FPGA Implementation of a Pattern Generator

Jakob Toft, s012012

Technical University of Denmark
Kongens Lyngby 2007

IMM-B.ENG-2007-48

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

DTU, IMM
B. Eng Project

Written by:
Jakob Andreas Toft(s012012)

Summary

The target of this project is to create a system that generates signal patterns based on user input. The system is intended to be used in test and verification of digital circuits. A *Logic State Analyzer* is used to analyze outputs based on specific input of the circuit being tested.

The *Pattern Generator* is implemented on a Spartan II FPGA in two different designs: "*Memory-based*" and "*buffer-based*".

Two different interfaces are used between PC and FPGA, more specific a serial and parallel port interface. The system can easily be expanded to faster connections due to the modular structure.

User input is based on 32 bit input vectors. This gives a very high flexibility for the user, since every signal can be specified for each clock period.

Test frequencies of up to 50 MHz are being investigated, and the issues introduced by SSO and "ground bounce" is demonstrated.

A simple 4 bit adder implemented on a secondary FPGA has been tested with the system to demonstrate its usability.

DTU, IMM
B. Eng Project

Written by:
Jakob Andreas Toft(s012012)

DTU, IMM
B. Eng Project

Written by:
Jakob Andreas Toft(s012012)

Resumé

Målet med dette projekt er et system der kan generere digitale test mønstre baseret på bruger input. Systemet forestilles at blive brugt med henblik på test og verifikation af digitale kredsløb. En *Logic State Analyzer* benyttes til at analysere outputs i henhold til bestemte inputs af kredsløbet der testes.

Signal generatoren er implementeret med to forskellige designs i en Spartan II FPGA, henholdsvis *"hukommelses baseret"* og *"buffer baseret"*.

To forskellige interfaces er benyttet mellem PC og FPGA, henholdsvis seriel og parallel port. Systemet har dog rig mulighed for at implementere andre interfaces pga. det modulære design.

Bruger input er specificeret som 32 bit input vektorer. Dette giver en stor fleksibilitet da brugeren kan specificere hvert enkelt signal specifikt for hver enkelt klok periode.

Frekvenser op til 50 MHz er blevet undersøgt og eventuelle støj problemer forårsaget af SSO og "ground bounce" er blevet demonstreret.

En simpel 4 bit adder, implementeret på en sekundær FPGA, er blevet testet med systemet for demonstration dets anvendelse.


DTU

DTU, IMM
B. Eng Project

Written by:
Jakob Andreas Toft(s012012)

Preface

This report is the result of my final project as a Bachelor of Engineering from the Institute for Informatics and Mathematical Modelling at the Technical University of Denmark.

The project was proposed by Associate Professor Alberto Nannarelli of IMM who also made sure I got a desk in the laboratory, along with the needed hardware and software, to carry out the task.

I would like to thank Alberto for giving me the opportunity to work on this project and for being helpful whenever I needed to discuss specific topics.

I would also like to thank Matthias Bo Stuart for introducing me to the LA-500 Logic Analyzer and supplying me with various data sheets for the FPGA.

Last but not least I would like to thank Søren Toft for many stimulating discussions.

Jakob Toft

October 1, 2007


DTU

Table of Contents

Figures and Tables

DTU, IMM
B. Eng Project

Written by:
Jakob Andreas Toft(s012012)

CHAPTER 1

Introduction

The project

This project will explore various solutions to real world testing of FPGA designs. When a hardware designer wants to test his FPGA design in the real world, he will need to be able to control the input of the hardware while monitoring the output. To control the input of the *Design Under Test* the hardware engineer will need a device capable of generating electrical patterns, this device is usually referred to as a *Pattern Generator*.

The following figure gives an overview of a fictive test setup using the Pattern Generator presented in this project:

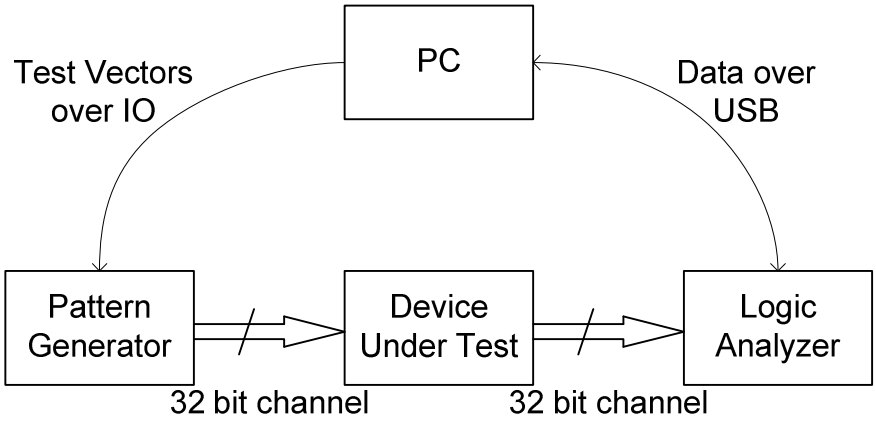


Figure 1: A fictive test setup.

The PC is the responsible for delivering the human interaction, giving the engineer one point where he can control the testing. The IO of the DUT is at a most 32 bit on each channel giving a maximum of 4,294,967,296 possible combinations for one 32bit input vector.

It is the intent of the project to be able to test digital hardware implemented on FPGAs by connecting the output pins of the PG to the input pins of the DUT with a 32 wire cable.

The task of this project includes writing the specification of the project as well as presenting a feasible solution. The project specification can be found in table 1 located in the analysis chapter.

Method

The development method used throughout this project has been an iterative process focused on four key points:

- Analysis
- Design
- Implementation
- Test

Being an iterative process each point has been revised several times, addressing the issues that were identified throughout the project. The project is focused on developing a prototype giving a solid foundation to work from when improving the design.

The analysis will cover the theoretic part of the project, it will explain the concepts used in the design and implementation of the project and hopefully give an understanding of the considerations that should be taken into account before making a pattern generator for FPGA's.

The design phase will cover the design considerations of the system. It will outline the structure of the software and hardware of the design in conceptual terms, and describe the components chosen for completing the task. The design chapter will also include a prototype of the graphical user interface visualizing how the user will operate the product.

The implementation phase will cover the most important aspects of the actual implementation. The software will be described by class diagrams and as well as code examples showing the key points of the implementation along with the implemented GUI. The hardware is described by diagrams showing the components used.

The test phase consists of individual testing of components as well as functional tests of hardware and/or software working alone or together. The tests are done as simulations as well as real world testing using a logic analyzer to capture the actual signals of the IO pins.

After completing these four key points the test and conclusion will summarize the results and reflect on the early design decisions that made the premises of the project. The conclusion will also include a comparison of the different improvements done in the implementation phase and its observed effect in the testing phase. The conclusion is divided into a project conclusion and a product conclusion, each concluding on the task from different perspectives.

CHAPTER 2

Analysis

Specification

The project specification handed in for approval at DTU was as follows:

Logic state analyzers (LSA) are used to help debug circuits by analyzing the output of the hardware and compare it with the input. To help automate this, it is desired to make a Pattern Generator (PG) that enables a computer to run Test Benches (TB) from a software environment directly to a piece of hardware. The LSA are connected to the output of the unit under test (UUT). The target of this project is a hardware unit, which generates signal patterns according to the test bench being chosen on the computer, along with software to parameterize the test bench. More specifically, the following task are performed: 1. Design a PC program with a GUI - give the user the ability to choose between different test benches, specify the clock frequency of the signals and reset the test. 2. Design the interface PC/PG. 3. Design the hardware implementing the PG. 4. Test the system.

Table 1: The project specification approved by DTU.

The specification was made together with – and approved by Alberto Nannarelli at IMM before handed in to the DTU administration for approval.

The project has mostly been about the Spartan 2 FPGA from Xilinx provided from DTU; however, a Spartan 3A of my own has also been used when working from home.

Priorities

The proposed solution of this project is mainly focused on the hardware entity used to generate the signals for the DUT and in much less regard on the actual test patterns being used. The user will have to manually create test patterns on the PC which can be loaded onto the Pattern Generator for execution. This offers total flexibility when defining the test inputs, but limits the size of the test to the memory capacity of the Pattern Generator.

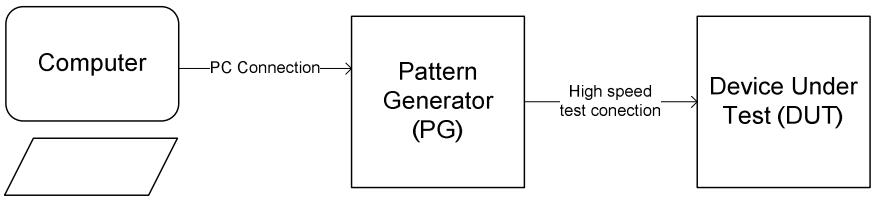


Figure 2: Overall system diagram.

As can be seen from figure 2, the user should be capable of controlling the inputs to the DUT from the PC by connecting them through the PG.

Requirements

The target of this project is to create a system that generates signal patterns based on user input. The system is intended to be used with a *Logic State Analyzer* to debug circuits by analyzing outputs based on specified input. To help automate this, it is desired to make a *Pattern Generator* that enables a computer to run tests from a software environment directly to a piece of hardware.

To facilitate this behavior a hardware unit is needed with the capability of generating signal patterns based on a testbench and a software program which lets the user parameterize the testbench. The system should be able to handle as much data as possible at the highest possible frequency. Having 32 bit input vectors equates ~4 billion different combinations of test vectors. This universe of possibilities cannot be handled in the same test due to several technical reasons as described in the following chapters.

Since the *UUT* might operate at various frequencies, it is desired that the connection between *PG* and *UUT* supports different frequencies controlled from the PC.

The following requirements have been identified:

Functional requirements		
Type	ID	Requirement
Software	1	*Graphical User Interface* (GUI).
	2	Ability to define signal patterns.
	3	Ability to define testing frequency.
	4	Ability to reset everything.
	5	Ability to control the running of the test.
Hardware	7	PG Controller capable of handling the input from the software.
	8	Connection to *UUT*.
	9	PC connection interface.
Non-functional requirements		
Software	10	Dynamically load of connection DLL's.
Hardware		Modular design making new connection types easy to implement.

Table 2: Identified project requirements.

Since the user of the software is deciding on the test patterns, the main requirements are with the interfaces between PC, PG and DUT.

Problem analysis

The PC program has been made with a core containing the application entry point, *GUI* etc. and some peripheral classes handling the data and I/O. The program should give the user an easy and fast way of defining test data for the system, either by letting him write the instructions manually or giving him/her some kind of WYSIWYG UI.

There are two ways to approach the design requirements with regards to the hardware; each has its own strong and weak points.

The first and most obvious way of implementing this design would be to download the test vectors to a memory connected to the FPGA before starting the test operation. This implementation will be referred to as the *'Memory'* implementation. When using a memory the testing frequency is only limited by the frequency of the FPGA clock and not on the speed of the connection to the PC. However, the amount of test data is limited by the amount of memory available to the FPGA e.g. 56K.

The second possibility is to have the test run 'on the fly' while the data is sent to the FPGA. This can be achieved by using a FIFO buffer instead of a regular memory. This implementation will be referred to as the *'Buffer'* implementation. When using a buffer scheme, the amount of test vectors is no longer limited in any way; however the testing frequency cannot surpass the frequency of the data coming from the PC unless we want to introduce burst data transfers. A *'Buffer'* implementation would benefit from the use of flow control ensuring that buffer overflows doesn't happen by negotiating the highest possible input and output frequency in regard to hardware capabilities.

Implementation type	**Buffer based**	**Memory based**
Test Frequency Limit	PC Connection Speed	None
Test Vector Limit	None	FPGA Memory Size

Table 3: Limitation of the different implementation types.

Choosing the 'right' type of implementation will be up to the end user, when designing for FPGA's this isn't as big a problem as it sounds, due to their reconfigurable nature. The main difference between the two types of implementation is the need for instructions when using the *Memory* Implementation.

Problem Solving

The software can work in various ways; I have decided to implement it as a simple notepad application with a send button and a dropdown menu giving the user the ability to choose between the connections available to the program. The user can then write his test bench as a small program using hex values. The editor should support line comments marked with a '#' sign and be able to recolor them for ease to read.

As a starting point the hardware connects to the PC through a serial interface. The reason for choosing serial over *parallel*, *PCIe*, *Ethernet* or *USB* is simplicity; the design should be made in a way that new connection types should be possible to add without too much hassle. The onboard memory should be the *SRAM* of the *FPGA* which is supplied with the Spartan 2.

After completing the base design, a chapter will describe the optimizations done to the design to achieve slightly different or better results. The chapter will outline a better controller component, the possibility of using a faster PC connection by implementing a parallel port interface and introducing a whole new structure for the PG that will be referred to as the *'Buffer'* implementation. The *'Buffer'* implementation is focused on using a buffer rather than a normal memory to store the test vectors, having the actual test run 'on the fly'.

Testing and verification in general

When testing and verifying the operation of a digital circuit, two types of testing is usually employed. The first is simulating the behavior of the hardware model, most likely described in some sort of hardware description language such as VHDL code. However there are many things that a simulator cannot estimate properly in a reasonable amount of time, therefore we also employ 'real world testing'. To test a design physically it has to be implemented in physical hardware, and verified by looking at the input and outputs of the hardware unit itself. This paper is focused on FPGA implementations. In simple and slow FPGA designs a designer can use switches and LED's to test the design but once the complexity and frequency of the design increases this method becomes unrealistic.

In order to test designs effectively we want to automate tests working at high speed using complicated input and output vectors. To do that, we need measuring equipment usually referred to as *Logic Analyzers* to capture the outputs of the circuit. To generate input vectors for the test we will also need a *Pattern Generator* connected to the input pins, generating the electrical impulses. Further information of testing in general can be found in (1) chapter 9.

Logic Analyzers

The *Logic Analyzer* captures data from its probes. The LA can either validate the data 'on the fly' by comparing the previous result to the next, or capture the signals to a memory for further inspection by the engineer operating the instruments when the test is done. Naturally the LA will need a lot of memory to be able to store 32 bit signals sampled at say 500 MHz, limiting the runtime of the LA. When validating 'on the fly' the LA can theoretically run for unlimited amount of time.

The LA either samples the data at a specified frequency or whenever activity is detected on the data lines. To start sampling the LA needs one or more triggering rules such as a specific bitstream or maybe just the rising edge of a specific signal. Once the trigger is detected the LA will either sample once, or until the memory is full. After sampling the data it can be viewed in various ways to determine if the *Device Under Test* is functioning as it should.

Pattern Generators

A *Pattern Generator* generates signals to be used for testing digital circuits. These tests can help ensuring correct behavior as well as determine the cause of failure of the circuit under the specified conditions. The patterns used by the generator can either be chosen by the engineer having a specific problem in mind, or be generated automatically from an algorithm.

Creating algorithms for automatic test pattern generation is a study in itself, and there is no real definitive answer to what algorithm to use. Sequential circuits are in general harder to control than the combinatorial circuits and require more signals than the combinational circuit. Hence sequential circuits often use partial scan algorithms while the combinatorial circuits can employ a full-scan scheme.

Alternative ways of testing circuits

Instead of creating the test vectors manually and storing them on the PG, an automatic test pattern generator algorithm could be used on the PG hardware itself. When designing such a scheme the FPGA would have a softcore microprocessor being programmed from the PC. This would have loosened the coupling between the pulse generator and the PC; however the user would also lose direct control over the test being run.

A commercial pattern generator – The Agilent 81134A

The Agilent 81134A Pulse pattern generator is a commercial product from Agilent giving the designer the possibility of generating various pulse patterns. These patterns can be used to highlight design flaws or fault tolerance of electronic circuits by emulating complex real world signal anomalies.

Figure 3: Agilent 81134 pulse pattern generator. (2)

Pattern generators are often very expensive and huge amount of effort are being put into making sure that the signals being generated are as accurate and noise free as possible. The 81134 has a base price of 63,551$ and is capable of generating patterns such as:

- Square waves – clock signal with fixed width.

- Pulse – with selectable width or duty cycle.
- Bursts – followed by zero data, can be repeated.
- Data – selectable pulse width.
- Pseudo random binary sequences.

Distortion, delay and jitter can be added to the signals to emulate real world issues and artificially close the 'eye' for eye diagrams. The pulse periods can range from 15 MHz to 3.35 GHz. For more information see (2). I will test my system up against this impressive (and expensive) pulse generator to demonstrate functional and technical similarities and differences (see the test section).

IO standards

Choosing the right type of connection between different parts of the design is a hard choice. One implementation might seem better than another, but as with many other things simplicity is often very welcome when starting up a project.

RS232 Serial Interface

The *RS-232 serial interface* is a rather old and slow standard compared to modern connection types, however it is very simple to implement.

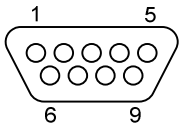


Figure 4: The DB9 connector pins.

A serial connection sends 1 bit at a time, starting with the least significant bit and ending with the most significant bit. Each byte transmitted has a start and stop bit sending the line back to the idle state. The idle state is defined as the line being constantly high, the start bit is defined as the transition from high to low and the stop bit is defined as the line stabilizing at high.

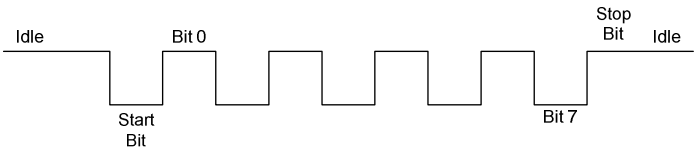


Figure 5: Sending the value 0x55 through the serial port.

Since there is no way of knowing when the next bit is ready to be read at the receiver, the sender and receiver has to agree on a transfer rate that the receiver can synchronize itself to. Some of the most common values are:

- 1200 Baud.
- 9600 Baud.
- 19200 Baud.
- 38400 Baud.

- 115200 Baud.

The term Baud relates to bits sent per second.

For more information see (3).

The parallel port

The parallel port comes in different implementations, for this project *EPP* has been chosen since it offers both speed and simplicity and is supported by the Spartan 2. *EPP* is specified in IEEE 1284 along with *SPP* and *ECP*.

The main advantages of *EPP* are:

- Bi-directional communication lines.
- 8 bit wide data signal rather than the 1 bit wide RS232 serial connection.
- Atomic transactions handled by a handshake protocol, always initiated by the host.

EPP can do both data and address reads and writes, making a total of 4 different types of transactions. There is no difference between the address and data transactions to the host. It is simply 2 different control signals and can be used for anything.

The parallel port uses a DB25 connector on each end.

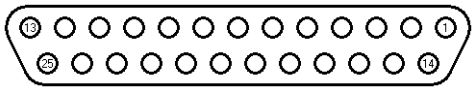


Figure 6: The DB25 connector pins. Picture taken from (4).

For EPP mode the pins are named as follows:

Pin	EPP signal name	Direction from PC	Description
1	Write	Out	Low indicates a write, high indicates a read.
2-9	Data 0-7	In-Out	Data bus lines. Bi-directional.
10	Interrupt	In	Can be used to signal interrupts back to the host.
11	Wait	In	Handshake signal, a cycle can be started when low and finished when high.
12-13	Spare	In	Not used in EPP mode.
14	Data Strobe	Out	Low indicates data transfer.
15	Spare	In	Not used in EPP mode.
16	Reset	Out	Reset signal, active low.
17	Address Strobe	Out	Low indicates address transfer.
18-25	Ground	GND	Ground pins, for increased signal integrity.

Table 4: The EPP connector pins.

The pins 12, 13 and 15 shown in Table 4 are only used in *SPP* mode and can be used as additional control lines in *EPP* mode; unfortunately they are all inputs to the host, limiting their use. To read pin 12, 13 and 15 the software will have to check the *SPP* status register at base + 1.

The EPP handshake

When doing a transaction the handshake consists of 3 signals:

- Wait
- Data Strobe
- Address Strobe

The address and data strobes are used to signal a data or address operation and are therefore mutually exclusive; we refer to them both as the Strobe signal.

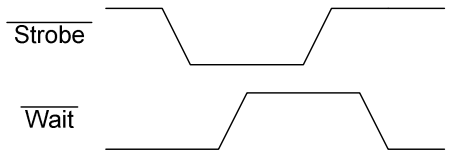


Figure 7: The EPP handshake.

Explanation of the transaction:

1. The host wants to start a transaction so it pulls one of the strobes low. If the intended transaction is a write, the write signal is also pulled low and the host starts driving the bus. If the intended transaction is a read, the write signal is driven high and the data bus is left floating.
2. The peripheral detects an asserted strobe and acknowledges by pulling wait high. If the intended transaction is a read it starts driving the data bus.
3. The host then waits for the wait signal to go high as acknowledgement before it de-asserts the Strobe signal. If the intended transaction was a write it stops driving the data bus.
4. The peripheral detects that the strobe is de-asserted and pulls wait low again. If the intended transaction was a read it stops driving the data bus.

The transfer rate of the *EPP* port usually lies around 0.5 – 2.0 MB/s.

EPP from a software standpoint

When using *EPP* mode the programmer has to take special care making sure the port is actually configured to work in *EPP* mode. Since the *EPP* port is memory mapped, 5 new registers are added to the existing *SPP* registers.

Address	Mode	Function
Base + 0	SPP	Data Port
Base + 1	SPP	Status Port
Base + 2	SPP	Control Port
Base + 3	EPP	Address Port
Base + 4	EPP	Data Port

Base + 5	Undefined (16/32bit Transfers)	-
Base + 6	Undefined (32bit Transfers)	-
Base + 7	Undefined (32bit Transfers)	-

Table 5: IO registers when running in EPP mode.

The EPP mode keeps the original SPP registers and they can be used exactly as before making EPP backward compatible with SPP.

To use the *EPP Data Port* the user has to write one byte at a time to the register at Base + 4. Unfortunately Windows XP doesn't allow user space programs to access the IO registers so a driver running in kernel mode must be used.

If the host is running in *ECP* mode, *EPP* compatibility mode can be enabled at the *Extended Control Register* located at Base + 0x402, doing so will open up the EPP Address and Data registers described in Table 5. For more information on the parallel port see (5), (6), (7), (8) and (4).

The FPGA

Logic analyzers and patterns generators benefit a lot from the reconfigurable nature of the *FPGA*. The *FPGA* is very resourceful when implementing algorithms exploiting parallelism due to its inherent structure of parallel logic resources and can compute notable throughput even at low clock frequencies at for example 50 MHz.

An *FPGA* is essentially a structure of logic blocks called *CLB*'s connected with a network of routing channels.

