BitHound

An FPGA based Logic Analyzer

Digilent Design Contest 2011

Lukas Schrittwieser
lukas.schrittwieser@aon.at

Mario Mauerer
mauerer.m@gmail.com

September 2011

Preface

This report of our group work is split into the following chapters:

Chapter Introduction explains the task to fulfill and gives an introduction to this project.

Chapter Interface Board presents the PCB providing additional features to the analyzer.

Chapter FPGA Design describes the logic design of the system.

Chapter Firmware handles the software programmed into the softcore.

Chapter Client Software shows how the program on the PC works.

Chapter Conclusion presents our results and further possible improvement.

Chapter Appendix features several schematics, drawings and a list of required Tools.

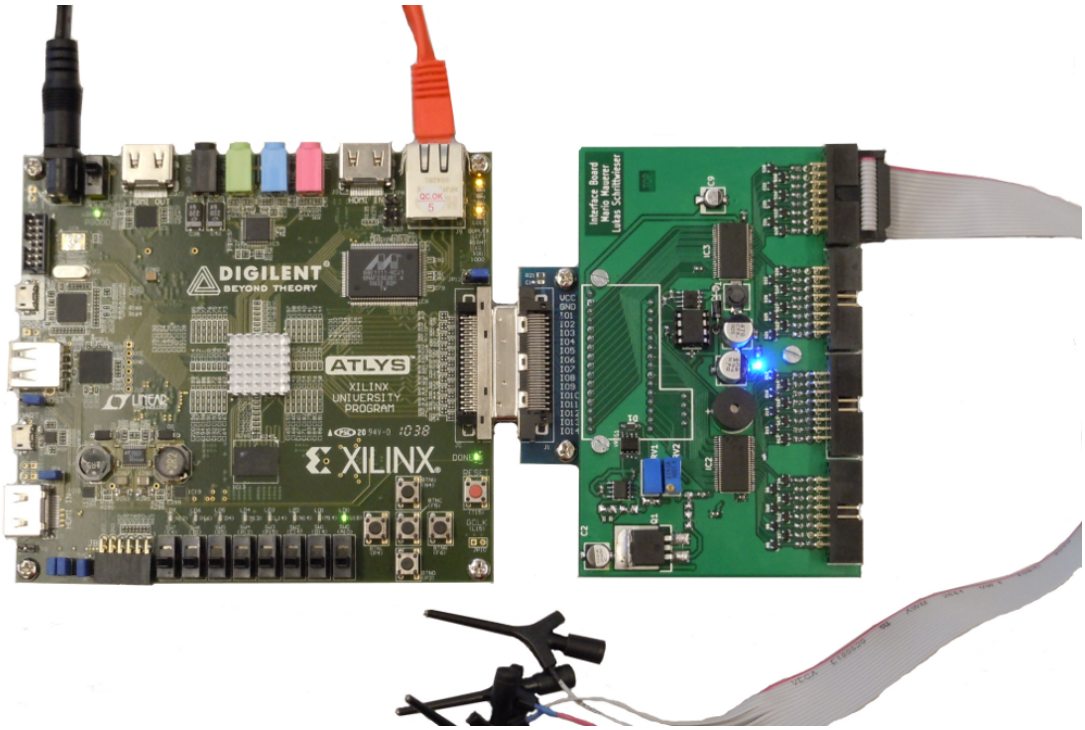

Figure 1: The BitHound Logic Analyzer without its case

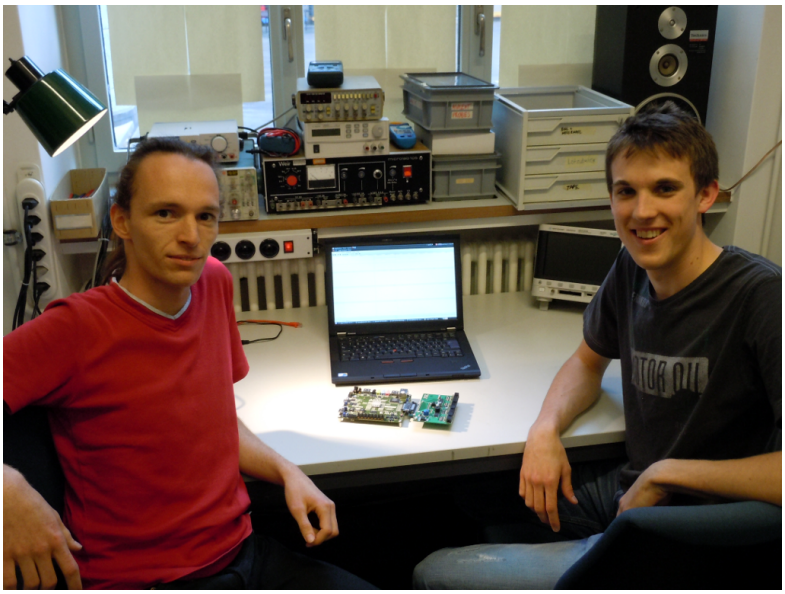

Figure 2: The Authors: Lukas Schrittwieser and Mario Mauerer

Contents

Chapter 1

Introduction

This is the documentation of the ,,BitHound" Digilent Design Contest entry realized by Lukas Schrittwieser and Mario Mauerer

In this project, we have built a logic analyzer using a Digilent Atlys Spartan-6 FPGA-Board and a self-made interface board.
A logic analyzer is a device used to display and analyze logic signal on e.g. data busses. It is a sort of oscilloscope, being capable of only displaying digital ,,low" and ,,high", but with much more channels (32 in our case).

The logic analyzer is based on an open-source project made by Michael Poppitz. (http://www.sump.org/projects/analyzer/)
That design uses a Spartan-3 FPGA as a 32-channel logic analyzer. A 1MB SRAM serves as sample memory. The data sampled is transferred via UART to a personal computer running a (self-written) Java client which displays the measurements.

The aim of our project was to improve this reference design in several points:

- Attaching a 128MB DRAM as sample memory.
- Using Ethernet to transfer the measured data to a computer.
- Building an Interface Board to attach 32 probes.
- Adapting the PC-client to our design.

Our logic analyzer fulfils our expectations and specifications. The above mentioned project goals were met and the system works very well. The analyzer is able to sample 32 channels with 200MHz or 16 channels with 400MHz. The data will then be transferred via 100MBit/s Ethernet from the 128MB DRAM to the PC client that will then display them.
We have therefore built a versatile tool that can be used to debug digital circuits.
Figure 1.1 shows the block-diagram representation of our system.

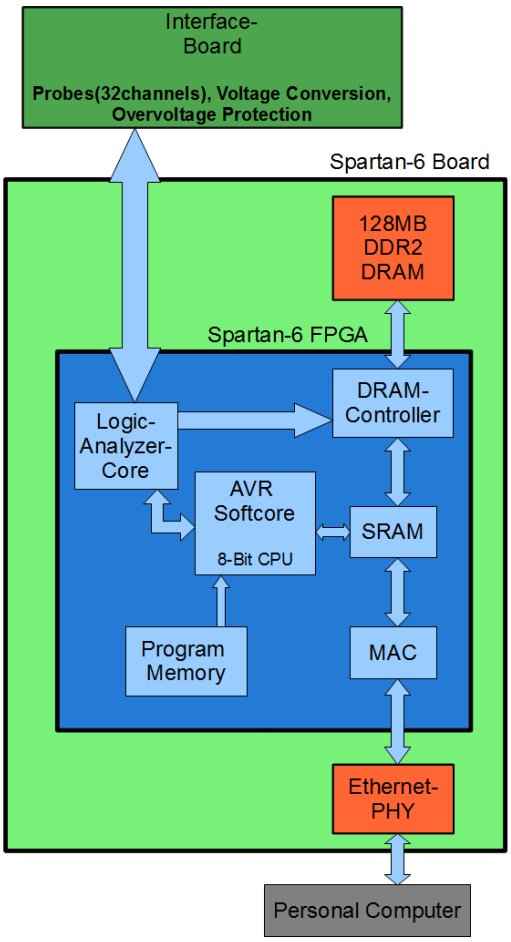

Figure 1.1: Block Diagram of the System

We chose to build this project because a logic analyzer is a well-suited application for an FPGA. This project also gave us the opportunity to mix hardware, software and digital design, which was great fun. Of course, we could have bought a logic analyzer with the same features, but building one is much more challenging and entertaining.

Chapter 2

Interface Board

The interface board connects the circuit to analyze with the Atlys Spartan-6 FPGA-board. Several aspects had to be regarded in order for the board to work properly even at higher frequencies. Therefore, a few key features are included:

- The Probes
- The Level-Shifters
- The Overvoltage-Protection Circuitry
- Microcontroller and Boost-Converter

The schematic drawings of the interface board can be found in the appendix.

These features mentioned above will now be presented in more detail.

2.1 The Probes

The probes can be connected to the interface board with flatband cables. Each cable holds 8 probes, so, there are 4 connectors to connect all 32 channels of the analyzer to the device under test(DUT). One separate probe on each cable is used as ground probe.

The probes themselves can be easily clamped to the signal paths on the device one wants to analyze. They can also be detached from the flatband cable and the remaining female connector can then be used to attach the cable to a testpoint on the DUT. Although flatband-cables are not very appropriate to conduct high frequency signals, they have proven to be a reasonable solution for this application.
This has several reasons: Firstly, every second wire in the cable carries ground potential which limits crosstalk among the signals. Secondly, the cables are not that long compared to the expected wavelengths. And, last but not least, the flatband cables are a very cost-effective solution.
Measurements with our system have showed that the flatband cables are suitable to frequencies up to approximately 50MHz.

2.2 Voltage Conversion

The level-shifters limit the input signals to a level suitable for the FPGA.
The Spartan-6 only accepts voltages up to 3.3V at its inputs, thus, there is need to adapt the voltage of the input signals, which can be as high as 5.5V.
This task is performed by specialized level-shifter ICs. The task of choosing the appropriate level-shifter was not very easy because several aspects had to be kept in mind: The input capacitance of the IC should be as low as possible to be able to transmit high-frequency signals and not to build a lowpass-filter. Additionally, the transition time through the level-shifter should be as low as possible, because then, straying in these times between different ICs has a minor effect to time delays between different channels.
It was also important that the circuitry needed to operate these shifters could be kept simple; the level-shifter should accept voltages from 2.5V to 5.5V without the need to change this input voltage range manually by the operator before the measurement takes place, this could then have lead to the possibilty of faulty operation and thus, result in damage.
Our choice fell to the ,,SN74ALVTH16244DL" from Texas Instruments. This is an active 16 channel buffer that operates on 3.3V but is compatible with 5V-logic. Its output voltage is 3.3V, thus, it is suitable for the Spartan-6. It has got a very small propagation delay of ca. 2ns and an input capacitance of only 3pF. With this level-shifter, its application got nice and simple.
The inputs of the level-shifter need to be pulled down using resistors when the probes are floating in order to bring the outputs of the level-shifter to a defined value. As these resistors have a very large value, they will not affect the measured signal in a considerable way.

This level-shifter can handle voltages up to 7V at its inputs without getting destroyed and without loosing its capability of limiting the applied voltage to 3.3V. This still leaves the possibility of damaging the level-shifter, for example if a probe gets attached to 12V. In such a condition, it is most important to protect the FPGA, therefore, no overvoltage may pass the levelshifter at all.

This circumstance made it necessary to protect the inputs of the level-shifter with an overvoltage-protection which limits large overvoltages to a maximum of 7V.

2.3 The Overvoltage-Protection Circuitry

This part of our design prevents damage to the level shifters and thus to the FPGA if an excessive overvoltage is applied to the probes which would damage and destroy the level-shifters.

As the elements of this construction have to be built directly into the signal path of each channel, it was of upmost importance that there would be as less distortion in the signals as possible. This meant, that the capacitive load to the signals had to be kept very low.

After some long and intensive brainstorming, we came up with the circuit depicted in figure 2.1

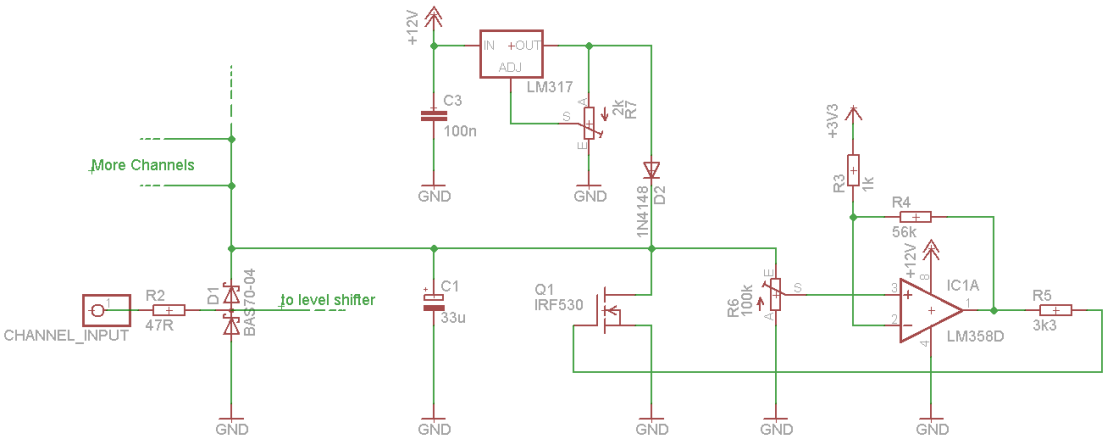

Figure 2.1: OVP-Circuitry

Each channel has one 47-ohm resistor and one BAS70-04 double shottky diode attached to its signal path. The rest of the circuit is common to all 32 channels. The BAS70 has, as it is a schottky diode, an input capacitance of only 1.5pF, which is very low, therefore it is well suited for our application.

2.3.1 Function Principle

The LM317 is an adjustable voltage regulator which is set with R7 to produce an output voltage of approximately 6V. Under normal operating conditions, this voltage lies on C1, the cathode of the upper diode in D1, the drain of the mosfet and the trimmer R6. This trimmer is set in such a way that the LM358, which is a standard operational amplifier (opamp), drives its output during normal operating conditions towards GND.
So, if no overvoltage is applied at the channels, not much happens because the upper diode in D1 is not conductive, as the signal voltage is lower than the clamping voltage. The mosfet is also not conductive because the opamp has its output near GND.

