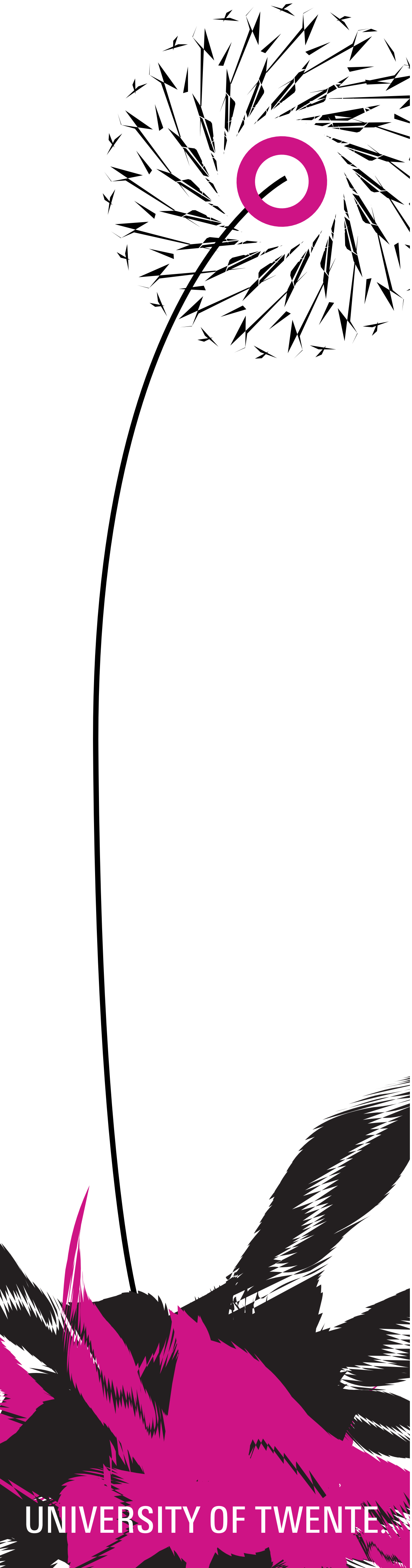

AMR CORRECTION IN AN FPGA.
BACHELOR ASSIGNMENT.

Silke Hofstra
s1074202

SUPERVISORS:
Andreina Zambrano & Hans G. Kerkhoff

FEBRUARY 1, 2016

CONTENTS

INTRODUCTION

In current automotive technology, anisotropic magnetoresistance (AMR) sensors are often used to detect angles. One of the most popularly used sensors is the KMA200**kma200** sensor made by NXP.

A common problem with these sensors is the existence of a voltage offset. This can be compensated for, but the usual methods do not allow this while the sensor is being used. **zambrano** developed a method for online compensation of this offset.

The architecture of an AMR-sensor can be seen as two Wheatstone bridges (figure 2.1). The angle can then be calculated using the two output voltages. In reality this includes an offset.

Because of this a method was devised to compensate for this offset**zambrano**. This is done using three measurements of different angles, out of which the voltage offset can be calculated.

1.1 Requirements

The performance requirements are:

- Accuracy improvement of 90% (factor of 10)
- Uses an ADC of less than 16 bits.
- Can do at least 10 calculations per second.

THEORY

MAGNETIC field sensors form the basis of AMR-sensors.

2.1 Workings of an MR sensor

An AMR sensor is commonly designed by creating two Wheatstone bridges. Because of magnetoresistance, the resistance in the Wheatstone bridges changes when exposed to a magnetic field in a certain direction. The consequence of this is a potential difference between either side of the Wheatstone bridge. This voltage is dependent on the angle. The second Wheatstone bridge is oriented with a rotation of 45° compared to the first Wheatstone bridge. This layout is shown in figure 2.1. The result of this design is that the output voltages are two sinusoids with a difference of 90°.

The angle can then be calculated as follows:

$$\alpha = \frac{1}{2} \arctan \left(\frac{V_{out_1}}{V_{out_2}} \right)$$

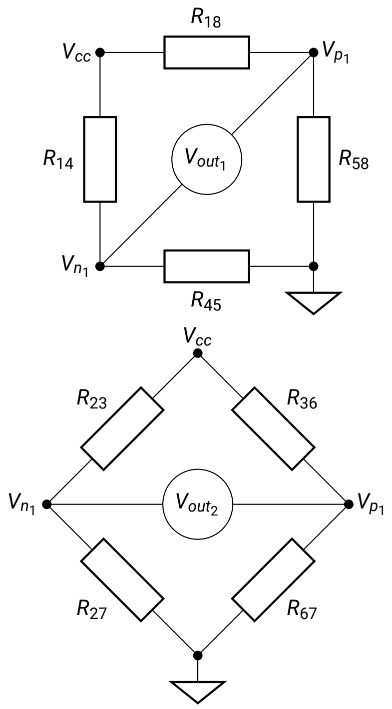

Figure 2.1: Schematic layout of an AMR-sensor

2.2 Correcting an offset

In their paper, **zambrano** propose the following equation system to compensate for the offset of the AMR-sensor:

