Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semesterarbeit

Controlling a Field Programmable Gate Array to achieve single shot read-out of the quantum state of a superconducting qubit

Verfasser:	Philipp Arnold
Matrikel-Nummer:	05-912-076
Studienrichtung:	Physik
Betreuer:	Univ.-Prof. Dr. Andreas Wallraff,
	Laboratorium für Festkörperphysik, ETH Zürich

Zürich, am 28.Mai 2008

Contents

1 Abstract **5**

2 Introduction **6**
2.1 The discovery of quantum mechanics 6
2.2 Exploiting quantum mechanics: quantum information 6
2.2.1 Classical computing 6
2.2.2 Quantum computing 7
2.3 Cavity quantum electrodynamics 7
2.4 The physical implementation of the qubit in this experiment 7
2.5 Exciting the qubit 8
2.6 Reading out the state qubit 9
2.7 Why to use an FPGA for the read-out? 10

3 The Xilinx Virtex-4 SX programming board **11**
3.1 General information about FPGAs 12
3.2 Components 12
3.2.1 The Analog Digital Converter (ADC) 13
3.2.2 The Digital Analog Converter (DAC) 14
3.2.3 ZBT-Random Access Memory (RAM) 15
3.2.4 User-LEDs 16
3.2.5 User-pins 16
3.2.6 FPGAs 16
3.3 Connection to ZBT-RAM via PCI 16

4 Programming the FPGA **18**
4.1 Standard modules 18
4.1.1 System Generator 18
4.1.2 Constant 19
4.1.3 Counter 19
4.1.4 Delay 20
4.1.5 Register 20
4.1.6 Reinterpret 20
4.1.7 Slice 21
4.1.8 Single port RAM 21
4.1.9 Gateway In/Out 21
4.2 User-defined modules 21

4.2.1 ZBT-RAM module . . . 21
4.2.2 ADC module . . . 22
4.2.3 DAC module . . . 22
4.2.4 LED module . . . 23
4.2.5 User-pins module . . . 23
4.3 Compiling a model in Simulink . . . 23
4.4 Running a simulation in Simulink . . . 24
4.5 Uploading the generated bitilfe onto the FPGA . . . 24

5 A first hardware simulation **27**
5.1 Experiment setup . . . 27
5.2 The user application . . . 28
5.3 The recorded data . . . 29

6 Summary **33**

7 References **34**

8 Appendix **35**
8.1 Pin connections . . . 35
8.1.1 ADC . . . 35
8.1.2 DAC . . . 36
8.1.3 User pins . . . 37
8.1.4 LEDs . . . 37

1 Abstract

Quantum information processing is a very promising application of quantum mechanics with the potential for incredible advances in computational power. Thus within the last two decades a lot of effort has been put into implementing and controlling qubits and a lot of progress has been made. One of the most promising approaches due to its scalability is using a two level quantum system which acts as artificial atom in a superconducting circuit. One major advantage is that fabrication techniques can be borrowed from conventional electronics. For this kind of superconducting qubit coherence times of several microseconds have been reached allowing in principle several hundred operations to be executed on the qubit before it decays.
In order to do information processing and quantum computing it is essential to be able to have a reliable method for reading-out the state of the qubit as some quantum algorithms need feedback based on a qubit state within coherence time.
Prof. Wallraff's group is experimenting with superconducting qubits and I was working on a project that concerned the reading-out of the state of the qubit. So far the group has used a predesigned averaging analogue to digital converter card which recorded the data of e.g. 10000 read-outs (one read-out takes $\approx 40\mu s$) and averaged the obtained data. So the state of the qubit could only be obtained after several thousand measurements. But in order to do real quantum computing it is crucial to decide the qubit's state within one measurement: a so called "single shot read-out". This should be achieved by using a Field Programmable Gate Array (FPGA). A FPGA allows fast data recording and is very flexible because it can be customized to match the needs for a specific task. The idea is to implement a filter function which preprocesses the data from a measurement in such a way that the decision which state the qubit was in can be made directly after one measurement.
During my project I have experimented with different ways of programming the FPGA and controlling tools such as an analog-digital converter, digital-analog converter, LEDs, User-pins and a RAM module. As a final experiment I implemented a program on the FPGA which is able to process sample data (rectangular pulses with white noise) which corresponds to the data provided in the qubit experiment.
As conclusion I can state that the fundamental problems concerning the control of the FPGA are solved and it should be possible to generate a program which is able to perform a single-shot read-out of the state of the qubit.

2 Introduction

2.1 The discovery of quantum mechanics

At the beginning of the 20th century the physical world was shuttered by a publication written by Max Planck who suggested that the ultraviolet catastrophe of black body radiation could be solved by assuming that the spectrum of the radiation consists of quantized frequencies. This assumption was in contradiction to common physical theories at that time. But as it turned out the idea of quantizing solved many open questions which couldn't be explained classically (electron orbit in an atom, photoelectric effect, just to mention a few).
This new theory is called "quantum theory". It holds in microscopic systems and the effects, with some exceptions such as superconductivity, cannot be seen on a macroscopic scale. The main idea behind quantum theory is that particles can be described by wave functions which give the information what the probability of finding an object in a specific state is. It is correct to talk about "states" since some physical properties (such as energy, spin...) can just be observed in discrete (thus quantized) values. But it is in general not possible to know the outcome of a measurement before it is done since in general a quantum object is in a superposition of different states. Only the probabilities of finding the object in one or the other state are known. An other constriction is that it is even theoretically not possible to measure physical properties of a system (such as position and impulse) at the same time with arbitrary precision.
Even though this might sound strange at first and contradict any physical intuition quantum theory is one of the best tested and most successful theories in physics.

2.2 Exploiting quantum mechanics: quantum information

As the first hurdles were overcome and quantum theory was widely accepted there was of course the question what could be achieved using this new knowledge. Davit Deutsch in 1985 was the first one to propose that quantum objects could be used to build a computer which is in many ways superior to classical computers.

2.2.1 Classical computing

Today's computers work basically on the same principles as the first computers build in the early 1940's. They consist of a central processing unit (CPU) a control unit and memory.

The unit for information in a classical computer is a single bit which can be either '0' or '1'. Today such a bit is commonly physically implemented in a solid state transistor. Within the last decades the size of such a bit has decreased dramatically and today it usually doesn't cover more than $10^4 nm^2$.
But the decrease in size cannot go on unlimited without considering quantum effects since as mentioned above classical theory doesn't hold in atomic structures.

2.2.2 Quantum computing

The fundamental difference between a classical computer and a quantum computer is the unit of information: in quantum computers quantum bits (qubits) are used. Not the state of a macroscopic transistor but the state of a quantum object (e.g. an atom) contains the information.
This means that as mentioned in 2.1 a qubit isn't necessarily either '1' or '0' as the classical analogue but in a superposition of the two. This gives rise to a whole new way of computing and algorithms.
Quantum computers can perform tasks highly parallel which makes extremely powerful algorithms, e.g. factorization of large numbers (Peter Shor's algorithm), possible. Problems which couldn't have been solved on a classical computer even within the lifetime of the universe could the be solved within a reasonable time.

2.3 Cavity quantum electrodynamics

Very promising for the realization of a quantum information processor is based on cavity quantum electrodynamics. A quantum system such as for example an ion is trapped in a cavity and state transitions can be achieved by letting a photon couple to the quantum system. It is therefore crucial to reach a high interaction between the photon and the quantum system.
The best coupling can be reached using circuit quantum electrodynamics (QED). In circuit QED a coplanar waveguide transmission line resonator acts as cavity. This allows to use well known fabrication techniques and thus the cavity and the dipole coupling can be very well engineered.
One major problem in every solid state qubit is the decoherence of the quantum system which comes from coupling with undesired degrees of freedom and leads to a loss of quantum information.

2.4 The physical implementation of the qubit in this experiment

The qubit in this experiment uses the technique of circuit quantum electrodynamics and consists of mainly two parts: an harmonic oscillator which is used as resonator or cavity for electromagnetic waves (photons) and a cooper pair box (CPB, see Fig. 2.2) which can be

considered to be an "artificial atom" as it can be excited to a higher energy level by absorbing a photon.
The cavity with the CPB is inside a cryostat which is cooled down to about 0.02 K as the desired effects can only be observed under superconductivity and minimal thermal noise.
The cavity is basically a simple harmonic oscillator consisting of an inductor L and a capacitor C as it can be seen in the schematic figure 2.1:

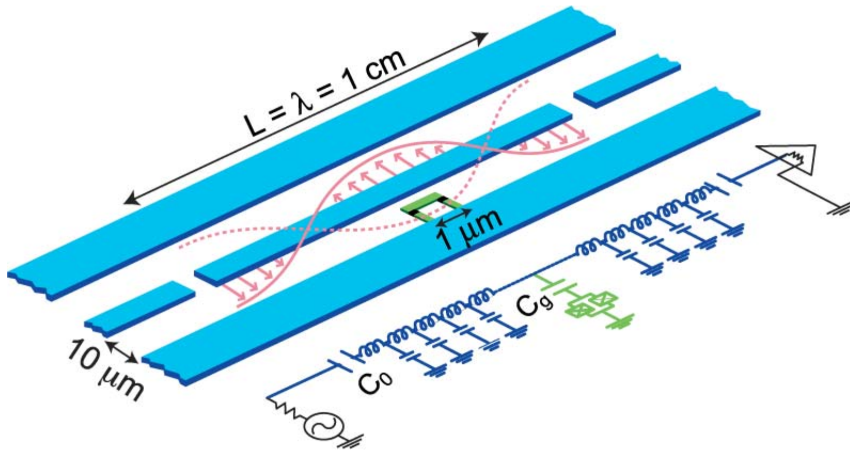


Fig. 2.1: Schematic view of the cavity and CPB (green)

The Hamiltonian for the cavity without the CPB is:

$$H_{LC} = \frac{q^2}{2C} + \frac{\Theta^2}{2L} \tag{2.1}$$

where q is the charge stored in the capacitor and $\Theta = L \cdot I$ the flux stored in the inductor. The resonance frequency ω_r is dependent on the setup of the cavity and in this case $\frac{\omega_r}{2\pi} \approx 6GHz$
The CPB consists of a reservoir for cooper pair electrons and a small superconducting island which is connected to the reservoir by two josephon junctions. This allows electron pairs to tunnel from the reservoir to the island and vice versa.

