Implementation and Design of a Bit-Error Generator and Logger for Multi-Gigabit Serial Links

Master’s Thesis
in Computer Engineering

by
Pedro Botella

LiTH-ISY-EX--07/3928--SE

Linköping, February 26, 2007
Implementation and Design of a Bit-Error Generator and Logger for Multi-Gigabit Serial Links

Master Thesis in Computer Engineering, Department of Electrical Engineering at Linköping University

by Pedro Botella

LiTH-ISY-EX--07/3928--SE

Supervisor: Fredrik Elmgren, Sectra Mamea AB
Examiner: Olle Seger, Linköping University

Linköping, February 26, 2007
Test Tools are very important in the design of a system. They generally simulate a working environment, only at a higher speed, or with less frequently occurring test cases. In the verification of protocols based on the Fibre Channel physical layer, this becomes a necessity, as errors can be non-existent or very unusual in normal operating environments. Most systems need to be able to handle these unexpected events nonetheless. Therefore, there is a need for a method of introducing these errors in a controlled way.

A bit error generation and logging tool for two proprietary protocols based on the Fibre Channel physical layer has been developed. The hardware platform consists mainly of a Virtex II Pro FPGA with accompanying I/O support. Control of the hardware is handled by a graphical user interface residing on a PC. Communication between the hardware and the PC is handled with a UART. The final implementation can handle four parallel one way links, or two full duplex links, independently. This report describes the implementation and the necessary theoretical background for this.

Keywords
Bit Error Generation, Fibre Channel, Implementation, FPGA, Virtex II Pro
Abstract

Test Tools are very important in the design of a system. They generally simulate a working environment, only at a higher speed, or with less frequently occurring test cases. In the verification of protocols based on the Fibre Channel physical layer, this becomes a necessity, as errors can be non-existent or very unusual in normal operating environments. Most systems need to be able to handle these unexpected events nonetheless. Therefore, there is a need for a method of introducing these errors in a controlled way.

A bit error generation and logging tool for two proprietary protocols based on the Fibre Channel physical layer has been developed. The hardware platform consists mainly of a Virtex II Pro FPGA with accompanying I/O support. Control of the hardware is handled by a graphical user interface residing on a PC. Communication between the hardware and the PC is handled with a UART. The final implementation can handle four parallel one way links, or two full duplex links, independently. This report describes the implementation and the necessary theoretical background for this.
Acknowledgments

I would like to thank Sectra Mamea AB and my supervisor, Fredrik Elmgren, for providing me with an interesting and challenging master’s thesis. Also, I’d like to thank my examiner Olle Seger for taking on my master’s thesis.
# Table of Contents

LIST OF FIGURES ........................................................................................ XI
LIST OF TABLES .......................................................................................... XIII
ACRONYMS ................................................................................................. XV

## CHAPTER 1 - INTRODUCTION ................................................................. 1
  1.1 Background ...................................................................................... 1
  1.2 Goals & Limitations ........................................................................ 1
  1.3 Disposition ...................................................................................... 1
  1.4 Report Target Group ....................................................................... 1

## CHAPTER 2 - THEORY & BACKGROUND ............................................ 3
  2.1 Sectra Mamea AB ........................................................................... 3
  2.2 Target System ................................................................................ 3
  2.3 Hardware Platform, AgentHW ....................................................... 4
  2.4 Xilinx Virtex II Pro FPGA ............................................................... 4
    2.4.1 Slices ....................................................................................... 5
    2.4.2 Block SelectRAM+ .................................................................... 5
    2.4.3 Processor Blocks ...................................................................... 5
    2.4.4 RocketIO ................................................................................ 6
  2.5 UART ............................................................................................... 7
  2.6 ASCII .............................................................................................. 7
  2.7 High-Speed Serial Communication .................................................. 8
    2.7.1 8b/10b Encoding ....................................................................... 8
  2.8 Random Number Generation ............................................................ 9
    2.8.1 Leap-forward Linear Feedback Shift Register ............................ 10
  2.9 Statistical Theory .......................................................................... 12
    2.9.1 Uniform Distribution ............................................................... 12
    2.9.2 Poisson Distribution ............................................................... 12
    2.9.3 Poisson Process ....................................................................... 13
  2.10 Xilinx CORE Generator ................................................................. 13
  2.11 Xilinx Embedded Development Kit ............................................... 13
  2.12 Scilab ........................................................................................... 14
  2.13 Microsoft Visual C# Express ......................................................... 14
  2.14 Active HDL .................................................................................. 14

## CHAPTER 3 - IMPLEMENTATION ......................................................... 15
  3.1 System Overview .......................................................................... 15
  3.2 Design of the Agent ..................................................................... 17
    3.2.1 Log FIFO ............................................................................... 18
    3.2.2 Logging .................................................................................. 18
    3.2.3 Packet State Machine ........................................................... 18
    3.2.4 Statistics Block ...................................................................... 19
    3.2.5 Error Control ......................................................................... 19
    3.2.6 8b/10b Decoder ...................................................................... 19
    3.2.7 Error Insertion ....................................................................... 19
    3.2.8 Agent Control ......................................................................... 19
    3.2.9 Agent I/O Signals .................................................................. 20
  3.3 Statistics Generation .................................................................... 22
List of Figures

Figure 2.1 - Target System ..................................................................................3
Figure 2.2 - AgentHW Block Diagram .................................................................4
Figure 2.3 - RocketIO Data Flow (Figure 2-12 in [2]) ........................................6
Figure 2.4 - UART frame ..................................................................................7
Figure 2.5 - 4-bit LFSR ..................................................................................9
Figure 2.6 - 4-bit Leap-forward LFSR ..............................................................11
Figure 2.7 - PDF of Poisson distribution with $\lambda = 10$ ................................12
Figure 3.1 - System Overview ........................................................................15
Figure 3.2 - Agent Block Diagram ..................................................................17
Figure 3.3 - Uniform Process Block Diagram ..................................................24
Figure 3.4 - Uniform Process Histogram Comparison ....................................25
Figure 3.5 - log2 Block Diagram ....................................................................28
Figure 3.6 - Comparison Between log2 Results .............................................28
Figure 3.7 - Difference Between log2 Results ...............................................29
Figure 3.8 - Poisson Process Block Diagram ..................................................30
Figure 3.9 - Comparison Between Poisson Histograms ................................30
Figure 3.10 - Statistics Component ..................................................................31
Figure 3.11 - Processor Block Architecture ...................................................32
Figure 3.12 - PPC Main Loop .........................................................................39
Figure 3.13 - Agent Log ..................................................................................39
Figure 3.14 - UART Handler ..........................................................................39
Figure 3.15 - Agent GUI ...............................................................................40
List of Tables

Table 2.1 - XC2VP30 Properties .......................................................... 4
Table 2.2 - Supported memory configurations ................................ 5
Table 3.1 - Agent Control Register .................................................. 20
Table 3.2 - Agent I/O Signals ............................................................ 21
Table 3.3 - RxFlags Composition .................................................... 21
Table 3.4 - log2 LUT Contents ......................................................... 27
Table 3.5 - IPBC Address Partitioning .......................................... 34
Table 3.6 - Currently Defined Devices ........................................... 34
Table 3.7 - Control Registers ......................................................... 34
Table 3.8 - Status and Control Register ...................................... 35
Table 3.9 - Agent Registers ............................................................ 36
Table 3.10 - UART Configuration .................................................. 37
Table 3.11 - Agent GUI Settings .................................................... 41

List of Tables, Appendixes

Table A.1 - Agent-FUP Header ..................................................... 47
Table A.2 - Defined Agent-FUP Commands ............................... 48
Table B.1 - Agent-SUP Command Limitations ........................... 49
Table B.2 - Agent-SUP Commands ............................................. 50
Acronyms

ASCII  American Standard Code for Information Interchange
BER    Bit Error Rate
BRAM   Block SelectRAM+
CE     Chip Enable
CLB    Configurable Logic Block
CPU    Central Processing Unit
CRC    Cyclic Redundancy Check
EDK    Embedded Development Kit
GUI    Graphical User Interface
FIFO   First-In First-Out
FPGA   Field Programmable Gate Array
I/O    Input/Output
IDE    Integrated Development Environment
IP-Core Intellectual Property Core
LED    Light Emitting Diode
LFSR   Linear Feedback Shift Register
LUT    Look Up Table
OCM    On-Chip Memory
OPB    On-Chip Peripheral Bus
PCS    Physical Coding Sublayer
PDF    Probability Density Function
PLB    Processor Local Bus
PLL    Phase Locked Loop
PMA    Physical Media Attachment
PPC    PowerPC
SERDES Serializer/Deserializer
V2P    Xilinx Virtex II Pro
UART   Universal Asynchronous Receiver/Transmitter
Chapter 1 - Introduction

1.1 Background
Test tools are very important in the design of a system. They generally simulate a working environment, only at a higher speed, or with less frequently occurring test cases. In the verification process of protocols based on the Fibre Channel physical layer, this becomes a necessity, as errors can be non-existent or very unusual under normal operating environments. Most systems need to be able to handle these unexpected events nonetheless. Therefore, there is a need for a method of introducing these errors in a controlled way.

1.2 Goals & Limitations
The goal of this thesis is to produce a tool that can introduce bit-errors on the two protocols that Sectra Mamea AB uses on their multi-gigabit serial links, namely BitStorm and ByteBreeze. Bit-errors are to be introduced as a uniform or a Poisson process with variable mean. The hardware must be implemented on an already existing hardware platform, namely the IPB hardware (from now on AgentHW). A graphical user interface (GUI), for control of the AgentHW, that will run on a normal PC will have to be developed. The GUI must be able to show logs of the packets that are corrupted and other relevant data.

This thesis will not cover the actual testing of how their systems react to the faults generated by this tool. This will be done by Sectra Mamea AB themselves.

1.3 Disposition
The report is divided into four main sections: Introduction, Theory & Background, Implementation and Conclusions. The Theory & Background chapter introduces the relevant background and theory for this thesis. The Implementation chapter describes the implementation of all the parts in this thesis and the result of each and every one of them. Finally, the Conclusions chapter presents a broader result and future improvements.

