <= 16'b%s;" % (counter, line)
    counter = counter + 1     # Increment the counter
print "endcase"               # End the case
```

Writing the script's output to a file resulted in a 10 002-line text file which I copied over to my working directory on *pasta2*. Thanks to the .vpp file format, I was able to include this in the top file without having to include all the code in the file itself. Finally I assigned the input to the module to be equal to the value of *numberin*. This gave a simulated input to my module containing the samples I had generated in Matlab. By applying this, I could see how my system as a whole would react to these inputs.

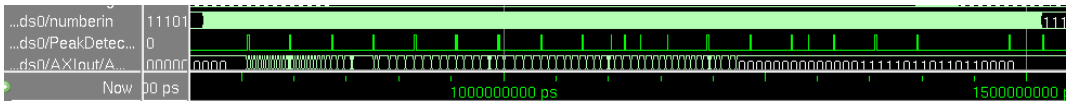


Figure 4.6: Modelsim showing the detection of preambles.

4.5.3 Test results

The results were good, overall, and my module worked with all the expected scenarios. Figure 4.6 is a screenshot from Modelsim, showing the preambles being detected by the system.

Calculating SNR

For my test results to have any meaning, the SNR in the case of detection has to be calculated. A common method for calculating the SNR of non-periodic signals, as demonstrated in [11] amongst others papers, is as follows:

$$(SNR)_{db} = 10\log_{10}^{\frac{\sigma^2_{Signal}}{\sigma^2_{Noise}}}$$

I used Matlab to calculate the variance of the noise at the point where the signal is being detected. The variance of the preamble used at that point in the test data was also taken. I then calculated the SNR using the equation above.

Detection Testing

Sample beginning at	Noise Variance	Signal Variance	SNR (dB)	Detected?
500	0.004	0.239	17.8	Yes
1000	0.005	0.074	11.7	Yes
1500	0.005	0.026	7.2	Yes
2000	0.015	0.239	12.0	Yes
2500	0.016	0.189	10.7	Yes
3000	0.016	0.144	9.5	Yes
3500	0.014	0.106	8.8	Yes
4000	0.015	0.074	6.9	Yes
4500	0.015	0.047	5.7	Yes
5000	0.017	0.036	3.3	Yes
5500	0.017	0.026	1.8	Yes
6000	0.040	0.189	6.7	Yes
6500	0.048	0.144	4.8	Yes
7000	0.035	0.124	5.5	Yes
7500	0.041	0.106	4.1	Yes
8000	0.043	0.074	2.4	Yes
8500	0.039	0.047	0.8	Yes
9000	0.033	0.026	-1.0	No
9500	0.040	0.018	-3.5	No

Table 4.1: The results of the packet detection tests.

Table 4.1 shows the results of this test. As can be seen, signals with a SNR as low as 0.8 dB are detected by the signal detector. This is significantly lower than any SNR at which papers expect VLC to be usable [12] [13], so I conclude from the test that the detection is good enough to handle all required cases.

Significant Bits Needed

For this module to work, a substantial number of values needs to be stored in order to detect the peaks. Further, a large number of calculations are carried out. Any reduction of significant bits used to store numbers would therefore yield large savings in terms of hardware. I designed the module using a parameter that could be varied to adjust the number of significant bits used. In this way I could easily test my module with different numbers of significant bits. I expected the module's performance to degrade gradually along with the reduction of significant bits. Surprisingly, this is not what happened.

As the number of significant bits used in the calculations was reduced to below 23, the system stopped working almost entirely, with false positives being detected constantly. No difference in performance was observed with the use of 23 bits, or any higher number. It

appeared that 23 bits was the resolution required for the computations to be sufficiently accurate. I therefore concluded that carrying out the multiplications and additions limited to 23-bit intermediate calculations was sufficient for the purposes of my system.

4.6 Conclusion

The detection of incoming packets is an important function that needs to be performed by the hardware receiving data. This chapter has described the design, and the challenges faced, in making this hardware. Detecting the signal is largely a mathematical operation, which is also reflected in the challenges that were encountered. The method used for detecting the preamble is purpose built, and exploit properties of periodic signals as well as the theory behind cross-correlations.

It has been shown that the proposed solution works sufficiently well down to SNR values of less than 1. Papers analysing the SNR in VLC systems such as [13] and [12] agree that the SNR in VLC systems should be as high as possible, and these papers use 19.5 and 15 respectively as arbitrary lower bounds for a VLC system. It can therefore be concluded that the SNR performance achieved in the detection module is unlikely to become a bottleneck in any VLC system currently envisioned.

Once a data packet has been detected in the incoming signal, this packet has to be written to memory. The next chapter describes how this is done.

Chapter 5

Signal Reception

One task of the system is to receive the incoming data signal and write it to the system memory through the AXI protocol. This chapter describes the specification which is needed to fulfil this, and reports on the final design of this module.

5.1 Introduction

One of the most crucial functions of a module that would allow a SoC to interact with Visible Light Communication is the part of the system that accepts data through VLC. One major advantage of VLC as a technology is its ability to transmit data at a very high throughput. Thus, any system using VLC to take advantage of this would require the capability to accept data arriving at a very high speed.

5.2 Specification

The module for receiving signals must accept a stream of data arriving through the ADC, and write this data to memory using the AXI protocol, in order to communicate with the AXI interconnect. The module must be able to use parameters which dictate what address it should write to, and it needs to write the incoming data to the correct memory addresses in the correct order. It also must ensure that the entire packet is written to the memory, and must therefore start writing data which has been stored in a buffer prior to a packet being detected, as well as writing some amount of additional data once the size of a full packet has been written.

In addition to this, the module must report back to the system, informing it of how much memory remains which can be written to.

5.2.1 Test to fulfil

The module will be tested using generated samples. These samples will be fed into the system, and the test will be whether the module correctly writes them to the correct addresses in memory, and whether this is done at a reasonable speed. Due to the impracticality of inspecting the actual contents of the memory in Modelsim, I will instead monitor the AXI signals received by the Slave. This will show what data is being written to the Slave and what address it is being written to, from the point of view of the memory itself. As the memory block of the system has already been used extensively, it can be assumed that it will handle incoming AXI signals correctly. Therefore I consider observing the incoming signals to be a valid way of verifying that the correct data is being written to the memory.

5.2.2 Throughput and Clock Frequencies

The specification of this module states that it has to be able to receive incoming data continuously, while writing this data in real time over the AXI protocol. This may seem unrealistic, but because of the clocking of the system, and the bit-width of the data, it can be done. As explained in Section 3.2.2, the system deals with two clocks. Receiving and transmitting samples is governed by one clock, while the AXI interconnect operates according to the other clock. The sample clock is assumed to operate at a maximum of 75 MHz, while the AXI clock is assumed to have a typical rate of 100 MHz. Each sample of the incoming data is 16 bits wide. At a maximum speed of 75 MHz, this means the rate of the incoming data will be $16 \times 75 \times 10^6 = 1200000000$ bits per second, or 1200 megabits per second.

The AXI interconnect operates at a clock rate of 100 MHz, and also transmits data using a width of 32 bits. This makes the maximum throughput through the AXI system, assuming a word of 32 bits was transmitted for each clock cycle, $32 \times 100 \times 10^6 = 3200000000$ bits per second, or 3200 megabits per second. Considering that 75 MHz is a worst-case speed, it can be said that the maximum potential rate of output is roughly three times that of the input. Of course, the AXI protocol will generally not allow a 32-bit word to be written for every clock cycle. For the system to work, it does not have to do this. It only needs to write on average one 32-bit word for every third clock cycle. Testing the system, this was shown to be the case. The AXI interconnect generally worked quickly enough to allow for writing of continuous data. The FIFO acts as a buffer in case the AXI system is delayed, or when a status word is written as described in Section 5.3.4, but on average the system works quickly enough to read data out from the FIFO more quickly than it is being written to.

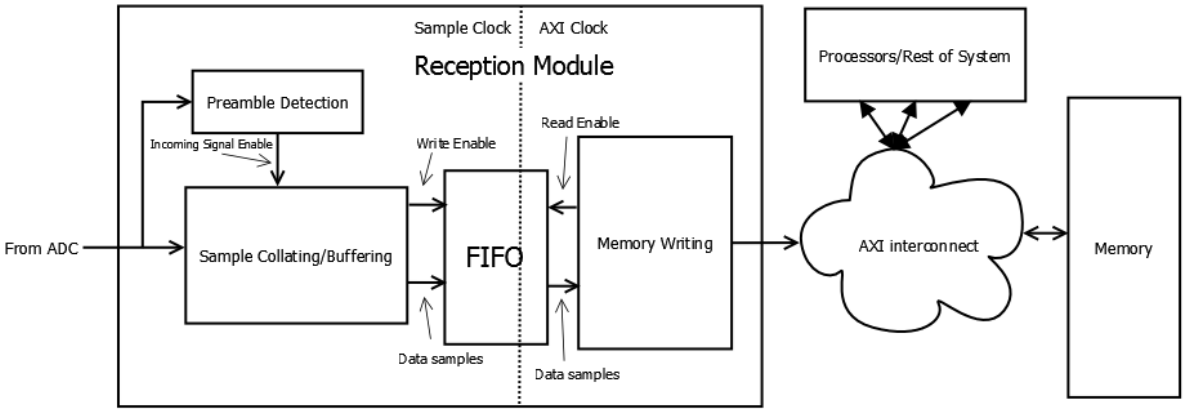


Figure 5.1: Block diagram showing the function of the reception module. From left to right, data-samples are received from the ADC. The data goes into a buffer, and a signal detection module. When a signal is detected, the buffer and further incoming samples are written into the FIFO. Whenever the FIFO contains any data at all, this data is written using a module that interacts with the AXI interconnect. Data is transmitted until the FIFO is empty. Note that separate clocks are being used for interaction with the AXI interconnect, and the reception of the data.

5.3 Reception Module Design

Figure 5.1 shows the construction of the module. The preamble detection is covered in Chapter 4. Even before an incoming signal is detected, the 32 latest data samples are always stored in a buffer. Once the signal detection indicates an incoming signal, the contents of this buffer, and all incoming samples following it, are written into the FIFO. Whenever the FIFO contains any data, this data is written to memory by the AXI module. This section describes how this is done.

5.3.1 Buffering

The final design outlined in this chapter is made using the FIFO, described in Section 2.3.3. The FIFO is a module which is included in the FGPA that my code was to be written for.

The module was originally designed around the use of an extremely large register as a buffer that would store incoming data before it had been written to the FPGA. The intention behind this was that the FPGA would detect that these registers were to be used as memory. This was not the case, and the synthesis process attempted to treat this memory as individual flip-flops, which led to the entire synthesis process freezing due to a lack of memory. To solve this problem, I instead implemented the memory as a large two-dimensional register array.

This was not ideal, as a large amount of hardware was required to do this. However, some memory was needed as a buffer, as the data would be arriving at a constant rate, while the AXI protocol works on a best-effort basis with regard to throughput. The main challenge in

using this buffer was keeping track of the pointers. It was implemented as a circular memory, where data was written sequentially into it, and read sequentially out of it. Keeping track of these pointers required considerable additional logic.

Another problem in using the register array was the transfer of data between clock-domains, a problem which remained largely unsolved in the original design. A method of doing this had been designed, but was only usable provided that the register array was large enough to contain an entire packet worth of data. In the best case, this would have led to a very large requirement in terms of hardware, while it in the worst case would have imposed a serious system limitation in terms of the packet size.

Later, the FIFO was taken into use by my system, making the existing solutions for pointers and clock-handling obsolete, as well as eliminating the packet-size bottleneck in my system entirely. The disadvantage in using the FIFO is that multiple AXI transactions cannot be carried out at the same time. This is explained further in Section 5.3.3.

5.3.2 Handling Incoming Signals

Signals are received by the system in the form of 16-bit samples. The 32 most recent samples are stored in a 32×16-bit register array, which works like a small buffer. This is done by using a simple counter, counting from 0 to 31, which increments by 1 for every incoming sample, then resets once 31 has been reached. The incoming sample is written to the corresponding location in the register array. This happens constantly, even while incoming signals are being written to memory.

Initially this buffer was implemented using a large, single-dimension, register. This was done under the assumption that the synthesis process would detect its usage and treat it as a memory block. However, during synthesis it turned out that, rather than being treated as a general-purpose memory, 512 individual flip-flops were used to store the individual bits. As this was unnecessary, the code was rewritten to use a two-dimensional register array instead, which was synthesised appropriately so as to be stored in a more efficient way in the FGPA.

When a packet is detected, and data should be written to memory, the oldest data in the buffer, that is, the data at buffer position $x+1$ where x represents the counter, is written to the FIFO during the clock cycle prior to it being overwritten. In this way the data being written to memory is always the data that arrived into the system 31 cycles prior to the data that is currently incoming.

These signals eventually need to be transmitted to the memory in chunks of 32 bits, hence the 16-bit samples have to be collated. Rather than writing the 16-bit samples directly into the FIFO, they are first grouped together in a 32-bit register. This is done using a single-bit counter. When it is zero, a 16-bit sample is written into positions 0-15 of the register. When

the counter bit is 1, the sample is written into positions 16-31. Finally, for every two 16-bit samples that have been written into the 32-bit register, the whole register is written into the FIFO, ready to be written to memory through the AXI protocol.

5.3.3 Writing over AXI

In terms of what it is required to do, the specification of the state machine for writing data to the memory over AXI is quite simple. Whenever there is any data in the FIFO, this data should be written to memory. Thanks to the asynchronous nature of the FIFO, this can be done fully independently of data being written into it. The writing of data to memory through AXI is therefore initiated as soon as the incoming data packet has started being written into the FIFO. A state machine that is used to write to memory over AXI handles the AXI transactions by asserting the correct signals when appropriate. In doing this, the machine has to make sure that it only writes to the memory locations it has been instructed to write to by the *startaddress* and *endaddress* parameters that it receives from the top module. It also needs to keep track of which address it is currently writing to, and which address to write to next.

As an input to the module, an address is accepted through the *processedaddress* signal. This has to be an address in the range between *startaddress* and *endaddress*, and can always be accepted by the module. This can be asserted by the SoC at any time, informing the module of what address in memory it has processed. This allows the module to re-use memory addresses which have already been written to, but have later been processed by the system. The writing is done in a continuous, circular manner. Once the end of the allocated address space has been reached, the module starts writing from the beginning of it, given that the *processedaddress* signal has indicated that this data has already been processed.

The state machine used for this is pretty straightforward. It transmits data through bursts consisting of 16 transfers of 32 bits of data. It continuously monitors the status of the FIFO, and whenever the FIFO has any data, this data is read and transmitted. A counter keeps track of how many bursts have been transmitted in a transaction, and handles the AXI formalities of declaring the ID of these transactions, the “last data” signals, and so on.

Only one write transaction is carried out at any one time. It would technically be possible for multiple AXI transactions to be carried out in parallel, which would result in a higher throughput. However, a limitation with this approach is that all the bursts of data within a transaction will be written to consecutive addresses in memory. For parallel write transactions to be carried out, this would require one transaction having access to the data which was to be written to the memory ahead of the last burst in another transaction. Due to the nature of a FIFO, this is not possible, as data can only be read out sequentially. In the course of testing the system, it was observed that dealing with one transaction at a time was sufficiently fast

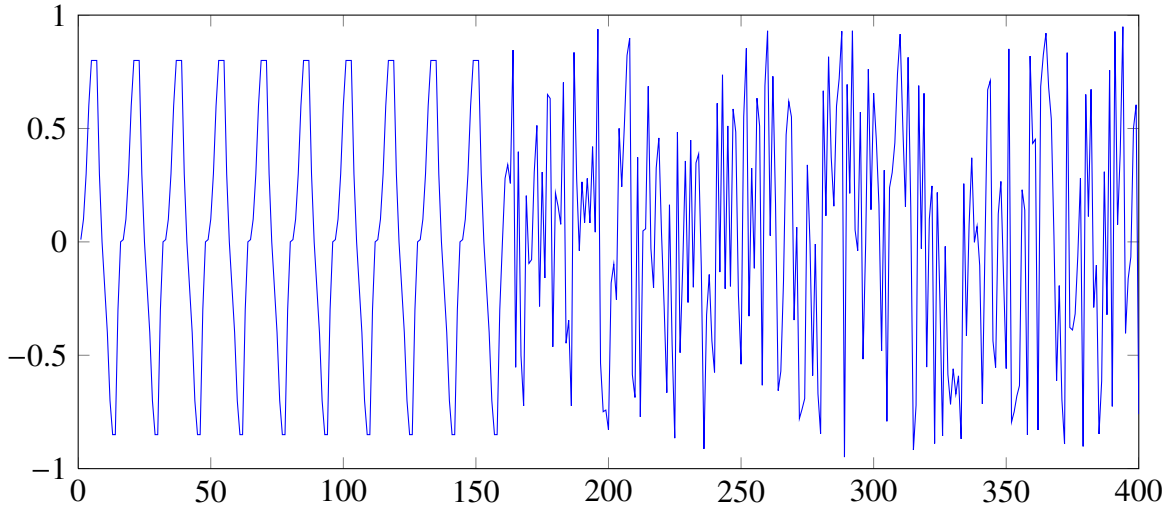


Figure 5.2: The packet used as dummy data for testing the reception module. The periodic 160 samples at the start make up the preamble. The x-axis represents the sample number, while the y-axis represents the value of a given sample.

for writing data to memory.

5.3.4 The Status Word

I needed a way to tell the system whether enough memory had been allocated for my reception module to write to, and to store this information in a location where it could be seen by the system as a whole. I solved this by making my module write a status word to the memory. This word, consisting of 32 bits, contains information on how much space remains in the memory for the module to write to, as well as information as to whether the system is set up correctly, and whether it is dropping packets or not. The typical use of this status word is for the system to have an indication of when it is running out of memory, and when it needs to transmit a new *processedaddress* to the module, effectively freeing up space which can be re-used.

The module contains a separate state machine for writing this status word. The rate at which it is written is configurable. It should be written as rarely as possible, as doing so takes up clock cycles which would otherwise be used for writing data. How often it is needed depends largely on the amount of space available in memory.

5.4 Testing

What needs to be tested for the reception module is whether the incoming data is correctly received, and written to the memory. As the system performance in terms of SNR was tested

in Chapter 4, the tests relevant to this module were as follows:

- Whether the data written to memory was correct
- Whether the data was written to memory quickly enough
- Whether an entire data packet worth of data has been written

Whether the written data is correct can be verified in the simulator by checking that the data actually written to the memory matches the data that was given as input to the module. Whether the data is being written quickly enough can be verified simply by having the module write data to memory continuously, observing whether this is possible without the FIFO overflowing. Whether sufficient data is written is simple to verify by monitoring the addresses being written to by the first and the last transaction. If the range of addresses is equal to the length of the expected amount of data written, the module is writing the appropriate amount.

5.4.1 Creating the Test Packet

The important part of creating a packet to test the system was to make a packet that would enable the manual verification of the data written. The test packet had to begin with a preamble to simulate the preamble on an actual packet, and the data following this preamble had to be known, so that it could be verified. I decided to use the test data which I had already generated with the purpose of verifying the packet detector in Chapter 4. While the data contained in these test packets is random, it can still be used to verify that the data coming into the system is identical to the data which is written to memory.

5.5 Test Results

Figure 5.3: Modelsim showing the data coming in to the system as a peak is detected. This represents the data that should be written to memory.

Figure 5.3 shows the data being received by the module as a peak is detected. This is the data to be written to memory. For clarity, the first 8 hex-values shown are:

`0C01  098F  12C9  2407  4168  5763  4B2C  5812`

This is compared to the data which is written to memory, shown in Figure 5.4 The data shown in Figure 5.4 as being written to memory is repeated below, for clarity:

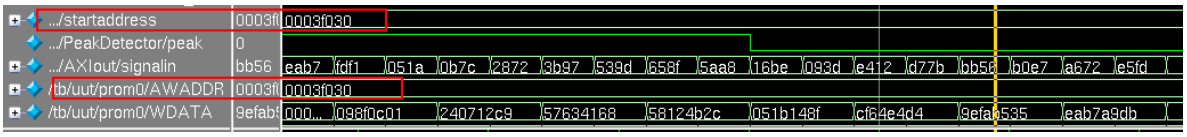


Figure 5.4: Modelsim showing the data which has been written to the memory. The two bottom signals represent the actual signals received by the memory module, the bottom line representing the data that is written to memory. Note that the address being written to matches the start address given to the module as a parameter, marked by red rectangles.