The CPB is mounted in the middle of the cavity, as can be seen in the figure 2.1, because there the electromagnetic field of the trapped photon is the highest. This means that the coupling between the photon and CPB is the strongest.

2.5 Exciting the qubit

The ground state ($|0\rangle$) of the qubit is defined as '0' additional electron pairs on the island with respect to neutrality. The energy of the ground state may be defined as E_0. If one cooper pair tunnels through the josephon junction onto the island the quantum mechanical system is in a higher energy level E_1 (first excited state: $|1\rangle$). Thus a photon with the energy

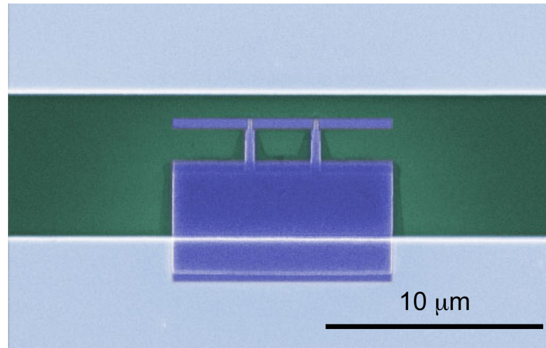


Fig. 2.2: Cooper pair box as used in the experiment

$E_{01} = E_1 - E_0$ can cause such a transition. The frequency such a photon is:

$$\omega_a = \frac{E_{01}}{\hbar} \tag{2.2}$$

2.6 Reading out the state qubit

During the read-out process another photon of a specific frequency ($\approx \omega_0$) is send into the cavity. Since the additional cooper electron pair (if in excited state) changes the electric configuration of the cavity the resonance frequency is slightly shifted (ω_r^*).
Now there are two ways to take advantage of this fact to read-out the qubit's state:

- Transmission probability
- Phase shift

The transmission probability for the cavity is dependent on the impedance of the cavity for a photon of frequency ω which is given by the formula:

$$Z_{LRC}(\omega) = \frac{R}{1 + 2iQ\frac{\omega - \omega_r^*}{\omega_r^*}} \tag{2.3}$$

where $Q = \omega_0 RC$ is the quality factor which tells how the amplitude of the oscillation decays with time (= loss of energy). From equation 2.3 follows that the amplitude of outgoing photons from the cavity is bigger for photons which are very close to the resonance frequency than for those which have a different frequency.

The second way, which is used in the experiment, is to measure the phase shift of the outgoing photon p_1 relatively to a reference photon p_2 which was in phase with p_1 before it was sent into the cavity. At the resonance frequency ω_r^* there is a phase shift of π.

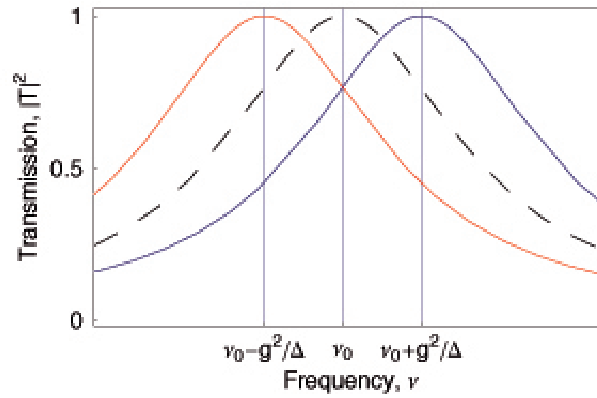


Fig. 2.3: Transmission probability of photons depending on their frequency

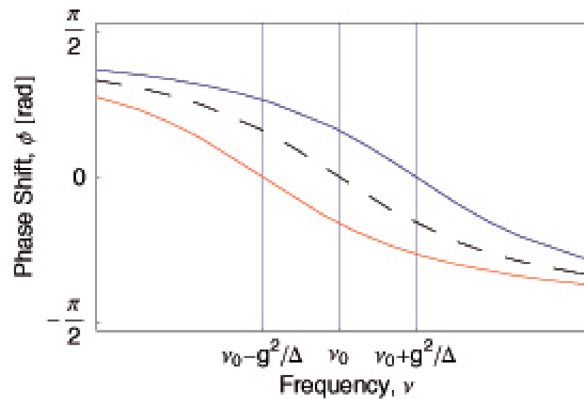


Fig. 2.4: Phase shift of photons depending on their frequency

2.7 Why to use an FPGA for the read-out?

Like in atoms the excited state of the CPB is not stable and will decay after a characteristic decay time $T_1 \approx 1\mu s$.
This means that within T_1 there has to be enough data collected in order to decide whether the qubit was excited or not. Thus the signal processing has to be very fast. Implementing a read-out method in software would be to slow and there would be a possible loss of data. A FPGA is made for exactly such applications. It is possible to record a sample every 10ns which makes 100 samples within T_1. Since the program on the FPGA is implemented in hardware there is no loss of information unlike in a microprocessor. This is due to the "pipe-in-pipe-out" system as the data proceeds one step at each clock cycle. There is no need for buffers and thus no overwriting of data in buffers which might happen in microprocessors.
As there are also digital to analogue converters on the board it is possible to output the the state of the qubit which was read out very quickly and apply some feedback on the quantum system before it decoheres.

3 The Xilinx Virtex-4 SX programming board

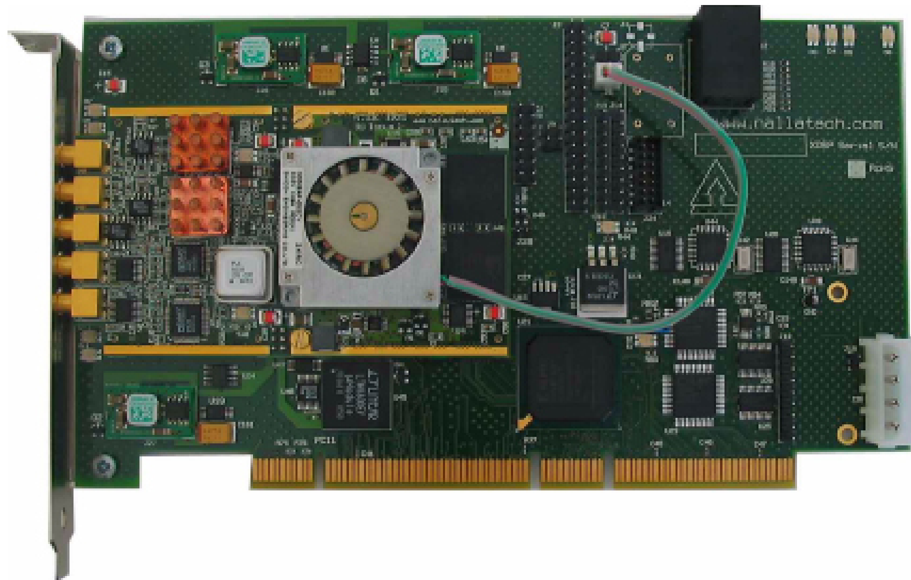


Fig. 3.1: The Xilinx Virtex-4 experimental board

Xilinx is together with Altera the market leader for FPGA solutions. The local vendor for Xilinx products is Silicia:

Silica (An Avnet Company)
Bernstrasse 392
CH-8953 Dietikon
Phone: + 41 43 322 49 49
Fax: + 41 43 322 49 50
Email: silica.schweiz@avnet.com
Website: www.silica.com

The toll free technical support phone line for Europe (EN, DE and FR) is:
00 800-5152-5152 (M-F 8:00 -17:00 GMT)

3.1 General information about FPGAs

A field programmable gate array (FPGA) is a semiconductor device consisting of logic blocks which can be programmed using hardware description language (HDL). The main difference between a microprocessor and a FPGA is the fact that the design of the FPGA-program is implemented in hardware which allows to execute specific tasks (such as signal processing) much faster and more efficiently than if it was implemented in software like in a microprocessor. The gain of efficiency is achieved because time critical operations can be executed parallel. If, for example, there is a need of many multiplications one can add the desired number of multiplicators to the program and within one clock cycle the result is ready.
Typical state of the art FPGAs consist of several million logic gates, external RAM and run at a clock speed of ca. 400 MHz.