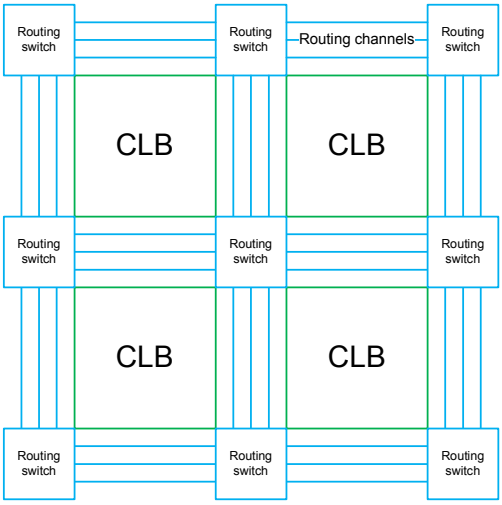


Figure 8: Simplified structure of CLB's and routing channels of an FPGA.

Clock signals and other high fanout signals are routed via dedicated networks offering better signal integrity. The behavior of the routing switch is configured from RAM to connect the CLB's wherever needed.

The *CLB* contains a *look up table* with all the possible outputs of the *CLB* and a *flip-flop* giving the possibility of registered or unregistered outputs.

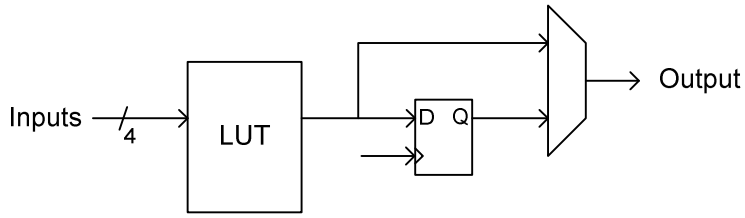


Figure 9: Structure of the 4-input CLB.

The LUT is defined by a piece of memory that can be reprogrammed, hence altering the logic behavior of the hardware. Below is shown the 4-input CLB look-up table of the logic operation A and B.

Address (A,B,C,D)	Content (A•B)
0000	0
0001	0
0010	0
0011	0
0100	0
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

Table 6: look-up table for the function A and B.

Obviously the best way of utilizing the hardware would be to keep the logic at multiples of 4 inputs since the only alternative is to split it into 2 nested CLB's increasing the size dramatically and adding more delay, impacting the setup time.

By programming the LUT, flip-flop and routing switches, the behavior of the CLB can be changed and hence the logic of the FPGA can be modified.

For more information on programmable logic, see chapter 8 of (1).

CHAPTER 3

Design

The solution proposed in this chapter is meant as a basic conceptual design, focused on creating the functionality described in the analysis.

Software

Revisiting the software requirements from table 2 in the analysis, we want to construct a GUI capable of letting the operator control the testing.

Prototype of graphical user interface

First we need a software prototype. To create a prototype the key points of the application is visualized and sketch is drawn.

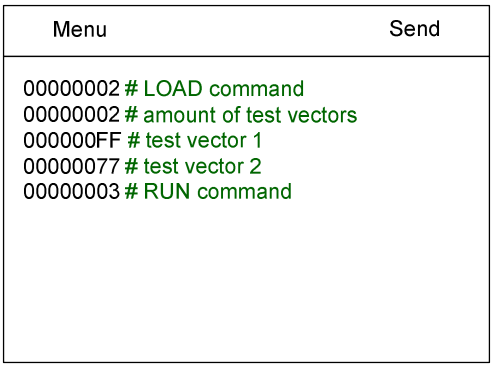


Figure 10: The visualized prototype.

The application should give the user a big text field where the testbench can be written and a few utilities such as copy/paste. We also want a send button to send the data to the FPGA and a menu where the user can choose a connection type. The commands for the editor are entered as hexadecimal values in the format

Syntax : <load command> <amount of test vectors> <test vectors> <run command>

Example 1: 00000002 00000002 000000FF 00000077 00000003

The example loads 2 bytes with the values 255 and 119 into the memory of the hardware and then run the test. For all the possible commands see table 10.

Alternatively the user can use the keywords LOAD, RUN, STOP, CLEAR to control the test, this way the before mentioned test string would look like

Example 2: LOAD 00000002 000000FF 00000077 RUN

The program should support syntax highlighting of the keywords along with comments marked by the '#' sign.

Overall design of the application

The application is built up as an n-Tier application where each tier can only communicate with the tier above and below. For information on the n-Tier model see (9).

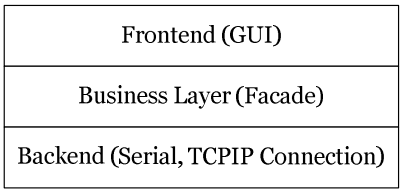


Figure 11: The controller program is built as a 3-layer application.

The Frontend consists of the application entry point along with the GUI, it communicates only with the Business Layer.

Business Layer

The Business Layer works as a facade such that the Frontend doesn't need to know anything about the backend. It implements the connection types available (different Backend's) through Reflection and provides the names of those connections to the GUI. The Business Layer acts as a middleman between the Frontend and the Backend.

Backend

The backend is the actual connections; to ensure interoperability with the Business Layer each connection adheres to a specified interface. If more connection types are needed, such as a USB connection, it can be made as a new DLL. As long as they implement the Connection Interface they will automatically get loaded by the Business Layer and presented to the user by the Frontend.

Design Patterns

List of design patterns used in the application and their use.

Facade Pattern

The facade class is responsible for communicating with the connection DLL's that the application is using.

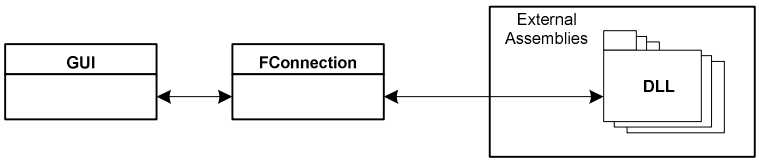


Figure 12: Calling assemblies.

When data has to be sent through a connection DLL, the caller just use the send method that the facade is providing such that the calling class doesn't need to know anything about how the connection is handled. In this way the responsibility of assemblies outside the application is reduced to a single class which reduces the complexity of the program structure.

Singleton Pattern

The Singleton pattern is responsible for making sure only one instance of a class is allowed which then can be used by all calling applications. If more than one application is accessing the same IO, errors are bound to happen. This is why it is desired to limit the access to a single point in the code and a single class, this goes well with the facade.

The Hardware Design

The hardware is implemented on an FPGA; it consists of a receiver, a controller, a clock divider and a memory element to store or buffer the test vectors.

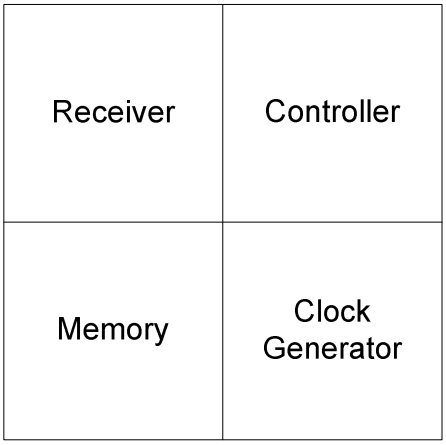


Figure 13: The intended design for the FPGA.

The FPGA

The proposed design has been implemented on a Xilinx Spartan II FPGA board from Digilent Inc.

Device	Logic Cells	System Gates (Logic and RAM)	CLB Array (R x C)	CLBs	Maximum Available User IO	Distributed RAM Bits	Block RAM Bits
XC2S200	5292	200000	28 x 42	1176	284	75264	56K

Table 7: Spartan II FPGA package information taken from the Spartan 2 data sheet (10).

Following data has been taken from the Digilab 2 development board reference manual (11):

- A 50MHz oscillator.
- A status LED.
- A push button.
- A 5-wire RS-232 serial port.
- An EPP capable parallel port. A switch toggles the behavior as either JTAG based programming or user data.
- Six 40 pin expansion connectors on the edges of the board.

The D2 board is a low cost development platform supporting only the essentials of the Spartan II FPGA. The expansion connectors ensures that the designer can get whatever peripherals he or she desires, such as the Digilab DIO2 expansion board offering a 4 segment display, 15 LEDs, 15 push buttons, 8 switches, PS2, VGA and a LCD display.

The instructions

The hardware instructions to the PG are described in the following section. There is a slight difference between the 8 bit base design and the 32 bit improved design.

The base design system takes 8 bit words as input, while in instruction mode the first 3 bits of the word describes the function and the rest defines the arguments of the system referred to as iData. In data mode the system uses 32 bit words meaning 4 bytes.

7:3	2:0
iData	Instruction

Table 8: The 8 bit data word while in instruction mode.

The implementation will be able to handle 1024 test vectors of 32 bit giving a total of 32K of data. Since the test vectors are downloaded to the PG before running the test a memory is needed, we will use the onboard SRAM of the FPGA. Table 7 shows that the Spartan 2 FPGA has a total of 56K block memory, hence 1024 test vectors can be achieved with plenty of memory blocks to spare.

31:3	2:0
iData	Instruction

Table 9: The 32 bit word of the improved design, while in instruction mode.

The improved design is using 32 bit instructions as well as data; hence the iData component changes from 5 to 29 bits. Other than that the instructions stay the same.

The *Pattern Generator* supports the following inputs while in instruction mode:

Instruction	Word[2:0]	Description
CLEAR	000	Resets the system.
SETCLOCK	001	Sets the clock to internal clock divided by iData.
LOAD	010	LOAD is followed by 4 byte describing the length of the data segment (max 2^{32}) and then the data itself, ranging from 2^0-2^{32} bytes of data.
RUN	011	Start generating signals until test is stopped.
STOP	100	Stop generating signals.

Table 10: Supported instructions of the controller.

The hardware should be capable of taking input commands through the PC interface and execute those as they arrive. Since this is human interaction and therefore impossible to pre determine from the hardware's point of view, the PC interface should notify whenever there is any new data on the line.

The communication between the different parts of the hardware can either go over a bus or dedicated lines. The dedicated lines are most likely better than agreeing on a bus type due to the limited amount of nodes and simplicity of design.

To facilitate different test frequencies a *programmable clock divider* is also needed. The clock divider divides the FPGA clock of the PG by a parameterized value giving the user control of the frequency of the test. The clock divider is controlled with the SETCLOCK instruction described in table 10. The command divides the clock with the iData number. Table 8 shows how the SETCLOCK instruction and iData should be defined.

As an example: If one wants the test frequency of 25 MHz the divider should be 2 when using a 50 MHz FPGA clock. The instruction should be:

iData	Instruction
00010	001

Table 11: Example demonstrating a clock divider of 2 with the base design.

This means the data word should be 0x11.

Divider [Decimal]	Testing Frequency [Mhz]	Data word [Hex]
1	50	09
2	25	11
4	12.5	21
8	6.25	41
10	10	51
16	3.125	81

Table 12: Relationship between clock divider and testing frequency.

The relationship between the clock divider and actual test frequency can be seen from table 12 along with its equivalent hex command.

The controller is responsible for decoding the instructions sent from the PC. To be able to use the full signal as either instructions or data it is important to distinguish between *Data mode* and *Instruction mode*.

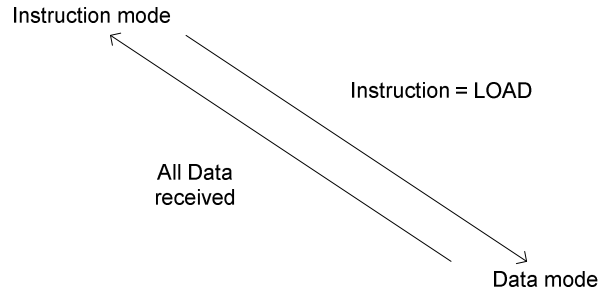


Figure 14: The correlation between Data and Instruction mode.

The system always starts in instruction mode. Whenever the controller is in instruction mode, a LOAD command will trigger the Data mode. After the load command is received, the next 4 byte of data defines when to return to Instruction mode.

This design has the benefit of freeing up the whole 32 bit for data transactions as well as data transactions; unfortunately it is very vulnerable when operated incorrect. Telling the system to download 2 test vectors and only feeding it one, will halt the system indefinitely. A timeout mechanism would be ideal here but has not been deemed necessary at this point.

To get out of this potential infinite stall, the user can feed arbitrary data vectors (except the one equivalent to the LOAD command) to the system until he or she is convinced that the PG have returned to Instruction mode. The memory will contain random data that must then be overwritten but at least the user will regain control of the system.

Components

The hardware design has been divided into different components based on their functionality; figure 13 show the overall definitions of the components.

Serial Receiver

A data ready signal is asserted each time a byte is ready to be read and stays active for half the time until a new byte is ready.

With a transfer rate of 19200 baud that means that there is a new byte ready every ~0.4167 ms and it is valid for another ~0.2083 ms; hence the controller must run at a frequency of 9600 Hz not to miss any input data.

$$1/\left(19200baud/8\right) \;=\; \sim 0.4167\, ms$$

$$1/\left(19200baud/8\right)/2 \;=\; \sim 0.2083\, ms$$

Controller

The controller is using a state machine to fetch and decode the input.

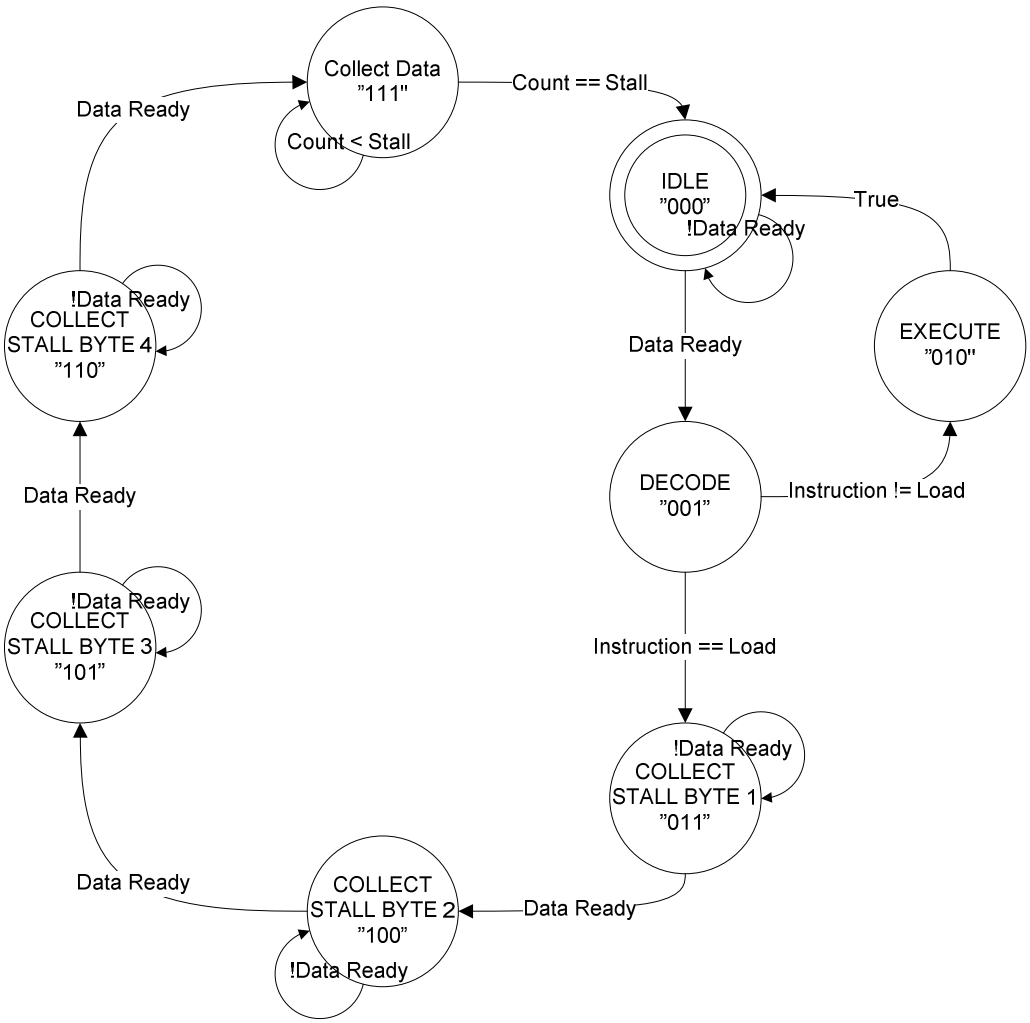


Figure 15: The Finite State Machine used to control the circuit.

Since the receiver collects 1 byte before forwarding it to the controller, the controller has to collect the bytes into words. Whenever a word has been assembled, it is written to the memory. For more information on state machines see (1) appendix B 5.

Pseudo noise

The Pattern generator does not have the ability to emulate noise on its signals. However if we want to run a test at 10 MHz on a 50 MHz Pattern Generator with the sequence 1, 2, 5, it can be written as

1, 1, 1, 1, 1,	0,	2, 2, 2, 2,	7,	5, 5, 5, 5
First period	Second period		Third period	

The 0 and 7 indicating a short instability of 20ns before the signal settles. In that way the user has some way of emulating noise by polluting his or her test vectors.

Clock generator

The clock generator is responsible for creating the output frequency specified by the user. It is basically a clock enable turning the output register on and off. The clock enable is generated from a comparator comparing a counter with an array of registers defining the divider of the clock.

Memory

The memory should have a width of 32 bit since the test vectors are of 32 bit length. Since the input from the serial connection is 1 byte, a shift register is needed concatenating the output into 32 bit words before loaded into the memory.

When looking at the circuit, it is important to notice that it consists of a slow and fast path, the slower part being the connection to the PC and the faster part being the path from the memory to the output pins. To maximize the possible output frequency we concatenate the bytes into 32 bit words on the input to the memory rather than storing 8bit words and concatenating them on the output. This way the delay of concatenating the bytes is put into the slower part of the circuit negating its impact due to the much slower PC connection.

For more information on the Spartan 2 block RAM see (12).

Implementation

Software

The software GUI laid out in the design phase is accomplished in the Visual Studio Form Designer. I chose to write my program in C# using Microsoft Visual Studio and .Net 2.0, because in my opinion, it is the fastest and easiest way to develop a quick Graphical User Interface. I also have previous experience with this from another course at DTU[1]. The source code can be found in Appendix C. For more information on design patterns see (13).

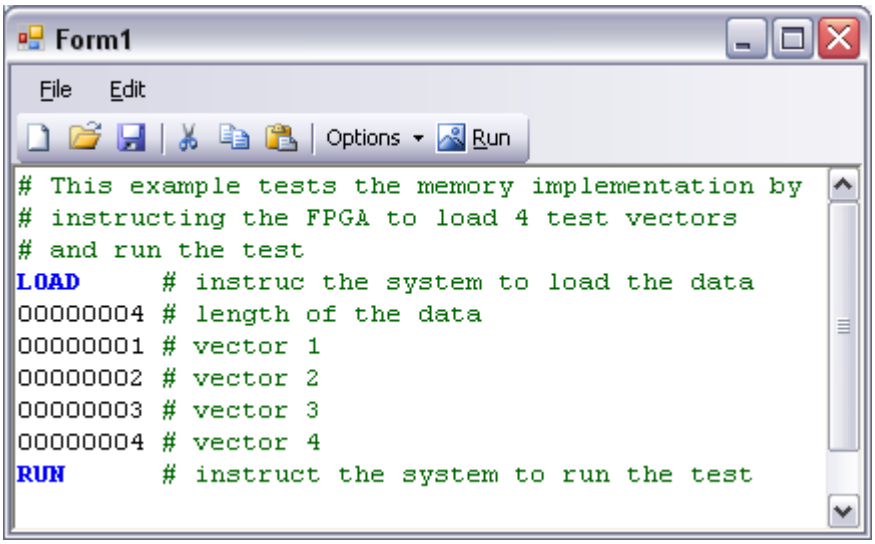


Figure 16: Running a 4 vector test.

[1] 02350 Windows Programming using C# and .Net

The 'options' dropdown menu lets the user choose between different connection types loaded from DLL's.

To reduce the flicker, happening when the syntax highlighter is replacing text with its formatted equivalent, the paint message is disabled. When the text has been replaced, the paint messages are enabled again.

```
        private void txtBody_TextChanged(object sender, System.EventArgs e)
        {
            FlickerFreeRichEditTextBox._Paint = false;
            // <text formatting code>
            FlickerFreeRichEditTextBox._Paint = true;
        }
```

The `FlickerFreeRichEditTextBox` inherits the `RichTextBox` control and is responsible of repainting itself depending on its `_Paint` variable.

```
public class FlickerFreeRichEditTextBox : RichTextBox
{
  const short WM_PAINT = 0x00f;
  public static bool _Paint = true;
  protected override void WndProc(ref System.Windows.Forms.Message m){
    if (m.Msg == WM_PAINT){
      if(_Paint)
        base.WndProc(ref m);
      else
        m.Result = IntPtr.Zero;
    }
    else
      base.WndProc (ref m);
  }
}
```

The `FlickerFreeRichEditTextBox` differ from the `RichTextBox` by having a special WndProc() method capable of ignoring `WM_PAINT` messages when the _Paint variable is not set.

Classes

The most important classes are described in the following sections.

IConnection and SenderResult

The IConnection class defines the interface to the connection DLL's such as the serial connection. The Send method returns an object of the SenderResult class containing the status of the send command.

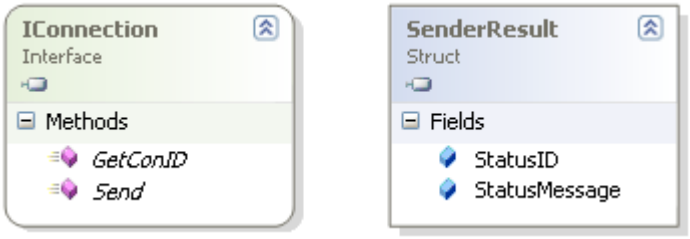


Figure 17: The IConnection and SenderResult Classes

FConnection

FConnection provides a facade to the GUI. It is responsible for giving the GUI information of the different connections and is acting as a middleman between the frontend (GUI) and the backend (the IO connections) of the application.

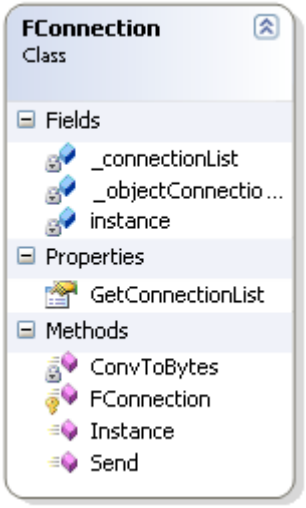


Figure 18: The Facade connection class.

