FPGA Implementation of MIMO Wireless Communications System

Ian Griffiths
Supervised by Assoc. Prof. Brett Ninness

November 1, 2005

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Engineering in Computer Engineering at The University of Newcastle, Australia.

Abstract

Wireless communications have grown tremendously over the last decade, wireless LAN and mobile telephones have been the main reasons for the growth. There is a demand for ever faster wireless communications as this will allow for new applications such as widespread wireless broadband Internet access.

Multi-Antenna transmission schemes, using multiple antennas at the transmitter and/or receiver, have been proposed as a way to fulfill the demand for increased capacity. They are particularly attractive because they do not require any additional transmission bandwidth, and unlike traditional systems use multi-path interference to their benefit.

The aim of this project is to implement a particular multi-antenna scheme, a 2×2 Alamouti code, on a PCI testbed card developed by the University. The testbed is very flexible, most of the computing power is provided by a 600,000 gate Xilinx FPGA. There are also 12 sockets that can be used for radio transceiver modules, or custom ASICs.

At the time of writing, designs have been created for all the major components of a MIMO system except for a channel estimator. The designs have been verified by simulation, both before mathematical simulation, and behavioural simulation of VHDL code. The simulation results have been favourable with the MIMO scheme significantly outperforming the equivalent SISO scheme.

Key Contributions

The key contributions I have made to this project are:

- Creation of Octave (MATLAB) simulation of a MIMO wireless communications system using the Alamouti code.
- Implementation of the components of MIMO system using the C programming language, allowing bit accurate simulation of final hardware design
- Design of hardware implementation of components of MIMO system and writing VHDL code to implement these designs

Ian Griffiths

Brett Ninness

Contents

Chapter 1

Introduction

1.1 Motivation

In recent years the telecommunications industry has experienced phenomenal growth, particularly in the area of wireless communication. This growth has been fueled by the widespread popularity of mobile telephones and wireless computer networking.

However, there are limits to growth, and the radio spectrum used for wireless communications is a finite resource. Therefore considerable effort has been invested in making more efficient use of it. Using the spectrum more efficiently caters for the ever increasing demand for faster communications since more bits per second can be transmitted using the same bandwidth.

Recently a major research focus in this area has been the use of multiple antennas for transmitting and receiving instead of the traditional single antenna systems [1]. It has been proposed that using multiple transmit and receive antennas, and associated coding techniques could increase the performance of wireless communication systems [3, 6, 7, 8]. So far there has been a lot of theoretical research but relatively few practical systems have been demonstrated [4, 5].

The university has undertaken a research project to create a testbed for multi-antenna wireless communications. The outcome of this project is a PCI card with a programmable logic chip and sockets for multiple pluggable modules that can be used for radio transceivers or custom signal processing hardware.

I will be implementing a particular scheme known as the "Alamouti scheme" (see Section 2.5 for more detail). It is one of the simplest multi-antenna schemes, as it uses only 2 transmit and 2 receive antennas.

Chapter 2

Theoretical Background

In this chapter the theory underlying this project and the MIMO system being implemented will be examined. We will begin with a brief overview of the capacity of wireless communication systems, and examine the environments in which they are used. Finally the theory of multi-antenna communications is introduced. Particular attention is paid to the Alamouti code, and associated techniques such as channel estimation.

2.1 Capacity of Wireless Communication Systems

In 1948 Claude Shannon discovered that there was an upper limit to the capacity of a channel for error free transmission of information:

$$C = B \log_2(1 + SNR) \tag{2.1}$$

where B is the transmission bandwidth, and SNR is the signal to noise ratio of the channel. This equation gives the absolute maximum capacity of the channel (in bits/second). Thus it appears the only way to increase the capacity of the communications system is to increase the bandwidth used in transmission, or to increase SNR.

Multi-Antenna systems use a rather novel approach to increase the overall capacity of a wireless communications system; use more channels. Each of the individual transmission channels is still limited according to Equation 2.1, however the overall capacity of the system is now the sum of the capacities of the individual channels.

In the case of multi-antenna systems these individual channels are not totally separate transmission channels. Instead, these systems exploit multi-path propagation to provide independent channels even though the radio signals are being sent across the

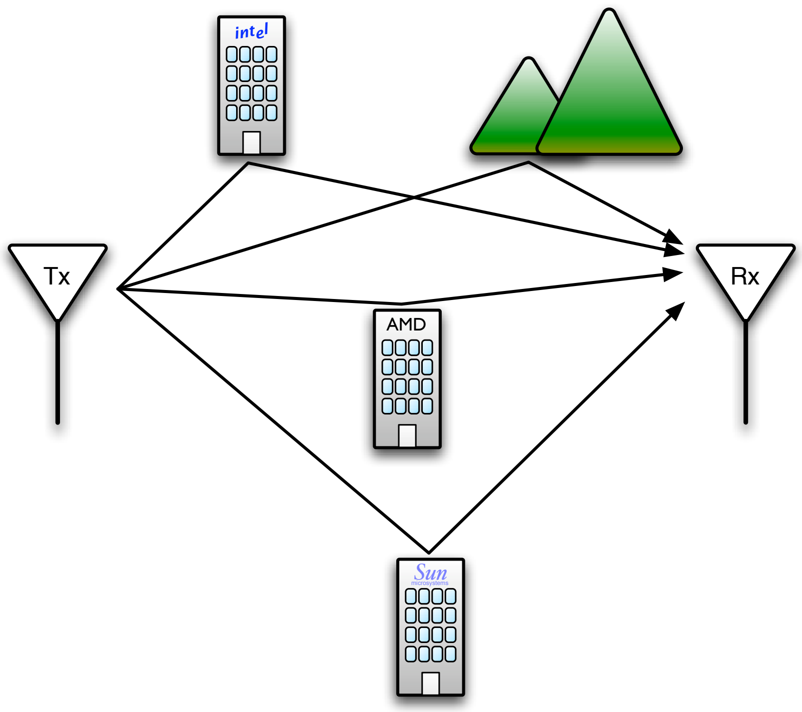

Figure 2.1: Simplified example of multi-path propagation

same transmission environment.

2.2 The Transmission Environment

It is useful to understand a little about the transmission environment of a modern wireless communication system before investigating how multi-antenna systems work. As stated in the introduction, the major drivers of wireless communication are mobile telephones and wireless LANs (e.g. IEEE 802.11b otherwise known as Wi-Fi), therefore it is prudent to examine the typical transmission environments in which these systems operate.

The wireless environment in which these technologies operate (urban settings) is typically characterised by multi-path propagation. As the name suggests, multi-path propagation occurs when there are multiple transmission paths between the transmitter and the receiver. In an urban environment this is typically caused by the radio waves reflecting off buildings and other obstacles. A simplified example of this effect can been seen in Figure 2.1

In a traditional single antenna system (henceforth referred to as Single Input, Single Output or SISO) multi-path propagation can be a problem as it causes Inter-Symbol Interference. The traditional response to multi-path interference has been to lengthen the symbol period so that most of the reflections have died out before the symbol is sampled at the receiver. Obviously, unless other measures are taken, this will reduce the data rate of the system.

Multi-Antenna systems (referred to as Multiple Input, Multiple Output or MIMO[1]) however, use multi-path propagation to their benefit, and in fact rely on some amount being present.

2.3 Modelling the Wireless Communications Channel

Under certain assumptions the complicated transmission environment can be mathematically modelled by using complex numbers to represent the magnitude and phase change of the transmission channel. The assumption made by this model is that the channel is a so called “flat fading” channel.

Flat fading refers to the frequency response of the channel being “flat”, meaning that all frequencies are subjected to the same attenuation. One of the side effects of flat fading is that there is no Inter-Symbol Interference (ISI).

Even if the actual transmission environment is not flat fading this model can still be used provided the bandwidth of the transmitted signal is small enough. In particular the bandwidth needs to be less than the inverse of the delay spread[2] of the channel for the flat-fading assumption to hold. This means that there should be negligible ISI.

The use of complex numbers in the model derives from the fact that it is possible to represent a real-valued bandpass signal using complex numbers, see appendix A.1 in [2]. It is from this complex number representation that the “in-phase” and “quadrature” components of a signal are derived. The in-phase component is the real part of the complex representation, and the quadrature component is the imaginary part.

For a SISO system this model can reduce the entire transmission environment to a single complex number. The system can then be represented using Equation 2.2, where h is the complex number representing the channel, x is the input signal, and e is a

[1]I will generally refer to MIMO systems, which have multiple antennas at both transmitter and receiver, however it is also possible to have Multiple Input, Single Output (MISO) or Single Input, Multiple Output (SIMO) systems. Much of the theory applies to these systems also.

[2]The delay spread of a channel is the elapsed time between when the first and last of the multi-path reflections arrive at the receiver.

complex number modelling the thermal noise at the receiver.

$$y = hx + e \tag{2.2}$$

Similarly MIMO systems can be modelled with Equation 2.3. The variables have the same meaning as for the SISO case, however instead of the scalar complex numbers in Equation 2.2 the variables are matrices of complex numbers.

$$Y = HX + E \tag{2.3}$$

2.4 Multi-Antenna Systems

One possible way to improve the reliability of wireless communications is to employ *diversity*. Diversity is the technique of transmitting the same information across multiple channels to achieve higher reliability. It operates on the principle that it is unlikely that all of the channels used to transmit the redundant information will be experiencing *deep fading*[3] at the same time. Even if one particular channel is unusable the information may still be recovered from the redundant transmission over the other channels. Therefore the overall reliability of the communications system is improved, at the cost of transmitting redundant information.

If multiple antennas are used at the transmitter or receiver there are potentially multiple transmission channels between the transmitter and receiver. See Figure 2.2 for an example of the potential channels in a 2×2 MIMO system. These multiple channels can be used to exploit diversity.

In the 2×2 system in Figure 2.2 there is the potential for both transmit and receive diversity. Receive diversity is when the same information is received by different antennas. For instance the information sent from Tx_1 is transmitted across channels $h_{1,1}$ and $h_{1,2}$, and received by both Rx_1 and Rx_2. Transmit diversity is when the same information is sent from multiple transmit antennas. One possible way to achieve this is to code across multiple symbols periods. For instance, at time t antenna Tx_1 could transmit the symbol s then at time $t + 1$ antenna Tx_2 would transmit the same symbol, s. The Alamouti scheme uses a method similar to this to obtain transmit diversity.

MIMO systems are able to achieve impressive improvements in reliability and capacity by exploiting the diversity offered by the multiple channels between the transmit and

[3]Wireless channels are time varying, and occasionally the channel gain may drop to zero. This is called deep fading, and makes the channel unable to transmit any useful information.