3.2 Components

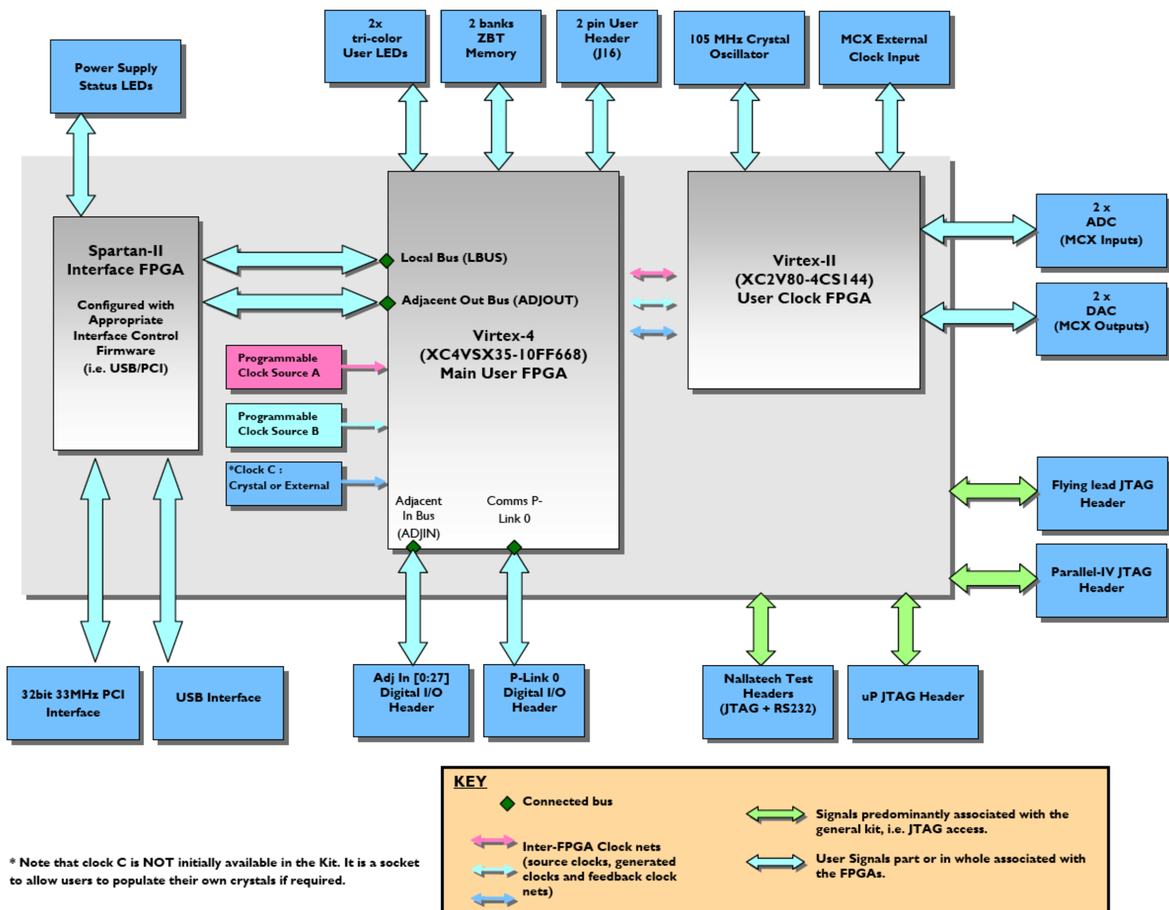


Fig. 3.2: A schematic view of the experimental board

The Xilinx Virtex-4 SX programming board is optimized for signal processing. The main hardware components mounted on the board are:

- 2 analog-digital converters (ADC) with 14 bit resolution
- 2 digital-analog converters (DAC) with 14 bit resolution
- 1 analog input to synchronize the internal Clock FPGA with an external clock
- 2 memory banks with each 8Mb memory
- 1 User FPGA XC4VSX35
- 1 Clock FPGA XC2V80
- 1 RS232 interface and numerous other user-pins

These components will be discussed in this section.

3.2.1 The Analog Digital Converter (ADC)

The ADC mounted on the board is an AD6645 with

- 14-bit resolution (2's complement format)
- maximum sampling rate of 105 MSPS
- recommended maximum signal magnitude: 2V p-p or ± 1V
- single-ended 50Ω impedance input

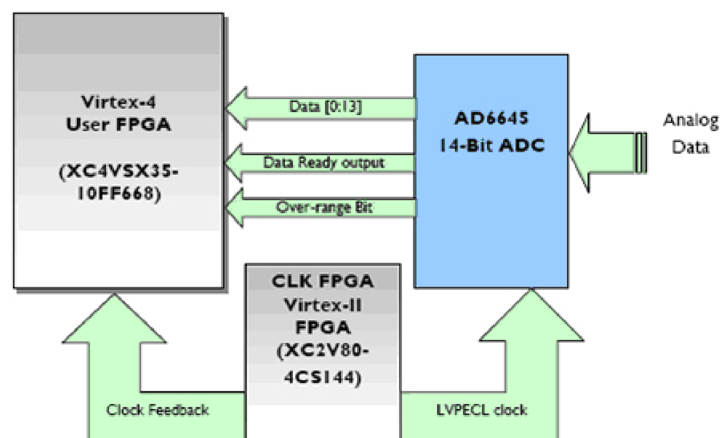


Fig. 3.3: A schematic overview of the ADC to FPGA interface

The clocking signal for the ADC comes from the Clock FPGA. This means that the clocking is dependent on the assigned bitfile for the Clock FPGA. There are 3 different options:

- on board 105 MHz crystal
- external clock input via middle MCX connector

- programmable oscillators available on the programming board

It is important that the ADC only supports clock input of up to 105MHz.
The effective number of bits (ENOB) is 12 (using the on board 105MHz oscillator) due to noise and crosstalk between the two input channels.
The signal to noise ratio (SNR) is, at best, 74.5dB.
The analog input signal is DC-coupled through a differential operation-amplifier (AD8138).
A schematic view for the data flow of the ADC can be seen in Fig. 3.3.
The hardware pin connections can be found in the appendix (8.1.1).

3.2.2 The Digital Analog Converter (DAC)

The DAC mounted on the board is an AD9772A with

- 14-bit resolution (offset-binary format)
- a maximum sampling rate of 160MSPS
- single ended (DC coupled) 50Ω ouput
- output value between -1V and 1V

The clocking for the DAC mainly works like the clocking for the ADC. There is one more option though: There is a PLL clock multiplier pin provided which doubles the clock frequency for the DAC. This can be useful because otherwise the DAC must be fed at half the reference clock frequency due to a interpolation filter that adds extra samples every second clock cycle. The output value is between -1V and +1V. The DAC expects input in offset binary format. Since in the FPGA the 2's complement format is used there needs to be a conversion. This conversion is straightforward: the MSB needs to be inverted.

The DAC can be operated in different modes to guarantee optimal SNR at different sampling rates. The optimal setup can be found in Fig. 3.4.

Input Data Rate (MSPS)	MOD1	DIV1	DIV0	Zero-stuffing	Divide-by-N-ratio
48-160	0	0	0	No	1
24-100	0	0	1	No	2
12-50	0	1	0	No	4
6-25	0	1	1	No	8
24-100	1	0	0	Yes	1
12-50	1	0	1	Yes	2
6-25	1	1	0	Yes	4

Fig. 3.4: Different modes of operation for the DAC

A schematic view for the data flow of the DAC can be seen in Fig. 3.5.

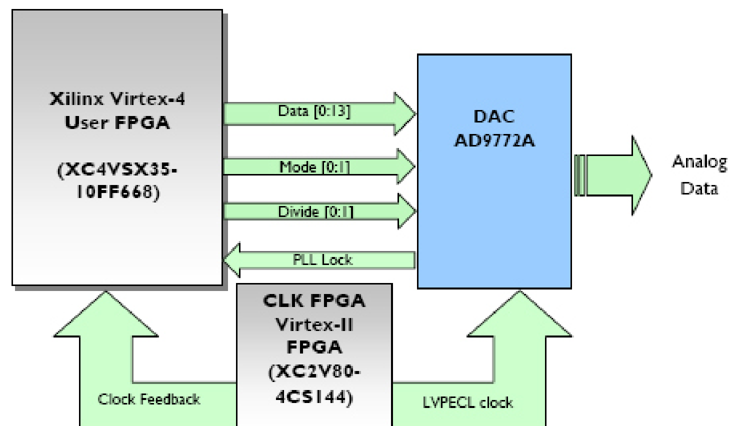


Fig. 3.5: A schematic overview of the DAC to FPGA interface

The hardware pin connections can be found in the appendix 8.1.2.

3.2.3 ZBT-Random Access Memory (RAM)

The ZBT-RAM (Zero Bus Turnaround - Random Access Memory) can only be accessed through the User FPGA. It can be used to store the experiments data and can be read out after the experiment is finished. It is important to know that the RAM is **NOT** erased when the FPGA is reprogrammed.
The RAM is organized in two memory banks A and B, each consisting of 2^{19} fields with 32-bit data with which makes 8MB each.
A schematic view for the data flow of the ZBT-RAM can be seen in Fig. 3.6.