Reflection

Fconnection uses reflection to load the connection DLL's. To do so, it looks through the Addins directory for possible connection DLL's.

```
string[] assemblies = Directory.GetFiles(AddInsDir,"*.dll");
```

Each assembly is opened and if it adheres to the *IConnection* interface, it is accepted as a valid connection.

```
if (type.IsClass == true && type.GetInterface
   ("S012012.ConnectionInterface.IConnection") != null)
```

Once we know it adheres to the interface, we know it has the **GetConID()** method which can help us identify the connection. Each valid connection is entered into a `connectionList` for future handling.

```
_connectionList.Add((string)type.InvokeMember("GetConID",
BindingFlags.Default | BindingFlags.InvokeMethod, null,
ibaseObject, new object[]{ }));
```

If no DLL's adhere to the interface, no connections will be made and the user will get an empty list of connections.

The Dynamic Link Libraries

Serial

The serial class is responsible of the actual connection to the port and is derived from IConnection.

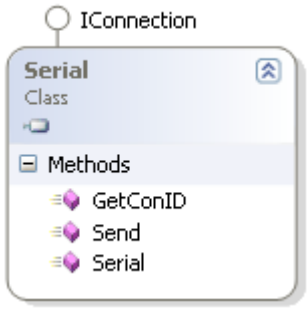


Figure 19: Serial class.

Serial RS-232 connections are supported directly in the .NET 2.0 framework and can be used as follows:

```
SerialPort port = new SerialPort("COM1", 19200, Parity.None, 8,
          StopBits.One);
port.Open();
port.Write(data, 0, data.Length);
```

To use the Serial port the program instantiate the serial port support class and use its write function to send the byte array.

Hardware

To get a fast working model, a baseline design without constraints is created; such a model help identify problems that can be taken care of in later design iterations. After getting a working model the next step would be to optimize it and redo any problematic points. The full hardware design can be found in Appendix E.

The idea of the first implementation is to create a system with the following components:

- A receiver maintaining the connection to the PC.
- A memory component.
- A controller acting as both controller and clock generator.

The components are connected as shown below.

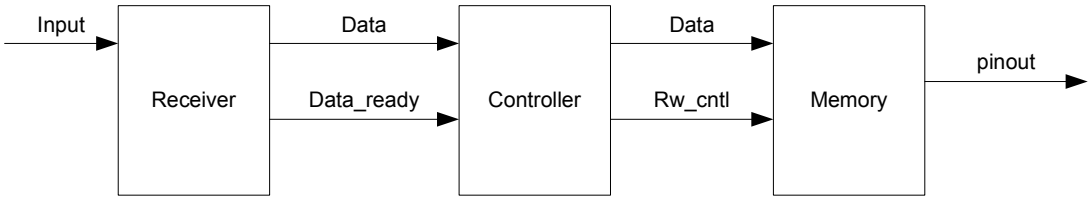


Figure 20: The envisioned datapath of the circuit.

The data path illustrates how the data goes from the receiver to the controller and memory. The memory outputs the data directly to the output pins. Each component is connected to the main clock of the FPGA. Since the ISE software is capable of automatically buffering signals with IOBUF buffers, the VHDL code contains no reference to buffers.

Controller

The controller is responsible for the control signals to the memory and thereby controlling the output of the circuit. It generates outputs to the memory based on the inputs received from the receiver component.

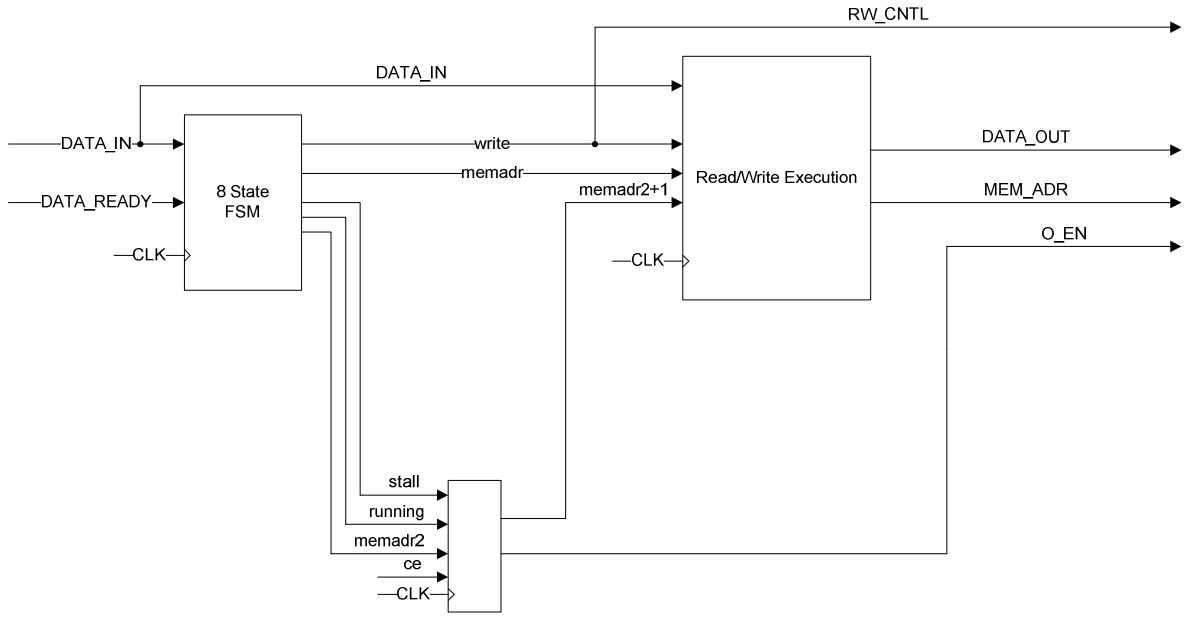


Figure 21: The implemented controller.

The controller collects bytes into words and makes sure they get written to the memory. It is also responsible for implementing the instructions needed to control the PG and the clock divider functionality. As described in the design phase, the controller is pretty much a big state machine and some added logic.

Frequency Divider

The frequency divider is actually a clock enable. Since the output of the FPGA is controlled by toggling the control lines of the embedded memory, the frequency divider will just have to enable/disable the logic responsible for generating the memory inputs at the right time. To generate the clock enable signal we need a comparator, a counter and registers for the clock division.

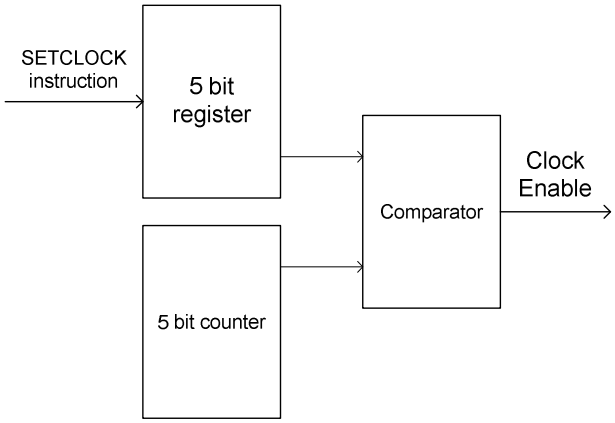


Figure 22: The clock divider.

The clock divider is not implemented as a actual component, instead it is embedded in the controller component.

Memory

The SRAM of the FPGA is used as memory. The Spartan 2 XC2S200 FPGA has its memory located at the edge of the board and divided into 14 x 4096 bit banks for a total of 56K as noted in table 7. When using the Xilinx block RAM reference designs, those blocks are connected to form one big memory with a slightly larger timing penalty. Since we require a 32768bit memory 8 blocks are consumed. The memory is a single port implementation with 32 bit wide words and a depth of 1024.

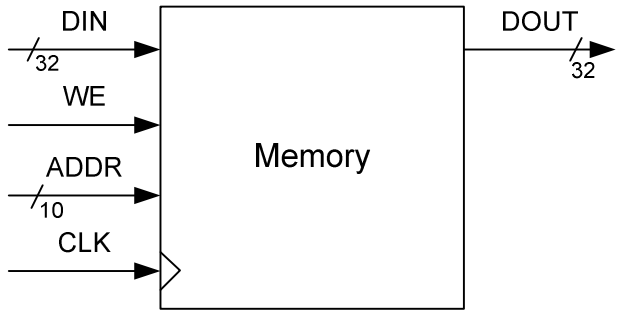


Figure 23: Single port block memory.

The memory has 4 inputs and 1 output. WE defines the operation as read or write, ADDR defines the address in the memory that the operation should access and DIN defines the input to the memory if it's needed. All inputs are mirrored on the outputs when writing to the memory. For more information see (10).

Improving the base design

Having a functional working base design was the main objective of this report; however there are plenty of points that could be improved upon. This section will outline the design and implementation of those new improvements.

Overall changes

Based on the first implementation there are obvious steps to take

1. Reduce the complexity of the controller, hopefully reducing the depth of the logic.
2. Remove the clock from as much of the circuit as possible to save registers and improve performance.

Having 8 bit instructions and 32bit data really only complicates things, therefore the instruction input are changed from 8 to 32 bit, this has the added benefit of adding more values when using the SETCLOCK instruction.

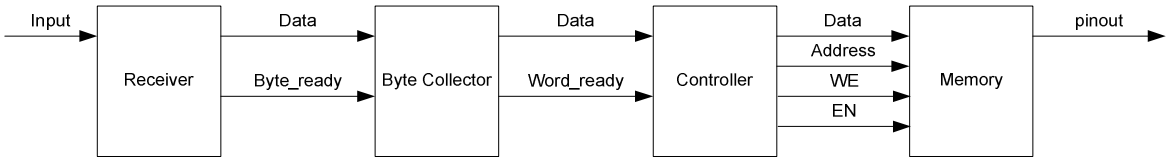


Figure 24: The data path of the improved circuit.

Now that data and instructions are of same bit length a unit that's responsible for generating the words can be made, which should also reduce the complexity of the controller FSM.

Byte Collector

We introduce the *Byte collector* which collects bytes from the receiver into words. The in and outputs of the *Byte Collector* is depicted in figure 24 and figure 25.

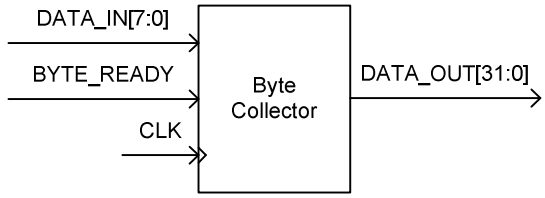


Figure 25: The Byte Collector.

The byte collector checks for positive edges on the Byte_ready input and loads the byte into a shift register. Each time 4 bytes has been received the word_ready is asserted for 1 clock cycle while the data is written to the output.

Revisiting the Controller

A new FSM is made, the biggest difference is how the FSM is more general due to the introduction of the byte collector, and that the FSM doesn't have to control the receiving of words.

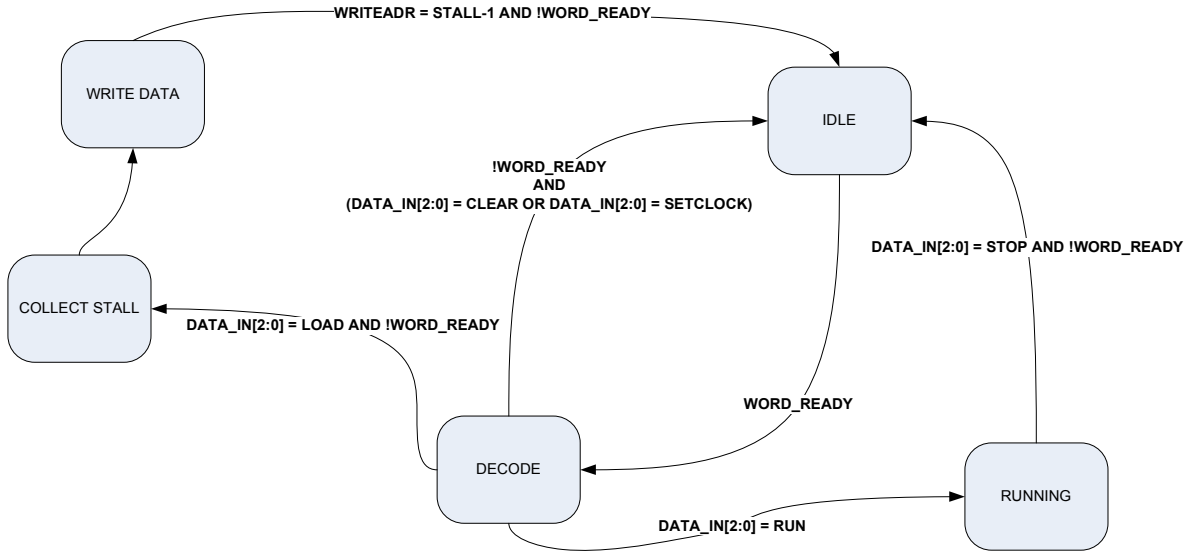


Figure 26: Second implementation of the controller FSM.

When the instruction has been executed the state machine returns to IDLE. Since 'LOAD' is a multiword instruction this means that the state machine stays in the WRITE DATA state for multiple cycles and cannot take any inputs while this is going on.

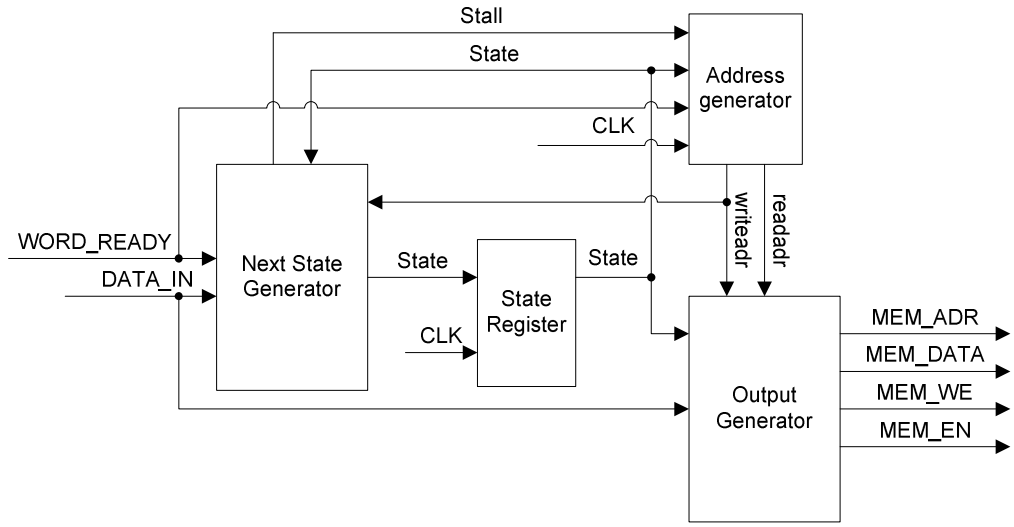


Figure 27: Data path of the improved controller. Connected lines are marked with a dot.

The controller is still responsible for creating the inputs to the memory. The main thing about the controller is how the timed circuits are removed from the rest of the logic and separating the next state

and output function from the state register. Figure 27 shows how the state machine has been moved out in 4 different blocks with two being clocked and 2 combinatorial blocks.

The memory is kept almost the same; an enable pin is added to complement the write enable.

Most of the optimizations were inspired by articles named TechXclusives written by the Xilinx staff. See (13), (14) and (15) for more information.

Unlike the base design which has a maximum frequency of 50 MHz, the improved design can at most operate at a frequency of 25 MHz. This frequency has been chosen due to signal instabilities at higher frequencies as described in the test section.

Evaluating the implementation

The implementation satisfies the requirements laid out in the analysis. Both the base design and the improved base design are working as expected. For more information on the test, see the test chapter. For the code see Appendix F.

Comparing statistics between the base design and improved design the following changes can be observed.

	Base Design	Improved Base Design	Buffer Design
Registers	357	180	46
Max Frequency [MHz]	65.441	75.982	79.592

Table 13: Synthesis report Statistics.

The buffer design will be described in Chapter 4.

CHAPTER 4

FIFO Buffer

Another approach to the design would be not to store the data on the FPGA but run the test 'on the fly' while the data gets transferred. This implementation requires a buffer rather than a memory.

The *Buffer* implementation is mainly being controlled from the software, with the FPGA acting as a transparent receiver forwarding the data to the UUT, while the *Memory* implementation requires additional control.

Design

Having a *First in First Out* (or FIFO) buffer on the FPGA rather than a memory with an instruction set reduces the complexity of the design a lot. The reason for this is partly because of a decrease in functionality and partly because the Xilinx COREGEN tool can generate a FIFO capable of replacing the supporting functions of the base design. Basically, all we have to do is connect the receiver component with the FIFO and we got ourselves a forwarding unit between the PC and DUT.

The Xilinx COREGEN tool is capable of generating Xilinx reference designs of various components including memory elements such as FIFO buffers.

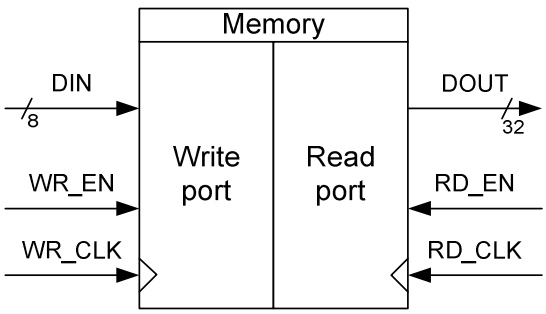


Figure 28: Xilinx FIFO buffer reference design.

The FIFO is as its name implies a buffer that outputs the content in the same sequence as it's entered. It supports different input and output frequencies and bit lengths making it possible to give it 8 bit inputs from the receiver and outputting 32 bit outputs to the output pins of the FPGA. To toggle the FIFO on and off Xilinx recommends gating the read and write enables rather than the clock signals.

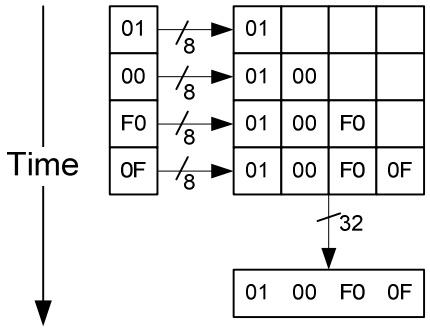


Figure 29: The FIFO doing 8 to 32 bit conversion.

The output frequency of the FIFO is limited by the input frequency and the input and output bit width. Better described as:

$$output\ frequency = \ input\ frequency\ x\ \frac{input\ bit\ width}{output\ bit\ width}$$

This is where the Parallel port has a direct influence with its much higher throughput.

Implementation

The *Buffer* implementation is rather simple compared to the *memory* implementation. The FIFO generated by the Xilinx COREGEN tool replaces the byte collector and memory, and since the controller isn't needed for this type of implementation, it is removed as well.

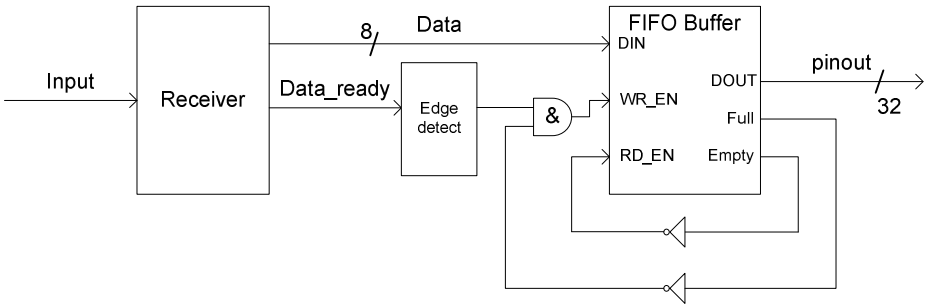


Figure 30: The new datapath.

Since the FIFO buffer writes each cycle when write enabled we make sure only to assert WR_EN on the positive edge of Data_ready, we read out of the buffer as long as it's not empty.

The implementation proposed does not make use of flow control, the FIFO reference design does support warning flags whenever the memory is about to be filled up such that a design based on two way communication could make use of some sort of signaling to the software whenever the memory was about to overflow.

According to the LogiCORE™ FIFO Generator documentation (17), the FIFO is non destructive, which basically means that overflows won't have any effect on the content of the FIFO. The data that is attempted to get written will however be lost. The VHDL source code can be found in Appendix F.

EPP Interface

To improve the design its worth looking at the PC connection. To be fair, a serial connection is far from optimal with its extremely low transfer rate of 9600-115200 bits per second, therefore we examine the parallel connection with a possible transfer rate between 1.5 and 2.0 megabyte per second. Further documentation of IEEE 1284 which contains the parallel port specification can be obtained from (5).

A serial connection might not have many problems transferring 32K of data, but once the size of the pattern generators memory increases, data transfer rates will become an issue. Also in the case of running the test 'on the fly', outputting signals as they arrive from the PC, a faster connection to the PC is desirable.

Software

Connecting to the parallel port in windows XP is a lot more complicated than one would think. The .NET framework doesn't support parallel port communication and since Windows XP does not allow direct access to the IO registers like earlier versions of windows did, a driver is needed. Windows XP only allow kernel mode drivers to access IO, and therefore we need one of those. Fortunately the website www.logix4u.com provides a DLL containing a IO port driver that is capable of loading itself into kernel mode when called from user mode. More information can be obtained from (16).

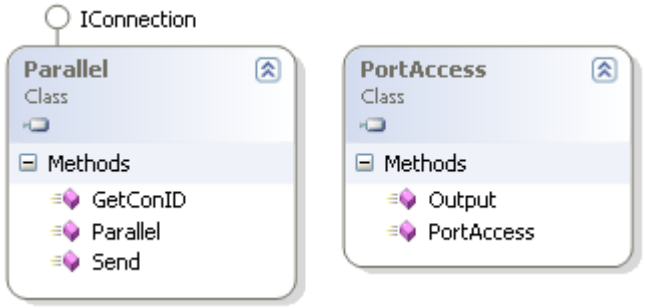


Figure 31: Parallel and PortAccess class.

The PortAccess class is responsible for loading the IO driver and provides the functionality to the Parallel class implementing the connection interface.

Loading the DLL:

```
[DllImport("inpout32.dll", EntryPoint = "Out32")]
public static extern void Output(int adress, int value);
```

The DLL itself is fairly easy to use, all that's needed is to pass the address and data argument to the `Output` function.

```
foreach (byte b in data)
{
    PortAccess.Output(BASE_ADR + 4, b);
}
```

The windows IO registers are memory mapped so the address that must be passed is the address of the EPP data register which resides at the base register + 4. The base register is usually located at address 0x378. In windows XP the address can be found through the device manager.