If a negative voltage is applied at the input, the lower diode in D1 becomes conductive and clamps the input voltage of the level shifter to approx. -0.4V, which is within

the allowed operating range. Resistor R2 limits the resulting current.

If, on the other hand, a positive overvoltage is applied, our circuit kicks in; The aim of the whole circuitry is to keep the voltage at C1 at about 6.5V. This is achieved with the control loop established by the opamp and the mosfet. This effects that the upper diode in D1 is conductive during an overvoltage condition and biased at a fixed voltage. As a result, the voltage at the input of the level shifter stays limited. The resistor R2 limits the current through D1 so that it does not get destroyed. The diode D2 protects the LM317, as the voltage at C1 rises above the output voltage of this voltage regulator during clamping.

The maximum overvoltage the system can clamp is limited by the power dissipation in R2. During an overvoltage situation, the affected resistors get very hot and will eventually burn if the situation lasts for too long. But this is not negative at all, as resistors are exchanged very quickly, they can be considered as fuses which trip after some overvoltage has been applied for too long. Experiments have shown that the resistors are the weakest links, the BAS-70 will not get destroyed before the resistor has burnt. The system can clamp voltages of about 12V without any resistors getting permanently destroyed.

One major concern when designing this circuit was its stability. As this is a cloosed loop system, it contains some feedback loops which had to be designed and analyzed very carefully. We did this in several ways.

2.3.2 State-Space Model

Our first approach was a system analysis using a state-space representation. We wanted to know, if our system is in principle stable and functional. Therefore, we have designed a simple model of our circuitry.
This model was built using the circuit presented in figure 2.2

The opamp is assumed to be ideal, so is the mosfet. The gate capacitance is modelled using C2. U1 ist the input- and Y1 the output voltage.
We have used this approach to see, if the circuit works in principal.

The mosfet has been modelled as a linear element: $I_D = U_G \cdot g = U_{C2} \cdot g$, where g is the transconductance and U_G the voltage at the gate. Thus, the drain current is a linear function of the gate voltage. Non-linearities such as the threshold-voltage have been neglected in this simple model.

The two resistors R5 and R4 form the feedback loop of the opamp and determine the loop gain. The output voltage of the opamp (denoted U_{OVP}) is given by this equation:

$$U_{OVP} = U_+ + \frac{R5}{R4} \cdot (U_+ - U_2) \tag{2.1}$$

Where U_+ is the voltage at the positive input terminal of the opamp.
Regarding the voltage divider formed by R2 and R3 (which generates U_+), the following simplification can be made, because the divider has a very high resistance and the opamp is assumed to be ideal (no current flowing into the input terminals): $U_+ = t \cdot Y1$, where t denotes the adjustable divider ratio. Thus, equation 2.1 can be written in this way:

$$U_{OVP} = t \cdot Y1 + \frac{R5}{R4} \cdot (t \cdot Y1 - U_2) \tag{2.2}$$

Keeping this in mind, we can determine the equations for the gate voltage of the mosfet. For capacitances, we know that $I_C = C \cdot dU_C/dt$. So, for our gate capacitance:

$$I_{R6} = C2 \cdot \frac{U_{C2}}{dt} = \frac{U_{OVP} - U_{C2}}{R6} \tag{2.3}$$

This leads to:

$$\frac{U_{C2}}{dt} = \frac{U_{OVP} - U_{C2}}{C2 \cdot R6} \tag{2.4}$$

Next, we will have a look at the input of the circuitry. The current flowing through R1 is the sum of the current into C1 (I_{C1}) and the current through the mosfet (I_D). The current through the voltage divider is neglectable, as it is very small. So, we see that:

$$I_{R1} = \frac{U_1 - Y1}{R1} = I_{C1} + I_D \tag{2.5}$$

I_D can be expressed as $U_G \cdot g$. I_{C1} can be expressed as follows:

$$I_{C1} = C1 \cdot \frac{dY1}{dt} \tag{2.6}$$

From this equation, we see:

$$\frac{dY1}{dt} = \frac{I_{C1}}{C1} \tag{2.7}$$

Now, we can solve equation 2.5 after I_{C1} and set it into equation 2.7:

$$\frac{dY1}{dt} = \frac{U_1}{R1 \cdot C1} - \frac{Y1}{R1 \cdot C1} - \frac{U_{C2} \cdot g}{C1} \tag{2.8}$$

We now have all the necessary equations to form the state-space representation of our system. As states, we will use $Y1$ and U_{C2} which are the two capacitor voltages. $U1$ and $U2$ are the inputs to our system and our states are also is the output. Using equation 2.4 and equation 2.8, we find:

$$\begin{bmatrix} \dot{Y1} \\ \dot{U_{C2}} \end{bmatrix} = \begin{bmatrix} -\frac{1}{R1 \cdot C1} & -\frac{g}{C1} \\ \frac{t}{C2 \cdot R6} + \frac{R5 \cdot t}{R4 \cdot C2 \cdot R6} & -\frac{1}{C2 \cdot R6} \end{bmatrix} \begin{bmatrix} Y1 \\ U_{C2} \end{bmatrix} + \begin{bmatrix} \frac{1}{R1 \cdot C1} & 0 \\ 0 & -\frac{R5}{R4 \cdot C2 \cdot R6} \end{bmatrix} \begin{bmatrix} U1 \\ U2 \end{bmatrix} \tag{2.9}$$

$$Y = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} Y1 \\ U_{C2} \end{bmatrix}$$

This state-space model can now be analyzed. In this simple analysis, we want to know, if the system is stable or at least stabilizable. So, we have to calculate the eigenvalues of the ,,A"-matrix:
We have to solve $det(\lambda I - A) = 0$ after λ. This leads to:

$$\underbrace{1}_{a} \cdot \lambda^2 + \lambda \cdot \underbrace{\left(\frac{1}{C1 \cdot R1} + \frac{1}{C2 \cdot R6} \right)}_{b} + \underbrace{\frac{g \cdot t \cdot \left(\frac{R5}{R4 \cdot R6} + \frac{1}{R6} \right)}{C1 \cdot C2} + \frac{1}{C1 \cdot C2 \cdot R1 \cdot R6}}_{c} = 0 \quad (2.10)$$

The two solutions to the quadratic equation 2.10 are:

$$\lambda_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (2.11)$$

From equation 2.11, we see, that the two eigenvalues can only have positive real parts (and the system thus unstable), if b would become negative. As this would represent negative values for capacitors or resistors, this solution is not applicable. Therefore, this system is stable for any transconductance g and/or voltage-divider ratio t.

2.3.3 Simulink Model

In the state-space analysis, we have neglected a bunch of very important factors. We assumed the opamp as well as the mosfet to be ideal although these are highly non-linear elements in real life.
So, there is need for a more detailed analysis of the whole system. We need to take the non-linearities of the various parts into account. This was achieved partly using simulink. The used schematic is shown in figure 2.3

The values of the capacitors and resistors are the same as we have built in our system. For how we got to these values: see section 2.5.2
R7 models the ESR of C1 and is assumed to be 0.5 ohms.
With this model, we were much more flexible when analyzing the behavior of the system. We were able to easily change resistor and capacitor values and we saw how the system reacted.
We have seen that the system is stable in this model for a very wide range of capacitances and resistors, we had to run the simulations with extreme values to get the system unstable. Figure 2.4 shows the system response to a 12V input step. One can recognize that the voltage gets very nicely limited to the desired value. With this configuration, there is no overshoot visible.

This simulink model is much more advanced than the state-space model. However, there are still better ways to investigate the system's behavior, as the simulink models still are not fully characterized. Therefore, we ran simulations using a SPICE simulator.

2.3.4 SPICE Model

We have used a SPICE simulator (microcap) to further investigate our system. We have also used this program to determine the best values for the feedback-resistors and for the

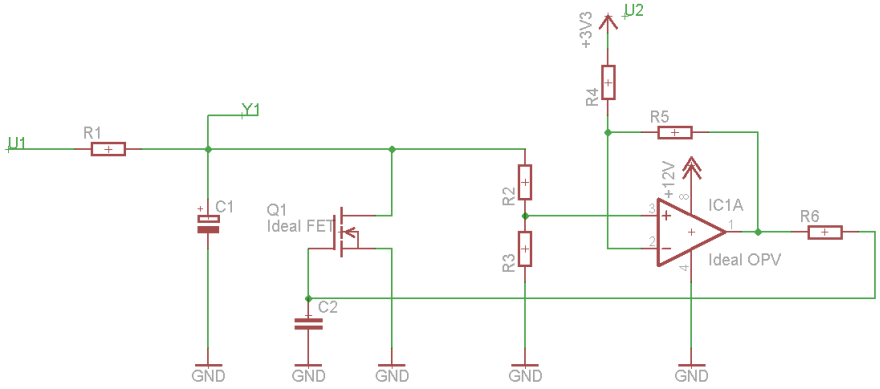

Figure 2.2: Simplified OVP-Circuit

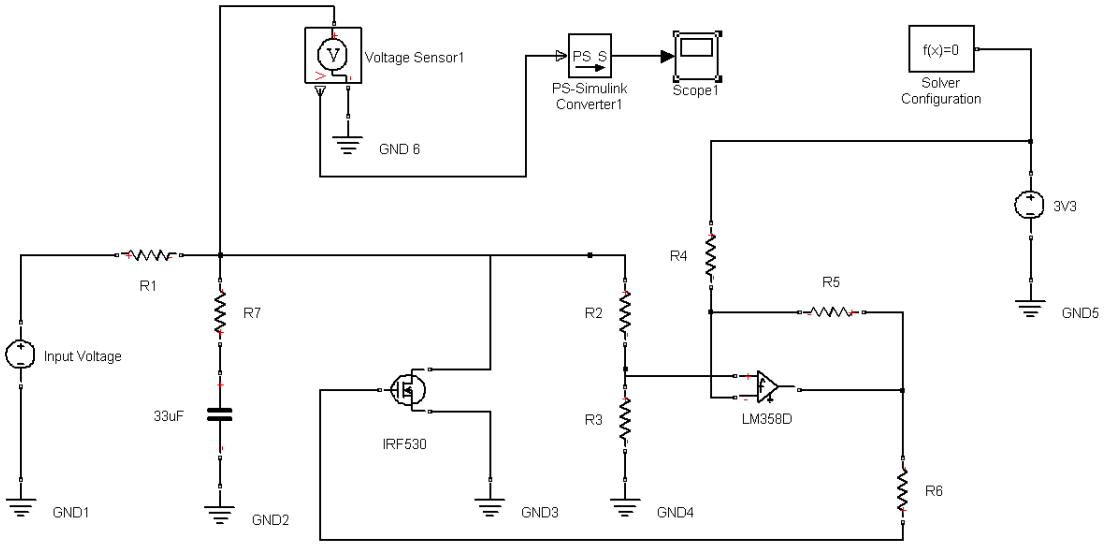

Figure 2.3: Simulink Model

other parts, so, the system got designed in an iterative process, using the simulations to improve the behavior of our protection circuit.
Figure 2.5 shows the schematic used for analyzation. V6 is the representation of the LM317. V5 generates a 12V input step. R7 is the ESR of C1.

With this tool, it was very easy to simulate different states of operation. In this example, we have applied a 12V step input simultaneously to 8 inputs. This is much harder for the system, as the inrush current is much higher and so, the system has to react quicker.

Figure 2.6 shows the step response of the system to our input.

Green is the 12V input step. Red shows the Gate voltage of the mosfet. The blue line represents the voltage at C1.
One can clearly see that the system behaves pretty good in this hard test. There is not much overshoot and the reaction is nice and quick. This is what we expect from our overvoltage protection circuit.

2.4 Microcontroller and Boost-Converter

As the Atlys Spartan-6 board from Digilent only provides a +3.3V regulated supply at the expansion connector, we had to build a boost-converter on our interface-board in order to generate a voltage of approximately 12V. This voltage is needed by the overvoltage-protection circuitry. The boost-converter is controlled by an AVR-Controller (Atmel ATtiny85) and has a small output power of approximately 360mW, however, this is sufficient for the overvoltage-protection. The controller measures the output voltage of the boost-converter (which is operating in the discontinuous-mode) and adjusts the duty-cycle using a PI-controller to keep the output voltage stable.

The AVR is also connected to the Gate of the large MOSFET in the overvoltage-protection-circuit. Thus, when an overvoltage situation happens, there is a voltage greater than 1V at the Gate of the MOSFET as the MOSFET must open its channel. This toggles a portpin of the controller and it will turn on a small piezo-buzzer that is also built in the interface board. The user will therefore be notified when a probe is attached to a excessive voltage and no resistors in the overvoltage-protection circuit will burn as the user can quickly detatch the probes.

2.5 Building and Testing

2.5.1 Assembly

After we had analyzed our circuits, we have had to build the Interface Board. The Layout was drawn using ,,KiCad". The schematics can be found in the appendix.