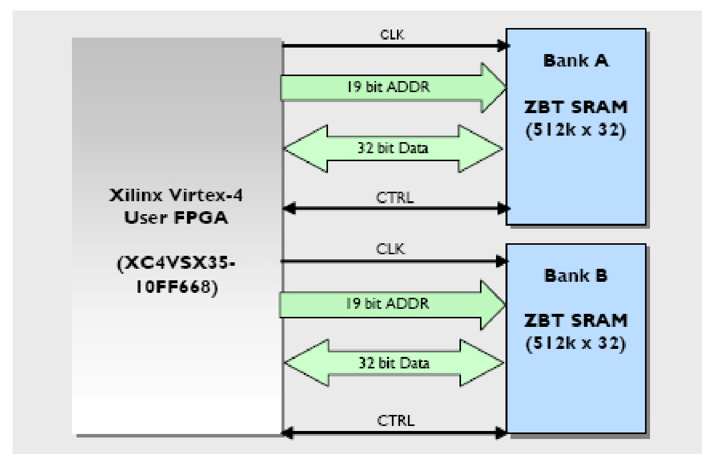


Fig. 3.6: A schematic overview of the ZBT-RAM

The reading out and writing to the RAM from the host computer will be discussed later in the section "Connection to ZBT-RAM via PCI" (3.3). Writing to and Reading from the RAM is mainly done by the RAM-controller element in Simulink which will be discussed in the section "User defined modules" (4.2).
The hardware pin connections can be found in the XtremeDSP Development Kit-IV Reference Guide.

3.2.4 User-LEDs

There are 4 User-LEDs mounted on the board which can be used as a very simple way for signaling the process of a running program on the FPGA.
The hardware pin connections can be found in the appendix (8.1.4)

3.2.5 User-pins

For triggers and other 1-bit data output or input it is very handy to have these user pins. 4 of them are connected to a BNC interface at the back of the host pc for easier access.
If one uses one of those pins to trigger an event, the input voltage must be above +1V in order to set the pin to "high" (= '1'). It is important **NOT** to have higher voltages than +3.3V since the pin header is directly connected to the FPGA which is **NOT** 5V tolerant!
If a pin is used as output the output voltage will be either 0V or +3.3V ('0' or '1')
The hardware pin connections can be found in the appendix (8.1.3)

3.2.6 FPGAs

There are two FPGAs mounted on the board: a User FPGA (XC4VSX35) and a Clock FPGA (XC2V80). Usually a standart bitfile (pl_clock.bit) is written onto the Clock FPGA. This clock is phase locked with an incoming 10MHz signal and internally transforms this into a 100MHz signal.
The user application is written onto the User FPGA. The transfer of the generated bitfile (user application) onto the FPGA will be explained in section 4.5

3.3 Connection to ZBT-RAM via PCI

The main interface between the Virtex-4 board and the host PC is the PCI slot. The communication between the board and the PC is done using the "DIMEscript" protocol. It can be either used within a console program (dimecontrol.exe) or there are also C libraries provided which makes it possible to write customized C++ applications. Together with Deniz Bozyigit

I wrote a simple C++ console application which can be used to read from or write to the ZBT-RAM. It needs the following input parameters:

- -r or -w for read or write operation
- filename (file to be written/read)
- range of data to be read (A-nnn+xxx)
  - A: Memory bank A or B
  - nnn: Base address
  - xxx: Words to be read

```
H:\Share VM\vsprojects\ZBT_readwrite\zbt_readwrite\Release>zbt_readwrite.exe -r
test.data A-0+80
Opening card...
RAM is read to file test.data
Finished
```

Fig. 3.7: C++ console program to read from / write to RAM

4 Programming the FPGA

The most basic way of programming a FPGA is using HDL. In contrast to programming languages like C++ the timing of operations must be stated explicitly. Since timing in electronic circuits is very crucial it must be handled with care, especially since the compiler's possibilities to track down timing errors are limited.
There is also a schematic way of programming FPGAs where subunits which are to be implemented in HDL such as flip flops, RAM or other modules are visualized by schemes with their specific input and output ports which then can be connected graphically.
The most top-level way of generating hardware code is using Simulink in Matlab. This way works graphically as well but also takes care of the timing. This is very convenient and works very well for basic applications but, as it turned out later, is limited in its applications.
For programming the FPGA using Simulink the following software needs to be installed:

- **Matlab**
  Simulink is part of Matlab and usually installed by default.
- **Nallatech FUSE**
  FUSE (Field Upgradeable Systems Environment) provides the ability to control and configure the FPGAs and provides facilities to transfer data between the board and the host PC either by a standalone program or a C based API.
- **Xilinx ISE Design Suite**
  Be sure to choose the right suite. It must be a 60-day evaluation version. It contains a HDL compiler and DSP tools which are automatically integrated into Simulink.

At the end of every compiling process there is a bitfile which then has to be loaded onto the FPGA. This is described in section "Uploading the generated bitilfe onto the FPGA" (4.5).

4.1 Standard modules

There are a lot of different modules but I want to describe the most basic ones which are used frequently.

4.1.1 System Generator

System Generator **MUST** be included in every model file. It is used to compile the model either into a bitfile or a NGC netlist.

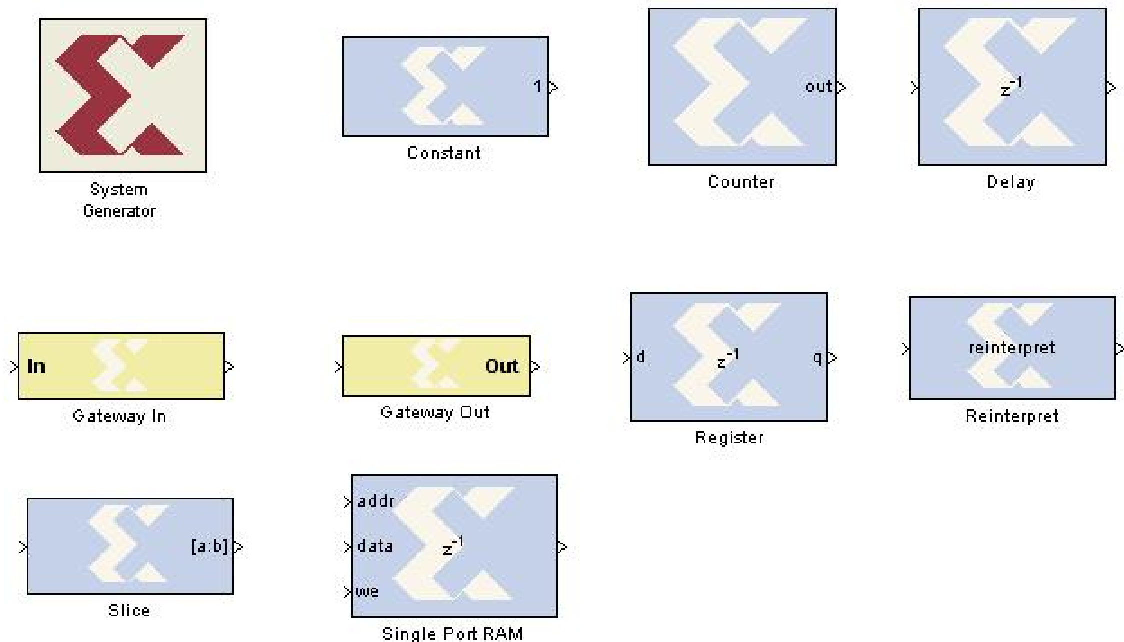


Fig. 4.1: Simulink symbols for commonly used modules

Double clicking on the symbol opens a window where different options can be chosen. Fig. 4.2 shows the adjustment as it should be in order to create an user application which can be integrated into a HDL wrapper.

4.1.2 Constant

The constant module holds a constant value of specific type which can be chosen by double clicking on the symbol.

4.1.3 Counter

The counter can be used to count from a initial value to a end value in integer steps either upwards or downwards. It is also possible to implement it as free running. The output type can be chosen in a similar way as for the constant. It is also possible to provide an enable port which allows to pause the counting process.
Counters are used for example to select the address of the RAM where data should be written to.

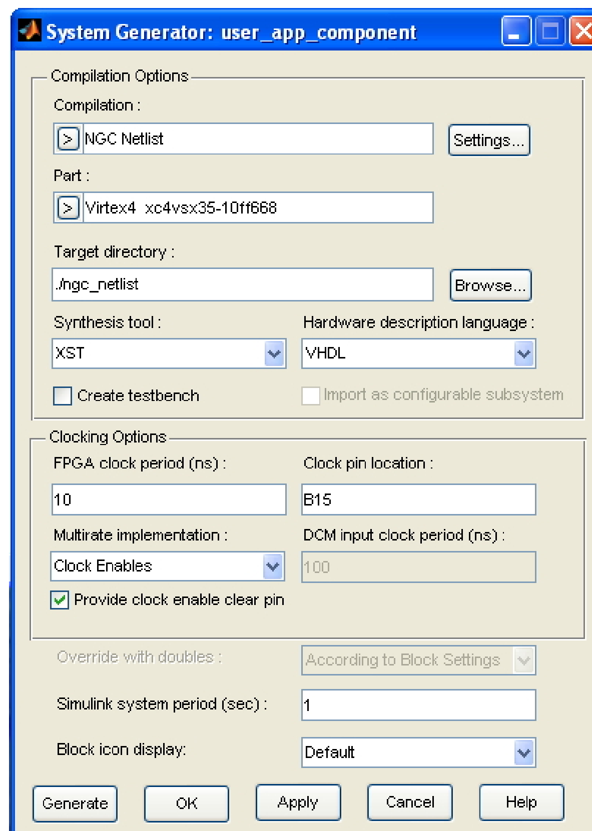


Fig. 4.2: Options for System Generator

4.1.4 Delay

The delay module basically just delays an incoming signal for a desired number of steps. It is really important to use delays in an adequate way to be sure that the data is at the right place at the right time.