```
098F0C01    240712C9    57634168    58124B2C
```

Comparing this to the data which arrives at the system, shown above, it matches up. Note that the order of the 16-bit samples is switched when collated into a 32-bit packet. This is required so that the data being received by the system first will be written into the memory first, as the least significant bit is stored at the initial address.

Next, it must be tested whether the data is being written to memory quickly enough for a packet to be continuous. This can be seen by looking at the rate at which the FIFO is being emptied, i.e. the rate at which data is being read from the FIFO and written into the memory.

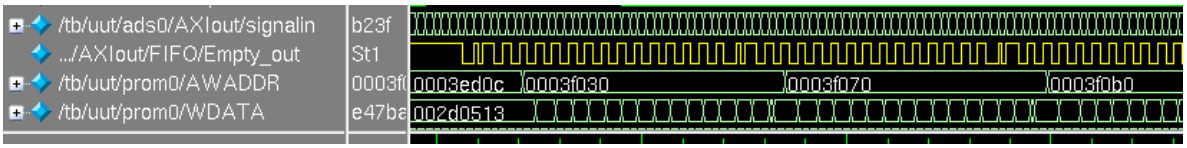


Figure 5.5: Modelsim showing the empty-signal, in yellow, being set high much of the time as data is written from the FIFO.

Figure 5.5 shows the FIFO declaring that it is empty much of the time. It can be seen that it occasionally stays non-empty for a longer period of time. This is when a new burst is being initiated. However, as multiple transmissions are then carried out in a burst, the system normalises and the FIFO is once again empty.

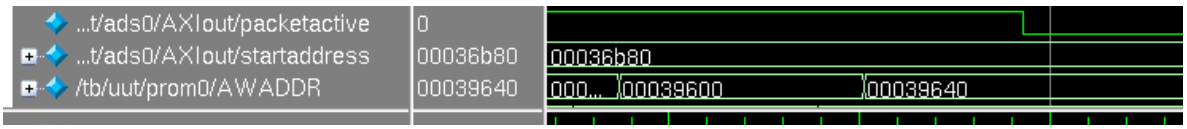


Figure 5.6: Modelsim showing the first address written to (*startaddress*), and the last address written to when a packet is received.

Lastly, it needs to be shown that enough data is written to memory to store an entire packet.

I set the packet length to be 10240 bytes long. In addition to this comes the 64-byte buffer of data prior to the detected packet, as well as the cushion of extra data which is written to

ensure that the entire packet is written to memory. The amount of extra data written in total, including the buffer preceding the detected data, is configurable. In this case I set it to be 6 segments of 128 bytes. This adds up to $10240 + (6 * 128) = 11008$ bytes.

Figure 5.6 shows the *startaddress*, i.e. the first address written to by the module, as well as the last address that memory has been instructed to write to. As can be seen from the figure, *startaddress* is `00036B80` while the last base-address written to is `00039640`. This last address represents the address at which the transaction's first data transfer is written to. 16 transfers of 32 bits, or 64 bytes, follows this. Hence the address following the location of the last written data will be `00039680` The difference between these two addresses, in hex, is `2B00`, or, in decimal 11008. Each of these addresses represents a byte.

This matches the expected amount of data, hence it can be concluded that the module writes the amount of data it is told to write by the parameters.

This also demonstrates the module's ability to successfully receive and write data packets which are much larger than the size of the FIFO.

5.6 Conclusion

This chapter has described the design of the module that writes incoming data to the memory. The module receives 16-bit samples from the ADC, collates these into 32-bit words and writes these words into a FIFO. Whenever any data is present in the FIFO, this data is written to memory. The module regards three parameters, telling it what memory address it can start writing to, what memory address it may not write past, and what memory addresses it may re-use. The module keeps track of these through maintaining a pointer deciding where in memory data should be written to. The module was then tested with regard to whether it wrote the correct data, whether it wrote data quickly enough to handle a constant stream of incoming data, and whether it wrote data for long enough to write a full data packet, plus a cushion, to memory. The module successfully passed all three of these tests.

The next level of functionality to be added to the system is the ability to transmit data. The following chapter describes how the hardware required for doing this was designed and implemented.

Chapter 6

Signal Transmission

This chapter describes the design behind the part of the system responsible for reading data from memory, and transmitting it. In doing this data has to be read from memory, buffered, then transmitted. This chapter describes how this was implemented, and finally how it was tested.

6.1 Introduction

Part of the finished system's required functionality is to read data from the system, and transmit that data out through a DAC. This is achieved by reading the data packet from a given address in memory using the AXI protocol, buffering it, and passing a full packet on to the DAC which transmits it in the form of light. As part of this process a preamble consisting of 160 samples is added to the start of the packet.

This chapter describes how I did the above, the design decisions I made during the process, and the resulting module design.

6.2 Specification

Once finished, the module will read data from memory and transmit 16-bit chunks of this data on the positive edges of the sample clock. The trigger for transmitting data will be an incoming signal from the top module, along with an address in memory to read data from. Once this has occurred, the module has to read data from memory over the AXI interface and transmit an entire packet's worth of data continuously, preceded by a preamble.

6.2.1 Test to fulfil

A C-program will be written which will write data to a range of addresses in the system memory. The program will then send the module an instruction to transmit this data. The test will be considered successful if the module transmits the correct data, preceded by the preamble.

6.3 Preamble

Part of transmitting a packet of data is adding the preamble on to the actual data which is to be transmitted. The purpose of doing this, and the theory behind it, is covered fully in Chapter 4.

The specification for the task is simple: a set pattern, consisting of 160 samples, is to be transmitted prior to every packet of actual data that is transmitted. To make my design as modular as possible, I created a separate module with the purpose of transmitting this preamble.

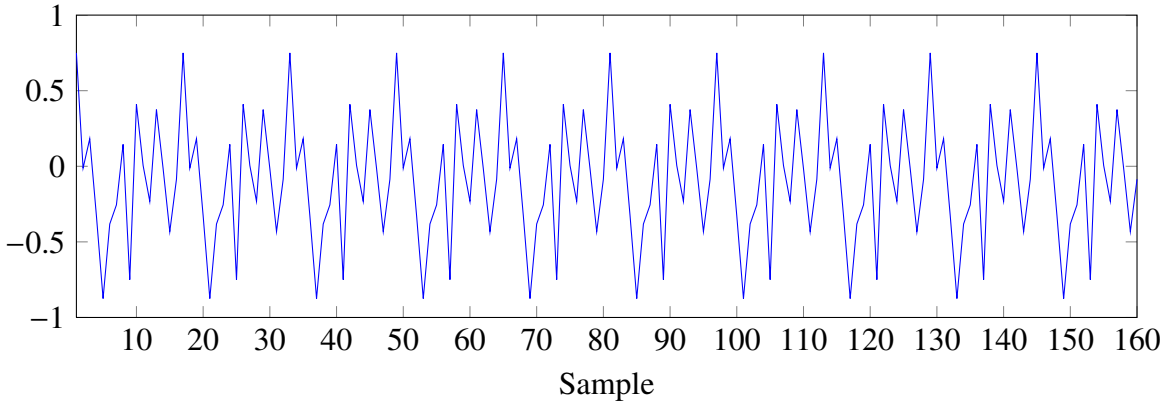
6.3.1 Design of the preamble signal

The first thing involved in the design of the preamble was to design the signal to be transmitted. No particular specification had to be conformed to for this design, other than that the signal had to be 16 samples long. I first made a preamble in which these 16 bits were almost entirely random in nature. Later, while testing the detection module which had as its purpose to detect the preamble, I found that this preamble worked, but not very well. Consequently, I reviewed the preamble signal, and realised that, due to how a cross-correlation is taken, a smoother signal would give a more reliable peak when the preamble was to be detected. I therefore modified the preamble to be much smoother, which resulted in more reliable detection.

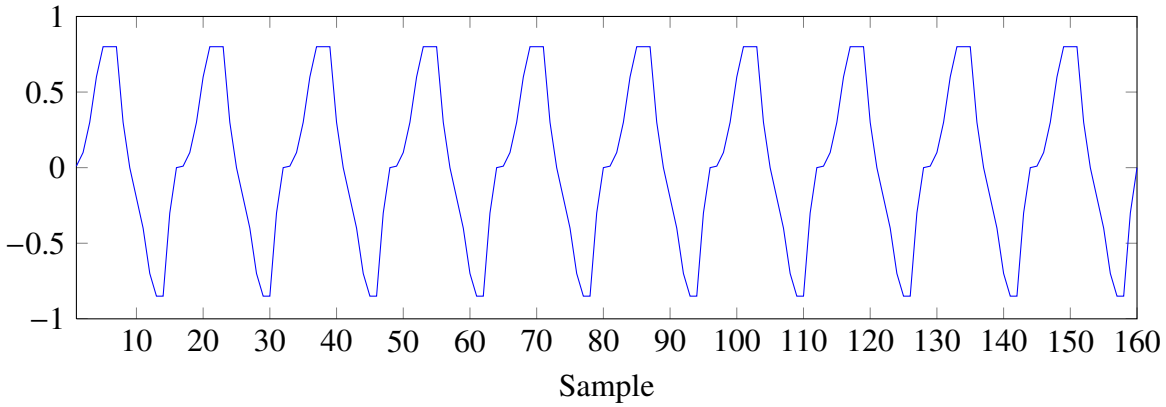
The reason for this is that a smoother signal will have a greater degree of overlap when compared to a slightly offset version of itself. With a non-smooth signal, even a small offset could lead to almost no overlap at all.

6.3.2 Design of the Preamble Module

The specification for this module was simple enough: when the main transmission module is ready to transmit data, transmit the preamble. As the preamble consists of 160 samples making up a signal, this signal needed to be embedded into the module. However, by design, this signal consists of the same 16 samples repeated ten times over. I exploited this by embedding



(a) Original preamble



(b) Revised preamble

Figure 6.1: The original, and the revised preamble.

these 16 samples in the module, and creating a state machine that transmitted the samples ten times. A wire for each sample is declared, and assigned to the 16 samples of data in the preamble. The state machine has 16 states for transmitting data, each state transmitting one of these samples. The state machine will wait in a wait-state, start transmitting data when enabled, and then go into a finishing state when a counter indicates that all samples have been transmitted ten times. Once the last of these 160 samples has been transmitted, a "ready" signal is asserted to indicate that the transmission of the preamble has finished. While my current transmission module does not make use of this signal, instead relying on counting the number of samples sent, it is present to aid the potential re-use of the design, for example if a system wanted to use a dynamic number of samples in the preamble.

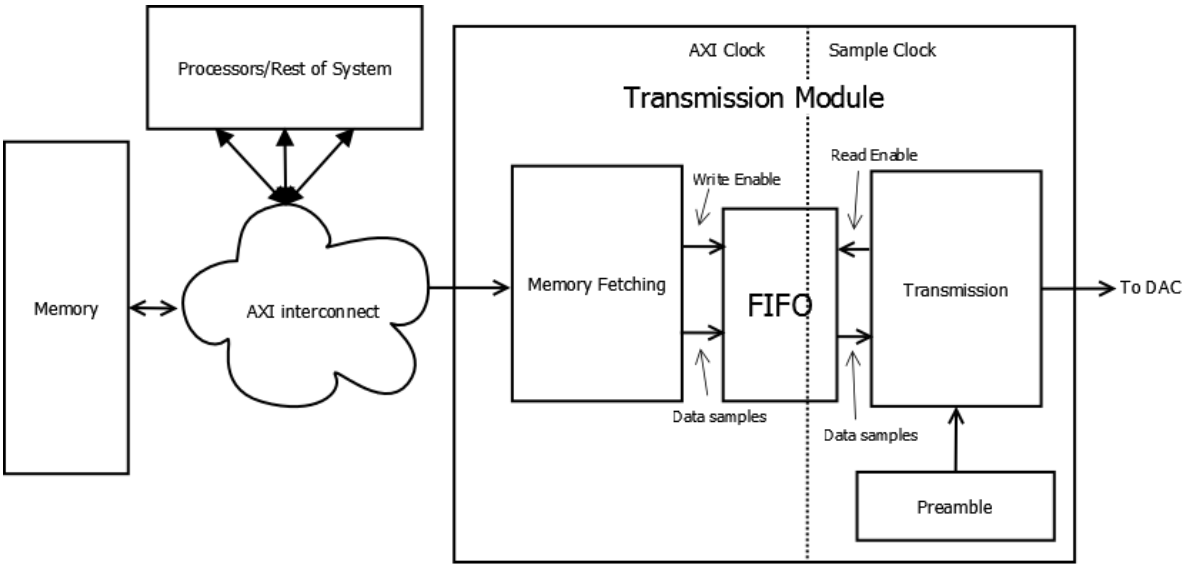


Figure 6.2: Basic functionality of the transmission module. From left to right, data is fetched from memory through the AXI interconnect. The data is then written into a FIFO. Once the FIFO is almost full, the preamble is transmitted, followed by the data in the FIFO. This continues until the FIFO is empty. Note that separate clocks are being used for interaction with the AXI interconnect, and the data transmission.

6.4 Transmission Module design

This section describes the actual work involved in making the module, outlining the design decisions I made, why I made them, and how I implemented them. Throughout the module, the size of a packet is configurable through a parameter which is included in the module design. This allows the module to work with any size of packet, provided that this has been correctly configured prior to the synthesis of the module.

6.4.1 Buffering of the signal

As part of the design of the module I had to consider how much of the packet I wanted to buffer in the module itself. Buffering data takes up a significant amount of space, and it would therefore be desirable to buffer as little data as possible. Initially I considered whether *part* of the packet could be buffered and whether I could then, based on the expected time taken to retrieve data from memory through the AXI protocol, transmit the start of the signal, re-using the space needed to buffer this data.

However, while AXI works quickly, it is not a reliable protocol in terms of throughput. While the simulator may work at a consistent speed, in practice there are no guarantees for how quickly data will be delivered. When transmitting a packet of data, every sample of the packet has to be transmitted back to back. Due to this, my initial approach was to buffer an

entire packet of data. I implemented this using a register which was large enough to contain an entire packet. This register was declared dynamically, according to the size of a packet as set in the configuration file.

The upside of this approach was perfect stability. As a packet would never be transmitted until the entire packet was buffered, a situation would never be encountered where the data to be transmitted had not yet been fetched from memory. The obvious downside lay in the amount of space that had to be set aside to do this buffering. This approach worked during testing. Unfortunately the amount of memory needed for buffering put a practical limit on the size of the packets that could be handled by the system.

I decided to attempt rewriting the module using a FIFO block, as this would make the code much simpler. It would also lead to better on-board memory management, as the FIFO is purpose built for doing just this. I rewrote the module, and successfully used the FIFO to buffer data. However, the size of the FIFO is limited, and this put a hard upper bound on how big a packet could be. This upper bound was too low to be acceptable, so I faced the choice between reverting to my original, space-inefficient, method, or re-thinking the way in which I buffered data.

I settled on partial buffering of the data. While the FIFO is not large enough to contain an entire packet in most cases, it is large enough to hold a significant amount of data. As detailed in Section 3.2.2, the system will normally work in a way which means the data is written into the FIFO quicker than it is read out. While this is not reliable, i.e. in situations in which the data is not fetched quickly enough from memory, the FIFO still works as a sizeable buffer, enabling the transmitter to continue sending data, despite problems that could potentially occur in fetching data from memory.

This solution turned out to work quite well. While not entirely reliable, a delay in memory fetching large enough to stop the module from functioning correctly is likely to be symptomatic of a severe system-wide problem that would probably overshadow the malfunctioning of the transmission module.

6.4.2 Reading from AXI

The main function of the module is reading data from the memory through the AXI interface. In order to do this my module acts as an AXI Master, sending out a request to read from the memory address given by an input signal. The module goes on to read the packet from this address.

When requesting data through AXI, an address is asserted over the AXI channels from which data is requested. Following this request, there is a pause before the Slave accepts the request, and starts serving the Master. This is time during which nothing happens as far as

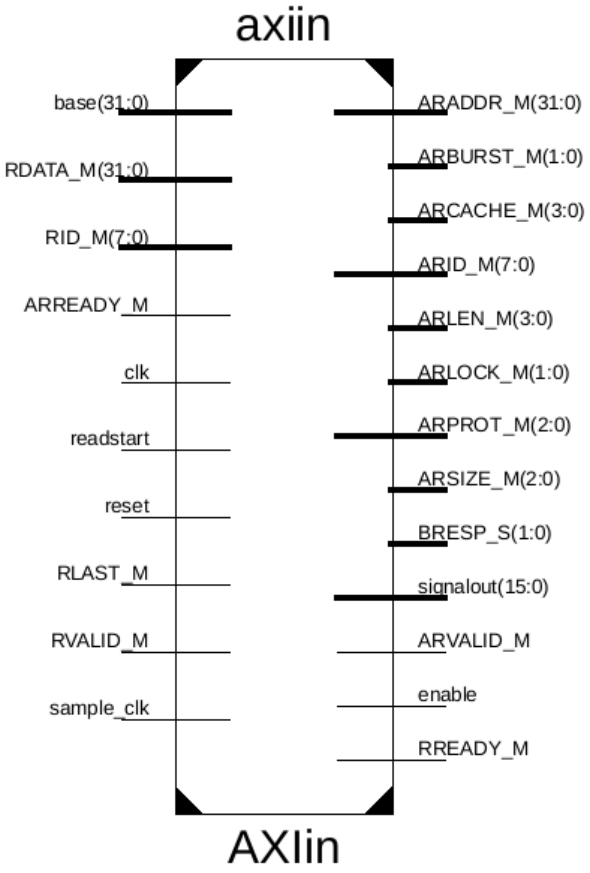


Figure 6.3: The standalone AXIin block fetching data from memory, and transmitting it. The ALL CAPS inputs and outputs are the AXI-signals, used for interacting with the AXI interconnect. The other signals interact with the top module, as described in Chapter 3. Note the two clocks, *clk* representing the AXI clock and *sample_clk* representing the clock used for the transmission of the samples.

the module is concerned, and it would be beneficial if this time could be utilised to actually receive data. I therefore decided to use multiple processes that read data, to enable the module to work even during these pauses.

The AXI protocol works by assigning an ID to each read or write request. In the case of reading, the ID is sent as the AWID signal along with the read request. Each segment of data returned from this request will be associated with the same ID. It is therefore possible for the same module to request data from different addresses at the same time, each request being associated with different ID tags. The module utilises this, and has three different state machines requesting data in parallel.

I use one overarching state machine for keeping track of which address has been read from, and whether any of the three read-processes are idle, as well as the amount of data that remains to read. If data remains to be read, and one or more of the read-processes is idle, this state machine sets a signal high to signify that a read should start, as well as specifying what address should be read from.

I use another state machine to send read requests on behalf of the three separate read-processes. Once a flag is asserted, signalling that data needs to be read, this state machine goes through the AXI motions of sending out a request for data to be read from the appropriate address, and assigns an ID to the transaction. The machine then passes this ID on to an idle read-process in a register. In theory, the read-processes do not necessarily finish in the order they were started. As I made use of a FIFO, order matters. Therefore I also have to keep track of the order in which the buffers of these individual processes should be written to the FIFO, which is accomplished by using a separate register. Each process has a register associated with it containing the values 0, 1, 2 or 3. 0 means the process is idle, while the other numbers represent positions in a queue. Any time a read-process is initiated, the process will take on the number following that of the read process currently assigned the highest number.

Once a read-process has received all sixteen 32-bit chunks of data in a burst, it asserts a "full" flag to signify that it is full. Then the entire buffer is written into the FIFO in 16-bit chunks for transmission. The queue number is then set to 0, and the numbers representing the other processes in the queue, if any, are decremented by one. Once this is done the full-flag is set low, the ready-flag is set high, and the read-process goes into an idle state, ready to receive the instructions to fetch more data.

6.4.3 Triggering and Halting of the module

The module is triggered by the input signal *readstart*, which instructs the module to start reading from the address specified in the input *base*. Once this input is detected, the module starts reading data, and starts transmitting the data using the method described in the previous section. A more complicated decision was when to stop transmitting data. On the face of it, as the size of a packet is known, the obvious solution would be to stop transmitting packets once a given number of segments has been transmitted. However, a potential edge case would be that something malfunctioned in the AXI interconnect, and that the FIFO became empty before a packet was fully transmitted. I therefore decided, that rather than halting the module after the transmission of a certain amount of data, the module will be halted when there is no data left in the FIFO to transmit. The part of the module fetching data from the memory and writing it into the FIFO has a counter associated with it, counting how many bytes that have been read from memory and written to the FIFO. Once this counter reaches the size of a packet, no more data is retrieved. In the regular case the FIFO will then be empty once an entire packet has been transmitted, and the transmission will stop at the end of a packet. In the irregular case of an AXI failure, the transmission of the packet would simply stop short once the FIFO has run out of data to transmit.

6.5 Testing

The first test of this module, testing for basic functionality, was done by using a C-program that first wrote data to memory, then instructed the module to read from the same position in memory and transmit that data. The result of the simulation was viewed in Modelsim, where I manually verified that the data being sent for transmission was the correct data, as well as checking that the correct amount of data was being transmitted.

The C-program stored data in memory in the form of 32-bit words, each word being the binary representation of 2 949 120 + k, where k represents a counter, starting at 1, which increments for every word written into memory. The binary representation of 2 949 120 is `00000000 00101101 00000000 00000000`. The counter will be added on to this. As the data will be transmitted in 16-bit chunks, this means the first four post-preamble samples that one would expect to be transmitted using this input are:

1. Binary: `00000000 00101101` Hex: `2d`
2. Binary: `00000000 00000001` Hex: `1`
3. Binary: `00000000 00101101` Hex: `2d`
4. Binary: `00000000 00000010` Hex: `2`

...and so on, with every alternate sample being the same, and every other alternate sample being an incrementing number.

For the purposes of this test, I set the packet length to be relatively small at 640 bytes. This made it easier to see from the simulation whether the correct number of samples was being transmitted.