We were not able to match the lengths of the signal paths or the impedances, but with the expected frequencies and risetimes, this should not cause problems. The well known rule of thumb for an second order system is given in equation 2.12. So an

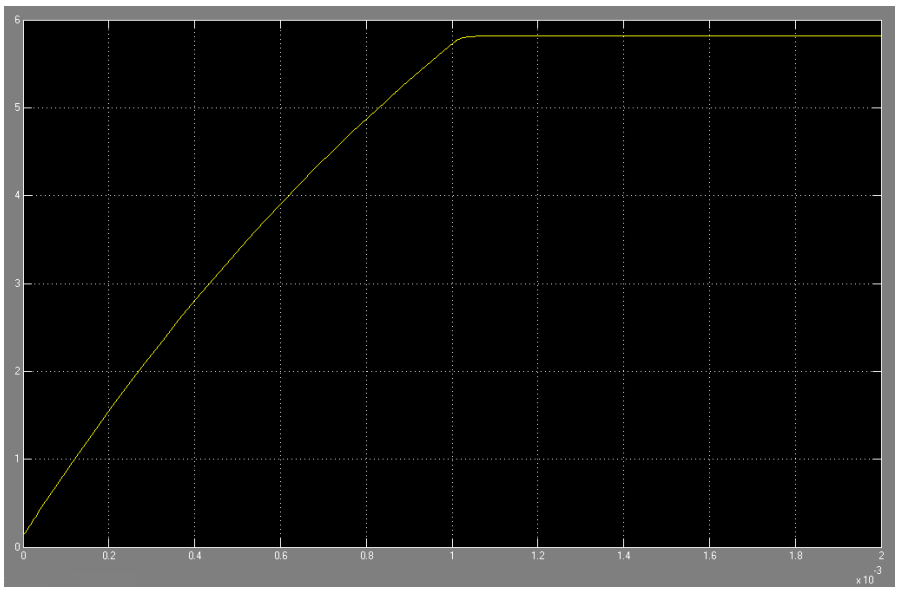

Figure 2.4: 12V Input Step

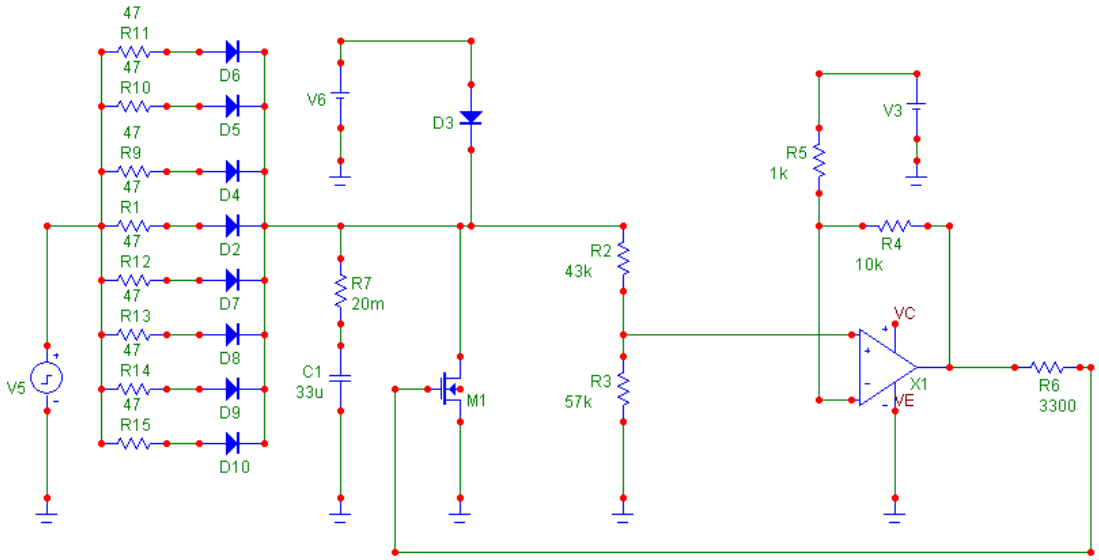

Figure 2.5: Microcap Schematic

expected minimum rise time (t_r) of roughly 2ns results in a maximum signal bandwidth of $f_{Max} \approx 175MHz$. The wavelength correspding to t_r is approximately $\lambda \approx ct_r$ where c is the speed of light in the given PCB material. For standard PCBs c is nearly $0.5 \cdot c_0$ where c_0 is the speed of light in vacuum. This gives $\lambda \approx 0.3m$ which is far above the length mismatches of the traces on the PCB.

$$f_{Max} = \frac{\omega_{Max}}{2\pi} \approx \frac{2.2}{t_r} \cdot \frac{1}{2\pi} \tag{2.12}$$

The interface board is soldered with standard pinheaders to the wirewrap-expansion board that provides enough FPGA-portpins for 32 channels.

2.5.2 Testing

After we had simulated our overvoltage protection circuit, we have also had to test it with the real circuit. So, we equipped it with the values for the resistors and capacitors we have found appropriate in our simulations. Therefore, we expected the circuit to react stable to an overvoltage step input.
Which it did not.
It was a bit frustrating, we have tested our circuit very excessively, but in real life, it just was not very stable with the simulated values, it was very easy to let it oscillate. So, we began fiddling with the values of the resistors and capacitors on the PCB until we got a stable system. These are the values we have used in the explanations in the chapters before.
With these values, our circuitry is now stable to a variety of input conditions. Figure 2.7 for example shows the response of the system to a simultaneous step input to 8 channels with 9V. Blue is the gate voltage of the mosfet and yellow the voltage at C1.

One can see that the system reacts stable.

We have also tried to get the system unstable in the simulations, but we had to assume huge parasitic capacitances to get it unstable, we never have these values on our PCB. So we think that the simulations proved wrong because they still do not model all parameters of the components, an opamp is an extremely complicated circuit and even the spice models can not be perfect. This is most probably why it never oscillated in the simulations, but did so in real life.
But we have adjusted the values so that we now have a stable overvoltage-protection on our PCB.
This was a very valuable experience, we have learned, that testing is very important as no simulation can approximate the reality good enough.

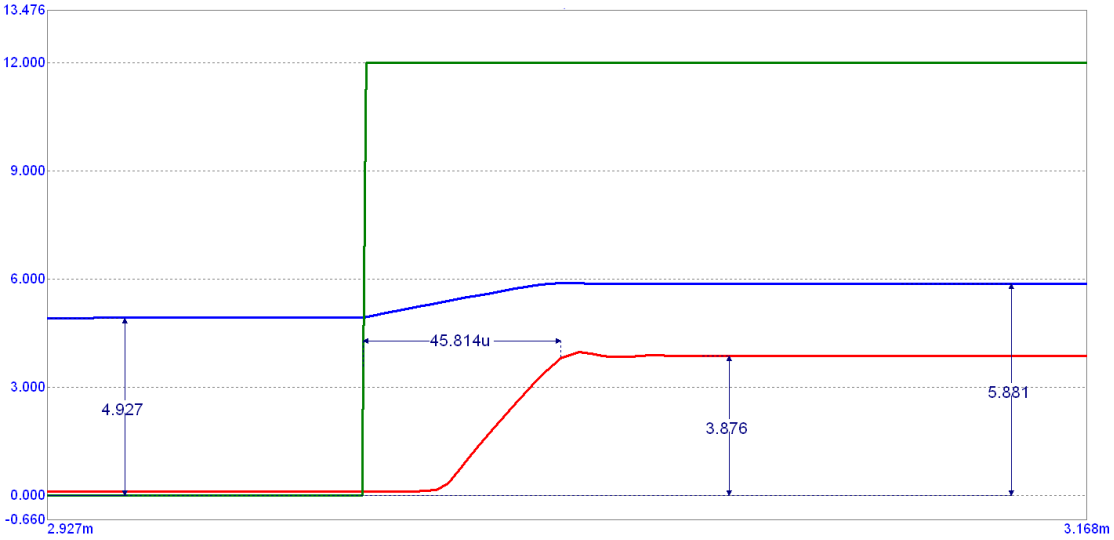

Figure 2.6: Step Response

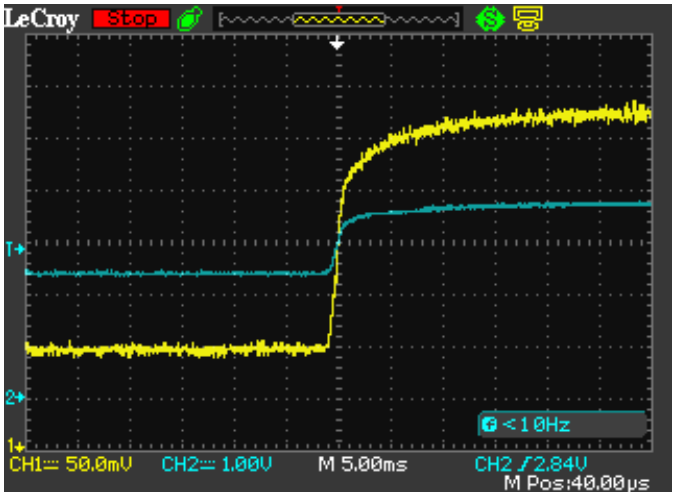

Figure 2.7: System response of the real system

Chapter 3

FPGA Design

3.1 Introduction

As BitHound uses Ethernet to communicate with the computer displaying the sampled data, it needs an integrated microcontroller. This is due to the fact that the TCP/IP protocol stack used in computer networks is too complex to be implemented in pure hardware. The OpenCores website (www.opencores.org) lists more than 110 different open source CPU projects, so there are plenty of cores to choose from. However some of these are marked as unstable, or come only with an assembler but no C compiler. After removing these cores, we've looked for architectures known to us because using a familiar system usualy saves the time of getting to know to a new CPU and compiler. So we were left with two candidates: the AVR-Core compatible with an AtMega103 from Ruslan Lepetenok and the OpenRisc1000. The first is an 8 bit microcontroller whereas the second is a 32 bit CPU. Both are supported by the GNU Compiler Collection (GCC). OpenCores also offers minsoc, an open source system-on-a-chip (SoC) for OpenRisc1000 which would suit our needs quite well. However it needs considerably more FPGA ressoruces than the AVR. Although it would fit into the Atlys' FPGA, we think the extra effort is not worth it. A quick examination showed that it might be possible to port the system to similar boards with FPGAs as small as an XC6SLX16. So the AVR was chosen because it is marked as stable, the architecture is well documented, a free C compiler is available and it requires rather few ressources. Note that the project of Ruslan Lepetenok is a whole AtMega103 microcontroller rather than a CPU alone. This doesn't fit our needs as we need different IO modules than the ones offered by the AtMega103. So we simply took out the CPU and embedded it into our own system-on-a-chip. After choosing a CPU, a bus system had to be choosen for the SoC. As we use other cores from OpenCores, like an Ethernet media access controller (MAC) and a serial interface (UART), we've chosen the Wishbone bus as specified by OpenCores (http://opencores.org/opencores,wishbone). A shared bus interconnect following the Wishbone specification is available as well from the OpenCores website: http://opencores.org/project,wb_conbus.

3.2 System On A Chip - SoC

The top level block diagram of the FPGA logic is shown in figure 3.1. The chosen AVR CPU uses a harvard architecture with a total of three busses: programm, data and input-output (IO). The programm memory is connected directly to the CPU whereas the data and IO interfaces connect to multiple cores over separate Wishbone busses. The whole SoC is clocked with 40MHz, this frequency was determined by a synthesis and implementation run of the Xilinx ISE toolchain. As power consumption is not critical a clock frequency as high as possible is desireable to get fast data transfer from the device to the Ethernet MAC and therefore to the client software.

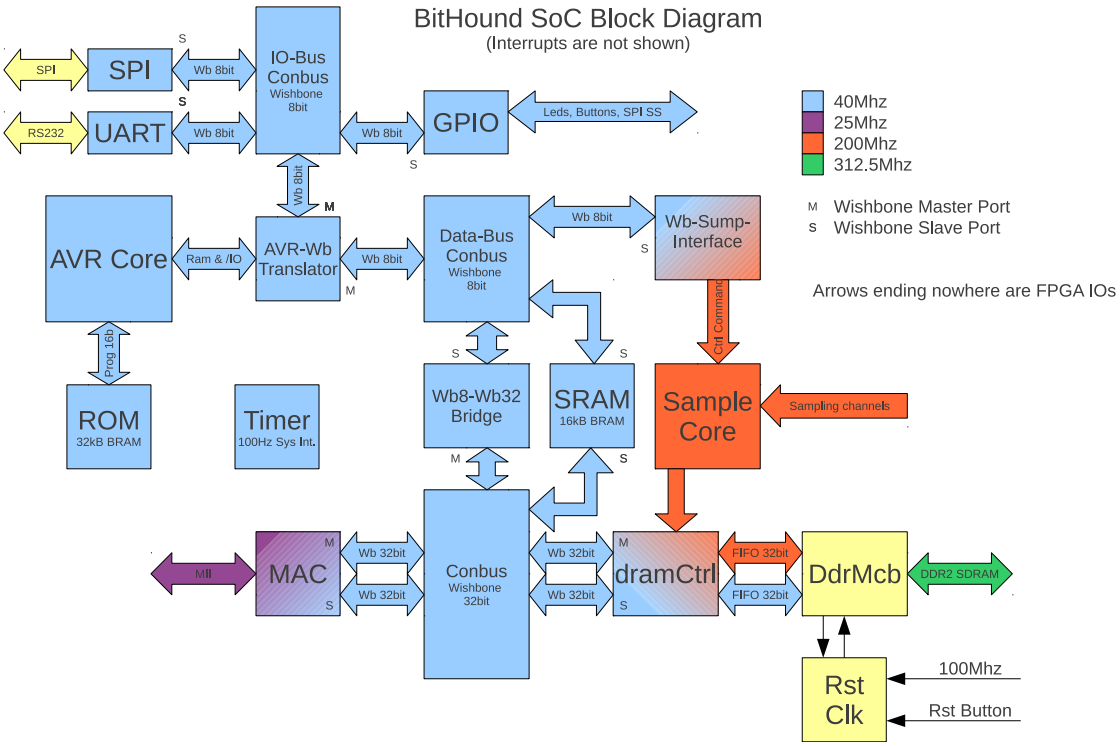

Figure 3.1: Block Diagramm of the SoC

3.3 Program Memory

3.3.1 Implementation

The programm memory of the CPU is implemented with FPGA Block RAM (BRAM) devices. This BRAMs are preloaded with the AVR firmware upon FPGA configuration

and never changed once the FPGA is running. Note that the CPU requires a combinatorial program memory, however the BRAM devices of the FPGA contain a register on the address lines. To solve this problem, the memory is clocked with an 180 degree phase shifted, i.e. inverted, clock. This means that the BRAM updates it's address register in the middle of the CPUs execution cycle and then has half the period time left to send the requested instruction towards the CPU. With this technique, the BRAM mimics a pure combinatorial memory, however the achievable clock frequency decreases.

3.3.2 Initialization

To simulate the SoC, the program memory has to be preloaded with machine code for the CPU. This requires the creation of initialization constants for the BRAM instances of the program memory. To do this, Xilinx offers a tool called *data2mem* (Xilinx UG658). It takes a *Block RAM Memory Map* (BMM) file and an executable elf file from the linker to create the initialization constants.