4.1.5 Register

A register is used to physically store data which for example comes from the DAC. This sometimes is necessary since you can be sure that the desired value is stored in the register as long as you want it to be kept.
It is possible to provide an enable and a reset port.

4.1.6 Reinterpret

The reinterpret module reinterprets an incoming binary sequence in a different way. This is used if you want to store for example 14-bit wide signed value into the RAM where a 32-bit

wide input is necessary. The bit pattern is **NOT** altered.

4.1.7 Slice

The slice module slices a number of bits out of an incoming binary sequence. The width of the slice and the relative starting point (MSB, LSB, binary point) can be chosen by double clicking on the symbol.

4.1.8 Single port RAM

The Single port RAM module implements a RAM module on the FPGA where temporary data can be stored during for example the measurement process. If the FPGA is reprogrammed the data can **NOT** be accessed any more since it is deleted.
To write data into the RAM the we-input (write enable) has to be set high and an address has to be provided. The data from data input is then stored in the RAM. The reading out works in a similar way, but the we-input must be low.

4.1.9 Gateway In/Out

The in/out-gateway module provides a physical connection to a pin on the experimental board. The pin can be selected by double clicking on the symbol and then choosing the "Implementation"-tab.
It is also used during simulations in Matlab to convert a "Matlab signal" into a "Xilinx signal" so that standard Simulink modules can be connected to a Xilinx module.

4.2 User-defined modules

During the testing phase I found out that a lot of modules shipped with the Xilinx DSP tools for Matlab didn't quite work as they should. One major issue was the reading from and writing to ZBT-RAM. So I wrote together with Deniz Bozyigit some user defined components which will be introduced in this section.
If you want to create your own user-defined module you just need to add the desired modules to the model, then mark them, right-click and choose "Create subsytem". It is possible to further customize the new module by right-clicking on the new subsystem and choosing "Mask subsystem". This is very useful since it makes it a lot easier to understand the program.

4.2.1 ZBT-RAM module

The ZBT-RAM module works like the Single port RAM described in the section 4.1.8.

Fig. 4.3: Simulink symbol for ZBT-RAM

4.2.2 ADC module

Fig. 4.4: Simulink symbol for ADC

The ADC module is really simple. There is just the 14-bit output of the converted analog signal and an OVR-signal which is high in case that the analog input signal was out of bounds. The dataIn is just used if you want to run a simulation in Matlab. Then you can connect a for example a "Sine Wave" (This specific module can be found in the Simulink library under: Simulink → Sources. To learn more about running simulations in Matlab read section 4.4.

4.2.3 DAC module

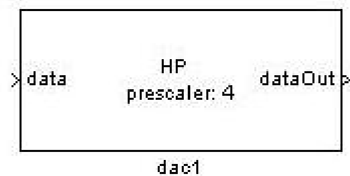


Fig. 4.5: Simulink symbol for DAC

The DAC module is rather simple as well: There is just an input port for the digital data. The output can be used when running a simulation in Simulink.
There are different options for setting up the DAC as described in Fig. 3.5. This setup can

simply be done by double-clicking on the symbol.

4.2.4 LED module

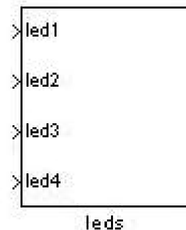


Fig. 4.6: Simulink symbol for LEDs

The LED module is simple and straight forward to use. There is a boolean input port provided for every LED. By double-clicking on the symbol you can choose whether the LED should be on if the input is "high" or "low".

4.2.5 User-pins module

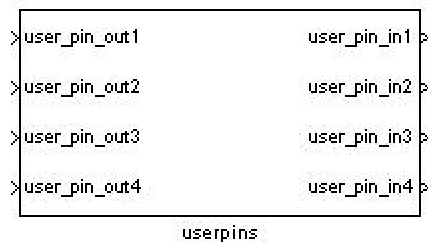


Fig. 4.7: Simulink symbol for User-pins

The user-pins module provides 4 boolean inputs and 4 boolean outputs. By double-clicking on the symbol you can choose which of the 4 ports should be used as input or as ouput. It is **NOT** possible to operate one pin in both directions. This feature is generally not supported in Simulink. If it should be necessary to have this functionality then there has to be programmed a module in HDL.

4.3 Compiling a model in Simulink

It is pretty simple to design a model in Simulink. The needed modules (see above) just have to be dragged from the Simulink library into the model file. The user-defined modules mentioned

above should always be included in the model if you want to compile it using the HDL wrapper. The modules are then connected by wires.
A sample program could look like Fig. 5.2.
It needs to be compiled using the system generator. Double clicking on the icon allows to choose some options which should be set like in Fig. 4.2. Then Simulink creates a netlist of the user application which then can be wrapped into a HDL module which was written by Deniz Bozyigit and compiled using Xilinx ISE Design Suite (For detailed explanation of this step please read Deniz Bozyigit's semester thesis.). This creates a bitfile which needs to be programmed on the FPGA.

4.4 Running a simulation in Simulink

One very handy thing about Simulink is that it is possible to simulate an application before it is generated. This makes it easier to find errors and find out whether the application acts in the desired way.
In Fig. (5.2) you can see an example for a simulation in Simulink. It behaves in exactly the same way as the program which is generated from the model file except for the fact that only 100 pulses are recorded. For a detailed description pleas read chapter 5.
The only difference is that the input ports which are in hardware connected by for example an oscilloscope or the trigger signal now are fed by Simulink modules. There is a wide variety of modules and they are very simple to use.
One has to take care of the fact though, that for some Xilinx modules it is important to convert a "Simulink-signal" into a "Xilinx-signal" in order to get the simulation running. This is simply done by using a "Gateway-In" which automatically does this conversion.
Here is the sample output of the "input scope" (Fig. 4.8) and the "ram scope" (Fig. 4.9). The input scope shows the trigger signal which indicates when the FPGA should record the input data and the input signal (yellow) with noise and the bare signal (purple). The ram scope shows the accumulated signal where you can see very well that the noise cancels out and it also shows the ram address where the data is stored.

4.5 Uploading the generated bitilfe onto the FPGA

The bitfile is uploaded onto the FPGA using DIMEscript. One needs to write a dsc-file which looks like Fig. 4.10.

Executing dimecontrol.exe and entering "call filename.dsc" will execute the dsc-file which then will upload the user application bitfile onto the main FPGA and an application which controls the clock onto the clock FPGA (See Fig. 4.11). Once the applications are loaded onto the FPGAs they start running immediately.

