Using the Zynq-7000 XADC and signal pre-conditioning

Author

Pascal Engeler

engelerp@phys.ethz.ch

Supervisor

Vlad Negnevitsky

nvlad@phys.ethz.ch

Institution

Trapped Ion Quantum Information Group

D-PHYS, ETH Zurich, 2017

Abstract

This report describes the findings of a project conducted in summer 2017 by Pascal Engeler. The goal of the project was to get a specific analog digital converter running such that photo diodes measuring LASER intensities can be monitored. The first part summarizes the functionalities of the `XADC` part of the Zynq-7000 SoC. In the second part a description of how the `XADC` can be programmed and used on the Zedboard is given by using hands-on examples. The final part concerns the properties and usage of a signal pre-conditioning PCB that was designed, assembled and tested as part of the project.

Contents

1 Introduction

LASERs are used to carry out operations on qubits encoded in electronic states of ions. As coherence of the qubit states must be preserved during the computation, precise control of the operating LASERs is necessary, especially the intensity must be well known such that the duration of pulses can be set correctly. In this project, developments in the direction of real-time intensity monitoring and control were made. To keep things simple, the analog digital converter (ADC), called XADC, present on the Zedboard's Zynq-7000 system-on-a-chip (SoC) is used, because this board is already present in the experimental setup and controls the synthesis of LASER pulses. Even though this ADC is too slow to monitor LASERs in real time, the efforts necessary to port the tools developed here to a faster ADC later on are greatly reduced by the experience gained in this project.

As a first step it was necessary to investigate how the XADC works and what it can and cannot do. This includes figuring out the functioning of the driver shipped by Xilinx. The results are presented in sections 2 and 3. Having the XADC running, we developed a printed circuit board (PCB) that is used to precondition signals that are to be monitored. Hence signals are first passed through this PCB before being fed to the XADC. The purpose of this board is filtering, amplifying, multiplexing and possibly shifting signals. In section 4 the functionality of this PCB is described.

In the next section an overview of the XADC is given.

2 Properties and Functionality of the XADC

The XADC is an analog-digital converter part of the Zynq-7000 chip present on the Zedboard. In the following sections the specifications and functionality of the XADC will be summarized and properties that were measured in the project will be stated.

2.1 XADC specifications

The XADC is a dual 12-bit precision ADC that takes input in a 1 Volt range. Thus the conversion error (maximum difference in voltage that will not flip any bit) amounts to $\Delta V \approx 0.25$ mV. The sample rate is 1 MHz, which means $1\ \mu$s / sample in theory, in practice this is only reached under special conditions.
Even though the XADC contains two ADCs, we will mostly restrict the discussion to only one of them. The second ADC can be used to sample different signals simultaneously in order to preserve their phase relationship, or one ADC is used as a watchdog for board sensors while the other fulfills other tasks.
The following sections will give an overview of the operational features of the device.

2.1.1 Channels

A number of distinct channels are present on the XADC on which conversions can be triggered. They can be classified into two categories, internal sensors and analog input. The internal sensors that can be accessed are

- Die temperature sensor,
- 6 power supply sensors.

By virtue of these sensors the XADC can be used as a monitoring device. Alarms can be set that trigger interrupts when the sensors report values outside of user-set thresholds. Also, the XADC keeps track of the highest and lowest values recorded since turn-on.
The analog input channels are

- Dedicated analog inputs (VP/VN),
- 16 auxiliary analog inputs (AUXP1/AUXN1 - AUXP16/AUXN16).

Each channel corresponds to an input pair, the XADC takes signals that are differential and the conversion result is the difference between the two channels. This is useful as it cancels any noise that is common to the two inputs. The dedicated channel can only be used for analog input and is active as soon as the XADC is instantiated in a design, while the auxiliary channels can be configured for digital input too and must be constrained separately. Other than that the main difference between dedicated and auxiliary channels

lies in the resistance present between the input pins and the sampling capacitor. The dedicated pair VP/VN has $100\ \Omega$, while the auxiliary channels have 10 kΩ. This means that the dedicated pair is around 100 times faster than the auxiliary inputs from an acquisition-time point of view, as described in the user guide [1], page 30.
All inputs can be configured to either unipolar (but differential nonetheless) or bipolar mode. In unipolar mode the difference P − N between the two poles must lie in the range $[0,\ 1]$ V, while the common mode voltage (with respect to the `XADC` ground `GNDADC`) must be between 0 V and 0.5 V. Bipolar mode supports P − N $\in [-0.5,\ 0.5]$ V with the same common mode voltage restriction.

2.1.2 Sampling settings, channel sequencer and single channel mode

The `XADC` contains a channel sequencer which is responsible for the selection of the next channel on which a conversion is to be triggered. It comes with a set of predefined operation modes and the way in which sampling is carried out can be influenced through the following settings:

- Analog-input mode (unipolar / bipolar),
- Averaging (0, 16, 64 or 256 samples),
- Acquisition time (lengthen by 10 ADCCLK cycles).

Analog-input mode has been discussed before. If averaging on a channel is set to n, the `XADC` will acquire n samples from that channel, calculate the (non-rolling) average and only then write the result to the status register. If a channel sequence is used, the whole sequence will be completed n times until a result is written. Acquisition time adjustment can be used to make the `XADC` wait for a longer period of time on a channel until a conversion is triggered. This is useful if the input signal is slow, for example due to the presence of large impedances.
The channel sequencer provides the following modes:

- Default mode
- Single pass mode
- Continuous sequence mode
- Simultaneous sampling mode
- Independent ADC mode

The default (safe) mode should be enabled whenever any sampling-related settings are changed. In this mode the XADC monitors the on-chip sensors with 16 sample averaging. This is the only mode that does not respect any user-defined settings. In all other modes the sequencer passes through a user-defined channel sequence using the settings mentioned earlier. In single pass mode a single pass through the sequence is carried out and subsequently the XADC switches to single channel mode (described later). Continuous sequence mode means that the sequencer will pass through the sequence and restart the sequence once it's finished. In simultaneous sampling mode the two ADCs sample in a time-coherent manner from different channels such that phase coherence can be retained. Independent ADC mode makes one ADC monitor the on-chip sensors while the other ADC can be used for any other application.

Apart from sequence operations the XADC also supports single channel mode. In this mode a single channel is specified from which the XADC obtains values. While it is possible to switch channels in single channel mode, the overhead can be significant if a switch is done after each conversion. The channel sequencer should then be used instead.

2.1.3 Continuous sampling mode and event driven sampling mode

Continuous sampling is the default mode of the XADC. Once a conversion is finished a conversion on the next channel in the sequence is triggered automatically. In this mode high throughput is achieved.

If conversions need to be done at specific points in time, for example as given by an external device, then event mode should be used. Note that event mode expects the DCLK clock to be present. In event mode, the XADC waits for a raising edge on its CONVST port (or CONVSTCLK). The waiting period is used as acquisition time. Once a trigger is registered, a conversion starts on the next rising edge of ADCCLK. Once the result of the conversion has been written to the corresponding register, the EOC (end of conversion) port will be driven high for one DCLK cycle. Note that only with averaging turned off this implies one can now fetch a new result from the status register. With finite averaging one should instead wait for the EOS (end of sequence) port to be driven high for one DCLK cycle.

For more precise information on the timing of the XADC one should consult chapter 5 of the user guide [1].

2.1.4 External multiplexer mode

The XADC can control an external multiplexer (mux) with up to 16 channels. Usually the dedicated VP/VN channel is connected to the multiplexer and the number of the AUX channel nominated by the channel sequencer for conversion is sent to the MUXADDR$[3:0]$ port. Hence one should connect this port to the external multiplexer's control bits.

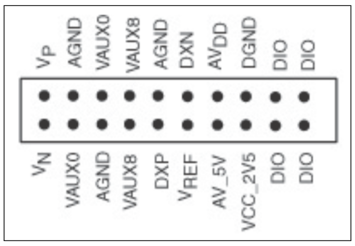


Figure 1: The XADC header on the Zedboard, taken from the user guide [2]. The V_N pin is the pin closest to the nearest corner of the Zedboard.

Pin	Voltage
Vcc_2V5	2.45 V
AV_5V	4.98 V
VREF	1.23 V
DXP	80 mV
AGND	-24 mV
DXN	80 mV
AVDD	1.77 mV

Table 1: Voltages measured on several pins of the Zedboard XADC header. The voltages were measured relative to the same reference. No errors are given as these voltages are quite stable and the measurements are rather a verification of the documentation than a precision-characterization of the Zedboard.

Everything else is controlled in the same fashion as when using the channel sequencer to control the internal multiplexer.

2.2 Accessing the XADC on the Zedboard

The Zedboard provides access to several XADC pins through the XADC header. The pinout is shown in figure 1.

One can see that only three analog input channels, namely VP/VN, AUX0 and AUX8, are exposed. If more than 3 channels are needed, an external multiplexer is a necessity. The DIO pins can be used to control the channel selection of an external mux if they are constrained correctly in the FPGA programming. The AGND pins are connected to the ground of the XADC, GNDADC. The AV_5V pin is the 5 V supply from the Zedboard and can deliver up to 150 mA. The results of measurements on the different (output) pins of the header are shown in table 1.