The same tool can also be used to merge the preloaded BRAM data with the bitstream without re-running the synthesis and implementation tools. This speeds up the firmware development considerably as running the synthesis and implemention takes between ten and 45 minutes.

We created simple shell scripts for both use cases which can be found in the folder named *fpga*. There are Windows and Linux versions of both scripts. In the Linux version, the path to the data2mem tool has to be adapted to the actual install path of the Xilinx tools. On Windows, the script has to be run from inside the *ISE Design Suite Command Prompt*. The path to the executable (in elf format) has to be passed as the first parameter for both scripts.

3.4 AVR To Wishbone Translator

This entity allows the AVR softcore to communicate with the rest of our SoC. This entity converts the AVR specific data-and-IO-bus to standard Wishbone master interfaces. The entity forms two Wishbone-interfaces, as we have to connect two busses from the AVR to our System; the IO-bus and the data-bus.
The peripheral devices connected in the original AVR microcontroller answer within an deterministic time and the bus therefore has no ,,wait" or ,,stall" signal. Unfortunately this is the case with Wishbone where an *Ack* line signals the end of a transfer. So we don't know how many clock cycles a bus transfer will take until the slave has finished its operations. Therefore, the CPU has to be stalled during an Wishbone transfer. Luckily, the AVR softcore we're using offers this functionality. The CPU has a *cpuwait* input which halts the entire cpu. This is exactly what we need because now, as soon as the AVR initiates a transfer on one the two mentioned busses, the entity stalls the AVR until the transfer on the wishbone-side is finished.nWith this system, the changed bus system is transparent to the CPU and therefore the firmware. Devices connected to the wishbone busses can be accessed like normal AVR peripherals using the appropriate

assembler instructions or C macros.
Another concern was the fact, that the CPU has only one 8-bit datapath for both busses. Therefore, the translator has to multiplex the incoming data, depending on which bus the transaction took place. The outgoing data is simply forwarded to both busses.
The IO-Bus has a 6 bits wide address and the data-bus a 16 bits wide address. Thus, both wishbone-interfaces have each the same address-widths. Because the AVR has only one 8-bit datapath, both wishbone-interfaces use 8 bits wide data busses.

3.5 IO Bus

The AVR IO bus has 6 address bits which results in an address space of 64 bytes. This is not enough to map all peripheral devices to this bus as the MAC alone has an address length of 11 bits on its configuration interface. Therfore, some peripherals are memory mapped. The memory map of the devices connected to the IO bus is shown in figure 3.2.

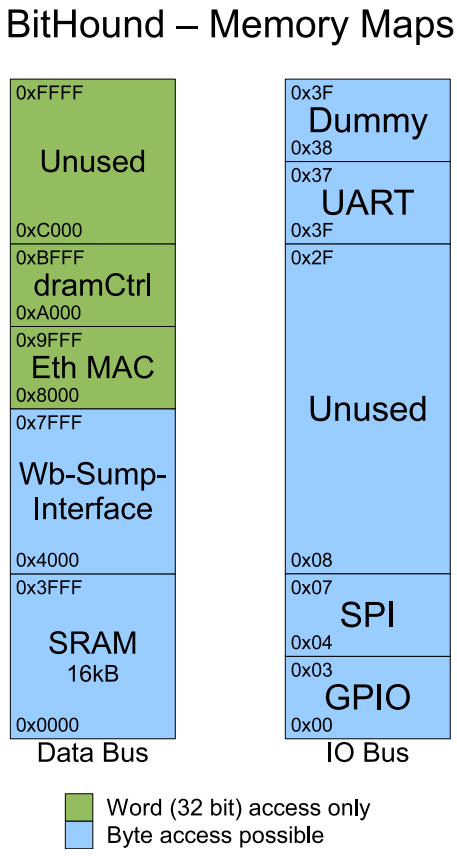

Figure 3.2: Memory maps of IO and data bus

3.5.1 UART

The universal asynchronous receiver and transmitter (UART) is a clone of the well knwon 16550 UART, which is widely used among PCs and embedded systems. The core was downloaded from the OpenCores website. It comes with it's own documentation which can be found in the directory *fpga/uart*. The serial receive and transmit signals, and optionally the clear-to-send (CTS) – request-to-send (RTS) handshaking pair are connected to the corresponding pins of the USB-to-UART IC on the FPGA board. This allows for a communication between the PC and the firmware, to transmit debugging information for example.

3.5.2 General Purpose Input Output - GPIO

For debugging purposes the LEDs on the FPGA board can be controlled by the firmware. This is done via a simple GPIO controller. Furthermore it is possible to read back the pushbuttons and switches on the board. Further it generates the *slave select (SS)* signals for the SPI interface.

3.5.3 Serial Peripheral Interface Controller - SPI

A simple SPI controller from OpenCores (OpenCores simple SPI) is used to communication with external devices. Currently, it is used to control 6 shift registers (74hc595) which drive 32 LEDs and an LCD-Display. The LEDs indicate the current state of the 32 sampling channels and the LCD displays the assigned IP-address. By using this approach, the firmware has full control over the LEDs and the LCD. The SS (slave select) signals for the SPI interface are not created by this controller but by the GPIO.

3.5.4 Dummy Device

One speciality of the AVR architecture is that it forwards internal accesses to the hardware stack pointer on its IO bus. In an AVR microcontroller from Atmel(r), the stack pointer is modified via the IO bus. This is handled internally in our CPU core, however it still outputs the bus access onto the IO bus. This would cause an endless CPU stall if no slave device ever acknowledges (Ack) the bus transfer on the Wishbone bus. Therefore, a simple dummy device was implemented and mapped to the corresponding IO address location. It simply generates an Ack for all incoming transfers without doing any data processing.

3.6 Data Bus

The AVR data bus has 16 address lines so it can address up to 64kBytes of memory or peripheral devices. The data bus was split into an 8 bit and an 32 bit section as the CPU has an 8 bit interface while the MAC only supports a 32 bit wide bus. Therefore, a bridge interface was created which forwards transfers from the 8 to the 32 bit bus.

3.6.1 Data Memoy - SRAM

A 16kBytes large data memory is available to the system. It stores the CPU stack, global variables, received Ethernet frames as well as Ethernet frames for transmission. To get the highest performance of the available dual port block memory in the FPGA, the memory is connected to both, the 8- and the 32 bit, busses. This allows the DMA masters on the the 32 bit bus (the MAC and the dram controller) to access the sram concurrently with the CPU on the 8 bit bus. As all the sampled data will be transferred over this DMA channels, this is a rather time critical transfer. Therefore the 32 bit Wishbone slave interface should be as fast as possible. For a single cycle transfer, a memory with a pure combinatorial read access would be neccessary. This follows directly from the Wishbone specification. However, the Block RAMs of the FPGA always have built-in address registers. To save the extra clock cycle needed by the register, the Block RAMs are clocked with an 180 degree phase shifted clock. This allows a single cycle access, however, it effectively cuts the available cycle time in half. The same technique could be used on the 8 bit interface too, however, timing problems arise when implementing the design. This is due to the fact that the 8 bit side has to use an additional 1-of-4 multiplexer to map the internal 32 bit wide data interface to the 8 bit bus.

3.6.2 Wishbone Sump Interface

As a Wishbone bus is used as interconnection between the different modules on the SoC, the logic-analyzer core needs a wishbone-interface in order to communicate with the CPU.
The entire logic analyzer core is controlled with an 1 byte opcode and an optional data value (4 bytes). Rhese commands, are used to ,for example, setup the trigger, the sampling rate, the number of samples to take etc.
In the original implementation of the logic analyzer (Sump), these commands were transferred from the PC via UART to the analyzer. Now, with BitHound using only Ethernet as the single interconnect to the PC, this isn't the case anymore. All the commandsare transferred via Ethernet to the firmware which forwards them in the appropriate form to the wishbone sump interface.
Thus, we had to attach a Wishbone interface to the logic analyzer which gets the opcodes and the data from the CPU and transfers them in the appropriate way to the analyzer.

This interface has 9 registers, each one byte wide. They can be accessed with the Wishbone bus; One byte is for the opcode, 4 bytes are for the data field and 4 bytes allow the firmware to read the current sampling channel states. Therefore, the address port is 3 bits wide. As the Wishbone-bus is 8-bit wide, the CPU has to make 4 writes to this interface in order to fill the 32-bit wide data signal.
Address 0 to 3 of the wishbone bus are for the four data-bytes. The single bytes will be placed in the correct location in the 32-bit signal using little endian encoding. So address 0 represents the least-significant byte and address 3 the most-significant. Note that the AVR arcitecture is neither little nor bigendian per se as it is an 8 bit architec-

ture, however the GCC compiler for AVRs defines a little endian sturcture for datatypes larger than one byte. A write to address 4 will place the data into the opcode-byte. As soon as this write is made, the interface asserts a special execute-signal to the logic analyzer for one clock cycle. Then, the analyzer processes the data stored in the opcode- and data-bytes and performs the appropriate actions. It is also possible to only access the opcode-byte without writing to the data-bytes. In this case the data loaded with the last command is used. The addresses 8 to 11 are read only locations which allow the CPU to directly read the current state of the 32 sampling channels. This is used by the firmware to display the channel states on 32 LEDs.
As the logic analyzer works with a different clock than the CPU, this interface performs a clock-domain crossing. The data from the ,,slow" Wishbone-side has to be transferred to the ,,fast" analyzer-side. This is achieved with registers. This works, because the two clocks are phase synchronized as they are generated by the same PLL circuitry.

3.6.3 Wishbone 8 - Wishbone 32 Bridge

The Wb8Wb32Bridge entity groups four individual transfers on the 8 bit bus into one transfer on the 32 bit bus. The two least significant bits of the address input from the 8 bit bus are used to group the four transfers. For a read cycle, the first transfer on the 8 bit bus, which has the two lowest address bits set to zero, causes a read on the 32 bit bus. The address from the 8 bit bus is passed to the 32 bit bus. The result of this read is stored in a register inside the bridge. Further reads which have two least significant address bits not equal to zero are completed by the bridge without a transfer on the 32 bit bus. Write transfers store the data inside the bridge. The last write cycle, which has the two least significant address bits set to 11, starts the write cycle on the 32 bit side. As the AVR is an 8 bit architecture, it does not specify the endianess of word data types. However the GCC C compiler uses little endian mode, so this bridge uses the little endian format too. This creates some limitations:

- All accesses (read and write) to the bridge MUST be word aligned, i.e. the last two bits of the data words address have to be zero.
- Only word transfers are possible, reading a byte or a halfword from the 32 bit bus is not possible and may cause data corruption.
- The data has to be in little endian format: the least significant byte has the last two address bits set to 00 and is transferred frist on the 8 bit side.

3.6.4 Media Access Controller - MAC

To use the Ethernet interface a media access controller (MAC) is neccessary. It takes care of Ethernet package reception, (re-)transmission and does the frame checksum calculations. Again, there are several projects on the OpenCores website to choose from. We took the ethmac core (http://opencores.org/project,ethmac) because it is marked

as stable, has an integrated DMA module, a Wishbone interface and has been used in commercial projects. The core comes with extensive documentation explaining all registers and features of the core. However, the core expects the data inside the memory to be in big endian format while we use little endian throughout the whole project. Therefore, the byte order of the master interface, which loads and stores data to and from system memory, is reversed in hardware. This means that the least significant data byte of the MAC is connected to the most significant data byte of the Wishbone 32 bit bus and vice versa. Note that this endianess conversion could be done in software too, however, this would create a lot of computational overhead which could decrease the system performance.

3.6.5 Dram Controller - dramCtrl

The Atlys FPGA board offers a 1GBit DDR2 SDRAM chip from Micron Technology Inc. which is used to store the samples. Compared to static RAM, the DDR2 SDRAM interface is rather complex, so a special controller is needed. It has to take care of various timing constraints, memory refreshing and a special startup calibration. Luckily, the Spartan-6 devices offer so called Memory Controller Blocks (MCB) which perform these tasks. The memory would allow bus speeds up to 400MHz, however, the MCB of our device (speed grade -2) is limited to 312.5MHz so this bus frequency will be used. The MCB has up to six individual data- and command-FIFO ports which mask the complexity of the connected RAM chip. To use the MCB, an IP-Core has to be created in the Xilinx ISE design suite using the *coregen* utility. Inside our design, the entity dramCtrl (dramCtrl.vhd) is the interface between the MCB and the sampling core as well as the microcontroller system. It uses two 32 bit ports of the MCB, one to store the samples arriving from the core, the other to transfer the memory contents to the microcontrollers data memory.

Interface to the sampling core

The interface towards the sampling core is straight forward, it consists of three signals:

- ClkxC : input : 1 bit
- DataxDI : input : 32 bits
- WritexSI : input : 1 bit

ClkxC is the clock signal for the sampling core, it is running at 100MHz. DataxDI is the sampled data from the core which will be stored into the memory if WritexSI is high during a rising edge of ClkxC. WritexSI might be assigned high at any time, the bandwidth of the memory interface is high enough to store samples arriving at every ClkxC edge as to following calculation shows:
The DDR2 memory is clocked at 312.5MHz and has a bus width of 16 bits. Because both edges of the memory clock are used, this yields a throughput of 312.5 MSamples/s. This is well above the 200 Msamples/s generated by the sampling core which runs at 200MHz

Precharge	(tRP)	12.5ns
Autorefresh	(tRFC)	127.5ns
Activate	(tRCD)	12.5ns
Total Refresh Time		152.5ns
Refresh Interval		7800ns

Table 3.1: Selected Timing Properties of MT47H64M16-25E DDR2 SDRAM

clock frequency. The bandwidth overhead needed for row activation and precharging doesn't play a significant role here as we write data perfectly in order, with incrementing addresses. So, while one bank is precharged, the seven other banks of the memory can be written. Another simple calculation shows the bandwidth needed to periodically refresh the memory. To perform a refresh the banks have to be precharged, an autorefresh command has to be executed and the needed bank has to be opend again. The timings for these commands as given in the datasheet are listen in table 3.1. The resulting total refresh time of 152.5ns takes about 1.96% of the 7800ns long slot between two refresh cycles.
The sampled data arriving on the interface is first put into a FIFO of the MCB, the number of samples currently stored in the FIFO is counted by the dramCtrl entity. Once a threshold is reached, a write command is sent to the command FIFO of the corresponding MCB port which starts the data transfer from the write FIFO to the actual memory device. Generating an individual write command for each sample is not possible as the write FIFO is 64 samples long whereas the command fifo can only store 4 commands. A threshold of 8 words was choosen as compromise. During the 152.5ns long total refreshing time of the memory, up to 31 samples might arrive from the logic analyzer. This is well below the capacity of the data FIFO so a sufficient security margin is available.