Another test which should be carried out is to check whether the module is able to transmit very long packets, containing a much larger amount of data than what can fit into the FIFO. To test this I set the packet size to be large, at 5120 bytes, then ensured that the entire packet had been transmitted.

6.5.1 Test Results

Figure 6.4 shows the transmitted signal, with markers located at the start and end of the signal which is being transmitted. This is done to verify that the outgoing signal is of the correct length. The packet length is set to be 640 bytes, which is equivalent to 320 16-bit samples. Added to this is the 160-sample long preamble. The total expected length of the transmission would therefore be 320 + 160 = 480 16-bit samples.

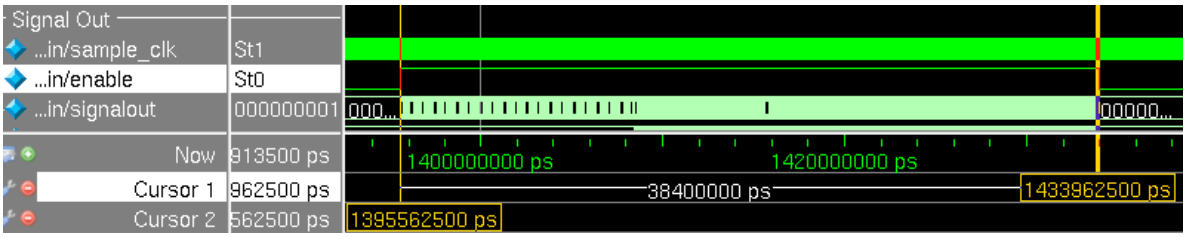


Figure 6.4: First simulation of the transmission model, showing the length of transmitted data. Note that a pattern can be observed in the first part of the transmitted packet. This is the preamble.

In this simulation, one cycle of the sample clock has a duration of 80 000 ps. As can be seen from Figure 6.4, the time taken for the whole transmission is 38 400 000 ps. The number of transmitted samples is therefore:

$$\frac{38400000}{80000} = 480$$

Hence, the duration of the transmission is correct.

The next test to be carried out is the verification of the data that is being transmitted. Due to the relative non-complexity and length of the transmission of the preamble, screenshots showing this in the simulation have not been included in this report. To check that the correct data is being transmitted, I examine the start of the signal after the preamble.

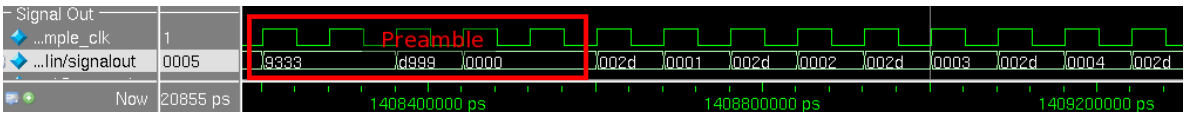


Figure 6.5: Modelsim showing the first transmitted samples in hex.

As can be seen from Figure 6.5, the first samples outputted match the expected values, stated on page 56. The last samples of data also match what would be expected. Hence, I conclude that the module is working as has been specified.

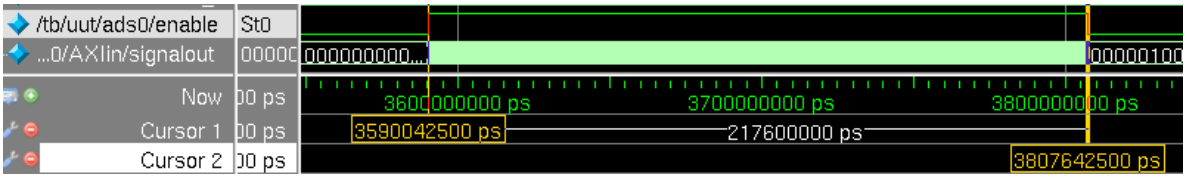


Figure 6.6: Modelsim showing the duration of the transmission of a packet, when transmitting 5120 bytes' worth of samples.

Finally the behaviour of the module was tested when transmitting a very large packet. The transmission is shown in Figure 6.6. The amount of data transmitted is 5120 bytes, equivalent to 2560 16-bit samples of data. Added to this is the 160 sample preamble. The

expected number of transmitted samples is therefore 2720. As can be seen in Figure 6.6, the time between the start and end of transmission of the packet is 217 600 000 ps. The simulation operates with a sample clock cycle of 80 000, hence the number of transmitted samples is given by:

$$\frac{217600000}{80000} = 1720$$

This is equal to the expected number of transmitted samples, so the conclusion is that the module works as expected when transmitting large packets of data.

6.6 Conclusion

This chapter has described the design of the signal transmission module. The module works by reading data from memory over the AXI protocol, and storing this data in the FIFO which acts as a buffer. Once the memory has been written to the buffer, it is then transmitted to the DAC in chunks of 16 bits, which will be converted to varying amplitudes of light. Before the actual data goes out, a preamble is transmitted which allows the data to be detected by a receiver.

Once the design was finished and implemented, the module was tested, both with regard to the accuracy of the written data, and with regard to its ability to transmit a large amount of data. The module passed all these tests successfully.

This chapter concludes the practical aspects of my work. The next chapter goes on to discuss related work, and compares and contrasts this work to what I have produced.

Chapter 7

Related Work

This chapter presents the reader with an overview of available work within the fields relating to my project. The chapter focuses on the aspects of available VLC technologies that I consider to be relevant to my project, and compares and contrasts this material to my work.

7.1 Introduction

The field of Visible Light Communication is under development, and the current applications of the technology are centred around providing a proof of concept. As a consequence, the work related to my project falls loosely into two categories. Firstly, there is the academic work relating to VLC, which is readily accessible through published papers such as [14], [15], [16] and [17] which will be discussed here. This work focuses primarily on transmission and modulation methods of VLC, and their performance in lab environments, which necessarily leads to a limited number of well controlled variables. In academic papers the actual reception of the visible light tends therefore to be built around the experiments, and is not treated as an area of especial academic interest. This is justified, as the practicalities around the implementation of VLC are heavily dependent upon the specific device and application, placing it outside the scope of academic research into the technology. Of other academic work, there is that which covers in more general terms the use of common interconnects and the AXI protocol, such as [18], [19], [20] and [21], which are also briefly discussed.

The other, non-academic, category of related work is that which has been presented commercially as proof of concepts of the technology. During recent years, some larger companies have researched Visible Light Communication, and demonstrated its use in the transmission of data. In addition, smaller companies, such as pureVLC, have been established with the

ambition of commercialising the technology, however, the details of the implementations used by such companies are held as proprietary information, and thereby not available in the public domain.

This chapter only discusses very specific aspects of the related work. For a broader discussion on the state of the art in the field of Visible Light Communication, see Section 3.1 of Phase One of my thesis [4].

7.2 Defining My Work

When comparing existing work to the work I have carried out for this project, it becomes apparent that my project had aims and goals which differed significantly from the existing work. This section defines my work with the purpose of assisting the reader in positioning it in a context relative to other work presented in the chapter. My task has been to create an AXI converter tailored to the reception and transmission of data through the medium of Visible Light Communication. While I have been making it for a SoC which uses the EnCore processor, part of the reason for using the AXI protocol has been to produce a design which is relatively system-agnostic, and that can be used as part of any SoC, provided it applies the open AXI protocol as a standard for its interconnect. This differs from related technologies in a few crucial ways, making the goals and challenges faced in my work different from most existing work.

One of the major differences is that the hardware described in my project is intended to be part of a SoC, similar to a bluetooth or Wi-Fi chip in existing electronic devices. This differs from demonstrated concepts, where standalone receivers and transmitters tend to be used. Also, the system I have designed does not make any effort to encode or decode data, beyond handling the direct output and input tied to the conversion between analogue light and digital data samples. Abstracting the data processing out of the system allows the hardware to be used generally, for a range of specific applications.

A discussion of my work as compared and contrasted to related work is found in Section 7.5.

7.3 Academic Work

A range of experiments have been carried out to test and prove the capabilities of VLC. In this section I will look at some of the most popular and most often cited experiments, and outline the setup that was used for transmission and reception of the signals relating to the work carried out.

In [14], it is reported that received data was recorded with a real-time storage oscilloscope, and then processed further. While it is reported that the signals were down-sampled, demodulated and de-mapped, the exact process behind this has not been specified. It is, however, a fair assumption to make that the hardware and software used for this processing was designed to prevent it from being a bottleneck with regard to measuring BER and speed of the data transmission. Little is reported about how samples making up the transmitted light are prepared, beyond that it is created using two independent waveform generators. Similarly, [15] contains limited information on the retrieval and storage of the data that has been transmitted, but the same experimental setup has been outlined. This is natural, as the data transmitted and its processing were of little relevance in these papers, where the main focus was the speed and accuracy achieved by the transmission itself.

An example of a more practically-oriented paper is [16] by Elgala et. al. Here a physical system for VLC transmission using OFDM is presented, with emphasis being put on actually realising the technology in hardware and demonstrating the feasibility of this, rather than on the more theoretical aspects. In this paper a DSP development board [22] was connected to each of the transmitting and receiving computers. This board has its own interface for transferring data to and from a computer, using a USB connection.

Theoretical papers, such as [17], ignore the practical aspects of data conversion entirely.

Despite the lack of academic papers around the data transactions specifically for VLC, there is a solid academic base demonstrating the advantages of using bus protocols such as AXI. [18] implements a general transaction-based network interface on a System on Chip, arguing for and demonstrating the necessity of using such a system for the benefits of performance and of re-usability. This is taken further in papers like [19], in which the use of the AXI protocol is realised on a SoC, and its advantages demonstrated.

Following on from the availability of a common bus protocol like AXI is the availability of components supporting this protocol, and wrappers to add this support to already existing components. ARM have published papers like [20] promoting the benefits of designing and using components designed specifically around such a common bus. [21] uses USB as an example to demonstrate the benefits of exploiting these interconnects, arguing and demonstrating how and why individual IP blocks benefit from being abstracted from the system. However, the specific performance of such components is largely of commercial, rather than academic interest, as demonstrated by papers such as [21], funded by Conexant Systems and UC Micro.

Detecting packets using a preamble is discussed in papers such as [10], which formed the basis for my method of carrying out frame detection in my project.

7.4 Commercial Implementations

Quite recently, a number of very diverse commercial implementations of VLC have begun to emerge.

One example is a Casio mobile phone application that takes advantage of the existing camera on mobile phones, *Picapicamera* [23]. This app enables the screen of a smartphone to display flashing lights that can be picked up and decoded by the camera on another phone with the app installed. An extended use of this app is the ability to view advertisements, or other "in the wild" applications using light in a manner supported by the application. However, while this utilisation of VLC is novel, it has little relevance to using VLC for transmitting data at high speeds.

The company PureVLC is working more specifically on facilitating the transmission of data through VLC. Their most recent development has been LightMessage [24], a mobile phone application which is able to receive data through VLC. While the data-rate is still low, 2.5 kbps, this crucially shows that VLC technology can work, using a commonly-available, off-the-shelf smartphone camera as a receiver. PureVLC will also be providing a "Li-Fi Development Kit" which can be used to develop applications using LEDs as a method of transmitting data. The hardware contained in this kit contains units that will encode and decode the data.

Research has also been carried out by larger companies, in an effort to explore how they might be able to utilise VLC in the future. One example here is Siemens, who reported that they achieved data rates of 500 Mb/s using white light [25]. However, the details behind such efforts, especially relating to device-specific handling of data, remain unpublished.

Aside from VLC there is a separate commercial field, related to my project, which has been thoroughly explored. IP blocks that utilise the AXI protocol for receiving and transmitting data are readily available. This ranges from Synopsys making their Ethernet IP compatible with the AXI protocol [26], to Xilinx making AXI compatible IP available for I/O interfaces such as Ethernet and USB [27] [28]. A search in any IP catalogue will return a large number of similar components. In addition, ARM provide AXI compatible hardware such as PrimeCell UART [29]. Despite not being directly related to VLC, these products are more generally related to the work that I have carried out for my project. They work as general, application-specific, interfaces between an AXI interconnect and methods of exchanging data.

7.5 Comparing and Contrasting

In reviewing the available related work, it becomes clear that this differs to a large extent from the work I have carried out. Understandably, the academic work focuses on the academic aspects of VLC, aiming to develop and mature the technology itself rather than dealing with its implementation on a hardware level. In the commercial sphere, the development of VLC is still in its infancy, and the efforts to date have been aimed at giving VLC a viable commercial foothold.

My work falls somewhere in between these two categories. It is of academic interest, since development of an AXI interface which works on a SoC will be a proof of concept of the ability of consumer devices to contain VLC dedicated hardware. The academic papers focus primarily on aspects of VLC such as modulation techniques and data encoding. These are largely irrelevant to my work, apart from giving an indication of what bandwidth and SNR a VLC system can be expected to deal with. Due to the practical nature of my project, this was as anticipated. I consider work resulting from my project to be a platform for potential future research. Once VLC has reached a point where the challenges lie in its implementation on devices, developing interfaces compatible with standards such as the AXI protocol is an inevitable next step.

For this reason, the subject matter of my work is also of commercial interest. VLC is not yet sufficiently mature to make a serious impact in the commercial market. However, as is evident from efforts made by companies such as PureVLC, the technology is approaching the point at which it will be viable to include dedicated VLC-related hardware in pocket-sized devices. Once the technology reaches the stage where it is likely to become embedded in devices, rather than used for external hardware, systems similar to what I have worked on in my project will be a natural progression. Commercial efforts, such as LightMessage, demonstrate that this point may not be far off. LightMessage shows a potential use case, but more importantly it proves that even the current optical technology contained in smartphones is capable of detecting data transmitted through VLC. The data rate is currently much slower than that which might be desired from a technology such as VLC, but this is most likely due to the limited capability of the phone's software to handle the incoming light quickly enough. If the phone contained hardware to handle packet detection and the reception of data, similar to that developed for my project, a significant load would be taken off the phone's main processing unit. This unit would then have more capacity to efficiently decode the data, allowing the phone to receive and interpret data at higher speeds.

While I have been unable to find work relating more closely to my project, this does not necessarily mean that it does not exist. It is possible, even likely, that hardware for interfacing VLC with interconnects has already been developed as part of commercial efforts or as part

of research at companies such as Siemens. At this stage such work is likely to be proprietary.

It is natural for me also to comment on where my work stands in relation to other available systems that interface with data using the AXI protocol. AXI controllers for I/O protocols such as Ethernet and USB have been in use for a long time, and are readily available. These interface with very mature technologies, and have been developed and improved over a long period of time. Their design and implementation are therefore considered too application-specific to compare directly to my work. Of interest, however, is how widespread such IP-blocks are. This demonstrates a demand for AXI-components that can interface with data transmission technologies.

What I have done for my project is to develop some groundwork upon which further efforts can be based. Throughout my work I have attempted to make my hardware as customisable as possible, making it simple to integrate into any system. This is what sets my work aside from the related work I have been able to identify. Efforts so far, both academic and commercial, have been very specific, whether it be with the purpose of researching aspects of the VLC technology, or using the technology for given applications. This makes it unsuitable for use in broader research, and for the purposes of further exploring the potential behind VLC as part of a SoC. My project, on the other hand, aims to be general, avoiding over-specialisation, while still offering options enabling it to be adapted to a range of systems. This makes it suitable for research, as it can easily be adapted to cater to a range of requirements which one might require when further exploring VLC. It is also completely open, and modular, meaning that individual parts of it can be used, if this is desired.

7.6 Conclusion

The existing body of work related to my project is very diverse, mainly found in academia but also beginning to emerge as commercial efforts. A common factor in this work is that it is very specialised. In the case of research this follows on from the scientific approach to the technology, where certain aspects of VLC need to be studied in isolation. Commercial efforts are focused on proving the potential of the technology, and establishing a commercial interest around it by demonstrating the viability of certain applications.

It is therefore natural that my work, intended to be general and re-usable, has limited overlap with existing work. Until VLC becomes a technology which features as a requirement for hardware, the commercial market will show little interest in creating a general purpose VLC interface, as the one created. The rationale behind the development of such an AXI interface for my project was to enable the PASTA project to start using VLC as part of their research. My work is therefore designed to be as adaptable as possible, catering for use of the end product in whatever way is required. I believe this is where it differs from the other