1.4 Report Target Group
The intended reader is an electrical engineer with a broad knowledge in digital hardware and software. This report contains no source code; the code is kept internally for Sectra Mamea AB.
Chapter 2 - Theory & Background

2.1 Sectra Mamea AB
Sectra Mamea AB is a subsidiary of Sectra AB. At Sectra Mamea AB they develop a digital mammography solution based on photon counting. In their Linköping office they develop most of the electronics and software for the product, while the mechanical hardware is developed in their Stockholm office.

2.2 Target System
The system that the tool developed in this thesis will work in is outlined in Figure 2.1.

![Target System Diagram]

As can be seen in Figure 2.1, the system consists of four devices communicating across two different high-speed serial links. These two links are named ByteBreeze and BitStorm. ByteBreeze and BitStorm have bit rates of 1.25 Gbit/s and 2.5 Gbit/s respectively. Both the protocols are based on the physical layer for Fibre Channel (see section 2.7). Each direction of one link is a separate independent serial link. The bit error generator has got to be able to insert errors in both links and in both directions.
2.3 Hardware Platform, AgentHW

The hardware platform used is designed by Sectra Mamea AB. From here on this hardware will be referred to as AgentHW. A block diagram of the relevant parts of the AgentHW can be seen Figure 2.2.

![AgentHW Block Diagram](image)

The AgentHW has Input/Output (I/O) connectors and electronic interfaces for both the multi-gigabit serial links (BitStorm and ByteBreeze) and the slower speed UART. For the UART, an rs232 level converter chip is installed as well as a DSUB-9 connector. LEDs and push-buttons are included for debug/status and external reset purposes. There is a JTAG interface for configuration and programming of the FPGA.

2.4 Xilinx Virtex II Pro FPGA

The Virtex II Pro (V2P) FPGA is a state of the art FPGA from Xilinx. They offer an entire embedded system on one chip, including CPU, memories and FPGA fabric to design additional devices. The V2P model used in this thesis is the XC2VP30; some of its more important properties can be seen in Table 2.1 [1].

<table>
<thead>
<tr>
<th>RocketIO Transceiver Blocks</th>
<th>PowerPC Processor Blocks</th>
<th>Logic Cells</th>
<th>Slices</th>
<th>18 x 18 Multiplier Blocks</th>
<th>Block SelectRAM+</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>30816</td>
<td>13696</td>
<td>136</td>
<td>136</td>
</tr>
</tbody>
</table>

Table 2.1 - XC2VP30 Properties
2.4.1 Slices
The FPGA logic consists of what Xilinx refers to as slices. A slice consists of two 4-input function generators, two storage elements and some additional logic to speed up various logic functions. A group of four slices is called a Configurable Logic Block (CLB).

A slice is widely configurable and contains logic to accelerate e.g. shifting, multiplexing and other common logic functions. The 4-input function generator can be configured as a look up table, distributed RAM or a 16-bit shift register [1].

2.4.2 Block SelectRAM+
Block SelectRAM+ (BRAM) is a very useful feature of the V2P. It offers access to fast synchronous ram embedded on the FPGA. The BRAM can be configured as either single or dual port memories. Both ports can be independently clocked. The supported memory configurations can be seen in Table 2.2.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16k x 1 bit</td>
<td>4k x 4 bit</td>
<td>1k x 18 bit</td>
</tr>
<tr>
<td>8k x 2 bit</td>
<td>2k x 9 bit</td>
<td>512 x 36 bit</td>
</tr>
</tbody>
</table>

Table 2.2 - Supported memory configurations

Larger memories can be built by combining several BRAMs. This proves to be an extremely important feature when implementing them as high-speed FIFOs (First-In First-Out memories) [1].

2.4.3 Processor Blocks
The V2P series FPGAs can have up to two processor blocks. These processor blocks consist of a PowerPC 405 (PPC) core, On-Chip Memory (OCM) controllers and interfaces, clock control/interface logic and CPU-FPGA interfaces [1].

2.4.3.1 PowerPC 405 Core
The PPC is a 0.13 µm implementation of the PPC405D4 core that can be clocked at 300+MHz. With logic that connects the PPC to the FPGA fabric, a very fast and deterministic embedded CPU is obtained. A FPGA is very powerful on its own, but it is not very well suited for controlling purposes, such as a UART controller or settings manager. There are solutions where a soft core can be used. A soft core means that the CPU is implemented in the FPGA fabric. This solution takes up more silicon, is slower and has got higher power consumption. But this solution is commonly used when there is no CPU included in the silicon [1].
2.4.3.2 **CPU-FPGA Interface**

All the processor block pins connect directly to the FPGAs routing resources through the CPU-FPGA interface. As a result, all the processor signals can be routed by the same means as any other FPGA signal. Data interface signals are connected to the CPU via its busses, namely the fast Processor Local Bus (PLB) or the slower On-Chip Peripheral Bus (OPB) [1].

2.4.4 **RocketIO**

To be able to communicate over fast serial interfaces the V2P FPGA has RocketIO transceivers. These transceivers are based on MindSpeeds SkyRail™ technology. The transceiver can operate at several speeds varying from 600 MB/s to 3.125 Gb/s. The transceivers can receive and transmit data based on Fibre Channels physical link specification.

![RocketIO Data Flow](image)

**Figure 2.3 - RocketIO Data Flow (Figure 2-12 in [2])**

A RocketIO transceiver block consists of two main parts: the Physical Media Attachment (PMA) and the Physical Coding Sublayer (PCS). The main connections and components of these two blocks can be seen in Figure 2.3.

The PMA is responsible for transmitting and receiving the actual serial data stream. It contains the clock recovery/generation unit and the serializer/deserializer (SERDES).

On a higher level the PCS is found. It is responsible for the preparing and handling of the data before it is sent and after it is received. It contains an elastic buffer on the receiver. The elastic buffer is responsible for the clock domain correction between the recovered receiving clock and the clock used in the FPGA fabric. There are also CRC-units (Cyclic Redundancy Check) that can be activated if needed. The 8b/10b encoder/decoder can also be disabled if it is not needed [2].
2.5 UART

The UART (Universal Asynchronous Receiver/Transmitter) is a serial interface for asynchronous communication between digital devices. It is, due to its ease to implement, very commonly used. A UART is a single wire (per direction), point-to-point interface where the two devices communicate over a predefined transmission rate. If the communication must be in both directions, then two separate wires are needed [3].

In Figure 2.4 a typical UART frame is shown. When communication is idle the transmitter ties the line to '1'. A new frame is always started with a start bit, which is a '0'. The start bit is followed by transmission of the data, least significant bit first. The width of the data sent can range from five to eight bits. Directly after the data comes an optional parity bit. The parity bit is calculated by XORing all the data bits together. Finally the frame is terminated with one or two stop bits. A stop bit is represented with a '1' and is used to separate adjacent frames.

![UART Frame Diagram](image)

**Figure 2.4 - UART frame**

The configurable parts have to be set before the transmission starts. Both the sender and the receiver need the same configuration for the communication to be successful. What needs to be configured is:

- Transmission rate
- Number of data bits, five to eight
- If parity is used
- Number of stop bits, one or two

2.6 ASCII

ASCII (American Standard Code for Information Interchange) is a character encoding for representation of text in computer communication (such as UART communication). The encoding is based on the English alphabet. Originally ASCII used seven bits to define 128 different characters. The ASCII standard has since then been extended (Extended ASCII) to eight bits to allow for international characters.
The first 32 characters (0-31) consist of control characters that cannot be printed. These characters are used to provide information about the data stream. Since these codes are separated from the rest, it is easy to separate the data from the control [4].

2.7 High-Speed Serial Communication

The high-speed serial links that are discussed in this thesis are all of the same type. They use the same physical layer to transmit and receive data. This layer is specified in the Fibre Channel specification and has the following properties:

- Serial one-directional link, encoded as a differential signal for better signal integrity.
- 8b/10b encoded for guaranteed clock recovery and DC-balancing.

2.7.1 8b/10b Encoding

Since there is no separate channel for sending the clock signal, it must be embedded in the data signal. This is done by ensuring that there are enough transitions in the transmitted signal for a Phase Locked Loop (PLL) to recover the clock. When Fibre Channel was developed, the 8b/10b encoding scheme was chosen to avoid this problem. This coding ensures that the data has enough transitions for clock recovery. It also has the additional benefit of giving separate control characters, running disparity retention and better error detection.

8b/10b encoding works by encoding the high 3 bits into 4 bits and the low 5 bits into 6 bits. These two parts are then combined into new 10-bit character. Each character is named Dxx.y where xx and y are the values of the 5- and 3-bit group respectively. An example of the naming scheme can be seen in Example 2.1.

The 8-bit number 131 in decimal notation (d131) is 10000011 in binary notation (b10000011). When this is divided into groups, the following is obtained b100.00011 (y.xx). This is then translated to D03.4.

Example 2.1

The number of transitions in a character is very important for the clock recovery. If the data sent was not encoded, then a block of ten characters with the value zero would mean eighty bits without a single transition. This makes it hard for the PLL to recover the clock correctly. Three to eight transitions per 10-bit character is guaranteed when the 8-bit character is encoded with 8b/10b coding. For example, the character 0b00000000 is encoded to 0b100110100 (-) or 0b0110001011 (+). The number of transmissions has clearly increased.
There are also twelve special control characters, these are similarly named Kxx.y. These characters are encoded separately and can thus be easily identified in a stream of normal characters.

The disparity is defined as the sum of the individual bits in a 10-bit character where a ‘1’ gets a value of +1 and a ‘0’ gets a value of -1. Each 10-bit character is coded to never have a disparity of more than +2 or less than -2. Each 8-bit character can then be encoded into two different 10-bit values which have inverted disparity. This ensures that the running disparity, the sum of the disparity of all previous characters, will always remain within -2 and +2. The two encoded versions of the same characters are called Dxx.y+ and Dxx.y- for the positive and negative version respectively.