The Zedboard includes an anti-aliasing filter, shown in figure 2. It has a cutoff frequency of

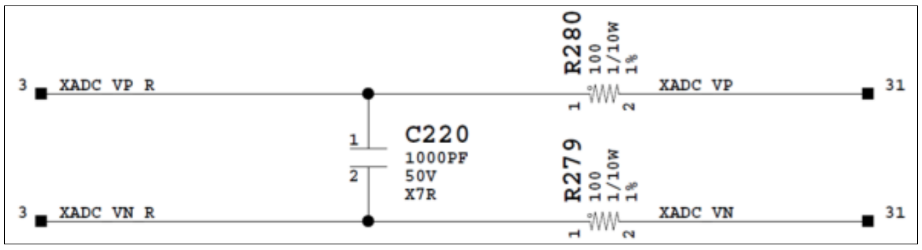


Figure 2: The anti-aliasing filter included in the Zedboard [2]. The cutoff is around $\omega \approx 5$ MHz · rad.

$$f_{\text{cutoff}} = \frac{\omega_{\text{cutoff}}}{2\pi} \approx \frac{1}{4\pi RC} \approx 800 \text{ kHz}.$$

This is a limiting factor of the speed the XADC can be operated at in external multiplexer mode. In order to generate correct results one should take into account that this built-in filter adds an impedance to the VP/VN input channel and slows down the sampling rate.

2.3 Measured properties of the XADC

The XADC was tested and certain properties were characterized. The results are presented in the following sections.

2.3.1 Conversion

The results from XADC conversions are stored in XADC status registers as 12-bit values. As will be explained in section 3, the XADC driver includes macros to convert these values to voltages or temperatures. To test this, a triangle signal at 1 kHz with amplitude 200 mV centered at 0 V was synthesized with the Analog Discovery Kit's DAC and fed into the VP/VN pins of the XADC header. For this measurement the XADC was operated in bipolar mode. The results are shown in figure 3.
One directly sees that the device is not gauged correctly, the maximum is supposed to lie at 0.2 V—as verified with an independent oscilloscope—but it is at about 0.6. The XADC driver supplied by Xilinx suggests that there is automatic calibration functionality, one might try and investigate this. I was unable to make it work, so I just took it as a given that the XADC values do not directly correspond to voltages and one should calculate the conversion function oneself.
For this purpose the triangle output of the Analog Discovery Kit is assumed perfect. Then, for times between 0 ms and 0.5 ms in figure 3, it can be modeled as

$$V(t) = 750\,\frac{\text{V}}{\text{s}}t - 0.2\,\text{V}.$$

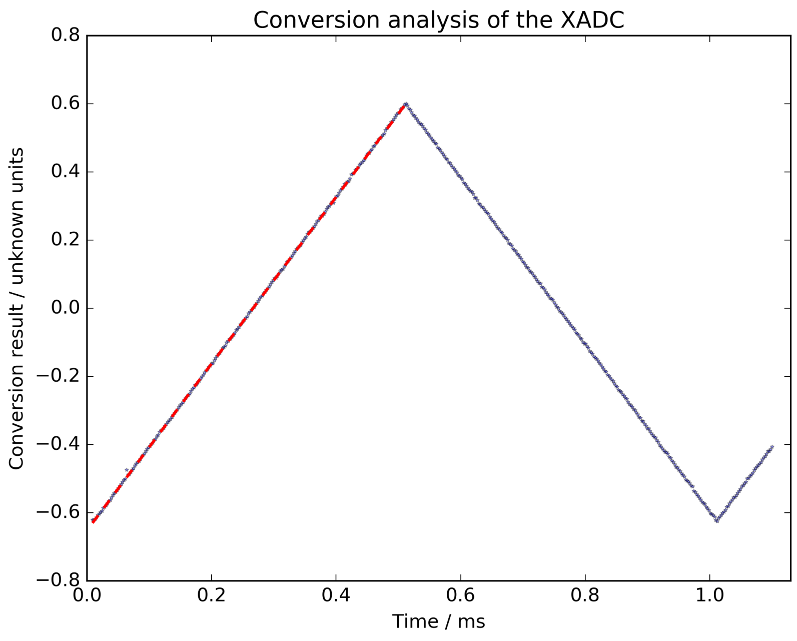


Figure 3: Conversion result of the XADC when a triangle signal from the Analog Discovery Kit (frequency 1 kHz, amplitude 0.2 V) is fed into the VP/VN pins of the XADC header. The red dashed line is a linear least squares fit to the rising edge and is given by $V_{\mathsf{XADC}}(t) = 2446\ \mathsf{Hz} \cdot t - 0.65$.

With this assumption the conversion from XADC macro output to voltage can be calculated from the data in figure 3. It is found to be

$$V_{\mathsf{real}}(V_{\mathsf{XADC}}) = 0.30668\ \mathsf{V} \cdot V_{\mathsf{XADC}} - 8 \cdot 10^{-5}\ \mathsf{V},$$

where V_{XADC} is the (dimensionless) output of the XADC conversion macro and V_{real} is the input voltage we want to recover.

The conversion function should be recalculated before any real-world applications are implemented, preferably using a high-precision DAC. As a rule of thumb one may keep in mind that the XADC conversion macro outputs a value that is around 3 times the applied voltage. Note that this result only holds for the VP/VN input channel, but having tested the AUX channels exposed on the Zedboard I assume they follow similar behaviour.

2.3.2 Noise and drift analysis

The drift of the XADC and the noise produced between the Zedboard XADC header VP/VN pins and the XADC were analysed by shorting the two pins and triggering several conversions per second on the VP/VN channel for about 14 hours. The result can be seen in figure 4.

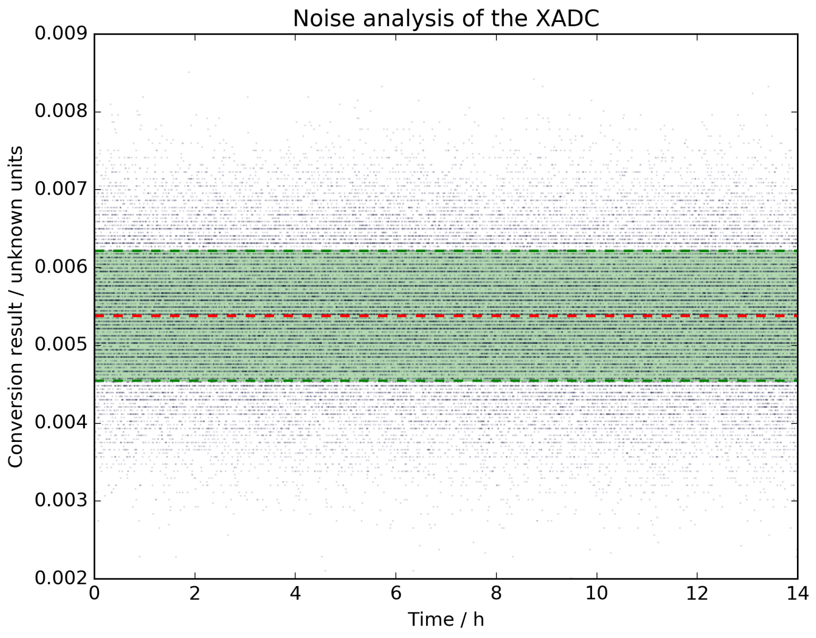


Figure 4: Noise analysis of the `XADC` on the Zedboard. The black dots are conversion results of the VP/VN channel with the corresponding pins on the Zedboard `XADC` header shorted together. The red line is the average value of the data, the green area highlights points closer than one standard deviation from the mean. The mean has y-value 0.0054, the standard deviation is 0.00083. The data allows the extraction of the drift via a rolling average. A rolling average of length 100 on the data yielded a maximum value of 0.0057 and a minimum value of 0.0050, the difference of which is smaller than one standard deviation or less than 3 LSB flips in 12-bit precision on a 1 V range. The two extremal values have a temporal separation of just over 2.5 hours

There are a few peculiarities to be observed in this plot. Firstly, we should keep in mind that the XADC outputs values that are around three times the applied voltage. Hence the mean voltage is around 0.0018 V, and the standard deviation translates to 0.28 mV. While not impossible per-se, the standard deviation is suspiciously small considering the 12-bit resolution of the XADC. Taking a close look at figure 4, the problem becomes evident. While the output values indeed make up a discrete set—the black dots lie on horizontal lines—the spacing between the different values is around 0.00004559, corresponding to about 0.01520 mV. This is much smaller than a flip of the least-significant-bit which corresponds to 0.25 mV. However, this is quite precisely the least-significant-bit flip of a 16-bit ADC on a 1 V input range.
The conundrum is resolved in the following way. The XADC doesn't store the conversion values in 12-bit registers but in 16-bit registers. The additional 4 bits are used to improve precision when averaging is applied. As it seems the noise measurement was carried out with averaging turned on. This is strange, checking the code used for this measurement I find no evidence, neither for nor against this hypothesis. It might be that the bitstream used at the time had averaging turned on by default, unfortunately this can no longer be verified.
The data of figure 4 allows the analysis of the drift. A rolling average of length 100 was calculated on the data and yielded a maximum average of 0.0057 and a minimum of 0.0050, corresponding to a difference of about 0.23 mV. The two extremal values have a temporal separation of about 2.5 hours. Being less than one standard deviation of the data, the drift over 14 hours is negligibly small.

2.3.3 Sampling speed

A first approach to monitoring photodiodes needed the following sequence to be repeated:

- Sample channel 0 of the external mux for 20 μs
- Sample channel 1 of the external mux for 20 μs
- Sample channel 2 of the external mux for 20 μs
- Sample channel 3 of the external mux for 20 μs

For this, one needs to select a channel, trigger conversions for a certain amount of time, switch channel and repeat. In a first approach the channel sequencer with a single channel enabled was used. As it turns out this is suboptimal, because switching channels needs the sequencer to be turned off first and turned back on after the switch. The implied temporal overhead means that switching from one channel to another takes around $10 - 13$ μs, only leaving $7 - 10$ μs to actually obtain samples. Sampling can be done at a rate close to 1 MHz, the duration from the conversion trigger to the EOC

signal is typically around 1.2 μs. However, one needs to fetch the result before the next conversion can be carried out. There is no deterministic timing for this process, it usually increases the time per sample to around 3 μs. One could use averaging such that only one readout per channel is necessary, but the smallest number of samples per average is 16, which needs at least 16 μs of sampling time.
A better approach is using single channel mode, as switching channels can be done in around half the time required when using the sequencer. However, using single channel mode with an external multiplexer was not achieved; the driver contains conflicting information about this functionality and several tests led nowhere.