$$AMR_1 \pm (off_{p_1} - off_{n_1}) = V_{out_1,\alpha_1} \tag{2.1}$$
$$AMR_1 \pm (off_{p_2} - off_{n_2}) = V_{out_2,\alpha_1} \tag{2.2}$$

$$offp_1 + offn_1 = V_{R45} - V_{R18} \tag{2.3}$$
$$offp_2 + offn_2 = V_{R27} - V_{R36} \tag{2.4}$$

$$\begin{aligned} & 2 \cdot (off_{p_1} - off_{n_1}) \cdot (V_{out_1,\alpha_2} - V_{out_1,\alpha_1}) \\ & +2 \cdot (off_{p_2} - off_{n_2}) \cdot (V_{out_2,\alpha_2} - V_{out_2,\alpha_1}) \\ & = V^2_{out_1,\alpha_2} + V^2_{out_2,\alpha_2} - V^2_{out_1,\alpha_1} - V^2_{out_2,\alpha_1} \end{aligned} \tag{2.5}$$

$$\begin{aligned} & 2 \cdot (off_{p_1} - off_{n_1}) \cdot (V_{out_1,\alpha_3} - V_{out_1,\alpha_1}) \\ & +2 \cdot (off_{p_2} - off_{n_2}) \cdot (V_{out_2,\alpha_1} - V_{out_2,\alpha_1}) \\ & = V^2_{out_1,\alpha_3} + V^2_{out_2,\alpha_3} - V^2_{out_1,\alpha_1} - V^2_{out_2,\alpha_1} \end{aligned} \tag{2.6}$$

METHOD

WITH the requirements given in section 1.1, a basic design can be formulated (figure 3.2). The system can be broken into four pieces, three of which are integrated in the design.

- Magnetic sensor: a magnetic sensor. For analysis of the system as a whole, the properties of the KMZ41 magnetic field sensor will be used.
- ADC: converts the analog signals from the magnetic sensor to digital signals for processing
- Digital processor: an embedded solution that calculates the offset and corrects the received values.
- Digital interface: an interface from which the corrected voltages and the offsets can be retrieved. This interface could also provide additional information like its accuracy.

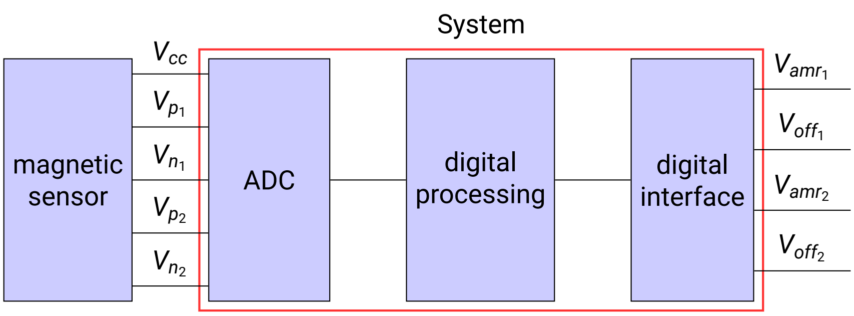

Figure 3.2: Design of the system

The design itself consists of a couple of steps that have to be taken:

1. Mathematics
   (a) Solution to the equation system.
   (b) Analysis of the ADC.
2. Simulation
   (a) Testing of the processing.
   (b) Testing of the processing with an ADC.
3. Implementation
   (a) Implementation in CλaSH/VHDL
   (b) Simulation and design of the embedded flatform

3.1 Mathematics

3.1.1 Solution to the system of equations

The equation system explained in section 2.2 has to be solved in a computationally cheap way, before any implementation can be done.

For the mathematics, it is assumed that V_{p_1}, V_{n_1}, V_{p_2}, V_{n_2} and V_{cc} are known. To get the output voltage, the following relations are required:

$$V_{out_1} = V_{p_1} - V_{n_1} \tag{3.7}$$
$$V_{out_2} = V_{p_2} - V_{n_2} \tag{3.8}$$

To ease the calculation in the system of quadratic equations, a number of shorthand notations are defined.

$$V_{sqr_1} = V^2_{out_1,a_2} + V^2_{out_2,a_2} - V^2_{out_1,a_1} - V^2_{out_2,a_1} \tag{3.9}$$
$$V_{sqr_2} = V^2_{out_1,a_3} + V^2_{out_2,a_3} - V^2_{out_1,a_1} - V^2_{out_2,a_1} \tag{3.10}$$
$$off_1 = off_{p_1} - off_{n_1} \tag{3.11}$$
$$off_2 = off_{p_2} - off_{n_2} \tag{3.12}$$
$$V_{x_1,a_2} = V_{out_1,a_2} - V_{out_1,a_1} \tag{3.13}$$
$$V_{x_1,a_3} = V_{out_1,a_3} - V_{out_1,a_1} \tag{3.14}$$
$$V_{x_2,a_2} = V_{out_2,a_2} - V_{out_2,a_1} \tag{3.15}$$
$$V_{x_2,a_3} = V_{out_2,a_3} - V_{out_2,a_1} \tag{3.16}$$