Bit error detection is made easier with this scheme. An error can be directly detected if the received character has been converted to a new 10-bit character that is not in the decoding table or if it has the wrong disparity [5].

2.8 Random Number Generation

Generation of uniform random (from now on simply referred to as random) numbers in hardware is a common task. Random numbers are often used in simulation tasks, communication algorithms and many other applications. In this thesis they are used in the generation of statistical distributions.

The easiest way to generate pseudo random bit sequences is by using a linear feedback shift register (LFSR). A typical LFSR generates the new bit to be shifted in to a shift register by feeding back two or more bits, from the shift registers current state, through a XOR gate. The bits that are fed back are called taps. By choosing the taps carefully, a $2^n - 1$ bit (where n is the length of the shift register) repeating pseudo random sequence can be constructed. An example of a 4-bit LFSR with the (maximum length) tap equation $x^4 + x^1$ is shown in Figure 2.5 (a FDD unit is normal D-flip-flop) [6].

![Figure 2.5 - 4-bit LFSR](image-url)
Unfortunately this simple method can not be used when random numbers with multiple bits are needed. Since n-1 bits out of n stay the same (only shifted one step) between each shift the correlation between adjacent shifts is too big. Some methods to solve this problem are briefly presented below; the chosen method is then described more carefully [7].

1) **Running multiple LFSRs in parallel.**  
This way multiple bits can be generated. These will be independently generated random bits and thus the whole word will be random. This method is rejected due to the need of several seeds (one seed for each LFSR) and the space inefficiency.

2) **Letting the LFSR tick n times per word.**  
Another trick that can be used is letting the LFSR tick through a whole word between adjacent reads. There will then be no correlation between bits as all the old bits are shifted out before a new word is read. This method is rejected due to the number of clock cycles it takes to generate a new word.

3) **Lagged Fibonacci Generator.**  
This method produces a new n-bit random number directly from two n-bit random numbers from the generators previous states. It is fairly simple to implement but is rejected due to the need multiple seeds or a more complicated initialization than the Leap-forward LFSR.

4) **Leap-forward LFSR.**  
This method utilizes the fact that the LFSR is a linear system. The state of the LFSR can then be calculated many steps into the future by changing the transformation matrix of the linear system. This method uses a substantial amount of resources in form of XOR-gates, but its single clock cycle random generation and single seed makes it an appropriate method.

The choice of method is motivated in section 3.3.2.

### 2.8.1 Leap-forward Linear Feedback Shift Register

A LFSR is, as the name suggests, a linear system. The LFSR in Figure 2.5 can be represented with Eq. 2.1 where:

\[
A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \quad q(t) = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}
\]

\[
q(t+1) = A \cdot q(t)
\]

Eq. 2.1
Each row in the matrix represents the input of the corresponding flip-flop (row zero corresponds to the input of the latch with q0 as its output). When there are more than one ‘1’ in a row, it means that these two signals have to be XORed together (modulo-2 addition, that is \(1 + 1 = 0\)) to generate the new input. All the elements in the vectors/matrices are 1-bit numbers. As a result of this, all additions (when doing matrix-vector multiplications) become modulo-2. By using Eq. 2.1 in multiple steps the following can be calculated:

\[
q(t + 2) = A \cdot q(t + 1) = A \cdot A \cdot q(t) = A^2 \cdot q(t)
\]

More generally, if \(q(t + n)\) is sought, one can simply use:

\[
q(t + n) = A^n \cdot q(t) \tag{Eq. 2.2}
\]

To leap four steps at the time with the LFSR in Figure 2.5, the following transformation matrix is calculated:

\[
A^4 = \begin{bmatrix} 
1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 \\
\end{bmatrix}
\]

The end result represents a more complicated design with many more XOR-gates, as can be seen in Figure 2.6. The method is still usable as the resource use is not too exaggerated.

Figure 2.6 - 4-bit Leap-forward LFSR

The final output vector is \([q_3 \ q_2 \ q_1 \ q_0]\). This vector will be equivalent to the original LFSRs output, but having it tick four steps between each read.
2.9 Statistical Theory
The background to understand the statistical theory used in this thesis will be presented in this section.

2.9.1 Uniform Distribution
A uniform distribution has the same probability over the whole data range. Its probability density function (PDF) is defined as [8]:

\[ f(x) = \begin{cases} \frac{1}{\text{max} - \text{min}} & \text{for } \text{min} \leq x \leq \text{max} \\ 0 & \text{otherwise} \end{cases} \]

The mean of a uniform distribution can be calculated with Eq. 2.3.

\[ \text{mean} = \frac{\text{max} + \text{min}}{2} \]

Eq. 2.3

2.9.2 Poisson Distribution
The Poisson distribution is useful for applications that involve counting the number of times a random event occurs in a given time. Examples of events that can be described with a Poisson distribution include incoming telephone calls to a service department or errors occurring on a transmission. The Poisson distribution has a single parameter, \( \lambda \), that is both the mean and the variance of the distribution. The probability distribution function (PDF) of the distribution is [8]:

\[ f(x) = \frac{\lambda^x \cdot e^{-\lambda}}{x!} \quad \text{for } x = 0, 1, 2, 3, \ldots \]

A plot of a Poisson PDF with \( \lambda = 10 \) can be seen in Figure 2.7.

![Figure 2.7 - PDF of Poisson distribution with \( \lambda = 10 \)](image)
2.9.3 Poisson Process
The following is scilab (see section 2.12) code for generation of a Poisson process with intensity "i" \((i=\lambda)\) [8]:

```scilab
function [x]=poi(i)
    t = 0;
    b = 0;
    while (b == 0)
        t = t - log(rand())/i;
        x = x + 1;
        if (t > 1)
            b = 1;
        end
    end
endfunction
```

This code can be modified to act as a real time Poisson process. The method of doing this will be shown in section 3.3.4.

2.10 Xilinx CORE Generator
Xilinx CORE Generator (from now on referred to as CoreGen) is a tool used to generate IP-cores (Intellectual Property cores). An IP-core is a hardware block that is pre-designed by a company/individual that can be inserted into a design and used.

Within CoreGen there exist possibilities to generate many types of IP-cores with many different configurations. Some of the more advanced cores have to be licensed from Xilinx, but most of the simpler cores are free to use. The use of IP-cores can help speed up the design of a product substantially as these blocks are already validated. Time can thus be saved both in the design and the testing stage.

2.11 Xilinx Embedded Development Kit
The Xilinx Embedded Development Kit (EDK) is a tool suite that facilitates the design of embedded environments on Xilinx FPGAs. With the help of EDK it is easy to start an embedded design. There are wizards to help generate the base environment with CPU, buses and memory connections. EDK has many components predefined that can easily be added to the CPU buses, or custom components can be written in VHDL and added.

The finished component can be directly synthesized from the EDK tools, or it can be instantiated as a component in a bigger design. The design can then be synthesized without having to do it through EDK.
The code that is written for the embedded CPU can be compiled separately. The compiled bit-file is then inserted into the FPGA logic bit-file. This way, the design does not have to be synthesized every time the CPU code changes. This is very valuable, as the synthesis time can be very long for complex designs.

2.12 Scilab

Scilab is an open source alternative to Matlab. It is a platform for numerical computation. It uses virtually the same syntax as Matlab, and many of the native Matlab functions can be used. Tools like these are often used to test algorithms before implementing them and also to verify the output of an implemented algorithm. There are many toolboxes written for scilab. Toolboxers are predefined functions and tools for special advanced functions. Scilab and its many toolboxes can be downloaded for free at http://www.scilab.org.

2.13 Microsoft Visual C# Express

C# (pronounced C-sharp) is a programming language developed by Microsoft. It is an evolution from C and C++ and it is a type safe object oriented language. C# code is compiled to managed code, this means that the code is compiled to machine code on runtime. This method allows for benefits such as machine specific optimizations, garbage collection, language interoperability and more.

Microsoft Visual C# Express is a free to use version of the Microsoft Visual Studio IDE. The Express edition lacks some of the more advanced tools of the Visual Studio IDE, but it is often enough. The Express edition has no licensing conditions that restrict its use in a commercial product.

More information, and possibilities to download Microsoft Visual Express, can be found at http://msdn.microsoft.com/vstudio/express/.

2.14 Active HDL

Active HDL is an IDE for FPGA development that is developed by Aldec. It is a comprehensive tool suite with visual features such as syntax highlighting and automatic indentation, as well as syntactic validation. It has a powerful integrated simulation environment that can handle large simulation tasks with many complex stimuli possibilities.
Chapter 3 - Implementation

3.1 System Overview

An overview of the whole system can be seen in Figure 3.1. The control and data flow in the system will be briefly described to give a better base for the forthcoming sections that describe the individual parts in detail.

In Figure 3.1, there are two devices per link communicating in full duplex. This configuration is just an example of how it can be connected. It doesn’t have to be full duplex, it can just as well be a single ended communication.

As soon as a link is connected, the Agent will start passing through the data so that the link is intact. The sender and receiver will not notice that the Agent is connected in between them. When error generation is activated in an Agent (when a “session” is started), it will start introducing errors in the data stream that passes through it. It will accomplish this by inverting single bits in the data stream.
All the control a user has of the system is managed from the GUI. Commands and data to/from the AgentHW are sent/received over the UART. The settings from the GUI are sent to the AgentHW when the user initiates a new session. Theses setting are received by the PPC. The PPC will then:

- Write the new seed to the seed register.
- Write the control registers of each Agent.
- Write the statistics control registers of each agent.
- Write to the global control register. Here it will enable all Agents and write the restart bit. This will restart all Agent error generation with the new settings.

When a session is running, the PPC will read the logs from the Agents and update the statistics counters. It will send the logs along to the GUI when this is needed.

A session will run until the user stops it from the GUI. As long as the session is running, errors will continuously be inserted in the data stream. The errors will be introduced according to the statistical settings that were setup when the session was initiated. When the necessary data has been collected, the log file can be saved for further analysis and a new session with a new set of data can be initiated.
3.2 Design of the Agent

The Agent is the device responsible for interfacing with the high speed data stream. Its purpose is to keep track of what state the transmission is currently in. It will also introduce errors into the stream based on the current state of the transmission and the current error generation configuration. The Agent will act as a “man-in-the-middle”. The data stream thus goes through the Agent, where the errors are inserted.

