Low power deformable mirror actuator controller

de Bruijn, W.

Award date:
2009

Link to publication

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Low Power Deformable Mirror Actuator Controller

by
W. de Bruijn

Supervisors:

prof. dr. ir. C.H. van Berkel (TU/e)
ir. R.M.L. Ellenbroek (TU Delft)
ir. R.F.M.M. Hamelinck (TU/e)
dr. R.H. Mak (TU/e)

Eindhoven, February 2009

Abstract

A deformable mirror may be used to improve the image quality of ground-based optical telescopes, by counteracting the distortions caused by atmospheric turbulence. Dedicated controller boards, carrying multiple FPGAs, are used to drive the actuators inside the deformable mirror. The power dissipation of the FPGAs causes additional distortions by heating the air surrounding the mirror. An attempt is made to reduce these distortions by optimizing the power usage of the FPGAs. The software in the FPGAs is migrated to a development board and methods for measuring power usage are developed. The power measurements have led to improved controller designs that use less power than the original designs.

Contents

List of Figures

List of Tables

List of Acronyms

AO	Adaptive optics
DAC	Digital-to-analog converter
DCM	Digital clock manager
FPGA	Field-programmable gate array
LVDS	Low-voltage differential signaling
PLL	Phase-locked loop
PWM	Pulse-width modulation
VCD	Value change dump
XUP	Xilinx university program

Chapter 1

Introduction

Ground-based optical telescopes have the disadvantage of atmospheric turbulence, causing a degradation in quality of the images taken by the telescopes. Since a number of years adaptive optics (AO) systems have been used to overcome these problems. Adaptive optics systems may use a deformable mirror to correct a distorted optical wavefront entering a telescope. The reflective surface of these mirrors is deformed by arrays of actuators with nanometer accuracy. Further developments of ground-based optical telescopes require increasingly large deformable mirrors and accompanying control systems. To meet the demands of the latest generation of large telescopes, a new adaptive mirror system is being designed and tested in a collaborative project between the Eindhoven University of Technology, the Delft University of Technology and TNO Science and Industry [1, 2, 3].

1.1 Adaptive mirror system

A general overview of the components of the adaptive mirror system that is being developed, can be seen in figure 1.1. Light emitted by objects in space, such as stars, reaches the earth as a parallel wavefront. When this wavefront travels through the earth's atmosphere, atmospheric turbulence causes the wavefront to change shape, resulting in undesired distortions of the images taken by a telescope. A way to solve this problem is to introduce a deformable mirror in the optical path of the telescope. A deformable mirror has the ability to correct a distorted wavefront by changing its own shape to the opposite of the wavefront's shape. The mirror used in this system [1], consists of one or more hexagonal arrays of 61 electromagnetic actuators that are attached to a thin reflective membrane. Each array has its own actuator controller board, carrying the electronics needed to drive the actuators.

The actuator controllers communicate with a control system, that has the task of determining the shape of the wavefront by reading out a wavefront sensor and calculating the corresponding shape of the mirror, needed to correct the wavefront. The wavefront sensor receives part of the light beam, after it has been corrected by the deformable mirror. The other part of the corrected light beam is led to a detector that is used to make an image out of the received light.

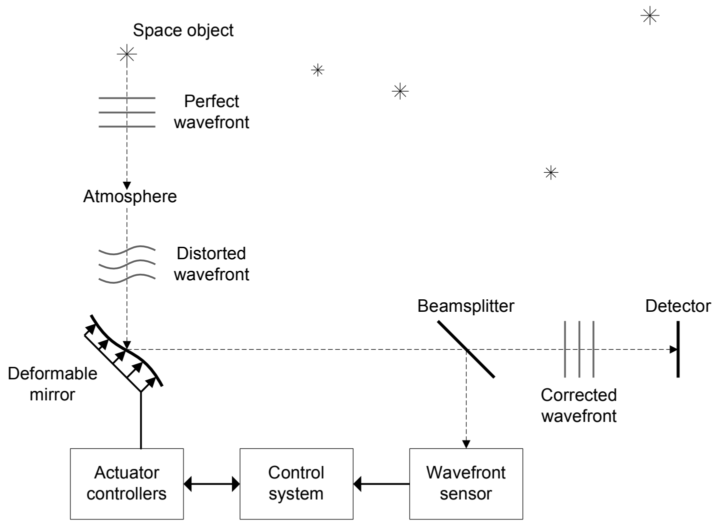

Figure 1.1: Adaptive mirror system

1.2 Actuator controllers

The electromagnetic actuators of the deformable mirror are controlled by low-voltage signals in the range of -0.8 V to 0.8 V. The main task of an actuator controller is to generate 61 of these signals for each of the actuators, with a resolution of 14 bits (see appendix A). To do so, an actuator controller is equipped with three field-programmable gate arrays (FPGAs).

FPGAs are chips containing large numbers of programmable logic blocks that can be connected together with high-speed interconnects. The functionality contained in an FPGA can be described in a hardware description language such as Verilog [4] or VHDL [5]. As opposed to ordinary microprocessors which run software sequentially, FPGA designs are compiled to real hardware on a chip. This approach allows FPGA designs to make use of parallelism and high clock speeds. Because of these advantages, FPGAs are ideally suited to generate the control-signals for the deformable mirror.

Like all other chips, FPGAs also use power, which is dissipated in the form of heat. This power usage can be split up into two categories. Static power is the constant amount of power consumed by the FPGA at all times. Static power is mainly a result of the way chips are produced and is not dependent on the calculations performed by the FPGA. The power consumption resulting from the calculations is called dynamic power and this part of the total power usage can be changed by loading different designs in the FPGA.

1.3 Pulse-width modulation

The low-voltage signals, needed to drive the actuators of the deformable mirror, are generated by a scheme called pulse-width modulation (PWM). Digital systems may use PWM to directly generate analog signals without the need for digital-to-analog converter chips (DACs). As each of the 61 electromagnetic actuators of a mirror segment needs a separate analog signal, using PWM instead of digital-to-analog converters, greatly reduces the system's overall complexity.

PWM operates by modulating the on-off times (duty cycle) of a square wave with a fixed frequency. When such a modulated square wave is passed through a low pass filter, an analog signal can be retrieved that corresponds to the average value of the square wave. Figure 1.2 shows an example PWM signal with its resulting analog signal.

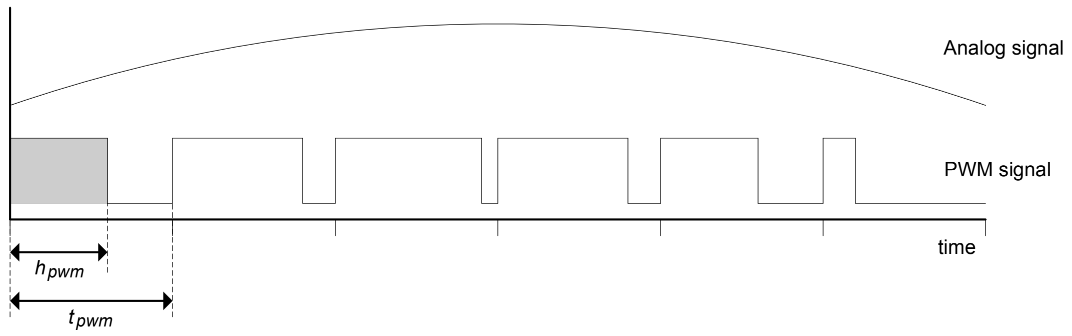

Figure 1.2: Pulse-width modulation

The frequency f_{pwm} of the PWM signal is defined as $\frac{1}{t_{pwm}}$ where t_{pwm} is the period of the square wave as shown in figure 1.2. The duty cycle of a PWM signal is defined as the time the signal is high h_{pwm}, divided by the period-time t_{pwm}. It is clearly visible that the value of the analog signal directly corresponds to the value of the duty cycle of the PWM signal. As the duty cycle increases, so does the average value of the PWM signal and consequently the value of the analog signal.

The PWM signals are generated by a digital system, so the duty cycle can only be set with a certain resolution. If we call this resolution (in bits) n_{pwm}, we can calculate the clock frequency f_{clk}, at which the digital system needs to operate, as follows:

$$f_{clk} = \frac{2^{n_{pwm}}}{t_{pwm}} = 2^{n_{pwm}} f_{pwm} \tag{1.1}$$

This formula clearly shows that an increase in resolution or frequency of the PWM signal, leads to a higher internal clock frequency.

1.4 H-bridge

The actuators of the deformable mirror are designed to be driven by both positive and negative voltages. To enable the FPGA to output negative voltages, an additional circuit called an H-bridge is needed. Each actuator has a separate H-bridge circuit, that is controlled by five different PWM signals from the FPGA. A simplified version of such an H-bridge circuit can be seen in figure 1.3.

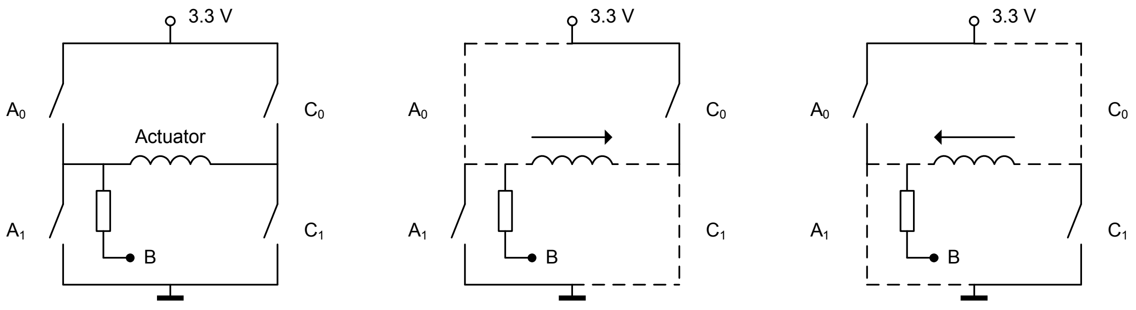

Figure 1.3: H-bridge circuit in three different states

The H-bridge consists of four electronic switches (transistors), controlled by the FPGA, that allow current to flow in either direction through the coil of an actuator. When switches A_0 and C_1 are closed, current flows from left to right through the coil. When A_1 and C_0 are closed, current flows in the opposite direction. The states that have to be avoided at all times are the states where both A_0 and A_1 or C_0 and C_1 are closed at the same time, as these would lead to a short circuit between the positive power supply and ground. In order to save electronic components, the task of enforcing this constraint is handled inside the FPGA.

To generate positive and negative voltages with the H-bridge, the FPGA has to output different PWM signals with varying duty cycles. These PWM signals switch the transistors of the H-bridge on and off, letting current flow through the actuator for specific amounts of time. Because the actuator and surrounding electronics act as an analog filter, the result of this process is a positive or negative voltage over the actuator, moving it to a certain position.

Besides the four PWM signals, that control the transistors of the H-bridge, a fifth signal B is present that is connected to one side of the actuator coil with a resistor. PWM signal B is used to directly generate the 4 least significant bits of the voltage setting (setpoint) of the actuator, by controlling the amount of time, current is allowed to flow through the resistor in a certain direction. This approach allows the 4 least significant bits of the setpoint to be generated in parallel with the other bits, reducing the requirements on the internal clock frequency of the FPGA.

1.5 PWM resolution

Each of the four switches of the H-bridge circuit, shown in figure 1.3, is controlled by a PWM signal with a period of 16.384 μs ($\approx$ 61 kHz) and a resolution of 11 bits. According to equation 1.1, the minimum internal clock frequency at which these signals should be generated is equal to: $\frac{2^{11}}{16.384\,\mu\text{s}} = 125$ MHz.

The A_1 and C_1 switches of the H-bridge are driven by the inverted versions of the PWM signals, controlling switches A_0 and C_0. The resulting unfiltered output signal, generated by the H-bridge, is therefore proportional to the difference between the signals driving A_0 and

C_0. This differential output signal can only assume three different voltage levels (-3.3 V, 0 V or 3.3 V), so this type of PWM is called three-level or class-BD PWM [6]. The three-level signal, created out of the two 11 bits PWM signals, driving A_0 and C_0, has a resolution of 12 bits.

PWM signal B has the same period as the other four PWM signals (16.384 μs), but a lower resolution of only 4 bits. The 12 bits differential signal and the 4 bits B signal, determine the voltage setting over the full voltage range (-3.3 V to 3.3 V) with a resolution of 16 bits. This corresponds to a 14 bits resolution over the voltage range of -0.8 V to 0.8 V (see appendix A).

1.6 Problem description

The power dissipated by the FPGAs of the actuator controller, causes a rise in temperature and therefore an increase in air-turbulence of the air surrounding the mirror. This turbulence is unwanted, as it causes additional distortions that have to be compensated by the deformable mirror. High power dissipation creates the need for more expensive cooling measures, adding to the cost and complexity of the entire system. The problem with the current implementation of the actuator controllers is that the power dissipated by the FPGAs is too high with respect to the power dissipated by the actuators. Without active cooling, the actuator controllers would violate the heat dissipation requirements, listed in appendix B.

The goal of this project is to reduce the power dissipation of the FPGAs to a minimum, while meeting the actuator controller requirements, described in appendix A. On one hand this can be done by carefully selecting the right type of FPGA, reducing the static power dissipated. On the other hand this can be done by implementing the algorithms in the FPGA in a power-efficient manner, reducing the dissipated dynamic power.