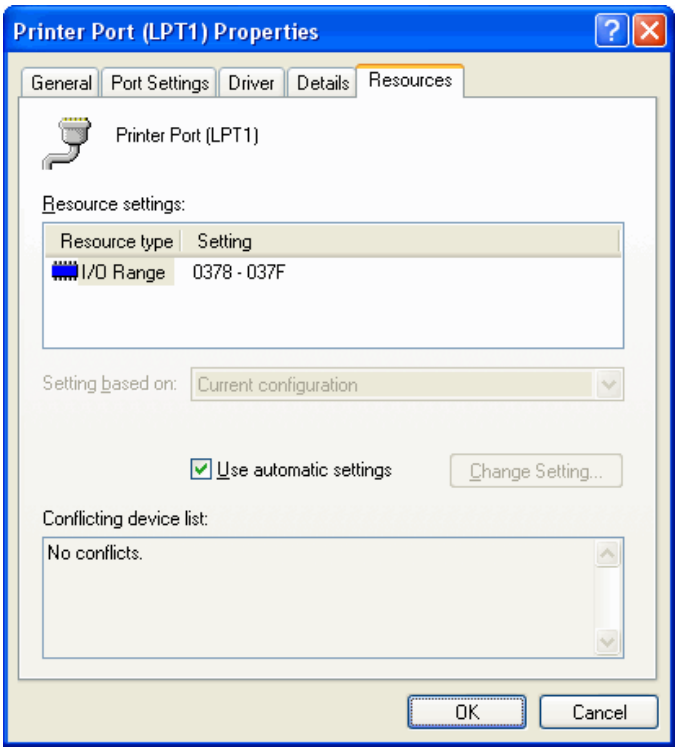


Figure 32: Device manager showing LPT1.

Since the Xilinx IMPACT FPGA configuration software tends to put the parallel port into ECP mode, we also have to put the port back into EPP compatible mode after configuring the FPGA. This is done by writing 0x80 to the Extended Control Register (ECR) located at base reg + 0x402. Apparently, even though the Digilent D2 Board data sheet specifies the parallel port as an EPP interface, the board seems to function with ECP mode as well.

Hardware

The Digilent D2 Spartan 2 FPGA Board has a Parallel port implementing the Enhanced Parallel Port (EPP) protocol. The EPP uses an 8 bit data line along with a 3 signal handshake for a data transmission. All the control signals are active low as indicated in the figure. For more information see the board specification (11).

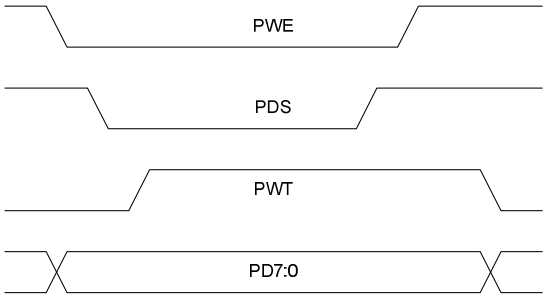


Figure 33: Handshake of the EPP Data Write, all control signals are active low.

Being a handshake protocol we don't have to worry about agreeing on a transfer rate.

With a theoretical transfer rate between 1.5 and 2.0 Mbps it's a significant improvement. Compared to the serial connections max transfer rate of 115200 Baud, we are looking at an improvement of at least a factor of 12.

Since we use the same output signals from the receiver, we don't have to change the controller component.

Future improvements

Aside from the implemented improvements, a few ideas have been under investigation. These improvements were chosen not to be implemented, primarily because they were outside the scope of the project and would have taken a substantial amount of time to implement. The improved Pattern generator would have the following specifications:

- Buffer implementation
- USB 2.0 PC interface
- Matlab plugin capable of writing to the PC interface

The hardware would be capable of handling an unlimited amount of test data because of the buffer implementation and would also have the benefits of the best possible testing frequency obtainable, due to the USB 2.0 interface. The only way to improve the speed beyond that would be a custom built IO interface.

The software would give the designer the possibility of algorithmic generated test data. This might also increase productivity since the engineer would have a familiar working environment.

Another possibility would be having a network interface on the PG, making it work like a network resource instead of having it physically connected to a PC. This mechanism would make the device easier accessible but would require some sort of IP lock or semaphore for mutual exclusive access.

CHAPTER 5

Test

Introduction

The test chapter will document some of the more interesting test cases. The testing can generally be said to be divided in four parts.

- Software testing
- Hardware simulation
- Functional real world testing
- Application testing

The software test makes a functional test of the PG Controller software by running a fictive test and analyzing the output of the software.

The hardware simulation is simulating the receiver components, both serial and parallel, along with the improved controller component. The base design has been tested as well but I felt having one test of the controller would be enough for this chapter.

The real world testing is focused on the functionality of the system working on a real FPGA. Both a serial and parallel PC interface is used, along with the clock divider instruction. A Logic Analyzer is used to analyze the output of the system.

To test the application of the system, a 4 bit adder is implemented on a second FPGA and connected with the Pattern Generator through a cable.

More test results can be found in Appendix I.

PG Controller software

To test the software I create a TCP/IP connection DLL that is sending to localhost. Another program is made to capture the TCP/IP data and show it for further inspection. See Appendix C for the sourcecode of the PG controller software.

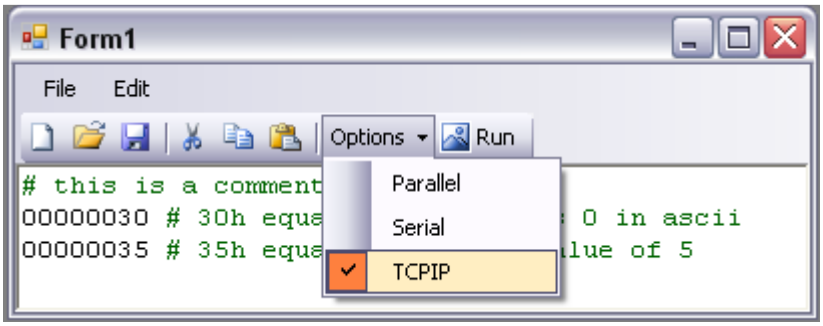


Figure 34: The PG software sending to localhost for inspection of the data.

We choose to send the ASCII values of 0 and 5 (0x30 and 0x35) over TCP/IP to localhost for inspection.

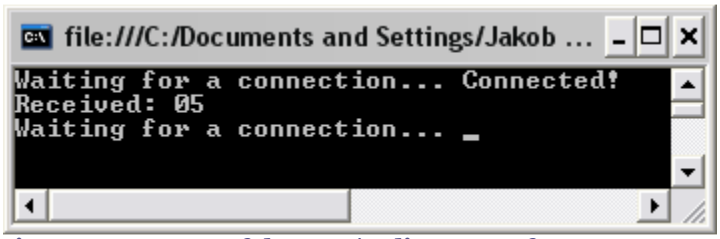


Figure 35: Output of the TCP/IP listener software.

The TCPIP listener captures the data and writes it to the console. Once knowing the base program is working as intended, the actual test can continue testing on the actual hardware. See Appendix D for the sourcecode of the TCP/IP listener. We conclude that the base program works as expected.

At 1M (2^{20}) test vectors the application is consuming roughly 155MB memory according to the Windows XP task manager. With a 15MB overhead from the program itself this raises the question if it would be possible to find a better way to store the test vectors in memory.

Number of testvectors	Memory consumed [K]
0	15,036
2^{20}	154,512

Table 14: Memory consumption of the PG controller software.

To summarize, the memory consumption of the PG can be approximated to about:

$$15MB + 0{,}133KB \times Number\ of\ test\ vectors$$

It should be mentioned that in the practical use of the system, there is a maximum of 1024 testvectors due to the limited RAM on the Spartan II.

Simulating the hardware

Modelsim from Mentor Graphics is used to simulate the design, while doing the early design stages the ISE embedded simulator has also been used. The sourcecode of the test benches can be found in Appendix B.

When simulating the components a test of 28 bytes is being used with the sequence of LOAD 4, 1, 2, 4, 8, RUN. The binary inputs to the testbench can be found in Appendix A.

The serial receiver Testbench

The success criteria of the serial receiver is to be able to receive data over a serial connection at 19200 baud with one start bit followed by 8 data bits and one stop bit. The serial receiver should signal whenever a byte is ready.

To simulate the receiver we create a testbench capable of loading data from a file and supply them to the unit under test following the correct protocol.

```
RXD <= '0';                    -- start bit
wait for period;
readline(cmdfile,line_in);     -- Read a line from the file
for i in 0 to 7 loop           -- Fetch all data from the line
   read(line_in,RXD1,good);    -- Read a bit from the line
   RXD <= RXD1;
   wait for period;
end loop;
RXD <= '1';                    -- stop bit
wait for period;
```

The file being read from contains 8 bit values each line and since serial connections send the LSB first the data in the file is reversed MSB at LSBs position and vice versa.

Simulating the serial receiver

When simulating the design we expect the data to follow the pattern of 00000002 00000004 00000001 00000002 00000004 00000008 00000003. Seing as we are sending 28*(8+2) = 280 bits with a speed of 19200 bits/second we will need to simulate the design for a period of ~14,6 ms.

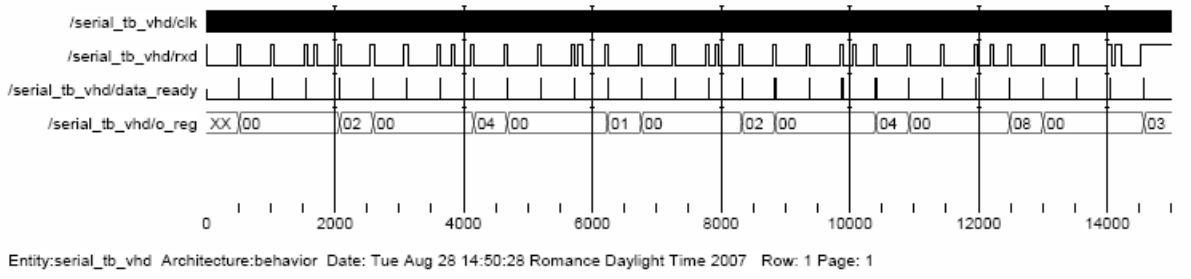


Figure 36: Serial receiver simulation. Time markers indicate microseconds.

Notice how the data_ready signal gets asserted regularly and that the data itself, represented as the signal o_reg, follows the expected pattern of 00 – 00 – 00 – 02 – 00 – 00 – 00 – 04 – 00 – 00 – 00 – 01 -00 – 00 – 00 – 02 -00 – 00 – 00 – 04 -00 – 00 – 00 – 08 -00 – 00 – 00 – 03.

The EPP receiver testbench

The EPP receiver receives data over an 8bit data bus at 1.5 – 2.0 MB/s and uses additional pins for handshaking.

The parallel port has a slightly different protocol than the serial connection, instead of just sending the data at a pre determined frequency the EPP testbench has to wait for the receiving device to assert or deassert the wait signal.

```
EPP_WRITE_ENABLE <= '0';        -- start handshake
EPP_DATA <= '0';
readline(cmdfile,line_in);      -- Read a line from the file
for i in 0 to 7 loop            -- Fetch all data from the line
        read(line_in,tmp,good);     -- Read a bit from the line
        assert good report "Text I/O read error" severity ERROR;
        EPP_DATA_BUS(i) <= tmp;
end loop;
wait until rising_edge(EPP_WAIT);
wait for 40 ns;                 -- added delay such that the handshake doesn't happen
                                -- instantly
EPP_WRITE_ENABLE <= '1';        -- finish handshake
EPP_DATA <= '1';
wait until falling_edge(EPP_WAIT);          -- handshake done
wait for 546 ns;
```

The 546 ns wait statement at the end is only to make the connection function at the speed of the EPP specification and has nothing to do with the actual functionality of the testbench.

Simulating the EPP receiver

Sending the data should take about

$$\frac{28\ byte}{1500000^{byte}/_{second}} = 18.7\ \mu s.$$

The data_out signal is expected to repeat the epp_data_bus signal with a slight delay while the byte_ready signal gets asserted periodically.

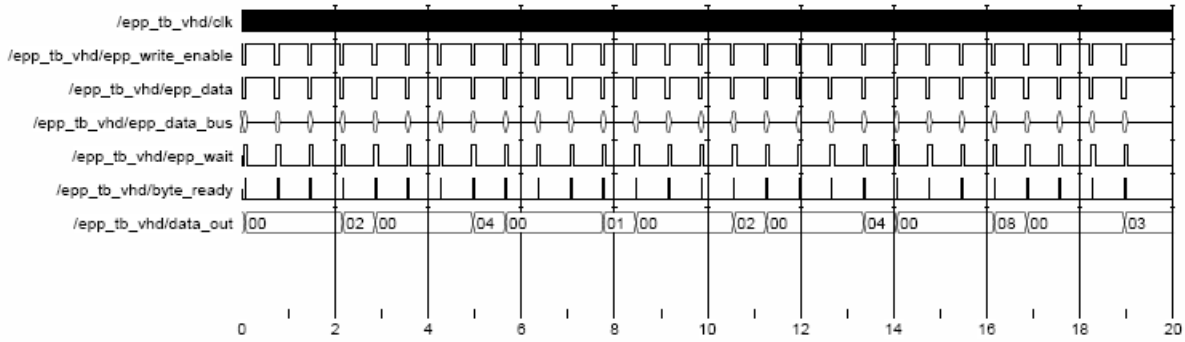


Figure 37: Simulating the EPP receiver. Time is measured in microseconds.

The wave figure shows the ingoing and outgoing signals of the EPP receiver. Again, the output (this time called data_out) follows the expected pattern.

Figure 37 shows the full duration of the test while Figure 38 shows a close-up of the handshake happening.

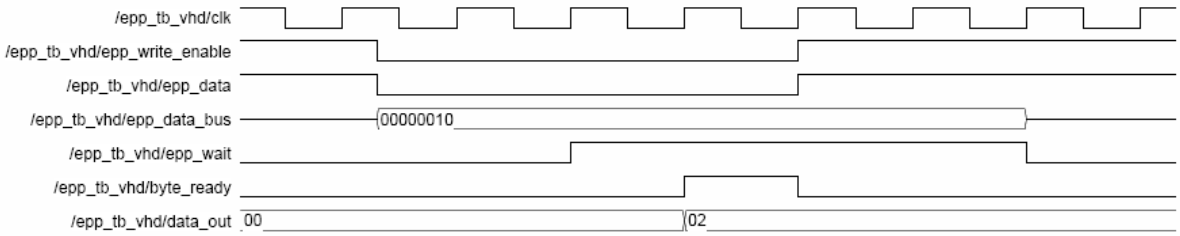


Figure 38: Close-up of the handshake.

Recall Figure 33 describing the handshake in detail. As can be seen from Figure 38, the handshake is working exactly as expected, data_ready gets asserted for one clock cycle while the data is being written to the output register.

The Controller

The controller for the memory implementation is a bit more complicated. It decodes instructions, forwards data to the memory and controls how the memory operates.

It is expected to see the instructions getting decoded and the memory control lines toggled accordingly.

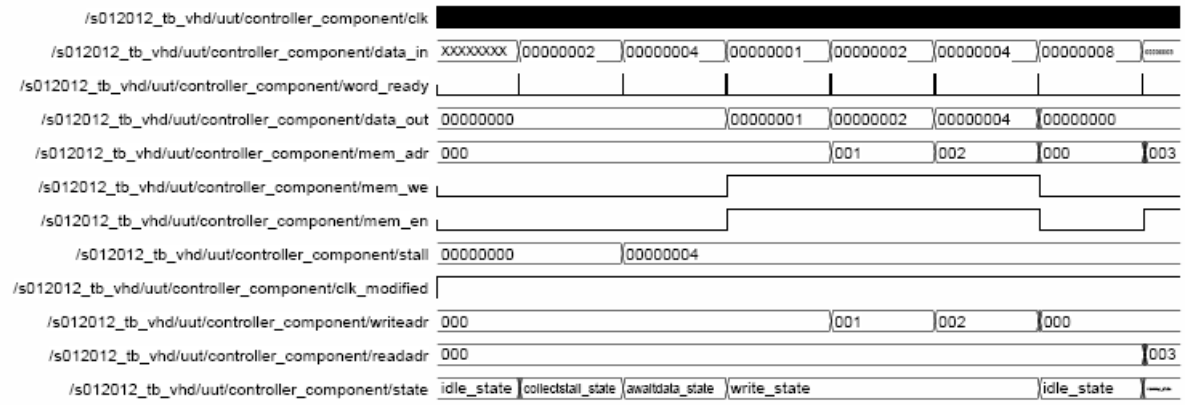


Figure 39: Simulation of the controller.

The Modelsim simulation show correct behavior by the controller component, sending 32 bit values to the memory while toggling the control signals. It isn't really possible to see the reads from the memory in the waveform above due to the much faster frequency so here is a close-up.

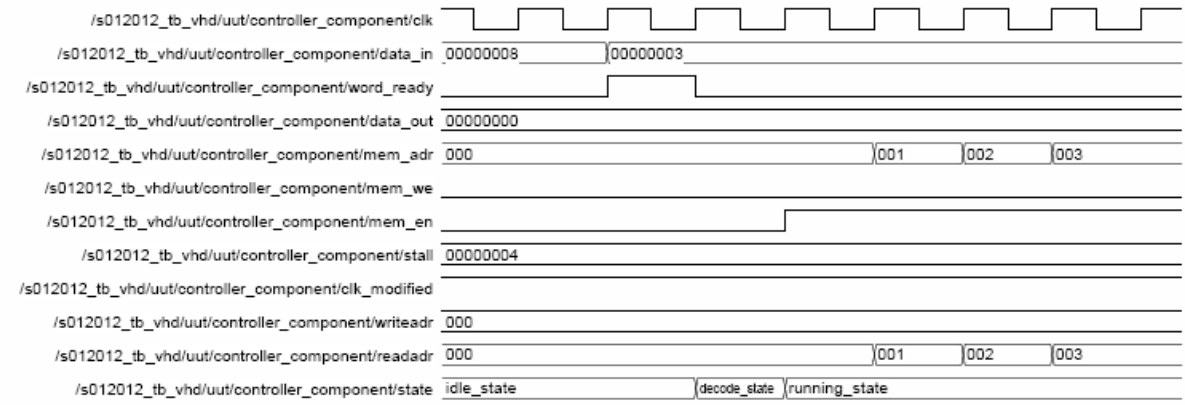


Figure 40: Close-up of the RUN command getting executed in the controller component.

As expected the address counter gets incremented each clock cycle while the enable signal is asserted and the write enabled signal is low indicating a read operation to the memory. It is also possible to see the run command (0x03) getting executed by looking at the state signal of the controller component entering the running state. We can conclude that the controller works as expected.

Functional Real World Test

The AT-LA 500 logic analyzer is used to capture the output of the FPGA with the design implemented. The FPGA is set to output the data at a frequency of 25 MHz.

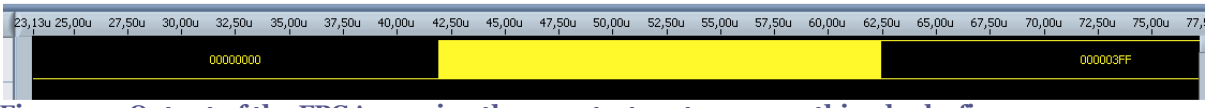


Figure 41: Output of the FPGA running the 1024 test vectors, everything looks fine.

Figure 41 shows how the output of the PG switches ~1024 times indicated by the yellow block. The implementation seems to be working great, however there is a problem when looking closer.

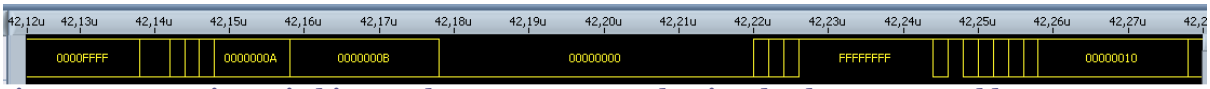


Figure 42: excessive switching on the outputs causes the signal to become unstable.

What we see here, is how the signal becomes unstable when switching. The Spartan 2 data sheet (10) refers to this problem as SSO or simultaneous switching outputs causing ground bounce and VCC bounce. Ground bounce can happen when a signal switches from high to low causing a voltage to develop in the ground plane. Likewise signals switching from low to high can cause a surge in the power supply lines.

Normal countermeasures include reducing the drive strength, adding more ground signals, reduce the frequency of the signals and/or add debounce circuitry to the input of the DUT. With 32 signals switching at the same time a 50 MHz signal can be so distorted it's impossible to decipher, effectively invalidating it. For more information on SSO see (18).

At 50 MHz the noise becomes so heavy the intended signal is almost not visible when having all 32 outputs switching as can be seen from Figure 42 where the instability period approaches 16ns. Reducing the frequency to 25 MHz increases the signal integrity giving only short instabilities of up to 8ns with maximum (32) bits switching. These numbers were obtained by sampling the output at 500 MHz with 4 ground signals as recommended by (10). See Appendix I ReqID: 5 for tests regarding ground bounce.

The observed transfer rates of the connections are shown below.

Connection Type	Observed runtime [s]	Amount of data [Bits]	Transfer rate [Bit/second]
Serial	2.136	41080	19232
Parallel	0.02293	32864	1433204

Table 15: Observed transfer rates of the connections.

Note that the serial connection transfers more data due to the start and stop bit.

Using the buffer implementation the output speed is limited to the frequency of the PC connection; this alleviates any issues with ground bounce caused by SSO with the current PC connection types. However, using a high-speed PC connection such as USB 2.0 or PCI Express would reintroduce this problem.

The speed increase when going from a serial to parallel interface is obvious. However, due to the size of the memory, the parallel port never really gets to shine when implemented with the memory solution. When using a buffer implementation, the use of the parallel port becomes much more interesting.

Clock divider

The clock divider is tested by setting the frequency divider to 16.

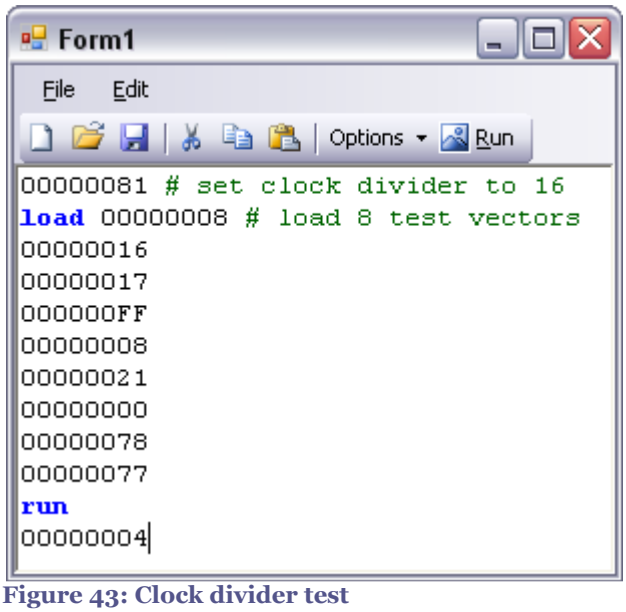


Figure 43: Clock divider test

A 320ns period on the output is expected.