3 Programming the XADC

This section gives an overview of the usage of the Xilinx provided xadcps driver for the XADC. Interrupt functionality within the Zynq-7000 SoC are treated, especially those originating from within the programmable logic (PL) interrupting the processing system (PS).

3.1 Initialization

The first thing to do when programming the XADC is to include the driver provided by Xilinx. This means

```
#include "xadcps.h"
```

Next up we need to initialize the XADC. This is done through the following lines of code:

```
 1 #define XADC_DEVICE_ID XPAR_PS7_XADC_0_DEVICE_ID
 2 
 3 XAdcPs XADC_Driver_Instance;
 4 XAdcPs_Config* cfg = XAdcPs_LookupConfig(XADC_DEVICE_ID);
 5 if(cfg == NULL)
 6 {
 7         printf("Config Lookup Failed!");
 8         return;
 9 }
10 XAdcPs_CfgInitialize(&XADC_Driver_Instance, cfg, cfg->BaseAddress);
```

First we need to create an object of type XAdcPs, which holds the XADC instances data. Next we need to fetch the config of the device we want to target—namely the XADC—, which is defined through its device ID. This number is defined to be called XADC_DEVICE_ID in the first line, the preprocessor will take care of the rest. Hence we look up the config and keep a pointer to it in an object of type XAdcPs_Config* in line 4. If the lookup fails, this method will return a nullptr, so we can check for errors in lines 5-9. On line 10 we finally initialize the config, don't ask me what exactly this does, but it is necessary. One quick note: I saw several code examples that merely create a pointer (XAdcPs*) instead of an actual object. This seems dangerous to me, if there is never an actual allocation for an XAdcPs object, I would expect one might easily run into memory problems. I think my way is the right way to do it. The XAdcPs* pointer &XADC_Driver_Instance is the central object used to access the XADC. Hence this is the object that is passed to all functions that involve the XADC.
Now the XADC is ready, and its state is as defined in the bitstream (i.e. in Vivado). However, any setting you can apply in the hardware definition—except for the actual wiring—can be changed through the software. If only one setting is necessary, it is easiest to just set it up in Vivado and not worry about it anymore later.

There are now several things one can do. In the following sections we go down different paths to illustrate how the driver works. From here on I will always assume that initialization has been taken care of in the sense of the above.

3.2 Self-Test

The driver comes with a self-test which checks certain functionalities of the XADC. Unfortunately I don't currently have a copy of the driver and thus can't look up what exactly it does. I have never seen it fail, and I guess only a fatal hardware error could trigger failure. The self-test can be called in the following way:

```
1 int Status = XAdcPs_SelfTest(&XADC_Driver_Instance);
2 if(Status != XST_SUCCESS)
3 {
4         printf("Self Test Failed!");
5         return;
6 }
```

Line 1 performs the self test, which returns an integer. It is common that methods which can either fail or succeed return either XST_SUCCESS or XST_FAILURE; hence we can again check for errors. Note that the self test takes a pointer to the XAdcPs object as argument; this is common, all methods that involve the XADC take this as argument in order to target the right instance of the XADC.
An important thing to note about the self test is that it changes the state of the XADC. Hence **if your XADC is nicely set up, do not call the self test as it will reset its state**.

3.3 Setting up the sequencer

The sequencer essentially decides on which channels conversions are triggered and in which way. On one hand one can choose one of several sequencer modes, on the other hand one can enable and disable different channels, choose the input mode for each channel (i.e. unipolar or bipolar), set the averaging for each channel and choose the acquisition time (if the input signal is slow, additional time might be needed for the ADC capacitors to settle, as described in section 2.1.2).
Choosing the sequencer mode is done in the following fashion:

```
XAdcPs_SetSequencerMode(&XADC_Driver_Instance, XADCPS_SEQ_MODE_SAFE
    );
```

In the above example the sequencer mode is set to the default safe mode. The available modes are (as #defined in xadcps.h):

- XADCPS_SEQ_MODE_SAFE: Monitor the internal sensors. It's called safe mode because it allows writing to setup-registers (channel enables, channel input modes, alarm enables etc.) as the XADC functions independently from all of these. Whenever such settings are changed, the sequence should first be put into safe mode, else the setup functions will fail.

- XADCPS_SEQ_MODE_ONEPASS: The sequencer takes a single pass through the sequence. This means it will trigger one conversion on each enabled channel and then stop.

- XADCPS_SEQ_MODE_CONTINPASS: Continuous mode. In this mode the sequencer goes through the sequence of enabled channels and triggers conversions on each. At the end of the sequence it restarts, so it will just keep sampling and converting the enabled channels indefinitely.

- XADCPS_SEQ_MODE_SIMUL_SAMPLING: Simultaneous mode. I'm not sure what it does, I believe it involves simultaneously sampling two channels at once, but this involves two XADCs. Apparently this is useful when phase relationships between signals should be preserved.

- XADCPS_SEQ_MODE_SING_CHAN: Single channel mode. In this mode the XADC only samples and converts a single channel. Further setup is necessary, this will be discussed later on.

- XADCPS_SEQ_MODE_INDEPENDENT: Independent ADC mode. One ADC monitors the on-chip sensors, the other can be used for other applications.

The safe mode is only used when settings are changed. The most useful modes for my purposes were continuous pass mode to just sample a set of channels as fast as possible (I think only this mode can truly come close to the 1MSPS limit) and single channel mode to look at a single channel for as long as desired. One can also use continuous mode to sample only one channel by only enabling one channel in the sequence, which is what I sometimes did, as described in section 2.3.3.

Setting the sequencer mode is in itself not sufficient to operate the XADC. One should also specify some further options, such as channel enables, averaging and input mode:

```
XAdcPs_SetSequencerMode(&XADC_Driver_Instance, XADCPS_SEQ_MODE_SAFE
    );
XAdcPs_SetSeqChEnables(&XADC_Driver_Instance, XADCPS_SEQ_CH_VPVN |
    XADCPS_SEQ_CH_AUX00 | XADCPS_SEQ_CH_AUX07);
XAdcPs_SetSeqInputMode(&XADC_Driver_Instance, XADCPS_SEQ_CH_AUX00 |
     XADCPS_SEQ_CH_AUX07);
XAdcPs_SetAvg(&XADC_Driver_Instance, XADCPS_AVG_16_SAMPLES);
```

```
5 XAdcPs_SetSequencerMode(&XADC_Driver_Instance,
    XADCPS_SEQ_MODE_CONTINPASS);
```

In line 1 we first disable the sequencer to allow for changing of the sequencer options. In line 2 we enable three channels, namely VPVN, AUX1 and AUX8, the three analog input channels exposed on the Zedboard. This function takes as arguments a pointer to the `XAdcPs` instance and a 4 byte number (e.g. `uint32_t`). Each bit in this number corresponds to one channel. If the bit corresponding to channel i is set to 1, then the channel is enabled and will be part of the sampling sequence, if it is zero the channel is disabled. In the file `xadcps_hw.h` there are some useful `#defines` which we use here. Some important correspondences are:

• XADCPS_SEQ_CH_VCCPINT	VCCPINT voltage sensor
• XADCPS_SEQ_CH_VCCPAUX	VCCPAUX voltage sensor
• XADCPS_SEQ_CH_VCCPDRO	VCCPDRO voltage sensor
• XADCPS_SEQ_CH_VCCINT	VCCINT voltage sensor
• XADCPS_SEQ_CH_VCCAUX	VCCAUX voltage sensor
• XADCPS_SEQ_CH_TEMP	die temperature sensor
• XADCPS_SEQ_CH_VPVN	dedicated VPVN analog input channel
• XADCPS_SEQ_CH_AUX00	AUX1 channel
• XADCPS_SEQ_CH_AUX07	AUX8 channel
• XADCPS_SEQ_CH_AUX15	AUX16 channel

There are AUX channels 1 to 16, but on the Zedboard most of these are only accessible through an external multiplexer.
In line 3 we set the input mode for each channel. This function takes the same arguments as the one on line 2, but now each channel that has a 1 bit is set to bipolar input mode, while the ones with 0s are set to unipolar mode. Usually I used the same input mode on all channels.
Line 4 sets up the channel averaging, which is applied to all channels (same averaging for all). The `XADC` can either convert a channel and write the result directly to the corresponding register, or it can do $N \in \{0, 16, 64, 256\}$ conversions, average them and only then write the averaged value to the register. Note that in a sequence, each channel will still be converted once. However, the sequence is only finished (i.e. `EOS` goes high) and results are only written to the registers once there have been N conversions on each channel. The signal `EOC` will still transition high after each conversion. The values passes to achieve a certain averaging are (as `#defined` in xadcps.c):

- XADCPS_AVG_0_SAMPLES
- XADCPS_AVG_16_SAMPLES
- XADCPS_AVG_64_SAMPLES
- XADCPS_AVG_256_SAMPLES

Remember that all these settings can also be set via the FPGA programming. In line 5 we finally restart the sequencer, and we choose to set it to continuous mode.