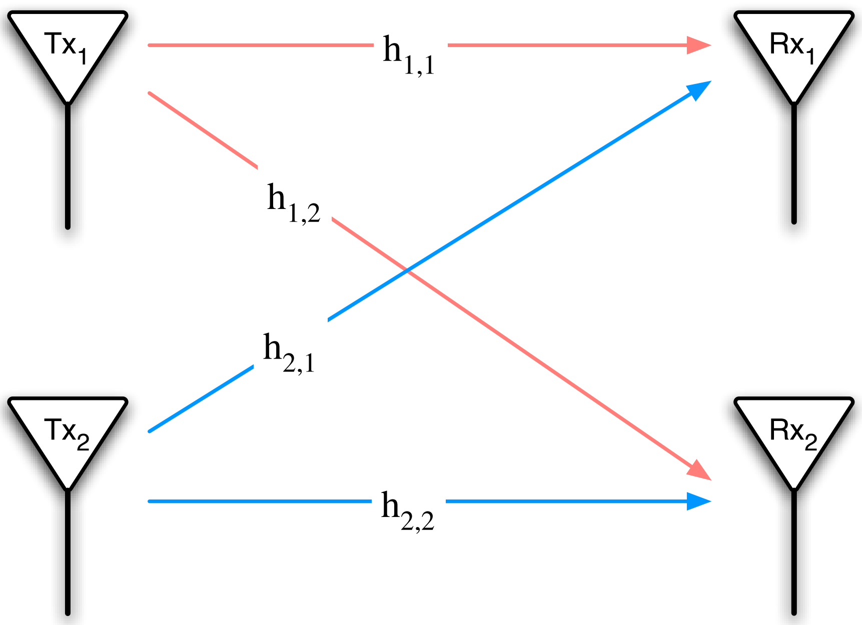

Figure 2.2: Potential Communications Channels in a 2×2 MIMO system

receive antennas. Different coding schemes vary in their exact approaches, however all seek to use the available channels to increase capacity and/or reliability.

2.5 The Alamouti Code

The coding scheme implemented in this project is an Alamouti code, therefore this code will be examined in closer mathematical detail.

The Alamouti code, so called because it was proposed by S.M Alamouti in [7], belongs to a class of codes called Space-Time Block Codes (STBC). The *Space-Time* refers to coding across space and time. Coding across space by using multiple transmit and receive antennas, and across time by using multiple symbol periods. Like normal block codes the Alamouti code operates on blocks of input bits, however rather than having 1 dimensional code vectors it has 2 dimensional code matrices.

STBCs can be described by a code matrix, which defines what is to be sent from the transmit antennas during transmission of a block. The code matrix is of dimension $N_t \times t_b$ where N_t is the number of transmit antennas and t_b is the number of symbol periods used to transmit a block. So the rows of the matrix represent the transmit antennas, and the columns are the time (symbol) periods.

The code matrix for the Alamouti code is given in Equation 2.4.

$$X = \begin{bmatrix} s_1 & -s_2^* \\ s_2 & s_1^* \end{bmatrix} \tag{2.4}$$

The code belongs to a special subclass of STBCs known as Orthogonal Space Time Block Codes (OSTBC). The code matrices of OSTBCs satisfy the following constraint.

$$XX^H = \sum_{n=1}^{n_s} |s_n|^2 \cdot (\alpha I) \tag{2.5}$$

where n_s is the number of symbols, s_n is the nth complex symbol, α is an arbitrary constant and $(.)^H$ denotes the Hermitian conjugate[4].

There are a number of properties that make OSTBCs particularly interesting. Foremost is that Maximum Likelihood (ML) detection of different symbols is *decoupled.* In the case of the Alamouti code this means that the two symbols which are coded together can be detected independently at the receiver. In other words the same techniques used to detect symbols one at a time in a SISO scheme can be used in the Alamouti scheme as well.

Using Equations 2.3 and 2.4 the received matrix in a 2x2 system can be written as

$$Y = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} s_1 & -s_2^* \\ s_2 & s_1^* \end{bmatrix} + \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} \tag{2.6}$$

now, let

$$r_{11} \triangleq h_{11}s_1 + h_{12}s_2 + e_{11} \tag{2.7}$$

$$r_{12} \triangleq -h_{11}s_2^* + h_{12}s_1^* + e_{12} \tag{2.8}$$

$$r_{21} \triangleq h_{21}s_1 + h_{22}s_2 + e_{21} \tag{2.9}$$

$$r_{21} \triangleq -h_{21}s_2^* + h_{22}s_1^* + e_{22} \tag{2.10}$$

These are the signals that are received by each of the antennas at the receiver across the two time periods. The above expressions can be obtained by expanding Equation 2.6. The first digit of the subscript denotes the receive antenna, and the second digit is the

[4]The Hermitian conjugate of a matrix is the complex conjugate, transpose, i.e. $X^H = (X^*)^T$

time period when the signal is received. Equation 2.6 can now be re-written as

$$Y = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix} \tag{2.11}$$

In [7] Alamouti states that the transmitted symbols $s_1$and s_2 can be estimated in a maximum likelihood fashion by first combining the received signals according to the following equations

$$\tilde{s_1} = h_{11}^* r_{11} + h_{12} r_{12}^* + h_{21}^* r_{21} + h_{22} r_{22}^* \tag{2.12}$$

$$\tilde{s_2} = h_{12}^* r_{11} - h_{11} r_{12}^* + h_{22}^* r_{21} - h_{21} r_{22}^* \tag{2.13}$$

and then using a standard Maximum Likelihood detector to attempt to recover s_1 and s_2 from $\tilde{s_1}$ and $\tilde{s_2}$. This is the *decoupled* ML detection that is common to all OSTBCs.

The validity of Alamouti's proposed system can been seen by substituting the values of r_{11}, r_{12}, r_{21} and r_{22} from Equations 2.7, 2.8, 2.9 and 2.10 into Equations 2.12 and 2.13 to obtain the following.

$$\begin{aligned} \tilde{s_1} &= h_{11}^*(h_{11}s_1 + h_{12}s_2 + e_{11}) \\ &\quad + h_{12}(-h_{11}^* s_2 + h_{12}^* s_1 + e_{12}^*) \\ &\quad + h_{21}^*(h_{21}s_1 + h_{22}s_2 + e_{21}) \\ &\quad + h_{22}(-h_{21}^* s_2 + h_{22}^* s_1 + e_{22}^*) \\ &= s_1(|h_{11}| + |h_{12}| + |h_{21}| + |h_{22}|) \qquad (2.14) \\ &\quad + h_{11}^* e_{11} + h_{12} e_{12}^* + h_{21}^* e_{21} + h_{22} e_{22}^* \end{aligned}$$

similarly

$$\begin{aligned} \tilde{s_2} &= s_2(|h_{11}| + |h_{12}| + |h_{21}| + |h_{22}|) \qquad (2.15) \\ &\quad - h_{11} e_{12}^* + h_{12}^* e_{11} - h_{21} e_{22}^* + h_{22}^* e_{21} \end{aligned}$$

Equations 2.14 and 2.15 show that when the received signals are combined according to Equations 2.12 and 2.13 the transmitted symbols are combined coherently and weighted by a positive factor, i.e. $|h_{11}| + |h_{12}| + |h_{21}| + |h_{22}|$. The noise samples however, get combined in an incoherent manner. This is how the Alamouti scheme is able to achieve an improvement in performance over SISO systems.

2.6 Channel Estimation

To use the equations in the above section to decode the received signal the receiver needs to have so-called *channel knowledge.* This means the values of the h_{xy} terms in Equation 2.6 must be known. In practice it is not possible to obtain exact values for these terms, however they can be estimated.

There are a number of methods for estimating the channel matrix, the simplest being training based estimation. With training based channel estimation a data block known to both the transmitter and receiver, called the training block, is transmitted before the start of the actual data in each code block. The channel matrix can then be estimated at the receiver using the following equation.

$$\hat{H} = Y_t X_t^H (X_t X_t^H)^{-1} \tag{2.16}$$

where X_t is the known training block sent by the transmitter, Y_t is the received training block, and $(.)^H$ denotes the Hermitian conjugate.

Equation 2.16 relies on a the training block being designed to satisfy the following equation

$$X_t X_t^H = \rho^2 I \tag{2.17}$$

fortunately, by design the Alamouti code matrix, and any other OSTBC, satisfies this equation. So a possible training block is simply a known pre-amble prepended before the actual data.

The validity of this method for channel estimation can be seen by substituting Equations 2.3, and 2.17 into Equation 2.16.

$$\begin{aligned}
\hat{H} &= (HX_t + E)X_t^H (X_t X_t^H)^{-1} \\
&= HX_t X_t^H (X_t X_t^H)^{-1} + EX_t^H (X_t X_t^H)^{-1} \\
&= H + EX_t^H (\rho^2 I)^{-1} \\
&= H + error\, term
\end{aligned}$$

The channel estimate obtained via this method can then be used in the detector described in Section 2.5. This method is not optimal in a maximum likelihood sense, however it is fairly easy to understand and implement.

Chapter 3

Newcastle University Wireless Testbed Project

This chapter will review the wireless testbed that is the target device for this project. First the motivations for creating the testbed are explained, then there will be a brief overview of the hardware present on the card. Finally some related final year projects are mentioned.

3.1 Motivation for Testbed

The reasons for wanting a device to be able to conduct practical testing of MIMO systems are obvious, however, there are many different approaches to building such a device ranging in complexity, cost and flexibility.

In [5] the authors put forward a classification scheme for different types of testbeds. The simplest approach they recognised is targeted towards burst mode transmissions, and offline signal processing. This design minimises the cost, however it also severely limits the scenarios in which the testbed can be used, because the signal processing is not done in real-time. The testbed card used in this project is much more powerful and provides for real-time operation, using a Field Programmable Gate Array (FPGA) chip to perform the signal processing. Thus it lies towards the opposite end of the spectrum presented by the authors.

Employing a more sophisticated approach allows the testbed card to more accurately reflect the environments where the MIMO algorithms are likely to be implemented. Not only is real-time transmission and decoding possible, but the hardware present is similar to the final deployment environment. Typically the deployment environments will have

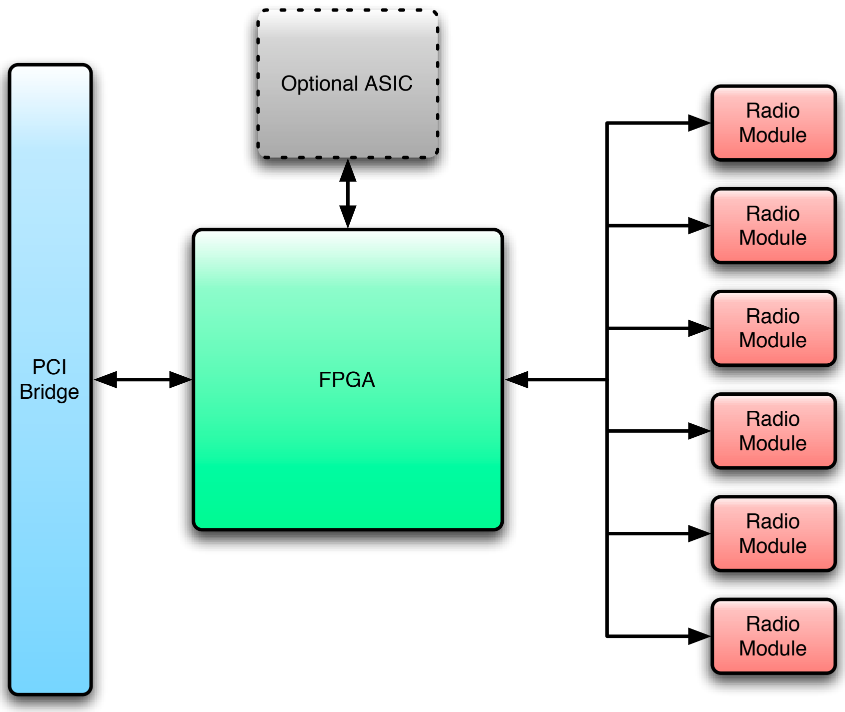

Figure 3.1: Testbed Block Diagram

limited computing power, or use Application Specific Integrated Circuits (ASICs). The testbed card has an FPGA for signal processing. Typically FPGAs are used as an intermediate step in the development of ASICs so the testbed card will also be valuable in the development of ASICs.