Figure 44: Measuring period of the adder output.

The delta field tells us that the marker 1 and marker 2 are 320.596ns apart, the test is a success.

Application Test

To test the application of the Pattern Generator a simple 4bit adder is implemented on a second Spartan 2 FPGA. The output pins of the PG are connected to the input pins of the adder.

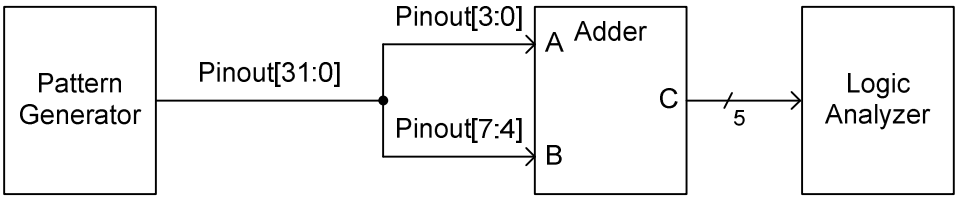


Figure 45: Using the PG to test an adder circuit.

A test is created in the PG software. Being a 4 bit adder, the last digit of the hexadecimal test vector defines the A input while the second last digit defines the B input.

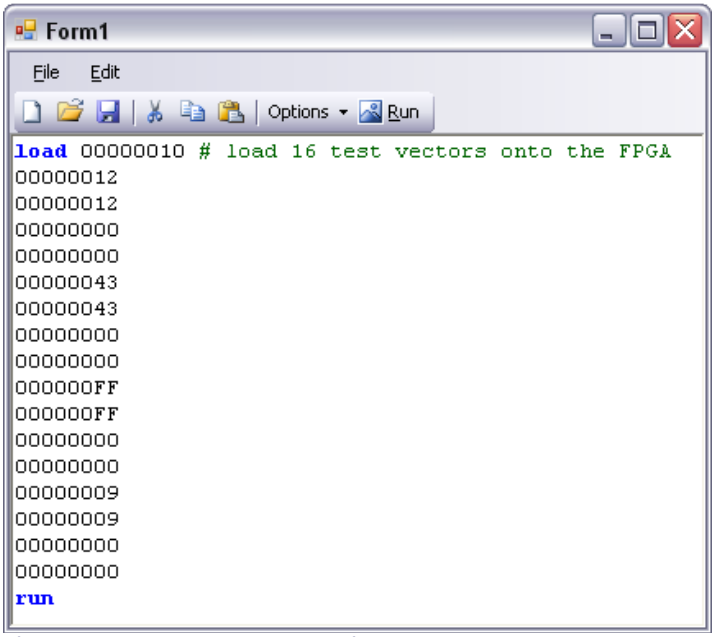


Figure 46: PG software running a 16 vector test.

Another way to achieve half the clock frequency is to generate the same values twice as show in figure 46. We expect the output values of the adder to be 3 – 0 – 7 – 0 – 1E – 0 – 9 – 0.

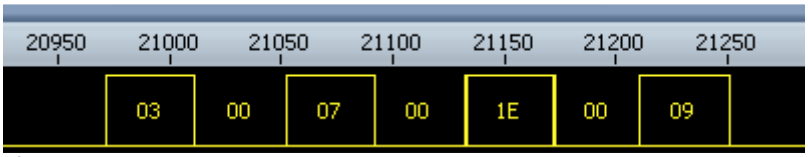


Figure 47: Adder output, captured by the LA.

At a slightly lowered frequency the test yields satisfactory results and the conclusion must be that the PG does deliver. Notice how ground bounce is barely visible around the 1E output.

More tests, specifically aimed at matching the Agilent 81134A described in the analysis, can be found in Appendix I.

CHAPTER 6

Conclusion

The project

This project has mostly been oriented towards a final product from a practical point of view. This also meant that a lot of time was spent on the development of the product that was mentioned in the project specification. During the project several new possibilities for the implementation opened up, some of these has been investigated while others were regarded too complex to consider at that point. That is not to say they were not good ideas that could be investigated in the future as written in the Future improvements section of the implementation chapter.

Time wise the project has progressed steadily; there was even time to investigate alternatives to the original project specification. In retrospect, the project could have had a more ambitious specification, but in the end the result is pretty satisfying, even though there are many things that could be improved upon. As with many first generation projects, the final solution is more of a prototype showing the strengths and weaknesses of the implementation, while at the same time delivering the promised functionality. A second project would hopefully be able to do things right from the bottom up, delivering a more elegant solution to the same functionality; some of this were attempted with success in the section called Revisiting the Controller of chapter 3.

The product

The product presented in this paper manages to deliver the promised functionality laid out in the analysis; however it does have some quirks.

A PC program has been developed capable of sending instructions to The PG implemented on the FPGA. Two interfaces between the PG and PC has been designed (serial and parallel). The PG has been designed to work in two different modes: “memory-based” and “buffer-based”. Both modes has been tested at frequencies up to 50 MHz.

The signal integrity of the output of the pattern generator seemed to be a problem at high frequencies. When testing the pattern generator at 50 MHz the output could not be trusted. The reason for this is ground bounce as described in the Spartan II documentation (10). Lowering the frequency to 25 MHz helped as well as adding more ground signals to the output.

Also, the first test vector had increased problems with noise due to possible switching. When writing to the memory, the inputs are echoed to the output as default by the reference design. This causes signal instability when the memory changes from write to read mode.

The functionality of the Agilent 81134A as described in the analysis has mostly been matched at a fraction of its price. This is quite a feat in itself even if the product does not have the same signal integrity and frequency capabilities. The main reason for the similarities in functionality is obtained simply by the fact that the user can decide exactly how the test vectors should look. Even noise can be emulated as long as the frequency is significantly lower than the maximum of the PG (50 MHz).

Literature

1. **Weste, Neil H. E. and Harris, David.** *CMOS VLSI Design*. s.l. : Addison Wesley, 2005. 0-321-26977-2.

2. **Agilent Technologies.** 81134A Pulse Pattern Generator, 3.35 GHz, dual-channel. *Agilent.com*. [Online] [Cited: 09 21, 2007.] http://cp.literature.agilent.com/litweb/pdf/5988-5549EN.pdf.

3. **Nicolle, Jean P.** Serial interface (RS-232). *fpga4fun.com*. [Online] http://www.fpga4fun.com/SerialInterface.html.

4. —. EPP (Enhanced Parallel Port). *fpga4fun.com*. [Online] http://www.fpga4fun.com/EPP.html.

5. **Axelson, Jan.** Parallel Port Central. *Lakeview Research*. [Online] [Cited: 09 21, 2007.] http://www.lvr.com/files/ppc1.pdf. 0-9650819-1-5.

6. **Peacock, Craig.** Interfacing the Standard Parallel Port. *beyondlogic.org*. [Online] [Cited: 09 21, 2007.] http://www.beyondlogic.org/spp/parallel.htm.

7. —. Interfacing the Enhanced Parallel Port. *beyondlogic.org*. [Online] [Cited: 09 21, 2007.] http://www.beyondlogic.org/epp/epp.htm.

8. —. Interfacing the Extended Capabilities Port. *beyondlogic.org*. [Online] [Cited: 09 21, 2007.] http://www.beyondlogic.org/ecp/ecp.htm.

9. **Chartier, Robert.** Application Architecture: An N-Tier Approach - Part 1. *15seconds.com*. [Online] [Citeret: 21. 09 2007.] http://www.15seconds.com/issue/011023.htm.

10. **XILINX.** Spartan-II 2.5V FPGA Family: Complete Data Sheet. *xilinx.com*. [Online] 08 02, 2004. [Cited: 09 21, 2007.] http://direct.xilinx.com/bvdocs/publications/ds001.pdf.

11. **Digilent, Inc.** Digilab 2 Reference Manual. *digilent.us*. [Online] 05 07, 2002. [Cited: 09 21, 2007.] https://digilent.us/Data/Products/D2/D2-rm.PDF.

12. **XILINX.** Using Block SelectRAM+ Memory in Spartan-II FPGAs. *xilinx.com*. [Online] 12 11, 2000. [Cited: 09 21, 2007.] http://www.xilinx.com/bvdocs/appnotes/xapp173.pdf.

13. **Data & Object Factory.** Design Pattern Framework For .Net. *dofactory.com*. [Online] [Citeret: 21. 09 2007.] http://www.dofactory.com/Patterns/Patterns.aspx.

14. **Whatcott, Rhett.** Timing Closure. *Xilinx.com*. [Online] 01 24, 2002. [Cited: 09 21, 2007.] http://www.xilinx.com/xlnx/xweb/xil_tx_display.jsp?iLanguageID=1&category=&sGlobalNavPick=&sSecondaryNavPick=&multPartNum=1&sTechX_ID=rw_tim_closure.

15. **Lesea, Austin.** Signal Integrity Tips and Tricks. *Xilinx.com*. [Online] 12 28, 2000. [Cited: 09 21, 2007.] http://www.xilinx.com/xlnx/xweb/xil_tx_display.jsp?iLanguageID=1&category=-1209667&sGlobalNavPick=&sSecondaryNavPick=&multPartNum=1&sTechX_ID=al_signal.

16. **Chapman, Ken.** Get Your Priorities Right - Make your design up to 50% smaller. *Xilinx.com*. [Online] 07 08, 2004. [Cited: 09 21, 2007.] http://www.xilinx.com/xlnx/xweb/xil_tx_display.jsp?iLanguageID=1&category=&sGlobalNavPick=&sSecondaryNavPick=&multPartNum=1&sTechX_ID=kc_priorities.

17. **XILINX.** LogiCORE™ FIFO Generator 4.1 User Guide. *xilinx.com*. [Online] 08 08, 2007. [Cited: 09 21, 2007.] http://www.xilinx.com/bvdocs/ipcenter/data_sheet/fifo_generator_ug175.pdf.

18. **Logix4u.** Inpout32.dll for Windows 98/2000/NT/XP. *http://logix4u.net*. [Online] [Cited: 09 21, 2007.] http://logix4u.net/Legacy_Ports/Parallel_Port/Inpout32.dll_for_Windows_98/2000/NT/XP.html.

19. **Lattice Semiconductor,Troy Scott.** *Tips to Avoid Simultaneous Switching Output (SSO) Noise Problems*. [Webcast] s.l. : Lattice Semiconductor, 29. 08 2007. http://www.journalwebcasts.com/.

20. **Ashenden, Peter J.** *The Designer's Guide to VHDL*. s.l. : Morgan Kaufmann Publishers, 2002. 1-55860-674-2.

Appendix A Test Data

Test data for simulating the serial and parallel receiver.

```
00000000
00000000
00000000
01000000
00000000
00000000
00000000
00100000
00000000
00000000
00000000
10000000
00000000
00000000
00000000
01000000
00000000
00000000
00000000
00100000
00000000
00000000
00000000
00010000
00000000
00000000
00000000
11000000
```

Appendix B Testbenches used

Serial _TB.vhd

```
--------------------------------------------------------------------------------
-- Company: DTU
-- Engineer:      Jakob Toft, s012012
--
-- Create Date:
-- Design Name:   Serial
-- Module Name:   F:/temp/project/s012012_v3/serial_TB.vhd
-- Project Name:  s012012_v3
-- Target Device: xc2s200-5pq208
-- Tool versions: ise 9.1.03i
-- Description:   Testbench for serial receiver module
--
-- VHDL Test Bench Created by ISE for module: Serial
--
-- Dependencies:  std.textio.all;
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.02 - serial logic added
-- Revision 0.03 - added capability to load test vectors from file
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_textio.all;
USE std.textio.all;

ENTITY serial_TB_vhd IS
END serial_TB_vhd;

ARCHITECTURE behavior OF serial_TB_vhd IS

  -- Component Declaration for the Unit Under Test (UUT)
  component Serial
    port(
      CLK         : in std_logic;
      RXD         : in std_logic;
      DATA_READY  : out std_logic;
      O_REG       : out std_logic_vector(7 downto 0)
    );
  end component;

  --Inputs
  signal CLK      :  std_logic  := '0';
  signal RXD      :  std_logic  := '1';

  --Outputs
  signal DATA_READY :  std_logic              := '0';
  signal O_REG      :  std_logic_vector(7 downto 0) := "00000000";

  --Internal
```

```
  signal s_clock    :  std_logic  := '0';
  constant period   : time      := 52083 ns;

begin

  -- Instantiate the Unit Under Test (UUT)
  uut: Serial
    port map(
      CLK => CLK,
      RXD => RXD,
      DATA_READY => DATA_READY,
      O_REG => O_REG
    );

  global_clock: process
  begin
    CLK <= '1', '0' after 10 ns;
    wait for 20 ns;
  end process;

  stimuli_generator  : process
    variable good    : boolean;      -- Status bit
    variable line_in : Line;       -- Line buffers
    variable RXD1    : std_logic;    -- byte buffer
    file cmdfile     : TEXT;         -- Define the file 'handle'
  begin
    RXD <= '1';

    file_open(cmdfile,"testvecs.txt",READ_MODE);
    loop
      if endfile(cmdfile) then      -- Check EOF
        wait for period*10;         -- only for readability of waveforms
        assert false
          report "End of file encountered; exiting."
          severity NOTE;
        exit;
      end if;
      RXD <= '0'; -- start bit
      wait for period;
      readline(cmdfile,line_in);    -- Read a line from the file
      next when line_in'length = 0; -- Skip empty lines

      for i in 0 to 7 loop
        read(line_in,RXD1,good);    -- Read bits from line
        assert good report "Text I/O read error" severity ERROR;
        RXD <= RXD1;
        wait for period;
      end loop;
      RXD <= '1';             -- stop bit
      wait for period;
    end loop;
  end process;
end architecture;
```

EPP_tb.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   12:12:14 08/27/2007
-- Design Name:   EPP
-- Module Name:   F:/temp/project/s012012_v3/EPP_tb.vhd
-- Project Name:  s012012_v3
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: EPP
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_textio.all;
USE std.textio.all;

ENTITY EPP_tb_vhd IS
END EPP_tb_vhd;

ARCHITECTURE behavior OF EPP_tb_vhd IS

  -- Component Declaration for the Unit Under Test (UUT)
  COMPONENT EPP
  PORT(
    CLK              : IN std_logic;
    EPP_WRITE_ENABLE : IN std_logic;
    EPP_DATA         : IN std_logic;
    EPP_DATA_BUS     : INOUT std_logic_vector(7 downto 0);
    EPP_WAIT         : OUT std_logic;
    BYTE_READY       : OUT std_logic;
    DATA_OUT         : OUT std_logic_vector(7 downto 0)
    );
  END COMPONENT;

  --Inputs
  SIGNAL CLK              :  std_logic := '0';
  SIGNAL EPP_WRITE_ENABLE :  std_logic := '1';
  SIGNAL EPP_DATA         :  std_logic := '1';

  --BiDirs
  SIGNAL EPP_DATA_BUS :  std_logic_vector(7 downto 0);

  --Outputs
  SIGNAL EPP_WAIT   :  std_logic;
  SIGNAL BYTE_READY :  std_logic;
```

```
  SIGNAL DATA_OUT   :  std_logic_vector(7 downto 0);

BEGIN

  -- Instantiate the Unit Under Test (UUT)
  uut: EPP PORT MAP(
    CLK => CLK,
    EPP_WRITE_ENABLE => EPP_WRITE_ENABLE,
    EPP_DATA_BUS => EPP_DATA_BUS,
    EPP_WAIT => EPP_WAIT,
    EPP_DATA => EPP_DATA,
    BYTE_READY => BYTE_READY,
    DATA_OUT => DATA_OUT
  );

  global_clock: process
  begin
    CLK <= '1', '0' after 10 ns;
    wait for 20 ns;
  end process;

  stimuli_generator : process
    variable good       : boolean;              -- Status bit
    variable line_in    : Line;                 -- Line buffers
    variable tmp        : std_logic;            -- byte buffer
    file cmdfile        : TEXT;                 -- Define the file 'handle'
  begin

    file_open(cmdfile,"testvecs.txt",READ_MODE);
    loop
      if endfile(cmdfile) then                  -- Check EOF
        wait for 666 us;   -- only added for readability of wave form
        assert false
          report "End of file encountered; exiting."
          severity NOTE;
        exit;
      end if;
      EPP_WRITE_ENABLE <= '0';                    -- start handshake
      EPP_DATA <= '0';
      readline(cmdfile,line_in);                  -- Read a line from the file
      next when line_in'length = 0;               -- Skip empty lines

      for i in 0 to 7 loop
        read(line_in,tmp,good);                   -- Read bits from line
        assert good report "Text I/O read error" severity ERROR;
        EPP_DATA_BUS(i) <= tmp;
      end loop;

      wait until rising_edge(EPP_WAIT);
      EPP_WRITE_ENABLE <= '1';                    -- finish handshake
      EPP_DATA <= '1';
      wait until falling_edge(EPP_WAIT);          -- handshake done
      wait for 586 ns;
    end loop;
  end process;
END;
```

Top_tb.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   19:29:20 07/19/2007
-- Design Name:   top
-- Module Name:   F:/temp/project/s012012_v2/s012012_tb.vhd
-- Project Name:  s012012_v2
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: top
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY s012012_tb_vhd IS
END s012012_tb_vhd;

ARCHITECTURE structure OF s012012_tb_vhd IS

  -- Component Declaration for the Unit Under Test (UUT)
  COMPONENT top
  PORT(
    BTN_SOUTH, CLK_50M  : in    STD_LOGIC;
    RS232_DCE_RXD       : in    STD_LOGIC;
    PINOUT              : out   STD_LOGIC_VECTOR(31 downto 0);
    PINOUT_CLK          : out   STD_LOGIC;
    WE,CS,OE            : out   STD_LOGIC;
    ADDRESS             : out   STD_LOGIC_VECTOR (5 downto 0);
    DATA                : inout   STD_LOGIC_VECTOR (7 downto 0)
    );
  END COMPONENT;

   COMPONENT clock
      GENERIC (period:    TIME := 20 ns);
      PORT (clk:      OUT std_logic := '0');
   END COMPONENT;

  type test_array is array (0 to 21) of std_logic_vector(7 downto 0);
  constant testmatrix : test_array := ( "00000010",
                           "00000000","00000000","00000000","00000100",
                           "00000000","00000000","00000000","00000001",
                           "00000000","00000000","00000000","00000010",
                           "00000000","00000000","00000000","00000100",
                           "00000000","00000000","00000000","00001000",
                           "00000011"
```

```
                          );

  --Inputs
  SIGNAL BTN_SOUTH :  std_logic := '0';
  SIGNAL CLK_50M   :  std_logic := '0';
  SIGNAL t_clock   :  std_logic := '0';
  SIGNAL RS232_DCE_RXD :  std_logic := '0';

  signal c1,c2 : integer := 0;

  --Outputs
  SIGNAL PINOUT : STD_LOGIC_VECTOR(31 downto 0) :=
"00000000000000000000000000000000";

BEGIN
    BTN_SOUTH <= '1', '0' after 200 ns;

    tb : process(t_clock, BTN_SOUTH)
    begin
      if (BTN_SOUTH = '1') then
        RS232_DCE_RXD <= '1';
      elsif (t_clock'event) and (t_clock='1') then
        RS232_DCE_RXD <= '1';
        if(c2 >= 22) then
          null;
          --c2 <= 0;
        else
          if(c1 = 0) then
            RS232_DCE_RXD <= '0';
            c1 <= c1 + 1;
          elsif(c1 > 0 and c1 < 9) then
            RS232_DCE_RXD <= testmatrix(c2)(c1-1);
            c1 <= c1 + 1;
          elsif(c1 >= 9) then
            RS232_DCE_RXD <= '1';
            c1 <= 0;
            c2 <= c2 + 1;
          end if;
        end if;
      end if;
    end process;

  -- Instantiate the Unit Under Test (UUT)
  uut: top PORT MAP(
    BTN_SOUTH => BTN_SOUTH,
    CLK_50M => CLK_50M,
    RS232_DCE_RXD => RS232_DCE_RXD,
    PINOUT => PINOUT
  );

  theclock: clock
    PORT MAP (clk=>CLK_50M);


  anotherclock: process
  begin
    t_clock <= '1', '0' AFTER 26041 ns;
    WAIT FOR 52083 ns;
  end process;

END structure;
```

Appendix C PG Software Source Code

ConnectionFacade DLL

FConnection.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using S012012.ConnectionInterface;
using S012012.Connections;
using System.Collections;
using System.Reflection;
using System.CodeDom;
using System.Windows.Forms;
using System.IO;
using System.Drawing;
using System.Text.RegularExpressions;

namespace S012012.ConnectionFacade
{
    public class FConnection
    {
        private static FConnection instance;
        private List<String> _connectionList = new List<String>();
        private List<object> _objectConnectionList = new List<object>();
        public List<String> GetConnectionList
        {
            get
            {
                return _connectionList;
            }
        }

        protected FConnection()
        {
            // Find all the assemblies in the Add-ins directory:
            string AddInsDir = string.Format("{0}/Addins",Application.StartupPath);
            string[] assemblies = Directory.GetFiles(AddInsDir,"*.dll");
            foreach (string assemblyFile in assemblies)
            {
                try
                {
                    Assembly asm = Assembly.LoadFrom(assemblyFile);
                    // Find and install command handlers from the assembly.

                    foreach (Type type in asm.GetTypes())
                    {
                        if (type.IsClass == true && type.GetInterface(
                                "S012012.ConnectionInterface.IConnection") != null)
                        {
                            object ibaseObject = Activator.CreateInstance(type);
                            _objectConnectionList.Add(ibaseObject);
                            _connectionList.Add((string)type.InvokeMember(
                                "GetConID", BindingFlags.Default | BindingFlags
                                .InvokeMethod, null, ibaseObject,
                                new object[] { }));
                        }
                    }
                }
                catch
                {
                    MessageBox.Show("The program couldn't load the chosen .dll.",
                        "Connector Error", MessageBoxButtons.OK, MessageBoxIcon
                        .Exclamation);
```

```
                }
            }
        }

        // provide singleton instance
        public static FConnection Instance()
        {
            if (instance == null)
            {
                instance = new FConnection();
            }
            return instance;
        }