Interface to the microcontroller subsystem

Once the samples are stored in the memory, they will be sent via Ethernet to the client software on the PC. To build the necessary packet for the network transfer, the samples have to be copied from the DRAM to the data SRAM of the microcontroller subsystem. To offload the cpu, this is done by direct memory access (DMA). Therefore, the dramCtrl entity has two 32 bit wide bus interfaces, which both follow the Wishbone specification, which is used throughout the whole microcontroller system. The slave interface receives configuration information from the microcontroller, the master interface transfers the samples from the dram to the Wisbone bus.

Configuration Port (Slave) The slave port provides read- and write access to 5 registers which control the DMA transfer. All accesses to the slave port have to be 32 bits wide, byte or halfword accesses are not supported. For 5 registers 3 address bits are needed, these are the address bits 4 down to 2 of a byte-addressed system. An overview

Name	Address Bits 4..2	Offset in Bytes	Read/Write
Config and Status	000	0	r/w
Source Address	001	4	r/w
Destination Addr	010	8	r/w
Transmit Length	011	12	r/w
Sampling Pointer	100	16	r only

Table 3.2: dramCtrl Configuration Registers

Bit	Read/Write	Name	Description
31..4	–	unused	write 0, read as 0
3	r/w	Err-Flag	Set when a wishbone bus error is detected during a DMA transfer.
2	r/w	Done-Flag	Set when the requested transfer is finished (Transmit Length has reached 0)
1	r/w	IE-Flag	Interrupt output is set when IE-Flag is high and Done- or Err-Flag is high
0	r	Busy-Flag	Goes to high during a transfer, automatically cleared when transfer is done

Table 3.3: dramCtrl status register bits

of the registers is given in table 3.2.
A transfer is set up by first writing the source and destination addresses. The source address referes to the DRAM, the destination address to the address on the wishbone bus. Afterwards the number of transfers to be execute is written to transmit length register. This automatically starts the DMA engine. Both address registers are autoincremented with each successful word transfer, the length register is decremented until it reaches zero. When the transfer is done, an interrupt might be generated depeding on the IE Flag in the configuration register.

Configuration and Status Register Currently this register implements the four configuration and status flags listed in table 3.3.
The interrupt output is high as long as either the Err-Flag or the Done-Flag and IE-Flag are high. The Err- and Done-Flags are set by hardware and can be cleared by writing a 1 to the corresponding bit location. They can't be set via the Wishbone bus.

Source Address Register This register holds the DRAM byte address of the next data (i.e. analyzer sample) to be transferred over DMA. This is a pointer in the DRAM address space. It is 27 bits wide to address the 128MByte long sampling memory. The unused bits should be set to zero. The counter overflow for autoincrement is handled correctly in hardware, meaning the register goes from 0x07ffffc to 0x00000000. The source address has to be word aligned, which means the lower two bits have to be zero.

Therfore they are hardwired to zero, i.e. they can't be written to 1.

Destination Addr Register This register holds the destination address where the transfered data will be stored, so it is in the Whishbone address space. The DMA module does not support unaligned transfers so the destination address has to be word aligned. Therefore the lower two bits of this register are hardwired to 0.

Transmit Length Register Writing this register immediately starts the DMA transfer. The source and destination address must be valid once this register is written. Every successful word transfer decrements this register by one. So the register takes the number of words to transfer, not the number of bytes. Once the register is set to a value other than zero, it should not be rewritten until all DMA transfers are done.

Sampling Pointer The sampling core uses the dram as a ring buffer. This register holds the dram address that will receive the next sample. The value of this register is needed to calculate the source address of a DMA transfer. Note that this register is read-only, the sampling pointer can't be modified by the firmware. Only a hardware reset clears the sampling pointer to zero.

3.7 Sampling Core

The Sampling Core is the actual logic analyzer itself. Some parts of this module have been taken over from the original Sump design. However, most parts have been rewritten to adapt it to its new surrounding and to improve the clock performance. To allow higher sampling rates, a new, more flexible trigger was implemented. The block diagram of the original implementation of the logic analyzer can be found in the Appendix. All we are using from this design is the ,,core" module. This is the heart of the whole analyzer; it offers a configurable trigger, synchronization logic and a simple glitch filter.
The core also contains a decoder which decodes the opcode and forwards the commands and control data to the correct modules so they get correctly initialized.

3.7.1 Decoder

This module decodes the opcodes sent by the firmware by using a rather simple demultiplexer. The incoming opcode is decoded into a variety of enable signals. These signals are used to control the sampling cores configuration registers. It therefore defines the meaning of the numerical opcodes used in the firm- and software. It further implements a register to store configuration flags which modify several core functions, like sample clock source or glitch filter.

3.7.2 Trigger

The trigger consists of a configurable number of identical, independent, chained units. The units can be configured by the firmware using 5 registers. The *select unit register* is

used to specify which unit is configured by the other four. The default implementation has 8 units, the first one has the index 0. Each unit consists of a compare block, chaining logic, a delay counter and fire logic. To allow high sampling rates, each unit is pipelined, therefore several samples are processed simultaneously by each unit. The basic diagram of one trigger unit is shown in figure 3.3. The registers at the outputs of the unit are clocked by the sample clock and allow a series connection of any number of units. As the samples are processed in a pipelined manner, the fact that the units are serialized is hidden from the user.
After a reset, the trigger is in an idle state and can be configured by the firmware. When the trigger receives the *arm* signal from the firmware, it changes to an active state and monitors the samples. Reconfiguring the trigger in the active state is possible but highly discouraged. Once the trigger has fired, it has to be resetted using the *reset* command before it can be rearmed again. For more details about the configuration protocol please see the according section 4.6.2 in the firmware chapter.

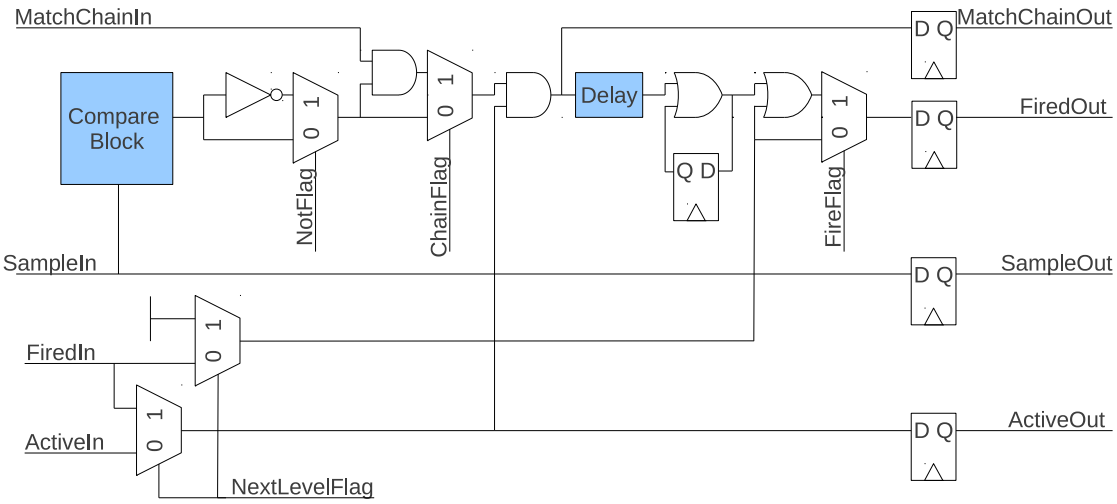

Figure 3.3: Basic blockdiagram of one trigger unit.

Compare Block The compare block can be configured to either parallel or serial mode. In parallel mode, it compares all channels with a configured match value, i.e. one bit per channel. Additionally, there is a bit mask which specifies the channels that should be checked. All channels with their corresponding bit set to 1 in the mask are checked against the match value. In serial mode, the function of the mask and match value are the same, however, the 32 input channels are replaced by the 32 last values sampled on one of the channels. The monitored channel can be selected using the configuration register of the unit. If a match occurs, the compare block sets its output to 1, otherwise

it is 0. The *Not-Flag* in the configuration register of the unit allows it to invert the match signal, i.e. a match occures every time the specified condition is not fulfilled.

Match Chaining Logic Following the Compare Block, there is a block which allows an wired-AND like combination of adjacent units. If the *Chain-Flag* of a unit is set, it can only fire the trigger if it matches at the same sample as the precedent unit. This allows it to trigger if two independent conditions are met simultaneously, for example to trigger on a falling edge on channel 0 while channel 1 is on 'low' state. Any number of units can be chained given that there indexes are next to each other, e.g. units 0, 1, 2 and 3. Note that it makes no sense to set the *Chain-Flag* of unit 0 as it has no preceding unit. Therefore the *Chain-Flag* of unit 0 is hardwired to 0.

Delay Counter and Fire Logic Using the delay counter, it is possible to delay the chained match signal before the *Fired* output goes high. The counter width is configurable, the default is 16 bits which allows a signal delay of up to 65535 sample times. When the configured delay has passed, the following OR-gate and register go to high state. If the *Fire-Flag* is set this rises the *Fired* output regardless of the *Fired* input. This second OR-gate allows multiple match chains to be ORed into one *Fired* signal.
Note: at least one has to have the *Fire-Flag* set. If not, the trigger will never fire, regardless of the input data. The logic analyzer and the client therefore wait for ever. This situation can be cleared by resetting the core.

Next Level Flag Trigger units can be arranged in increasing levels which influences the activation of the units. If a unit has the *Next-Level-Flag* set, this and the following units are on the next level compared to the preceeding one. When the trigger is armed, only units on the first level get activated. Once one of these units asserts the *Fired* signal, all units of the very next level become active. Please note that the sample firing the lower level is also processed as the first valid sample on the next level.
The *Fired* signal of the last level starts the sampling process of the analyzer core. This happens regardless of the number of units configured by the software as the units are transparent to the *Fired* signal in the default configuration.

Trigger Unit Registers There are four registers for each trigger unit:

- Mask
- Match Value
- Configuration
- Delay

The first two are used by the compare block, the last one specifies the match delay. The configuration register holds the flags which configure the unit, as well as the analyzer

Bit	Name	Remarks
31..13	unused	reserved for future use, set to 0 when writing the register
12	Fire-Flag	
11	Next-Level-Flag	
10	Chain-Flag	
9	Not-Flag	
8	Serial-Flag	
4..0	Serial Channel	Analyzer channel for serial mode, these bits are don't care when Serial-Flag is 0

Table 3.4: Trigger unit configuration register bits

channel used by the compare block in serial mode. The bit positions of the various flags inside the *Configuration Register* are given in table 3.4.

3.8 System Timer

The firmware needs some kind of clock as some of the protocols in the TCP/IP stack make use of timeouts. Therefore, a rather simple module generates a periodic interrupt at a fixed rate of 100Hz. This is used by the firmware to implement the needed system tick timer.

3.9 Clocking and Reset

The hardware has a 100MHz crystal oscillator on board. This clock is fed directly into the phase locked loop (PLL) of the memory controller block (MCB). This is required by the MCB core generated by the Xilinx Coregen. This PLL creates the necessary DRAM clocks, a so called startup clock used inside the MCB and a selectable user clock for the remaing system. This user clock is fed into a clock managment core generated by Xilinx Coregen which produces the differential 40MHz SoC clock and the 200MHz base clock for the sampling core. According to the Xilinx documentation, the startup clock has to be between 50MHz and approx 100MHz, no specification is made for the user clock. However, an implementation run with the Xilinx toolchain showed, that timing errors arise when the startup clock and the user clock have different frequencies. Therefore, the startup and the user clock are set to run at 62.5MHz which is one tenth of the PLL oscillator frequency. Note that the MCB needs a clock at twice the DDR2 interface frequency, so the PLL oscillator runs at 625MHz.
The reset signal propagates in the same way as the clock signal. A reset request from the push button resets the MCB and its PLL which negates the ready output. This resets the second clock management core which in turn disables its locked output. This finally resets all the logic inside the FPGA.

Chapter 4

Firmware

4.1 Introduction

This chapter covers the most important aspects of BitHounds firmware. It gives a general introduction of the firmware files and what they do. For details about the various functions implemented in these modules please see the source code, it is fully commented.
As with the FPGA design, some opensource code got reused to speed up the development and improve the reliability. The whole firmware is written in C and was compiled using avr-gcc 4.3.3 from the WINAVR 20100110 package. However other versions or even compilers from different vendors should work too.
For the handling of the IP, ARP, ICMP, TCP and UDP protocols, the uIP stack from http://www.sics.se/~adam/uip/index.php/Main_Page is used. AVR Studio 4.18 from Atmel (AVR Studio website) was used as development environment, however, a separate Makefile for Linux is available as well.

4.2 Ethernet - eth.c

This module uses the MAC hardware to control the reception and transmission of data packets over the Ethernet interface. The upper 12kBytes of the microcontroller's data memory are used as ethernet buffers. Each buffer is 1536 bytes long which is the maximum size for an ethernet frame. This gives a total of 8 frame buffers. The module manages this buffers, it keeps track of data lengths as well as transmission and reception status. Four of these buffers are used for reception and transmission, the other four are used exclusively for transmission. We've chosen this configuration because the logic analyzer sends much more data than it receives. This is due to the fact that we only receive configuration data but we have to send back all the sampled data, which is up to 128MBytes.
The module also performes the neccessary configuration of the MAC and the physical layer hardware (PHY) after reset. For the MAC, the available buffers, interrupt settings and the device's MAC address are set. Please note that the board has no MAC adress

device. A fixed MAC address is therefore compiled into the firmware. It is therefore not possible to have two BitHounds connected to the same broadcast domain at same time. If you need this you have to alter the MAC address of one of your BitHounds in *eth.h*, recompile the firmware and copy the firmware into the bitstream. See section 3.3.2 for a more information.
Further the default PHY configuration is altered to disable the Gigabit Ethernet support. This is neccessary as the PHY might autonegotiate to Gigabit Ethernet although the MAC does not support it. Please note that the datasheet of the Marvell 88E1111 PHY on the FPGA board is confidential. It can be requested from Marvell (http://www.marvell.com) once a nondisclosure agreement has been signed. Therefore, we unfortunately can't provide the datasheet along with the project.