3.2 Testbed Hardware

The testbed that has been developed at the university has been designed for flexibility. There are sockets for 12 expansion modules on the card. These sockets may be used for radio transceiver modules, or a custom ASIC, or numerous other possibilities.

The main computing power of the board comes from a programmable logic device, which can be easily reconfigured to implement any coding scheme, even SISO schemes. In addition an ASIC may be added to the board to provide additional signal processing capabilities.

A block diagram of the architecture of the testbed can be seen in Figure 3.1. This shows only one possible configuration of the card, however, it gives an idea of the reconfigurability of the testbed. The radio modules may be swapped for different units, or even exchanged for a Digital Signal Processing (DSP) chip, or something else entirely. The FPGA, which is central, can be reprogrammed to perform different tasks, or route signals in different directions. The only function that is fixed is the PCI bridge, however this is obviously not a drawback as the PCI standard is somewhat fixed.

The testbed also has provisions for using a custom ASIC, which will not be used in this project. However, in the long term this expandability will greatly increase the possible applications for the testbed. In addition to being used as a prototyping tool the testbed could be used to easily verify ASIC designs in a realistic setting before they go into large scale production.

The radio modules used in this project are based on a commercially available 2.4GHz transceiver (Maxim MAX2822). These chips are compatible with the physical layer of the IEEE 802.11b standard for wireless networking. However they are not being used in this manner on the testbed, rather they are being used simply as radio transmitters and receivers.

3.3 Related Final Year Projects

A number of other students have worked on the testbed at various stages of its development. While my work stands alone to some degree, it also relies on the work of these students. Therefore it is prudent to reference their work.

In 2004 Chris Shaw completed a final year project entitled *"Linux Device Driver for Wireless Testbed"*. He worked on a Graphical User Interface (GUI) program to ease the use of the testbed hardware, and extended a driver written by Alan Murray in 2003/2004. However, at the time there was no hardware available to him, so he implemented a simulation of the hardware in the driver.

This year in his project titled *"Linux Device Driver and Graphical Interface Support for Research Testbed"* Nathan Tomkins is re-implementing much of Chris' work. He is porting Alan Murray's driver to the 2.6 series Linux kernel[1], and writing a new GUI using the Python programming language.

In addition to these students John Dalton has been working on the testbed hardware. He designed the testbed card and is also carrying out testing.

[1]The original driver was based on the 2.4 series Linux kernel, however since it was written nearly every Linux distribution has switched to the newer 2.6 kernels. Thus it is becoming rather difficult to use the driver, a situation which will only get worse with time.

Chapter 4

Simulation

I followed the general hardware design process in this project, the first stage of which is to conduct a high-level simulation of the proposed design to work through any algorithmic or mathematical issues. Typically this simulation is produced using MATLAB, or a similar maths package.

After the simulation is completed and the algorithm is correct the next stage is to move onto a low-level "bit accurate" C implementation. Bit accurate refers to the fact that for a given set of input bits the C implementation will produce the correct output bits. This step is used because typically C is much easier to write and debug than Hardware Description Language (HDL) code such as VHDL.

The next stage is to implement the design using the chosen HDL, in this case VHDL. The bit accurate C code is used to verify that the VHDL is correct by comparing the outputs of the two implementations.

Once the VHDL is debugged in simulation and producing the correct output the design can be uploaded to the FPGA for final testing.

In this chapter the simulations, both high- and low-level that were created during the project will be examined, and the results obtained will be presented.

4.1 High-Level Simulation

The initial high-level simulation was implemented using Octave, an open source equivalent of MATLAB. The code used for simulation can be found in Appendix A.

4.1.1 Alamouti Encoder

The first component in the system that was simulated was the encoder. This was chosen first as it is a fairly simple component.

There are two distinct steps in the encoding process. First the input bits are modulated into symbols (represented by complex numbers), then the complex symbols are encoded using the Alamouti code matrix given in Equation 2.4.

I have chosen to use a Binary Phase Shift Keying (BPSK) constellation for modulation. The main reason for using BPSK is because it is a very simple scheme. A side effect of the Alamouti scheme, which is a rate 1 code, is that the overall system has the same data rate as the SISO system using BPSK.

Initially the encoder I implemented was designed as a combined BPSK modulator and Alamouti encoder. The input bits were used to decide which of four matrices were output. The matrices were manually constructed and hard-coded into the simulation. This design made it fairly difficult to switch the modulation or coding scheme. It was also fairly error prone as the code matrices were manually constructed, and it was fairly easy to leave out a negative sign or make other simple mistakes. The main reason for using the combined design at first was because it was very simple to implement.

I revised the design to simulate the modulator and encoder separately in a slightly more modular fashion. This design allows for the modulation scheme to be easily changed, say to QPSK, or QAM. This more modular design was used at the lower level implementations also.

The source code for the simulated encoder can be found in Appendix A.1.

4.1.2 Channel Estimator

As stated in Section 2.6 the Alamouti decoder needs channel knowledge, so a channel estimator is required.

In the high-level simulation the high level features of Octave were taken advantage of and Equation 2.16 was simply converted to Octave code. This approach is not possible for the low level implementations, instead the matrix operations must be implemented manually.

The source code for the channel estimator can be found in Appendix A.2.

4.1.3 Alamouti Decoder

As with the encoder a simple, but fairly inflexible design was used initially for the decoder. This was design was chosen for the same reasons as with the encoder, simplicity and ease of simulation.

The initial decoder design used a brute force technique that was by no means optimal in a computational complexity sense. It used the channel estimate, and the four [1] possible code matrices to construct an estimate of the potential received matrices. These were then compared to the actual received matrix and the one which was "closest" was deemed to be the correct output. The "closeness" of the pairs of matrices was evaluated by taking the Frobenius norm of the difference of the two.

The final decoder design uses the method presented in Section 2.5 with a combiner and a separate symbol detector. In the case of BPSK the symbol detector can just be a simple threshold detector. This decoder design is also used in the low level implementations.

The source code for the decoder can be found in Appendix A.3.

4.2 Low-Level Simulation

The low-level simulation was carried out using programs written in C, which output data to, and read input data from plain text files. A number of supporting programs were written to enable the results to be imported into Octave for analysis and graphing.

As mentioned above the high level constructs such as matrix operations, and complex numbers had to be manually implemented for this simulation. The representation of complex numbers in particular took a number of revisions before a final structure was settled upon. The initial approach was to use the `struct` keyword of C to create a complex number "structure". This approach was discarded because this approach could not be used in the VHDL hardware design. Instead the complex numbers were simply represented as separate arrays or variables for the real and imaginary parts of each number.

The source code for the low-level simulation can be found in Appendix B

4.2.1 Alamouti Encoder

The encoder used the same design as the high-level simulation, with a separate BPSK modulator and Alamouti encoder.

The BPSK modulator took an 8-bit `char` input, and produced two arrays, representing the real and imaginary parts of the symbols, for output. It simply runs through the input testing a bit at a time. If the bit in question is a 1 then the symbol for a 1 is

[1]When using BPSK modulation there are only 4 possible code matrices (X), corresponding to the input bits 00, 01, 10, and 11.

placed into the output arrays, otherwise the symbol for a 0 is put into the output. The actual symbols that are used to represent 1 and 0 are defined in a header file, so can be easily changed.

The Alamouti encoder takes the two arrays output by the BPSK modulator as input and produces two 2-dimensional arrays as output. These arrays represent the real and imaginary parts of the symbols that are sent to the Radio Frequency (RF) "front-ends" on each of the transmit antennas of the testbed card. It loops through the input arrays operating on pairs of symbols at a time. In line with Equation 2.4 the symbols are first copied straight through to the output arrays unmodified. Then the symbols are swapped over to the opposite transmit antenna and complex conjugated, also one symbol is negated. Complex conjugation is achieved by simply negating the imaginary part of the input before placing it into the output. Also the complex conjugation, and extra negation operations are combined into a single step for the relevant symbol by negating the real part instead of the imaginary.

The source code for the low-level encoder can be found in Appendix B.2

4.2.2 Channel Estimator

When implementing the channel estimator it became obvious that if the training block was a pre-defined fixed matrix then Equation 2.16 simplifies to multiplying a matrix by another constant matrix. Equation 2.16, with the constant term highlighted, is repeated below

$$\hat{H} = Y_t \times \underbrace{X_t^H (X_t X_t^H)^{-1}}_{constant\ term} \tag{4.1}$$

This means that if the training block is fixed then the channel estimator is simply a complex matrix multiplier.

This is the basis for the design of the low-level channel estimator simulation. The training block, and the constant part of the channel estimation equation are stored in header files and are used in the code by using the `#include` directive. A small Octave script was written so that the training block could be defined in the script, then the constant term would be automatically calculated, and both then output straight into a header file ready for use. This script was then incorporated into the build process using the `Makefile`.

No real attempt was made to optimise the matrix multiplication process, three nested `for` loops were used, and one element of the output matrix was calculated at a time. It was decided that trying to optimise the C code would not be overly useful as the main purpose of the simulation was to be correct not optimal.

The source code for the low-level simulation of the Alamouti encoder can be found in Appendix B.3

4.2.3 Alamouti Decoder

The Alamouti decoder uses the same design as the decoder in the high-level simulation, with a separate "combiner" and demodulator.

The actual algorithm implemented by the combiner is fairly straightforward, however for the low-level implementation the mathematical expressions for each symbol estimate were expanded and simplified to remove the complex numbers and operations. The resulting expressions are shown in Equations 4.2 – 4.5

$$\begin{aligned}
s0_{re} &= Re\{h_{0,0}\} \times Re\{y_{0,0}\} + Im\{h_{0,0}\} \times Im\{y_{0,0}\} \\
&\quad + Re\{h_{0,1}\} \times Re\{y_{0,1}\} + Im\{h_{0,1}\} \times Im\{y_{0,1}\} \\
&\quad + Re\{h_{1,0}\} \times Re\{y_{1,0}\} + Im\{h_{1,0}\} \times Im\{y_{1,0}\} \\
&\quad + Re\{h_{1,1}\} \times Re\{y_{1,1}\} + Im\{h_{1,1}\} \times Im\{y_{1,1}\} && (4.2) \\
s0_{im} &= Re\{h_{0,0}\} \times Im\{y_{0,0}\} - Im\{h_{0,0}\} \times Re\{y_{0,0}\} \\
&\quad - Re\{h_{0,1}\} \times Im\{y_{0,1}\} + Im\{h_{0,1}\} \times Re\{y_{0,1}\} \\
&\quad + Re\{h_{1,0}\} \times Im\{y_{1,0}\} - Im\{h_{1,0}\} \times Re\{y_{1,0}\} \\
&\quad - Re\{h_{1,1}\} \times Im\{y_{1,1}\} + Im\{h_{1,1}\} \times Re\{y_{1,1}\} && (4.3) \\
s1_{re} &= Re\{h_{0,1}\} \times Re\{y_{0,0}\} + Im\{h_{0,1}\} \times Im\{y_{0,0}\} \\
&\quad - Re\{h_{0,0}\} \times Re\{y_{0,1}\} - Im\{h_{0,0}\} \times Im\{y_{0,1}\} \\
&\quad + Re\{h_{1,1}\} \times Re\{y_{1,0}\} + Im\{h_{1,1}\} \times Im\{y_{1,0}\} \\
&\quad - Re\{h_{1,0}\} \times Re\{y_{1,1}\} - Im\{h_{1,0}\} \times Im\{y_{1,1}\} && (4.4) \\
s1_{im} &= Re\{h_{0,1}\} \times Im\{y_{0,0}\} - Im\{h_{0,1}\} \times Re\{y_{0,0}\} \\
&\quad + Re\{h_{0,0}\} \times Im\{y_{0,1}\} - Im\{h_{0,0}\} \times Re\{y_{0,1}\} \\
&\quad + Re\{h_{1,1}\} \times Im\{y_{1,0}\} - Im\{h_{1,1}\} \times Re\{y_{1,0}\} \\
&\quad + Re\{h_{1,0}\} \times Im\{y_{1,1}\} - Im\{h_{1,0}\} \times Re\{y_{1,1}\} && (4.5)
\end{aligned}$$

So, after expansion and simplification, the expression for each component is essentially a sum of products.