Because only a limited number of FPGAs with the right amount of logic elements and I/O pins exist, choosing the right type of FPGA isn't very difficult. The main focus of this research will therefore be on reducing the dynamic power dissipation. To this end we will be developing and comparing different synchronous and asynchronous implementations of the control algorithms by means of simulation and measurements on real hardware.

To aid the development process and the measurements on real hardware, the existing actuator controller will be re-implemented on an FPGA development board together with some custom-made electronics to recreate the communications interface and the actuator circuits. After this has been done, power usage data of the actuator controller software will be obtained from three different sources:

- From measurements on the original actuator controller.
- From measurements on the FPGA development board.
- From simulations of the controller software.

This data can then be used to see what effect certain code-optimizations have on power usage. After evaluating a number of possible optimizations, the most effective ones will be incorporated into the final system.

Chapter 2

XUP implementation

Developing new software for the actuator controller requires a flexible development platform, that allows the FPGAs involved, to be easily reconfigured. Furthermore, the system needs to have provisions for measuring the power usage of its individual components. The original actuator controllers are specifically designed for the task of driving the deformable mirror's actuators, so they lack some of the features that are present in a development board. Because of this reason, the decision was made to create a new implementation of an actuator controller with the help of a commercial FPGA development board and some custom electronics.

The development board allows for easy reprogramming and testing of code on a large FPGA, and the custom electronics are used to emulate the communications interface and the actuator electronics on the existing controller hardware. The rest of this chapter describes the various parts of the development system and the process of re-implementing the existing software on the development board.

2.1 Original actuator controllers

The original actuator controllers are built around three FPGAs manufactured by Altera [7]. One of these FPGAs, called the master, has the task of handling the controller's incoming and outgoing communications. Communication with the controller is performed through low-voltage differential signaling (LVDS), which is a technique used to transmit data reliably at high speeds with low power usage. The controller has two dedicated LVDS converter chips, that have the task of converting the LVDS signals to signals the master FPGA can handle and vice versa. The LVDS communication is serial and runs at a speed of 40 Mbps. To enable easy communication between a PC and the actuator controllers, an ethernet-LVDS bridge is available that is designed to convert ethernet packets to LVDS packets.

The other two FPGAs, called the slaves, are identical and are used to generate the PWM signals for the actuators. Each of the actuator controller's 61 actuators has an H-bridge circuit, that is used to generate precise voltages over the actuator coils, moving the actuators to a certain position. The slave FPGAs have the task of controlling the position of the actuators by sending the correct signals to the H-bridge circuits, based on the information

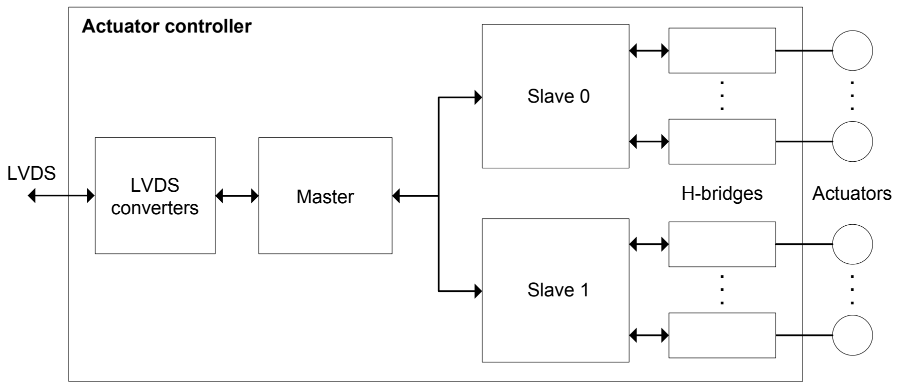

Figure 2.1: Original actuator controller architecture

received from the master FPGA. Besides controlling the position of the actuators, the slaves can also be used to check the integrity of the coils in the actuators. The global architecture of one of the actuator controllers is shown in figure 2.1.

Figure 2.2 shows the master FPGA, situated on a separate board together with the LVDS interface and the power supplies. The two slave FPGAs are mounted on another board, depicted in figure 2.3, together with the H-bridge circuits and accompanying analog electronics.

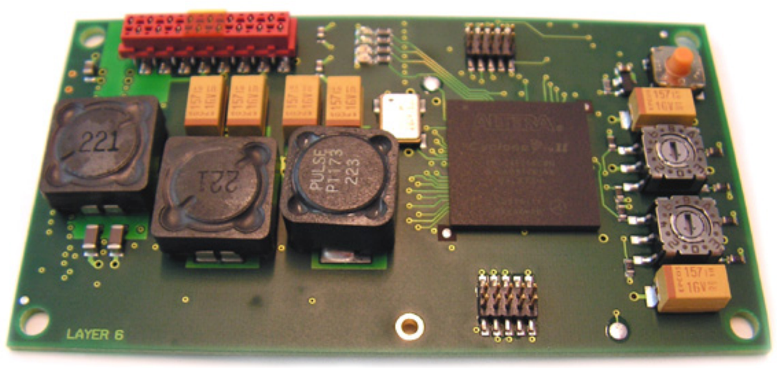

Figure 2.2: Actuator controller master board

The master communicates with the slaves through a 16 bit wide data bus and a 6 bit wide address bus. These busses, along with some control- and power-lines, are routed through a connector, that connects the master and slave boards together.

In order to implement the actuator controller on a development board, custom electronics were designed to match the functionality of the LVDS interface and the H-bridge circuits of the original actuator controller boards.

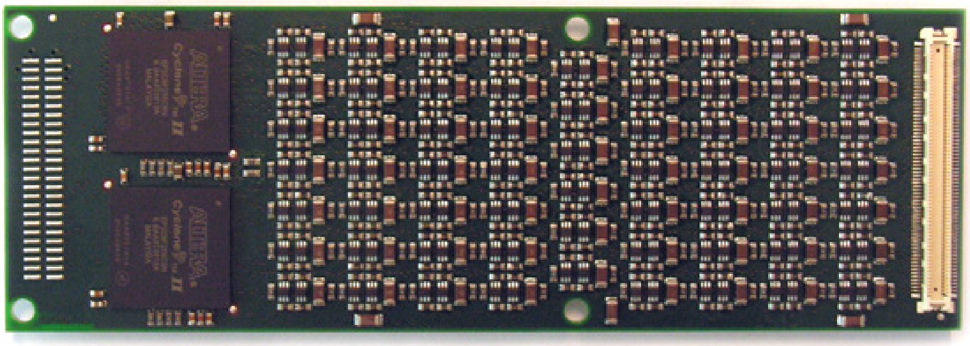

Figure 2.3: Actuator controller slave board

2.2 XUP development board

The FPGA development board that is used as a basis for the implementation of the actuator controller is the XUP Virtex-II Pro [8] made by Xilinx. This development board contains a large Virtex-II Pro FPGA [9] and a number of peripheral hardware units, such as an ethernet- and video-interface, which can be connected to the FPGA. The only peripheral hardware units that are used however, are the I/O expansion connectors, mounted at the front side of the board. These connectors are used to connect to two circuit boards, containing the custom electronics.

To power the FPGA and the peripheral hardware, the development board houses three independent power supplies (2.5 V, 3.3 V and 1.5 V), with connectors for measuring the total current. Because these power supplies power the complete development board and no connectors are present for measuring the current through the FPGA, determining power usage of the FPGA is only possible by measuring the differences in total current.

The main reason for choosing the XUP development board was the fact that they were already available for use and there was previous experience with development on these boards. It should be noted however that this board contains only one Xilinx Virtex-II Pro FPGA, whereas the original actuator controllers contain three Altera Cyclone II FPGAs [7]. Re-implementing the actuator controller on the development board, requires the code from the three smaller FPGAs of the original controller to be mapped onto the single large FPGA of the development board.

2.3 Custom electronics

Figure 2.4 shows the two small circuit boards, designed specifically for the XUP development board. These circuits were manufactured by the university's technical department, based on the self-designed schematics, shown in appendix C. The board on the left is the LVDS interface and contains, besides components used to protect both the XUP board and the ethernet-LVDS bridge, two LVDS converter chips [10]. Because a single converter chip only supports half-duplex communication, two chips are used to provide a full duplex communication channel. Although both chips are able to send and receive data, one of the chips is permanently configured to only transmit and the other to receive.

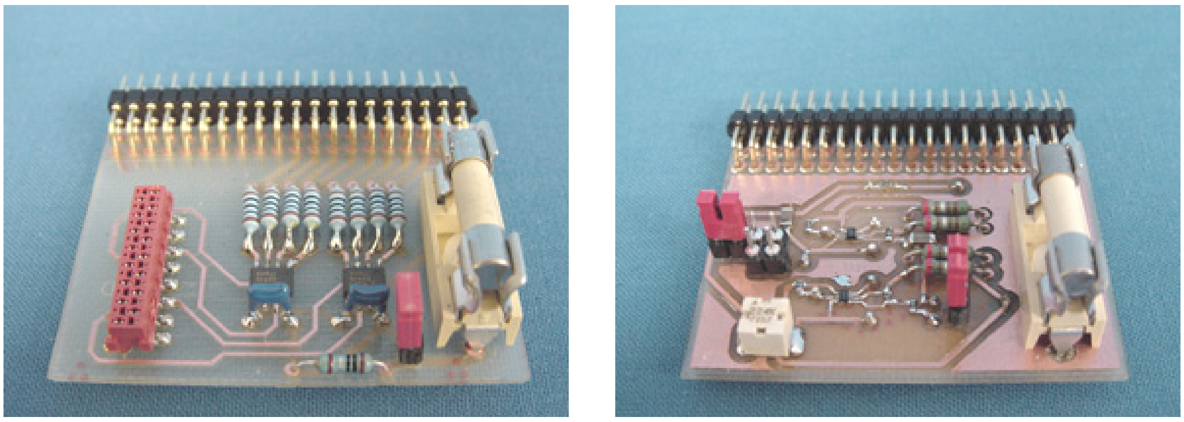

Figure 2.4: The LVDS interface and the H-bridge circuit

The board on the right is a reconstruction of the circuit associated with 1 actuator of the deformable mirror. It can be used to estimate the power usage of the actuators and to verify the output voltages, generated by different controller designs. The choice to implement only 1 channel instead of 61 was based on the limited number of control signals present on the I/O connectors of the development board and the practical difficulties associated with implementing a board with 61 of these circuits. The disadvantage of this arrangement is that it's not possible to measure the power usage of all 61 circuits at the same time, as can be done with the original actuator controllers. The total power usage however, can be approximated by multiplying the power usage of the single H-bridge circuit by 61, as all circuits are identical. This and the fact that the main focus of this research is on the power usage of the FPGA, makes this board a valid replacement for the actuator electronics in the original controller.

Appendix C shows the two schematics associated with the circuit boards. These schematics are partially based on the schematics of the original controller. The schematic for the LVDS interface shows the two converter chips connected to the development board's expansion connector on one side and to the LVDS connector on the other side. The other schematic shows 4 transistors arranged in an H-bridge configuration with the actuator coil and some analog components in the middle. The 4 transistors can be switched on and off directly by the FPGA to allow current to flow in either direction through the actuator coil. During tests, the actuator coil is replaced by a 41 Ω resistor, because the inductive properties of the coil are not of much importance when measuring power usage.

Both circuits incorporate resistors in their power supply lines that can be used to measure the total current drawn. Additionally, the H-bridge circuit has provisions for measuring the current through and voltage over the actuator coil. These measurements can be used to exactly calculate the amount of power used by the actuator itself. The fuses and resistors in the control lines from the FPGA are meant as a protection against short-circuits, but have the disadvantage of consuming a small amount of power themselves.

The circuit boards are designed in such a way that they can be fitted directly next to each other onto the expansion connectors of the XUP development board. This allows them to be easily removed and attached, whenever this is needed. The two boards receive their power from the power supplies on the development board and operate on 3.3 V.

2.4 Migrating controller software

The FPGAs on the original controller are made by Altera and the FPGA on the development board by Xilinx, so migrating the software from the original controller to the development board can't be done without some modifications to the code. Altera specific hardware units have to be replaced by equivalent Xilinx units and the code sets from the master and slave FPGAs have to be joined together to fit into one FPGA.

Both the master and slave FPGAs make use of phase-locked loop (PLL) units to create internal clock signals with higher clock frequencies out of a slower external clock. As these units are Altera specific, they were replaced by Xilinx digital clock managers (DCMs). These DCMs are able to convert the incoming 100 MHz clock signal to 200 and 125 MHz clocks for the master and slaves modules. Another hardware module that had to be replaced was a dual port RAM unit that is used by the master to store data from received communication packets. This module was replaced by a Xilinx specific, 64 word long, RAM unit.

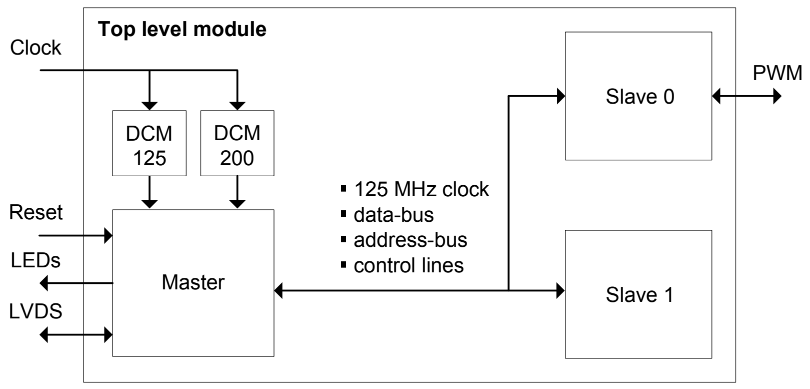

Figure 2.5: Architecture of the code for the development board

Figure 2.5 shows the global architecture of the new code created for the development board. The new top level module, written in Verilog, contains one instance of the original master module and two identical instances of the original slave modules. Both the master and slave modules are written in VHDL.