The main blocks of the Agent, as seen in Figure 3.2, will be presented one-by-one. Something that must be noted is that there are two versions of the Agent, one version for each protocol. There have to be two versions since ByteBreeze and BitStorm work at different bitrates and have different protocols. The general architecture of the Agent remains the same between both versions. The two versions will from now on be separated by calling them AgentBB and AgentBS for the ByteBreeze and BitStorm versions respectively.
3.2.1 Log FIFO
The log FIFO is used as a buffer for the log messages that are generated. Since the messages are read by the PPC, a buffer is needed. This is because sometimes the PPC will be busy and will not have time to read a message before one or several new messages have been generated. It also serves a second purpose. On the AgentBB the data path is clocked at 62.5MHz while the rest of the FPGA is clocked at 125MHz. The FIFO will then serve as an excellent clock domain synchronization device, since it can be clocked with different read and write clocks.

The depth of the individual FIFOs can easily be changed by simply generating new FIFO core in CoreGen and re-synthesizing the design. No HDL code needs to be changed since FIFOs have no external addressing, thus the depth of the FIFO will not change the width of any input signals.

3.2.2 Logging
As the name reveals, the logging block is responsible for generating the packet logs. The log will be based on the state that the packet state machine (see section 3.2.3) is currently in. It will then build up a packet log with the chosen data. On the special logging events this data will then be written to the log FIFO, otherwise it will just be discarded.

Since the AgentBS and AgentBB log two entirely different protocols, their logs will also be very different. In AgentBS, the whole packet header is logged with some additional data, such as a packet counter, an inter-packet timer, error logs (for external errors that are detected), error generation logs and more. In the AgentBB case, just the different packet type headers are stored and some additional control data. The structure of the log and the amount of data logged can easily be changed for both protocols – it is just to widen the FIFO and/or change its input vector.

3.2.3 Packet State Machine
This state machine is used to keep track of the current transmission state. As this is heavily protocol based, it will be completely different between AgentBS and AgentBB. In general a protocol can usually be divided into the following state: ERROR, IDLE, HEAD and DATA. These general states can then be partitioned into more detailed cases, but they give a good overview of what is currently being received. Exactly what is currently happening can then be figured out based on the current state, the previous state and the incoming data. For example, if the state machine is in the IDLE state and SOP (Start of Packet) is received, then this can be assumed to be a valid SOP. If instead the current state was IDLE and a EOP (End of Packet) was received, then the next state will be the ERROR state as this was not supposed to happen.
3.2.4 Statistics Block
The statistics block is used to generate error events. When an error event is generated, a bit error is inserted into the data stream. The event generation is controlled via a CE (Clock Enable) input. The statistics block will be further described in section 3.3.

3.2.5 Error Control
One of the requirements for this thesis was that errors could be steered into certain pre-decided positions (still randomly occurring, just at certain positions). The error control block does this by controlling the ce input of the statistics block. If, for example, errors are to be generated only in packets (not when IDLE or ERROR), then the error control block controls ce so that it is only high when a packet is being received. The statistics block will only be active when ce is high, thus events will only be generated in packets.

3.2.6 8b/10b Decoder
The RocketIO transceiver has a built in 8b/10b decoder (see Figure 2.3) that takes no additional FPGA space. Unfortunately this decoder can not be used. The reason for this is that the errors need to be inserted into the stream just as if it was an external error. This means that the errors need to be introduced on the encoded data. Therefore the raw 10-bit data needs to be taken from and written to the PMA directly without it being decoded or encoded in the PCS.

To be able to interpret the data in the packet state machine there is still a need to decode the data, but this must be done outside the path of the raw data stream. As Xilinx has an already verified and compact 8b/10b decoder available as a free IP, this was used.

3.2.7 Error Insertion
The error generation is controlled by the event signal from the statistics block. If the event signal is high, the data stream is XORed with an error mask. The error mask is simply a shift register with the same width as the data path. The register has one single bit set to '1' and the rest set to '0'. This register is then rotated one step every clock cycle. By using this method, the bit that gets inverted will change every clock cycle.

The data stream is delayed a number of clock cycles in this block. This must be done since the incoming data is analyzed (in the packet state machine and in the error control block) to see if errors are allowed to appear in this specific data. Therefore it is important that the error events generated in the statistics block corresponds to this particular data in the stream.

3.2.8 Agent Control
This block handles the reset and control routines of the Agent. It contains the Agent control register (detailed in section 3.2.8.1) and it takes different actions depending on what is currently stored in the register.
3.2.8.1 **Agent Control Register**

This 8-bit register contains the different Agent control bits. The bits are detailed in Table 3.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Unused</th>
<th>Unused</th>
<th>Unused</th>
<th>EGDS</th>
<th>EGCS</th>
<th>EGE</th>
<th>ELE</th>
<th>PLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit number</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Accessibility*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
</tr>
</tbody>
</table>

(*) R – Read only, W – Write only, RW – Read Write, X – Not Used

**Table 3.1 - Agent Control Register**

The bits have following properties:

**PLE** - Packet Log Enable
1 if packet logging is enabled, 0 otherwise.

**ELE** - Error Log Enable
1 if error packet logging is enabled, 0 otherwise.

**EGE** - Error Generation Enable
1 if error generation is enabled, 0 otherwise.

**EGCS** - Error Generation CE Selection
1 if errors should be steered into packets only, 0 if errors can occur anywhere.

**EGDS** - Error Generation Device Selection
Is connected directly to the device selection signal of the statistics block.

### 3.2.9 Agent I/O Signals

All the Agents I/O signals are briefly detailed in Table 3.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td>1</td>
<td>In</td>
<td>Clock synchronized to the RocketIO data.</td>
</tr>
<tr>
<td>ClkExternal</td>
<td>1</td>
<td>In</td>
<td>System clock. This input only exists in the AgentBB since AgentBS has the same Clk and ClkExternal.</td>
</tr>
<tr>
<td>Reset</td>
<td>1</td>
<td>In</td>
<td>Reset, active high.</td>
</tr>
<tr>
<td>ClearBuffer</td>
<td>1</td>
<td>In</td>
<td>Clear FIFO, active high.</td>
</tr>
<tr>
<td>ErrGenRestart</td>
<td>1</td>
<td>In</td>
<td>Restart error generation, active high.</td>
</tr>
<tr>
<td>AgentEn</td>
<td>1</td>
<td>In</td>
<td>Agent enable, active high. Agent is enabled as long as this signal is asserted.</td>
</tr>
</tbody>
</table>
### Chapter 3 - Implementation

#### Table 3.2 - Agent I/O Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentCtrl</td>
<td>8</td>
<td>In</td>
<td>Agent control. Input signals for the Agent control register.</td>
</tr>
<tr>
<td>AgentCtrlWe</td>
<td>1</td>
<td>In</td>
<td>Agent control write enable. The Agent control input is written into the register when this signal is high.</td>
</tr>
<tr>
<td>StatCtrl</td>
<td>32</td>
<td>In</td>
<td>Statistics control. Inputs signals for the statistics control register.</td>
</tr>
<tr>
<td>StatCtrlWe</td>
<td>1</td>
<td>In</td>
<td>Statistics control write enable. The statistics control input is written into the register when this signal is high.</td>
</tr>
<tr>
<td>Seed</td>
<td>32</td>
<td>In</td>
<td>Seed. Used by the statistics block.</td>
</tr>
<tr>
<td>SeedWe</td>
<td>1</td>
<td>In</td>
<td>Seed write enable. The seed is written to its register when this signal is high.</td>
</tr>
<tr>
<td>DataOut</td>
<td>256/32</td>
<td>Out</td>
<td>Log FIFO output. The width of the signal is 256 bits for AgentBB and 32 bits for AgentBS.</td>
</tr>
<tr>
<td>DataOutRE</td>
<td>1</td>
<td>In</td>
<td>Log FIFO output read enable. A new FIFO line is read when this signal is high.</td>
</tr>
<tr>
<td>DataOutOvf</td>
<td>1</td>
<td>Out</td>
<td>Log FIFO overflow, active high. Indicates that the FIFO has overflowed. Is cleared on Reset, ClearBuffer or DataOutRE.</td>
</tr>
<tr>
<td>DataOutEmpty</td>
<td>1</td>
<td>Out</td>
<td>Log FIFO empty, active high. Indicates that the Log FIFO is empty.</td>
</tr>
<tr>
<td>TxData</td>
<td>20</td>
<td>Out</td>
<td>The RocketIO data to be sent out.</td>
</tr>
<tr>
<td>RxData</td>
<td>20</td>
<td>In</td>
<td>The incoming RocketIO data.</td>
</tr>
<tr>
<td>RxFlags</td>
<td>4</td>
<td>In</td>
<td>Status signals from the RocketIO transceiver.</td>
</tr>
</tbody>
</table>

All the Agents I/O signals are synchronized to ClkExternal except TxData, RxData and RxFlags. Clock domain synchronization is handled inside the Agent. Seed and StatCtrl are written to internal buffer registers. These are then transferred into the statistics block on an ErrGenRestart event.

RxFlags is a collection of important signals from the corresponding RocketIO block. The signals are specified in Table 3.3.

<table>
<thead>
<tr>
<th>Bit</th>
<th>3</th>
<th>2-1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RocketIO signal</td>
<td>TXBUFERR</td>
<td>RXBUFSTATUS</td>
<td>RXCOMMADEDET</td>
</tr>
</tbody>
</table>

#### Table 3.3 - RxFlags Composition
3.3 Statistics Generation

This section will describe the statistics block and how it was implemented. The basic necessities of this block, such as data rate and data width in general, will be discussed first. After that, the implementations are described.