The combiner inputs are four 2×2 arrays, two for the real and imaginary parts of the channel estimate, and two for the real and imaginary parts of the received samples.

The outputs are two 2×1 arrays, representing the real and imaginary parts of the two symbol estimates.

The BPSK demodulator part of the decoder exploits the sign bit of the 2's complement binary number format used in computers, and the fact that the BPSK constellation in use is made up of only real numbers. Because the transmitted symbols are real numbers only, the imaginary part of the input to the demodulator is discarded. Thus the demodulator simply outputs the inverse of the sign bit of the input. Therefore, any symbol with a negative real part is demodulated as 0, and any with a positive real part is demodulated as a 1.

The source code of the low-level implementation of the Alamouti decoder can be found in Appendix B.4.

4.2.4 Fixed Point

In addition to implementing the complex numbers and matrix operations manually the low-level simulation was also converted to run using fixed-point arithmetic. The reason for this conversion is because the use of floating-point arithmetic in the final hardware design is infeasible because of the associated complexity. Therefore to maintain the bit accurate nature of the simulation it must also be converted to use fixed-point.

The conversion process involved first working out the dynamic range of the numbers at each stage in the system, and trying to assess the required accuracy. This assessment needed to be done so that a fixed point number format could be chosen. The choice of number format constrains both the dynamic range, and accuracy of the numbers represented, therefore care must be taking in choosing an appropriate number format.

A few formats were evaluated in the simulation, each with varying ranges and accuracies. The aim was to find the format that used the least bits, but still provided acceptable performance. The reason for wanting as few bits as possible is to try to make the hardware implementation as simple as possible. It takes less time to multiply two 8 bit numbers than it does to multiply two 32 bit numbers, and it uses far less hardware also. Thus it is easier to have an efficient hardware implementation if the number format used has as few bits as possible. The final design uses a 16-bit format with 8 bits for the integral part, and 8 bits for the fractional part, this is known as an 8.8 fixed point format.

After the number format was chosen, all the mathematical operations needed to be converted to fixed point also. This conversion process involves making sure that the radix point is in the correct place after the operation. For addition and subtraction the radix point does not move. However multiplication and division both move the radix point, so they must be corrected. For multiplication the correction is achieved using an arithmetic right shift, for division it is a left shift. Note, there are no divisions in the algorithms being implemented, only multiplication's.

The fixed-point conversion process was performed by first writing a header file that defined the fixed point types and also some functions to convert fixed-point numbers to floating-point and vice-versa. These functions were mostly used for debugging, however the floating-to-fixed conversion functions were used to simulate the analogue to digital converters at the receiver. Finally a function that performed fixed point multiplication was written, and all the multiplication operators were replaced with calls to this function.

The fixed-point conversion was carried out on a copy of the source code of the original floating-point simulation. This resulted in one fixed point simulation and one floating point one, this was intentional. Having two copies allowed the comparison of the fixed-point implementation to the floating-point one to make sure that the fixed-point implementation performed acceptably.

The source code for the fixed point functions can be found in Appendix B.5

4.3 Simulation Results

The simulation produced the expected results, confirming that there is a considerable performance gain from using the MIMO coding scheme. Figure 4.1 shows a comparison of the simulated MIMO and SISO schemes. As can be seen from the plot the MIMO scheme achieves a much lower Bit Error Rate at the same Signal to Noise Ratio than the SISO scheme. This simulation is perhaps a little unfair on the SISO scheme as the channel model in the simulation is very simplistic, and likely much "worse" than a real channel would be. The MIMO scheme is not greatly affected by this harsh channel because it is designed to work in this kind of environment.

Also, note that there is a curve for the fixed point implementation of the MIMO system. As can be seen in Figure 4.1 the fixed point implementation performs nearly as well as the floating point one, there is only a very minor difference in BER. This difference only really becomes apparent at higher signal to noise ratios, up to roughly 15 dB the two MIMO implementations are virually indistinguishable.

This plot was created using the low-level simulation. Previously an attempt was made to create a similar plot using the high-level simulation. This was not as successful because the simulation ran too slowly to capture enough data to make the plot accurate. It was calculated that using the high-level simulation it could take up to 70 hours to

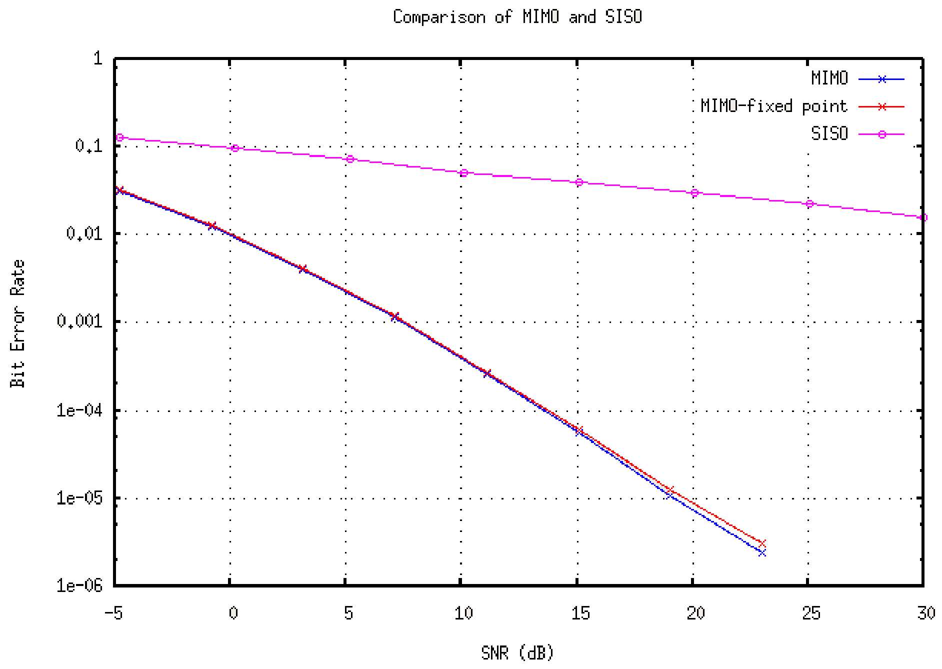

Figure 4.1: Simulation Results - MIMO and SISO Comparison

obtain reliable data for a single point on the plot. However using the low-level simulation made it possible to collect all the data used in Figure 4.1 within one day.

4.4 Other Work Completed

Before the low-level simulation was implemented the initial high-level simulation was ported to C++ using the extension interface the Octave provides [2]. The major reason for doing this work was to speed up the simulation, and it was also thought that this could provide the basis for the low-level, bit accurate, simulation.

As noted in Section 4.3 the initial Octave simulation ran fairly slowly. After porting this simulation to C++ it was able to simulate around 450 bits/s on a PC with an AMD Athlon 2000+ processor. However after writing the C++ version the initial Octave implementation was revised and optimised. After optimisation the Octave simulation was able to simulate 400 bits/s on the same PC.

In the meantime it was realised that it would be fairly difficult to make the C++ code form the basis of the bit accurate implementation. The main reason for this difficulty is because the interface to Octave requires that high-level C++ features, such as object-

[2]This is similar to the Mex interface that MATLAB provides to allow functions to be coded in C.

orientation, be used. These features do not map too well into hardware, so it was decided to cease developing the C++ code, and begin the low-level simulation again from scratch using C rather than C++.

Chapter 5

Hardware Design

After the simulations were completed and the expected results were verified the final step in the hardware design process was to actually implement the designs in a Hardware Description Language (HDL). For this project VHDL was chosen to implement the designs as I have previous experience using it.

Implementing the designs in hardware poses some unique challenges. Considerations such as how many clock cycles a given operation takes, or whether an operation can be completed in parallel with another, rarely matter at earlier stages in the process. However details like these are critically important when implementing hardware.

At the time of writing all the major components in the MIMO system have been implemented as VHDL except for the channel estimator. The implemented components have all been tested to verify correct operation. All components produce exactly the same output as the bit accurate low-level simulation, so performance will be identical. However, the individual components have not yet been joined together to form a complete system. See Section 6.1 for more details.

This chapter will examine the individual components that have been implemented in VHDL. The VHDL source code can be found in Appendix C

5.1 BPSK Modulator

The BPSK modulator is fairly straightforward, as it operates on a single bit at a time there is no state machine for control, it is simply combinational logic. The modulator takes a single bit as input and outputs two 8 bit numbers representing the real, and imaginary[1] parts of the modulated symbol.

The BPSK constellation in use in this project is purely real, i.e. a 1 is represented by the symbol $1 + 0i$ and a 0 is represented by $-1 + 0i$. This constellation hard coded into the modulator rather than a "header file" like the low-level simulation. However the modulator is less than 20 lines of code so it is fairly easy to change if needs be. Because the constellation is purely real the modulator has the quadrature (imaginary) part of it's output constantly assigned to 0.

The source code for the BPSK modulator can be found in Appendix C.1

5.2 Alamouti Encoder

The Alamouti encoder is more complicated than the BPSK modulator, it contains sequential logic and thus requires some control logic, and a clock signal. The encoder has four 8 bit inputs, the real and imaginary parts of the 2 symbols being encoded. The inputs are not registered, and are assumed to be held constant for the duration of the encoding process (2 clock cycles). There are also four 8 bit outputs for the real and imaginary parts of the encoded symbols. These outputs are designed to be fed into the radio modules on the testbed, which have 8 bit digital to analogue converters. It is designed to operate at the same clock speed as the data rate of the system, so one clock cycle is assumed to be one symbol period.

Since it takes 2 clock cycles to encode 2 symbols the modulator must maintain a state to indicate if it is currently the first or second time period. This state is implemented as a single bit signal that is toggled each clock cycle.

The source code for the encoder can be found in Appendix C.3

5.3 Alamouti Decoder

Like in the low-level simulation the hardware implementation of the Alamouti decoder is based on Equations 4.2 – 4.5. However, unlike the low-level simulation they are not simply converted into the programming language in use.