The modules in the top level design are connected together in roughly the same way as they are on the board of the actuator controller. This time however, the wires and busses are inside the FPGA instead of on the circuit board. Besides the master and slave modules, the two DCM units are instantiated in the top level module, providing 200 and 125 MHz clock signals to the rest of the design. The reason to put the DCMs in this module and not in any of the submodules, as was originally the case with the PLL units, is to reduce the skew on the 100 MHz input clock, which could prevent the DCM units from operating correctly. Table 2.1 shows how much FPGA resources are used by the new controller design, implemented using the Xilinx ISE software [11].

Resource	**Used**	**Available**	**Utilization**
Slice flip-flops	6,013	27,392	21%
4-input LUTs	9,817	27,392	35%
Digital clock managers	2	8	25%
Logic slices	7,376	13,696	53%

Table 2.1: Actuator controller resource usage

2.5 Testing the development board

Before using the development board to do power measurements, it had to be tested for functional correctness. The communication over the LVDS interface with the master module in the FPGA was tested with a small Java program that was used to send control-packets to the ethernet-LVDS bridge, which would forward them to the development board. Besides the Java program, a Matlab toolbox was also available to send LVDS packets to the actuator controller.

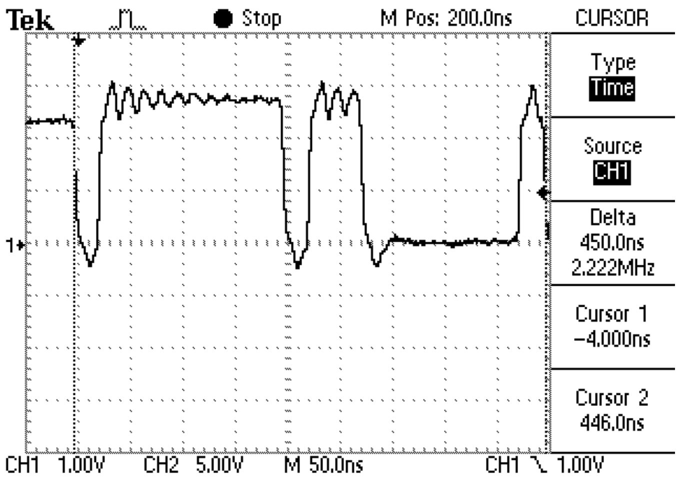

Figure 2.6: Oscilloscope image of the LVDS communication

Figure 2.6 shows a sample of 18 bits of LVDS communication, measured after the LVDS converter chips. The 18 bits take 450 ns to transmit, so one bit of the LVDS signal is 25 ns long, corresponding to a bit-rate of 40 Mbps. The LVDS packets consist of a variable number of 18 bit words, depending on the type of packet. Each 18 bit word includes a start-bit with value 0 and a stop-bit with value 1, so the actual data transmitted consists of 16 bit words.

To test the H-bridge circuit, commands were sent to the development board to enable the generation of PWM signals. Figure 2.7 shows two PWM signals, measured with the H-bridge circuit detached from its I/O connector.

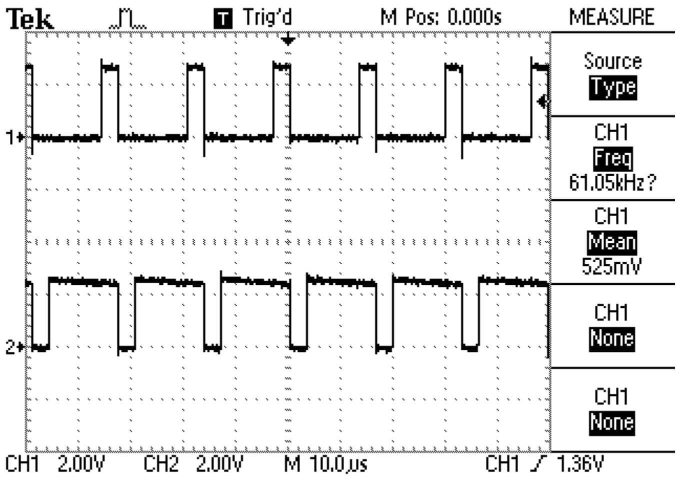

Figure 2.7: Oscilloscope image of the generated PWM signals

The two signals seen here, are used to control the A_0 and C_0 switches of the H-bridge, as defined in figure 1.3. The period of these signals is 16.384 μs, corresponding to a frequency of approximately 61 kHz. Attaching the H-bridge circuit to the development board, results in a positive or negative voltage being generated over the actuator resistor.

Chapter 3

Power usage

Optimizing the power usage of the actuator controller requires a reliable and accurate way of determining the power that is used by the software in the FPGAs. More specifically, we are interested in the dynamic power usage of the FPGAs, as this part of the total power usage can be reduced by modifying the code. In order to measure the effects of optimizations, power usage of the original controller software has to be determined first.

Power usage was measured on the two different implementations of the actuator controller (the original one and the one on the XUP development board). A third way of determining power usage is offered by the Xilinx XPower tool [11], which has the ability to estimate the power usage of designs with the help of simulation data. This chapter describes how each of these three types of measurements was carried out and which results were obtained.

3.1 Power usage in FPGAs

FPGAs are manufactured using the same processes as most other integrated circuits, so standard formulas can be used to calculate how much power is used by an FPGA. As stated before, the total power usage of a chip can be split up into two parts. Static power is a result of standby and leakage currents through a device, and is mainly a result of the way the chips are manufactured. Dynamic power is a combination of short-circuit and capacitive power dissipations, and can be described by the following formula [12]:

$$P_{dyn} = P_{sc} + \alpha C V^2 f_{clk} \tag{3.1}$$

In this formula, P_{dyn} and P_{sc} represent the dynamic and short-circuit power dissipations. The last term in the formula, describes the capacitive power dissipation, with α representing the average number of 0 to 1 output transitions (switching activity), C the load capacitance, V^2 the square of the input voltage and f_{clk} the clock speed at which the chip runs.

Roughly speaking, the load capacitance C is a measure of how much hardware in a chip is used. The total capacitance of a device increases when more transistors and interconnecting wires are used. The clock speed f_{clk} and α together, give an indication of how much switching activity takes place on a chip. The formula shows that dynamic power usage is linearly dependent on the switching activity, the capacitance and the clock frequency, and quadratically dependent on the input voltage. Because the short-circuit power dissipation P_{sc} is also linearly dependent on the clock frequency [12] and the input voltage is constant, reducing the power dissipation of the FPGAs in the actuator controller will be mainly based on reducing the switching activity and the amount of hardware resources used.

3.2 Original actuator controllers

The power usage of the original actuator controllers, configured with the original code, was measured by Rogier Ellenbroek and Roger Hamelinck. This was done by measuring the current drawn from their single 12 V power supply, located in the ethernet-LVDS bridge. Each actuator controller also has its own power supply, that generates several different voltages out of the incoming 12 V, for the chips on the controller. The regulators in the power supply have a limited efficiency, so to make comparisons possible, the power they dissipate had to be subtracted from the total amount of power measured.

Figure 3.1 shows how the total amount of power used by one actuator controller relates to the actuator voltage. The amount of power used by the master and slaves includes both the static and dynamic power dissipations of the FPGAs.

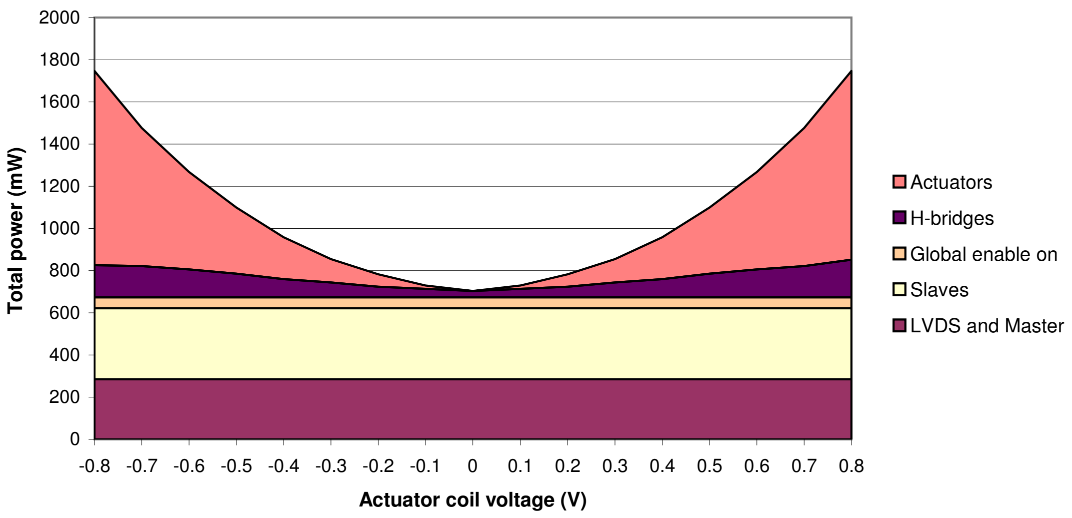

Figure 3.1: Power usage of the original actuator controller

The graph clearly shows that the power used by the actuators and H-bridges is highly dependant on the actuator voltage setting, and that the power used by the FPGAs remains constant. The increase in power usage of the actuators can be explained by the fact that a higher voltage directly leads to more power being used by the analog electronics. Power usage of the FPGAs remains constant, because the internal high frequency clocks of the FPGAs keep running at the same speed, regardless of the value of the PWM duty cycle. Changing the voltage setting does cause some changes inside the FPGAs, but these occur at such low frequencies that they only have a small amount of influence on the total amount of power.

3.3 XUP development board

The power usage of the FPGA and custom electronics on the XUP development board was determined by measuring the current drawn from the three independent power supplies on the board (see appendix D). The problem with these power supplies is that they power the complete development board, including a number of unused peripheral hardware units such as an ethernet interface, and that the development board does not offer an easy way of measuring the current drawn by the FPGA. Establishing the power used by the FPGA could only be done by calculating the difference between the current drawn by a fully configured FPGA and by the FPGA in a minimally configured state. The assumption with this approach is that the power usage of the peripheral hardware doesn't change much when loading different configurations into the FPGA. The minimal configuration, used as a basis for the power measurements, doesn't perform any calculations so its dynamic power usage is 0. It does, however, use an amount of static power that can't be measured directly, so another assumption is that the real amount of static power used, matches the values given in the data sheet of the FPGA [9]. Figure 3.2 displays the results of the measurements on the XUP development board, including the static power dissipation of the FPGA.

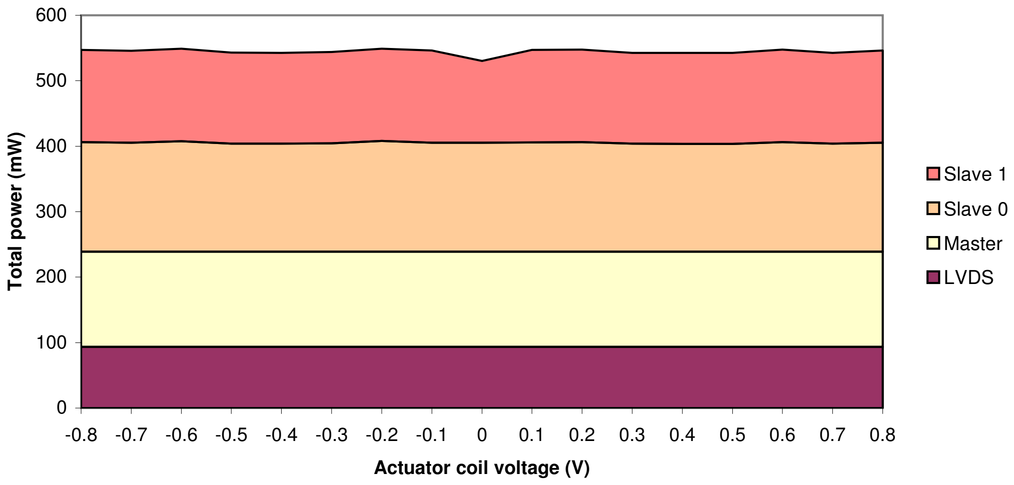

Figure 3.2: Power usage of the XUP development board

Like the graph in figure 3.1, this graph also shows that the power used by the FPGA stays roughly the same, independent of the actuator voltage. A small difference exists between the total power usage of the FPGAs and LVDS circuit on the original actuator controller ($\approx$ 670 mW), and the power usage of these on the development board ($\approx$ 550 mW), which can be caused by any of the following reasons:

- The different type of FPGAs used and the difference in manufacturer.

- The differences in the way power usage was measured.
- The alterations to the original code that were needed to let it run on the XUP development board.
- The transition from three small FPGAs to one large FPGA.

The real cause for this difference, however, can't be determined exactly, because very detailed measurements are not possible.

The measurements done on the development board, have the extra advantage of showing how the power usage is divided over the components of the system. The power used by the LVDS circuit was established by measuring the voltage over its current sense resistor. The power used by the other components (the master and two slaves), was calculated by disabling the components one by one and measuring the total current drawn by the FPGA. This way of measuring power usage is the reason for the difference in power between slave 0 and slave 1. While they have identical code, slave 0 has some added overhead, that is caused by the optimizations done on a design with less components.