4.3 UART - uart.c

The serial interface hardware (UART) is used for debugging purposes. Although it offers a bidirectional communication channel, until now, the firmware does not process any data received via the UART. However, it transmits various status and debugging information. There header file defines a FILE structure named *mystdout* which can be used to reinitialize *stdout*. This allows it to use C standard IO functions like printf. See function *main()* in file *main.c* for an example. However, please note that AVR architecture has to copy static strings into the data SRAM at application start. This is rather inefficient as constant strings can be stored in the much larger program ROM as well. This done using the *PSTR()* macro and library functions with a *_P* suffix like *printf_P*. Again examples can be found in *main.c* and other source files. To speed up the printing functions an internal ring buffer is used to store output data until the UART is ready to transmit it. However, excessively using debug outputs may significantly decrease system performance. So many debugging outputs are commented in the source code, they can be reenabled and compiled in when needed.
The default configuration for the UART is 115200 baud, 8 data bits, 1 stop bit and no parity.

4.4 uIP Stack - uip.c

Processing data packets of the TCP/IP protocol familly can be a rather complex and challening task. Luckily, several open source TCP/IP protocol stacks for microcontrollers exist on the internet. After a web research, the uIP stack written by Adam Dunkels from the Swedish Institute of Computer Science was chosen. Mainly because it is RFC compliant, has already been implemented on AVR devices, has an extensive online documentation and comes with various example applications.

4.5 DHCP Client - dhcpc.c

Most modern Ethernet networks use the Dynamic Host Configuration Protocol (DHCP) to assign IP addresses to network devices. This greatly simplfies the address management in the network. Therefore, BitHound integrates a DHCP client in its firmware. The uIP stack already comes with a dhcp client. However, this client makes use of a special socket library called Protosockets (http://www.sics.se/~adam/uip/uip-1.0-refman/) included in the uIP project. Due to code size limitation, this library is not used in the firmware. So, some modifications were made to the original client to integrate it.
When a UDP packet was received, the stack calls the high level application (see 4.6) which forwards the call to the DHCP client if no IP address was received so far. Once an IP address was received from the DHCP server, the client calls a function implemented in *main.c* to register the new address with the stack. The client tries several times to obtain an address. If all attempts fail, the client gives up after approximately 20 seconds. It then calls the main program which sets *10.0.0.2* as a default IP address. This makes sure that the logic analyzer is useable without a DHCP server in the network. However, the client has to use an IP address in the same subnet, i.e. the client's IP address has to start with *10.* too.
When an address was received or the default address was set BitHounds IP is shown on a LC display so the users knows the device is ready to use.

4.6 High Level Application - app.c

This module handles the communication between the client software and the sampling hardware. It receives calls from the TCP/IP stack if new data has arrived or an event like a timeout has occured. The received data is parsed, decoded and possibly forwarded to the sampling hardware.
The transfer of control information is done over a Transmission Control Protocol (TCP) stream between the client software and the firmware while the transfer of sampling data is based on the User Datagram Protocol (UDP). This is due to the fact that UDP requires far less computational overhead to create a datagram and therefore speeds up the transfer. The drawback is that UDP datagrams might get lost in the network or on the receiving PC. Therefore, the communication protocol offers a method to request the retransmission of a lost datagram.

4.6.1 Advertisement Broadcasts

When the logic analyzer firmware was configured and has received an IP address, it starts to send so called advertisement broadcasts to the network. This UDP datagrams are sent once every second and contain general information about the logic analyzer. The client software receives this broadcasts to find the analyzer in the network, so the user doesn't have to know the IP address of the analyzer. The UDP destination port number used by the firmware has to be known by the client. Therefore, this is a

fixed constant which can be changed by recompiling the firmware. We chose a default value of 1024. Please note that port number 1024 is marked as reserved by the Internet Assigned Numbers Authority (IANA, http://www.iana.org/assignments/port-numbers). We therefore assume that no other service in the network uses the same port which might cause communication problems. This might not be the best solution, however, we think it is the best possible as registering a sepcial port with IANA would be too complicated for this project. If problems are encountered due to selection of this port number, it could be easily changed by editing the define statement in *app.c* and recompiling the firmware.
The structure of the Advertisement Broadcast is given in figure 4.1. All numeric fields are in network byte oder, i.e. in big endian. The graphic also shows the values sent by our implementation. The datagram also contains information about the hardware which can be used by the client software. This allows a greater flexibility as the client software doesn't have to contain compiled-in information about hardware properties. A descriptions of these fields follows.

Magic Name This is a fixed constant of six bytes containing the string *SumpLA* in ASCII encoding. The client MUST check this field against the given constant. If it does not match, the datagram MUST be ignored. The idea of this is to reduce the risk that datagrams generated by other services on the same port in the network are mistaken as advertisement broadcasts.

ID Code This is the ID code of the logic analyzer. It is the same value as used in the communication protocol ID command (see 4.6.2). It specifies the highlevel communication protocol used by the device. The current version is *SLA2*. Note that *SLA0* and *SLA1* are incompatible protocal versions of the original Sump analyzer design based on a serial UART interface.

Port This is the port number where the logic analyzer listens for incoming TCP connections. This connection is used to transfer control commands from the client software to the firm- and hardware.

IP Address This is the IP version 4 address of the logic analyzer.

Sample Memory Length The total number of samples which can be stored in the sample memory of the logic analyzer.

Channels The number of hardware sampling channels. This also defines of how many bytes one sample consists.

Base Frequency This is the sampling core base frequency in MHz. This is the maximum sampling frequency if all channels are used. Note that the effective sampling frequency of the logic analyzer can be doubled if only half the channels are used.

4.6.2 Communication Protocol: Firmware – Software

The communication protocol of the original Sump logic analyzer design was adopted and extended to fit the new design. Each command consists of a one byte opcode followed by an optional 4 byte data word. The most significant bit of the opcode, when set, indicates that a data word follows.

Short Commands

These are commands without additional data.

Reset – 0x00 This command resets the sampling core hardware, i.e. trigger, sample counter, etc. Sending the command five times ensures the firmware state machine recognizes it as opcode in case the first four resets are interpreted as data portion of a long command.

Arm – 0x01 The firmware prepares the UDP socket for data transfer and starts the sampling core. Note that the trigger is not armed immediately after the command. First, the requested number of pre-trigger samples is taken. The hardware then automatically arms the trigger. Once the trigger tripped, the requested number of post-trigger samples is copied into the sample memory. When this is done, the hardware generates an interrupt and the firmware starts sending the sampled data to the client software.
Note that this command has to be the last one for a measurement. Any command other than Reset or Id is ignored between arm and the end of the data transfer to the client software.

Id – 0x02 The client uses this command to check the protocol version and the status of the logic analyzer. If the client was granted hardware access to the analyzer, the firmware responds with the same Id code given in the advertisement broadcast. For the protocol version documented here, this is the string *SLA2*, the left character is sent first over the TCP connection. If the analyzer is already in use by another client connection, the error code *ERR0* is returned instead.

Long Commands

These commands consist of five bytes: one opcode followed by 4 data bytes. The data field is a 32 bits long unsigned integer transferred in little endian byte order.

Set Divider – 0x80 Sets the core's sampling frequency to $F_{bc}/(x+1)$. F_{bc} is the sampling core base clock as reported in the advertisement broadcast and x is the data value of the command. The actual value of F_{bc} in our design is $200MHz$.

Please note that the hardware divider is actually 16 bits wide, so divider values grater than 0xffff are not possible. This fact is taken care of by the client software.

Set Flags – 0x82 Sets hardware flags which configure the sampling core. This command works the same way as in the original Sump logic analyzer, see http://www.sump.org/projects/analyzer/protocol/ for details.

Set Forward – 0x83 This command configures how many samples are copied into the sampling memory once the tigger has tripped.

Set Backward – 0x84 This command defines how many bytes are taken before the trigger is started. The delayed start of the trigger ensures that the requested amount of pre-trigger samples has been stored in the sampling memory. Together with Set Forward this command allows to set the number of samples to take and the trigger position within this length.

Set Dataport – 0x85 This configures the destination port number to which UDP datagrams with sampled data will be sent. The source port of this datagrams is the same as the destination port. Note that although the UDP port number is a 2 byte value this command has a 4 byte long data field. This is to simplify the control stream decoding. The two most significant bytes are ignored and should be set to zero.

Resend – 0x86 This command requires that arm was executed previously. It allows the client to request the retransmission of a data datagram which got lost. The data field of this commmand specifies the number of the first sample in the requested datagram. Only one datagram is sent in response to this command, if the client has missed more than one datagram it has to repeat the command for each datagram.

Select Trigger Unit – 0xC4 The trigger consists of several (default: 8) identical units. To keep the number of opcodes need at a fixed value even if the number of units is changed only can be configured at a time. By executing this command the unit specified in the data field is selected to be configured by commands 0xC0 .. 0x0C3. The selection is valid until the command is executed again or the *Reset* command is executed.
The counting of units starts with zero, so zero is the first valid index. If the data value is bigger than the index of the last unit (default: 7) it gets masked, i.e. it is mapped to a valid unit. The value has to be placed in the data fields least significant bits.

Set Trigger Mask – 0xC0 Writes the passed data value into the selected trigger units *Mask Register*. For more details about the trigger see section 3.7.2.

Set Trigger Value – 0xC1 Writes the passed data value into the selected trigger units *Value Register*. For more details about the trigger see section 3.7.2.

Set Trigger Configurations – 0xC2 Writes the passed data value into the selected trigger units *Configuration Register*. For more details about the trigger see section 3.7.2.

Set Trigger Delay – 0xC3 Writes the passed data value into the selected trigger units *Delay Register*. Please note that this register has a configurable length (default: 16 bits). The value has to be stored in the data fields least significant bits. For more details about the trigger see section 3.7.2.

4.6.3 Sampled Data Transfer

When the hardware has finished sampling, the firmware sends the sampled data to the client software. This is done using UDP datagrams. Each datagram contains up to a fixed number of samples, 360 in our case. This is determinded by the maximum transfer size of an Ethernet frame (1500 bytes data payload) minus the size of IP and UDP headers. The UDP destination port to which the datagram will be sent can be configured, see 4.6.2. The client's IP address as given by the TCP connection is used as destination address. Therefore, it is not possible to have a fFirwall or Router performing Network Address Translation (NAT) between the PC running the client software and the hardware. This however is not a very realistic scenario anyways as a local measuring device is hardly ever remote controlled over, e.g. the internet. The use of NAT inside a local network is very uncommon.
The structure of such a datagram is shown in figure 4.2. All values in the datagram are in little endian byte order. This is because the original Sump design is in little endian too. The *sample index* field in the datagram is the index of the packet's first sample in the record. This field always starts with zero in the first datagram and increments with the number of samples sent per datagram. Note that the samples are counted and not the bytes. This field allows the client to insert the packet into the correct positon in its sample buffer. The length field gives the number of samples transmitted in this datagram. This length allows to check the datagram for its completeness because given on the number of samples the total number of payload bytes can be calculated and compared to the number of bytes received by the client software.

4.7 Main Program - main.c

This module does general low level tasks like hardware initialization. It implements the main loop which exchanges frame buffers between the Ethernet Module and the TCP/IP stack. It further implements a system tick counter needed by the stack to measure timeouts.

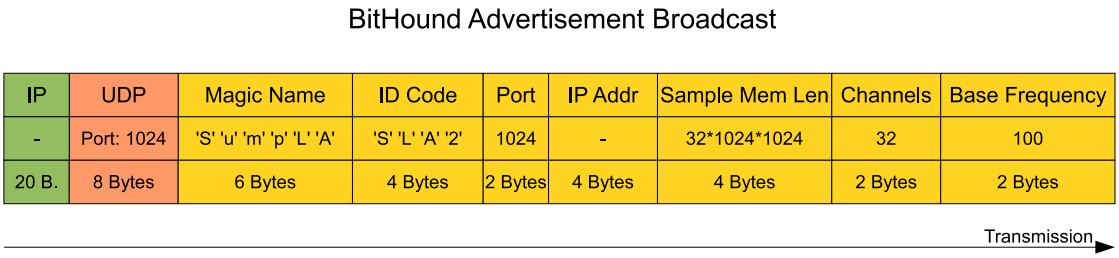

Figure 4.1: Format of Advertisement Broadcast Datagram

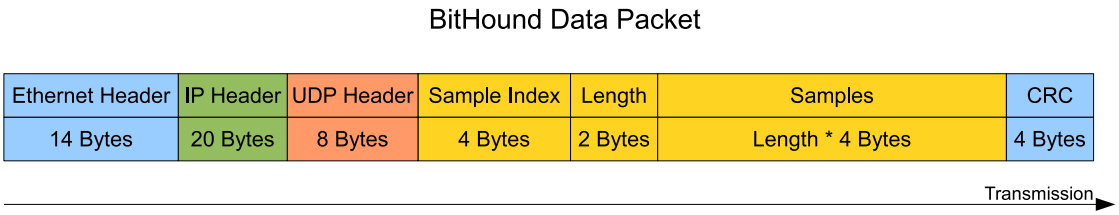

Figure 4.2: Format of Sampling Data Frames

Chapter 5

Client Software

The client software manages the visual presentation of the sampled data and implents the graphical user interface (GUI) to control the logic analyzer.

The original client software of the Sump logic analyzer was modified and improved in many ways to work with the new Ethernet interface of the logic analyzer. It also offers new features compared to the original sump-design to make use of the longer sample memory: any numbers of markers (vertical lines) can be placed and removed on the waveform, these cursors can be centered on the screen to easily jump between relevant positions. Another new feature allows to center the next edge of a selected signal. This allows it so easily jump over ,,idle" where signals do not change without zooming out and in again.