This straight conversion was tested initially, however it was quickly abandoned. The equation to estimate $s0_{re}$ was converted into VHDL and synthesised. When converted in this manner the single equation used over half the resources available on the FPGA chip

[1]At this level the real and imaginary parts of a symbol are also known as in-phase and quadrature components.

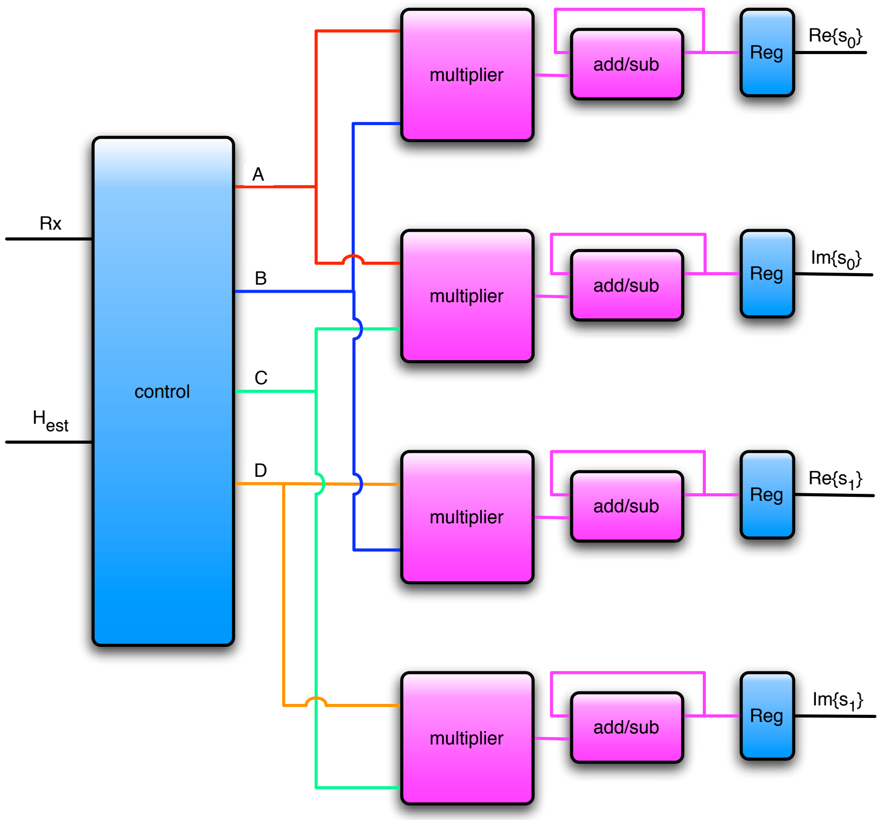

Figure 5.1: Block Diagram of Hardware Implementation of Alamouti Decoder

on the testbed. Obviously this in unacceptable as not only are there 3 other equations, but there are also other components that need to fit on the FPGA as well.

Instead a new design was created, Figure 5.1 shows a block diagram of this revised design. The design consists of four multiplier functional units, and four associated add/subtract units with registers to accumulate the totals. There is also control logic, implemented as a state machine, to multiplex inputs through to the various functional units, and also control whether the add/subtract units add or subtract (these control lines are not shown in the diagram). The meaning of the A, B, C, and D signals is not immediately obvious, however it is explained below how these signals are related to the input signals.

From Figure 5.1 it can be seen that the design calculates all the equations for the symbol estimates in parallel. There is one multiplier and one add/subtract unit for each equation being implemented. The design is a multi-cycle implementation, it takes

Pair	First Usage	Second Usage
A	$s0_{re}$ operand 1	$s0_{im}$ operand 1
B	$s0_{re}$ operand 2	$s1_{re}$ operand 2
C	$s0_{im}$ operand 2	$s1_{im}$ operand 2
D	$s1_{re}$ operand 1	$s1_{im}$ operand 1

Table 5.1: Pairs of Operands Output by the Control Logic in Alamouti Decoder.

multiple clock cycles to compute the results. The multipliers take one clock cycle to calculate a product and the add/subtract units also take one clock cycle. Therefore two symbol estimates (real and imaginary parts) are produced every 8 clock cycles. When synthesised for the testbed the decoder can run at a maximum clock frequency of 62.135 MHz.

The meaning of the A, B, C and D signal can be found by careful examination of Equations 4.2 – 4.5. In particular, note that there are four distinct sets of operands for the multiplication operations. These four sets, which have been labelled A, B, C and D, are shown in Table 5.1.

To further explain the meaning of Table 5.1 take pair A as an example. The first usage of A is listed as "$s0_{re}$ operand 1" and the second is "$s0_{im}$ operand 1". Now examine Equations 4.2 and 4.3, the equations for $s0_{re}$ and $s0_{im}$, reproduced in part below as Equations 5.1 and 5.2.

$$\begin{aligned} s0_{re} &= Re\{h_{0,0}\} \times Re\{y_{0,0}\} + Im\{h_{0,0}\} \times Im\{y_{0,0}\} \\ &\quad + Re\{h_{0,1}\} \times Re\{y_{0,1}\} + Im\{h_{0,1}\} \times Im\{y_{0,1}\} \ldots \qquad (5.1) \\ s0_{im} &= Re\{h_{0,0}\} \times Im\{y_{0,0}\} - Im\{h_{0,0}\} \times Re\{y_{0,0}\} \\ &\quad - Re\{h_{0,1}\} \times Im\{y_{0,1}\} + Im\{h_{0,1}\} \times Re\{y_{0,1}\} \ldots \qquad (5.2) \end{aligned}$$

Note, in particular, that the first (left hand) operand of any multiplication in Equation 5.1 is the same as the first operand of the corresponding multiplication in Equation 5.2. Because these operands are always the same they are grouped together as pair A. Table 5.1 similarly specifies the members of the other pairs. These grouping can be verified by checking them against Equations 4.2 – 4.5.

By exploiting these pairings the control logic is able to multiplex the required inputs through to all of the multiplier functional units using only four multiplexers instead of the eight that would otherwise be required.

The source code for the Alamouti decoder, along with the various functional units inside it, can be found in Appendix C.4.

5.4 Channel Estimator

As stated in Section 4.2.2 it is possible to implement the channel estimator as a single complex matrix multiplication. After some investigation it was found that this task would be more difficult than actually implementing the Alamouti decoder. So a ready made multiplier was sought out.

In 2002 a student at the University of Newcastle, Geoff Knagge, completed a final year project that implemented an efficient complex matrix multiplier. His design was written using VHDL and should be suitable for use on the testbed. Geoff has been contacted to see if it is possible to use his work in this project, however at the time of writing this had not been finalised.

If it is possible to use his design then the channel estimator will be implemented as the matrix multiplier, with one input coming from a Read Only Memory (ROM), and the other input being the received training block.

Chapter 6

Conclusions and Further Work

6.1 Further Work

At the time of writing this report I have not yet completed all the goals I set out to achieve with this project. The design has not been implemented on the testbed, however, most of the work is completed. There is only one major component not yet implemented: the channel estimator. The channel estimator is simply a complex matrix multiplier. This is the kind of component that has likely already been designed by someone and made available in a VHDL library. In fact the possibility of using a previous project student's design is being examined. Once a multiplier design is found it will be simple to incorporate it into the overall design.

Also, the various components that have been implemented in VHDL are not connected together as a system. Therefore the obvious work left to be done is to join these components together for demonstration on open-day. However there may be problems with this approach.

Currently the testbed hardware that is in Newcastle[1] is not fully functional. Nathan Tomkins has been working to get his driver talking to a hardware design that John Dalton implemented on the card. At this stage I do not believe this work has been entirely successful, however, my information is not up to date, so this may be no longer be the case.

A different approach may be to change the design on the FPGA and use the testbed as a simple radio modulator, then use the low-level simulation written in this project to perform the signal processing tasks. This method should be simpler to get working "across the air", however it will not allow for real-time operation.

[1]There is other hardware in Sydney that John Dalton has been using for testing

In addition to simply getting the system to function there is much room for optimisation in the hardware designs created for this project. The Alamouti decoder design is a “multi-cycle” design, after talking to more experienced hardware engineers it became obvious that a “pipelined” design would be more efficient. Also all the multipliers were implemented by simply using the `*` operator in VHDL, it should be possible to use a more advanced multiplier instead.

Once the system is working correctly there are many other possible extensions of this project that could be carried out. One interesting area is *channel sounding.* This is the process of transmitting data that is known to the receiver to try to obtain accurate estimates of the channel matrix. These estimates can then be analysed to see how the channel behaves. Different transmission environments can be examined, and compared. Other extensions include:

- Using a higher order constellation such as 16QAM
- Investigating the use of different modulation techniques, such as OFDM, when used in conjunction with space-time coding.
- Evaluating the “real world” performance of MIMO systems under various circumstances to investigate how much diversity is available in different transmission environments.

6.2 Conclusion

Chapter 2 showed the theoretical basis for the Alamouti code and other MIMO systems. These theories were confirmed by the simulations carried out and documented in Chapter 4. Even though the model used to simulate wireless transmission was “worse” than reality the MIMO system still greatly outperformed the SISO system. Finally, Chapter 5 described the hardware designs created to actually implement the Alamouti code.

While these designs have not yet been implemented on the testbed, I am confident that we will be able to build a working MIMO system once the VHDL code is finalised, and the testbed is fully debugged. The system is not so complicated as to prohibit its practical implementation. The most complicated component is the complex matrix multiplier used in the channel estimator, and that problem has already been tackled by others.

Therefore, this project has proved that it is quite feasible to implement an Alamouti code using commercially available FPGAs. This puts the possibility of further testing and research into MIMO systems within reach.

Appendix A

High Level Simulation Source Code

This appendix contains the code that was used in the high level simulation. The code is written for Octave, which is generally compatible with MATLAB, however I have not tested this code in MATLAB.