        // Cleans the string for comments/keywords and replaces commands with hex
        // code, then convert a string of hex digits (ex: E4 CA) to a byte array.
        // The string containing the hex digits and command words (without
        // spaces carriage returns or linefeeds).
        // Returns an array of bytes.
        private byte[] ConvToBytes(string s)
        {
            s = Regex.Replace(s, @"#(.+)", "");// filter out comments marked with #
            s = s.Replace(" ", "");            // filter out space characters
            // replace keywords with their hex instruction
            s = s.Replace("load", "00000002");
            s = s.Replace("clear", "00000000");
            s = s.Replace("stop", "00000004");
            s = s.Replace("run", "00000003");
            s = s.Replace("\n", ""); // filter out newline characters
            // convert hex string to byte array
            byte[] buffer = new byte[s.Length / 2];
            for (int i = 0; i < s.Length; i += 2)
                buffer[i / 2] = (byte)Convert.ToByte(s.Substring(i, 2), 16);
            return buffer;
        }

        // sends byte array to the connection specified.
        public void Send(string data, string conID)
        {
            // convert to bytes
            object[] arg = new object[] { ConvToBytes(data) };

            foreach (object o in _objectConnectionList)
            {
                // identify connection
                if (((string)o.GetType().InvokeMember("GetConID", BindingFlags
                    .Default | BindingFlags.InvokeMethod, null, o,
                    new object[] { })) == conID)
                {
                    // Send data
                    o.GetType().InvokeMember("Send", BindingFlags.Default |
                        BindingFlags.InvokeMethod, null, o, arg);
                }
            }
        }
    }
}
```

ConnectionInterface DLL

IConnection.cs

```
using System.Windows.Forms;
namespace S012012.ConnectionInterface
{
    public interface IConnection
    {
        SenderResult Send(byte[] data);
        string GetConID();
    }
}
```

SenderResult.cs

```
using System;

namespace S012012.ConnectionInterface
{
    public struct SenderResult
    {
        public String StatusMessage;
        public int StatusID;
    }
}
```

The Frontend

FlickerFreeRichEditTextBox.cs

```
using System;
using System.Windows.Forms;

namespace Frontend
{
    public class FlickerFreeRichEditTextBox : RichTextBox
    {
        const short WM_PAINT = 0x00f;
        public FlickerFreeRichEditTextBox(){}
        public static bool _Paint = true;
        // WndProc intercepts the message
        protected override void WndProc(ref System.Windows.Forms.Message m){
            if (m.Msg == WM_PAINT){
                // a paint is attempted from the system
                if(_Paint)
                    // paint away
                    base.WndProc(ref m);
                else
                    // dont paint, control is being updated
                    m.Result = IntPtr.Zero;
            }
            else
                // do what youre supposed to
                base.WndProc (ref m);
        }
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace Frontend
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using S012012.ConnectionFacade;
using System.Text.RegularExpressions;

namespace Frontend
{
    public partial class Form1 : Form
    {
        private OpenFileDialog dlgOpenFile;
        private SaveFileDialog dlgSaveFile;
        private FConnection myCon;

        //used when create new document to add a number to the document name like
        //"myDoc0.txt", so when you create another one it will be "myDoc1.txt"
        private int m_intDocNumber = 0;

        //Hold the file name that created or choosed
        //for the Save action, not "Save As..."
        private string m_strFileName = "";

        //Hold the DialogResult enum result
        //Ok or Cancel
        private DialogResult dlgResult;

        //Stream to write to the file
        private StreamWriter m_sw;

        //Check if the file is modified or not
        private bool m_bModified = false;

        public Form1()
        {
```

```
            InitializeComponent();
            // instatiate the connections
            myCon = FConnection.Instance();
            ToolStripMenuItem tmpItem;
            // populate the connection dropdown list
            foreach (string s in myCon.GetConnectionList)
            {
                tmpItem = new ToolStripMenuItem();
                tmpItem.CheckOnClick = true;
                tmpItem.Text = s;
                toolStripDropDownButton1.DropDownItems.Add(tmpItem);
            }
        }

        private void openToolStripMenuItem_Click(object sender, EventArgs e)
        {
            //Check if the current document is modified to save it
            CheckChanged(sender, e);

            // dlgOpenFile is an OpenFileDialog object
            dlgResult = dlgOpenFile.ShowDialog();
            if (dlgResult == DialogResult.Cancel)
                return;

            try
            {
                // get the file name
                m_strFileName = dlgOpenFile.FileName;
                // open to read that file
                StreamReader sr = new StreamReader(m_strFileName);
                // read the whole file
                txtBody.Text = sr.ReadToEnd();
                sr.Close();
                m_bModified = false;
            }
            catch (Exception err)
            {
                MessageBox.Show(err.Message, "Error", MessageBoxButtons
                    .OK, MessageBoxIcon.Error);
            }
        }

        private void CheckChanged(object sender, EventArgs e)
        {
            if(m_bModified)
            {
                dlgResult = MessageBox.Show("Do you want to save?"
                    ,"Note",MessageBoxButtons.YesNo,MessageBoxIcon.Exclamation);
                if(dlgResult == DialogResult.Yes)
                {
                    // Save the current work document
                    saveToolStripMenuItem_Click(sender, e);
                }
            }
        }

        // The TextChanged Event for the txtBody Control.
        // implements syntax highlighting.
        private void txtBody_TextChanged(object sender, System.EventArgs e)
        {
            m_bModified = true; //Document has been modified.

            // Calculate the starting position of the current line.
            int start = 0, end = 0;
            for (start = txtBody.SelectionStart; start > 0; start--)
```

```
{
    if (txtBody.Text[start-1] == '\n') { break; }
}

// Calculate the end position of the current line.
for (end = txtBody.SelectionStart; end < txtBody.Text.Length; end++)
{
    if (txtBody.Text[end] == '\n') break;
}

// Extract the current line that is being edited.
String line = txtBody.Text.Substring(start, end - start);

// Backup the users current selection point.
int selectionStart = txtBody.SelectionStart;
int selectionLength = txtBody.SelectionLength;

// Split the line into tokens.
Regex r = new Regex("([ \\t{}();])");
string[] tokens = r.Split(line);
int index = start;
FlickerFreeRichEditTextBox._Paint = false; // disable WM_PAINT messages
foreach (string token in tokens)
{

    // Set the token's default color and font.
    txtBody.SelectionStart = index;
    txtBody.SelectionLength = token.Length;
    txtBody.SelectionColor = Color.Black;
    txtBody.SelectionFont = new Font(txtBody.Font
        .FontFamily, txtBody.Font.Size, FontStyle.Regular);

    // Check for a comment.
    if (token == "#" || token.StartsWith("#"))
    {
        // Find the start of the comment
        // and then extract the whole comment.
        int length = line.Length - (index - start);
        string commentText = txtBody.Text.Substring(index, length);
        txtBody.SelectionStart = index;
        txtBody.SelectionLength = length;
        // color the text
        txtBody.SelectionColor = Color.DarkGreen;
        break;
    }

    // Check whether the token is a keyword.
    String[] keywords = { "load", "run", "stop", "clear"};
    for (int i = 0; i < keywords.Length; i++)
    {
        if (keywords[i] == token.ToLower())
        {
            // Apply color and bold to highlight keyword.
            txtBody.SelectionColor = Color.Blue;
            txtBody.SelectionFont = new Font(txtBody.Font
                .FontFamily, txtBody.Font.Size, FontStyle.Bold);
            break;
        }
    }
    index += token.Length;
}
// start painting the control again
```

```
        FlickerFreeRichEditTextBox._Paint = true;

        // Restore the users current selection point.
        txtBody.SelectionStart = selectionStart;
        txtBody.SelectionLength = selectionLength;
    }

    private void newToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // Check if document has been changed.
        CheckChanged(sender,e);

        // cleare the file name to avoid overwriting old documents

        m_strFileName = "";

        // cleare the editor area.
        txtBody.Clear();

        // new document means not modified yet.
        m_bModified = false;

        // do not replace the m_bModified position in this method
        // beacuse Clear() Method invoke the TextChanged event
    }

    private void saveToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // Check if for the file name to save on the current
        // working document or create new one
        if(m_strFileName == "")
        {
            // Create new one
            saveAsToolStripMenuItem_Click(sender,e);
            return;
        }
        else
        {
            // Save on the current document
            m_sw = new StreamWriter(m_strFileName);
            m_sw.Write (txtBody.Text);
            m_sw.Close();
        }
        m_bModified = false;
    }

    private void saveAsToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // dlgSaveFile is a SaveFileDialog object created from the toolbox
        // like mydoc0.txt
        dlgSaveFile.FileName = "mydoc" + m_intDocNumber.ToString() + ".txt";
        dlgResult = dlgSaveFile.ShowDialog();

        if(dlgResult == DialogResult.Cancel)
            return;

        try
        {
            // Saving our file name for Quick Save
            m_strFileName = dlgSaveFile.FileName;

            // Create new StreamWriter to write the text to it
            m_sw = new StreamWriter(m_strFileName);
            m_sw.Write (txtBody.Text);
```

```
            // close resources when done
            m_sw.Close();
            saveToolStripMenuItem.Enabled = true;

            // increasing document number to avoid overwriting old docs.
            m_intDocNumber++;
            m_bModified = false; //document is saved
            this.Text = m_strFileName;
        }
        catch(Exception err)
        {
            MessageBox.Show(err.Message,"Error", MessageBoxButtons
                .OK,MessageBoxIcon.Error);
        }
    }

    // Exit the program
    private void exitToolStripMenuItem_Click(object sender, EventArgs e)
    {
        CheckChanged(sender, e);
        this.Dispose();
    }

    // Create a new file
    private void newToolStripButton_Click(object sender, EventArgs e)
    {
        newToolStripMenuItem_Click(sender, e);
    }

    // open file
    private void openToolStripButton_Click(object sender, EventArgs e)
    {
        openToolStripMenuItem_Click(sender, e);
    }

    // Save file
    private void saveToolStripButton_Click(object sender, EventArgs e)
    {
        saveToolStripMenuItem_Click(sender, e);
    }

    // save data before quitting
    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
        CheckChanged(sender, e);
    }

    // dropdown menu click handler
    private void toolStripDropDownButton1_DropDownItemClicked(
        object sender, ToolStripItemClickedEventArgs e)
    {
        // check or uncheck the clicked menu item
        foreach (ToolStripMenuItem t in toolStripDropDownButton1.DropDownItems)
        {
            if (t != e.ClickedItem)
            {
                t.Checked = false;
            }
        }
    }

    // the Run button
```

```
        private void runToolStripButton_Click(object sender, EventArgs e)
        {
            foreach (ToolStripMenuItem t in toolStripDropDownButton1.DropDownItems)
            {
                // identify the chosen connection type
                if (t.Checked == true)
                {
                    // send the content of the text field
                    myCon.Send(txtBody.Text, t.Text);
                }
            }
        }

        // select all menu item
        private void selectAllToolStripMenuItem_Click(object sender, EventArgs e)
        {
            txtBody.SelectAll();
        }

        // cut menu item
        private void cutToolStripMenuItem_Click(object sender, EventArgs e)
        {
            // copy data to the clipboard and clear the text field
            Clipboard.SetDataObject(txtBody.SelectedText);
            txtBody.SelectedText = "";
        }

        // copy menu item
        private void copyToolStripMenuItem_Click(object sender, EventArgs e)
        {
            txtBody.Copy();
        }

        // paste menu item
        private void pasteToolStripMenuItem_Click(object sender, EventArgs e)
        {
            txtBody.Paste();
        }
    }
}
```

Connection DLL

TCPIP.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
using S012012.ConnectionInterface;

namespace S012012.Connections
{
    public class TCPIP : IConnection
    {
        string conID = "TCPIP";

        static SenderResult Connect(byte[] data)
        {
            SenderResult sr = new SenderResult();
            sr.StatusID = 0;
            sr.StatusMessage = "OK";
            try
            {
                // open a connection to localhost port 13000
                Int32 port = 13000;
                TcpClient client = new TcpClient("localhost", port);
                NetworkStream stream = client.GetStream();

                // Send the message to the connected TcpServer.
                stream.Write(data, 0, data.Length);

                // Close everything.
                stream.Close();
                client.Close();
            }
            catch (ArgumentNullException)
            {
                sr.StatusID = 1;
                sr.StatusMessage = "No argument";
            }
            catch (SocketException)
            {
                sr.StatusID = 2;
                sr.StatusMessage = "Socket Error";
            }
            return sr;
        }

        public SenderResult Send(byte[] data)
        {
            return Connect(data);
        }

        public string GetConID()
        {
            return this.conID;
        }
    }
}
```

Serial.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.IO.Ports;
using S012012.ConnectionInterface;
using System.Windows.Forms;

namespace S012012.Connections
{
//    public class Serial : ToolStripMenuItem, IConnection
    public class Serial : IConnection
    {
        string conID = "Serial";

        static SenderResult Connect(byte[] data)
        {
            SenderResult sr = new SenderResult();
            sr.StatusID = 0;
            sr.StatusMessage = "OK";

            try
            {
                // Instantiate the communications port with some basic settings
                SerialPort port = new SerialPort(
                    "COM1", 19200, Parity.None, 8, StopBits.One);

                // Open the port for communications
                port.Open();
                // Write the set of bytes
                port.Write(data, 0, data.Length);

                // Close the port
                port.Close();
            }
            catch (IOException)
            {
                sr.StatusID = 1;
                sr.StatusMessage = "IO Error";
            }
            return sr;
        }

        public SenderResult Send(byte[] data)
        {
            return Connect(data);
        }

        public string GetConID()
        {
            return this.conID;
        }
    }
}
```

Parallel.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using S012012.ConnectionInterface;
using System.Globalization;

namespace S012012.Connections
{
    public class Parallel : IConnection
    {
        string conID = "Parallel"; // identify DLL
        static int BASE_ADR = 888; // EPP Base address

        public Parallel()
        {
            // initiallize the parallel port
            PortAccess.Output(BASE_ADR + 2, 4);
        }

        // Send the data
        static SenderResult Connect(byte[] data)
        {
            SenderResult sr = new SenderResult();
            sr.StatusID = 0;
            sr.StatusMessage = "OK";

            try
            {
                foreach (byte b in data)
                {
                    // write byte
                    PortAccess.Output(BASE_ADR + 4, b);
                }
            }
            catch (Exception)
            {
                sr.StatusID = 1;
                sr.StatusMessage = "IO Error";
            }
            return sr;
        }

        public SenderResult Send(byte[] data)
        {
            return Connect(data);
        }

        // provide identification of the DLL
        public string GetConID()
        {
            return this.conID;
        }
    }
}
```

PortAccess.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;

namespace S012012.Connections
{
    public class PortAccess
    {
        [DllImport("inpout32.dll", EntryPoint = "Out32")]
        public static extern void Output(int adress, int value);
    }
}
```

Appendix D TCP/IP listener Source Code

Program.cs:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace test_tcpip
{
    class MyTcpListener
    {
        public static void Main()
        {
            TcpListener server = null;
            try
            {
                // Set the TcpListener on Localhost port 13000.
                Int32 port = 13000;
                IPAddress localAddr = IPAddress.Parse("127.0.0.1");
                server = new TcpListener(localAddr, port);

                // Start listening for client requests.
                server.Start();

                // Buffer for reading data
                Byte[] bytes = new Byte[256];
                String data = null;

                // Enter the listening loop.
                while (true)
                {
                    Console.Write("Waiting for a connection... ");

                    // Wait for connection
                    TcpClient client = server.AcceptTcpClient();
                    Console.WriteLine("Connected!");

                    data = null;

                    // Get a stream object for reading and writing
                    NetworkStream stream = client.GetStream();

                    int i;

                    // Loop to receive all the data sent by the client.
                    while ((i = stream.Read(bytes, 0, bytes.Length)) != 0)
                    {
                        // Translate data bytes to a ASCII for output.
                        data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);
                        Console.WriteLine("Received: {0}", data);
                    }

                    // Shutdown and end connection
                    client.Close();
                }
            }
            catch (SocketException e)
            {
                Console.WriteLine("SocketException: {0}", e);
```

```
            }
            finally
            {
                // Stop listening for new clients.
                server.Stop();
            }
        }
    }
}
```

Appendix E PG Base Design (VHDL)

Top.vhd:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:        Jakob Toft, s012012
--
-- Create Date:     09:46:05 06/11/2007
-- Design Name:     Top entity of the PG
-- Module Name:     top - Behavioral
-- Project Name:    Pattern Generator
-- Target Devices:  Spartan 2
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.02 - Serial Receiver code added
-- Revision 0.03 - Bus interface to DIO2 board added
-- Revision 0.04 - Counter LED's added
-- Revision 0.05 - Switches added to bus interface
-- Revision 0.06 - Removed Counter LED's, serial receiver is now working.
-- Revision 0.10 - Moved design from spartan II to Spartan 3A
-- Revision 0.11 - Made the design Structural and removed busmaster code
-- Revision 0.12 - Added busmaster code as module
-- Revision 0.13 - Added controller
-- Revision 0.14 - Added memory block
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.types.ALL;

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

entity top is
  Port (
    CLK_50M       : in  STD_LOGIC;
    RS232_DCE_RXD : in  STD_LOGIC;
    PINOUT        : out STD_LOGIC_VECTOR(31 downto 0)
  );
end top;

architecture Structural of top is

  component Serial
    port(
      CLK        : in  STD_LOGIC;
      RXD        : in  STD_LOGIC;
      DATA_READY : out STD_LOGIC;
      O_REG      : out STD_LOGIC_VECTOR (7 downto 0)
    );
  end component;
```

```
  component Controller
    port(
      CLK        : in  STD_LOGIC;
      DATA_IN    : in  STD_LOGIC_VECTOR (7 downto 0);
      DATA_READY : in  STD_LOGIC;
      DATA_OUT   : out STD_LOGIC_VECTOR (31 downto 0);
      MEM_ADR    : out STD_LOGIC_VECTOR (9 downto 0);
      RW_CNTL    : out STD_LOGIC_VECTOR (0 downto 0);
      O_EN       : out STD_LOGIC
    );
  end component;

  component mymem
    port (
      CLK  : in  STD_LOGIC;
      DIN  : in  STD_LOGIC_VECTOR(31 downto 0);
      ADDR : in  STD_LOGIC_VECTOR(9 downto 0);
      WE   : in  STD_LOGIC;
      DOUT : out STD_LOGIC_VECTOR(31 downto 0)
    );
  end component;

  -- Signal declarations
  signal dataready  : STD_LOGIC := '0';
  signal dataline   : STD_LOGIC_VECTOR (7 downto 0) := "00000000";

  signal dina     : STD_LOGIC_VECTOR(31 downto 0) := word_zero;
  signal addra    : STD_LOGIC_VECTOR(9 downto 0) := "0000000000";
  signal wea      : STD_LOGIC_VECTOR(0 downto 0) := "0";
  signal douta    : STD_LOGIC_VECTOR(31 downto 0) := word_zero;
  signal oea      : STD_LOGIC := '0';

begin

  SerialC_Component : Serial
    port map(
      CLK => CLK_50M,
      RXD => RS232_DCE_RXD,
      DATA_READY => dataready,
      O_REG => dataline
    );

  Controller_Component : Controller
    port map(
      CLK => CLK_50M,
      DATA_IN => dataline,
      DATA_READY => dataready,
      DATA_OUT => dina,
      MEM_ADR => addra,
      RW_CNTL => wea,
      O_EN => oea
    );

  myMem_Component : mymem
    port map (
      CLK => CLK_50M,
      DIN => dina,
      ADDR => addra,
      WE => wea(0),
      DOUT => douta
    );

  -- Logic
  process(CLK_50M)
```

```
  begin
    if CLK_50M'event and CLK_50M = '1' then
      if(oea = '1') then
        PINOUT <= douta;
      end if;
    end if;
  end process;
end Structural;
```

controller.vhd:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    13:49:08 07/06/2007
-- Design Name:
-- Module Name:    Controller - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.02 - FSM implemented
-- Revision 0.03 - Rewrote clock dividers as clock enables

-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.types.ALL;

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

entity Controller is
  Port (
    CLK        : in  STD_LOGIC;
    DATA_IN    : in  STD_LOGIC_VECTOR (7 downto 0);
    DATA_READY : in  STD_LOGIC;
    DATA_OUT   : out STD_LOGIC_VECTOR (31 downto 0);
    MEM_ADR    : out STD_LOGIC_VECTOR (9 downto 0);
    RW_CNTL    : out STD_LOGIC_VECTOR (0 downto 0);
    O_EN       : out STD_LOGIC
  );
end Controller;

architecture Behavioral of Controller is

  signal stall        : STD_LOGIC_VECTOR (31 downto 0) := word_zero;
  signal s_count      : STD_LOGIC_VECTOR (31 downto 0) := word_zero;
  signal Count4       : INTEGER := 0;
  signal Count5       : INTEGER := 0;
```

```
  signal Count6       : STD_LOGIC_VECTOR (4 downto 0) := "00000";
  signal FQD          : STD_LOGIC_VECTOR (4 downto 0) := "00001";
  signal running      : STD_LOGIC := '0';
  signal write        : STD_LOGIC := '0';
  signal clkd         : STD_LOGIC := '0';
  signal CLK_MODIFIED : STD_LOGIC := '0';
  signal sbreg      : STD_LOGIC_VECTOR (23 downto 0) := "000000000000000000000000";
  signal data       : STD_LOGIC_VECTOR (31 downto 0) := word_zero;
  signal memadr     : STD_LOGIC_VECTOR (9 downto 0) := "0000000000";
  signal memadr2    : STD_LOGIC_VECTOR (9 downto 0) := "0000000000";
  signal State      : STD_LOGIC_VECTOR(2 downto 0) := "000";

begin

  -- clock enable
  -- (50 MHZ/(19200 * 8) Baud) / (2+2) = 81 clock cycles.
  process (CLK)
    begin
    if CLK'event and CLK = '1' then
      Count4 <= Count4 + 1;
      if Count4 = 163 then
        Count4 <= 0;
        clkd<= '1';
      else
        clkd<= '0';
      end if;
    end if;
  end process;

  -- the user defined clock enable
  process (CLK)
    begin
    if CLK'event and CLK = '1' then
      Count6 <= Count6 + 1;
      if Count6 >= FQD then
        Count6 <= "00001";
        CLK_MODIFIED <= '1';
      else
        CLK_MODIFIED <= '0';
      end if;
    end if;
  end process;

  process(CLK)
  begin
    if CLK'event and CLK = '1' then
      if clkd = '1' then
        case State is
          when "000" => -- idle stage
                    if DATA_READY = '0' then State <= "000";
                    else State <= "001"; end if;
          when "001" => -- decode stage
                    if DATA_IN(2 downto 0) = LOAD then State <= "011";
                    else State <= "010"; end if;
          when "010" =>  -- execute stage
                    if DATA_IN(2 downto 0) = START then
                      running <= '1';
                    elsif DATA_IN(2 downto 0) = STOP then
                      running <= '0';
                    elsif DATA_IN(2 downto 0) = CLEAR then
                      memadr <= "0000000000";
                      s_count <= word_zero;
                      stall <= word_zero;
                      running <= '0';
                      Count5 <= 0;
```

```
--                      FQD <= "00001";
                      State <= "000";
                    elsif DATA_IN(2 downto 0) = SETCLOCK then
                      FQD <= DATA_IN(7 downto 3);
                    end if;
                    State <= "000";
          when "011" =>  -- load detected, collect length of data segment
                    if DATA_READY = '0' then
                      State <= "011";
                    else
                      stall(31 downto 24) <= DATA_IN;
                      State <= "100";
                    end if;
          when "100" => -- collect length of data segment
                    if DATA_READY = '0' then
                      State <= "100";
                    else
                      stall(23 downto 16) <= DATA_IN;
                      State <= "101";
                    end if;
          when "101" => -- collect length of data segment
                    if DATA_READY = '0' then
                      State <= "101";
                    else
                      stall(15 downto 8) <= DATA_IN;
                      State <= "110";
                    end if;
          when "110" => -- collect length of data segment
                    if DATA_READY = '0' then
                      State <= "110";
                    else
                      stall(7 downto 0) <= DATA_IN;
                      State <= "111";
                    end if;
          when "111" => -- grab the data segment
                    if DATA_READY = '0' then
                      if(s_count < stall) then
                        State <= "111";
                        write <= '0';
                      else
                        State <= "000";
                        write <= '0';
                      end if;
                    else
                      if(s_count < stall) then
                        State <= "111";
                        sbreg <= sbreg(15 downto 0) & DATA_IN;
                        if Count5 = 3 then
                          Count5 <= 0;
                          s_count <= s_count + 1;
                          memadr <= memadr + 1;
                          write <= '1';
                          data <= sbreg(23 downto 0) & DATA_IN;
                        else
                          write <= '0';
                          Count5 <= Count5 + 1;
                        end if;
                      else
                        write <= '0';
                        State <= "000";
                      end if;
                    end if;
```

```
          when others =>  state <= "000";
        end case;
      end if;
    end if;
  end process;

-- memory stuff
-- memory controller mux
  process(CLK)
  begin
    if CLK'event and CLK = '1' then
      if(write = '1') then
        DATA_OUT <= data;
        MEM_ADR <= memadr - 1;
        RW_CNTL <= "1";
      else
        RW_CNTL <= "0";
        MEM_ADR <= memadr2;
      end if;
    end if;
  end process;

  -- read data at the defined clock speed
  process(CLK)
  begin
  if CLK'event and CLK = '1' then
    if CLK_MODIFIED = '1' then
      if running = '1' then
        if(memadr2 < stall-1) then
          memadr2 <= memadr2 + 1;
          O_EN <= '1';
        else
          O_EN <= '0';
        end if;
      elsif(running = '0') then
        memadr2 <= "0000000000";
        O_EN <= '0';
      end if;
    end if;
  end if;
  end process;
end Behavioral;
```