5.1 Trigger

To cope with the new trigger system of the ,,BitHound", the GUI was also renewed regarding the configuration of the trigger units. They can easily be configured with a matrix of checkboxes that allows a simple yet powerful set-up and combination of the units. This new trigger-configuration makes way to complex trigger-schemes, with this, it's easy to extract even sophisticated signal conditions out of the waveforms.

The trigger offers a ,,simple-" and an ,,advanced-" mode. In simple-mode, the user can set the trigger to fire on a rising or falling edge on any channel. This allows a quick set-up of a simple trigger condition.

In advanced-mode, all features of the trigger can be configured. Currently, the trigger offers 8 individual stages that can each have its own configuration. These stages can then be interconnected to form the complete trigger.

Each stage can either be set to parallel- or serial mode. In parallel mode, all channels are compared against a user-definable value. (the channels can also be masked). If this comparison is correct, the stage fires. In serial mode, the data of one user-selectable

channel will be shifted serially. Then, the software compares this stream of data with a user-definable mask. If the stream matches the mask, the stage triggers. In the GUI, the data is shifted from the LSB to the MSB, from right to the left.

Once the desired stages are configured, they can be connected. This is done with a PAL-like matrix. (PAL = simple (and outdated) logic device). The individual stages can be AND- or OR-connected. This means that the sampling starts, when this logical stage-interconnection is ,,true". E.g. if stage 0 AND stage 1 have fired, then the sampling starts. The trigger-stages can also be armed by a certain trigger-condition. E.g. when stage 3 fires, it arms stage 4. Unless a stage is armed, it won't fire. If no special leveling is specified, all stages are armed. The different trigger-levels are highlighted in the matrix with an individually colored background.

5.2 Diagram

When the BitHound has transfered its captured data to the PC client, it will be displayed in the main screen.

Various tools allow an analysis of the waveforms. By clicking on the screen, a cursor appears. In the bottom-left screen, it displays the time-position of this cursor in the diagram. By clicking, holding and moving the cursor, the relative time-position as well as the corresponding frequency is also displayed in the bottom-left of the screen. This makes it easy to measure times and frequencies.

There are several tools like ,,add marker", ,,remove marker", ,,go to next/previous marker" and ,,go to next/previous edge" that are self-explanatory. By clicking on a channel, it gets highlighted and the ,,go to next/previous edge" feature will focus the display to the next/previous edge of this selected channel. Markers can be removed by setting the cursor on the marker and clicking the ,,remove marker" button.

5.3 Code

In the following, the different and relevant java-files are presented.

5.3.1 DeviceController.java

This class creates the GUI dialog which is used to control the analyzer, especially the trigger. Larger changes have been made in this file to implement the new trigger scheme and to improve the usability.

The sampling-rates were adapted to the faster hardware, sampling rates of up to 400MHz are now possible. Together with the recording size (number of samples to take), the GUI now also displays the total sampling-time.

The Trigger-configuration has been made much more intuitive and easier to configure, see above for details.

This file also controls the entire trigger-stage-handling. It generates the interconnection-matrix, enables or disables the checkboxes according to the settings and also highlights the background if different trigger-levels are selected. It also generates a text-string that represents the trigger-stage-interconnection.

5.3.2 Device.java

This class communicates with the firmware using TCP and UDP sockets. For the reception of the UDP datagrams containing sampled data, a second class called *UdpReceiver* is implemented in the same source file. This class uses its own thread to receive the datagrams from the socket passed to its constructor. The received datagrams are inserted into a linked list without any processing. The thread executing the *run* method of the *Device* class takes the datagrams and processes them. It is necessary to use one thread exclusively for datagram reception as datagrams are not stored by the operation system. This means that any datagram arriving while the application stores the last one is lost. Therefore, the time between two calls to the sockets receive method must be as short as possible.

There were also changes made regarding the trigger configuration, as the trigger scheme has greatly changed since the original ,,sump"-design.

5.3.3 Diagram.java

This class renders the waveform on the screen to display the samples. A few methods were added to this class to implement the new features like markers or cursors. Also, various features like zooming were improved a little.

Chapter 6

Conclusion

During the last few months, we have succeeded to design, build and improve the logic analyzer according to our ideas and requirements.

During the improvement, we were able to rise the maximum sampling clock frequency from 200MHz to 400MHz. We were able to achieve this by greatly altering the trigger of the ,,BitHound" - it is now much more flexible and faster than before. The GUI was also adapted to fit the new trigger; its configuration is now much more intuitive.

We have also built a body for the device that holds the ATLYS-Board, our interface-board, the channel-LEDs as well as the LCD display.

In the design process, we have recognized that some aspects of the system could be improved in the following way:

PC Client The functionality GUI could be extended: pattern search, an UART or USB protocol decoder could be added. The buffer handling of the client could also be implemented a bit better. This would result in a better, and probably faster, data transmission.

Gigabit Ethernet At the moment, the system is configured for 100MB Ethernet transfer speed. This could be improved to Gigabit Ethernet to even further speed the transfer of the sampled data up.

The entire project was a very valuable experience for us as we were confronted with many different problems and situations that required further analysis and investigation to find a suitable solution. The problems were also very widespread, we had to deal with FPGA-specific stuff, network protocols and traffic analysis, as well as with analog and digital control loop design.

In the end, we think we have come up with a useful tool that can be used to solve many different problems in engineering situations.

Chapter 7

Appendix

7.1 Tools required

- Atlys Spartan-6 FPGA-Board (or similar Spartan-6-board).
- Xilinx ISE (Windows/Linux) or similar software for a Spartan-6.
- KiCad (Windows/Linux) or similar schematic/layout editor.
- PC running Java (PC-client-software, Java-based, Windows/Linux)
- The PC client was developed with Eclipse
- Interface Board (build one yourself)

7.2 Schematics

The schematics of the Interface Board are presented in the following.

Figure 7.1 shows the top-level hierarchy-sheet of the schematic showing all components.
Figure 7.2 shows the overvoltage protection circuitry.
Figure 7.3 shows one input connector (8 channels) as well as the resistors and diodes needed for the overvoltage-protection-circuitry. On the Interface board, 4 such modules are implemented.
Figure 7.4 shows a level shifter. 2 of these are implemented on the Interface Board.
Figure 7.5 shows the connector on the Interface Board. It connects this board with the wirewrap expansion-board.
Figure 7.6 shows the Mikrocontroller and the boost-converter.

7.3 Interface Board - Layout

Figure 7.7 shows the layout of the Interface Board.