3.4 Getting converted values

In order to obtain converted values for further processing in software, one must first turn on the sequencer, e.g. in continuous mode, and then read out the register corresponding to the channel of interest. Reading out status registers is done in the following fashion:

```
1 u16 volt_raw = XAdcPs_GetAdcData(&XADC_Driver_Instance,
     XADCPS_CH_VPVN);
2 float volt_f = XAdcPs_RawToVoltage(volt_raw);
3
4 u16 temp_raw = XAdcPs_GetAdcData(&XADC_Driver_Instance,
     XADCPS_CH_TEMP);
5 float temp_f = XAdcPs_RawToTemperature(temp_raw);
```

Here lines 1 and 4 read the status registers of VPVN and the internal temperature sensor. The `#defines` for certain important channels are

• XADCPS_CH_TEMP	temperature sensor
• XADCPS_CH_VPVN	VPVN
• XADCPS_CH_AUX_MIN	1st AUX channel
• XADCPS_CH_AUX_MAX	16th AUX channel

and more are `#defined` in `xadcps.h`. Note that the function `XAdcPs_GetAdcData(XAdcPs*, u8)` returns a 12-bit number contained in a `u16` (i.e. in the range 0x000 - 0xFFF in unipolar mode, and two's complement for bipolar mode, do cast to `s16`). To get a physical value, one can either measure the input-output relation oneself and implement a conversion, or one uses the provided macros as shown on lines 2 and 5. One should be careful though; at least the voltage conversion macro does not do its job well, as described in section 2.3.

If the value read out is constant over time, then probably the `XADC` isn't running the sequencer or the read out channel isn't part of the sequence. Another reason can also be that the `XADC` is in event mode instead of continuous mode.

3.5 Single channel mode

If only one channel should be monitored, then single channel mode is a good alternative to the sequence mode of operation. To use it one first needs to set the channel sequencer to single channel mode and then apply some settings:

```
1 XAdcPs_SetSequencerMode(&XADC_Driver_Instance, XADCPS_SEQ_MODE_SING_CHAN);
2 XAdcPs_SetSingleChParams(&XADC_Driver_Instance, XADCPS_CH_VPVN, FALSE, FALSE, FALSE);
```

In line 1 the sequencer is turned off, and in line two settings are applied. We set VPVN as the single channel to monitor, we do not want to increase the number of acquisition cycles (number of ticks the ADC waits for the capacitors to settle), we don't want event mode and we don't want bipolar input mode. From this point on one can acquire converted values from the status register just as in sequence mode. Switching channels in single channel mode can be done quite a bit quicker than when using the sequencer with a single channel, because in single channel mode one doesn't have to turn off and restart the sequencer in order to do so. This saves several microseconds.
I have not been able to figure out how to use an external multiplexer in single channel mode; the driver suggests that this is possible, but conflicting information is provided on how to do so.

3.6 Event Mode

Up until now the discussion was limited to continuous sampling. The `XADC` has an event mode, in which a conversion on the next channel in the sequence is triggered by an external signal. This is the way to go if time synchronization is crucial. In event mode, the `XADC` listens for rising edges on the `convst_in` channel which trigger a conversion. In order to get the timing right (signal acquisition and the like), one should consult the manual [1]. Apart from these operational differences, everything works exactly the same as in continuous mode.
An example setup of event mode using the channel sequencer to monitor channel VPVN is shown below:

```
#define CONVST 54
#include "sleep.h"

//set up the sequencer
XAdcPs_SetSequencerMode(&XADC_Driver_Instance, XADCPS_SEQ_MODE_SAFE);
XAdcPs_SetSeqChEnables(&XADC_Driver_Instance, XADCPS_SEQ_CH_VPVN);
XAdcPs_SetSequencerEvent(&XADC_Driver_Instance, TRUE);

```

```
 9 //set up the GPIO system
10 XGpioPs Gpio;
11 int Status_GPIO;
12 XGpioPs_Config* GPIOConfigPtr;
13 Statis_GPIO = XGpioPs_CfgInitialize(&Gpio, GPIOConfigPtr,
       GPIOConfigPtr ->BaseAddr);
14 if(Status_GPIO != XST_SUCCESS)
15 {
16         return XST_FAILURE;
17 }
18 XGpioPs_SetDirectionPin(&Gpio, CONVST, 1);
19 XGpioPs_SetOutputEnablePin(&Gpio, CONVST, 1);
20
21 //turn on the sequencer
22 XAdcPs_SetSequencerMode(&XADC_Driver_Instance,
       XADCPS_SEQ_MODE_CONTINPASS);
23
24 //trigger a conversion
25 XGpioPs_WritePin(&Gpio, CONVST, 0x1);
26 XGpioPs_WritePin(&Gpio, CONVST, 0x0);
27
28 //give conversion time to finish, wait 1us
29 usleep(1);
30
31 //fetch result
32 float result = XAdcPs_RawToVoltage(XAdcPs_GetAdcData(&
       XADC_Driver_Instance, XADCPS_CH_VPVN);
```

In this example events are triggered by the PS via GPIO, hence the need to set up the GPIO system. The pin connected to `convst_in` is number 54. In line 7 the sequencer is set to event mode. In single channel mode this would have to be done via the corresponding setup function. We then pulse the `CONVST` GPIO on lines 25, 26 and subsequently need to wait a little until we can confidently fetch the newly written result.
There is a better way to do this, namely interrupts. This will be treated in the next section.

3.7 Interrupts

Waiting until the conversion is finished is not very elegant. Firstly it blocks the CPU, secondly we can't really be sure how long we have to wait. A nice way out is provided by interrupts.
In general, interrupts are used when a certain state requires immediate action, be it for synchronization purposes or for error handling. As the `XADC` pulses high on the `EOC` port when a conversion result has been written to the corresponding register, we can use this

signal to trigger an interrupt in the PS. The handler of this interrupt can then fetch the new data and, for example, add it to an array containing converted values.
Let us now have a brief overview of how the interrupts in the Zynq-7000, especially PL→PS ones (i.e. the programmable logic interrupts the processing system), work.
In the following it is assumed that the FPGA programming includes a connection from the `XADC` `eoc_out` port to the PS `IRQ_F2P`$[0 : 0]$ port. One can identify the ID for interrupts occurring on any IRQ_F2P bit by checking the GIC options in the PS re-customization menu (Interrupt Port → Fabric Interrupts → PL-PS Interrupt Ports → `IRQ_F2P`$[0 : 15]$ → ID). In my case, the entry is $[91 : 84]$,$[68 : 61]$ such that bit 0 is on port 61.
There are different types of PL→PS interrupts:

- `IRQ_F2P`: 16-bit shared interrupt port. Both CPUs are interrupted.
- `Core0_nFIQ`: 1-bit fast private interrupt to CPU0.
- `Core0_nIRQ`: 1-bit slow private interrupt to CPU0.
- `Core1_nFIQ`: 1-bit fast private interrupt to CPU1.
- `Core1_nIRQ`: 1-bit slow private interrupt to CPU1.

We consider the first case, shared PL→PS interrupts.
The mechanics of interrupts (at least for PL→PS ones) is roughly the following: There is a generic interrupt controller (GIC), which owns a look-up table that contains one function and its arguments for each interrupt port that is registered (turned on). The GIC continuously monitors if any of the registered interrupts has occurred. If it detects an interrupt on a registered port, it interrupts program execution and the GIC interrupt handler calls the interrupt handler it finds in the corresponding look-up table entry. When the interrupt handler returns, the GIC restores the previous context and normal program execution continues, with possible changes done by the interrupt handler.
It is important to note that interrupt handlers in general should return as quickly as possible, as this exceptional state seems to be error-prone. For example, it is not a good idea to write to `std::cout` in an interrupt handler; I tried this once, and when normal execution resumed, totally unrelated variables were changed to seemingly random values.
Interrupts have different priorities. Priorities are 1-31, where lower values correspond to higher priorities. However, nested interrupts are NOT supported by the `xscugic` driver.
Interrupts can be triggered on rising edge, level, or falling edge on the interrupt port.
For more information, one should have a look at the driver (start at `xscugic.h`).
As a first example, let's look at a trivial interrupt handler. The simplest possible interrupt handler has the signature `void IntrHandler(void* baseaddr_p)` with a mystical void pointer argument that seems to correspond to the driver instance of the interrupting device.

In order for the interrupts to work, one needs to initialize the GIC, connect its interrupt handler (which handles all interrupts by calling the corresponding handlers) and then register the custom handler to the correct interrupt port (here it's port 61). The corresponding code is

```
//Our custom (trivial) handler
void IntrHandler(void* baseaddr_p)
{
        return;
}


//we assume that an XADC is instantiated and configured. Hence
    there exists:
XAdcPs XADC_Driver_Instance;

//allocations we need
XScuGic InterruptController; //instance of the GIC driver
XScuGic_Config* GicConfig; //pointer to the GIC config
int Status_Gic; //setup status

//setup GIC
Xil_ExceptionInit();
GicConfig = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID); //
    fetch config
if(GicConfig == nullptr) //check for failure
{
        printf("Config Loading Failed\n");
        return;
}
Status_Gic = XScuGic_CfgInitialize(&InterruptController, GicConfig,
     GicConfig->CpuBaseAddress); //initialize config
if(Status_Gic != XST_SUCCESS) //check for failure
{
        printf("Config Initialization Failed!\n");
        return;
}

//connect GIC interrupt handler which processes all interrupts and
    then calls the correct handlers
XIL_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT, (
    Xil_ExceptionHandler) XScuGic_InterruptHandler, (void*) &
    InterruptController);

//Connect our custom handler. This is done with the XScuGic_Connect
     function. To disconnect handlers, use XScuGic_Disconnect.
//The first arguments are: Pointer to GIC driver, interrupt port we
```

```
target, interrupt handler (cast to Xil_ExceptionHandler), argument of interrupt handler
Status_Gic = XScuGic_Connect(&InterruptController, 61, (Xil_ExceptionHandler) IntrHandler, (void*) &XADC_Driver_Instance);
if(Status_Gic != XST_SUCCESS)
{
        printf("Handler Connection Failed!\n");
        return;
}

//Enable interrupts on port 61
XScuGic_Enable(&InterruptController,61);

//Enable interrupts in ARM
Xil_ExceptionEnable();

//Set the priority and trigger type for interrupts on port 61. I'm not sure how the last two arguments work, but they set the trigger type (rising,level,falling,..) and priority
XScuGic_SetPriorityTriggerType(&InterruptController,61,0xa0,3);

printf("Interrupt setup on port 61 complete. eoc_out will interrupt the PS, and the interrupt will be handled by IntrHandler(void*).\n");
```