Types.vhd:

```
--  Package File Template
--
--  Purpose: This package defines supplemental types, subtypes,
--           constants, and functions

library IEEE;
use IEEE.STD_LOGIC_1164.all;

package types is

-- Declare types and subtypes
  subtype byte_t     is STD_LOGIC_VECTOR (7 downto 0);
  subtype halfword_t is STD_LOGIC_VECTOR (15 DOWNTO 0);
  subtype word_t     is STD_LOGIC_VECTOR (31 DOWNTO 0);

  type data_t is array (7 downto 0) of byte_t;

-- Declare constants
  constant byte_zero      : byte_t := "00000000";
```

```
  constant halfword_zero    : halfword_t := byte_zero & byte_zero;
  constant word_zero      : word_t := halfword_zero & halfword_zero;

-- Instructions
  constant CLEAR    : STD_LOGIC_VECTOR (2 downto 0) := "000";
  constant SETCLOCK : STD_LOGIC_VECTOR (2 downto 0) := "001";
  constant LOAD     : STD_LOGIC_VECTOR (2 downto 0) := "010";
  constant START    : STD_LOGIC_VECTOR (2 downto 0) := "011";
  constant STOP     : STD_LOGIC_VECTOR (2 downto 0) := "100";

end types;

package body types is
end types;
```

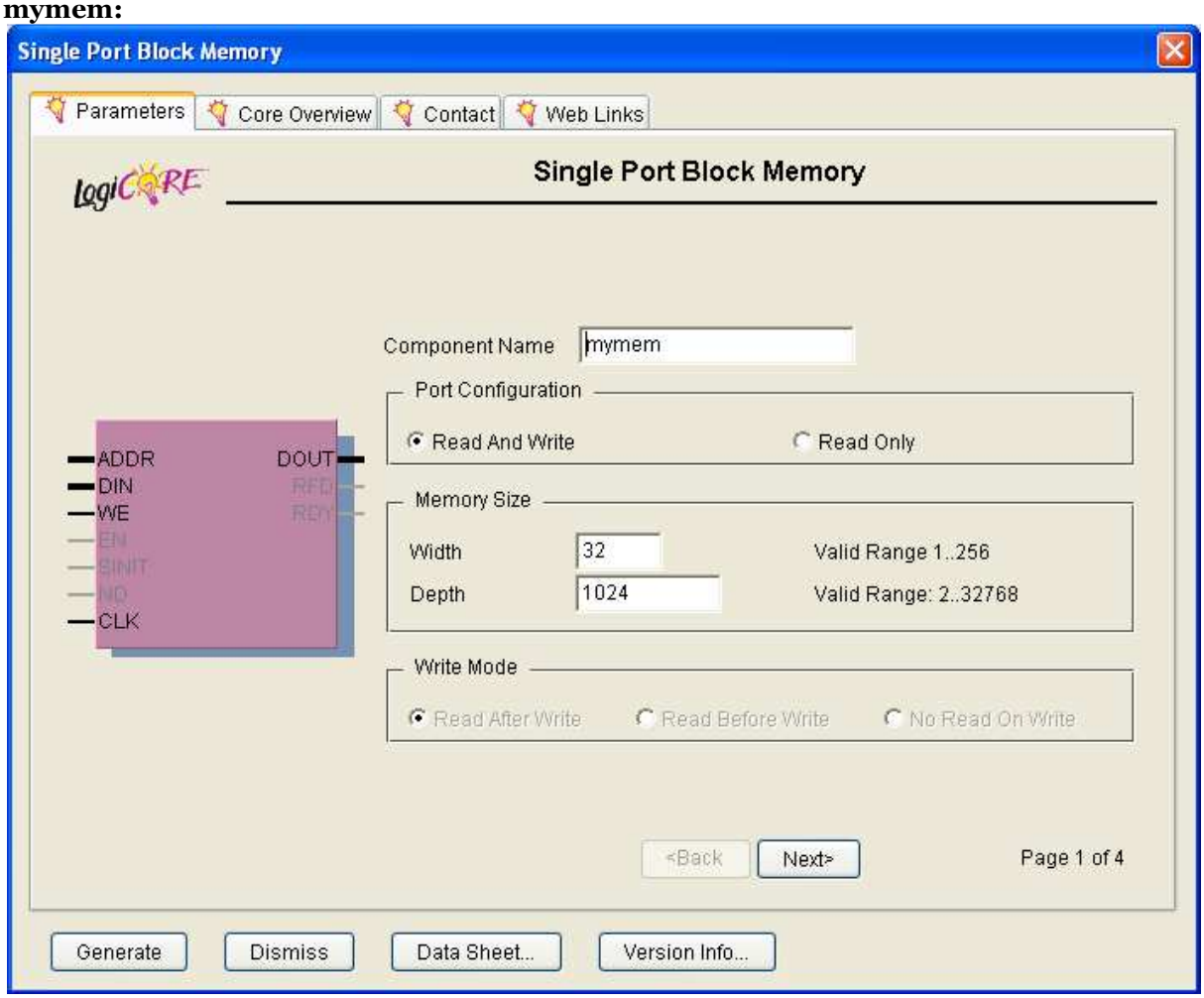
Serial:vhd:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:       Jakob Toft, s012012
--
-- Create Date:    11:34:45 07/05/2007
-- Design Name:    Receiver
-- Module Name:    Serial - Behavioral
-- Project Name:   Pattern Generator
-- Target Devices: Spartan 3A
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.02 - new implementation of the Receiver added
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.types.ALL;

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

entity Serial is
  Port (
    CLK        : in  STD_LOGIC;
    RXD        : in  STD_LOGIC;
    DATA_READY : out STD_LOGIC;
    O_REG      : out STD_LOGIC_VECTOR (7 downto 0)
  );
end Serial;

architecture Behavioral of Serial is

  -- signal declaration --
  signal clk_153600  : STD_LOGIC := '0';
  signal Counter     : INTEGER := 0;
```

```
  signal State       : STD_LOGIC_VECTOR(3 downto 0) := "0000";
  signal Sreg        : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
  signal next_bit, RXD_data_ready : STD_LOGIC := '0';
  signal RXD_bit     : STD_LOGIC := '1';
  signal RXD_sync    : STD_LOGIC_VECTOR(1 downto 0) := "11";
  signal RXD_count   : STD_LOGIC_VECTOR(1 downto 0) := "11";
  signal bit_spacing : STD_LOGIC_VECTOR(2 downto 0) := "000";

begin

  -- logic --

  -- 8x oversampling --
  -- 19200 Baud * 8 = 153600 Baud
  -- 50 MHZ/153600 Baud = 326 clock cycles.
  process (CLK)
    begin
    if CLK'event and CLK = '1' then
      Counter <= Counter + 1;
      if Counter = 163 then         -- since we only operate on positive edges
        Counter <= 0;             -- of the clock to trigger a change in the
        clk_153600 <= not clk_153600; -- sample clock we only count to 326/2 = 163.
      end if;
    end if;
  end process;

  -- oversample the incoming signal and synchronize it to the clock.
  process (clk_153600)
    begin
    if clk_153600'event and clk_153600 = '1' then
      RXD_sync <= RXD_sync(0) & RXD;
    end if;
  end process;

  -- filter RXD input
  process (clk_153600)
    begin
    if clk_153600'event and clk_153600 = '1' then
      if (RXD_sync(1) = '1' and RXD_count /= "11") then
        RXD_count <= RXD_count + 1;
      elsif (RXD_sync(1) = '0' and RXD_count /= "00") then
        RXD_count <= RXD_count - 1;
      end if;

      if (RXD_count = "00") then
        RXD_bit <= '0';
      elsif (RXD_count = "11") then
        RXD_bit <= '1';
      end if;
    end if;
  end process;

  -- state machine to grab the bits
  process (clk_153600)
    begin
    if clk_153600'event and clk_153600 = '1' then
      case State is
        when "0000" => if RXD_bit = '0' then State <= "1000"; end if;  -- start bit
        when "1000" => if next_bit = '1' then State <= "1001"; end if; -- 1st bit
        when "1001" => if next_bit = '1' then State <= "1010"; end if; -- 2nd bit
        when "1010" => if next_bit = '1' then State <= "1011"; end if; -- 3rd bit
        when "1011" => if next_bit = '1' then State <= "1100"; end if; -- 4th bit
        when "1100" => if next_bit = '1' then State <= "1101"; end if; -- 5th bit
        when "1101" => if next_bit = '1' then State <= "1110"; end if; -- 6th bit
        when "1110" => if next_bit = '1' then State <= "1111"; end if; -- 7th bit
```

```
        when "1111" => if next_bit = '1' then State <= "0001"; end if; -- 8th bit
        when "0001" => if next_bit = '1' then State <= "0000"; end if; -- stop bit
        when others => State <= "0000";
      end case;
    end if;
  end process;

  -- wait for next bit
  process (CLK, clk_153600, State)
    begin
    if State = "0000" then
      bit_spacing <= "000";
    elsif clk_153600'event and clk_153600 = '1' then
      bit_spacing <= bit_spacing + 1;
    end if;
  end process;

  -- shift register to collect the data as its sent
  process (clk_153600)
    begin
    if clk_153600'event and clk_153600 = '1' then
      if next_bit = '1' and State(3) = '1' then
        Sreg <= RXD_bit & Sreg(7 downto 1);
      end if;
    end if;
  end process;

  -- wait for stopbit and notify when the data is ready
  process (CLK)
    begin
    if CLK'event and CLK = '1' then
      if (clk_153600 = '1' and next_bit = '1' and state = "0001" and RxD_bit = '1')
then
        RXD_data_ready <= '1';
      else
        RXD_data_ready <= '0';
      end if;
    end if;
  end process;

  -- grab next bit
  next_bit <= '1' when bit_spacing = "111" else '0';

  -- output data when its ready and supply ready signal
  O_REG <= Sreg when RXD_data_ready = '1';
  DATA_READY <= RXD_data_ready;

end Behavioral;
```

mymem:



Top.ucf:

```
net "RS232_DCE_RXD"     loc = "p202";
net "CLK_50M"           loc = "p80";

net PINOUT<0>  loc = "p180";# C5  on connector C
net PINOUT<1>  loc = "p178";# C7  on connector C
net PINOUT<2>  loc = "p175";# C9  on connector C
net PINOUT<3>  loc = "p173";# C11 on connector C
net PINOUT<4>  loc = "p168";# C13 on connector C
net PINOUT<5>  loc = "p166";# C15 on connector C
net PINOUT<6>  loc = "p164";# C17 on connector C
net PINOUT<7>  loc = "p162";# C19 on connector C
net PINOUT<8>  loc = "p160";# C21 on connector C
net PINOUT<9>  loc = "p152";# C23 on connector C
net PINOUT<10> loc = "p150";# C25 on connector C
net PINOUT<11> loc = "p148";# C27 on connector C
net PINOUT<12> loc = "p146";# C29 on connector C
net PINOUT<13> loc = "p140";# C32 on connector C
net PINOUT<14> loc = "p138";# C34 on connector C
net PINOUT<15> loc = "p135";# C36 on connector C
net PINOUT<16> loc = "p133";# C38 on connector C
net PINOUT<17> loc = "p129";# C40 on connector C
net PINOUT<18> loc = "p69"; # E5  on connector E
net PINOUT<19> loc = "p67"; # E7  on connector E
net PINOUT<20> loc = "p62"; # E9  on connector E
net PINOUT<21> loc = "p60"; # E11 on connector E
```

```
net PINOUT<22> loc = "p58"; # E13 on connector E
net PINOUT<23> loc = "p49"; # E15 on connector E
net PINOUT<24> loc = "p47"; # E17 on connector E
net PINOUT<25> loc = "p45"; # E19 on connector E
net PINOUT<26> loc = "p43"; # E21 on connector E
net PINOUT<27> loc = "p41"; # E23 on connector E
net PINOUT<28> loc = "p36"; # E25 on connector E
net PINOUT<29> loc = "p34"; # E27 on connector E
net PINOUT<30> loc = "p31"; # E29 on connector E
net PINOUT<31> loc = "p29"; # E31 on connector E
```

Appendix F PG Improved Design (VHDL)

Top.vhd:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:      Jakob Toft, s012012
--
-- Create Date:     09:46:05 06/11/2007
-- Design Name:     Top entity of the PG
-- Module Name:     top - Behavioral
-- Project Name:    Pattern Generator
-- Target Devices:  Spartan 2
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.02 - Base design used as template.
-- Revision 0.03 - Byte collector added between receiver and controller.
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.types.ALL;

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

entity top is
  Port (
    CLK_50M       : in  STD_LOGIC;
    RS232_DCE_RXD : in  STD_LOGIC;
    PINOUT        : out STD_LOGIC_VECTOR(31 downto 0)
  );
end top;

architecture Structural of top is

  -- Component declarations

  component Serial
    port(
      CLK        : in  STD_LOGIC;
      RXD        : in  STD_LOGIC;
      BYTE_READY : out STD_LOGIC;
      O_REG      : out STD_LOGIC_VECTOR (7 downto 0)
    );
  end component;

  component Byte_Collector
  Port (
    CLK         : in  STD_LOGIC;
    DATA_IN     : in  STD_LOGIC_VECTOR (7 downto 0);
    DATA_OUT    : out STD_LOGIC_VECTOR (31 downto 0);
    BYTE_READY  : in  STD_LOGIC;
    WORD_READY  : out STD_LOGIC
```

```
  );
  end component;

  component Controller
    port(
      CLK        : in  STD_LOGIC;
      DATA_IN    : in  STD_LOGIC_VECTOR (31 downto 0);
      WORD_READY : in  STD_LOGIC;
      DATA_OUT   : out STD_LOGIC_VECTOR (31 downto 0);
      MEM_ADR    : out STD_LOGIC_VECTOR (9 downto 0);
      MEM_WE     : out STD_LOGIC;
      MEM_EN     : out STD_LOGIC
    );
  end component;

  component mymem
    port (
      ADDR      : IN  STD_LOGIC_VECTOR(9 downto 0);
      CLK       : IN  STD_LOGIC;
      DIN       : IN  STD_LOGIC_VECTOR(31 downto 0);
      DOUT      : OUT STD_LOGIC_VECTOR(31 downto 0);
      EN        : IN  STD_LOGIC;
      WE        : IN  STD_LOGIC
    );
  end component;

  -- Signal declarations
  signal byteready : STD_LOGIC := '0';
  signal wordready : STD_LOGIC := '0';
  signal dataline  : STD_LOGIC_VECTOR (7 downto 0) := "00000000";

  signal dina     : STD_LOGIC_VECTOR(31 downto 0) := word_zero;
  signal word     : STD_LOGIC_VECTOR(31 downto 0) := word_zero;
  signal addra    : STD_LOGIC_VECTOR(9 downto 0) := "0000000000";
  signal wea      : STD_LOGIC := '0';
  signal douta    : STD_LOGIC_VECTOR(31 downto 0) := word_zero;
  signal ena      : STD_LOGIC := '0';
begin

  SerialC_Component : Serial
    port map(
      CLK => CLK_50M,
      RXD => RS232_DCE_RXD,
      BYTE_READY => byteready,
      O_REG => dataline
    );

  Byte_Collector_Component : Byte_Collector
    port map(
      CLK => CLK_50M,
      DATA_IN => dataline,
      DATA_OUT => word,
      BYTE_READY => byteready,
      WORD_READY => wordready
    );

  Controller_Component : Controller
    port map(
      CLK => CLK_50M,
      DATA_IN => word,
      WORD_READY => wordready,
      DATA_OUT => dina,
```

```
      MEM_ADR => addra,
      MEM_WE => wea,
      MEM_EN => ena
    );

  myMem_Component : mymem
    port map (
      ADDR => addra,
      CLK => CLK_50M,
      DIN => dina,
      DOUT => douta,
      EN => ena,
      WE => wea
    );

  -- Logic
  process(CLK_50M)
  begin
    if CLK_50M'event and CLK_50M = '1' then
      if wea = '0' and ena = '1' then
        PINOUT <= douta;
      end if;
    end if;
  end process;
end Structural;
```

controller.vhd:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    13:49:08 07/06/2007
-- Design Name:
-- Module Name:    Controller - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.02 - Base design used as template
-- Revision 0.03 - new optimized FSM using 32 bit inputs.

-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.types.ALL;

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

entity Controller is
  Port (
```

```
    CLK       : in  STD_LOGIC;
    DATA_IN   : in  STD_LOGIC_VECTOR (31 downto 0);
    WORD_READY: in  STD_LOGIC;
    DATA_OUT  : out STD_LOGIC_VECTOR (31 downto 0);
    MEM_ADR   : out STD_LOGIC_VECTOR (9 downto 0);
    MEM_WE    : out STD_LOGIC;
    MEM_EN    : out STD_LOGIC
  );
end Controller;

architecture Behavioral of Controller is

  signal stall              : STD_LOGIC_VECTOR (31 downto 0) := word_zero;
  signal fe_count           : STD_LOGIC_VECTOR (28 downto 0) :=
"00000000000000000000000000000";
  signal FQD                : STD_LOGIC_VECTOR (28 downto 0) :=
"00000000000000000000000000001";
  signal CLK_MODIFIED       : STD_LOGIC               := '0';
  signal writeadr           : STD_LOGIC_VECTOR (9 downto 0)   := "0000000000";
  signal readadr            : STD_LOGIC_VECTOR (9 downto 0)   := "0000000000";
  signal WORD_READY_DELAYED : STD_LOGIC := '0';

  type state_type is
(IDLE_STATE,DECODE_STATE,RUNNING_STATE,COLLECTSTALL_STATE,WRITE_STATE);
  signal state, next_state: state_type;

begin

  -- the user defined clock enable
  process (CLK)
    begin
    if CLK'event and CLK = '1' then
      fe_count <= fe_count + 1;
      if fe_count >= FQD then
        fe_count <= "00000000000000000000000000000";
        CLK_MODIFIED <= '1';
      else
        CLK_MODIFIED <= '0';
      end if;
    end if;
  end process;

  -- 3 process FSM
  state_register: process(CLK)
  begin
    if (CLK'Event and CLK = '1') then
      state <= next_state;
    end if;
  end process state_register;

  nextstate_function : process (state, DATA_IN, WORD_READY, writeadr, stall)
  begin
    case state is
      when IDLE_STATE =>          -- IDLE
        next_state <= IDLE_STATE;
        if WORD_READY = '1' then
          next_state <= DECODE_STATE;
        end if;
      when DECODE_STATE =>
        case DATA_IN(2 downto 0) is
          when CLEAR =>
            next_state <= DECODE_STATE;
```