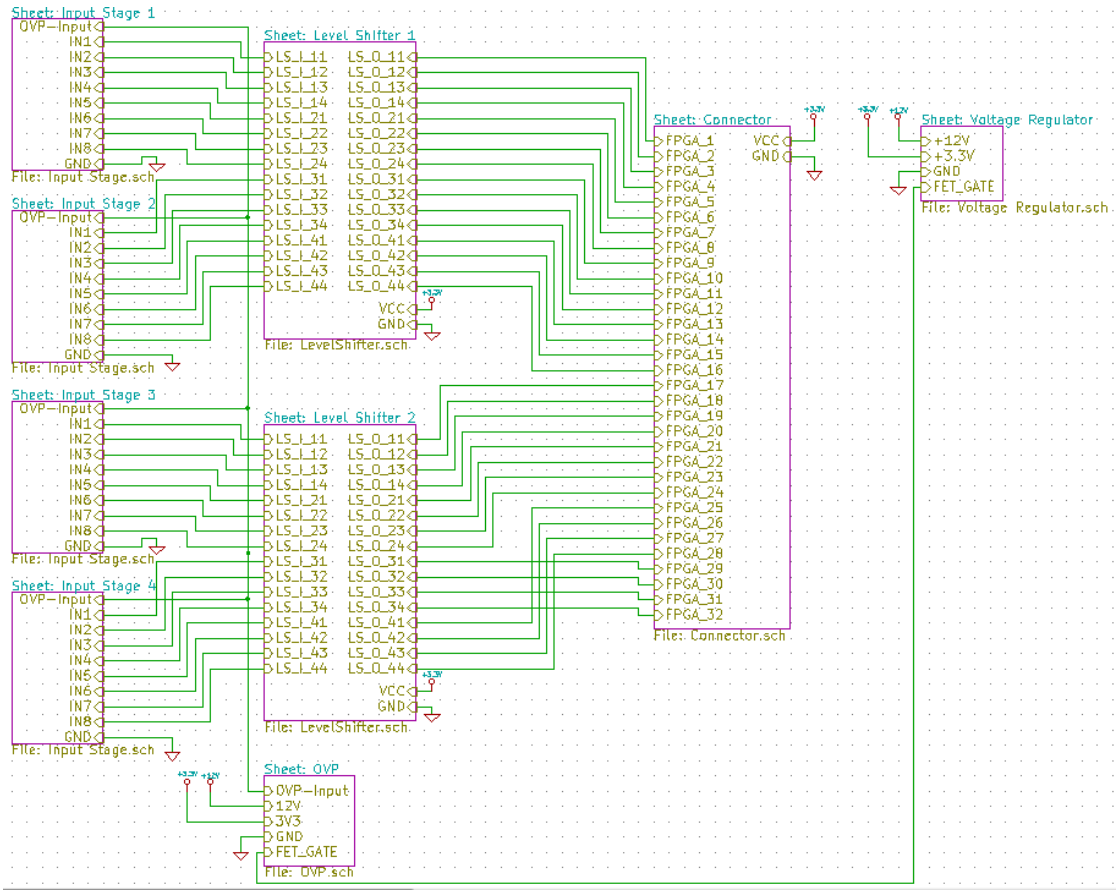

Figure 7.1: Top-level schematic sheet

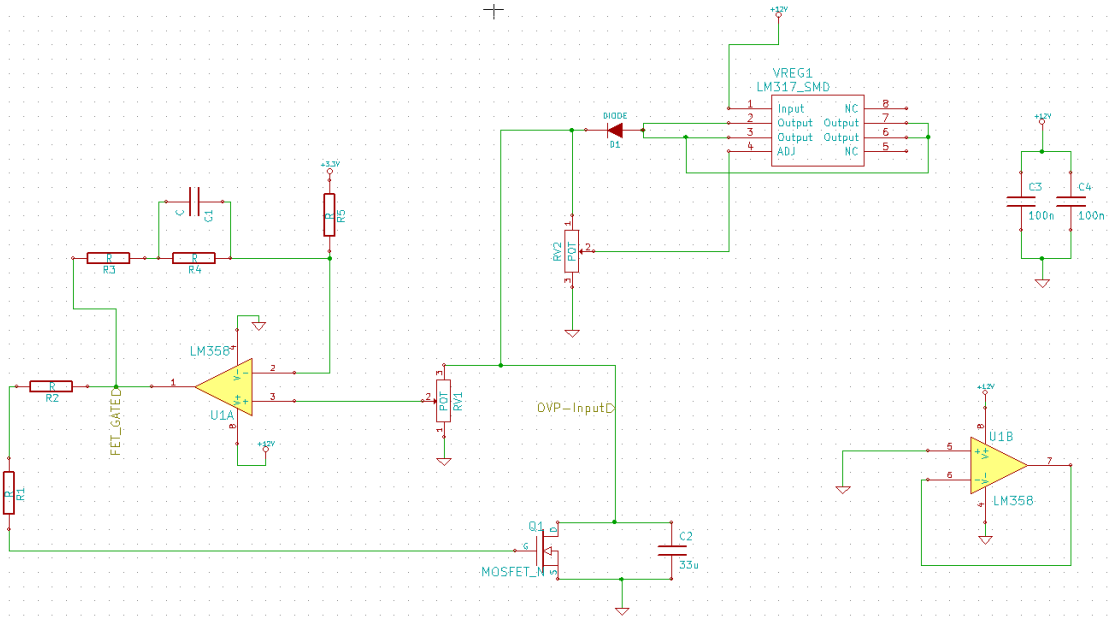

Figure 7.2: Overvoltage Protection

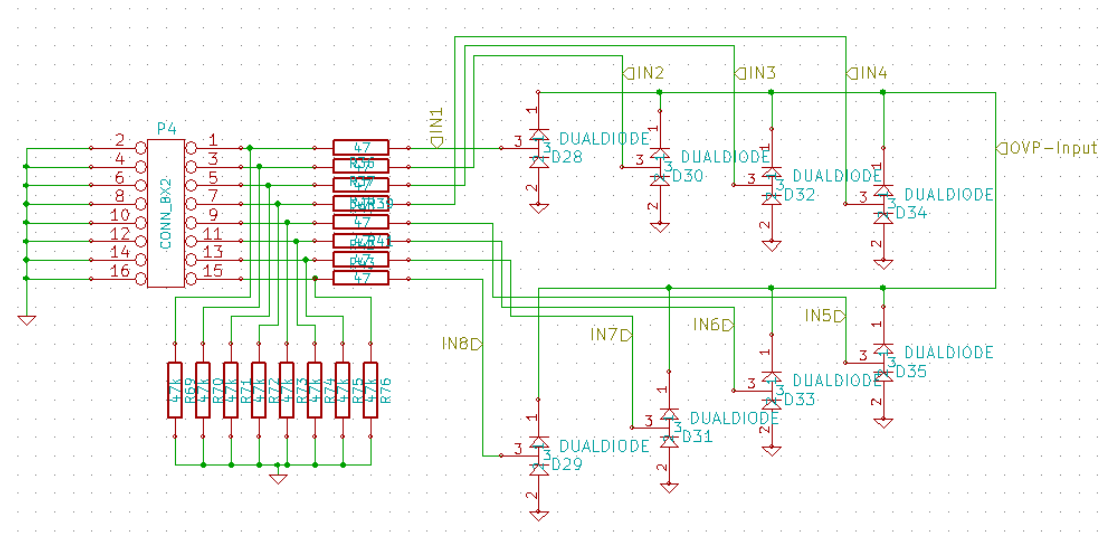

Figure 7.3: Input stage

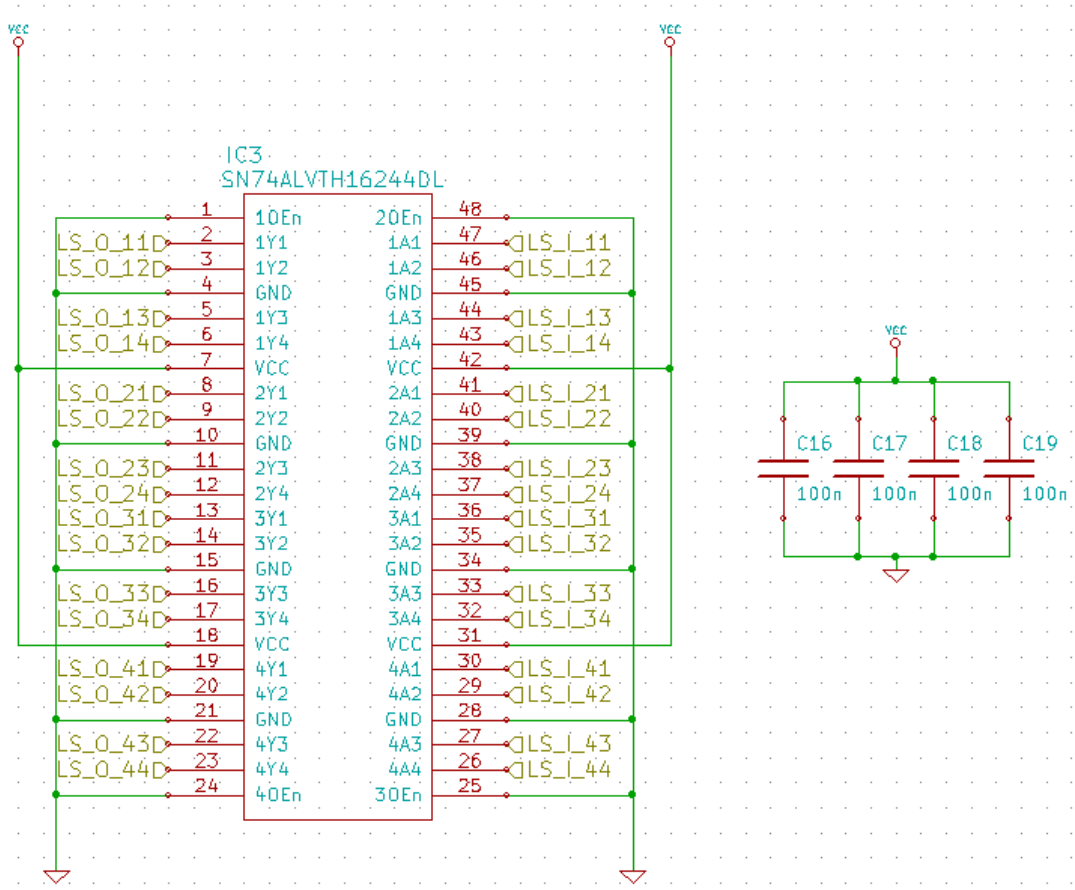

Figure 7.4: Levelshifter

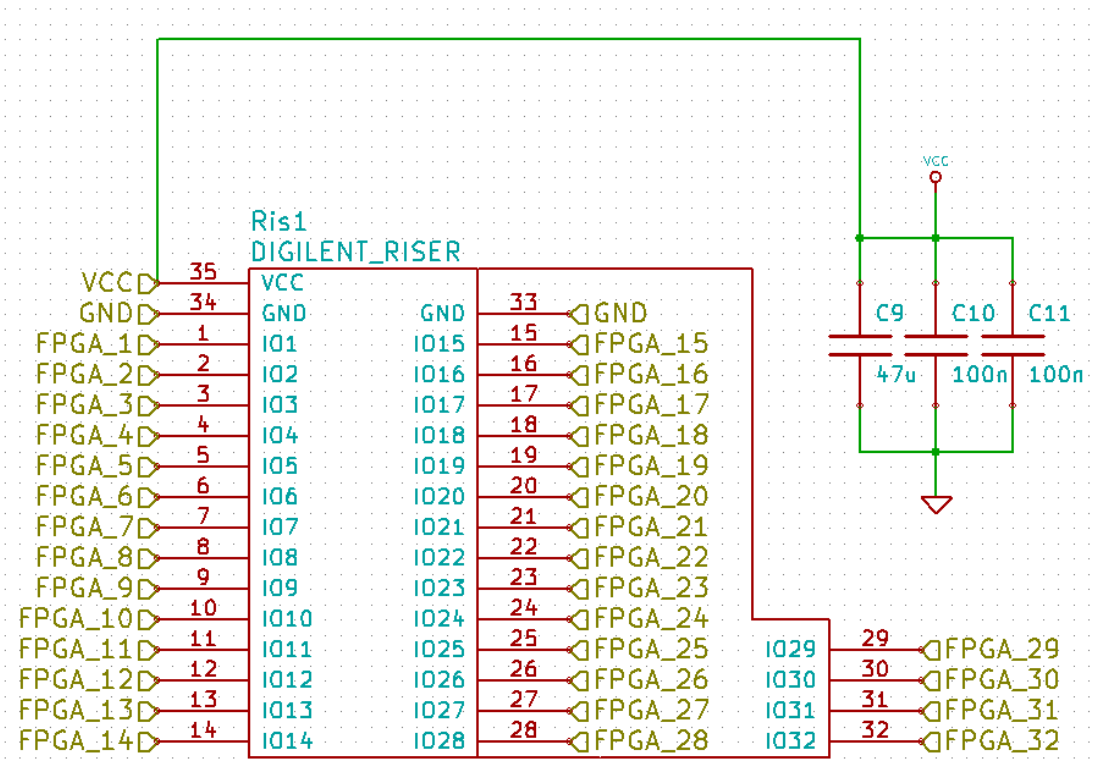

Figure 7.5: Connector

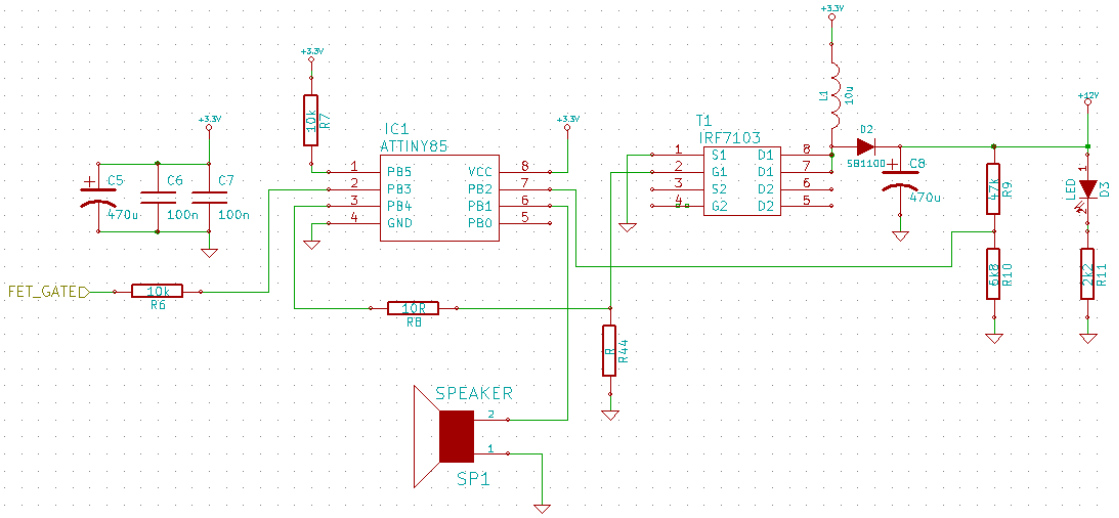

Figure 7.6: Microcontroller and boost-converter

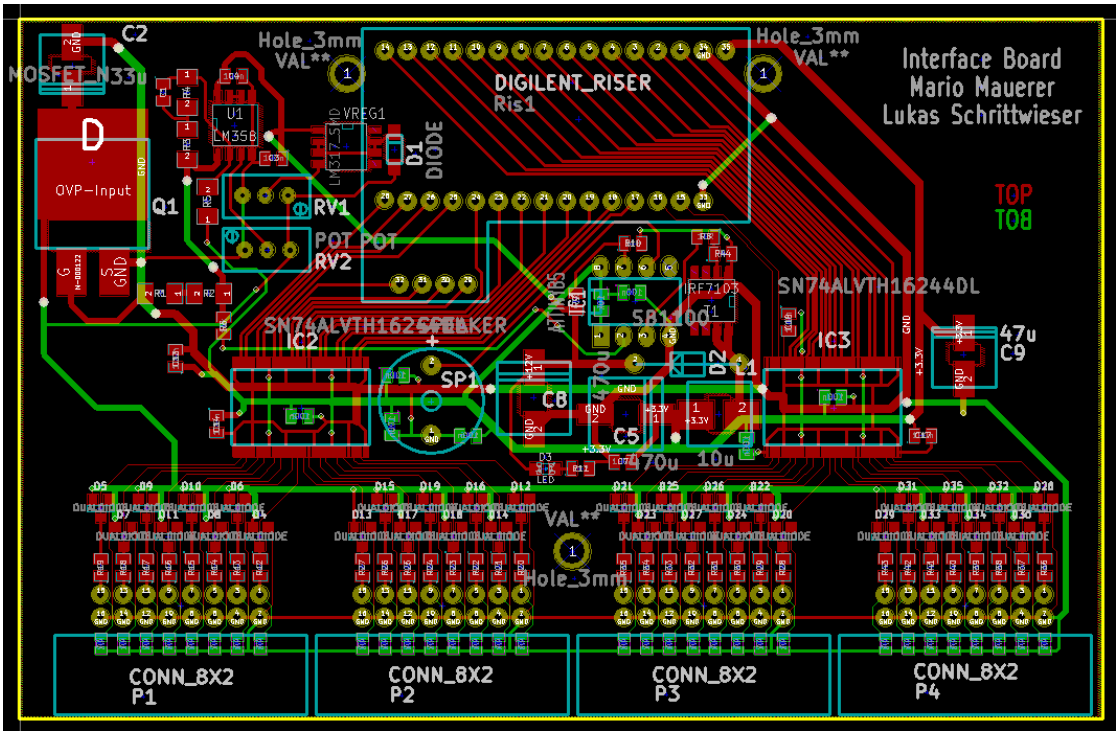

Figure 7.7: Layout

7.4 Block Diagram of the original sump

Figure 7.8 shows the block diagram of the original sump Logic Analyzer.

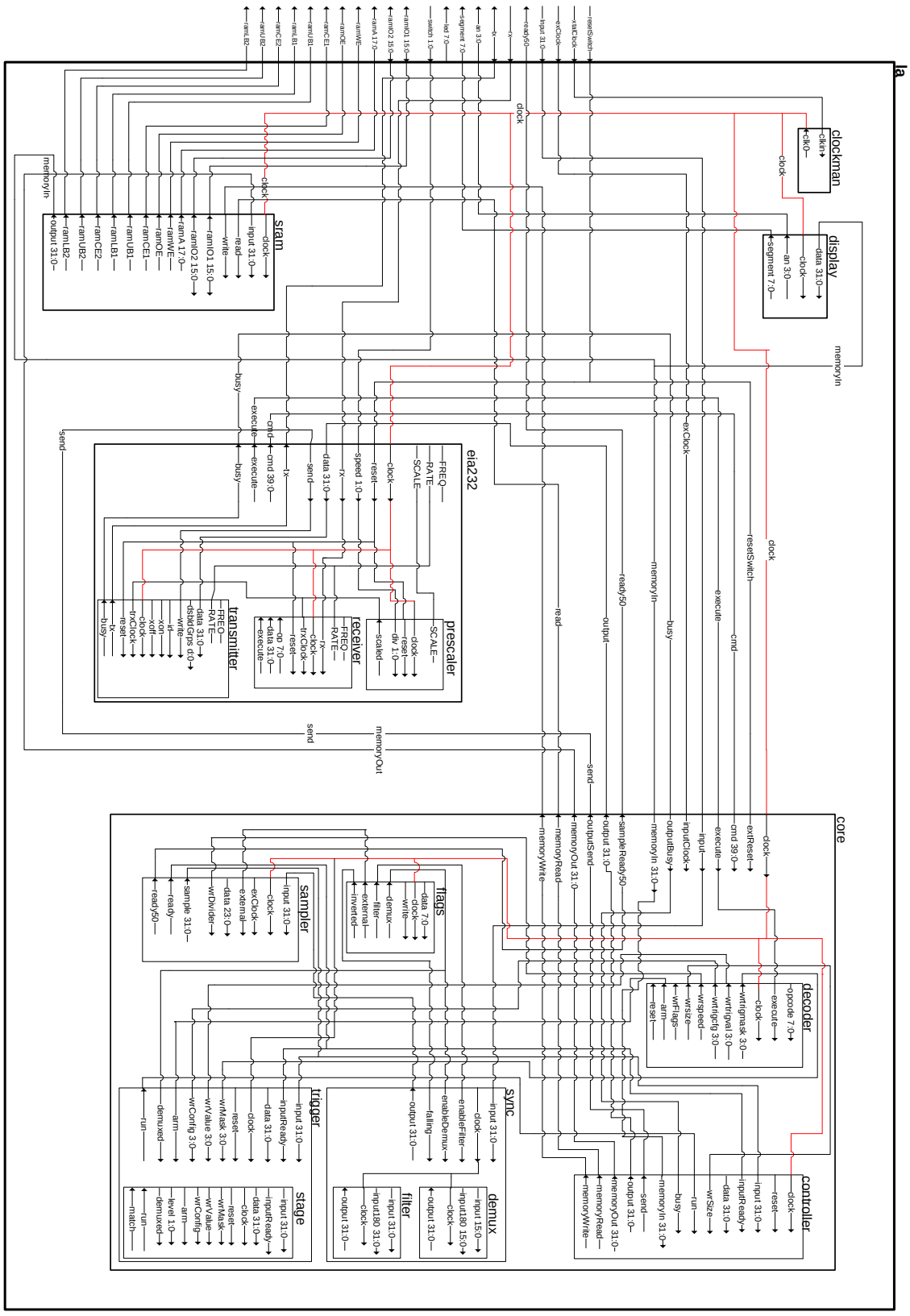

Figure 7.8: Block Diagram of the original sump Logic Analyzer