3.3.1 Initial data rate and width investigation

The statistics block is going to be the base for the error generation. It will be used to decide if the current sample in the data path will have a bit error or not. As a result, the statistics has to be updated at the same rate as the data. This means that the statistics block needs to be clocked at least 125MHz (the BitStorm clock). Initiation time is not important to be low for this application. This means that the statistics block can be pipelined as much as needed. To have correct statistics from the first valid clock cycle, the pipeline must be filled to the end.

The width of the random number generation block is dependant on the width that the subsequent blocks need. These are the uniform and Poisson process blocks. Their requirement on the width of the random numbers will be further discussed in their sub-sections following below, but the result is that a 32-bit width will be used.

3.3.2 Random Number Generation

To be able to generate a uniform and a Poisson distribution, a pool of uniform random numbers is needed. As stated in section 3.3.1, a new 32-bit random number is needed at a rate of at least 125MHz. The choice of method for the random number generation stood between the leap-forward LFSR and the Lagged Fibonacci Generator, the other two methods were rejected due to size and speed.

The Lagged Fibonacci Generator is fairly simple to implement and it has a compact size, but it was rejected due to its necessity of multiple seeds. There are methods to get around this problem. One could use a single seed to feed a LFSR that can generate the new seeds. This would lead to an even more complex and lengthy initialization.

Instead the Leap-forward LFSR was chosen (see section 2.8). The implementation of this block is described in section 3.3.2.1.

3.3.2.1 Implementation of the Leap-forward LFSR

As stated in section 2.8.1, a Leap-forward LFSR is constructed from a normal LFSR base. To create a 32-bit Leap-forward LFSR, a normal 32-bit LFSR to base it on is needed. The following tap positions were taken, from Table 8 in the Xilinx LogiCore Linear Feedback Shift Register v3.0 datasheet, for maximum length sequences:

\[ x^{32} + x^{22} + x^2 + x^1 \]
A new transformation matrix can then be calculated based on this polynomial. This is a 32 by 32 matrix multiplication. Scilab was used to perform this calculation.

The next problem is translating this new matrix to VHDL. Every ‘1’ in a row of the matrix means that signal needs to be XORed with the rest of the elements that are ‘1’ in that row. Converting the matrix to VHDL code by hand would be very time consuming and error prone. Instead a program was written in C# to translate the matrix (from a file created in scilab containing the matrix). This program simply generated the input signals for the shift register elements. This is an example of such a line generated (the shortest line actually):

\[
\text{LfsrIn}(0) \leftarrow \text{LfsrOut}(31) \oplus \text{LfsrOut}(30) \oplus \text{LfsrOut}(10) \oplus \text{LfsrOut}(0)
\]

The worst case in the implemented Linear-feedback LFSR had 11 XOR gates in such a statement.

### 3.3.3 Uniform Process

Implementing a uniform process is not too complicated once uniform random numbers can be generated. The uniform random numbers that are generated have a mean of \(2^{\text{max}} - 1/2\). Modifying the mean of this distribution can be done by limiting the range of \(r\). Since zero is the lower limit of the random numbers the mean will become \(\text{max}/2\).

The normal way of limiting the range of random numbers is by using modulo. The modulo (operator: \%) operation calculates the remainder of an integer division. To generate a new distribution based on a distribution with a higher mean the following operation can be done:

\[
\text{rand} \mod (\text{mean} \cdot 2 + 1)
\]

This is easy to do on a computer, but it is not as easy to do in hardware. Doing modulo in hardware basically means implementing a divisor. This is complicated and requires a great deal of logic. Another way to do this is to limit the modulo value (and thus the mean) to values of the power of two. The modulo operation then becomes a simple right shift operation, and this is very easy to do in hardware. The downside is that the steps between adjacent means will be very coarse, but this is of no great importance in this application.

Once the uniform distribution with variable mean is calculated it is easy to generate a process. It is as simple as taking a random value from the uniform distribution and loading it in a down counter. When the counter reaches zero there is an event. The counter is then loaded with a new value from the uniform distribution and the event signal is asserted.
The width of the incoming random number decides max value of the uniform distribution. A lower limit of at least one error in 1.25 gigabit data was set. This means that the max value must be at least \( 1.25 \cdot 10^9 \) if each bit is counted. But because the internal data path of the data stream is 20-bit wide, each error event introduces an error into a 20-bit word. Therefore, the max value can be divided to \( \frac{1.25 \cdot 10^9}{20} = 6.25 \cdot 10^7 \leq 2^{26} \). This means that the incoming random number must have at least 26 bits for this to be possible.

### 3.3.3.1 Implementation of the Uniform Process

The block diagram of the implemented uniform process can be seen in Figure 3.3.

![Uniform Process Block Diagram](image)

A new session is initiated by pulsing the `restart` pin. The restart control block then takes care of holding `restarthold` high while the pipeline is filled to the end. When `restarthold` is held low the uniform process is ready, the `ready` signal is thus simply `restarthold` inverted. To mask out any possible events happening while the device is initializing, the `event` signal is filtered via an and-gate with the `ready` signal. After a reset, `ready` will be held low until `restart` has been pulsed.
When `restart` is pulsed the current input on `scale` is sampled into the scaling register. The mean of the uniform process is controlled via the `scale` signal. To generate a mean of $2^{10} - 0.5$, the max value of the uniform distribution must be set to $2 \cdot 2^{10} - 1 = 2^{11} - 1$. Since a 32-bit uniform random number is used, it needs to be scaled by $2^{32-11} = 2^{21}$. This means that the incoming uniform random numbers have to be right shifted 21 steps. The scaling factor would then be 21.

Each time the counter reaches zero it loads a new value from the scaled uniform random number pool. The event is only forwarded to the output if `ce` is asserted, but the counter only loads a new value if `ce` is asserted so no event will be lost.

### 3.3.3.1.1 Result

A simulation, with the mean set to 15.5, was done from ActiveHDL. The output was written to a file which could then be read by scilab. The histograms of the simulation and a scilab generation of an equivalent uniform process can be seen in Figure 3.4. 60414 samples were generated, and the resulting mean of the simulation was calculated to 15.552.

![Figure 3.4 - Uniform Process Histogram Comparison](image)

As can be seen in Figure 3.4, the data range (number of clock cycles between events) was $[0, 31]$. This is what was expected. Further more it can be seen that the histogram is very even, just like it should be.

The resulting component can be clocked at 125MHz and takes two clock cycles to initialize.
3.3.4 Poisson Process

In section 2.9.3 an algorithm for generating a Poisson process was presented. That function takes the intensity as a parameter and returns the number of time units until next event. The returned value is the same as the number of times the loop had to be iterated before the variable t was at least one. In the context where it will be used here, there is no need to generate the return value. What instead is interesting is to generate an event signal when t reaches one (or more). This way, events will be generated with a Poisson distribution.

The algorithm is fairly simple, there is only one calculation done and it is the following:

\[ t = t - \log(\text{rand()})/i \quad \text{Eq. 3.1} \]

Since "i" is a static variable that will only change when a new session is initiated, this value can be pre-computed. Furthermore, \( \log_2 \) is easier to calculate than \( \log \) (will be explained further in section 3.3.4.1). This is no problem as \( \log \) can be rewritten:

\[ \log(x) = \frac{\log_2(x)}{\log_2(e)} \quad \text{ Eq. 3.2} \]

Eq. 3.1 can be rewritten using Eq. 3.2 to:

\[ t = t - \log_2(\text{rand()}) \cdot \frac{1}{\log_2(e) \cdot i} = t - C \cdot \log_2(\text{rand()}) \quad \text{where} \quad C = \frac{1}{\log_2(e) \cdot i} \quad \text{Eq. 3.3} \]

The variable "c" can then be pre-computed. The only complicated thing that is left is computing \( \log_2 \). How this is done will be explained in section 3.3.4.1.

The width of the random number will mostly limit how big the \( \log_2 \) part of Eq. 3.3 can be. In section 3.3.4.1 it is discussed how the integer and fractional part of the \( \log_2 \) result must be shared in seventeen bits. Each additional bit that is added to the random number will make the integer part one unit larger. If 32 bits are used, then the integer part must be 5 bits (must be able to represent the integer value 31). The next step is having an integer part of 6 bits, or up to 63 in integer value. This would require a 64-bit random number. Therefore a 32-bit random number was deemed suitable, as this is in line with the 26-bit random number required by the uniform process.

3.3.4.1 \( \log_2 \) Implementation

The input to the \( \log_2 \) unit has the following range \( 0 \leq \text{din} < 1 \). Although \( \log_2(0) \) is not allowed, it will be returned as the maximum output in this implementation. The output from the \( \log_2 \) unit will thus always be a negative number. To save the cost of a sign bit, this is simply implied. When this unit is used this will have to be taken into consideration.

When calculating \( \log_2 \), the following rule is used:
In more general terms, if there are zeros between the fractional point and the first ‘1’, these can be counted and considered as the integer part of the result. The remaining log2 part to be calculated is now a number within the range \([0.5, 1)\). When log2 is calculated on this range it will return a result in the range of \([-1, 0)\). As can be seen, in one case the result will return an integer.

In practice, the leading zeros are removed after they have been counted. The remaining part will then be a fractional number starting with ‘1’ (with the one exception where input is zero). This part is then used to look up the remaining part of the result in a pre-configured look up table (LUT).

The width and the depth of this LUT will depend on various factors; the width will be investigated first. The output of the log2 unit is set to be seventeen bits (see section 3.3.4.2). These bits have to be divided into an integer and a fractional part. Since the incoming random number is 32-bits wide, the maximum number of zeros before a one is 31. This means that five bits have to be used for the integer part. If all incoming bits are zero the result will saturate to the largest representable number, which is very close to 32. This leaves twelve bits for the fractional part.