Now let's have a look at a more sophisticated interrupt handler. In the following piece of code the difference to the previous example is the implementation of the interrupt handler and how it is connected to the vector table of the GIC. We now create an interrupt handler that checks which channel is enabled and fetches the value contained in the corresponding register. If more arguments are to be used, one can pass as argument some data structure containing them, for example an `std::pair<T*,Q*>`. We assume everything is set up as in the previous example, and only need to make the changes shown below:

```
//interrupt handler that accesses the XADC registers. Still not useful, but educational.
void IntrHandler(XAdcPs* XADC_Driver_Instance)
{
        u32 channel_mask = XAdcPs_GetSeqChEnables(XADC_Driver_Instance);
        u32 channel = 0;
        //only 4 channels are supported
        if(channel_mask == 0x00010000)
                channel = 16;
        else if (channel_mask == 0x00020000)
```

```
                channel = 17;
        else if(channel_mask == 0x00040000)
                channel = 18;
        else if(channel_mask == 0x00080000)
                channel = 19;
        else
                return;

        float asdf = XAdcPs_RawToVoltage(XAdcPs_GetAdcData(
            XADC_Driver_Instance,channel));
}

//The connection must now be changed, as the handler takes a
    different argument
Status_Gic = XScuGic_Connect(&InterruptController, 61, (
    Xil_ExceptionHandler) IntrHandler, (void*) &XADC_Driver_Instance
    );
```

3.8 Putting it all together

A complete example using interrupts to fetch results in event mode (triggered via GPIO) is shown below. The handler puts the new conversion result into a pair and flags it, such that during normal execution the program can identify the data as new and add it to the list of collected conversion results. Once the list is full, the content is printed and the list is overwritten.

```
//standard libraries
#include <cstdio>
#include <cmath>
#include <bitset>
#include <cassert>
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <utility>

#define _USE_MATH_DEFINES

//General driver defines
#include "xparameters.h"
#include "xstatus.h"

//GPIO
#include "xgpiops.h"

```

```
//timing, sleeping
#include "xtime_l.h"
#include "sleep.h"

//interrupts
#include "xscugic.h"
#include "Xil_exception.h"

//XADC device ID
#define XADC_DEVICE_ID XPAR_PS7_XADC_0_DEVICE_ID
//GPIO pin connected to conversion_start (event mode triggering)
#define CONVST 54
//interrupts through GPIO (interruptor: XADC, interruptee: PS)
#define GPIO_DEVICE_ID   XPAR_XGPIOPS_0_DEVICE_ID
#define INTC_DEVICE_ID  XPAR_SCUGIC_SINGLE_DEVICE_ID
#define GPIO_INTERRUPT_ID   XPS_GPIO_INT_ID
#define INTR_PIN 0


//Handler that collects the data and saves it to a vector
//The pair contains XAdcPs* and std::vector<float>*.
//the only supported channel is channel 16 (mux 0)
void InterruptHandler(void* pair)
{
        //don't change the data if it hasn't been fetched yet. might corrupt readout (race condition)!
        //if((*(((std::pair<XAdcPs*,std::vector<float>* >*)pair)->second))[1] > 0)
        //      return;
        float data = XAdcPs_RawToVoltage(XAdcPs_GetAdcData(((std::pair<XAdcPs*,std::vector<float>* >*)pair)->first,16));
        //in first entry of data vector store the actual data
        (*(((std::pair<XAdcPs*,std::vector<float>* >*)pair)->second))[0] = data;
        //the second entry is a flag that communicates that the data is new
        (*(((std::pair<XAdcPs*,std::vector<float>* >*)pair)->second))[1] = 1.;
        return;
}




int main(){

//-------------------------------------
```

```
//Initialization and configuration
//--------------------------------------
//XADC
printf("Initializing XADC...\n");
XAdcPs XADC_Driver_Instance;
XAdcPs_Config* cfg = XAdcPs_LookupConfig(XADC_DEVICE_ID);
if(cfg==NULL)
{
        printf("Config lookup failed\n");
        return XST_FAILURE;
}
XAdcPs_CfgInitialize(&XADC_Driver_Instance, cfg, cfg->BaseAddress);


printf("XADC initialization complete.\n");
printf("Setting up channel input mask...\n");
XAdcPs_SetSequencerMode(&XADC_Driver_Instance, XADCPS_SEQ_MODE_SAFE
    );
XAdcPs_SetSeqInputMode(&XADC_Driver_Instance, 0); //all unipolar
XAdcPs_SetSeqChEnables(&XADC_Driver_Instance, XADCPS_CH_VPVN); //
    only VPVN
printf("Sequencer setup complete\n");

printf("XADC setup complete\n");


//Interrupts
printf("Setting up interrupt system...\n");

//pair
std::vector<float> data (2,0.);
std::pair<XAdcPs*,std::vector<float>* > mypair (&
    XADC_Driver_Instance, &data);

XScuGic InterruptController;
XScuGic_Config* GicConfig;
int Status_Gic;
Xil_ExceptionInit();
GicConfig = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
if(GicConfig == NULL)
{
        printf("GIC config lookup failed!\n");
        return -1;
}
Status_Gic = XScuGic_CfgInitialize(&InterruptController, GicConfig,
     GicConfig->CpuBaseAddress);
if(Status_Gic!=XST_SUCCESS)
```

```
{
        printf("GIC config initialization failed!\n");
        return -1;
}
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT, (
    Xil_ExceptionHandler) XScuGic_InterruptHandler, (void*) &
    InterruptController);
Status_Gic = XScuGic_Connect(&InterruptController, 61, (
    Xil_ExceptionHandler) InterruptHandler, (void*) &mypair);
XScuGic_Enable(&InterruptController, 61);
Xil_ExceptionEnable();
XScuGic_SetPriorityTriggerType(&InterruptController, 61, 0xa0, 3);
if(Status_Gic!=XST_SUCCESS)
{
        printf("Handler connection failed!\n");
        return -1;
}
printf("Interrupt setup complete!\n");



//GPIO for event triggering
printf("Setting up GPIO system...\n");
XGpioPs Gpio;
int Status_GPIO;
XGpioPs_Config *GPIOConfigPtr;
GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
Status_GPIO = XGpioPs_CfgInitialize(&Gpio, GPIOConfigPtr,
    GPIOConfigPtr ->BaseAddr);
if (Status_GPIO != XST_SUCCESS)
{
        return XST_FAILURE;
}
XGpioPs_SetDirectionPin(&Gpio, CONVST, 1);
XGpioPs_SetOutputEnablePin(&Gpio, CONVST, 1);
printf("GPIO system setup complete!\n");
printf("Setup complete.\n");

printf("Enter any key to start the sequencer\n");
char asdf;
std::cin >> asdf;

//--------------------------------------
//This is where the good stuff happens
//--------------------------------------
XAdcPs_SetSequencerMode(&XADC_Driver_Instance,
    XADCPS_SEQ_MODE_CONTINPASS);
```

```

XTime begin, current;
XTime_GetTime(&begin);

std::vector<float> data_storage (1000,0);
std::vector<float> time_storage (1000,0);
unsigned current_index = 0;

//trigger first conversion
XGpioPs_WritePin(&Gpio, CONVST, 0x1);
XGpioPs_WritePin(&Gpio, CONVST, 0x0);

while(true)
{
        //print data if the buffer is full
        if(current_index == data_storage.size())
        {
                for(unsigned i = 0; i < data_storage.size(); ++i)
                        printf("%f   %f\n", time_storage[i],
                           data_storage[i]);
                //start overwriting the buffer from the beginning
                current_index = 0;
                //set flag to "old data"
                (*(mypair.second))[1]=0.;
                //initialize next conversion
                XGpioPs_WritePin(&Gpio, CONVST, 0x1);
                XGpioPs_WritePin(&Gpio, CONVST, 0x0);
        }

        //fetch the data if the interrupt handler has updated it
        if((*(mypair.second))[1]>0.)
        {
                //trigger next conversion
                XGpioPs_WritePin(&Gpio, CONVST, 0x1);
                XGpioPs_WritePin(&Gpio, CONVST, 0x0);
                //set the flag to "old data"
                (*(mypair.second))[1]=0.;
                //get the data
                float data_real = (*(mypair.second))[0];
                //get the time
                XTime_GetTime(&current);
                float time = static_cast<float>(current-begin)/
                   static_cast<float>(COUNTS_PER_SECOND)*1000000.f;
                //store data and time in buffers
                data_storage[current_index] = data_real;
                time_storage[current_index] = time;
                //next element
                ++current_index;
        }
}
}
```

4 Signal pre-conditioning PCB

In order to monitor four channels with the `XADC` on the Zedboard, external multiplexing is a necessity. Hence a circuit containing a differential 4-to-1 multiplexer was designed and prototyped on a breadboard. For this purpose we had to rely on slow components only, as tests with fast components failed. Feedback loops on high-speed amplifiers are very sensitive to capacitance and inductance which standard breadboard wires and components have plenty of. After several design-test iterations, the design was finally put onto a printed circuit board (PCB). The final version of this PCB implements further functionality than just channel switching. The signals are buffered, filtered and amplified. In the following sections this board is discussed in more detail.

4.1 Specifications

The circuit schematic of the board is shown in figure 5. The names of any components in the following are derived from that schematic.