material I have identified, making it an AXI interface which is general enough to fit into a wide range of systems, and easy to customise and modify to the specific needs of such a system.

Chapter 8

Conclusion

This chapter concludes the report, focusing on the aspects of the work which turned out to be especially challenging. A summary of the way in which I approached, and solved, some of these problems is presented. Finally, the report is concluded with an assessment of the work I carried out, as well as some personal observations relating to it.

8.1 Introduction

This conclusion discusses the aspects of my project which I found to be most challenging, and therefore the most interesting in terms of what I achieved. I consider the work carried out for my project to have been fairly successful, and I have succeeded in accomplishing the work I had aimed to do. In addition to the work which fell within my original project scope, I was also tasked with making a packet detector for incoming data. This added significantly to the complexity of the project, however in a limited amount of time I managed to describe a packet detector in Verilog that worked according to the specification.

8.2 Summary of Main Project Achievements

This section summarises the main parts of my project by focusing on the most challenging aspects of my work and outlines how I solved them.

8.2.1 Packet Detection

The most challenging part of my work proved to be the packet detection. This was the only module which had to carry out a large number of operations for every signal to enter into the

system. Its purpose was to look for the presence of a given pattern in the signal, and detect when this pattern occurred. This signified that a data packet was about to arrive, and that this data should start being written to the system memory.

- Speed and Calculation Efficiency

  The packet detection part of the system had to be able to process a large amount of data very quickly. The speed at which the system was able to work was directly related to the efficiency of the calculations which had to be carried out, as more efficient calculations take less time to execute. This turned out to be more challenging than was first assumed, as a number of relatively heavy calculations needed to be carried out frequently. The calculations needed for the frame detection were:

  - Cross-correlation of two 16-sample signals, to get the cross-correlation signal
  - Taking the absolute value of 31 values and adding them together, to get a representation of the signal energy
  - Calculating the average of eight values and carrying out a comparison, to detect a peak

  Attempting to carry out all these operations in one clock cycle is quite ambitious.

  The immediate matter that had to be resolved was the cross-correlation. The generally accepted way to calculate these efficiently involves the use of an FFT. This was not an option in the system I created, as FFT operations are computationally heavy, and the FFT logic needed to be available for use by the software for decoding of incoming data.

  The problem to be solved, finding the level of similarity between two sets of samples, was one which is best solved by a cross-correlation, and it was apparent that I had to find a way to implement it. On the face of it, I had to carry out a calculation which was very inefficient. However, due to the very specific way in which I was to use the definition, where the signals to be correlated shifted along relative to each other, there was a symmetry in this calculation which could be utilised. I designed a method of carrying out this calculation by exploiting this symmetry, which turned out to be sufficiently efficient for my purposes.

  The result of the correlation efforts was a signal which increased in energy in proportion to the similarity of the signals. To take advantage of this, I needed a second module which would calculate the signal energy of the cross-correlation signal. This proved to be much more complicated than first expected, but I eventually found an approach which worked within the time constraints. This is discussed at length in Section 4.4.3. Finally, the calculation of averages turned out to be quite straightforward.

In addition to deriving computationally-viable methods, I took an additional step to ensure that my detection module was as good as was possible. The calculations were all carried out in separate, independent modules. This means they could easily be optimised independently, or other, more efficient modules could be used as black boxes as a way of improving the efficiency of the signal detection.

- Accuracy

  In testing, the performance of the peak detection module was very good. Using a cross-correlation, as I do, is a reliable method of checking for signal correspondence, which makes the signal detector very accurate. It was able to detect incoming signals even when they were only slightly stronger than the noise interfering with them, down to an SNR of 0.8 dB.

- Hardware utilisation

  While it was a secondary goal, making the module efficient with regard to the amount of hardware used was also an important aspect of my work. Unfortunately, an unavoidable consequence of having to do calculations quickly is a relatively large amount of hardware. I have attempted to minimise the hardware required by making the number of significant bits used in calculations configurable. Due to the large number of calculations involved, even a small reduction in significant bits yields large savings in space needed for the hardware. The configuration feature allowed me to adjust the number of bits in use easily. This enabled me, through trial and error, to find the optimum tradeoff in terms of hardware and performance.

8.2.2 Signal Reception

The signal reception was very challenging at first, as I had to address problems such as clock domains, buffering and managing pointers. This was simplified significantly by the decision to use the FIFO. There were also some other challenging aspects to this module, which are presented here.

- Buffering

  The buffering was initially done using a large register array contained within the module. This method was complicated, and involved keeping track of pointers and space, as well as resolving issues arising from the use of different clock domains. Once the FIFO was used, this acted as the buffer. As the FIFO works asynchronously, it immediately solved the issues relating to the different clock domains. It also inherently keeps

track of all memory management, so buffering became manageable once the module was designed around a FIFO.

- Writing to Memory

  The challenge in writing to memory was twofold. Firstly, the AXI interaction itself needed to be implemented. This was done in accordance with the AXI specification. The more challenging aspect of writing to memory was to ensure that the module wrote the data to the appropriate memory location.

  This was solved by giving the top module the functionality of an AXI Slave which allowed it to be used as a small write-only memory by the rest of the system. Information about where the system wanted data to be placed in memory was written to this memory, and the reception module used this information to decide where to put data. Another input used by the system was a 32-bit word of data representing the current address which had been processed by the memory. This gave the module information on which addresses in memory it could safely re-use.

- Reporting Back

  The module needed a way of communicating to the system information such as whether it was operating properly and how much available memory remained. This was solved by implementing a 32-bit status word. The module was already set up to write to the memory over the AXI protocol. At a configurable interval, instead of writing data, the module would write status information, which took the form of a 32-bit word. The location to which this word was being written was configurable through a register which could be set in the top module.

8.2.3 Signal Transmission

Another basic function of the system was to transmit data. This involved reading data through the AXI interconnect, buffering it, and transmitting it. A preamble also had to be transmitted. Below is a summary of the two main challenges encountered in this area and how I designed the system to address them.

- Buffering

  As for the reception model, data had to be buffered. Again, this was initially done using a register array which was complicated, and imposed significant limitations on the system.

Ultimately the FIFO was used for buffering. The challenge in doing this for the transmission module, was that I had intended to read data in parallel over the AXI interconnect. This required state machines to keep track of which processes were reading, as well as asserting the correct signals with regard to the AXI interconnect. Using the FIFO as a buffer, the data to be transmitted crucially had to be written into this buffer in the correct order. A solution to handle all this was implemented successfully by using a number of state machines. The data was read quickly from the AXI interconnect, thanks to the parallel read-operations, and it was transmitted successfully.

- Transmission of Preamble

  Prior to any signal that was to be transmitted, a 160-sample preamble had to be transmitted. A separate module was created to facilitate this, for the sole purpose of transmitting these 160 samples. As these samples were periodic, 16 samples were hard-coded into the module and transmitted using a state machine which looped through them ten times.

8.2.4 Simulation and Testing

Throughout the design of the system I ran tests to ensure it was functioning properly. Several challenges arose with regard to the testing of the system. Here is how I overcame them.

- Generating sample signals

  For the system to be tested, a large number of sample inputs had to be generated, and fed into the system. Generating the samples themselves was relatively simple, and could be done using Matlab functions. I had started creating a script to convert these generated samples to the required format, Q15, when I discovered that Matlab also had a function which could carry out this conversion for me. The challenge then became feeding these samples into the system in order to see how the system reacted to them. I realised this could be done through a counter that selected the appropriate input from a huge case statement. I created a Python script to generate the Verilog code that was required for this. The process used is described in Section 4.5.

- Simulating “real-life” operation of the system

  Prior to initiating the work, I had assumed the main challenge to be the design of the noise-handling aspects of the system. As I stress-tested the system with a very low SNR the system handled this very well, as described in Section 4.5.3. Another challenge for the system would be for it to deal with back-to-back input of data, that is, consecutive data packets. This was tested by making the system perform continuous receptions and transmissions, and the system succeeded in handling this load. The system was not able to handle simultaneous transmission and reception, as noted in Section 8.3.

8.2.5 Overall Goals

In addition to the goals relating to specific aspects of the system that has been designed, I have also aimed to meet some more general goals.

- Abstracting parameters out of the code

  As this has been a prototype design, an important part of my design process has been to give as many elements as possible the potential for re-use in other systems in the future. I think I succeeded in this. Most importantly, the module is completely configurable with regard to what packet length the system should be designed for. I also made other aspects of the module configurable, meaning that decisions on tradeoffs can be made that differ from mine. The amount of additional data written following a packet of data can be configured, allowing dynamic decisions to be made with regard to accuracy vs. memory requirements. In addition, the frequency at which the status word is delivered, informing the system of how the module is operating, can be configured. This status should be sent as infrequently as possible, but has to be sent more often in systems with less memory.

- Creating a robust AXI implementation

  Creating a robust AXI implementation has been a basic requirement, but it has also been a complicated one. There are a number of subtle requirements in the AXI protocol that had to be implemented correctly. When something was not quite right, the cause of the problem was not always immediately obvious, and quite rigorous troubleshooting had to be carried out with the aid of the simulator. Eventually I implemented the AXI interactions with the system in a manner that seems to work, and I am confident that the AXI specification has been adhered to.

8.3 Further Work

I regard the module I have created for this project as a fairly robust system prototype. However, there are improvements that could be made. Due to the limited amount of time available, I focused my efforts on implementing all the features required, at the expense of refining these features. Therefore I see a number of aspects of this project that could form the basis for future work.

Most importantly, the testing that has been carried out on this module has been entirely simulated. This is due to adjacent systems not being available yet, making “real-life” testing

of the module impossible. Testing the module as part of an actual system is likely to uncover new problems with the design, that also will have to be solved.

Although my implementation of the features has been successful, there is undoubtedly a potential for further improvement of the efficiency. The signal detection module is an example where I for calculations use a relatively "off-the-shelf" implementation of the cross-correlation method. While I have optimised this to adapt to a specific use case, it is likely that a method which is more efficient than a standard cross-correlation could be used. This is partly the reason for the modular approach of my design. Where possible, code representing different operations in the system has been implemented as separate modules. This allows many parts of the system to be treated as black boxes which can be further optimised without regard to the rest of the system, as long as the same inputs and outputs are supported.

The final testing of the complete system posed a potential problem which has not yet been solved. Due to the traffic on the AXI protocol, the system is not able to read and write simultaneously. The system will therefore not be able to receive a packet of data, while also transmitting one. This is a limitation of the AXI interconnect, as the amount of data that needs to be passed through it is too large. Reading and writing simultaneously was not part of the original specification, and this problem may be largely theoretical, as it is unlikely that a system would need to transmit while receiving. However, the possibility of this occurring is not presently dealt with in an elegant manner. I considered implementing a rather simple block, that would entirely prevent the system from even attempting to read and write at the same time. However, this would be implementing a limitation which would be better dealt with in software. A potential way of dealing with the problem more appropriately would be to receive transmission instructions from the processor, and queue these up for execution once the reading of the current data packet had been completed.

8.4 Concluding Remarks

In summary, I think I have been successful in achieving my goals. I have created a working prototype that can be used for facilitating the use of VLC in a SoC. The system which has been designed conforms to the AXI specification, making it usable not only on the SoC for which it is intended, but also in other systems which are built around the AXI protocol. The work has been challenging and time-consuming. At the outset of the project, I had a very academic background in Verilog, and the approach I first took was coloured by this. In progressing with my work, I realised that working with Verilog in a more practical context was very different, and writing code to be used in a larger system has been a useful learning process. I am happy with the end result, and feel I have accomplished what I set out to do.

I have designed and implemented a sizeable module, consisting of a number of individual

parts. The original scope of my project consisted of creating the modules used for interacting with the AXI protocol, subsequently expanded to include dealing with signal detection. This was challenging, as there was little prior work on which to base my approach. I had to largely invent, design, and implement a method of my own. The computation required to carry this out was quite complex, and implementing this became one of my main challenges. Eventually I devised a method, and an implementation, which worked within the hardware limitations.

The AXI implementation also presented a number of challenges, as I had no prior experience with the AXI protocol. This work became easier as I gained some familiarity with the protocol, and how it is implemented in practice. My experience with this has been positive, and has led me to conclude that the AXI protocol is well-suited to implementations such as this.

Overall, I have enjoyed working on my project. The main objective of my work has been academic, as I have designed a system which will initially be used for research purposes. However, as VLC matures, and becomes more widespread, AXI-compatible hardware for VLC will be in demand. I feel that I have designed something which may be of real practical use when VLC reaches a stage where dedicated hardware supporting it is to be implemented in mobile devices. The result of my work is a prototype, but I think it, or parts of it, represent hardware which could eventually be used in tangible systems.

I feel fortunate to have had the opportunity to do my masters project on something which I feel is meaningful, and I am satisfied with the results of my work.

Acknowledgements

I would like to thank Dr. Björn Franke, as my supervisor, for the advice and assistance he has provided in the work on this report, and Professor Harald Haas for being my thesis examiner.

Thanks also to my proofreaders, including Stephen McGruer who also provided me with useful technical input.

Finally, a huge thanks to Oscar Almer for the invaluable assistance he provided throughout my project, for the time he spent explaining the system to me, for the help he provided whenever I required any, for looking over and giving feedback on my code, and finally for reading and giving feedback on my report.

Appendix A

Samples of Code

This appendix contains code snippets demonstrating the Verilog-implementation of some parts of the system. Note that these are only parts of the code.

A.1 The State Machine used for the AXI Slave