The minimum width of the LUT is then known to be thirteen, one integer plus twelve fractional. To make it as compact as possible, the LUT will have to fit into a single BRAM. Table 2.2 shows that a 1k x 18-bit table must be used to get thirteen bits width. The address to the LUT must be ten bits to address the full depth of the LUT. The content of the LUT is detailed in Table 3.4.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>(\text{abs}\left(\log_2\left(0.5+1\cdot\frac{0.5}{1024-1}\right)\right))</td>
</tr>
<tr>
<td>2</td>
<td>(\text{abs}\left(\log_2\left(0.5+2\cdot\frac{0.5}{1024-1}\right)\right))</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>1022</td>
<td>(\text{abs}\left(\log_2\left(0.5+1022\cdot\frac{0.5}{1024-1}\right)\right))</td>
</tr>
<tr>
<td>1023</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4 - log2 LUT Contents
When both the integer part and the fractional part have been obtained, these two are bit aligned and added. An extra guard bit is used for overflow detection. If an overflow is detected, the result is saturated to the maximum value. Saturation will only occur in the case of the input being all zeros.

The final implementation of the log2 unit can be seen in Figure 3.5. The registers that are used along the way are pipeline registers used to be able to clock the unit at a higher rate.

![Figure 3.5 - log2 Block Diagram](image)

### 3.3.4.1.1 Result

A simulation on the log2 unit was done with a vector with the data range \((0, 1)\). The stimuli vector and the output vector were written to a file. This file then served as an input to scilab, where the result could be analyzed. In Figure 3.6 the result of a plot of the result vectors generated both in scilab and by the log2 unit implemented can be seen.

![Figure 3.6 - Comparison Between log2 Results](image)
As it is difficult to draw any conclusions from Figure 3.6, since the results are so similar, another plot was made. This plot, Figure 3.7, shows the difference between the scilab and the VHDL implementation result. The difference was obtained by subtracting the absolute value of the scilab result from the absolute value of the simulation result. The difference seen in the picture is mostly due to quantization errors, both in the resolution of the table and of the values contained in the table.

![Figure 3.7 - Difference Between log2 Results](image)

The final component can be clocked at 125MHz and takes three clock cycles to initialize.

### 3.3.4.2 Implementation of the Poisson Process

A new session is initiated by pulsing the restart pin. The restart control block then takes care of holding notready high while the pipeline is filled to the end. When notready is held low the Poisson process is ready, the ready signal is thus simply notready inverted. To mask out any possible events happening while the device is initializing, the event signal is filtered via an and-gate with the ready signal. After a reset, ready will be held low until restart has been pulsed.

When restart is pulsed the current input on C is sampled into the C-register. The mean of the Poisson process is controlled via the C signal. Eq. 3.3 describes how to calculate C out of a specific mean.

When the addition generates a result that is greater or equal to one (when the integer part is non-zero) an event is generated and the accumulator is cleared. The block diagram of the implementation can be seen in Figure 3.8.
3.3.4.2.1 Result

The histogram of a simulated Poisson process with roughly 100,000 samples, compared to a scilab implementation can be seen in Figure 3.9. The histogram is of the clock cycle count between events. The Poisson process was configured to have a mean of 10. The resulting mean of the simulated data was 9.975 and the variance was 9.961.

The final component can be clocked at 125MHz and takes five clock cycles to initialize.
3.3.5 Statistics Component

The statistics component combines the uniform and Poisson processes in a single statistics block. The block diagram of this can be seen in Figure 3.10.

![Figure 3.10 - Statistics Component](image)

A new statistics run is initiated by setting `restart` high for at least one clock cycle. `Seed`, `devCtrl` and `devSel` are then sampled.

The `devSel` signal is used to choose between the uniform and Poisson process. The `devCtrl` signal is used to set the properties of the two processes. Care must be taken on the data alignment. The uniform process uses the lower five bits and interprets them as an integer ranging from zero to thirty one. On the other hand the Poisson process uses the seventeen most significant bits and interprets them as a seventeen bit fractional.
3.4 Agent Control and Log

As the Agent has the ability to be reconfigured and to send data logs, external control circuitry must be designed. This circuitry consists of the interface between FPGA and the PPC and the code on the PPC that controls the FPGA fabric and communicates with the PC.

3.4.1 Processor Block

The processor block component can be seen in Figure 3.11. This picture shows the system cores that are connected to the PLB and the peripherals that are connected to the OPB. This component is designed and implemented in EDK. The component as such (a normal VHDL entity) can then be instantiated in the VHDL code. The signals and the individual blocks are presented in the sections below.

![Processor Block Architecture](image)

**Figure 3.11 - Processor Block Architecture**

3.4.1.1 Reset Control

The reset control is a CoreGen component that controls the PPC reset. It has one external reset signal as its input. This signal is then used to reset the processor block from the FPGA fabric.

3.4.1.2 Memory Unit

This block contains the BRAM controller and the connections to the BRAMs. These are CoreGen blocks. They required no extra user setup except for the memory space configuration. The PPC has 128k ram allocated, which is more than enough.
3.4.1.3 **IPB Control**

This block is written by Sectra Mamea AB. It’s a bus interface to simplify the connection to the PLB bus from the FPGA fabric. The external signals of the IPB Control (IPBC) are described below.

- **clk** - Bus Clock. All the signals must be synchronized to this clock.
- **reset** - Bus Reset. Indicates a bus reset.
- **cs** - Chip Select. This signal is logic high when the block is selected. If CS is not high then all the traffic on the bus (all other signals) should be ignored.
- **RnW** - Read not Write. Indicates if it is a read or write access that is requested. High indicates a read and low a write.
- **addr** - Address. The address that is being read/written.
- **dataOut** - Data Out. Contains the data to be written on a write access.
- **tOutSupr** - Time Out Suppress. If a buss access is going to take more than the allowed six clock cycles a slave can assert this signal to suppress the time out.
- **ack** - Acknowledge. This is used to acknowledge that a transaction has been completed. If it is a read then it indicates that the appropriate data exists on dataIn. An ack is indicated with a rising edge.
- **dataIn** - Data In. Data to be read is returned with this signal.

3.4.1.4 **PLB to OPB Bridge**

The PLB to OPB bridge is another CoreGen device. It is used to connect the OPB to the PLB.

3.4.1.5 **UART Lite**

The OPB UART Lite Xilinx CoreLogic™ device is used as the system UART. This device has two external signals, uartTx and uartRx. These signals are connected to the appropriate I/O pins on the FPGA for external communication with the PC.

3.4.1.6 **Timer**

This is the Xilinx CoreLogic™ OPB Timer. It can be used for several timing purposes that work independently of the PPC.

3.4.2 **PPC to FPGA Interface**

In section 3.4.1.3 the IPBC interface was described. This is the interface that is used for communication between the main FPGA fabric and the PPC. The register map and their properties will be explained in the sections below.

3.4.2.1 **FPGA Register Map**

IPBC has a 32-bit address, but only eight bits are used. The address is partitioned according to Table 3.5.
Table 3.5 - IPBC Address Partitioning

The fields of the address are further specified below.

- **Control**: 1 bit. This single bit is used to indicate if it is a global control register that is being addressed or an individual device.
- **Device**: 3 bits. These bits are used to point out a specific device. Up to eight devices are supported.
- **Register**: 4 bits. These bits are used to point out a specific register within a device. Up to sixteen registers per device are supported.

The addressing structure is designed to be easy to extend. If additional devices are needed, it is only to extend that field. There is already some slack built in as there are only four agents (the only devices used) and addressing space for eight. Same goes for the register map.

The control bit is used for easy identification of global properties. The global structures could have been used as a device, but to make it more clear, a specific control bit was used.

The current device map can be seen in Table 3.6.

<table>
<thead>
<tr>
<th>Device</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AgentBB-1</td>
</tr>
<tr>
<td>1</td>
<td>AgentBB-2</td>
</tr>
<tr>
<td>2</td>
<td>AgentBS-1</td>
</tr>
<tr>
<td>3</td>
<td>AgentBS-2</td>
</tr>
<tr>
<td>4-7</td>
<td>Not defined.</td>
</tr>
</tbody>
</table>

Table 3.6 - Currently Defined Devices

### 3.4.2.2 Control Registers

There are two control registers. They are detailed in Table 3.7.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Status and Control</strong>&lt;br&gt;Contains global status flags and control bits.</td>
</tr>
<tr>
<td>1</td>
<td><strong>Seed</strong>&lt;br&gt;The global seed for the random generators.</td>
</tr>
</tbody>
</table>

Table 3.7 - Control Registers
Chapter 3 - Implementation

3.4.2.2.1 Status and Control Register

This register contains global status flags and control bits. In Table 3.8 the bit names and properties are detailed.

<table>
<thead>
<tr>
<th>Name</th>
<th>Unused</th>
<th>RAEG</th>
<th>CAB</th>
<th>GAE</th>
<th>Unused</th>
<th>A3BO</th>
<th>A3HD</th>
<th>A2BO</th>
<th>A2HD</th>
<th>A1BO</th>
<th>A1HD</th>
<th>A0BO</th>
<th>A0HD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit number</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27-8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Accessibility*</td>
<td>X</td>
<td>W</td>
<td>W</td>
<td>RW</td>
<td>X</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

(*) R – Read only, W – Write only, RW – Read Write, X – Not Used

Table 3.8 - Status and Control Register

The bits have following properties:

AxHD - Agent x Has Data
1 if the Agent has data in its FIFO, 0 otherwise.

AxBO - Agent x Buffer Overflow
1 if the Agent fifo has had an overflow, 0 otherwise.

GAE - Global Agent Enable
1 if the Agents are enabled, 0 otherwise.

CAB - Clear Agent Buffers
Write 1 to this bit to clear all Agent FIFOs.

RAEG - Restart Agent Error Generation
Write 1 to this bit to restart all Agent error generators.

3.4.2.2.2 Seed Register

This is the global seed register. It is 32 bits wide and can both be read and written. All other seeds are generated from this global seed by XORing it with static masks for each new seed.
3.4.2.3 **Agent Registers**

The Agent Registers are detailed in Table 3.9.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0-7 | Log FIFO Out  
The Log FIFO output, split into several registers. |
| 8 | Statistics Control  
Control data for the statistics block. |
| 9 | Agent Control  
Agent control data. |

Table 3.9 - Agent Registers

3.4.2.3.1 **Log FIFO Out**

This 32-bit register group contains one line of the Log FIFO output. The least significant register contains the least significant bits.