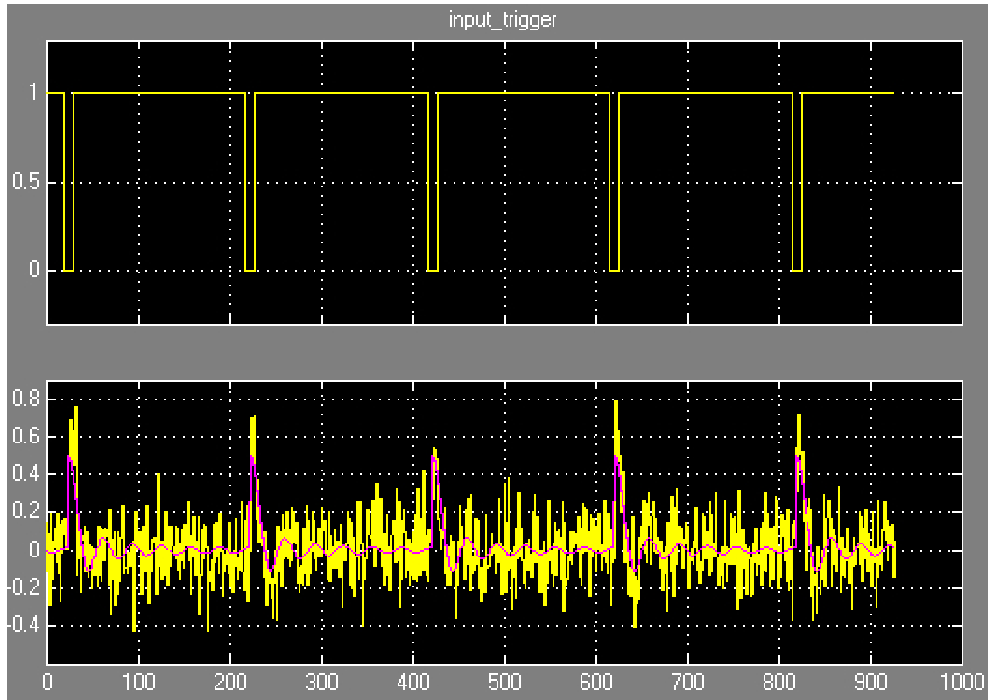


Fig. 4.8: Shows the trigger signal and the input signal with and without noise

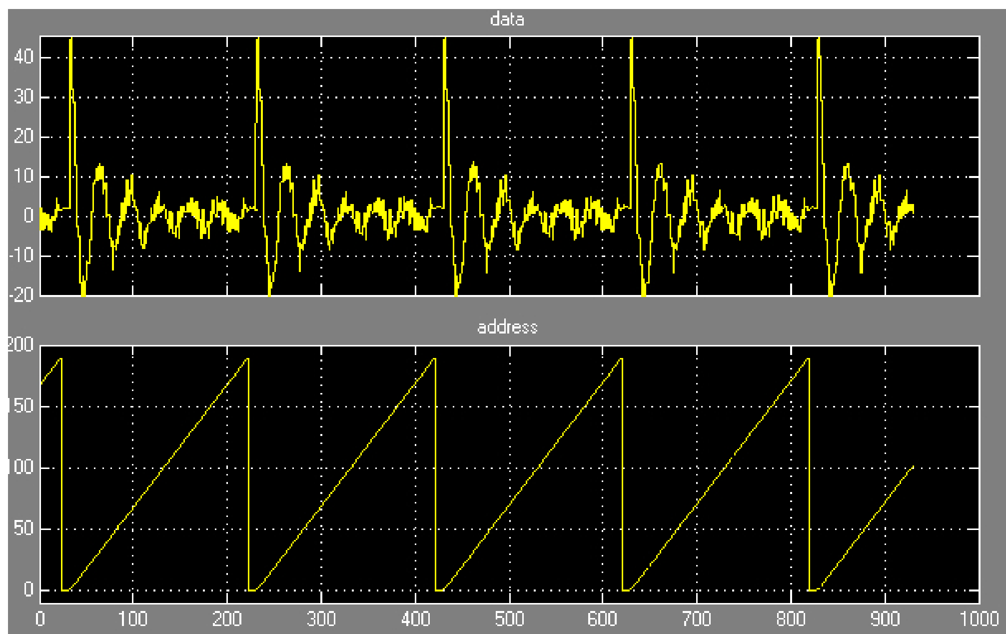


Fig. 4.9: Shows the accumulated signal and ram address

```
# First loading script for the averager
# Author: Deniz Bozyigit
# Date: 16.1.2008
# Version: 0.1

opencard pci

sysreset e
fpgareset e

config 0 0 ../@bitfiles/pl_clock.bit
config 0 1 ../averager/toplevel_wrapper.bit

print averager is running on phase locked clock from port B15 now

pcireset
fpgareset d
sysreset d

closecard
```

Fig. 4.10: Sample file for uploading the user application onto the FPGA

Fig. 4.11: Loading the user application onto the FPGA

5 A first hardware simulation

After some intense experimenting with different ways of programming the Virtex-4 board it was finally possible to make a first hardware simulation with data which was comparable to the one provided during the experiment.
The goal of this experiment is to show how to use the main features of the FPGA and their abilities.

5.1 Experiment setup

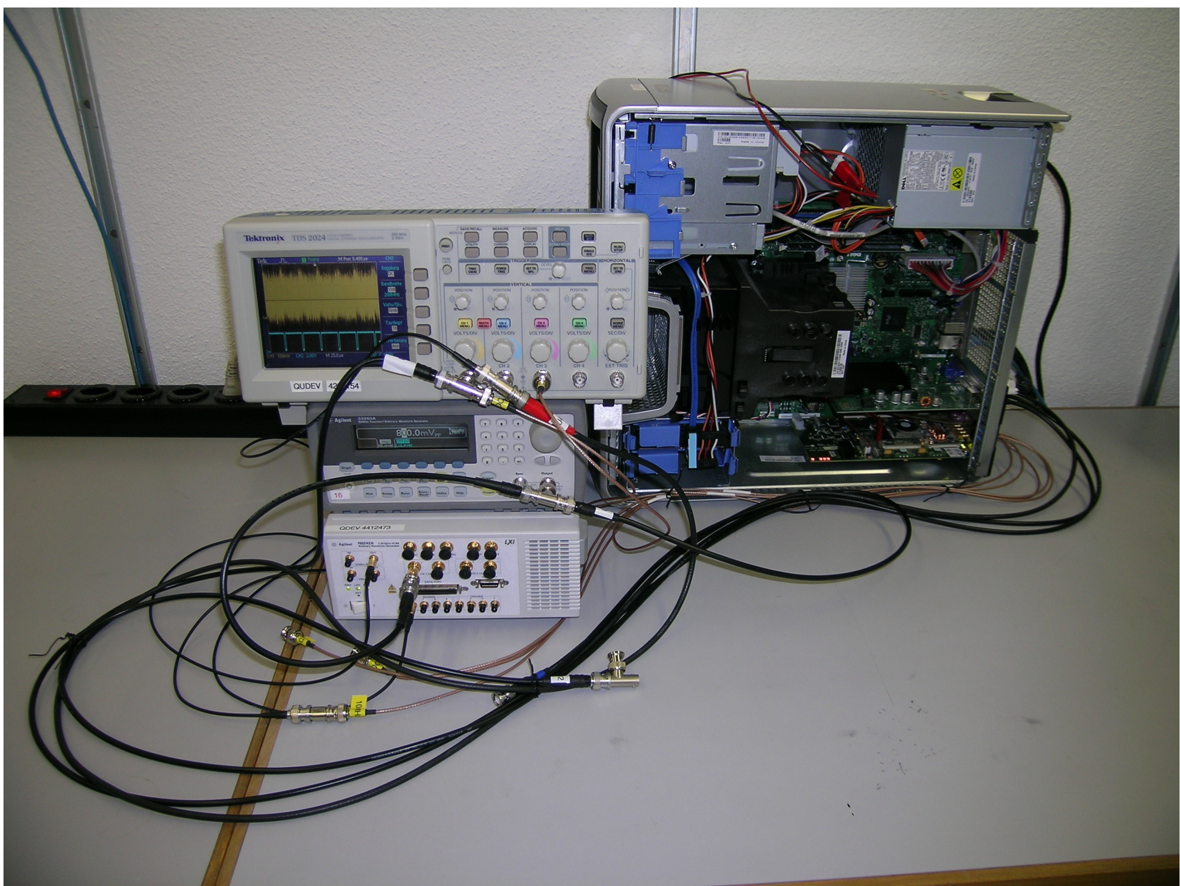


Fig. 5.1: Experiment setup

An Agilent Arbitrary Waveform Generator generates a sequence of 100 rectangular pulses of exponentially distributed length which correspond to a qubit being excited and decaying after some mean lifetime T_1. The mean lifetime in this simulation is set to $T_1 = 5\mu s$. There is also a trigger signal generated which is "high" $2\mu s$ before the positive edge of the rectangular pulses and stays "high" for $37\mu s$. The trigger is "low" for $3\mu s$ between two pulses. The trigger signal ("input_trigger") is connected to user-pin 4 and the rectangular pulse signal ("input_with_noise") is connected to ADC1. In between the waveform generator and the ADC1 there is a second source which adds white noise to the pulse signal. The signal voltage p-p is 100 mV and the Noise is 800mV p-p and has an offset of 200mV.

5.2 The user application

The user application does a very simple averaging of the input signal. It records the same noisy signal from each trigger cycle and simply adds the digitalized data to the already accumulated data stored in the RAM from the previous cycles. This leads to canceling out the noise as it can be seen in Fig 4.8 and Fig. 4.9. Of course one has to take care of overflow of the data since the adding up leads to huge numbers after some time.

The Simulink model of this application can be seen in Fig. 5.2

The user application has the following inputs:

- input_trigger
- input_with_noise

The input_trigger signal is fed into the "maximum event number switch". This module detects the positive edge of the trigger signal and counts the number of pulses. The "input_trigger" signal is emitted without being changed for further processing. There is no delay of the signal. The "finisched_l" signal is high as long as the "max event no" is not reached. If it is reached then the LED 4 lights up which indicates that the measurement is done.
The "input_trigger" signal enables the averager module and a counter which provides ascending addresses (beginning at zero at the beginning of every trigger pulse and then counting upwards) for the RAM which is included in the averager module. The "input_with_noise" signal is digitalized in the ADC1 module and casted into a signed 14-bit floating point number with decimal point at position 13. This is connected to the "increment" input of the averager module. The inside of the averager module can be seen in Fig. 5.3. The main part of this module is a dual port RAM. There are two input ports (a and b) which both are physically connected to the same RAM. In a first step the address from the counter (addr) is pushed to "addra". This leads to a read-out of the memory at address "addr" which is then available at the ouput "A". This all happens within one clock cycle.
Now in the second clock cycle the data from the increment and the data which was read out from the RAM are added in the "AddSub". This value is then stored at the same address

("addr") as before. This means that the old value at "addr" is overwritten by the incremented value.
A copy of this incremented value is stored in the ZBT RAM at the same address ("addr=addrOut").
The data ("dataOut") therefore needs to be reinterpreted as 32-bit number.