The H-bridge circuit operates, just like the LVDS circuit, on a voltage of 3.3 V and has a current sense resistor to measure its total power usage. Figure 3.3 shows the results of power measurements done on the H-bridge circuit.

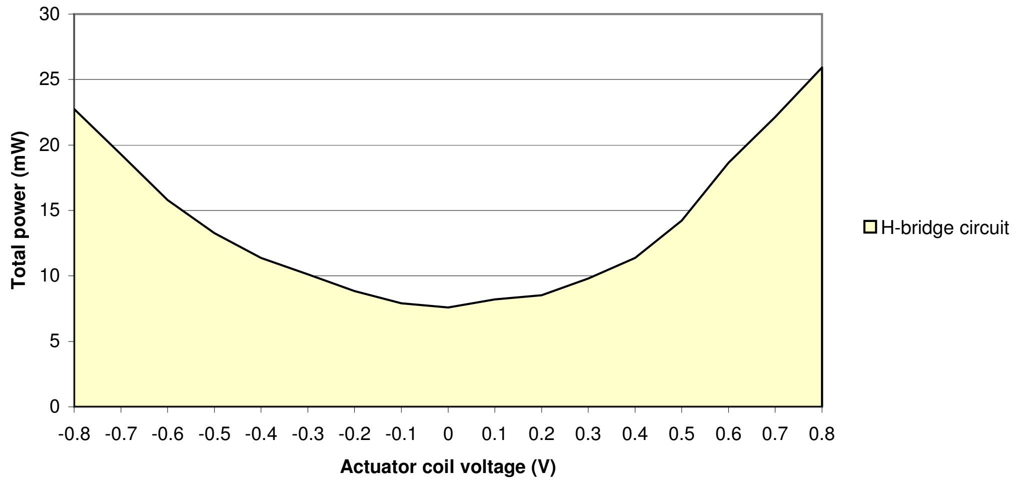

Figure 3.3: Power usage of the XUP H-bridge circuit

This graph has approximately the same shape as the graph for the actuators and H-bridges in figure 3.1, but there is clearly too much power usage when the actuator voltage is 0, compared to the original actuator controllers. The added power usage is likely to be caused by the extra safety measures incorporated into this circuit, such as a fuse and current limiting resistors, which consume some amount of power. There is also a small asymmetry in the graph, which is likely to be the result of the different electrical characteristics of the components used to construct the H-bridge. These differences in power usage are not of much importance, because the power measurements of the H-bridge circuit won't be used when making optimizations.

3.4 Simulation

The XPower tool [11] offers a way of determining power usage of a design, without implementing it on real hardware. XPower uses formulas similar to equation 3.1, in combination with simulation data and FPGA characteristics, to estimate power usage. To obtain the simulation data, a test bench for the original controller software was written (see appendix F.1). The test bench has the ability to read LVDS packets from a file and to send them, during simulation, to the master in the same way the packets are sent to the real actuator controller by the ethernet-LVDS bridge. This mechanism is used to enable the generation of PWM signals by writing the correct values to the actuator controller's internal registers. Besides setting the internal registers of the actuator controller, the test bench also has the task of generating the necessary clock and reset signals.

The simulation data was obtained by simulating the post-place and route netlist of the controller software with the ModelSim PE [13] simulator. The resulting value change dump (VCD) file can be used by the XPower tool in combination with the compiled design file to estimate power usage. Figure 3.4 shows the estimated distribution of dynamic power usage over the various parts of the master code.

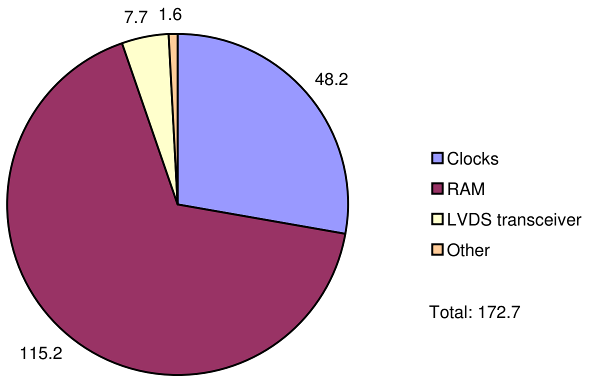

Figure 3.4: Master power distribution (mW)

It is very clear that most of the power usage in the master is due to the clocks and the RAM, used to buffer received LVDS packets. Only a small percentage is consumed by the LVDS transceiver and other hardware resources in the FPGA.

Figure 3.5 shows an estimation of the distribution of dynamic power usage for both slaves. This chart shows that most of the power, used by the slaves, is also due to the switching activity of the system clocks. The other part of the total power usage is occupied by the PWM counter and comparators. These components are both part of the system, used to generate the PWM signals for the actuators.

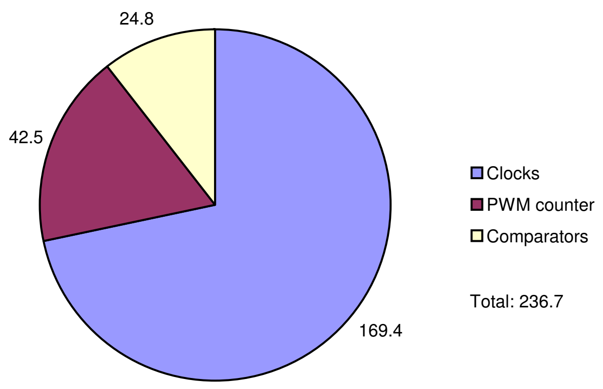

Figure 3.5: Slave 0 and slave 1 power distribution (mW)

3.5 Conclusion

The data obtained from simulating the actuator controller software clearly shows that the system clocks are a big source of power usage, both for the master and slave FPGAs. Reducing this power usage can be achieved by reducing the frequency of the system clocks or by reducing the amount of clocked hardware that is used in the design. One way to do this is by converting the synchronous circuits of the original design into asynchronous circuits.

The RAM used in the master FPGA also uses a relatively large amount of power, that is linearly dependant on the size of the memory. Since this memory is used as a temporary buffer for storing LVDS packets, an obvious optimization would be to reduce the size of the buffer or to remove the buffer completely. The PWM counters and comparators inside the slaves can be optimized by implementing them in a more power efficient way, possibly using a different design for the counters and a PWM clock with reduced frequency.

Chapter 4

Design

The power measurements, listed in chapter 3, show which parts of the original software use most of the power. In both the master and slaves, the power used by the system clocks is significant, since these modules run on high clock frequencies and use a lot of FPGA resources. Besides the power used by the system clocks, a large portion of the total power for the master is consumed by the internal RAM. In the slaves, most of the power is used by the system clocks, but a fairly large amount of power is also consumed by the PWM counter and comparators.

To optimize the power usage of the actuator controller, new designs for the controller software were made, based on the results of the power measurements. This chapter describes these new designs and shows how they reduce the total amount of power used.

4.1 Power reduction techniques

There are several techniques that can be used to reduce the power usage of a design. The most obvious one is to lower the frequency of the system clock. Power usage is linearly dependent on the clock frequency, so any change to the clock has an immediate effect on the power usage of a design. Since power usage is also linearly dependent on the amount of hardware used, reducing resource usage can also lead to a design with lower power usage.

Another commonly used technique is clock gating. Clock gating reduces power by temporarily disabling the clock to parts of a design that are unused at that moment. Since the clock is disabled, no dynamic power is used by the corresponding hardware. More advanced techniques include power gating, to temporarily disable power to certain parts of a design, and lowering the operating voltage of the FPGA. Since power usage is quadratically dependent on the operating voltage, large power reductions can be achieved with this technique.

The optimizations made to the actuator controller will be mainly based on lowering the clock frequency and reducing the amount of hardware used, since the other techniques are difficult to apply to the existing controller software.

4.2 Asynchronous master

The main task of the master is to transmit and receive LVDS packets and to decode their contents for the slave modules. In the original design, the master uses a 64 word RAM to temporarily store the received packets, before sending their contents to the slaves. This setup allows the master to reject packets with an invalid checksum, as the checksum can only be calculated after a complete packet has been received. The RAM buffer also helps to resolve timing-issues related to the communication between the master running at 200 MHz and the slaves running at a lower speed of 125 MHz. Because the slaves are slower, the master has to store a packet until all of the data in the packet has been received and processed by the slaves. Figure 4.1 shows the internal architecture of the original master.

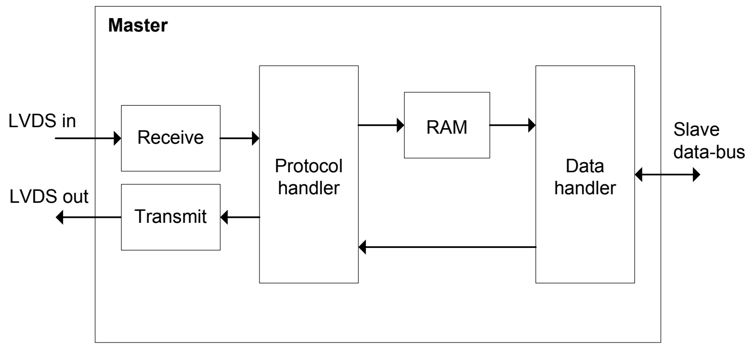

Figure 4.1: Original master architecture

The transmit and receive modules are used to serialize and de-serialize the LVDS data. The protocol handler decodes the LVDS data and writes it to the RAM, which is read by the data handler, that has the task of sending the data to the slaves. Responses from the slaves are sent directly to the transmit module without first passing through the RAM.

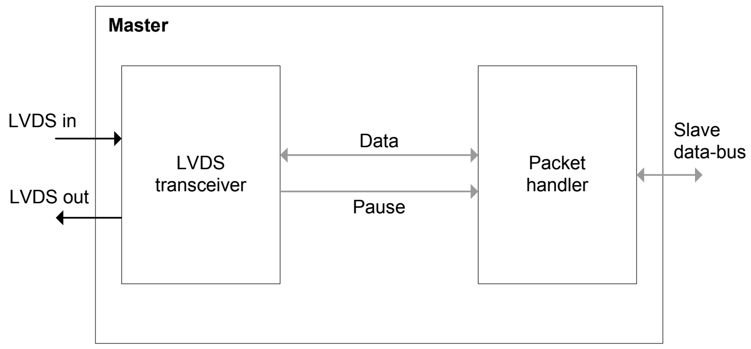

Figure 4.2: Optimized master architecture

The optimized design for the master, shown in figure 4.2, consists of two modules, connected together with handshake channels. The LVDS transceiver has the same functionality as the transmit and receive modules in the original master. It is connected to the packet handler with a 16 bit wide data channel in each direction and a pause channel, used to indicate the pause between two consecutive LVDS packets. The packet handler combines the functionality of the protocol handler and the data handler, but doesn't use a RAM to buffer the packets. Instead, it processes the packets asynchronously and sends the data words contained in them directly to the slaves. The biggest advantage of this approach is that the RAM, responsible for most of the power usage, is eliminated from the design. The downside however is that packets with an invalid checksum will also be sent directly to the slaves. This disadvantage was considered to be acceptable, since packet errors occur only rarely and usually don't cause much trouble when they do occur. Using asynchronous logic instead of synchronous logic for the modules reduces the power used by the system clocks.

4.3 Asynchronous slave

An actuator controller consists of a master and two slave modules. Figure 4.3 shows the internal architecture of one of these slave modules. Each slave provides the PWM signals needed to drive 31 actuators of the deformable mirror. Since the mirror only has 61 actuators, one PWM unit of the second slave is left unconnected. All data sent to and from the slaves is handled by the address decoder. The address decoder holds the global configuration registers and controls the settings of the PWM units, including their setpoints and enable states. To communicate with the PWM units, the address decoder is connected to each of them with a data-bus for reading and writing, and a number of control lines. This setup leads to a high fan-out at the address decoder, since each wire has to go to all 31 PWM units. A high fan-out may cause timing related issues and an increase in resource usage, resulting in higher power usage. Another module with a high fan-out is the PWM counter. The PWM counter increments its internal 11 bit register every clock cycle and resets when the maximum number has been reached. Like the address decoder, the PWM counter is also connected to every PWM unit.

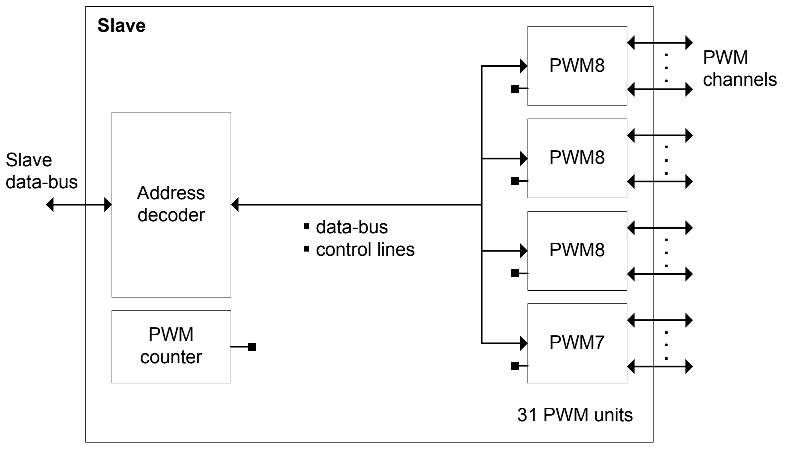

Figure 4.3: Original slave architecture

Figure 4.4 shows the architecture of the optimized slave. This new design combines the functionality of the two original slaves into one module, providing control signals for all 61 actuators of the deformable mirror.