When register zero is read a new line in the FIFO is read out. All succeeding reads from the higher registers will thus be from the same FIFO line as register zero.

AgentBB has its Log FIFO output split into eight 32-bit registers. AgentBS has one single register.

3.4.2.3.2 **Statistics Control**

This 32-bit register contains the statistics control data that is fed into the statistics block. The register is write-only. Depending on how the statistics block is configured this data will have different properties. This is explained closer in section 3.3.5.

3.4.2.3.3 **Agent Control**

This 8-bit register contains the Agent Control bits; it is specified in section 3.2.8.1.

3.4.3 **PPC to PC Interface**

As there is a GUI on the PC, there needs to be a communication channel between the PC and the PPC. The communication channel was chosen to be in the form of a UART. This interface was selected because of its ease of use (both in hardware and on the PC) and the fact that there was hardware on AgentHW to support it.

The UART has two main purposes; to control the AgentHW from a PC and to send log data back to the PC from the AgentHW. A welcomed addition is the added debug capability, as there is now a way to read/write debug data from/to the AgentHW. The UART uses two separate channels for the two directions, each channel is only one way.
3.4.3.1 **UART configuration**
To accomplish as high a data rate as possible, the UART was configured with as little transmission security as possible. This means only one stop bit and no parity was used. The complete configuration can be seen in Table 3.10.

<table>
<thead>
<tr>
<th>Property</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmission rate</td>
<td>115200 bps</td>
</tr>
<tr>
<td>Data bits</td>
<td>8</td>
</tr>
<tr>
<td>Parity</td>
<td>None</td>
</tr>
<tr>
<td>Stop bits</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.10 - UART Configuration

3.4.3.2 **PPC to PC Protocol, Agent-FUP**
This link is mainly used to send log data to the PC. That means that all the Agent logs are sent over this relatively low speed link (115.2kbps compared to 1.25-2.5Mbps of Fibre Channel). To be able to send as much data as possible, the overhead of the protocol needs to be as small as possible.

A problem in single wire UART transmissions is that there is no separation between control and data. There are 256 different characters (since 8-bit data is used) that can be sent and there is no way to tell if it is a data or control byte. One way of solving this problem is to encode the data. This way control characters can be separated from the data. The problem with this is that the encoded data would not fit into a single byte transmission. This would generate additional overhead and that can not be accepted.

A solution to this problem is to use a synchronized transmission method where the state of the transmission indicates whether the received byte is a control character or a data byte. The method used to synchronize the PPC and PC is described in section 3.4.3.4. Note that the synchronization is on protocol level, not bit level. The actual bit transmission is still asynchronous.

To keep overhead at a minimal, a single byte header was chosen. The header contains enough information for the receiver to know what will come next. The high data rate packets (agent specific logs) always have the same structure and byte count. Thus the byte count can be encoded in the actual command. With this configuration it will only be a single byte overhead. For further information about the protocol see Appendix A.

3.4.3.3 **PC to PPC Protocol, Agent-SUP**
The GUI uses this link to send control data to the AgentHW. These types of transmissions have no special speed requirements. In section 3.4.3.2 there was a discussion about the problem with data and control separation on a single wire UART link. Since this link does not have the same speed requirements, the data and control can be separated by encoding the data.
ASCII was chosen to encode the data. This is the common way to encode data sent over UART and there is no reason to make up a new encoding scheme for it. For more information about ASCII codes see section 2.6.

The protocol structure is a simple space separation protocol. A space character separates different control words and a typical transmission looks like:

```
command arg1 arg2
```

Each transmission is ended by sending either a line feed (0x0A) or a carriage return (0x0D) or both. Further information about the protocol and its commands can be found in Appendix B.

3.4.3.4 Synchronization

Since Agent-FUP is a synchronous protocol it needs some means of synchronizing the receiver and the transmitter. This is implemented as a handshake-synchronization. The PC sends a synchronization command over the Agent-SUP. The PPC then aborts all its current activities and responds with a single acknowledge byte ('a'). If the PC receives the acknowledge correctly it will acknowledge that by sending back a "resume byte" ('r'). The PPC will then continue its normal operation. The receiver state machine will then be synchronized to the transmission state machine.

3.4.4 PPC Software

The software that is executed on the PPC is mostly acting as a communications layer between the GUI and the FPGA fabric. The code is written in C and since the memory available is limited, the code is kept as simple as possible. The flow of the program can be seen in Figure 3.12. The main tasks of the PPC are to handle incoming commands from the PC, configure the Agents and log the data generated by the Agents. These steps will be further explained in the sections below.

3.4.5 Agent Logging

The Agent logging is slightly different if it is an AgentBS or an AgentBB. But the main structure is the same. The general flow of the logging can be seen in Figure 3.13. In the AgentBB case the "Incoming Error" part would not be executed, since it doesn’t have that feature.

3.4.6 UART Handling

Since the UART data rate is so slow, there is no need for an interrupt driven UART handler. Instead the UART receive buffer is polled once per main loop. The receive buffer is read one character at a time and then added to an internal buffer. If an end of command character is received (see Appendix B), the buffer is parsed and the command is executed. The flow of the UART handler can be seen in Figure 3.14 - UART Handler.
3.4.7 Timer

Every time data is logged and there was no error generated, the PPC simply updates the statistics counters. The PPC can not transmit every packet to the PC as the UART is not fast enough. Instead the statistics counters are sent out on a constant time basis. To do this a hardware timer was used. This timer simply counts up and one time every loop the counter is read. If the counter has reached the update limit then all stats counters are sent (if they contain any data) to the PC.

![Figure 3.12 - PPC Main Loop](image1)

![Figure 3.13 - Agent Log](image2)

![Figure 3.14 - UART Handler](image3)
3.5 Agent GUI

To control the AgentHW, a GUI was written in C#. The main purpose of the Agent GUI is to provide an easy method of setting up and logging the output from the AgentHW. The main features that had to be implemented are listed below:

- Ability to setup and communicate over UART
- Synchronization of the Agent-FUP
- Agent configuration
- Statistics seed configuration/generation
- Console output
- Manual input of Agent-SUP commands
- Display packet counters
- Log output
- Ability to save the log

A screenshot of the finalized Agent GUI can be seen in Figure 3.15.

![Figure 3.15 - Agent GUI](image-url)
C# was chosen as the programming language for the GUI based on its reputation of an easy to learn, but yet advanced language. Visual Basic or normal C++ could also have been used, as there are free Express versions for these languages too. Visual Basic was rejected due to C# being "the next generation Visual Basic" whilst having a C-style syntax. The reason for choosing C# over C++ was that writing this thesis was a perfect time to learn this new C-derivate language, as the GUI is a rather simple program to write.

### 3.5.1 I/O Setup

This section of the program is used to setup and open the serial port on the local computer. There are also options to get a local echo of the commands sent and to synchronize the communication. To be able to communicate with the AgentHW the serial port must be opened first. When synchronization is in progress the Agent Settings will be disabled.

### 3.5.2 Agent Settings

This section of the program is used to setup the individual agents. The settings that can be modified for each Agent are specified in Table 3.11.

<table>
<thead>
<tr>
<th>Name</th>
<th>Settings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Log</td>
<td>On/Off</td>
<td>Enable logging of individual packets.</td>
</tr>
<tr>
<td>Error Log</td>
<td>On/Off</td>
<td>Enable logging of incoming errors. (only applicable to AgentBB devices, other devices will ignore this option)</td>
</tr>
<tr>
<td>ErrGenEn</td>
<td>On/Off</td>
<td>Enable generation of errors.</td>
</tr>
<tr>
<td>Distribution</td>
<td>Uniform/Poisson</td>
<td>Choice of error generation distribution.</td>
</tr>
<tr>
<td>Error In</td>
<td>Everywhere/Packet</td>
<td>Choice of where the errors will be generated.</td>
</tr>
<tr>
<td>Uniform Max</td>
<td>$2^x - 1, 1 \leq x \leq 32$</td>
<td>Choice of max value of the uniform process, this is a power of two.</td>
</tr>
<tr>
<td>Poisson Mean</td>
<td>$1 - (2^{16} - 1)$</td>
<td>Choice of rate for the Poisson process.</td>
</tr>
<tr>
<td>Seed</td>
<td>-</td>
<td>Choice of global seed. It can be manually typed in or automatically generated.</td>
</tr>
</tbody>
</table>

Table 3.11 - Agent GUI Settings

The logging options can be changed during a run and written separately with the “Write Log Settings” button. The error generation can be easily stopped with the “Stop Error Generation” button.
Pressing the “Start New Session” button will initialize a new error generation session. The settings will then be sent down to the AgentHW and the log and statistics in the GUI will be cleared. If there is any previous log or statistics data to be saved, the program will ask if they should be saved before erasing them. The current statistics will then be added to the end of the log file.

3.5.3 Console

This section is used to manually control the AgentHW and to see the console (system) messages sent from the PPC. Debug functions, like reading specific memory addresses (see Appendix B), can be executed manually from here. There are also buttons to clear or freeze the console log.

3.5.4 Statistics

The different counters are presented here. They can be cleared during a session with the “Clear” button. Otherwise they are cleared when a new session is initialized. The counters are updated automatically as they are received.

3.5.5 Log

The logs are presented in this view. The buttons on the bottom are for: saving the log, freezing the log (the log will not update until it is unfrozen again) and clearing the log.

The received logs are parsed and then presented in a human-understandable form. Each presented log is tagged with the name of the unit that it was logged from. The packet counter changes since the last generated error is appended at the end of the log. An example of a log is shown here below:

```
----------------------------------------------
IPB->PC (238) | Type   - 0x888E (Channel)
| Channel- IPB_CHANNEL
| Flags  - 0xC0
| First  - Yes
| Id     - 107
| Length - 44
| Timer  - 11us
| ErrGen - DATA
| Stats  - ACK-2980, SYS-0, DU-2980, SCS-0, OVF-0, ERR-0
----------------------------------------------
```
Chapter 4 - Conclusions

4.1 Result
As all the goals were met, the result of this thesis was successful. The tool has been tested against the hardware it is supposed to work with and it performs as expected.