```
always @(*)
begin
        case(state)
        `WAIT: nextstate = AWVALID_S ? `INIT : `WAIT;
        `INIT: nextstate = readynow ? `READY : `INIT;
        `READY: nextstate = WVALID_S ? `WRITE1 : `READY;
        `WRITE1: nextstate = donewriting ? `WRITE2 : `WRITE1;
        `WRITE2: nextstate = lastdata ? `RESPOND : `READY;
        `RESPOND: nextstate = alldone ? `FINISH : `RESPOND;
        `FINISH: nextstate = `WAIT;
        default: nextstate = `WAIT;
        endcase
end
always @(posedge clk or posedge reset)
begin
        if(reset)
        begin
                trans_id <= 0;
                trans_addr <= 0;
                readynow <= 0;
                writedata <= 0;
                alldone <= 0;
                WREADY_S <= 0;
        end
        else
        begin
                if(state == `WAIT) begin
                        AWREADY_S <= 1;
                end
                else begin
                        AWREADY_S <= 0;
```

```
                end
                if(state == `INIT) begin
                        trans_id <= AWID_S;
                        trans_addr <= AWADDR_S;
                        readynow <= 1;
                        donewriting <= 0;
                        BVALID_S <= 0;
                end
                else begin
                        trans_id <= trans_id;
                        trans_addr <= trans_addr;
                        readynow <= 0;
                end
                if(state == `READY) begin
                        WREADY_S <= 1;
                        BID_S <= trans_id;
                end
                if(state == `WRITE1) begin
                        data <= WDATA_S;
                        readynow <= 0;
                        writedata <= 1;
                        donewriting <= 1;
                end
                else begin
                        data <= data;
                end
                if(state == `WRITE2) begin
                        WREADY_S <= 0;
                        writedata <= 0;
                end
                else begin
                        writedata <= writedata;
                end
                if(state == `RESPOND) begin
                        if (BVALID_S == 0 && alldone == 0) begin
                        BVALID_S <= 1;
                        alldone <= 1;
                        end
                        else begin
                                BVALID_S <= 0;
                        end
                end