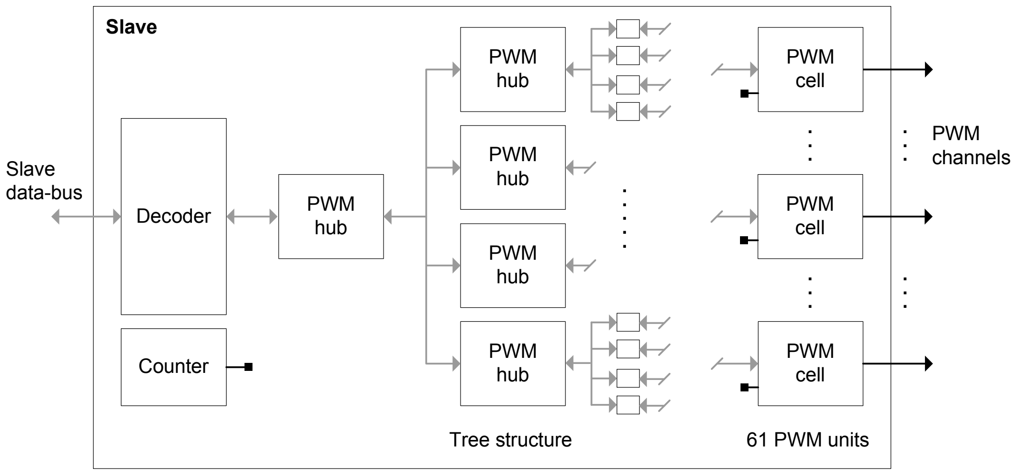

Figure 4.4: Optimized slave architecture

Combining the two slaves is possible since the optimized module is designed to run on one FPGA. The new slave module reduces resource usage, since the address decoder and PWM counter don't have to be duplicated for the two slaves. The internal components of the optimized slave are connected together with handshake channels and are constructed out of asynchronous logic. This helps to reduce the power usage resulting from the system clocks, since less clock signals are needed when using asynchronous logic.

The decoder and counter modules have approximately the same functionality as the address decoder and the PWM counter in the original slaves. The only difference is that the new decoder has to handle 61 PWM units instead of 31. This leads to a small increase of the resources used by the decoder, but the total amount will still be smaller than the amount of resources used by the two separate address decoders. Another difference between the original and the optimized design is the way in which the decoder is connected to the PWM units. Instead of a high fan-out data-bus, a tree structure is used to make the connections to the PWM units. The tree structure, with branching factor 4 and depth 3, can accommodate $4^3 = 64$ PWM units. Using a tree has the advantage of a low fan-out per module and smaller timing delays, but also requires some extra hardware for the PWM hubs, that are used to route the data through the tree.

4.4 Recursive PWM

The PWM units of the original slaves are based on the design shown in figure 4.5. The value of a counter register, that is incremented every clock cycle, is continuously compared to a setpoint value by a comparator. When the counter is smaller than the setpoint, the output of the comparator becomes zero. When the counter is greater than or equal to the setpoint, the output of the comparator changes to one. Since the PWM signal needs to start at one at the beginning of a duty cycle, and turn zero when the setpoint has been reached, the output of the comparator is connected to an inverter. The output of the inverter is connected to a flip-flop, which stores the value at its input every rising clock-edge. The flip-flop is effectively a 1 bit memory and is needed to keep the PWM output stable during each clock period.

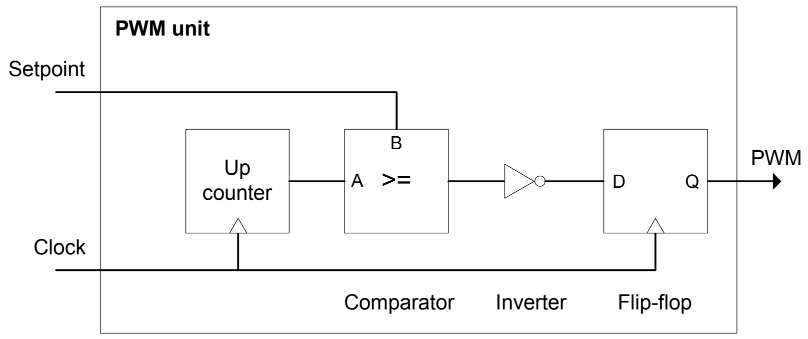

Figure 4.5: Counter-comparator PWM

The power simulations of the original controller software have shown that approximately 28% of the total power used by the slaves, is due to the PWM counter and comparators. The PWM counter is incremented every positive edge of the 125 MHz system clock, so a lot of switching takes place in the counter, resulting in a relatively high power usage. The comparators also use a lot of power, because they are connected directly to the counter.

To reduce the power used by the PWM units, several alternatives for the counter-comparator scheme were created. Figure 4.6 shows the design, called recursive PWM, that was considered most promising in terms of power usage and practical implementation.

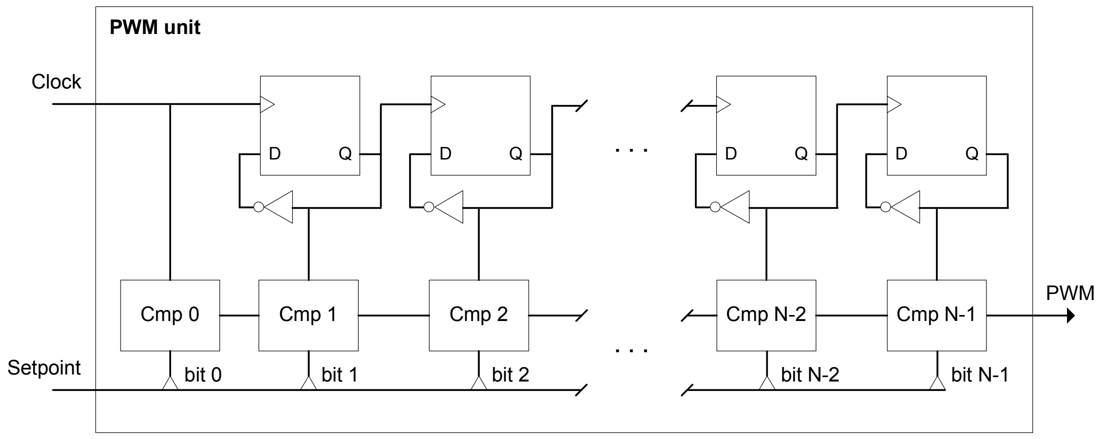

Figure 4.6: N-bit recursive PWM

The recursive PWM unit is partially based on the PWM designs found in [14]. It is essentially a modularized version of the standard counter-comparator PWM scheme. The flip-flops at the top of the diagram have the same function as the counter in the original design. Each flip-flop divides the frequency of its incoming clock signal by two, creating an inverted up-counter. The modules below the flip-flops contain only combinatorial logic, that is used to compare the setpoint to the counter value, stored in the flip-flops. Each module functions as part of an N-bit comparator, passing on information about the comparison to the modules further down the chain. Since all comparator modules, except the first one, are the same, extending the recursive PWM unit is just a matter of adding more of the bit modules, depicted in figure 4.7.

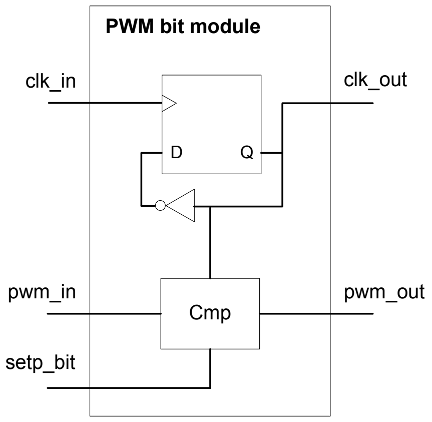

Figure 4.7: Recursive PWM bit module

The following equation shows how the frequency of the outgoing clock f_{clk_out} is related to the frequency of the incoming clock f_{clk_in} for each bit module:

$$f_{clk\_out} = \frac{f_{clk\_in}}{2} \tag{4.1}$$

Equation 4.2 describes the combinatorial function contained in each comparator module *Cmp*:

$$pwm\_out = ((clk\_out \wedge setp\_bit) \vee pwm\_in) \wedge (clk\_out \vee setp\_bit) \tag{4.2}$$

The biggest advantage of this design over the original counter-comparator design, is the fact that the recursive PWM unit is active on both clock-edges. Because it is active on both clock-edges, the input clock can be halved while still maintaining the same PWM output frequency, reducing overall power usage. A disadvantage of this design is the higher use of FPGA resources and the possible instability of the output signal as a result of gate-delays, introduced by the chain of divider flip-flops.

Chapter 5

Implementation

The designs from chapter 4 were implemented using different hardware description languages. The asynchronous master and slave were programmed in Haste [15, 16]. Haste is a programming language, specifically designed for the construction of asynchronous logic (see appendix E). To implement the asynchronous logic on an FPGA, the synchronous design flow of Haste is used. The synchronous design flow uses clocked hardware to simulate asynchronous circuits, as it is difficult to implement these circuits directly onto an FPGA. The recursive PWM unit was implemented in Verilog and integrated into the original controller code in two different ways.

This chapter describes how the designs for the master and slave modules were implemented, and what issues had to be resolved in order to get a working design.

5.1 Asynchronous master

The asynchronous master consists of two processes, connected together with a bidirectional data channel and a pause channel. The first process, called *trans*, has the task of serializing and de-serializing the LVDS data, and detecting the pauses between LVDS packets. To receive the LVDS data, the transceiver first waits for the negative edge of a start-bit. After a start-bit has been detected, the transceiver begins to sample the incoming LVDS signal every 25 ns. The sampled bits are stored in a 16 bits shift register and sent to the packet handler. Data words, received from the packet handler, are converted to LVDS signals by first sending a start-bit, followed by 16 bits of data and a stop-bit. The appropriate delays between the bits are created with a number of skip instructions. Since the LVDS transceiver runs on a clock frequency of 200 MHz, each skip instruction causes a delay of 5 ns. A resolution of 5 ns is needed to sample the 25 ns long bits. Pauses between two LVDS packets are detected by a separate process, that counts the number of consecutive high bits. When 18 consecutive bits are high, a signal is sent to the packet handler.

The second process in the master, called *packet*, handles the LVDS data at the packet level. After a pause has been detected, the packet header, containing the module id and command type, is received. The module id determines whether the packet should be processed further or not. The command type decides how the words, following the header, are handled. The packet handler sends the data, contained in the packets, directly to the slave without first storing it in a RAM. Responses received from the slave are sent to the LVDS transceiver, after adding the appropriate header and checksum. In the current implementation, the packet handler runs on a clock frequency of 100 MHz, but this may be lowered to reduce power usage in further implementations.

Resource	**Used**	**Available**	**Utilization**
Slice flip-flops	280	27,392	1%
4-input LUTs	331	27,392	1%
Digital clock managers	1	8	12%
Logic slices	266	13,696	1%

Table 5.1: Asynchronous master resource usage

Table 5.1 shows the resources used by the asynchronous master on the Virtex-II Pro FPGA. Only one DCM is used to create the 200 MHz system clock, since the 100 MHz clock is already present on the development board. The resource usage of the master is very low, so the power usage of the master is expected to be low as well.

5.2 Asynchronous slave

The asynchronous slave consists of a decoder process, a counter process and a tree structure, containing 61 PWM units. The decoder controls which internal registers are written to or read from, and handles the commands coming from the packet handler. The counter process is used by the 61 PWM units to generate their PWM signals. It has an internal register, that is incremented every positive edge of the 125 MHz system clock. The tree structure consists of PWM hubs and PWM cells. The PWM hubs are at the nodes of the tree and are used to route the data from the decoder to the PWM cells and vice versa. The PWM cells are placed at the leafs of the tree and have the task of generating the PWM signals. Each PWM cell has a small decoder with internal configuration registers to handle the commands from the main decoder, located at the base of the tree. The PWM cell generates the PWM signals in much the same way as in the original actuator controllers, using the data stored in the configuration registers.

Resource	**Used**	**Available**	**Utilization**
Slice flip-flops	10,234	27,392	37%
4-input LUTs	11,905	27,392	43%
Digital clock managers	1	8	12%
Logic slices	9,873	13,696	72%

Table 5.2: Asynchronous slave resource usage

Table 5.2 shows the resource usage of the asynchronous slave. In contrast with the master, the slave module does use a lot of resources. It uses one DCM to generate its 125 MHz system clock, but it also takes up 72% of the available logic slices in the FPGA. Most of these resources are due to the 61 PWM cells and the attached tree structure. The implementation of the slave in Haste uses more resources than the original slave, so in terms of power usage, the original implementation is likely to be more efficient.

5.3 Asynchronous actuator controller

The asynchronous master and slave were connected together to form a complete actuator controller, capable of controlling 61 actuators. The Haste design flow doesn't support connecting modules running at different clock speeds, so the generated Verilog modules had to be assembled manually. The passivator modules, used to connect the handshake lines of two Haste modules, were replaced by the passivators listed in appendix F.2. These new passivators can handle synchronous handshake signals, originating from modules running at different clock speeds.

Resource	**Used**	**Available**	**Utilization**
Slice flip-flops	10,514	27,392	38%
4-input LUTs	12,236	27,392	44%
Digital clock managers	2	8	25%
Logic slices	10,139	13,696	74%

Table 5.3: Asynchronous actuator controller resource usage

Table 5.3 shows how much FPGA resources are used by the asynchronous actuator controller. Most of the resources are used by the slave, while only a small percentage of the resource usage is due to the master and various other modules like the passivators and reset logic. Because so much resources are used, the timing constraints of the controller design only hold on speed-grade 7 FPGAs.