A.1 Alamouti Encoder Code

```
function a_mod_out = alamouti_mod(in, block_sz, x_t)
    symb_one = 1;
    symb_zero = -1;

    %variable to keep track of the "current position" in the output
    out_i = 1;

    %sanity checks...always needed when i'm around ;-)
    if !( mod(length(in),block_sz) == 0)
        printf('Whoops:  the input stream length needs to be a multiple of the block size\n');
        return;
     end

     if !( mod(block_sz,2) == 0)
        printf('Whoops:  The block size needs to be a multiple of 2\n');
        return
     end

     % loop through the input a block at a time...
     for in_i=1:block_sz:columns(in)
        % put the training block at the start
        for tr_i=1:length(x_t)
            a_mod_out(:,[out_i++])=x_t(:,tr_i);
        end

        % now loop through the rest of the block and encode the symbols
        for block_i=in_i:2:(in_i + block_sz - 1)
            bits = in([block_i++ block_i]);

            % BPSK modulator
            if bits(1) == 0
                s1 = symb_zero;
            else
                s1 = symb_one;
            end

            if bits(2) == 0
                s2 = symb_zero;
            else
                s2 = symb_one;
            end

            % Alamotui encoder
            a_mod_out(:,[out_i++ out_i++]) = [s1 -conj(s2); s2 conj(s1)];
        end % end symbols for loop
    end % end block for loop
end % end function
```

A.2 Channel Estimator Code

```
function h_est = chan_est(y_t, x_t)

% stright implementation of channel estimation equation
h_est = y_t * x_t' * inv(x_t * x_t') ;

end %end function
```

A.3 Alamouti Decoder Code

```
function [out, est_h_list] = alamouti_demod(in,nb,x_t)
    symb_zero = -1;
    symb_one = 1;

    % position marker in the input stream
    in_i = 1;

    % position marker in the output stream
    out_i = 1;

    % Iterate through the input one block (including training data) at a time
    for block_i=1:(nb + length(x_t)):columns(in)
        % grab the training block
        y_t = in(:,[in_i++ in_i++ in_i++ in_i++]);

        % now use the training block to make a channel estimate
        %H_est = chan_est(y_t,x_t) % normal octave code version
        H_est = chan_est_f(y_t, x_t); % c++ octave extension version

        % now iterate through the "data" block decoding symbols
        for sym_i=1:2:nb
            y = in(:,[in_i++ in_i++]);

            % soft decision decoder
            s0_squig = conj(H_est(1,1))*y(1,1) + H_est(1,2)*conj(y(1,2)) + conj(H_est(2,1))*y(2,1) + H_est(2,2)*conj(y(2,2));
            s1_squig = conj(H_est(1,2))*y(1,1) - H_est(1,1)*conj(y(1,2)) + conj(H_est(2,2))*y(2,1) - H_est(2,1)*conj(y(2,2));

            % hard decision decoder
            if real(s0_squig) < 0
                out(out_i++) = 0;
            else
                out(out_i++) = 1;
            end % end s0 detector

            if real(s1_squig) < 0
                out(out_i++) = 0;
            else
                out(out_i++) = 1;
            end % end s0 detector
        end % end decoding for loop
    end % end "block" for loop
end % end function
```

Appendix B

Low-Level Simulation Source Code

Some of the source code used for the low-level simulation is presented in this appendix. For the sake of the trees producing the paper this report is printed on, not all of the header files and associated supporting code is included.

B.1 BPSK Modulator

```c
#include "bpsk_const.h"

void bpsk_mod(unsigned char input, unsigned char output_re[8], unsigned char output_im[8])
{
    unsigned short int i;

    for(i=0; i<8; i++)
    {
        /* test a single bit at a time */
        if( (input>>i) & 0x01 )
        {
            output_re[i] = SYMBOL_ONE_RE;
            output_im[i] = SYMBOL_ONE_IM;
        }
        else
        {
            output_re[i] = SYMBOL_ZERO_RE;
            output_im[i] = SYMBOL_ZERO_IM;
        }
    }
    return;
}
```

B.2 Alamouti Encoder

```c
#include <stdio.h>
#include "bpsk_mod.h"

void alamouti_enc(unsigned char input, char out_re[2][8], char out_im[2][8])
{
    int i=0;
    unsigned char bpsk_mod_re[8], bpsk_mod_im[8];

    /* modulate the block using BPSK */
    bpsk_mod(input, bpsk_mod_re, bpsk_mod_im);

    for(i=0; i<8; i+=2)
    {
        int i_1 = i + 1;

        /* time t=T */
        out_re[0][i] = bpsk_mod_re[i];
        out_im[0][i] = bpsk_mod_im[i];

        out_re[1][i] = bpsk_mod_re[i_1];
        out_im[1][i] = bpsk_mod_im[i_1];

        /* time t=T+1 */
        out_re[0][i_1] = -bpsk_mod_re[i_1];
        out_im[0][i_1] = bpsk_mod_im[i_1];

        out_re[1][i_1] = bpsk_mod_re[i];
        out_im[1][i_1] = -bpsk_mod_im[i];
    }
}
```

B.3 Channel Estimator

```c
#include "chan_est_const.h"
#include "chan_est.h"
#include "matrix.h"
#include "matrix_fix.h"
#include "fixed.h"

void chan_est(float rec_re[TR_ROWS][TR_COLS], float rec_im[TR_ROWS][TR_COLS], float *est_re, float
*est_im)
{
    /* I'm passing in the received training block, the transmitted training
     * block, and the constant term are defined in "chan_est.h"
     */

    comp_matrix_mult_f((float *)rec_re, (float *)rec_im, TR_ROWS, TR_COLS, (float *)const_term_re,
(float *)const_term_im, CONST_ROWS, CONST_COLS, est_re, est_im);
}

void chan_est_fix16(fix16_t rec_re[TR_ROWS][TR_COLS], fix16_t rec_im[TR_ROWS][TR_COLS], fix16_t *est_re,
fix16_t *est_im)
{
    fix16_t const_term_re_16[CONST_ROWS][CONST_COLS];
    fix16_t const_term_im_16[CONST_ROWS][CONST_COLS];
    int row,col;

    /* make a fixed point version of the constant term */
    for(row = 0; row < CONST_ROWS; row++)
    {
        for(col=0; col < CONST_COLS; col++)
        {
            const_term_re_16[row][col] = quantise_16bit_l((const_term_re[row][col]));
            const_term_im_16[row][col] = quantise_16bit_l((const_term_im[row][col]));
        }
    }

    comp_matrix_mult_fix16((fix16_t *)rec_re, (fix16_t *)rec_im, TR_ROWS, TR_COLS, (fix16_t *)const_term_re_16,
(fix16_t *)const_term_im_16, CONST_ROWS, CONST_COLS, est_re, est_im);
}
```

B.4 Alamouti Decoder

```
#include "combiner.h"

#define COMB_DEBUG 0

void combine( float recv_re[2][2], float recv_im[2][2],
        float h_re[2][2], float h_im[2][2],
        float symb_re[2], float symb_im[2] )
{
    symb_re[0] = h_re[0][0]*recv_re[0][0] + h_im[0][0]*recv_im[0][0] +
            h_re[0][1]*recv_re[0][1] + h_im[0][1]*recv_im[0][1] +
            h_re[1][0]*recv_re[1][0] + h_im[1][0]*recv_im[1][0] +
            h_re[1][1]*recv_re[1][1] + h_im[1][1]*recv_im[1][1] ;

    symb_im[0] = h_re[0][0]*recv_im[0][0] - h_im[0][0]*recv_re[0][0] -
            h_re[0][1]*recv_im[0][1] + h_im[0][1]*recv_re[0][1] +
            h_re[1][0]*recv_im[1][0] - h_im[1][0]*recv_re[1][0] -
            h_re[1][1]*recv_im[1][1] + h_im[1][1]*recv_re[1][1] ;


```

```c
    symb_re[1] = h_re[0][1]*recv_re[0][0] + h_im[0][1]*recv_im[0][0] -
            h_re[0][0]*recv_re[0][1] - h_im[0][0]*recv_im[0][1] +
            h_re[1][1]*recv_re[1][0] + h_im[1][1]*recv_im[1][0] -
            h_re[1][0]*recv_re[1][1] - h_im[1][0]*recv_im[1][1] ;

    symb_im[1] = h_re[0][1]*recv_im[0][0] - h_im[0][1]*recv_re[0][0] +
            h_re[0][0]*recv_im[0][1] - h_im[0][0]*recv_re[0][1] +
            h_re[1][1]*recv_im[1][0] - h_im[1][1]*recv_re[1][0] +
            h_re[1][0]*recv_im[1][1] - h_im[1][0]*recv_re[1][1] ;

    return;
}


void combine_fix16(fix16_t recv_re[2][2], fix16_t recv_im[2][2],
            fix16_t h_re[2][2], fix16_t h_im[2][2],
            fix16_t symb_re[2], fix16_t symb_im[2] )
{
    long int temp_re_0, temp_re_1, temp_im_0, temp_im_1;


    temp_re_0 = fix16_mult(h_re[0][0],recv_re[0][0]) + fix16_mult(h_im[0][0],recv_im[0][0]) +
            fix16_mult(h_re[0][1],recv_re[0][1]) + fix16_mult(h_im[0][1],recv_im[0][1]) +
            fix16_mult(h_re[1][0],recv_re[1][0]) + fix16_mult(h_im[1][0],recv_im[1][0]) +
            fix16_mult(h_re[1][1],recv_re[1][1]) + fix16_mult(h_im[1][1],recv_im[1][1]) ;

    temp_im_0 = fix16_mult(h_re[0][0],recv_im[0][0]) - fix16_mult(h_im[0][0],recv_re[0][0])-
            fix16_mult(h_re[0][1],recv_im[0][1]) + fix16_mult(h_im[0][1],recv_re[0][1]) +
            fix16_mult(h_re[1][0],recv_im[1][0]) - fix16_mult(h_im[1][0],recv_re[1][0]) -
            fix16_mult(h_re[1][1],recv_im[1][1]) + fix16_mult(h_im[1][1],recv_re[1][1]) ;

    temp_re_1 = fix16_mult(h_re[0][1],recv_re[0][0]) + fix16_mult(h_im[0][1],recv_im[0][0]) -
            fix16_mult(h_re[0][0],recv_re[0][1]) - fix16_mult(h_im[0][0],recv_im[0][1]) +
            fix16_mult(h_re[1][1],recv_re[1][0]) + fix16_mult(h_im[1][1],recv_im[1][0]) -
            fix16_mult(h_re[1][0],recv_re[1][1]) - fix16_mult(h_im[1][0],recv_im[1][1]) ;

    temp_im_1 = fix16_mult(h_re[0][1],recv_im[0][0]) - fix16_mult(h_im[0][1],recv_re[0][0]) +
            fix16_mult(h_re[0][0],recv_im[0][1]) - fix16_mult(h_im[0][0],recv_re[0][1]) +
            fix16_mult(h_re[1][1],recv_im[1][0]) - fix16_mult(h_im[1][1],recv_re[1][0]) +
            fix16_mult(h_re[1][0],recv_im[1][1]) - fix16_mult(h_im[1][0],recv_re[1][1]) ;

    symb_re[0] = (fix16_t) temp_re_0;
    symb_re[1] = (fix16_t) temp_re_1;
    symb_im[0] = (fix16_t) temp_im_0;
    symb_im[1] = (fix16_t) temp_im_1;

    return;
}
```