                if(state == `FINISH) begin
                        BVALID_S <= 0;
                        alldone <= 0;
                end

                if (WLAST_S) begin
                        lastdata <= 1;
                end
                else begin
                        lastdata <= 0;
                end
end
end
```

A.2 Receiving Signals and Writing them to the FIFO

A.2.1 The Continuous Buffering

```
// Buffers all incoming signals into the 2D register array, prebuffer.
always @(posedge clk or posedge reset) begin
        if (reset) begin
                prebuffercounter <= 0;
        end else begin

                if (prebuffercounter < 32) begin
                        prebuffer[prebuffercounter] <= signalin[15:0];
                        prebuffercounter <= prebuffercounter + 1;
                end else begin
                        prebuffer[0] <= signalin[15:0];
                        prebuffercounter <= 1;
                end
        end
end
```

A.2.2 The Reception of Data at Incoming Packets

```
// For writing the content of the pre-buffer into the FIFO when a signal is
// incoming, in the process grouping 16-bit segements together to 32 bits.
always @(posedge clk or posedge reset) begin
        if (reset) begin
                intbuffer <= 0;
                intbufferflag <= 0;
                fifowrite <= 0;
                fifoin <= 0;
        end else begin

        if (packetactive && !fifofull) begin
                if (intbufferflag) begin
                        intbuffer[31:16] <= prebuffer[prebuffercounter-1];
                        intbufferflag <= !intbufferflag;
                        fifowrite <= 0;
                end else begin
                        fifoin <= intbuffer;
                        fifowrite <= 1;
                        intbuffer[15:0] <= prebuffer[prebuffercounter-1];
                        intbufferflag <= !intbufferflag;

                end
        end

        end
end
```

A.3 Reading data from Memory into the FIFO

```
// Request-states
always @(*)
begin
        case(requeststate)
        `REQUESTWAIT: nextrequeststate = ((bin1free || bin2free || bin3free) &&
        (packetcounter < `PACKSEGS ) && readnow && !fifohalffull) ? `REQUESTNEW : `REQUESTWAIT;
        `REQUESTNEW: nextrequeststate = ARREADY_M ? `REQUESTDONE : `REQUESTNEW;
        `REQUESTDONE: nextrequeststate = `REQUESTWAIT;

        default: nextrequeststate = `REQUESTWAIT;
        endcase
end

// Read-states
always @(*)
begin
        nextreadstate =`READWAIT;

        case(readstate)
        `READWAIT: nextreadstate = RVALID_M ? `GETDATA : `READWAIT;
        `GETDATA: nextreadstate = !RREADY_M ? `GETDATA2: `GETDATA;
        `GETDATA2: nextreadstate = `READWAIT;
        default: nextreadstate = `READWAIT;
        endcase
end


// Determine whether to start reading.
always @(posedge clk or posedge reset) begin
        if (reset) begin
                readnow <= 0;
        end else begin

                if (readstart) begin
                        readnow <= 1;
                end else if (packetcounter >= `PACKSEGS) begin
                        readnow <= 0;
                end
        end
end

// State switching
always @(posedge clk or posedge reset)
begin
        if(reset) begin
                requeststate <= `REQUESTWAIT;
                readstate <= `READWAIT;
        end
        else begin
                requeststate <= nextrequeststate;
                readstate <= nextreadstate;
        end
end
```