5.4 Recursive PWM

The recursive PWM design from chapter 4 was implemented as a Verilog module (see appendix F.3). To test the module, it was integrated into the code of the original actuator controller. All counter-comparator units were replaced by recursive PWM units and the central counter was removed, as each PWM channel now had its own counter. The clock frequency for the two slaves was lowered from 125 MHz to 62.5 MHz and the write-delays in the master were doubled to compensate for the lower clock frequency. Figure 5.1 shows an example of two PWM signals being generated by the recursive PWM units on the XUP development board. This image clearly shows that the signals are generated at the correct frequency by a slave running at half its original clock speed. The only concern with the design of the recursive PWM units is the fact that glitches may occur in the output signal as a result of gate delays in the counter section of the PWM unit. Because these glitches are very small spikes in the PWM signal, they may reduce the accuracy of the output signal and cause wrong actuator positions. Simulations have shown the presence of glitches in the PWM signal, but they have not been found in the signals generated by the development board.

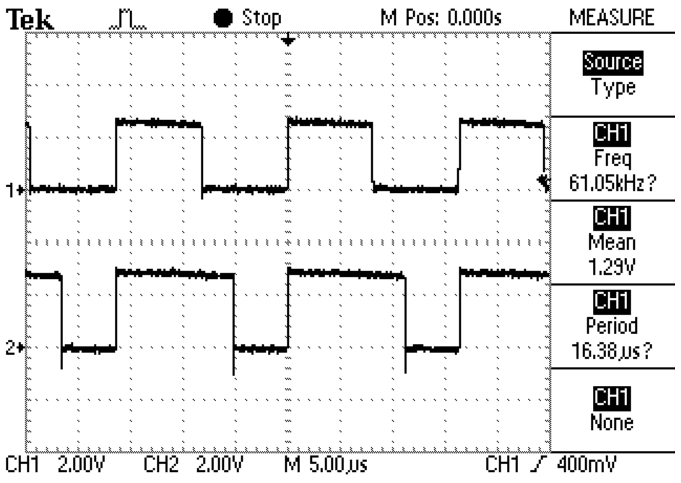

Figure 5.1: Oscilloscope image of the recursive PWM signals

Another version of the original actuator controller was made by taking the top part of the recursive PWM unit and creating the counter module, shown in figure 5.2, out of it. This counter module (listed in appendix F.4) replaces the original central counter, allowing the slaves to run at half their clock speed. Inverters are used to enable the counter to count up.

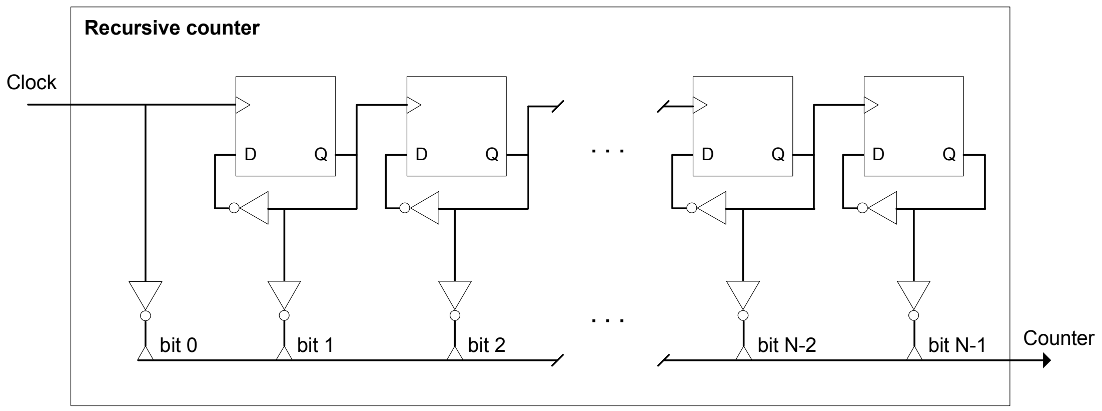

Figure 5.2: N-bit recursive counter

Because the resource usage of both these adaptations is almost the same as the resource usage of the original controller, lowering the clock speed of the slaves is likely to reduce overall power usage.

Chapter 6

Results

The power usage of the optimized controller designs was determined by measurements on the XUP development board and with the help of estimations from the Xilinx XPower tool. The measurements were carried out the same way as the measurements for the original controller software, shown in chapter 3. The only difference was that the asynchronous actuator controller needed to be implemented on a development board with a speed-grade 7 FPGA. A speed-grade 7 FPGA uses a little more power than the speed-grade 6 FPGA, used for the original controller software.

6.1 XUP development board

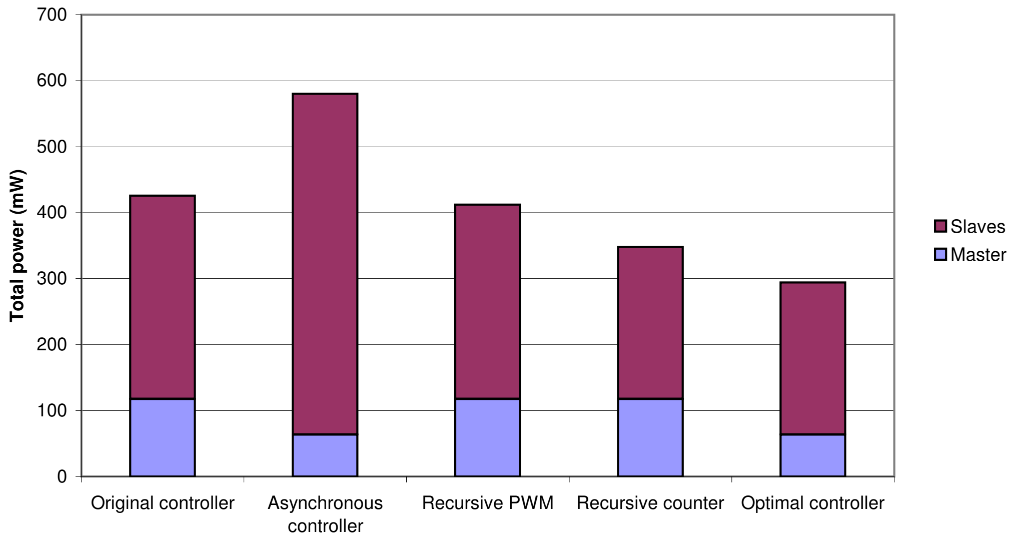

Figure 6.1: Power usage of controller designs on XUP development board

Figure 6.1 shows the results of the power measurements on the XUP development board. Each bar represents the total amount of power used by a controller design and also shows how this power is distributed over the master and slave modules. The static power, dissipated by the FPGA, is not included in the power usage of the designs.

The first bar represents the total amount of power used by the original controller software on the XUP board. The second bar shows how much power is used by the implementation of the actuator controller in Haste. While the total amount of power is higher than the original, the master needs less power to perform its tasks. The third and fourth bar show the power usage of the two implementations of the recursive PWM unit. These designs only change the slave modules, so the power usage of the master is the same as in the original controller. Both slave modules show an improvement in terms of power usage, but the biggest reduction of power can be seen in the recursive counter design. Combining the slaves of the recursive counter design with the asynchronous master would lead to a controller with power usage corresponding to the last bar in the graph. This optimal controller shows a reduction in power usage of approximately 29%.

6.2 Simulation

Figure 6.2 shows the power usage data, obtained by simulating the optimized controller designs. The bars represent the power usage of the designs in the same way as in figure 6.1. Although a lot of similarities exist between the simulated data and the measurements on the XUP board, a number of differences can be observed. First of all, there is a large difference between the measured power usage of the asynchronous controller and the simulated power usage. This difference is likely due to the fact that the power estimation tool can't determine the activity rates for the design with enough accuracy. Secondly, power usage is estimated too high for the master modules and too low for the slaves. This is probably also caused by the inability of the power tool to estimate the activity rates correctly.

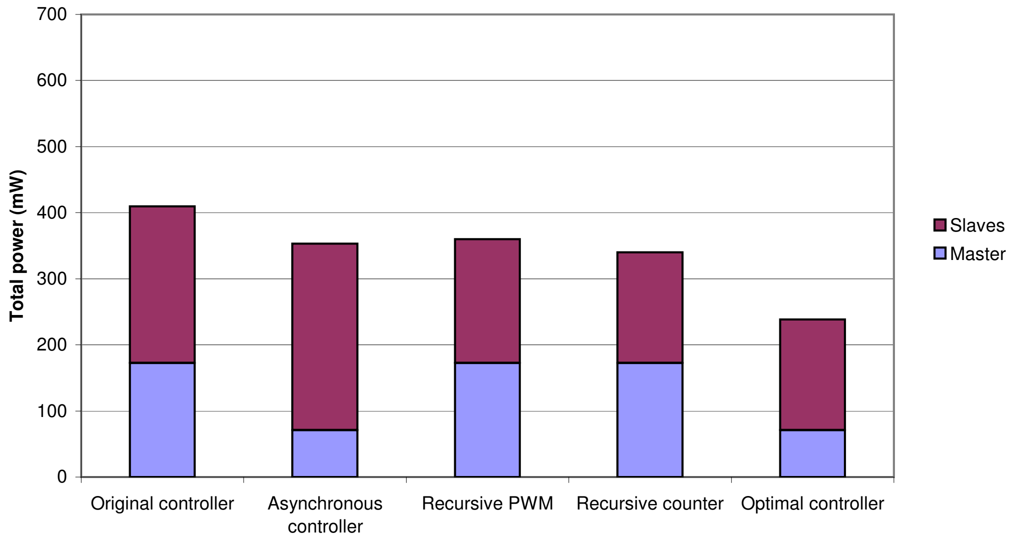

Figure 6.2: Power usage of controller designs in simulation

Since simulation doesn't always produce results with high enough accuracy, it is better to use the measurements on the development board to make decisions about the design of the controller. Simulations, however, are useful for determining the distribution of power in a design.

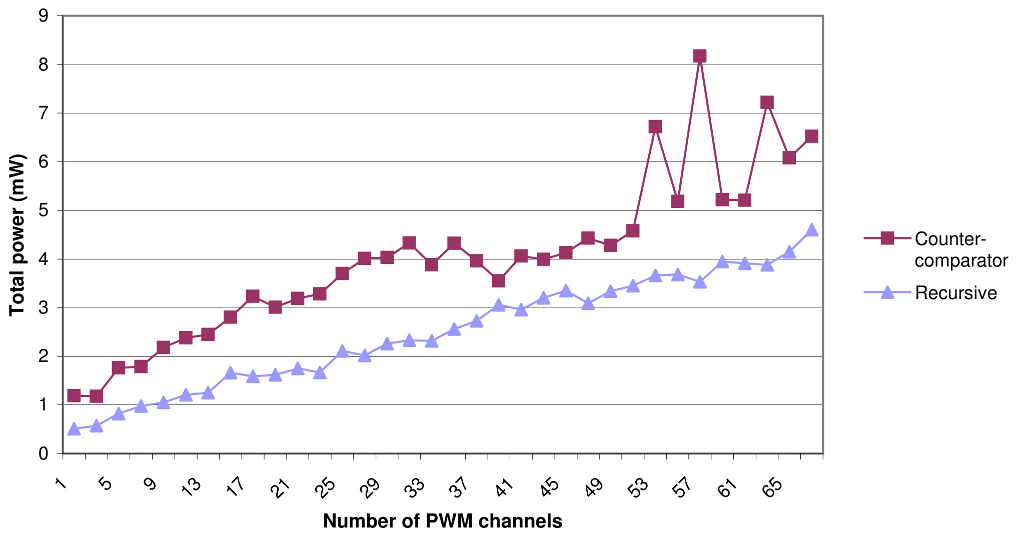

Figure 6.3: Power usage of different PWM units

Figure 6.3 shows another comparison of the standard counter-comparator PWM unit with the improved recursive PWM unit. Besides the fact that the recursive PWM units are more efficient, their power usage also increases more linearly.

Chapter 7

Conclusion

The software for the original actuator controllers was migrated successfully to the XUP development board. With the help of custom electronics, the modified code was tested by sending commands over the LVDS interface and observing the outputs of the development board and the H-bridge circuit. The migrated controller software has all the capabilities of the original software and meets the requirements listed in appendix A.

The power usage of the controller software was measured on the original actuator controllers, on the XUP development board and with the help of simulations. The development board did not have a straightforward way of measuring the power usage of its FPGA, so a less accurate differential measuring method had to be used. Only the dynamic power usage of the FPGA could be measured using this method, so the static power dissipation had to be estimated. The simulations were useful for finding out how the power usage was distributed internally over the modules and to test the functionality of new designs. The power measurements and simulation data were used to create optimized versions of the software for the actuator controller.

7.1 Power reduction

According to the measurements on the XUP board, the asynchronous actuator controller as a whole is less efficient in terms of power usage than the original actuator controller. This is primarily caused by the large amount of resources used by the slave module and the high clock speed it needs to run on. The asynchronous master uses less power than the original master, as it is able to process the LVDS packets without using a RAM. The downside of implementing an asynchronous design on an FPGA is the fact that the asynchronous design needs to be simulated by synchronous hardware. The synchronous simulation uses more power than is needed for a true asynchronous implementation, so ideally, the complete actuator controller would be implemented asynchronously on a dedicated chip. This implementation would have less restrictions on its design and offer the possibility of larger power reductions. If the controller would be implemented on a dedicated chip, other designs for the PWM units could be used, lowering the internal clock frequency of the slaves drastically. All intermediate logic, passing on information from the LVDS packets, would be asynchronous and switch only at the arrival of new LVDS packets. Reducing or eliminating the system clocks has an immediate effect on the total power usage of the actuator controller.