The board contains a header P2 that is a copy of the Zedboard `XADC` header. The BNC connectors P3 thru P6 allow the connection of up to 4 differential signals to be monitored by the `XADC`. In the current configuration, these signals should be restricted to 0-400 mV. Each of these signals is fed into one of the fully differential amplifiers U1 thru U4.

All fully differential amplifiers on the board are of the type THS4521; they have a 145 MHz bandwidth, a 490 V/μs slew rate and can be driven by a 5 V supply. Hence the components are well suited for our purpose, they are a lot faster than necessary and can be driven by the 5 V supply provided on the `XADC` header. Slower components aren't significantly cheaper and the choice at hand means the board is future-proof for the use with faster ADCs.

The first amplifier stage serves a quadruple purpose. It amplifies the signal to the maximum range of the `XADC`, low-pass filters the signal with a 1 MHz cutoff, buffers the signal for further stages and can shift the signal up or down. Note that the latter functionality is not a mere amplification, as modulations on the signal are not amplified in the process, only the levels of the two rails are shifted antisymmetrically around the common mode inherent to the amplifier. This feature facilitates mapping the level of signals violating the $[0,\, 400]$ mV range into the allowed region. More information on its usage is provided in section 4.2. Filter and amplification gain can be adjusted by populating the board with different capacitors and resistors, the shift can be put into effect and controlled by connecting a resistor between pins 1 or 3 and 2 of the headers P8 thru P11.

After the first amplification stage the signals are passed to the input of the analog multiplexer MUX1. The active input channel on the multiplexer is controlled by the P2 header pins DIO18 (LSB) and DIO17 (HSB).

The multiplexer is an ADG709 (dual 4-to-1). This component can be driven by 5 V single supply, takes signals rail-to-rail, has a low resistance of a few Ω between input and output channel that is quite constant as a function of input voltage and has a -3 dB bandwidth of 55 MHz. For the use with 60 MHz ADCs, as the one present on the DDS', one should replace the multiplexer if the speed can't be reduced to around 50 MHz.

The signal exiting the multiplexer is put through another amplification stage U5. Here the signal is filtered again and the common mode voltage applied by the differential amplifiers is reduced to a value acceptable to the `XADC`. To this end the signal is amplified with gain two and subsequently divided by the same factor. Then the signal is passed to the VP/VN pins of the header P2.

The PCB is shown in figure 6. It is a 4 layer PCB with ground and power plane. The ground plane is connected to the AGND pins on the P2 header, the power plane to the AV$5$V pin with a large tantalum capacitor nearby. All components present on the board are protected against supply modulations by bypass capacitors of several different sizes. This is important because the current drawn by one component might change abruptly, which leads to changes to the power supplied to all other components. Capacitors close to the supply pins can then bridge the fluctuation in power. As the modulation can have different frequencies, different capacities should be used. Switching components—e.g. multiplexers—are especially prone to this effect.

4.2 Usage instructions

In order to use the signal pre-conditioning board with the `XADC` one needs to take care of a couple of things. First the FPGA must be programmed correctly. More precisely, the `XADC` must be set to external multiplexer mode, and the two least significant channel control bits (muxaddr_out) should be connected to the `XADC` header pins 18 (LSB) and 17 (HSB).

Once the Zynq-7000 is set up correctly, one connects the `XADC` header on the Zedboard to the header P2 on the signal pre-conditioning board using a ribbon cable. There should be one available within the `XADC` box made by me. Take care that the header is connected the right way around, else components might easily be damaged.

Now apply a well known signal (e.g. slow (1 kHz or so triangle from a DAC) into the circuit and analyse the values that the `XADC` reports. This can now be used to calculate the conversion function that relates `XADC` readings to voltage input into the pre-conditioning board, in the same way demonstrated in section $2.1.3$.

The shifting functionality in the first amplification stage is controlled by connecting resistors between pins 1 or 3 and 2 on the headers P8 thru P11. Simulations in `LTspice` were run to characterize this. The circuit used for the calculations is shown in figure 7. One should note that, for convenience, the simulation circuit uses the LT1994 fully

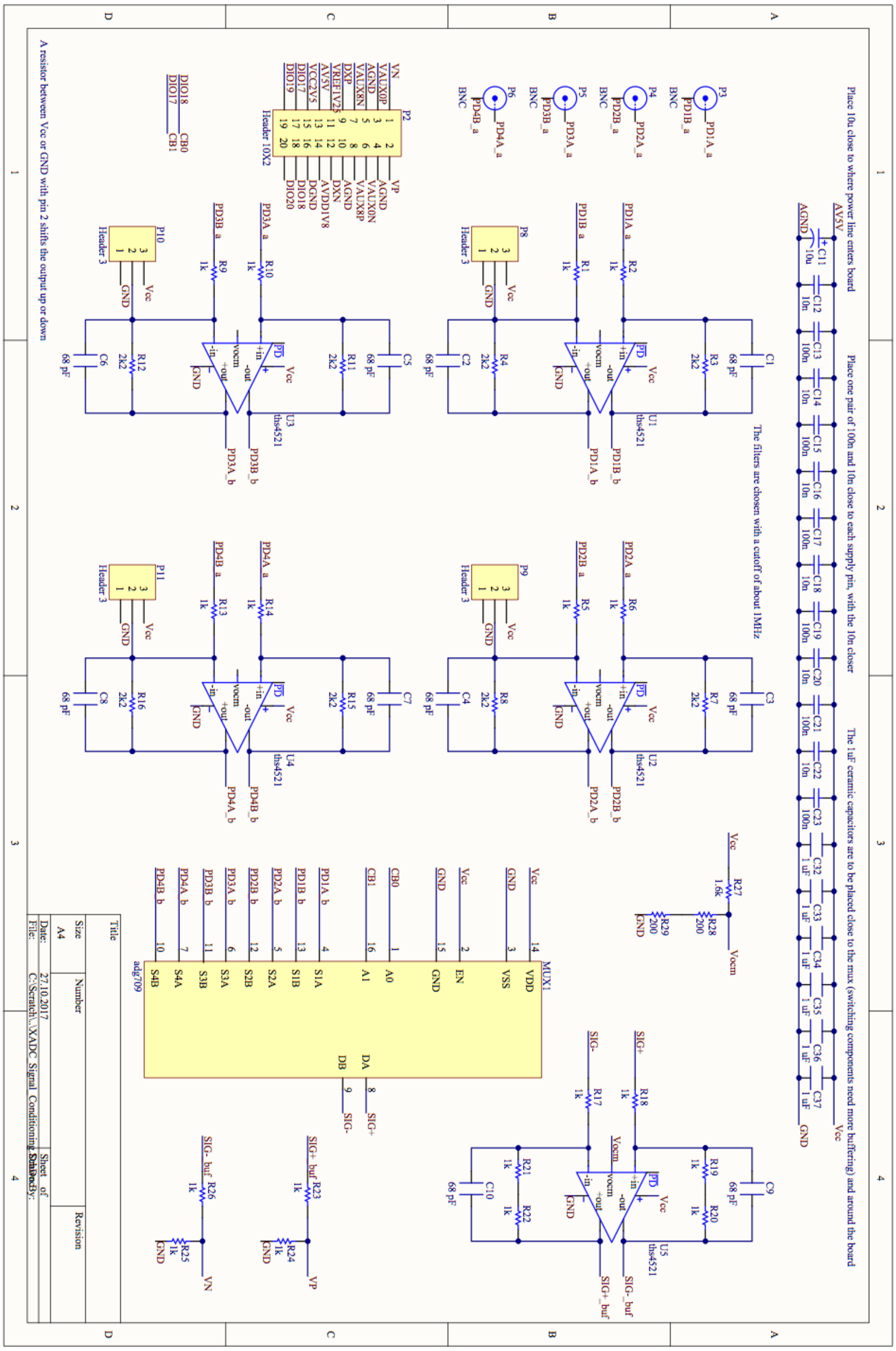

Place 10u close to where power line enters board
Place one pair of 100n and 10n close to each supply pin, with the 10n closer
The 1uF ceramic capacitors are to be placed close to the mux (switching components need more buffering) and around the board
The filters are chosen with a cutoff of about 1MHz
A resistor between Vcc or GND with pin 2 shifts the output up or down
Header 10X2
ths4521
adg709
MUX1
Title
Size
A4
Number
Revision
Date: 27.10.2017
File: C:\Scratch\..\XADC Signal Conditioning.SchDoc
Sheet of
Drawn By:

Figure 5: Circuit schematic of the signal pre-conditioning PCB.

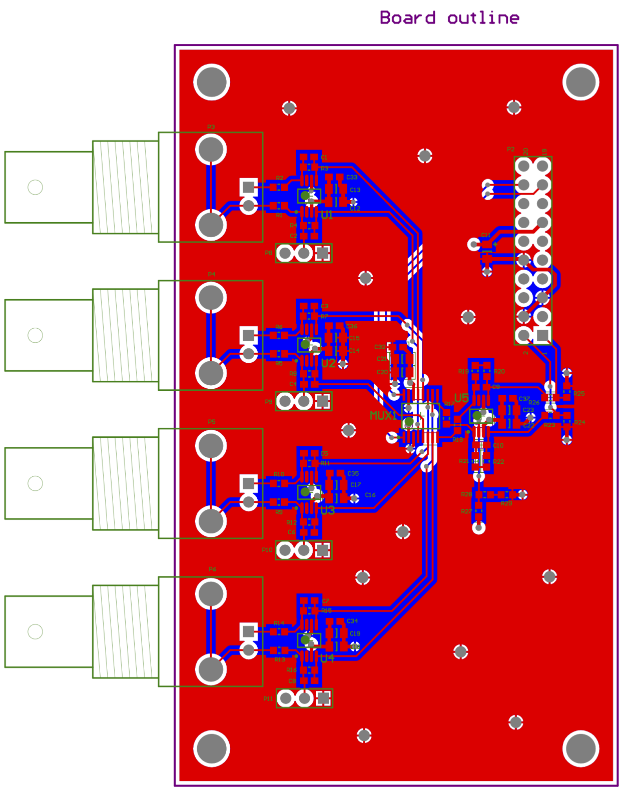


Figure 6: Top layer of the 4-layer signal pre-conditioning PCB. The bottom layer only includes some tracks that could not be fit to the top layer, the two intermediate layers are power and ground plane.