4.1.1 AgentHW
All the hardware parts that were written in VHDL were simulated and verified to the extent that was possible. Not all parts can be simulated, as some of it requires licenses that are not available. Those parts were the RocketIO transceivers and the processor blocks. These parts were partly simulated by generating simulation entities that supplied test data. The simulation and verification of the statistical parts has been covered in section 3.3.

The state machines that keep track of the current protocol state have been verified against actual test runs on the systems it will work against. These have been successful and the logging is correct.

The logic design fit within the FPGA resources with margin, only 27% of the logic and 70% of the BRAM resources were used. Some minor pipelining had to be done to pass the timing requirements.

The code for the PPC fit into the 128k BRAM that is allocated for it. This memory resource could possibly be brought down as low as 32k (from 64 to 16 BRAMs) if the BRAM resources were needed elsewhere.

4.1.2 Agent-GUI
The Agent-GUI fills the requirements set on it. It has no problem keeping up with the incoming UART data on the PCs that it has been tested on. The program requires a resolution of at least 1024x768 to be displayed properly, but this is no problem on modern monitors.

4.2 Future Improvements
A faster interface between the PC and the AgentHW would be good. With this, all packet headers could have been logged. It would then be possible to see exactly how the system reacts to the generated errors. This could help in finding out why the system failed, not just that it failed when an error was inserted into a certain position. The faster link could in this case be a Gigabit Ethernet link, as there is already hardware in place to support it.
The uniform process could be modified so that it is not limited to means of the power of two. The modulo operator could easily be implemented as a fractional multiplication if the uniform random number is seen as a 32-bit fractional number and the scaling variable is seen as an integer. The result of the multiplication is then simply truncated at the fractional bit, and the scaled result is thus obtained. Multiplication is virtually free since there are many multiplication blocks in the FPGA. This was realized too late, but is an easy change to do on a future update of the design.

The random number could be improved by using a wider Leap-forward LFSR internally and only outputting 32-bits. As it is implemented now, it has to tick through the whole LFSR sequence before the same number is repeated. The impact of this is not crucial in this implementation though, so it is good enough as it is.

It would also have been good to extend its uses and have it serve as a bit error analyzer, to analyze the bit error rate (BER) of a certain link. This would be helpful in debugging purposes if there are suspicions that certain hardware is generating too many errors, and which hardware it is that is generating it.

Some improvements on the AgentGUI that has been considered are graphical statistics and the possibility to save the current configuration. These improvements are not critical though, so they were not prioritized.

Incorporating the addressing control/bus interface directly into each Agent would make the system easier to expand. Agent components could then be added directly to the processor bus. The global status register would then have to be maintained in the PPC code and would then have to be distributed out to each Agent.
References


Appendix A - Agent-FUP

This section contains a description of the Agent-FUP (Fast UART Protocol) and its defined commands.

The first byte sent is used as a header. The header is split into two nibbles where the lower nibble is the command and the higher is the command specifier, as seen in Table A.1.

<table>
<thead>
<tr>
<th>Bit</th>
<th>7-4</th>
<th>3-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Command Specifier</td>
<td>Command</td>
</tr>
</tbody>
</table>

Table A.1 - Agent-FUP Header

The command specifier is usually used to specify which of the Agents that the command applies to. All defined commands and their descriptions can be seen in Table A.2.

<table>
<thead>
<tr>
<th>Command</th>
<th>Name</th>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Not Defined</td>
</tr>
<tr>
<td>1</td>
<td>ASCII Console</td>
<td>-</td>
<td>The data sent is interpreted as ASCII characters and will be output to the console view. The command is terminated with an ETX (End Of Text, 0x03) character.</td>
</tr>
<tr>
<td>2</td>
<td>ASCII Log</td>
<td>-</td>
<td>Same as command 1, but the text is sent to the log view.</td>
</tr>
<tr>
<td>3</td>
<td>ASCII Broadcast</td>
<td>-</td>
<td>Same as command 1, but the text is sent to both the console and the log view.</td>
</tr>
<tr>
<td>4</td>
<td>Overflow</td>
<td>0</td>
<td>Indicates that an Agent has had a buffer overflow, no additional data is sent. The command specifier is used to specify which Agent it applies to.</td>
</tr>
<tr>
<td>5</td>
<td>AgentBB Packet</td>
<td>32</td>
<td>The data sent is a raw packet log consisting of the whole FIFO output.</td>
</tr>
<tr>
<td>6</td>
<td>AgentBB Compact Packet</td>
<td>8</td>
<td>The data sent is a compressed version of the FIFO output.</td>
</tr>
<tr>
<td>Command</td>
<td>Name</td>
<td>Bytes</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7</td>
<td>AgentBB Error Generated</td>
<td>0</td>
<td>Indicates that an error has been generated. The command specifier is used to specify which Agent it applies to.</td>
</tr>
<tr>
<td>8</td>
<td>AgentBB Error</td>
<td>32</td>
<td>Indicates that an error has been detected on incoming data. The data consists of error log specific data.</td>
</tr>
<tr>
<td>9</td>
<td>AgentBB Statistics</td>
<td>8</td>
<td>The data sent consists of the four different 2-byte packet type counters. They are sent least significant byte first and they are sent in the following order: ACK, SYS, DU/IPB, SCS</td>
</tr>
<tr>
<td>10</td>
<td>AgentBS Control Packet</td>
<td>2</td>
<td>The data sent is a BS control packet log.</td>
</tr>
<tr>
<td>11</td>
<td>AgentBS Data Packet</td>
<td>2</td>
<td>The data sent is a BS data packet log.</td>
</tr>
<tr>
<td>12</td>
<td>AgentBS Statistics</td>
<td>4</td>
<td>The data sent consists of the two different 2-byte packet type counters. They are sent least significant byte first and they are sent in the following order: DATA,CONTROL</td>
</tr>
<tr>
<td>13</td>
<td>AgentBS Idle Error Generated</td>
<td>0</td>
<td>Indicates that an error was generated when transmission was in IDLE.</td>
</tr>
<tr>
<td>14</td>
<td>DataDump</td>
<td>-</td>
<td>This is a command to send general data dumps. The command is followed by a two byte counter (most significant byte first) that indicates how many data bytes will follow.</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>Not Defined</td>
</tr>
</tbody>
</table>

Table A.2 - Defined Agent-FUP Commands
Appendix B - Agent-SUP

This section contains a description of the Agent-SUP (Slow UART Protocol) and its defined commands.

The commands are sent and received over a UART link. All the commands sent are encoded in ASCII. The protocol is a simple space separated command interface. A command is terminated with a line feed (h0A) or a carriage return (h0D) or both. A general command structure looks like this:

```
command arg1 arg2
```

There are some built-in limitations as to how many arguments that can be passed with a single argument and how long the total command line can be. These limitations are due to predefined buffers. These buffers can be easily extended if needed. The current limitations are shown in Table B.1.

<table>
<thead>
<tr>
<th>Property</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max argument count (*)</td>
<td>8</td>
</tr>
<tr>
<td>Max argument length (**)</td>
<td>64</td>
</tr>
<tr>
<td>Max total command length (***)</td>
<td>1024</td>
</tr>
</tbody>
</table>

(*) command + arguments
(/**) Length of individual argument (bytes)
(*** ) Length of the whole command with all the arguments (bytes)

Table B.1 - Agent-SUP Command Limitations

All defined commands can be seen in Table B.2. All numbers sent are interpreted as decimal.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Synchronization of Agent-FUP</td>
<td>-</td>
</tr>
<tr>
<td>wr</td>
<td>Write to register</td>
<td>arg1 – Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>arg2 – Data</td>
</tr>
<tr>
<td>rr</td>
<td>Read from register</td>
<td>arg1 – Address</td>
</tr>
<tr>
<td>cf</td>
<td>Clear all Agent FIFOs</td>
<td>-</td>
</tr>
<tr>
<td>ge</td>
<td>Global Agent enable</td>
<td>-</td>
</tr>
<tr>
<td>gd</td>
<td>Global Agent disable</td>
<td>-</td>
</tr>
<tr>
<td>gdc</td>
<td>Global Agent disable and clear FIFOs</td>
<td>-</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Arguments</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>eel</td>
<td>Enable error logging on all Agents</td>
<td>-</td>
</tr>
<tr>
<td>del</td>
<td>Disable error logging on all Agents</td>
<td>-</td>
</tr>
<tr>
<td>epl</td>
<td>Enable packet logging on all Agents</td>
<td>-</td>
</tr>
<tr>
<td>dpl</td>
<td>Disable packet logging on all Agents</td>
<td>-</td>
</tr>
<tr>
<td>pcr</td>
<td>Print control register</td>
<td>-</td>
</tr>
<tr>
<td>was</td>
<td>Write Agent log settings</td>
<td>arg1 – The log part of the Agent control register. The four control registers are concatenated together. The log part is masked out and written to each Agent.</td>
</tr>
<tr>
<td>sns</td>
<td>Start new session</td>
<td>arg1 – seed&lt;br&gt;arg2 – The stats part of the Agent control register. The four control registers are concatenated together. The stats part is masked out and written to each Agent.&lt;br&gt;arg3 – Agent0 statistics control register&lt;br&gt;arg4 – Agent1 statistics control register&lt;br&gt;arg5 – Agent2 statistics control register&lt;br&gt;arg6 – Agent3 statistics control register</td>
</tr>
<tr>
<td>seg</td>
<td>Stop all error generation</td>
<td>-</td>
</tr>
<tr>
<td>gs</td>
<td>Get all device stats</td>
<td>-</td>
</tr>
<tr>
<td>db</td>
<td>Debug function</td>
<td>Variable, depending on how it is currently defined</td>
</tr>
<tr>
<td>h</td>
<td>Help (display all available commands)</td>
<td>-</td>
</tr>
</tbody>
</table>

Table B.2 - Agent-SUP Commands
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/