```
//ReadReady-signal
always @(posedge clk or posedge reset) begin
        if (reset) begin
                RREADY_M <= 0;
        end else begin
                if(readstate == `READWAIT) begin
                        if (RVALID_M) begin
                                RREADY_M <= 1;
                        end else begin
                                RREADY_M <= 0;
                        end
                end

                if(readstate == `GETDATA) begin
                        if (RREADY_M) begin
                                RREADY_M <= 0;
                        end
                end

        end
end

// The FIFO-stuff
always @(posedge clk or posedge reset) begin
        if (reset) begin
                fifowrite <= 0;
                fifoin <= 0;
                buffflag <= 0;
                buffoutcount <= 0;
        end else begin

        if (!fifohalffull || fifowrite) begin

                if (bin1full && (bin1pos == 1)) begin
                        if (buffoutcount < 16) begin
                                if (!buffflag) begin
                                        fifowrite <= 1;
                                        fifoin <= bin1buff[buffoutcount][31:16];
                                        buffflag <= 1;
                                end else begin
                                        fifowrite <= 1;
                                        fifoin <= bin1buff[buffoutcount][15:0];
                                        buffflag <= 0;
                                        buffoutcount <= buffoutcount + 1;
                                end
                        end else begin
                                fifowrite <= 0;
                                buffoutcount <= 0;
                        end
                end else if (bin2full && (bin2pos == 1)) begin
                        if (buffoutcount < 16) begin
                                if (!buffflag) begin
                                        fifowrite <= 1;
                                        fifoin <= bin2buff[buffoutcount][15:0];
                                        buffflag <= 1;
                                end else begin
```

```
                                        fifowrite <= 1;
                                        fifoin <= bin2buff[buffoutcount][31:16];
                                        buffflag <= 0;
                                        buffoutcount <= buffoutcount + 1;
                                end
                        end else begin
                                fifowrite <= 0;
                                buffoutcount <= 0;
                        end
                end else if (bin3full && (bin3pos == 1)) begin
                        if (buffoutcount < 16) begin
                                if (!buffflag) begin
                                        fifowrite <= 1;
                                        fifoin <= bin3buff[buffoutcount][15:0];
                                        buffflag <= 1;
                                end else begin
                                        fifowrite <= 1;
                                        fifoin <= bin3buff[buffoutcount][31:16];
                                        buffflag <= 0;
                                        buffoutcount <= buffoutcount + 1;
                                end
                        end else begin
                                fifowrite <= 0;
                                buffoutcount <= 0;
                        end
                end

        end else begin
                fifowrite <= 0;
        end

        end
end

// Bin signals, full or free.
always @(posedge clk or posedge reset) begin
        if (reset) begin
                bin1full <= 0;
                bin2full <= 0;
                bin3full <= 0;

                bin1free <= 1;
                bin2free <= 1;
                bin3free <= 1;
        end else begin
                if (bin1full && (bin1pos == 1)) begin
                        if (buffoutcount >= 16) begin
                                bin1full <= 0;
                                bin1free <= 1;
                        end
                end else if (bin2full && (bin2pos == 1)) begin
                        if (buffoutcount >= 16) begin
                                bin2full <= 0;
                                bin2free <= 1;
                        end
                end else if (bin3full && (bin3pos == 1)) begin
```

```
                        if (buffoutcount >= 16) begin
                                bin3full <= 0;
                                bin3free <= 1;
                        end
                end

                if(readstate == `GETDATA) begin
                        if (RREADY_M) begin
                                if (RLAST_M) begin
                                        if (RID_M[3:0] == bin1id[3:0]) begin
                                                bin1full <= 1;
                                        end else if (RID_M[3:0] == bin2id[3:0]) begin
                                                bin2full <= 1;
                                        end else if (RID_M[3:0] == bin3id[3:0]) begin
                                                bin3full <= 1;
                                        end
                                end
                        end
                end

                if(requeststate == `REQUESTNEW) begin
                        if (bin1free) begin
                                bin1free <= 0;
                        end else if (bin2free) begin
                                bin2free <= 0;
                        end else if (bin3free) begin
                                bin3free <= 0;
                        end
                end
        end
end

// Address request handling
always @(posedge clk or posedge reset) begin
        if (reset) begin
                ARVALID_M <= 0;
                ARADDR_M <= 0;
                ARID_M <= 0;
        end else begin

                if(requeststate == `REQUESTWAIT) begin
                        ARVALID_M <= 0;
                end

                if(requeststate == `REQUESTNEW) begin
                        ARADDR_M <= requestedaddress;
                        ARID_M <= ARID_M + 1;
                        ARVALID_M <= 1;
                end

                if(requeststate == `REQUESTDONE) begin
                        ARVALID_M <= 0;
                end
        end
end
```

```
// Writing to the buffers
always @(posedge clk or posedge reset) begin
        if (reset) begin
                bin1counter <= 0;
                bin2counter <= 0;
                bin3counter <= 0;
        end else begin

        if(readstate == `GETDATA) begin
                if (RREADY_M) begin
                        if (RID_M[3:0] == bin1id[3:0]) begin
                                bin1buff[bin1counter] <= RDATA_M;
                                if (RLAST_M) begin
                                        bin1buff[bin1counter] <= RDATA_M;
                                        bin1counter <= 0;
                                end else begin
                                        bin1counter <= bin1counter +1;
                                end
                        end else if (RID_M[3:0] == bin2id[3:0]) begin
                                bin2buff[bin2counter] <= RDATA_M;
                                if (RLAST_M) begin
                                        bin2buff[bin2counter] <= RDATA_M;
                                        bin2counter <= 0;
                                end else begin
                                        bin2counter <= bin2counter + 1;
                                end

                        end else if (RID_M[3:0] == bin3id[3:0]) begin
                                bin3buff[bin3counter] <= RDATA_M;
                                if (RLAST_M) begin
                                        bin3buff[bin3counter] <= RDATA_M;
                                        bin3counter <= 0;
                                end else begin
                                        bin3counter <= bin3counter + 1;
                                end
                        end

                end
        end
        end
end

// Request address and counter handling
always @(posedge clk or posedge reset) begin
        if (reset) begin
                requestedid <= 0;
                requestedaddress <= 0;
                packetcounter <= 0;
        end else begin

        if(requeststate == `REQUESTWAIT && !readnow) begin
                requestedaddress <= base;
        end

        if(requeststate == `REQUESTDONE) begin
```

```
                requestedid <= ARID_M;
                requestedaddress <= requestedaddress + 64;
                packetcounter <= packetcounter + 1;
        end

        if (packetcounter >= `PACKSEGS) begin
                packetcounter <= 0;
                requestedaddress <= base;
        end

        end
end

// bin address and position handling
always @(posedge clk or posedge reset) begin
        if (reset) begin
                bin1id <= 0;
                bin2id <= 0;
                bin3id <= 0;

                bin1pos <= 0;
                bin2pos <= 0;
                bin3pos <= 0;
        end else begin
                if(requeststate == `REQUESTNEW) begin
                        if (bin1free) begin
                                bin1id  <= ARID_M + 1;
                                if (((bin2pos == 1) || (bin3pos == 1)) && ((bin2pos == 2) ||
                                (bin3pos == 2))) begin
                                        bin1pos <= 3;
                                end else if ((bin2pos == 1) || (bin3pos == 1)) begin
                                        bin1pos <= 2;
                                end else begin
                                        bin1pos <= 1;
                                end
                        end else if (bin2free) begin
                                bin2id  <= ARID_M + 1;
                                if (((bin1pos == 1) || (bin3pos == 1)) && ((bin1pos == 2) ||
                                (bin3pos == 2))) begin
                                        bin2pos <= 3;
                                end else if ((bin1pos == 1) || (bin3pos == 1)) begin
                                        bin2pos <= 2;
                                end else begin
                                        bin2pos <= 1;
                                end

                        end else if (bin3free) begin
                                bin3id  <= ARID_M + 1;
                                if (((bin2pos == 1) || (bin1pos == 1)) && ((bin2pos == 2) ||
                                (bin1pos == 2))) begin
                                        bin3pos <= 3;
                                end else if ((bin2pos == 1) || (bin1pos == 1)) begin
                                        bin3pos <= 2;
                                end else begin
                                        bin3pos <= 1;
                                end
```

```
                end
            end

            if (bin1full && (bin1pos == 1)) begin
                if (buffoutcount >= 16) begin
                    bin1pos <= 0;
                    if (bin2pos == 2) begin
                        bin2pos <= 1;
                        if (bin3pos == 3) begin
                            bin3pos <= 2;
                        end
                    end else if (bin3pos == 2) begin
                        bin3pos <= 1;
                        if (bin2pos == 3) begin
                            bin2pos <= 2;
                        end
                    end
                end
            end else if (bin2full && (bin2pos == 1)) begin
                if (buffoutcount >= 16) begin
                    bin2pos <= 0;
                    if (bin1pos == 2) begin
                        bin1pos <= 1;
                        if (bin3pos == 3) begin
                            bin3pos <= 2;
                        end
                    end else if (bin3pos == 2) begin
                        bin3pos <= 1;
                        if (bin1pos == 3) begin
                            bin1pos <= 2;
                        end
                    end
                end
            end else if (bin3full && (bin3pos == 1)) begin
                if (buffoutcount >= 16) begin
                    bin3pos <= 0;
                    if (bin1pos == 2) begin
                        bin1pos <= 1;
                        if (bin2pos == 3) begin
                            bin2pos <= 2;
                        end
                    end else if (bin2pos == 2) begin
                        bin2pos <= 1;
                        if (bin1pos == 3) begin
                            bin1pos <= 2;
                        end
                    end
                end
            end
        end
    end
```

A.4 Writing to Memory over AXI

```
// State-switching for the writing state machine.
always @(*)
begin

        nextwritestate ='WAIT;

        case(writestate)
        'WAIT:  if ((writeready)&& (addresscounter >= 64)) begin // Extra check to see if there's space
                        nextwritestate = 'WRITESTART;            // needed for edgecase
                end else if (statuswrite) begin
                        nextwritestate = 'STATUSADDRESS;
                end else begin
                        nextwritestate = 'WAIT;
                end
        'WRITESTART: nextwritestate = AWREADY_M ? 'WRITE1 : 'WRITESTART;
        'WRITE1: nextwritestate = ((burstcounter == 14) && WVALID_M && WREADY_M) ? 'WRITELAST : 'WRITE1;
        'WRITELAST: nextwritestate = WREADY_M ? 'RESPONSE : 'WRITELAST;
        'RESPONSE: nextwritestate = !BREADY_M ? 'WAIT : 'RESPONSE;

        'STATUSADDRESS: nextwritestate = AWREADY_M ? 'STATUSWRITE : 'STATUSADDRESS;
        'STATUSWRITE: nextwritestate = WREADY_M ? 'STATUSRESPONSE : 'STATUSWRITE;
        'STATUSRESPONSE: nextwritestate = !BREADY_M ? 'WAIT : 'STATUSRESPONSE;

        default: nextwritestate = 'WAIT;
        endcase
end


// Setting BREADY_M, handling response.
always @(posedge clk or posedge reset) begin
        if (reset) begin
                BREADY_M <= 0;
                replyid <= 0;
        end else begin
                if (BVALID_M && (BID_M[3:0] == replyid)) begin
                        BREADY_M <= 0;
                end
                if (writestate == 'RESPONSE) begin
                        if (!BREADY_M) begin
                                replyid <= AWID_M[3:0];
                                BREADY_M <= 1;
                        end
                end
                if (writestate == 'STATUSRESPONSE) begin
                        if (!BREADY_M) begin
                                replyid <= AWID_M[3:0];
                                BREADY_M <= 1;
                        end
                end
        end
end

// Address-signals, and keeping track of current address being written to.
always @(posedge clk or posedge reset) begin
```

```
        if (reset) begin
                AWLEN_M <= 4'b1111; // Default is burst-mode
                AWVALID_M <= 0;
                AWADDR_M <= 0;
                AWID_M <= 0;
                currentaddress <= 0;
        end else begin
                if (writestate == `WRITESTART) begin
                        AWLEN_M <= 4'b1111;
                        AWVALID_M <= 1;
                        AWADDR_M <= currentaddress;
                end else if (writestate == `STATUSADDRESS) begin
                        AWLEN_M <= 4'b0000;
                        AWVALID_M <= 1;
                        AWADDR_M <= statusaddress;

                end else begin
                        AWVALID_M <= 0;
                        AWADDR_M <= AWADDR_M;
                end

                if (writestate == `WAIT) begin
                        if (writeready || statuswrite) begin
                                AWID_M <= AWID_M + 1;
                        end
                end else begin
                                AWID_M <= AWID_M;
                end
                if (writestate == `WRITELAST) begin
                        if ((currentaddress + 128) < endaddress) begin
                                currentaddress <= currentaddress + 64;
                        end else begin
                                currentaddress <= startaddress;
                        end
                end else if (startaddress != startaddressp) begin // Updating currentaddress if
                        currentaddress <= startaddress;           // the start address has changed
                end else begin
                        currentaddress <= currentaddress;
                end
        end
end


// Write-signals, along with getting and assigning data from FIFO.
always @(posedge clk or posedge reset)
begin
        if(reset)
        begin
                burstcounter <= 0;
                WLAST_M <= 0;
                WVALID_M <= 0;
                WDATA_M <= 0;
                fiforead <= 0;
        end
        else begin
```

```
                if (writestate == `WAIT)
                begin
                        WLAST_M <= 0;
                        WVALID_M <= 0;
                        burstcounter <= 0;
                        if (writeready)
                        begin
                                fiforead <= 1;
                        end
                end
                if (writestate == `WRITESTART)
                        begin
                        WVALID_M <= 0;
                        if (AWREADY_M)
                        begin
                                fiforead <= 1; // Start reading from the FIFO (so data is available)
                        end
                end
                if (writestate == `WRITE1) begin
                        WDATA_M <= fifoout;
                        if (fifoenable && !fifoempty) begin
                                WVALID_M <= 1;
                        end else begin
                                WVALID_M <= 0;
                        end

                        if ((WVALID_M && WREADY_M && fifoempty) || (!fiforead)) begin

                                if (!fifoempty && WREADY_M && (burstcounter < 14)) begin
                                        fiforead <= 1;
                                        burstcounter <= burstcounter + 1;
                                end else begin
                                        fiforead <= 0;
                                end
                        end else if (WVALID_M && WREADY_M && (burstcounter < 14)) begin
                                fiforead <= 1;
                                burstcounter <= burstcounter + 1;
                        end else if (!WREADY_M) begin
                                fiforead <= 0;
                        end
                end

                if (writestate == `WRITELAST) begin
                        WDATA_M <= fifoout;
                        fiforead <= 0;
                        WVALID_M <= 1;
                        WLAST_M <= 1;
                end
                if (writestate == `RESPONSE) begin
                        WVALID_M <= 0;
                        WLAST_M <= 0;
                end

        end
end
```

A.5 Peak Detection

A.5.1 Cross-Correlation

The bins are here represented by registers (bin_1, bin_2, etc) while the "result*n*" and "out"-values are outputs from multipliers.