The designs with the recursive PWM units and the recursive counter are both more efficient than the original controller. The recursive PWM units take up a lot of resources from the FPGA, so the power reduction is not very large in this design. The recursive counter uses less resources, so in this case the power reduction is more significant. In both designs, the reduced power is a result of the lower clock frequency for the slave modules. The recursive PWM units and counter are easily integrated into the original controller code, but there are still some concerns regarding the accuracy and stability of the generated PWM signals.

When the master of the asynchronous controller is combined with the slaves of the recursive counter design, a total power reduction of approximately 29% can be achieved.

7.2 Future work

The following list shows a number of possible future research topics, regarding the optimization of the actuator controllers:

- The implementation of the actuator controllers using real asynchronous circuits and the effects on power usage.
- Implementing real asynchronous circuits on an FPGA without the need for synchronous simulations.
- The optimization of the electronics of the actuator controllers with respect to power usage. Examples are the use of a different type of FPGA, with lower static power dissipation or using one FPGA instead of three to reduce power overhead.
- Applying more advanced power reduction techniques, such as clock gating and FPGA voltage reduction.
- Finding a power simulation method that delivers more accurate results and gives a clear insight into the power usage of a design.
- Improving the measuring method for the power usage on the XUP development board.

Bibliography

[1] R. Hamelinck, N. Rosielle, M. Steinbuch, and N. Doelman. Large adaptive deformable membrane mirror with high actuator density: design and first prototypes. In *5th International Workshop on Adaptive Optics for Industry and Medicine. Edited by Jiang, Wenhan. Proceedings of the SPIE, Volume 6018, pp. 287-299 (2005).*

[2] R. Hamelinck, N. Rosielle, M. Steinbuch, R. Ellenbroek, M. Verhaegen, and N. Doelman. Actuator tests for a large deformable membrane mirror. In *Advances in Adaptive Optics II. Edited by Ellerbroek, Brent L.; Bonaccini Calia, Domenico. Proceedings of the SPIE, Volume 6272 (2006).*

[3] R. Ellenbroek, M. Verhaegen, N. Doelman, R. Hamelinck, N. Rosielle, and M. Steinbuch. Distributed control in adaptive optics: deformable mirror and turbulence modeling. In *Advances in Adaptive Optics II. Edited by Ellerbroek, Brent L.; Bonaccini Calia, Domenico. Proceedings of the SPIE, Volume 6272 (2006).*

[4] IEEE Computer Society. *IEEE Standard Verilog Hardware Description Language, 1364-2001.*

[5] IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual, 1076-2002.*

[6] I. Løkken. PCM-PWM analysis brief. `http://www.iet.ntnu.no/~ivarlo/`, 2004.

[7] Altera Corporation. *Cyclone II Device Family Data Sheet.*

[8] Xilinx Inc. *Xilinx University Program Virtex-II Pro Development System, Hardware Reference Manual.*

[9] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet.*

[10] National Semiconductor Corporation. *DS91D176/DS91C176 Multipoint-LVDS (M-LVDS) Transceivers, Data Sheet.*

[11] Xilinx Inc. *Xilinx ISE 9.1i Software Manuals and Help.*

[12] Keshab K. Parhi. *VLSI digital signal processing systems: design and implementation.* Wiley-Interscience, 1999.

[13] Mentor Graphics Corporation. *ModelSim User's Manual*, 2007.

[14] Jinwen Xiao. *An ultra-low-quiescent-current dual-mode digitally-controlled buck converter IC for cellular phone applications.* PhD thesis, University of California, Berkeley, 2003.

[15] Handshake Solutions. *Haste-Programming Language Manual*, 2007.

[16] Handshake Solutions. *Handshake Technology Design Flow Manual*, 2005.

[17] Emdes Embedded Systems. *Ethernet-LVDS bridge communication protocol specification*, 2007.

Appendix A

Actuator controller requirements

Authors: Rogier Ellenbroek, Roger Hamelinck

A.1 Input

1. The actuator controller communicates via LVDS, using the protocol described in [17].
2. The functionality implied by the protocol described in [17] needs to be supported by the actuator controller.
3. The number of setpoints that can be received every second is only limited by the speed of the LVDS communication.

A.2 Output

The output of the actuator controller consists of PWM signals that are suited to drive the existing actuators and accompanying analog circuitry.

1. Because the supply voltage is 3.3 V and the maximum actuator voltage 0.8 V, the duty cycle of the PWM signals only has to be set from 37.5% to 62.5%.
2. The range of 37.5% to 62.5% has to be set with a resolution of 14 bits. To accomplish this, the range of 0% to 100% may be set with a resolution of 16 bits.
3. Given the model specified in section A.4, the generated PWM signal needs to determine the actuator position in such a way that:
   - The average deviation in the steady-state is 0.
   - The RMS deviation is smaller than the movement resulting from the least significant bit of the setpoint. This is approximately 1 nm.

   All transient effects in the PWM signal, after a setpoint update, need to disappear in less than 100 μs.

A.3 Latency

The time between receiving a setpoint update via LVDS and adjustment of the PWM signal, needs to be smaller than 2 μs.

A.4 Electromechanical actuator model

Modeling the electronics and mechanics of the actuator have led to a continuous-time transfer function between the voltage delivered by the H-bridge and the position of the actuator. For the electronics, only the relative gain of the model is relevant (i.e. the gain relative to the steady-state). In this case, the following normalized transfer function (with Laplace variable s) holds:

$$H(s) = \frac{0.08814}{1.027 \cdot 10^{-23}s^5 + 8.279 \cdot 10^{-19}s^4 + 2.413 \cdot 10^{-14}s^3 + 8.331 \cdot 10^{-10}s^2 + 3.22 \cdot 10^{-6}s + 0.08814} \tag{A.1}$$

The Bode plot corresponding to this function can be seen in figure A.1. The gain at the PWM base frequency of 60 kHz is so low that the deviation of the actuator position is much smaller than the movement associated with the least significant bit of the setpoint. Improvements to the model have shown that the 60 kHz PWM base frequency may be lowered.

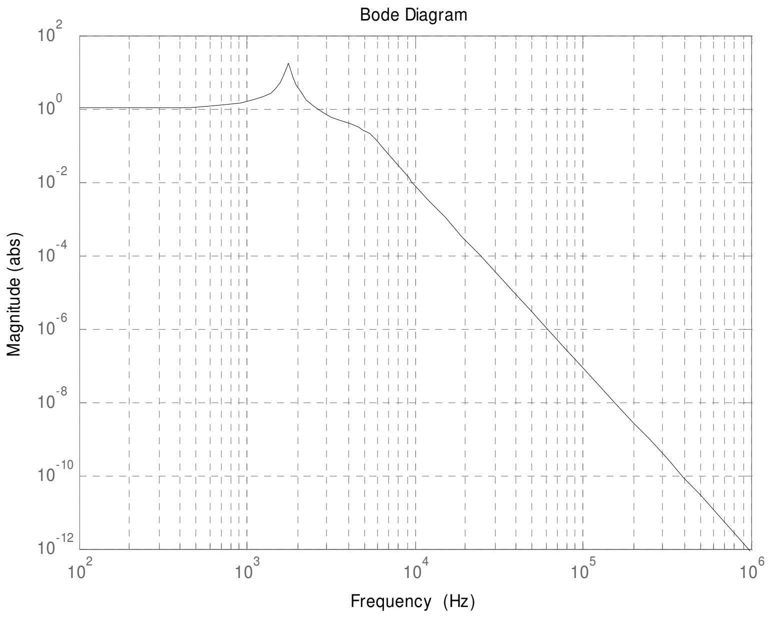

Figure A.1: Bode plot for transfer function $H(s)$

Appendix B

Heat dissipation

Author: Roger Hamelinck

B.1 Requirements

1. The surface temperature of the mirror should not be higher than 1 °C above the temperature of the surrounding air.
   - The assumption is that the heat at the mirror-side is transferred by natural convection. The typical heat transfer coefficient, associated with natural convection, is 10 W/m^2.
   - The generated heat is dissipated at the front- and backside of the actuator-plate.
   - The maximum dissipation per actuator P_{act}, is given by the following formula:

     $$P_{act} = \frac{2h}{N} \tag{B.1}$$

     In this formula, h represents the heat transfer coefficient and N the number of actuators per m^2.
   - If we choose: $h = 10$ W/m^2 and $N \approx 25000$ actuators per m^2 (at a 6 mm pitch), then $P_{act} \approx 1$ mW.
2. The total amount of power that may be dissipated to the environment is equal to 30 liters of air, with a temperature of 1 °C above the temperature of the surrounding air, every second.
   - The power needed to raise the temperature of 1 liter of air (at constant pressure) by 1 °C every second, is approximately 1 W.
   - The total amount of power that may be dissipated by the complete system (electronics and actuators) is therefore 30 W.
   - This requirement is based on a system with 8000 actuators, so the system is allowed to dissipate: $\frac{30\text{ W}}{8000} \approx 4$ mW per actuator.

B.2 Conclusion

If all of the generated heat is transferred to the surrounding air, the system is only allowed to dissipate a maximum of 4 mW per actuator. The maximum dissipation at the actuator-plate is 1 mW per actuator, so this leaves 3 mW per actuator available for the electronics.