5.3 The recorded data

After having recorded 200000 pulses the recording process is stopped. Then the data which is stored in the ZBT-RAM can be read out as described in section 3.3.
It is expected that a plot of the data shows an exponential curve without noise added to it.
Fig. 5.4 shows the plotted data with and without noise added to the pulses. It can be clearly seen that the noise pretty much cancels out. The x-Axis is the time axis (in seconds) and the y-Axis corresponds to the accumulated signal in arbitrary units.
The plot of the pulses with noise is shifted upwards since the noise has a 200mV offset

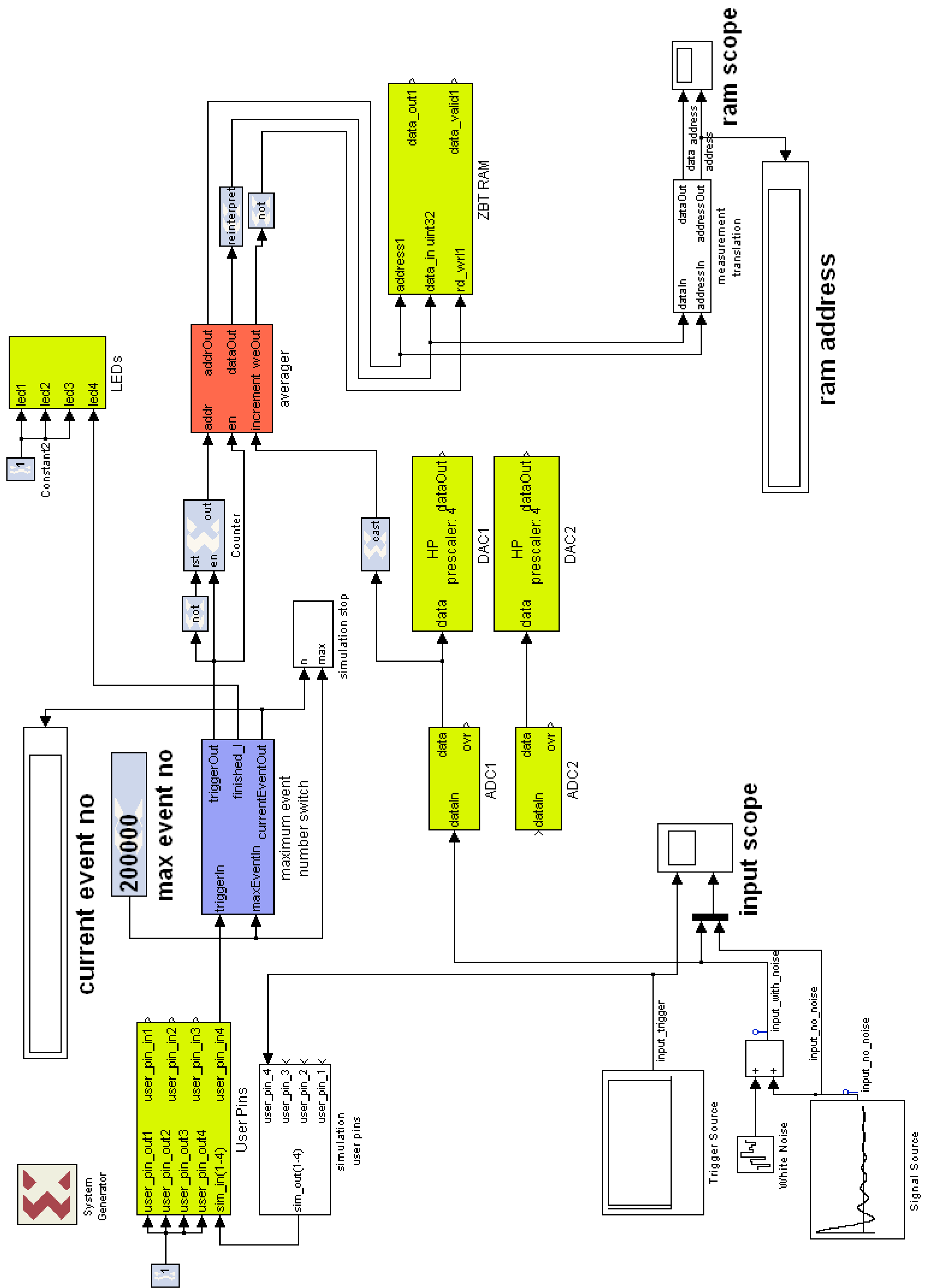


Fig. 5.2: Simulink model

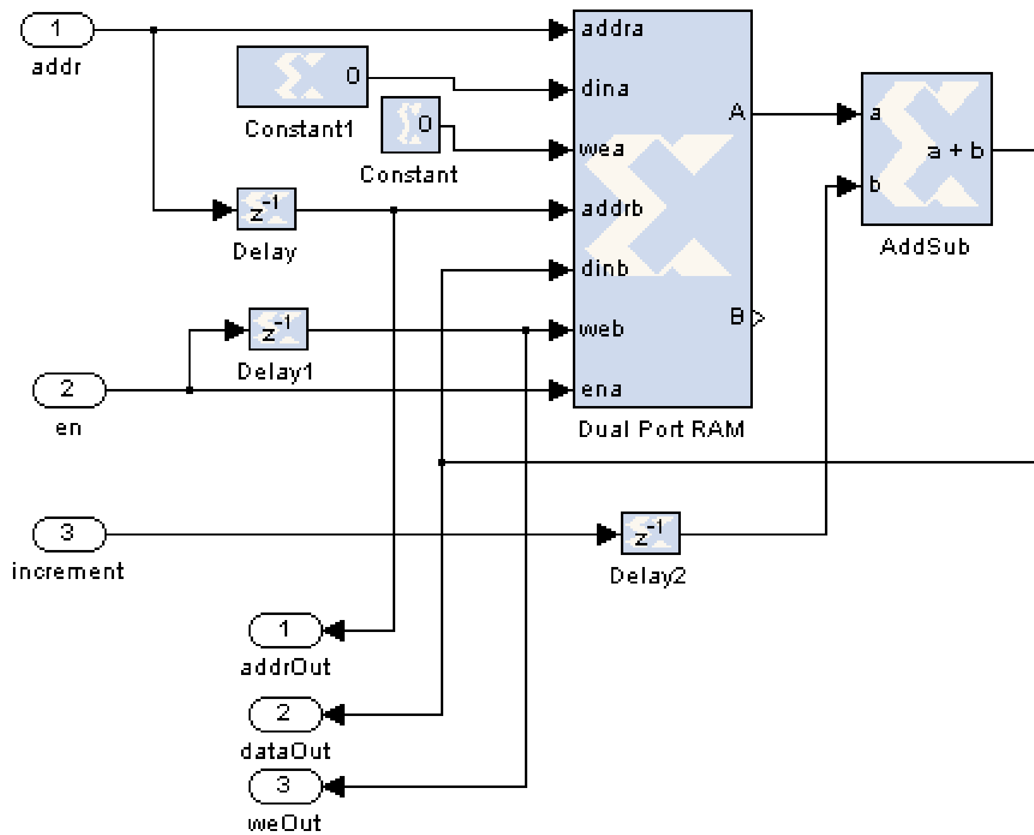


Fig. 5.3: Simulink model of the averager module

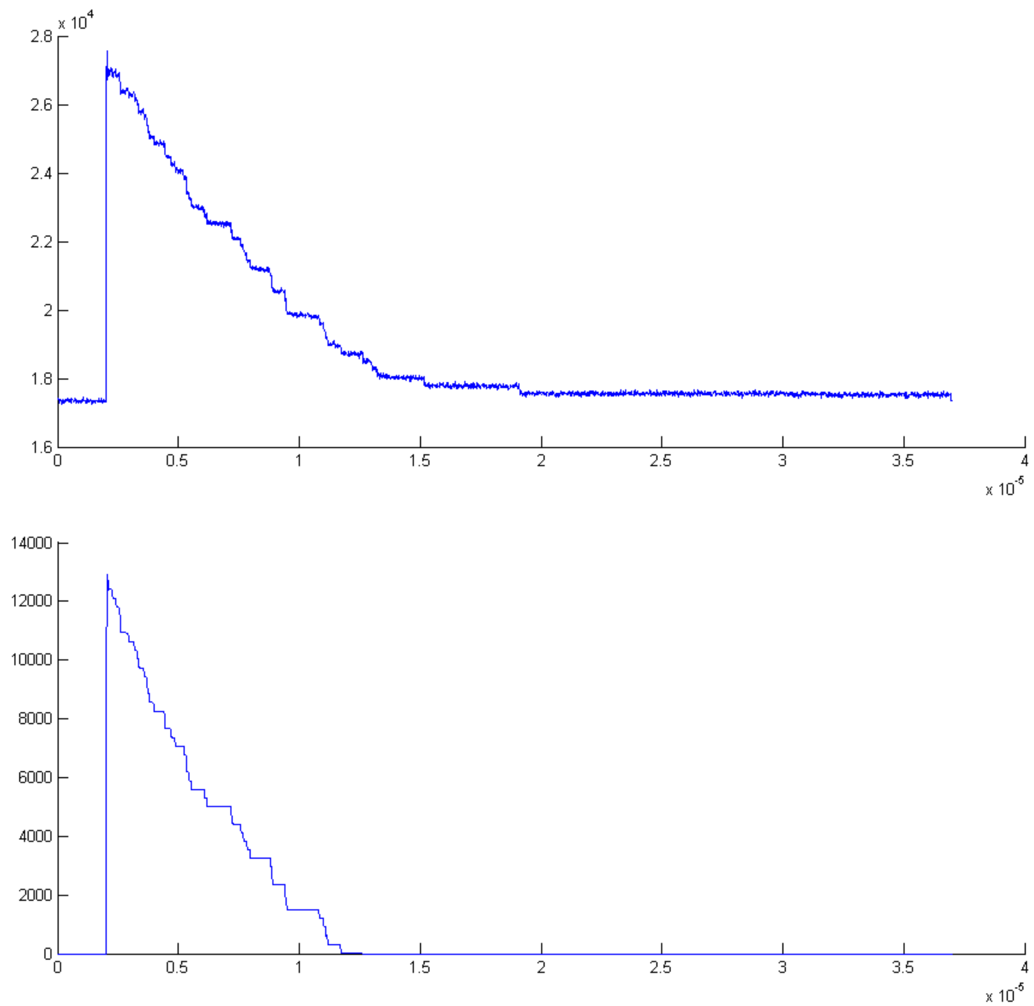


Fig. 5.4: Graphical output (with noise above, without noise below)