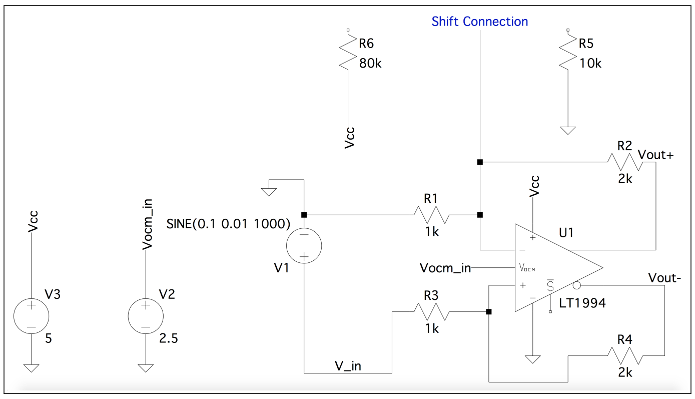


Figure 7: The circuit that was used to run the `LTspice` simulations of the shifting feature. The simulated differential amplifier works quite similarly to those present on the PCB. Connections in the shift-controlling headers on the PCB are simulated by connecting either R5 or R6 to the wire labeled "Shift Connection", or simply leaving it unconnected. The results of the simulation are shown in figure 8.

differential amplifier instead of the one present on the PCB. This should not affect the results, important properties coincide. The differential amplifier is supplied with a sinusodial signal of amplitude 5 mV that is shifted upwards by 100 mV. One should think of the 5 mV modulation as a small change to the signal, for example due to a drift in LASER intensity when monitoring photodiodes, or due to noise. The modulation amplitude is not affected by the shift.

When the wire labeled "Shift Connection" is left unconnected, the amplifier operats with two standard feedback loops that give a gain of 2. Then the output signal (Vout+) $-$ (Vout$-$) is 200 mV, as shown in figure 8 (b). Connecting resistor R5 to "Shift Connection" raises the current demand through resistor R2, the voltage over it must rise and the output signal is shifted to a larger value. Connecting R6 to "Shift Connection" leads to the inverse effect, the power supply delivers current towards the negative amplifier terminal, attenuating the current demand through the feedback loop such that the output voltage is lowered.

One should note that the voltage change on the output is shared symmetrically between the two rails. The voltages Vout$+$ and Vout- are always the same distance apart from the common mode voltage Vocm applied by the amplifier. Consequently the shifter cannot be used to change the common mode voltage. It is useful when signals violating the input constraints of the board should be monitored. If they have differential voltages larger than 400 mV they should be shifted downwards by connecting pins 3 and 2 on the corresponding header with a suitable resistor. If the signal is weak one can shift it up by connecting pins 1 and 2.

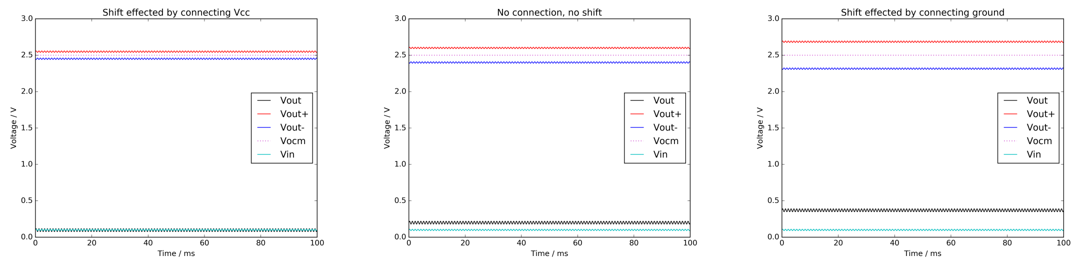
4.3 Measured properties

I carried out measurements on the pre-conditioning board to characterize its properties. More precisely, I extracted the frequency response in form of a Bode plot and the relation between input voltage and output voltage. To get a general feeling for the board I also looked at its response to signals at different frequencies. Note that during the measurements described in the following, only one signal line was populated on the board.

4.3.1 Transfer function

The Bode plot measured with the Analog Discovery Kit's network analyzer feature is shown in figure 9.

The bandwidth is about 1 MHz, which is what we expect. Even though exclusively high-speed components with bandwidths orders of magnitude larger than 1 MHz are used, the anti-aliasing filters incorporated in the differential amplifiers feedback loops have a cutoff frequency of 1 MHz, which, being the slowest part of the signal line, gives the



(a) Supply voltage connected (b) No connection (c) Ground connected

Figure 8: Results of LTspice simulations of the shifting functionality with the circuit shown in figure 7. The differential input voltage is shown in cyan and constant over the three cases. Voltages on positive and negative output terminals are red and blue, the difference of the two is shown in black. The dotted magenta line marks the common mode voltage around which the amplifier symmetrically places the output voltages. The three figures show the three possible cases in which the line "Shift Connection" can be connected. In figure (a) the supply voltage `Vcc` is connected and the rails are shifted towards the common mode voltage, the differential output is lowered. Figure (c) shows the case where ground is connected and the differential output is enhanced. Figure (b) shows the standard case with no connection.

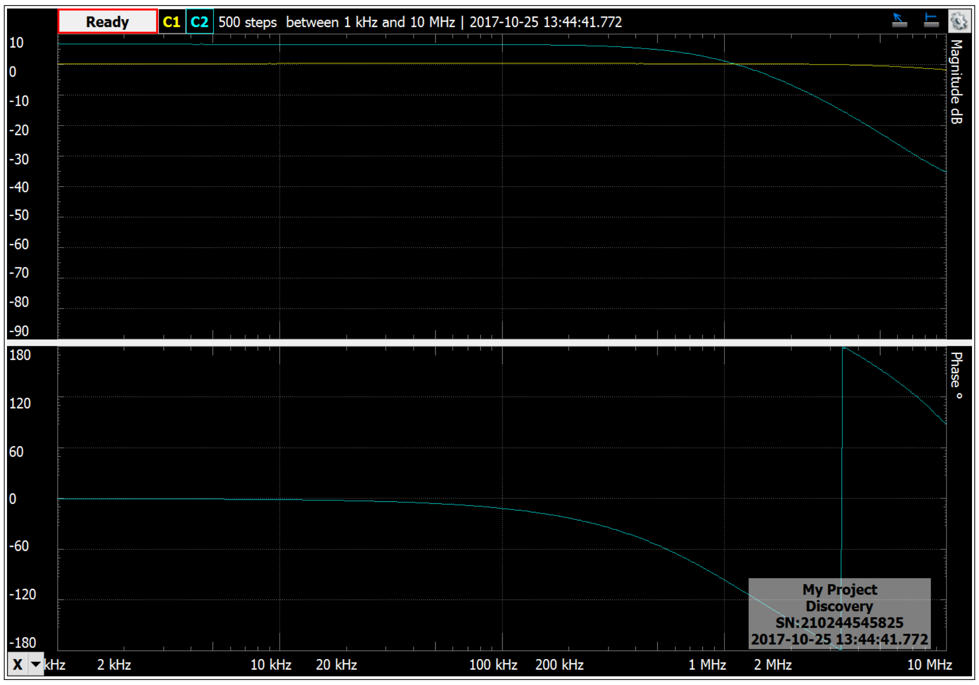


Figure 9: Bode plot of the signal pre-conditioning board. Measurements were done using the network analyzer feature of the Analaog Discovery Kit, which feeds a signal into an input BNC of the board while monitoring that input and the output at the VP/VN pins of header P2 simultaneously. In the top graph the blue curve shows the amplitude of the output signal with respect to the yellow input signal. They y-axis is in dB. As expected by construction, the 3 dB point is close to 1 MHz. The bottom graph shows the phase shift of the output signal with respect to the input, with the y-axis in degrees.

overall frequency restriction of the circuit.

4.3.2 Conversion function

For the board to be useful, one must know the relation between the signal on the output side, the VP/VN pins on header P2, and the signal on the input side, the BNC connectors. To this end, in the spirit of section $2.1.3$, I fed a slow (5 kHz) triangle signal synthesized by the Analog Discovery Kits DAC into the circuit and monitored the output. Figure 10 shows the input voltage in blue and the output voltage in yellow, as measured with the Analog Discovery Kits ADC. I then used the $100\ \mu$s long ramp seen in figure 10 to calculate the conversion from output voltage to input voltage. The result is shown in figure 11.

This measurement suggests a conversion according to

$$V_{\text{in}}(V_{\text{out}}) = 0.4717\, V_{\text{out}} - 0.0397,$$

consistent with our expectation by construction.

4.3.3 Step function response

The response of the board to step functions at different frequencies was analysed. The results are shown in figure 12.

One should keep in mind that a step signal with frequency ν means edges (rising or falling) occur at a frequency of $2\,\nu$, as any period contains one step up and one step down.

The way these measurements were carried out is not realistic, in the sense that the switching between high and low is not effected by the multiplexer switching channels, but rather by the DAC itself. As the multiplexer is a fast component, relative to the frequencies considered here, my analysis highlights a worst-case scenario. When the input signal switches, it is passed through the anti-aliasing filter of the first amplification stage as well as the one in the stage after the multiplexer. Were the switching done by the multiplexer, only the second stage would filter the change. This shouldn't change much, but it's important to be aware of the limitations when interpreting the results.