```
bin_1 <= result1;
  bin_2 <= result2 + out2;
  bin_3 <= bin_3 + result3 - bin_3_out;
  bin_4 <= bin_4 + result4 - bin_4_out;
  bin_5 <= bin_5 + result5 - bin_5_out;
  bin_6 <= bin_6 + result6 - bin_6_out;
  bin_7 <= bin_7 + result7 - bin_7_out;
  bin_8 <= bin_8 + result8 - bin_8_out;
  bin_9 <= bin_9 + result9 - bin_9_out;
  bin_10 <= bin_10 + result10 - bin_10_out;
  bin_11 <= bin_11 + result11 - bin_11_out;
  bin_12 <= bin_12 + result12 - bin_12_out;
  bin_13 <= bin_13 + result13 - bin_13_out;
  bin_14 <= bin_14 + result14 - bin_14_out;
  bin_15 <= bin_15 + result15 - bin_15_out;
  bin_16 <= bin_16 + result16 - bin_16_out;
  bin_17 <= bin_17 + result17 - bin_17_out;
  bin_18 <= bin_18 + result18 - bin_18_out;
  bin_19 <= bin_19 + result19 - bin_19_out;
  bin_20 <= bin_20 + result20 - bin_20_out;
  bin_21 <= bin_21 + result21 - bin_21_out;
  bin_22 <= bin_22 + result22 - bin_22_out;
  bin_23 <= bin_23 + result23 - bin_23_out;
  bin_24 <= bin_24 + result24 - bin_24_out;
  bin_25 <= bin_25 + result25 - bin_25_out;
  bin_26 <= bin_26 + result26 - bin_26_out;
  bin_27 <= bin_27 + result27 - bin_27_out;
  bin_28 <= bin_28 + result28 - bin_28_out;
  bin_29 <= bin_29 + result29 - bin_29_out;
  bin_30 <= result30 + out30;
  bin_31 <= result31;
```

A.5.2 Signal Energy Calculation

"SIG_BITS" represents the configurable amount of significant bits used for immediate calculations. Here it is used to access the most significant bit.

```
assign result = ((
((bin1[`SIG_BITS] * (~(bin1) + 1)) | ((1 - bin1[`SIG_BITS]) * bin1)) +
((bin2[`SIG_BITS] * (~(bin2) + 1)) | ((1 - bin2[`SIG_BITS]) * bin2)) +
((bin3[`SIG_BITS] * (~(bin3) + 1)) | ((1 - bin3[`SIG_BITS]) * bin3)) +
((bin4[`SIG_BITS] * (~(bin4) + 1)) | ((1 - bin4[`SIG_BITS]) * bin4)) +
((bin5[`SIG_BITS] * (~(bin5) + 1)) | ((1 - bin5[`SIG_BITS]) * bin5)) +
((bin6[`SIG_BITS] * (~(bin6) + 1)) | ((1 - bin6[`SIG_BITS]) * bin6)) +
((bin7[`SIG_BITS] * (~(bin7) + 1)) | ((1 - bin7[`SIG_BITS]) * bin7)) +
((bin8[`SIG_BITS] * (~(bin8) + 1)) | ((1 - bin8[`SIG_BITS]) * bin8)) +
((bin9[`SIG_BITS] * (~(bin9) + 1)) | ((1 - bin9[`SIG_BITS]) * bin9)) +
((bin10[`SIG_BITS] * (~(bin10) + 1)) | ((1 - bin10[`SIG_BITS]) * bin10)) +
((bin11[`SIG_BITS] * (~(bin11) + 1)) | ((1 - bin11[`SIG_BITS]) * bin11)) +
((bin12[`SIG_BITS] * (~(bin12) + 1)) | ((1 - bin12[`SIG_BITS]) * bin12)) +
((bin13[`SIG_BITS] * (~(bin13) + 1)) | ((1 - bin13[`SIG_BITS]) * bin13)) +
((bin14[`SIG_BITS] * (~(bin14) + 1)) | ((1 - bin14[`SIG_BITS]) * bin14)) +
((bin15[`SIG_BITS] * (~(bin15) + 1)) | ((1 - bin15[`SIG_BITS]) * bin15)) +
((bin16[`SIG_BITS] * (~(bin16) + 1)) | ((1 - bin16[`SIG_BITS]) * bin16)) +
((bin17[`SIG_BITS] * (~(bin17) + 1)) | ((1 - bin17[`SIG_BITS]) * bin17)) +
((bin18[`SIG_BITS] * (~(bin18) + 1)) | ((1 - bin18[`SIG_BITS]) * bin18)) +
((bin19[`SIG_BITS] * (~(bin19) + 1)) | ((1 - bin19[`SIG_BITS]) * bin19)) +
((bin20[`SIG_BITS] * (~(bin20) + 1)) | ((1 - bin20[`SIG_BITS]) * bin20)) +
((bin21[`SIG_BITS] * (~(bin21) + 1)) | ((1 - bin21[`SIG_BITS]) * bin21)) +
((bin22[`SIG_BITS] * (~(bin22) + 1)) | ((1 - bin22[`SIG_BITS]) * bin22)) +
((bin23[`SIG_BITS] * (~(bin23) + 1)) | ((1 - bin23[`SIG_BITS]) * bin23)) +
((bin24[`SIG_BITS] * (~(bin24) + 1)) | ((1 - bin24[`SIG_BITS]) * bin24)) +
((bin25[`SIG_BITS] * (~(bin25) + 1)) | ((1 - bin25[`SIG_BITS]) * bin25)) +
((bin26[`SIG_BITS] * (~(bin26) + 1)) | ((1 - bin26[`SIG_BITS]) * bin26)) +
((bin27[`SIG_BITS] * (~(bin27) + 1)) | ((1 - bin27[`SIG_BITS]) * bin27)) +
((bin28[`SIG_BITS] * (~(bin28) + 1)) | ((1 - bin28[`SIG_BITS]) * bin28)) +
((bin29[`SIG_BITS] * (~(bin29) + 1)) | ((1 - bin29[`SIG_BITS]) * bin29)) +
((bin30[`SIG_BITS] * (~(bin30) + 1)) | ((1 - bin30[`SIG_BITS]) * bin30)) +
((bin31[`SIG_BITS] * (~(bin31) + 1)) | ((1 - bin31[`SIG_BITS]) * bin31)) ) >> 8);
```

Bibliography

[1] (2006). The PASTA Project, [Online]. Available: http://groups.inf.ed.ac.uk/pasta/ (visited on 22/05/2012).

[2] H. Elgala, R. Mesleh and H. Haas, "Indoor optical wireless communication: potential and state-of-the-art," *Communications Magazine, IEEE*, vol. 49, no. 9, pp. 56 –62, Sep. 2011, ISSN: 0163-6804. DOI: 10.1109/MCOM.2011.6011734.

[3] "AMBA AXI Protocol," ARM, Specification, Mar. 2010.

[4] C. L. Quale, "Phase one project report - prototype design of a system on chip for visible light communication," School of Engineering, The University of Edinburgh, Report, Aug. 2012.

[5] O. Almer, "Automated application-specific optimisation of interconnects in multi-core systems," PhD thesis, School of Informatics, 2012.

[6] N. Topham, "Encore: a low-power extensible embedded processor," Talk on EnCore at the HiPEAC Industrial Workshop in Wroclaw, Poland, Oct. 2009. [Online]. Available: http://groups.inf.ed.ac.uk/pasta/pub/hipeac-wroclaw.pdf.

[7] Xilinx Inc. (2009). Virtex-6 FPGA ML605 evaluation kit, [Online]. Available: http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm (visited on 29/12/2012).

[8] E. Leaver, "Customisation of a system on chip supporting a visible light communication application," School of Engineering, The University of Edinburgh, Report, Jan. 2013.

[9] "Wireless lan medium access control (MAC) and physical layer (PHY) specifications," IEEE Computer Society, Standard 802.11a-1999(R2003), Jun. 2003.

[10] R. Thakur and K. Khare, "Frame detection for synchronization in ofdm," *International Journal of Engineering Science*, vol. 3, 2011.

[11] E. Nadernejad and H. Hassanpour, "A comparison and analysis of different pde-based approaches for image enhancement," *ICSPCS Australia*, 2007.

[12] S.-J. Lee and S.-Y. Jung, "A SNR analysis of the visible light channel environment for visible light communication," in *Communications (APCC), 2012 18th Asia-Pacific Conference on*, Oct. 2012, pp. 709 –712. DOI: 10.1109/APCC.2012.6388286.

[13] T.-H. Do and M. Yoo, "Received power and snr optimization for visible light communication system," in *Ubiquitous and Future Networks (ICUFN), 2012 Fourth International Conference on*, 2012, pp. 6 –7. DOI: 10.1109/ICUFN.2012.6261652.

[14] J. Vučić, C. Kottke, S. Nerreter, K.-D. Langer and J. W. Walewski, "513 mbit/s visible light communications link based on dmt-modulation of a white led," *Journal of Lightwave Technology*, vol. 28, no. 24, pp. 1512 –1518, Dec. 2010.

[15] J. Vučić, C. Kottke, S. Nerreter, A. Buttner, K. Langer and J. W. Walewski, "White light wireless transmission at 200 mb/s net data rate by use of discrete-multitone modulation," *Photonics Technology Letters, IEEE*, Oct. 2009.

[16] H. Elgala, R. Mesleh and H. Haas, "Indoor broadcasting via white leds and ofdm," *Consumer Electronics, IEEE Transactions on*, vol. 55, no. 3, pp. 1127 –1134, Aug. 2009, ISSN: 0098-3063. DOI: 10.1109/TCE.2009.5277966.

[17] Y. Tanaka, T. Komine, S. Haruyama and M. Nakagawa, "Indoor visible light data transmission system utilizing white led lights," *IEICE transactions on communications*, vol. 86, no. 8, pp. 2440–2454, 2003.

[18] A. Radulescu, J. Dielissen, S. Pestana, O. Gangwal, E. Rijpkema, P. Wielage and K. Goossens, "An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 4 –17, 2005, ISSN: 0278-0070. DOI: 10.1109/TCAD.2004.839493(410)24.

[19] F. ming Xiao, D. sheng Li, G. ming Du, Y. kun Song, D. li Zhang and M. lun Gao, "Design of axi bus based mpsoc on fpga," in *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, 2009, pp. 560 –564. DOI: 10.1109/ICASID.2009.5277006.

[20] P. Harrod, "Testing reusable ip-a case study," in *Test Conference, 1999. Proceedings. International*, 1999, pp. 493 –498. DOI: 10.1109/TEST.1999.805772.

[21] S. Pasricha, N. Dutt and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," in *Proceedings of the 41st annual Design Automation Conference*, ACM, 2004, pp. 113–118.

[22] *Tms320c6713 dsp starter kit*, Product, Texas Instruments, 2012.

[23] *Casio picapicamera - app information*, Online Product Information, Casio, 2012.

[24] (May 2012). Lightmessage by purevlc, [Online]. Available: http://visiblelightcomm.com/lightmessage-by-purevlc/ (visited on 31/12/2012).

[25] Siemens. (2010). 500 megabits/second with white led light, [Online]. Available: http://www.siemens.com/innovation/en/news/2010/500-megabits-second-with-white-led-light.htm (visited on 31/12/2012).

[26] I. Synopsys, *Synopsys enhances designware ethernet ip with support for ieee 1588 specification and arm amba 3 axi interface*, Press Release, Jan. 2009. [Online]. Available: http://news.synopsys.com/index.php?s=43&item=642.

[27] *Logicore ip axi ethernet lite mac*, Data sheet, Xilinx, Jul. 2012.

[28] *Logicore ip axi universal serial bus (usb) 2.0 device*, Data sheet, Xilinx, Oct. 2012.

[29] *Primecell uart technical reference manual*, version r1p5, ARM, 2007.