B.5 Fixed Point Functions

```
#include "fixed.h"

inline long int quantise_8bit_l(float input)
{
    if (input >= (8))
        return (long int) 127;
    else if (input < (-8))
        return (long int) -128;
    else
        return (long int) (input * (1 << RADIX_8));
}

inline long int quantise_10bit_l(float input)
{
    if (input >= (16))
        return (long int) 255;
    else if (input < (-16))
        return (long int) -256;
    return ((long int) (input * (1 << RADIX_10)));
}

inline long int quantise_12bit_l(float input)
{
    return ((long int) (input * (1 << RADIX_12)));
}
inline long int quantise_16bit_l(float input)
{
    return ((long int) (input * (1 << RADIX_16)));
}

inline long int quantise_24bit_l(float input)
{
    return ((long int) (input * (1 << RADIX_24)));
}

inline float fix8_to_float(fix8_t in)
{
    return ( in / (float) (1<<RADIX_8));
}

inline float fix10_to_float(fix10_t in)
{
    return ( in  / (float) (1<<RADIX_10));
}

inline float fix12_to_float(fix12_t in)
{
    return ( in  / (float) (1<<RADIX_12));
}
```

```c
inline float fix16_to_float(fix16_t in)
{
    return ( in  / (float) (1<<RADIX_16));
}

inline fix10_t fix8_to_fix10(fix8_t in)
{
    return (fix10_t) ( in<<(RADIX_10 - RADIX_8) );
}

inline fix12_t fix8_to_fix12(fix8_t in)
{
    return (fix12_t) ( in<<(RADIX_12 - RADIX_8) );
}

inline fix16_t fix8_to_fix16(fix8_t in)
{
    return (fix16_t) ( in<<(RADIX_16 - RADIX_8) );
}

inline fix12_t fix10_to_fix12(fix10_t in)
{
    return (fix12_t) ( in<<(RADIX_12 - RADIX_10) );
}

inline fix16_t fix10_to_fix16(fix10_t in)
{
    return (fix16_t) ( in<<(RADIX_16 - RADIX_10) );
}

inline fix16_t fix12_to_fix16(fix12_t in)
{
    return (fix16_t) ( in<<(RADIX_16 - RADIX_12) );
}

inline fix8_t fix8_mult(fix8_t a, fix8_t b)
{
    long int ans = 0;

    ans = ((long int) a * (long int)b)>>RADIX_8;
#if FIXED_DEBUG > 1
    printf("fix8_mult: Real - %f * %f = %f (%f => 0x%lx)\n",fix8_to_float(a),
            fix8_to_float(b), fix8_to_float(ans) , fix8_to_float(a) * fix8_to_float(b),
            quantise_8bit_l(fix8_to_float(a) * fix8_to_float(b)));

    printf("fix8_mult: Int - %d * %d = %ld\n",a,b,ans);
    printf("fix8_mult: Hex - 0x%x * 0x%x = 0x%lx\n",a,b,ans);
#endif

#if FIXED_DEBUG > 0
    if (ans > 0x7f)
    {
        printf("Overflow trying to do fix8_mult: 0x%x * 0x%x ?= 0x%lx\n",a,b,ans);
        exit(-1);
    }
#endif
    return (fix8_t) ans;
}


inline fix16_t fix16_mult(fix16_t a, fix16_t b)
{
    long int ans = 0;

    ans = ((long int) a * (long int)b)>>RADIX_16;
#if FIXED_DEBUG > 0
    printf("[0x%x * 0x%x = 0x%x]",(unsigned short)a,(unsigned short)b,(unsigned short)ans);
#endif
    return (fix16_t) ans;
}

inline fix10_t fix10_mult(fix10_t a, fix10_t b)
{
    long int ans = 0;

    ans = ((long int) a * (long int)b)>>RADIX_10;
    return (fix10_t) ans;
}

inline fix12_t fix12_mult(fix12_t a, fix12_t b)
{
    long int ans = 0;

    ans = ((long int) a * (long int)b)>>RADIX_12;
    return (fix12_t) ans;
}
```

Appendix C

Hardware Design Source Code

This Appendix contains the current[1] source code used for the hardware designs. The code is written in VHDL and has bee tested and synthesised using the no cost "Web-Pack" tools available from Xilinx.

C.1 BPSK Modulator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity bpsk_mod is port (
    input     : in    std_logic;
    i_out, q_out    : out    signed(7 downto 0)
);
end bpsk_mod;

architecture a of bpsk_mod is
begin
    with input select i_out <=
        "01111111" when '1',
        "10000000" when others;
    q_out <= "00000000";
end a;

```

[1] current at time of report writing

C.2 BPSK Demodulator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity bpsk_demod is port (
    i_in,q_in    : in    signed(7 downto 0);
    output     : out    std_logic
);
end bpsk_demod;

architecture a of bpsk_demod is
begin
    output <= not i_in(7);
end a;

```

C.3 Alamouti Enoder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


entity alamouti_mod is
    port
    (
        clk          : in    std_logic;
        i1_in,i2_in    : in    signed(7 downto 0);
        q1_in,q2_in    : in    signed(7 downto 0);
        i1_out,q1_out    : in    signed(7 downto 0);
        i2_out,q2_out    : in    signed(7 downto 0)
    );
end alamouti_mod;

architecture a of alamouti_mod is
    signal tmp1_i, tmp1_q : signed(7 downto 0); -- antenna1 variables
    signal tmp2_i, tmp2_q : signed(7 downto 0); -- antenna2 variables
    signal state : std_logic;
begin
    process(clk)
    begin
        if (clk'event and clk = '1') then
            if (state = '0') then-- first cycle
                tmp1_i <= i1_in;
                tmp1_q <= q1_in;

                tmp2_i <= i2_in;
                tmp2_q <= q2_in;
                state <= '1';
            else
                tmp1_i <= -i2_in;
                tmp1_q <= q2_in;

                tmp2_i <= i1_in;
                tmp2_q <= -q1_in;
                state <= '0';
            end if;
        end if;
    end process;


    i1_out <= tmp1_i;
    q1_out <= tmp1_q;

    i2_out <= tmp2_i;
    q2_out <= tmp2_q;
end a;

```

C.4 Alamouti Decoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;


library work;
use work.my_types.all;

entity combiner is port (
    clock : in std_logic;
    reset : in std_logic;

    rx_re_in : in t_2x2_matrix_16;
    rx_im_in : in t_2x2_matrix_16;
    h_re_in : in t_2x2_matrix_16;
    h_im_in : in t_2x2_matrix_16;

    s0re_est : out std_logic_vector(15 downto 0);
    s0im_est : out std_logic_vector(15 downto 0);
    s1re_est : out std_logic_vector(15 downto 0);
    s1im_est : out std_logic_vector(15 downto 0);

--    debug : out std_logic_vector(15 downto 0);
```

```

    done : out std_logic
);
end combiner;

architecture Behavioral of combiner is
    signal s0re_op : std_logic;
    signal s0im_op : std_logic;
    signal s1re_op : std_logic;
    signal s1im_op : std_logic;

    signal clear_control : std_logic;
    signal clear_units : std_logic;
    signal add_clear : std_logic;

    signal op_a : std_logic_vector(15 downto 0);
    signal op_b : std_logic_vector(15 downto 0);
    signal op_c : std_logic_vector(15 downto 0);
    signal op_d : std_logic_vector(15 downto 0);

    signal s0re_prod : std_logic_vector(15 downto 0);
    signal s0im_prod : std_logic_vector(15 downto 0);
    signal s1re_prod : std_logic_vector(15 downto 0);
    signal s1im_prod : std_logic_vector(15 downto 0);

    signal s0re_sum : std_logic_vector(15 downto 0);
    signal s0im_sum : std_logic_vector(15 downto 0);
    signal s1re_sum : std_logic_vector(15 downto 0);
    signal s1im_sum : std_logic_vector(15 downto 0);

    signal s0re_total : std_logic_vector(15 downto 0);
    signal s0im_total : std_logic_vector(15 downto 0);
    signal s1re_total : std_logic_vector(15 downto 0);
    signal s1im_total : std_logic_vector(15 downto 0);

    signal s0re_op_regd : std_logic;
    signal s0im_op_regd : std_logic;
    signal s1re_op_regd : std_logic;
    signal s1im_op_regd : std_logic;

    signal add_count : integer range 0 to 7;
    signal done_i : std_logic;
    -------------------------------------------------
    --              component declarations        --
    -------------------------------------------------
    component comb_control
    port(
        clock : in std_logic;
        reset : in std_logic;

        rx_re_in : in t_2x2_matrix_16;
        rx_im_in : in t_2x2_matrix_16;
        h_re_in : in t_2x2_matrix_16;
        h_im_in : in t_2x2_matrix_16;

        operand_a : out std_logic_vector(15 downto 0);
        operand_b : out std_logic_vector(15 downto 0);
        operand_c : out std_logic_vector(15 downto 0);
        operand_d : out std_logic_vector(15 downto 0);

        s0re_add, s0im_add: out std_logic;
        s1re_add, s1im_add: out std_logic;
        done : out std_logic;
        clear : out std_logic
        );
    end component;

    component add_sub_16 is port (
        a : in std_logic_vector(15 downto 0);
        b : in std_logic_vector(15 downto 0);
        add : in std_logic;
        ans : out std_logic_vector(15 downto 0)
    );end component;
---------------- end component declarations ------------------
begin

clear_units <= reset or clear_control;

    combiner_control_unit: comb_control port map(
        clock => clock,
        reset => reset,
        rx_re_in => rx_re_in,
        rx_im_in => rx_im_in,
        h_re_in => h_re_in,
        h_im_in => h_im_in,

        operand_a => op_a,
        operand_b => op_b,
        operand_c => op_c,
        operand_d => op_d,

        s0re_add => s0re_op,
        s0im_add => s0im_op,
        s1re_add => s1re_op,
        s1im_add => s1im_op,
        done => done_i,
        clear => clear_control
    );
    done <= done_i;

-- need to register the add/subtract signals because
-- the product gets registered, need to keep them in
-- sync!
process (clock)
begin
    if (clock'event and clock='1') then
        s0re_op_regd <= s0re_op;
        s0im_op_regd <= s0im_op;
        s1re_op_regd <= s1re_op;
        s1im_op_regd <= s1im_op;
    end if;
end process;

    -------------------------------------------
    -- Multipliers and registers
    -------------------------------------------
    s0re_mult: process (clock)
        variable result : signed(31 downto 0);
    begin
        if(clock'event and clock = '1') then
            result := conv_signed(conv_integer(op_a) * conv_integer(op_b), 32);
            s0re_prod <= conv_std_logic_vector(result(23 downto 8),16);
        end if;
    end process;

    s0im_mult: process (clock)
        variable result : signed(31 downto 0);
    begin
        if(clock'event and clock = '1') then
            result := conv_signed( conv_integer(op_a) * conv_integer(op_c) ,32);
            s0im_prod <= conv_std_logic_vector(result(23 downto 8),16);
        end if;
    end process;

    s1re_mult: process (clock)
        variable result : signed(31 downto 0);
    begin
        if(clock'event and clock = '1') then
            result := conv_signed( conv_integer(op_d) * conv_integer(op_b), 32);
            s1re_prod <= conv_std_logic_vector(result(23 downto 8),16);
        end if;
    end process;

    s1im_mult: process (clock)
        variable result : signed(31 downto 0);
    begin
        if(clock'event and clock = '1') then
            result := conv_signed( conv_integer(op_d) * conv_integer(op_c), 32);
            s1im_prod <= conv_std_logic_vector(result(23 downto 8),16);
        end if;
    end process;

    -----------------------------------------
```

```
-- Adders / Subtracters
------------------------------------------
s0re_add: add_sub_16 port map (
    a => s0re_total,
    b => s0re_prod,
    add => s0re_op_regd,
    ans => s0re_sum
    );

s0im_add: add_sub_16 port map (
    a => s0im_total,
    b => s0im_prod,
    add => s0im_op_regd,
    ans => s0im_sum
    );
s1re_add: add_sub_16 port map (
    a => s1re_total,
    b => s1re_prod,
    add => s1re_op_regd,
    ans => s1re_sum
    );

s1im_add: add_sub_16 port map (
    a => s1im_total,
    b => s1im_prod,
    add => s1im_op_regd,
    ans => s1im_sum
    );
------------------------------------------
-- Registers
------------------------------------------
s0re_reg: process (clock)
begin
    if(clock'event and clock = '1' ) then
        if (clear_units ='1') then
            s0re_total <= x"0000";
        else
            s0re_total <= s0re_sum;
        end if;
    end if;
end process;

s0im_reg: process (clock)
begin
    if(clock'event and clock = '1' ) then
        if (clear_units ='1') then
            s0im_total <= x"0000";
        else
            s0im_total <= s0im_sum;
        end if;
    end if;
```

```vhdl
    end process;

    s1re_reg: process (clock)
    begin
        if(clock'event and clock = '1' ) then
            if (clear_units ='1') then
                s1re_total <= x"0000";
            else
                s1re_total <= s1re_sum;
            end if;
        end if;
    end process;

    s1im_reg: process (clock)
    begin
        if(clock'event and clock = '1' ) then
            if (clear_units ='1') then
                s1im_total <= x"0000";
            else
                s1im_total <= s1im_sum;
            end if;
        end if;
    end process;


    ----------------------------------------------
    -- output assignements
    ----------------------------------------------

     s0re_est <= s0re_sum;
    s0im_est <= s0im_sum;
    s1re_est <= s1re_sum;
    s1im_est <= s1im_sum;

--debug <= x"000" & s0re_op_regd & s0im_op_regd & s1re_op_regd & s1im_op_regd;
--debug <= x"000" & '0' &'0' &'0' & clear_control;
end Behavioral;
```