Appendix C

Schematics for custom electronics

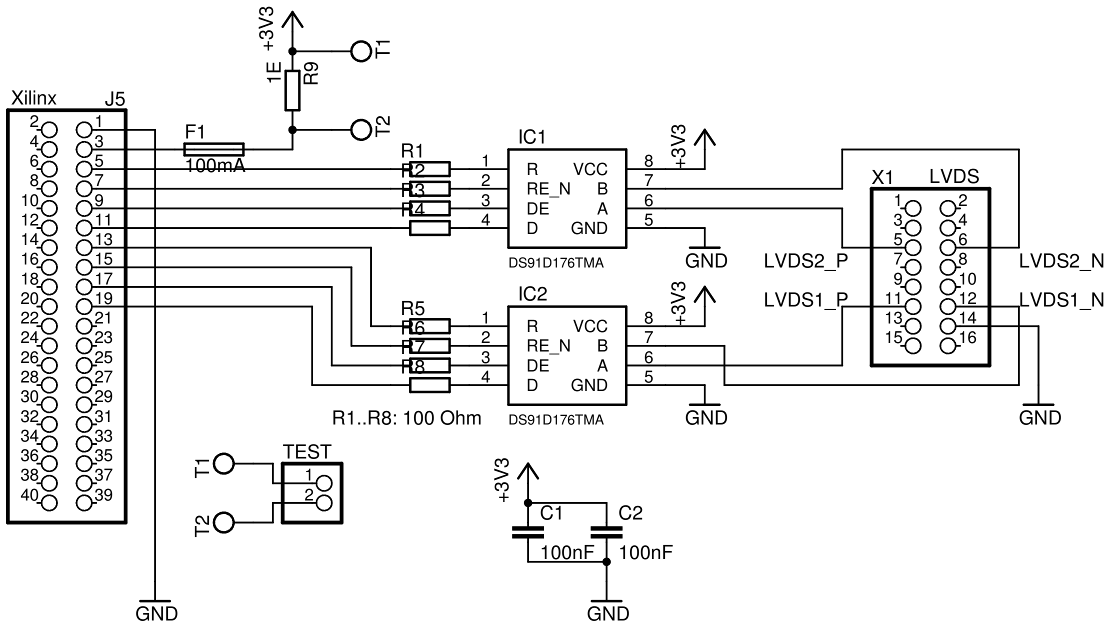

Figure C.1: Schematic for the LVDS interface

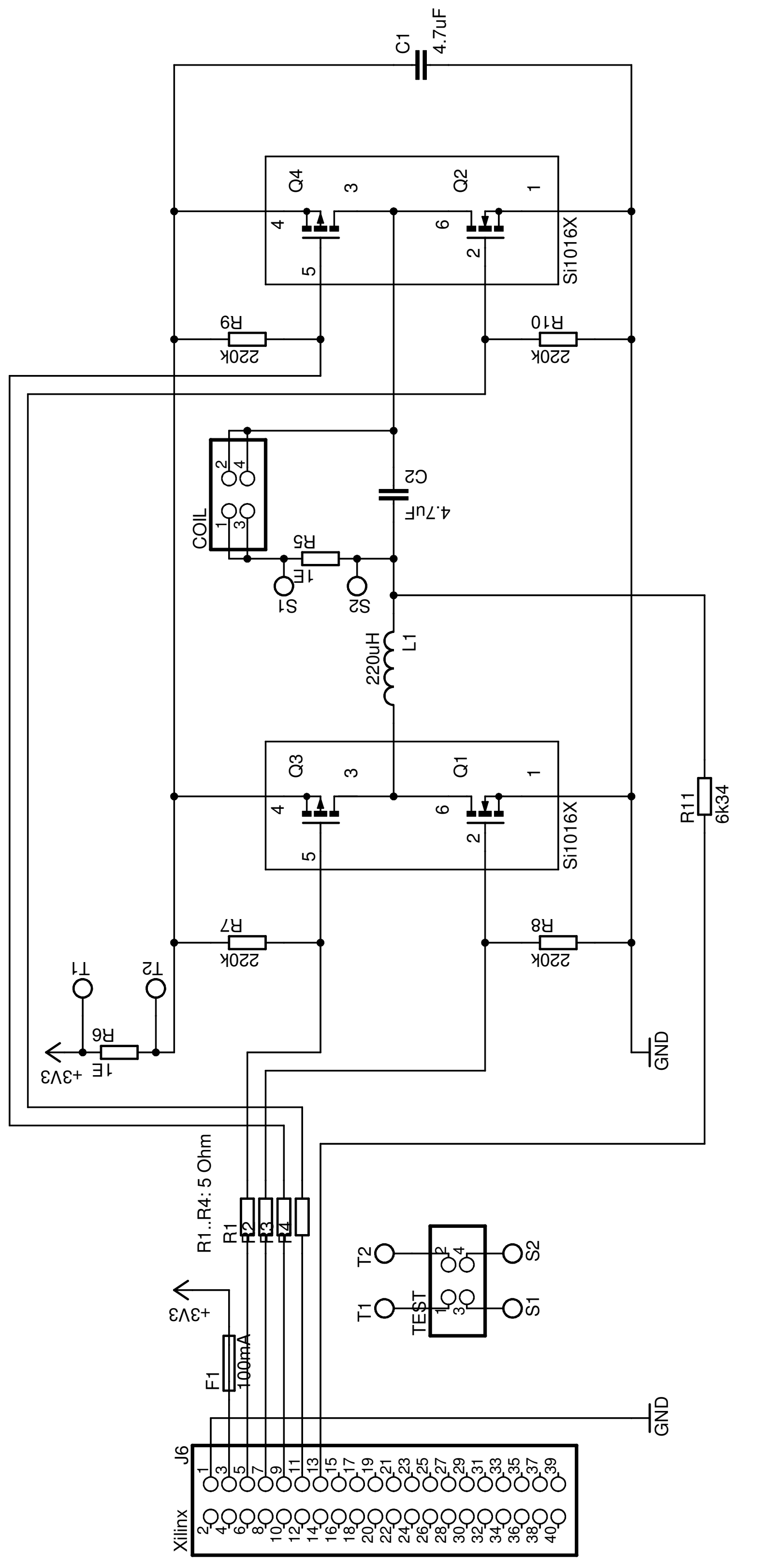

Figure C.2: Schematic for the H-bridge circuit

Appendix D

Power measurement method

The XUP development board doesn't have connectors for measuring the current through the FPGA directly, so another method for establishing the power usage of the FPGA had to be found. The development board has three independent power supplies, that generate 2.5 V, 3.3 V and 1.5 V for the components on the board. Each power supply has a connector that can be used to measure the total current drawn from the power supply. The power supplies provide power to a lot of other components besides the FPGA, so the current running through the FPGA can only be established by calculating the difference in current between a minimally configured FPGA and an FPGA configured with the design to be measured.

The code for the minimal configuration, shown in appendix F.5, doesn't perform any calculations so its dynamic power usage is 0. The difference in total power usage of a design and this minimal configuration is therefore equal to the dynamic power used by the design, assuming that the power usage of the other components on the development board doesn't change. Because the minimal configuration still uses static power, the static power of a design can't be measured by calculating the difference, so it has to be estimated with the help of information available in the data sheet of the FPGA [9]. Table D.1 shows the baseline current measurements of a minimally configured FPGA in two different speed-grades. These measurements already include the estimations of the static power dissipation.

Device	**2.5 V**	**3.3 V**	**1.5 V**
xc2vp30-6	0.0318 A	0.2566 A	0.0064 A
xc2vp30-7	0.0481 A	0.2587 A	0.0110 A

Table D.1: Baseline current measurements

Calculating the total power usage of a design P_{tot} is now just a matter of measuring the current at each power supply I_m, subtracting the baseline currents I_b from the measured currents and multiplying the resulting currents with their corresponding voltages, as shown in equation D.1.

$$P_{tot} = 2.5(I_{m2.5} - I_{b2.5}) + 3.3(I_{m3.3} - I_{b3.3}) + 1.5(I_{m1.5} - I_{b1.5}) \tag{D.1}$$

Appendix E

Haste introduction

Haste is a high-level programming language, comparable to Verilog or VHDL, that can be used to create asynchronous hardware designs. Asynchronous designs usually have low power usage because of the absence of clock signals. The Haste compiler, created by Handshake Solutions, translates the Haste code to a handshake circuit, which can be mapped to an asynchronous Verilog netlist for implementation on a dedicated chip or to a synchronous netlist for implementation on an FPGA [16]. The synchronous netlist is effectively a simulation of the asynchronous circuits, so it lacks some of the benefits of a real asynchronous implementation.

Haste supports multiple parallel processes, communicating with each other through handshake channels. To facilitate communication with the outside world, constructs are available to sample inputs directly without the use of handshakes. Variable types are created by combining the built-in boolean and range types into tuples. Haste has most of the standard programming language constructs like if-statements and repetitions, and some additional constructs to enable the communication through handshake channels.

The following Haste fragment shows an example program for a one-place FIFO buffer [15]:

```
  U8 = type [0..255]

& fifo:main proc(a?chan U8 & b!chan U8).
  begin
    t:var U8
  |
    forever do
      a?t
    ; b!t
    od
  end
```

This program continuously reads data from input channel *a* into variable *t* and then outputs it again using channel *b*.

Appendix F

Code listing

F.1 Actuator controller test bench

```
`timescale 1ns / 1ps

module main_sim;
  // Parameters
  parameter CLK = 5;
  parameter CLK_LVDS = 12.5;

  // Inputs of main
  reg clk_100;
  reg reset_n;
  reg LVDS1_RXD;
  reg LVDS2_RXD;

  // Outputs of main
  wire [3:0] led_n;
  wire LVDS1_TXD;
  wire LVDS1_TXEN;
  wire LVDS1_RXEN_N;
  wire LVDS2_TXD;
  wire LVDS2_TXEN;
  wire LVDS2_RXEN_N;
  wire [4:0] K;
  wire Z0, Z1;

  // Simulation variables
  reg clk_lvds;
  reg bit_lvds;

  // Instantiate main
  main UUT (.clk_100(clk_100), .reset_n(reset_n), .led_n(led_n),
            .LVDS1_TXD(LVDS1_TXD),     .LVDS1_RXD(LVDS1_RXD),
            .LVDS1_TXEN(LVDS1_TXEN), .LVDS1_RXEN_N(LVDS1_RXEN_N),
            .LVDS2_TXD(LVDS2_TXD),     .LVDS2_RXD(LVDS2_RXD),
            .LVDS2_TXEN(LVDS2_TXEN), .LVDS2_RXEN_N(LVDS2_RXEN_N),
            .K(K), .Z0(Z0), .Z1(Z1));

```

```
// Generate clock signals
always #CLK clk_100 = ~clk_100;
always #CLK_LVDS clk_lvds = ~clk_lvds;

// Send LVDS signal
always @(posedge clk_100) begin
  LVDS1_RXD <= bit_lvds;
end

initial begin
  // Initialize variables
  clk_100   = 0;
  LVDS1_RXD = 1;
  LVDS2_RXD = 0;
  clk_lvds  = 0;
  bit_lvds  = 1;

  // Generate reset signal
  reset_n = 1; #CLK
  reset_n = 0; #(20*CLK)
  reset_n = 1;

  // Wait a number of clock cycles before sending a packet
  #(120*CLK)

  sendPacket(0);  #(40*CLK_LVDS) // Write 0x1707 to 0x28
  sendPacket(1);  #(40*CLK_LVDS) // Write 0x1707 to 0x68

  sendPacket(2);  #(40*CLK_LVDS) // Write 0xffff to 0x20
  sendPacket(3);  #(40*CLK_LVDS) // Write 0xffff to 0x21
  sendPacket(4);  #(40*CLK_LVDS) // Write 0xffff to 0x22
  sendPacket(5);  #(40*CLK_LVDS) // Write 0xffff to 0x23
  sendPacket(6);  #(40*CLK_LVDS) // Write 0xffff to 0x24
  sendPacket(7);  #(40*CLK_LVDS) // Write 0xffff to 0x25

  sendPacket(8);  #(40*CLK_LVDS) // Write 0xffff to 0x60
  sendPacket(9);  #(40*CLK_LVDS) // Write 0xffff to 0x61
  sendPacket(10); #(40*CLK_LVDS) // Write 0xffff to 0x62
  sendPacket(11); #(40*CLK_LVDS) // Write 0xffff to 0x63
  sendPacket(12); #(40*CLK_LVDS) // Write 0xffff to 0x64
  sendPacket(13); #(40*CLK_LVDS) // Write 0xffff to 0x65

  sendPacket(14); // Send BurstWrite packet
end

task sendPacket (input integer file_num);
  reg     [319:0] filename;
  reg     [17:0] word;
  integer i;
  integer in_file;

  begin
    // Initialize variables
    word    = 18'h20000;
    in_file = 0;

    // Generate filename
```

```
      $sformat(filename, "packet%0d.bin", file_num);

      // Open LVDS packet file
      in_file = $fopen(filename, "rb");
      if (in_file == 0) begin
        $display("Unable to open packet file");
        $finish;
      end

      // Initialize word register
      word[8:1]  = $fgetc(in_file);
      word[16:9] = $fgetc(in_file);

      // Read data from packet file
      while (!$feof(in_file)) begin
        for (i = 0; i < 18; i = i + 1) @(posedge clk_lvds) begin
          // Transmit one bit
          bit_lvds = word[i];
        end

        // Read two bytes from the packet file
        word[8:1]  = $fgetc(in_file);
        word[16:9] = $fgetc(in_file);
      end

      // Close LVDS packet file
      $fclose(in_file);
    end
  endtask
endmodule
```

F.2 Passivator

```
`timescale 1ns / 1ps

module passivator (
  input  clk_0,
  input  clk_1,
  input  R_0,
  input  R_1,
  output reg A_0,
  output reg A_1);

  // Assignments
  assign A_01 = A_0 | A_1;

  // Registers
  reg t;

  initial begin
    // Initialize registers
    A_0 <= 0;
    A_1 <= 0;
    t   <= 0;
  end

  always @(negedge A_01) begin
    // Update toggle register
    t <= ~t;
  end

  // Generate acknowledge for clock 1
  always @(posedge clk_1) begin
    if (A_1) begin
      A_1 <= 0;
    end
    else begin
      if (R_0 && R_1 && !t)
        // Output acknowledge
        A_1 <= 1;
      else
        A_1 <= 0;
    end
  end

  // Generate acknowledge for clock 0
  always @(posedge clk_0) begin
    if (A_0) begin
      A_0 <= 0;
    end
    else begin
      if (t)
        // Output acknowledge
        A_0 <= 1;
      else
        A_0 <= 0;
    end
  end
endmodule
```

F.3 Recursive PWM

```
`timescale 1ns / 1ps

module pwm_rec #(
  parameter PWM_BITS = 11) (
  input clk,
  input [PWM_BITS-1:0] setp,
  output pwm);

  // Wires
  wire clk_i [PWM_BITS-1:0];
  wire pwm_i [PWM_BITS-1:0];

  // Assignments
  assign pwm = pwm_i[PWM_BITS-1];

  // Generate PWM bit modules
  generate
    pwm_0 m_pwm_0 (clk, clk_i[0], pwm_i[0], setp[0]);

    genvar i;
    for (i = 0; i < (PWM_BITS-1); i = i + 1) begin : m_pwm_i
      pwm_i m_pwm_i (clk_i[i], clk_i[i+1], pwm_i[i], pwm_i[i+1], setp[i+1]);
    end
  endgenerate
endmodule

module pwm_0(
  input  clk_in,
  output clk_out,
  output pwm_out,
  input  setp_bit);

  // Assignments
  assign pwm_out = (clk_out & setp_bit) & (clk_out | setp_bit);
  assign clk_out = clk_in;
endmodule

module pwm_i(
  input      clk_in,
  output reg clk_out,
  input      pwm_in,
  output     pwm_out,
  input      setp_bit);

  // Assignments
  assign pwm_out = ((clk_out & setp_bit) | pwm_in) & (clk_out | setp_bit);

  // Divide input clock
  always @(posedge clk_in) begin
    clk_out <= ~clk_out;
  end

  initial begin
    // Initialize clock register
    clk_out <= 0;
  end
endmodule
```

F.4 Recursive counter

```
`timescale 1ns / 1ps

module count #(
  parameter NUM_BITS = 11) (
  input clk,
  output [NUM_BITS-1:0] n);

  // Wires
  wire bit_i [NUM_BITS-1:0];

  // Assignments
  assign bit_i[0] = clk;

  // Generate counter bit modules
  generate
    genvar i;
    for (i = 0; i < (NUM_BITS-1); i = i + 1) begin : m_count_i
      count_i m_count_i (bit_i[i], bit_i[i+1]);
    end
  endgenerate

  // Generate bit assignments
  generate
    genvar j;
    for (j = 0; j < NUM_BITS; j = j + 1) begin : m_bit_j
      assign n[j] = ~bit_i[j];
    end
  endgenerate
endmodule

module count_i(
  input      clk_in,
  output reg clk_out);

  // Divide input clock
  always @(posedge clk_in) begin
    clk_out <= ~clk_out;
  end

  initial begin
    // Initialize clock register
    clk_out <= 0;
  end
endmodule
```

F.5 Minimal configuration

```
`timescale 1ns / 1ps

module main (
  output [3:0] led_n);

  assign led_n = 4'hA;
endmodule
```