The important difference to reality that will lead to better behaviour of the circuit in most real-world applications is the switching amplitude. In the tests carried out here, each switch travels all the way from maximum input voltage to minimum input voltage, or vice versa. In real applications, when the multiplexer switches channels, the signal will typically not change by that much and the circuit will perform better than highlighted here.

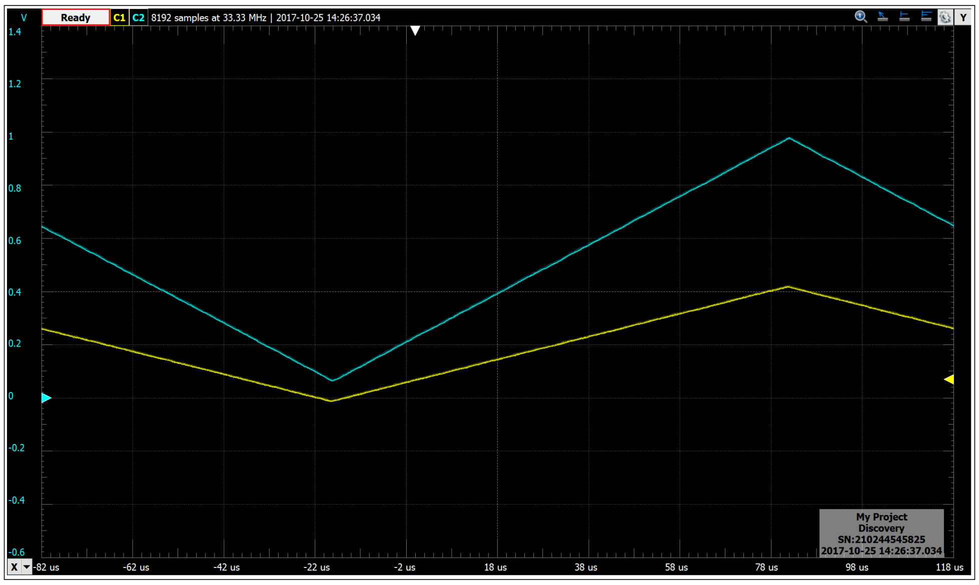


Figure 10: Response of the board to a 5 kHz triangle signal. The input voltage is shown in blue, the output voltage in yellow. The ramp between -18 μs and 82 μs can be used to calculate the conversion function between output and input voltage. The result of this procedure is shown in figure 11.

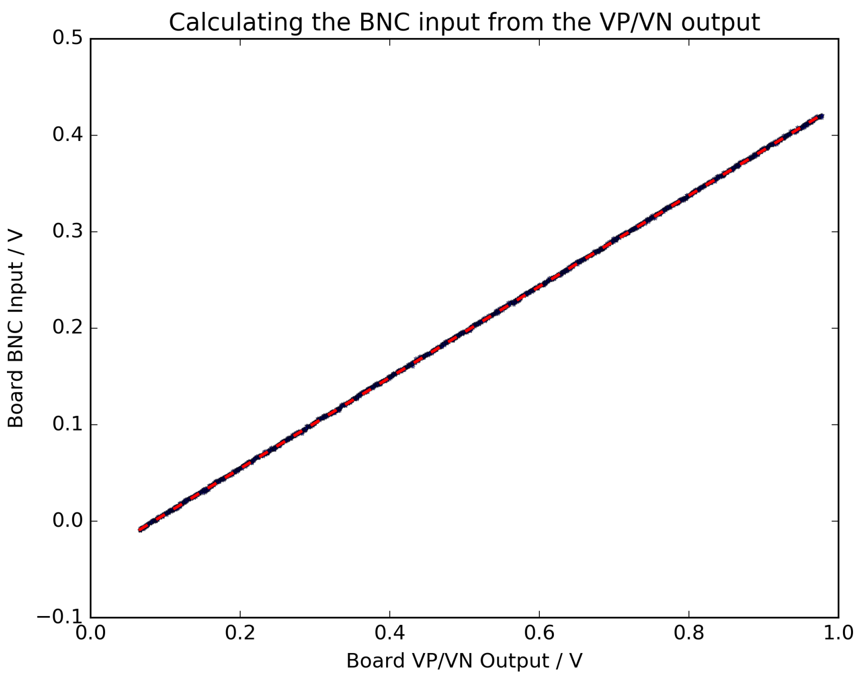
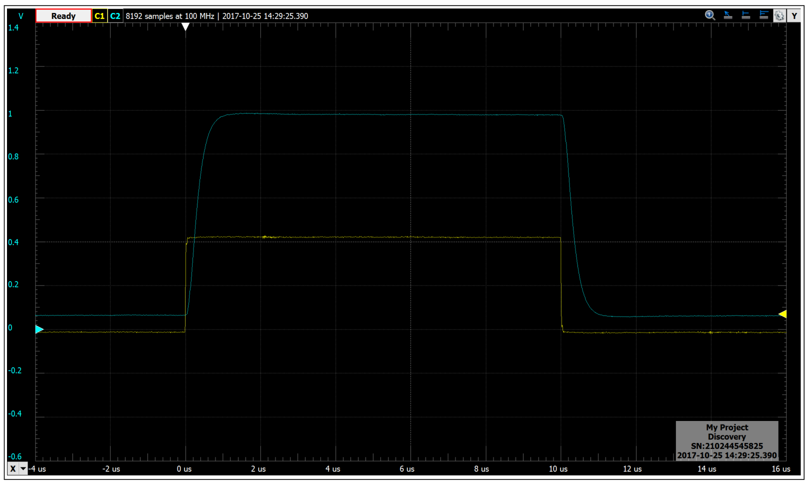
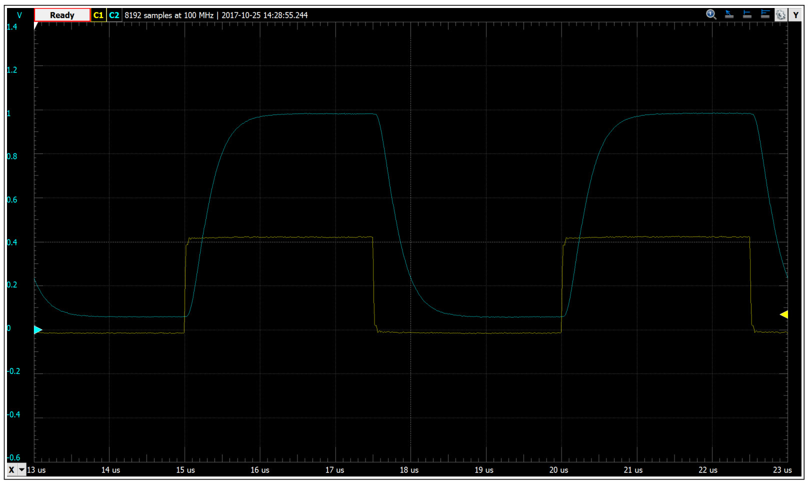


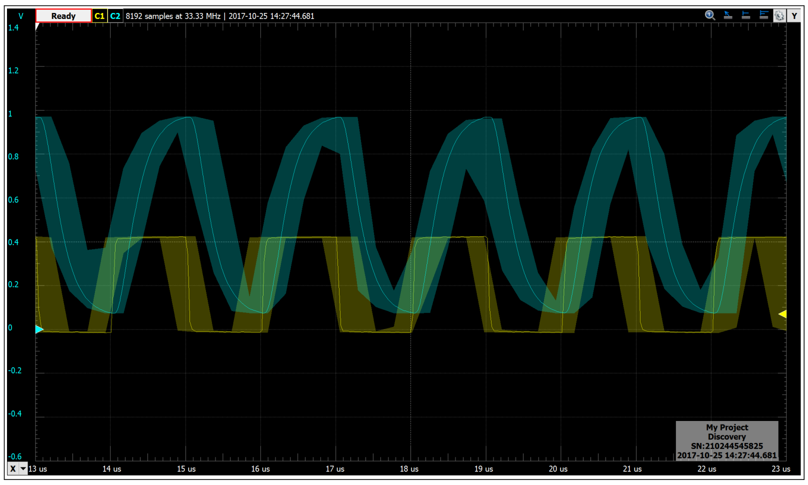
Figure 11: Calculation of board input (BNC) from board output (VP/VN). The plot shows the output voltage of the board on the x-axis and the input voltage on the y-axis. Dark points show data measured with the Analog Discovery Kit, the dashed red line is a least squares fit following the equation $y = 0.4717\,x - 0.0397$.



(a) 50 kHz



(b) 200 kHz



(c) 500 kHz

Figure 12: Response of the board to step functions at different frequencies. The input voltage is shown in yellow, the board output is shown in blue. The 200 kHz signal seems to be the cutoff at which the XADC can be supplied usefully, with the 500 kHz step not saturating at all.

The result of the worst-case analysis is that the multiplexer should not be operated at frequencies closer to 1 MHz than 500 kHz. Switching at 1 MHz can lead to wrong results as the circuit cannot saturate the signal in time. When switching at 400 kHz, one should only trigger conversions in the `XADC` more than 1 μs after a channel has been activated. This still leaves enough time for the `XADC` to convert.
The application described in section $2.1.3$ is not affected by the results of this test. A channel is active for 20 μs, which corresponds to operating the multiplexer at 50 kHz. As shown in figure 12 (a), this is easily possible, especially considering that this figure corresponds to the multiplexer switching at 100 kHz. Problems only become apparent when the sequencer is operated in non-trivial fashion. Then either the FPGA programming should be adjusted such that the `XADC`'s speed is choked, or the filters in both amplification stages should be modified to have a higher cutoff.

References

[1] 7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter (ug480)

[2] Zedboard Hardware User's Guide