6 Summary

As a conclusion I can state that the complexity of the experimental board is huge and it is not trivial to implement a working program on the FPGA. The design has to be handled with care. But the last experiment with sample data also revealed the power of the FPGA in signal processing.
The main problems concerning the fundamental programming of the FPGA are now solved. The basic tools like RAM, ADC, DAC, LEDs and User-pins can now be controlled. With the Simulink modules and the HDL-wrapper it is now possible to generate a program in a nice and understandable way since the "ugly" details like the timing and so on are hidden and taken care of.
The next steps towards a single-shot read-out now have to be taken by implementing a suitable filter function onto the FPGA. I am convinced that it is possible to reach this goal.

7 References

- Xilinx XtremeDSP Development Kit-IV User Guide
- "Wiring up quantum systems", nature vol. 451, 7. Feb. 2008
- "Approaching Unit Visibility for Control of a Superconducting Qubit with Dispersive Readout", PRL 95, 060501

8 Appendix

8.1 Pin connections

8.1.1 ADC

Signal Name (DAC 1)	User FPGA (XC4VSX35-10FF668) PIN No	Signal Name (DAC 2)	User FPGA (XC4VSX35-10FF668) PIN No
ADC1_D<0>	C17	ADC2_D<0>	A24
ADC1_D<1>	D19	ADC2_D<1>	D25
ADC1_D<2>	D20	ADC2_D<2>	C26
ADC1_D<3>	C21	ADC2_D<3>	B23
ADC1_D<4>	B18	ADC2_D<4>	B24
ADC1_D<5>	D18	ADC2_D<5>	C25
ADC1_D<6>	C19	ADC2_D<6>	D22
ADC1_D<7>	C20	ADC2_D<7>	C24
ADC1_D<8>	B20	ADC2_D<8>	A21
ADC1_D<9>	B17	ADC2_D<9>	D24
ADC1_D<10>	A17	ADC2_D<10>	C23
ADC1_D<11>	A18	ADC2_D<11>	D23
ADC1_D<12>	A19	ADC2_D<12>	A22
ADC1_D<13>	A20	ADC2_D<13>	C22
ADC1_DRY	D21	ADC2_DRY	B21
ADC1_OVR	D17	ADC2_OVR	A23

Fig. 8.1: ADC signals

$ADC_D\langle 0\rangle$-$ADC_D\langle 13\rangle$ are the pins for the 14-bit data output of the digitalized signal ($ADC_D\langle 13\rangle$ is the most significant bit (MSB)). The binary output format can be chosen but it is commonly a 14-bit floating point number with the decimal point at position 13.
ADC_DRY signalizes if the data is ready
ADC_OVR: if high, then the analog input exceeds $\pm$ Full Scale (FS)
There are no set-up or control signals.

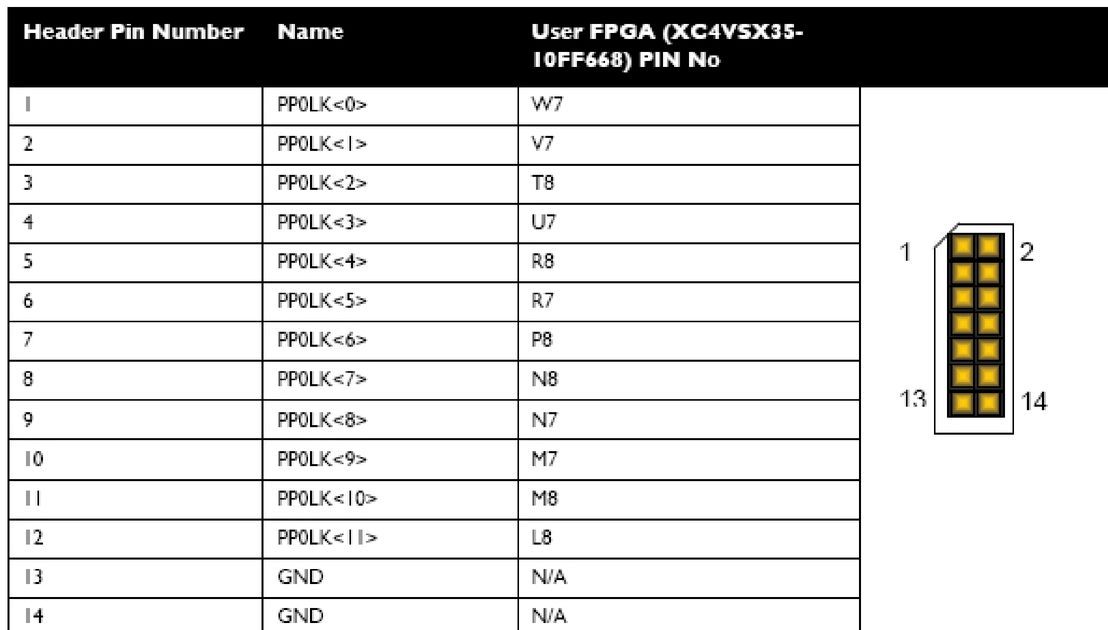
8.1.2 DAC

Signal Name (DAC 1)	User FPGA (XC4VSX35-10FF668) PIN No	Signal Name (DAC 2)	User FPGA (XC4VSX35-10FF668) PIN No
DAC1_D<0>	A7	DAC2_D<0>	D10
DAC1_D<1>	C7	DAC2_D<1>	F10
DAC1_D<2>	B7	DAC2_D<2>	C10
DAC1_D<3>	C5	DAC2_D<3>	E10
DAC1_D<4>	D4	DAC2_D<4>	D9
DAC1_D<5>	C4	DAC2_D<5>	F9
DAC1_D<6>	A4	DAC2_D<6>	E9
DAC1_D<7>	B3	DAC2_D<7>	A9
DAC1_D<8>	B6	DAC2_D<8>	D8
DAC1_D<9>	E6	DAC2_D<9>	C8
DAC1_D<10>	D6	DAC2_D<10>	E7
DAC1_D<11>	A6	DAC2_D<11>	D7
DAC1_D<12>	A5	DAC2_D<12>	B9
DAC1_D<13>	B4	DAC2_D<13>	F7
DAC1_DIV0	F4	DAC2_DIV0	C12
DAC1_DIV1	D5	DAC2_DIV1	C13
DAC1_MOD0	A3	DAC2_MOD0	A8
DAC1_MOD1	E4	DAC2_MOD1	F8
DAC1_PLLLOCK	C6	DAC2_PLLLOCK	B10
DAC1_RESET	E5	DAC2_RESET	A10

Fig. 8.2: DAC signals

$DAC_D\langle 0\rangle$-$DAC_D\langle 13\rangle$ are the pins for the 14-bit data input of the digital signal ($DAC_D\langle 13\rangle$ is the most significant bit (MSB)). The binary input format is binary offset
DAC_MOD and DAC_DIV signals are explained in section 4.3.2
$DAC_PLLLOCK$
DAC_RESET

8.1.3 User pins



Header Pin Number	Name	User FPGA (XC4VSX35-10FF668) PIN No
1	PP0LK<0>	W7
2	PP0LK<1>	V7
3	PP0LK<2>	T8
4	PP0LK<3>	U7
5	PP0LK<4>	R8
6	PP0LK<5>	R7
7	PP0LK<6>	P8
8	PP0LK<7>	N8
9	PP0LK<8>	N7
10	PP0LK<9>	M7
11	PP0LK<10>	M8
12	PP0LK<11>	L8
13	GND	N/A
14	GND	N/A

Fig. 8.3: User pins

Pins 9-12 are connected to BNC adapters at the backside of the host PC

8.1.4 LEDs

Signal Description	User LED	Signal Name	Main User FPGA (XC4VSX35-10FF668) Pin
Green Diode for LED2	D2	LED_Green2	D3
Red Diode for LED2	D2	LED_Red2	F3
Green Diode for LED1	D1	LED_Green1	E26
Red Diode for LED1	D1	LED_Red1	D26

Fig. 8.4: User pins