C.4.1 Control Unit

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.my_types.all;

entity comb_control is Port (
    clock : in std_logic;
    reset : in std_logic;

    rx_re_in : in t_2x2_matrix_16;
    rx_im_in : in t_2x2_matrix_16;
    h_re_in : in t_2x2_matrix_16;
    h_im_in : in t_2x2_matrix_16;


    -- Key:
    -- A -> s0_re op1, s0_im op1
    -- B -> s0_re op2, s1_re op2
    -- C -> s0_im op2, s1_im op2
    -- D -> s1_re op1, s1_im op1
    operand_a : out std_logic_vector(15 downto 0);
    operand_b : out std_logic_vector(15 downto 0);
    operand_c : out std_logic_vector(15 downto 0);
    operand_d : out std_logic_vector(15 downto 0);

    s0re_add, s0im_add: out std_logic;
    s1re_add, s1im_add: out std_logic;
    done : out std_logic;
    clear : out std_logic
);
end comb_control;

architecture Behavioral of comb_control is
    type state_type is (st_rst,st1, st2, st3, st4, st5, st6, st7, st8);
    -- state machine internal signals
    signal state, next_state : state_type;
    signal op_a_i : std_logic_vector(15 downto 0);
    signal op_b_i : std_logic_vector(15 downto 0);
    signal op_c_i : std_logic_vector(15 downto 0);
    signal op_d_i : std_logic_vector(15 downto 0);

    signal s0re_add_i, s0im_add_i : std_logic;
    signal s1re_add_i, s1im_add_i : std_logic;

    signal rx_re_reg, rx_im_reg : t_2x2_matrix_16;
    signal h_re_reg, h_im_reg : t_2x2_matrix_16;


    -- temp output logic signals
    signal done_i : std_logic;
    signal clear_i : std_logic;
begin
    input_regs : process (clock,reset)
    begin
        if(clock'event and clock='1') then
            if (reset = '1') then
                rx_re_reg <= (others => x"0000");
```

```
             rx_im_reg <= (others => x"0000");
             h_re_reg <= (others => x"0000");
             h_im_reg <= (others => x"0000");
         elsif (state = st8) or (state = st_rst) then
             rx_re_reg <= rx_re_in;
             rx_im_reg <= rx_im_in;
             h_re_reg <= h_re_in;
             h_im_reg <= h_im_in;
         end if;
     end if;
 end process;


SYNC_PROC: process (CLOCK, reset)
begin
   if (clock'event and clock = '1') then
       if ( reset = '1' ) then
          state <= st_rst;
             s0re_add <= '1';
             s0im_add <= '1';
             s1re_add <= '1';
             s1im_add <= '1';

             operand_a <= x"0000";
             operand_b <= x"0000";
             operand_c <= x"0000";
             operand_d <= x"0000";

             done <= '0';
             clear <= '1';
         else
             state <= next_state;

             s0re_add <= s0re_add_i;
             s0im_add <= s0im_add_i;
             s1re_add <= s1re_add_i;
             s1im_add <= s1im_add_i;

             done <= done_i;
             clear <= clear_i;

             operand_a <= op_a_i;
             operand_b <= op_b_i;
             operand_c <= op_c_i;
             operand_d <= op_d_i;
         end if;
     end if;
 end process;


--MOORE State Machine - Outputs based on state only
```

```
OUTPUT_DECODE: process (state,h_re_reg,h_im_reg,rx_re_reg,rx_im_reg)
begin

 -- Operands Key:
 -- A -> s0_re op1, s0_im op1
 -- B -> s0_re op2, s1_re op2
 -- C -> s0_im op2, s1_im op2
 -- D -> s1_re op1, s1_im op1

     case (state) is
         when st_rst =>
             op_a_i <= x"0000";
             op_b_i <= x"0000";
             op_c_i <= x"0000";
             op_d_i <= x"0000";

             s0re_add_i <= '1';
             s0im_add_i <= '1';
             s1re_add_i <= '1';
             s1im_add_i <= '1';

             done_i <= '0';
             clear_i <='1';

         when st1 =>
             op_a_i <= h_re_reg(0);
             op_b_i <= rx_re_reg(0);
             op_c_i <= rx_im_reg(0);
             op_d_i <= h_re_reg(1);

             s0re_add_i <= '1';
             s0im_add_i <= '1';
             s1re_add_i <= '1';
             s1im_add_i <= '1';

             done_i <= '1';
             clear_i <= '1';
         when st2 =>
             op_a_i <= h_im_reg(0);
             op_b_i <= rx_im_reg(0);
             op_c_i <= rx_re_reg(0);
             op_d_i <= h_im_reg(1);

             s0re_add_i <= '1';
             s0im_add_i <= '0';
             s1re_add_i <= '1';
             s1im_add_i <= '0';

             done_i <= '0';
             clear_i <= '0';
         when st3 =>
```

```
                op_a_i <= h_re_reg(1);
                op_b_i <= rx_re_reg(1);
                op_c_i <= rx_im_reg(1);
                op_d_i <= h_re_reg(0);

                s0re_add_i <= '1';
                s0im_add_i <= '0';
                s1re_add_i <= '0';
                s1im_add_i <= '1';

                done_i <= '0';
                clear_i <= '0';
            when st4 =>
                op_a_i <= h_im_reg(1);
                op_b_i <= rx_im_reg(1);
                op_c_i <= rx_re_reg(1);
                op_d_i <= h_im_reg(0);

                s0re_add_i <= '1';
                s0im_add_i <= '1';
                s1re_add_i <= '0';
                s1im_add_i <= '0';

                done_i <= '0';
                clear_i <= '0';
            when st5 =>
                op_a_i <= h_re_reg(2);
                op_b_i <= rx_re_reg(2);
                op_c_i <= rx_im_reg(2);
                op_d_i <= h_re_reg(3);

                s0re_add_i <= '1';
                s0im_add_i <= '1';
                s1re_add_i <= '1';
                s1im_add_i <= '1';

                done_i <= '0';
                clear_i <= '0';
            when st6 =>
                op_a_i <= h_im_reg(2);
                op_b_i <= rx_im_reg(2);
                op_c_i <= rx_re_reg(2);
                op_d_i <= h_im_reg(3);

                s0re_add_i <= '1';
                s0im_add_i <= '0';
                s1re_add_i <= '1';
                s1im_add_i <= '0';

                done_i <= '0';
                clear_i <= '0';
```

```vhdl
            when st7 =>
                op_a_i <= h_re_reg(3);
                op_b_i <= rx_re_reg(3);
                op_c_i <= rx_im_reg(3);
                op_d_i <= h_re_reg(2);

                s0re_add_i <= '1';
                s0im_add_i <= '0';
                s1re_add_i <= '0';
                s1im_add_i <= '1';

                done_i <= '0';
                clear_i <= '0';
            when st8 =>
                 op_a_i <= h_im_reg(3);
                op_b_i <= rx_im_reg(3);
                op_c_i <= rx_re_reg(3);
                op_d_i <= h_im_reg(2);

                s0re_add_i <= '1';
                s0im_add_i <= '1';
                s1re_add_i <= '0';
                s1im_add_i <= '0';
                clear_i <= '0';
                 done_i <= '0';
        end case;
   end process;

   NEXT_STATE_DECODE: process (state)
   begin
      --declare default state for next_state to avoid latches
      next_state <= state;  --default is to stay in current state
      case (state) is
            when st_rst => next_state <= st1;
            when st1 => next_state <= st2;
            when st2 => next_state <= st3;
            when st3 => next_state <= st4;
            when st4 => next_state <= st5;
            when st5 => next_state <= st6;
            when st6 => next_state <= st7;
            when st7 => next_state <= st8;
            when st8 => next_state <= st1;
         when others =>
            next_state <= st_rst;
      end case;
   end process;

end Behavioral;
```

C.4.2 Add/Subtract Unit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;


entity add_sub_16 is
    Port ( a : in std_logic_vector(15 downto 0);
           b : in std_logic_vector(15 downto 0);
           add : in std_logic;
              ans : out std_logic_vector(15 downto 0)
              );
end add_sub_16;

architecture Behavioral of add_sub_16 is
begin
    process(a,b,add)
        variable op_b : std_logic_vector(15 downto 0);
        variable carry_in : std_logic;
    begin
-- This way uses 33 of 3072 slices, and has a delay of 13.425ns
--         if(add = '1') then
--             ans <= a + b;
--         else
--             ans <= a - b;
--         end if;

-- This way uses 17/3072 slices, and 31/6144 4in-LUTs, delay of 13ns
        if (add = '1') then
            op_b := b;
            carry_in := '0';
        else
            op_b := not b;
            carry_in := '1';
        end if;
        ans <= a + op_b + carry_in;
    end process;
end Behavioral;
```

Bibliography

[1] David Gesbert, Mansoor Shafi, Da-shan Shiu, Peter J. Smith, and Ayman Naguib. From theory to practice: An overview of mimo space-time coded wireless systems. *IEEE Journal on Selected Areas in Communications*, 21(3):281–302, 2003.

[2] Erik G. Larsen and Petre Stoica. *Space-Time Block Coding for Wireless Communications*. Cambridge University Press, 2003.

[3] Hesham El Gamal and A. Roger Hammons, Jr. On the Design and Performance of Algebraic Space-Time Codes for BPSK and QPSK Modulation. *IEEE Transactions on Communications*, 50(8):907–913, June 2002.

[4] Markus Rupp, Andreas Burg, Eric Beck. Rapid Prototyping for Wireless Designs: the Five-Ones Approach. *Signal Processing*, 83:1427–1444, 2003.

[5] Raghu Mysore Rao et. al. Multi-Antenna Testbeds for Research and Education in Wireless Communications. *IEEE Communications Magazine*, pages 72–81, December 2004.

[6] Raleigh, G.G. and Cioffi, J.M. . Spatio-temporal coding for wireless communications. *IEEE Transactions on Communications*, 46(3):357–366, 1998.

[7] Siavash M. Alamouti. A Simple Transmit Diversity Technique for Wireless Communications. *IEEE Journal on Select Areas in Communcations*, 16(8):1451–1458, October 1998.

[8] Vahid Tarokh, Hamid Jafarkhani, and A. Robert Calderbank. Space-time block coding for wireless communications: Performance results. *IEEE Journal on Selected Areas in Communications*, 17(3):451–460, March 1999.