```
          if WORD_READY = '0' then
            next_state <= IDLE_STATE;
          end if;
        when SETCLOCK =>
          next_state <= DECODE_STATE;
          if WORD_READY = '0' then
            next_state <= IDLE_STATE;
          end if;
        when RUN =>
          next_state <= RUNNING_STATE;
        when LOAD =>
          next_state <= DECODE_STATE;
          if WORD_READY = '0' then
            next_state <= COLLECTSTALL_STATE;
          end if;
        when others =>
          next_state <= DECODE_STATE;
      end case;
    when RUNNING_STATE =>
      if (DATA_IN(2 downto 0) = STOP and WORD_READY = '0') then
        next_state <= IDLE_STATE;
      else
        next_state <= RUNNING_STATE;
      end if;
    when COLLECTSTALL_STATE =>
      if WORD_READY = '1' then
        stall <= DATA_IN;
        next_state <= WRITE_STATE;
      else
        next_state <= COLLECTSTALL_STATE;
      end if;
    when WRITE_STATE =>
      next_state <= WRITE_STATE;
      if WORD_READY = '0' and writeadr = stall then
        next_state <= IDLE_STATE;
      end if;
    when others => next_state <= IDLE_STATE;
  end case;
end process nextstate_function;

memory_controller : process(CLK)
begin
  if (CLK'Event and CLK = '1') then
    WORD_READY_DELAYED <= WORD_READY;
    case state is
      when RUNNING_STATE =>
        if(readadr < stall-1) then
          if CLK_MODIFIED = '1' then
            readadr <= readadr + 1;
          end if;
        end if;
      when WRITE_STATE =>
        if WORD_READY_DELAYED = '0' and WORD_READY = '1' then
          writeadr <= writeadr + 1;
        end if;
      when others =>
        writeadr <= "0000000000";
        readadr <= "0000000000";
    end case;
  end if;
end process memory_controller;


output_function : process (state, readadr, writeadr, DATA_IN)
  begin
```

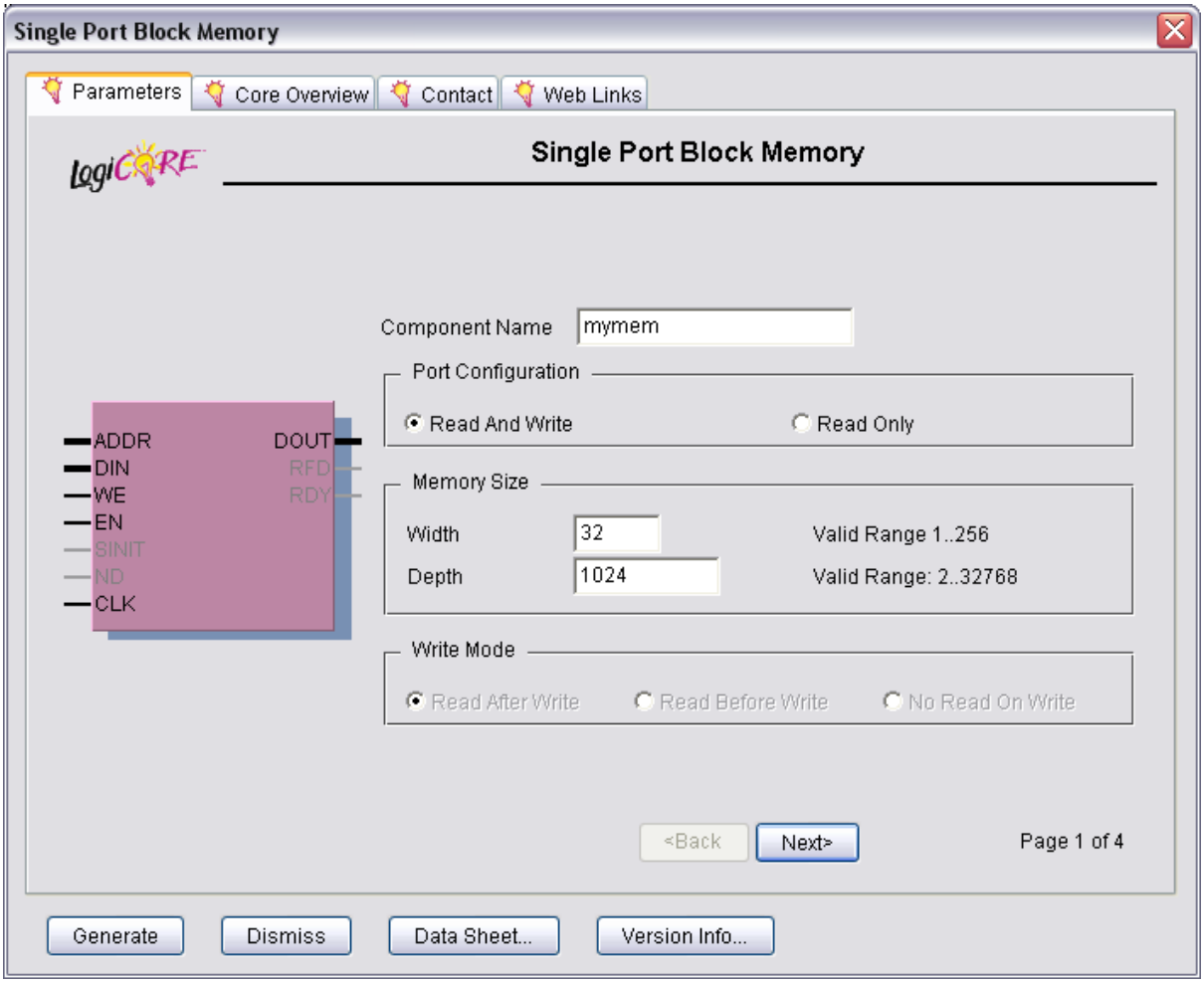
```
      case state is
        when DECODE_STATE =>
          MEM_WE <= '0';
          MEM_EN <= '0';
          MEM_ADR <= "0000000000";
          DATA_OUT <= word_zero;
          if DATA_IN(2 downto 0) = SETCLOCK then
            FQD <= DATA_IN(31 downto 3);
          end if;
        when RUNNING_STATE => -- RUN
          MEM_WE <= '0';
          MEM_EN <= '1';
          MEM_ADR <= readadr;
          DATA_OUT <= word_zero;
        when WRITE_STATE => -- WRITE WORD
          MEM_WE <= '1';
          MEM_EN <= '1';
          MEM_ADR <= writeadr;
          DATA_OUT <= DATA_IN;
        when others =>
          MEM_WE <= '0';
          MEM_EN <= '0';
          MEM_ADR <= "0000000000";
          DATA_OUT <= word_zero;
      end case;
  end process output_function;
end Behavioral;
```

Byte_Collector.vhd:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    12:27:00 08/06/2007
-- Design Name:
-- Module Name:    Byte_Collector - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.types.ALL;

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

entity Byte_Collector is
```

```
  Port (
    CLK     : in  STD_LOGIC;
    DATA_IN   : in  STD_LOGIC_VECTOR (7 downto 0);
    DATA_OUT    : out  STD_LOGIC_VECTOR (31 downto 0);
    BYTE_READY  : in  STD_LOGIC;
    WORD_READY  : out  STD_LOGIC
  );
end Byte_Collector;

architecture Behavioral of Byte_Collector is
  signal byte_count       : STD_LOGIC_VECTOR (1 downto 0) := "00";
  signal Sreg           : STD_LOGIC_VECTOR (31 downto 0) := word_zero;
  signal BYTE_READY_DELAYED : STD_LOGIC := '0';
  signal data_ready       : STD_LOGIC := '0';

begin
  process(CLK)
  begin
    if (CLK'Event and CLK = '1') then
      BYTE_READY_DELAYED <= BYTE_READY;
      if BYTE_READY_DELAYED = '0' and BYTE_READY = '1' then
        Sreg <= Sreg(23 downto 0) & DATA_IN;
        byte_count <= byte_count + 1;
      end if;
    end if;
  end process;

  process (CLK)
    begin
    if CLK'event and CLK = '1' then
      if (byte_count = "11" and BYTE_READY = '1' and BYTE_READY_DELAYED = '0') then
        data_ready <= '1';
      else
        data_ready <= '0';
      end if;
    end if;
  end process;

  DATA_OUT <= Sreg when data_ready = '1';
  WORD_READY <= data_ready;
end Behavioral;
```

mymem:



top.ucf – see the base design.

Serial.vhd – see the base design.

Types.vhd – see the base design.

Appendix G Adder Design (VHDL)

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:      Jakob Toft, s012012
--
-- Create Date:     13:30:05 09/17/2007
-- Design Name:     Test Adder
-- Module Name:     top - Behavioral
-- Project Name:    Pattern Generator
-- Target Devices:  Spartan 2
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.02 - behaviorable model added
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

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

entity top is
  Port (
    CLK_50M     : in    STD_LOGIC;
    A           : in    STD_LOGIC_VECTOR(3 downto 0);
    B           : in    STD_LOGIC_VECTOR(3 downto 0);
    C           : out   STD_LOGIC_VECTOR(4 downto 0)
  );
end top;

architecture Behavioral of top is

begin
  -- Logic
  process(CLK_50M)
    variable tmp: integer := 0;
  begin
    if CLK_50M'event and CLK_50M = '1' then
      tmp := conv_integer(A) + conv_integer(B);
      C <= conv_std_logic_vector(tmp, 5);
    end if;
  end process;
end Behavioral;
```

Appendix H PG Buffer Design (VHDL)

Top.vhd:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:      Jakob Toft, s012012
--
-- Create Date:     09:46:05 06/11/2007
-- Design Name:     Top entity of the PG
-- Module Name:     top - Structural
-- Project Name:    Pattern Generator
-- Target Devices:  Spartan 2
-- Tool versions:
-- Description:     Buffer implementation
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created from memory implementation template
-- Revision 0.02 - Added parallel port code
-- Revision 0.03 - added FIFO, cleaned up code.
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.types.ALL;

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

entity top is
  Port (
    CLK_50M         : in    STD_LOGIC;
    EPP_WRITE_ENABLE: in    STD_LOGIC;
    EPP_DATA_BUS    : inout STD_LOGIC_VECTOR (7 downto 0);
    EPP_WAIT        : out   STD_LOGIC;
    EPP_DATA        : in    STD_LOGIC;
    TEST            : out   STD_LOGIC_VECTOR(3 downto 0);
    PINOUT          : out   STD_LOGIC_VECTOR(31 downto 0)
  );
end top;

architecture Structural of top is

  -- Component declarations
  component EPP
    Port (
      CLK             : in    STD_LOGIC;
      EPP_WRITE_ENABLE: in    STD_LOGIC;
      EPP_DATA_BUS    : inout STD_LOGIC_VECTOR (7 downto 0);
      EPP_WAIT        : out   STD_LOGIC;
      EPP_DATA        : in    STD_LOGIC;
      BYTE_READY      : out   STD_LOGIC;
      DATA_OUT        : out   STD_LOGIC_VECTOR (7 downto 0)
    );
  end component;
```

```
  component fifo_1024x32
    port (
      din: IN STD_LOGIC_VECTOR(7 downto 0);
      rd_clk: IN STD_LOGIC;
      rd_en: IN STD_LOGIC;
      wr_clk: IN STD_LOGIC;
      wr_en: IN STD_LOGIC;
      dout: OUT STD_LOGIC_VECTOR(31 downto 0);
      empty: OUT STD_LOGIC;
      full: OUT STD_LOGIC
    );
  end component;

  -- Signal declarations

  -- receiver
  signal byteready: STD_LOGIC := '0';
  signal byte     : STD_LOGIC_VECTOR (7 downto 0) := byte_zero;

  -- FIFO
  signal din      : STD_LOGIC_VECTOR(7 downto 0) := byte_zero;
  signal dout     : STD_LOGIC_VECTOR(31 downto 0) := word_zero;
  signal wr_en    : STD_LOGIC := '0';
  signal rd_en    : STD_LOGIC := '0';
  signal empty    : STD_LOGIC := '0';
  signal full     : STD_LOGIC := '0';

  -- edge detect
  signal sync     : STD_LOGIC_VECTOR(1 downto 0) := "00";

begin

  EPP_Component : EPP
    port map(
      CLK => CLK_50M,
      EPP_WRITE_ENABLE => EPP_WRITE_ENABLE,
      EPP_DATA_BUS => EPP_DATA_BUS,
      EPP_WAIT => EPP_WAIT,
      EPP_DATA => EPP_DATA,
      BYTE_READY => byteready,
      DATA_OUT => byte
    );

  FIFO_Component : fifo_1024x32
        port map (
            din => din,
            rd_clk => CLK_50M,
            rd_en => rd_en,
            wr_clk => CLK_50M,
            wr_en => wr_en,
            dout => dout,
            empty => empty,
            full => full
      );

  -- Logic
  rd_en <= (not empty);
  wr_en <= '1' when (sync = "01" and full = '0') else '0';
  PINOUT <= dout;

  -- catch byte on the positive edge of BYTE_READY
  din <= byte when sync = "01";

  -- synchronize byteready to the clock
```

```
  process(CLK_50M)
  begin
    if CLK_50M'event and CLK_50M = '1' then
      sync <= sync(0) & byteready;
    end if;
  end process;
end Structural;
```

EPP.vhd:

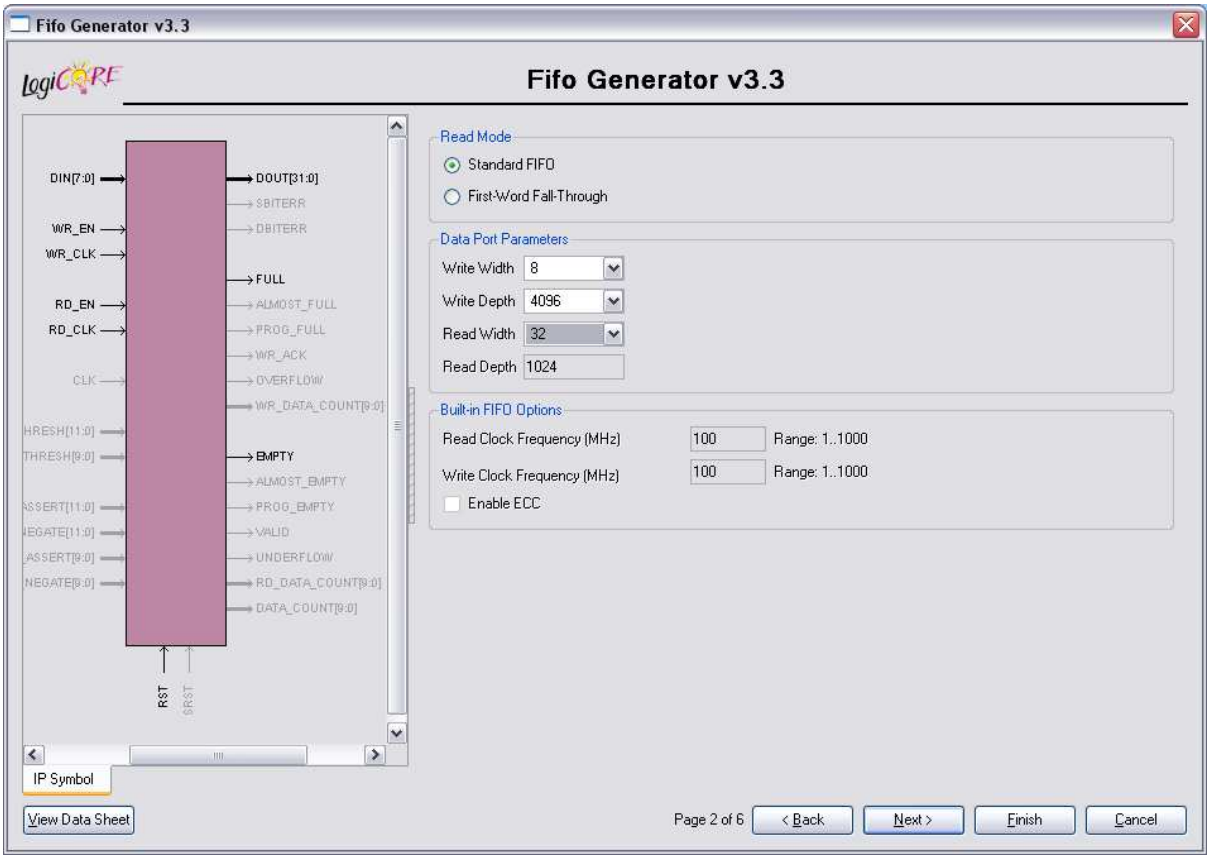
```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:    Jakob Toft, s012012
--
-- Create Date: 11:06:55 08/02/2007
-- Design Name:
-- Module Name: EPP - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:   EPP receiver module
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.02 - EPP handshake added
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

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

entity EPP is
  Port (
    CLK              : in    STD_LOGIC;
    EPP_WRITE_ENABLE : in    STD_LOGIC;
    EPP_DATA_BUS     : inout STD_LOGIC_VECTOR (7 downto 0);
    EPP_WAIT         : out   STD_LOGIC;
    EPP_DATA         : in    STD_LOGIC;
    BYTE_READY       : out   STD_LOGIC;
    DATA_OUT         : out   STD_LOGIC_VECTOR (7 downto 0)
  );
end EPP;

architecture Behavioral of EPP is
  signal EPP_SYNC        : STD_LOGIC_VECTOR(2 downto 0) := "000";
  signal EPP_STROBE_EDGE : STD_LOGIC := '0';
  signal pwe             : STD_LOGIC := '0';
  signal pds             : STD_LOGIC := '0';
  SIGNAL data            : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
  signal br              : STD_LOGIC := '0';
begin
```

```
  -- invert the signals who are active low
  pwe <= (not EPP_WRITE_ENABLE);
  pds <= (not EPP_DATA);

  -- synchronize the signals to the clock
  process (CLK)
    begin
    if CLK'event and CLK = '1' then
      EPP_SYNC <= EPP_SYNC(1 downto 0) & pds;
    end if;
  end process;

  -- detect the data strobe edges
  EPP_STROBE_EDGE <= '1' when (EPP_SYNC(2 downto 1) = "01") else '0'; -- rising
edge

  -- respond to handshake by asserting or deasserting EPP_WAIT
   EPP_WAIT <= EPP_sync(1);

  -- grab byte when data is valid.
  process (CLK)
    begin
    if CLK'event and CLK = '1' then
      br <= '0';
      if(EPP_STROBE_EDGE = '1' and pwe = '1' and pds = '1') then
        br <= '1';
        data <= EPP_DATA_BUS;
      end if;
    end if;
  end process;

  DATA_OUT <= data when br = '1';
  BYTE_READY <= br;
  EPP_DATA_BUS <= "ZZZZZZZZ";
end Behavioral;
```

top.ucf:

```
net "CLK_50M"           loc = "p80";
net "EPP_WRITE_ENABLE"  loc = "p206";
net "EPP_DATA_BUS<0>"   loc = "p15";
net "EPP_DATA_BUS<1>"   loc = "p14";
net "EPP_DATA_BUS<2>"   loc = "p10";
net "EPP_DATA_BUS<3>"   loc = "p9";
net "EPP_DATA_BUS<4>"   loc = "p8";
net "EPP_DATA_BUS<5>"   loc = "p7";
net "EPP_DATA_BUS<6>"   loc = "p6";
net "EPP_DATA_BUS<7>"   loc = "p5";
net "EPP_WAIT"          loc = "p3";
net "EPP_DATA"          loc = "p205";
net PINOUT<0>           loc = "p180";# C5  on connector C
net PINOUT<1>           loc = "p178";# C7  on connector C
net PINOUT<2>           loc = "p175";# C9  on connector C
net PINOUT<3>           loc = "p173";# C11 on connector C
net PINOUT<4>           loc = "p168";# C13 on connector C
net PINOUT<5>           loc = "p166";# C15 on connector C
net PINOUT<6>           loc = "p164";# C17 on connector C
net PINOUT<7>           loc = "p162";# C19 on connector C
net PINOUT<8>           loc = "p160";# C21 on connector C
net PINOUT<9>           loc = "p152";# C23 on connector C
net PINOUT<10>          loc = "p150";# C25 on connector C
net PINOUT<11>          loc = "p148";# C27 on connector C
net PINOUT<12>          loc = "p146";# C29 on connector C
net PINOUT<13>          loc = "p140";# C32 on connector C
net PINOUT<14>          loc = "p138";# C34 on connector C
net PINOUT<15>          loc = "p135";# C36 on connector C
net PINOUT<16>          loc = "p133";# C38 on connector C
net PINOUT<17>          loc = "p129";# C40 on connector C
net PINOUT<18>          loc = "p69"; # E5  on connector E
net PINOUT<19>          loc = "p67"; # E7  on connector E
net PINOUT<20>          loc = "p62"; # E9  on connector E
net PINOUT<21>          loc = "p60"; # E11 on connector E
net PINOUT<22>          loc = "p58"; # E13 on connector E
net PINOUT<23>          loc = "p49"; # E15 on connector E
net PINOUT<24>          loc = "p47"; # E17 on connector E
net PINOUT<25>          loc = "p45"; # E19 on connector E
net PINOUT<26>          loc = "p43"; # E21 on connector E
net PINOUT<27>          loc = "p41"; # E23 on connector E
net PINOUT<28>          loc = "p36"; # E25 on connector E
net PINOUT<29>          loc = "p34"; # E27 on connector E
net PINOUT<30>          loc = "p31"; # E29 on connector E
net PINOUT<31>          loc = "p29"; # E31 on connector E
```

fifo_1024x32.vhd:



types.vhd – see the base design.

Appendix I Tests

ReqID: 1	Compare to Agilent 81134A	
Description:	Action:	Result:
Square waves – clock signal with fixed width.	Run the test vector sequence: 0, 1, 0	OK
Pulse – with selectable width or duty cycle.	0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1	First output vector gets slightly distorted by the echo of the last data written to the memory.
Bursts – followed by zero data, can be repeated.	Run the test vector sequence: 0, 5, 5, 5, 0, 0, 0, 5, 5, 5, 0, 0, 0	OK
Data – selectable pulse width.	Run the test vector sequence: 0, 5, 5, 5, 0, 0, 0, 5, 5, 5, 0, 0, 0	OK, width of 3 as expected.
Pseudo random binary sequences.	Any test vector will do: 9, 7, 7, 3, 5	OK

ReqID: 2	Software	
Description:	Action:	Result:
Load data from file into software.	Load a set of test vectors into the program and observe the highlighting in action.	Syntax highlighting highlights wrong text; it is corrected whenever something is added to the lines.
Copy paste test vectors into software	Copy a set of vectors into the program, observe syntax highlighting.	Highlighting is not happening until something is added to the line.
Generate a Sine wave that is repeating.	Generate 16 values describing a sine curve and multiply by 2^{31}. Use the 32nd bit to signal negative values.	OK. (accomplished with matlab)

ReqID: 3	Memory implementation	
Description:	Action:	Result:
Error tolerance	Assign too few test vectors to the LOAD command. Command: LOAD, 4, 1, 2, 3, RUN	System stalls, waiting for RUN. Multiple 'stop' commands followed by 'run' can regain control of the system.
Ensure correct operation with multiple LOAD commands.	Command: LOAD, 4, 1, 2, 3, 4, RUN, STOP, LOAD, 2, 9, 8, RUN, STOP	OK

ReqID: 4	Buffer implementation	
Description:	Action:	Result:
Create an identity test.	comparing the input of the PG with the output	OK
Check for ground bounce	Examine the ground bounce when used with the parallel port receiver.	OK
Ensure correct operation with multiple inputs.	Command: 1, 2, 3, 4 (send) 9, 8, 7, 6 (send)	OK

ReqID: 5	Investigate Ground Bounce caused by SSO	
Description:	Action:	Result:
Frequency: 50 MHz SSO: 32 bit	Pattern: 00000000, FFFFFFFF, 00000000	Heavy ground bounce, signal is destroyed.
Frequency: 25 MHz SSO: 32 bit	Pattern: 00000000, FFFFFFFF, 00000000	Signal has 4ns instability at both transitions.
Frequency: 25 MHz SSO: 4 bit	Pattern: 00000000, 00000055, 00000000	Signal has 2ns instability at both transitions, measured with 500 MHz sampling (min period).