Solving the system of equations stated in equations 2.5 and 2.6 now results in:

$$off_1 = \frac{V_{sqr_2}V_{x_2,a_2} - V_{sqr_1}V_{x_2,a_3}}{2(V_{x_1,a_3}V_{x_2,a_2} - V_{x_1,a_2}V_{x_2,a_3})} \tag{3.17}$$
$$off_2 = \frac{V_{sqr_1}V_{x_1,a_2} - V_{sqr_2}V_{x_1,a_3}}{2(V_{x_1,a_3}V_{x_2,a_2} - 2V_{x_1,a_2}V_{x_2,a_3})} \tag{3.18}$$

The actual AMR-values can now be calculated:

$$amr_1 = V_{out_1,a_1} - off_1 \tag{3.19}$$
$$amr_2 = V_{out_2,a_1} - off_2 \tag{3.20}$$

The *p* and *n* offsets can also be calculated. But first equations 2.3 and 2.4 have to be rewritten because the voltages of individual resistors are unknown. This can be done by applying Kirchhoff's voltage law in the diagram as shown in figure 2.1.

$$off_{\Sigma 1} = off_{p_1} + off_{n_1} = V_{p_1,a_1} + V_{n_1,a_1} - V_{cc} \tag{3.21}$$
$$off_{\Sigma 2} = off_{p_2} + off_{n_2} = V_{p_1,a_2} + V_{n_1,a_2} - V_{cc} \tag{3.22}$$

Together with equations 3.11 and 3.12 this forms a system of equations. Solving this system results in:

$$off_{p_1} = \frac{1}{2}(off_1 + off_{\Sigma 1}) \tag{3.23}$$
$$off_{p_2} = \frac{1}{2}(off_2 + off_{\Sigma 2}) \tag{3.24}$$
$$off_{n_1} = off_{\Sigma 1} - off_{p_1} \tag{3.25}$$
$$off_{n_2} = off_{\Sigma 2} - off_{p_2} \tag{3.26}$$

3.1.2 Analysis of an ADC

The angle can be calculated with:

$$\alpha = \frac{1}{2}\arctan\left(\frac{V_{out_1}}{V_{out_2}}\right)$$

The output signals are sinusodial with an amplitude of V_{peak}, but after digitalisation by an ADC they can have an offset of 1 LSB at most. This means they can be considered to be:

$$V_{out_1} = V_{peak}\sin(\alpha) + LSB \tag{3.27}$$
$$V_{out_2} = V_{peak}\cos(\alpha) + LSB \tag{3.28}$$
$$\tag{3.29}$$

Because in reality this offset can be considered as white noise, it can not be removed by the correction algorithm without measuring many more angles. This results in an error which can be quantified as:

$$err(\alpha) = \frac{1}{2}\arctan\left(\frac{V_{peak}\sin(2\alpha) + LSB}{V_{peak}\cos(2\alpha) + LSB}\right) - \alpha$$

The maximum value of this function is the maximum error caused by the ADC. This error is dependent on both V_{peak} and LSB. According to the datasheet of the KMZ41**kmz41**, V_{peak} has a value between 73 mV (min) and 89 mV (max). The LSB for n bits can be calculated with:

$$LSB(n) = \frac{V_{cc}}{2^n}$$

The consequences of this relation have been plotted in figure 3.3. The most important values are also shown in table 3.1. Because the KMA200 provides a resolution of 0.05°**kma200**, an ADC of at least 16 bits has to be chosen if the accuracy of the system has to be able to a resolution that is equal or better. However, the actual accuracy of the KMA200 is a lot less, which means that improvements may still be seen with less bits.

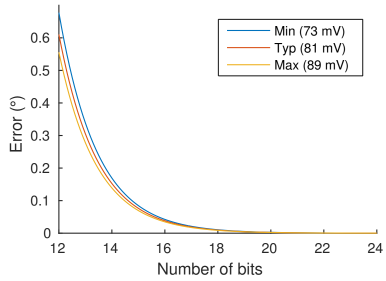

Figure 3.3: Error caused by the ADC for various numbers of bits and for peak voltages of the magnetic sensor

bits	min (°)	typ (°)	max (°)
12	0.6775	0.6106	0.5557
14	0.1694	0.1526	0.1389
16	0.0423	0.0382	0.0347
24	0.0002	0.0001	0.0001

Table 3.1: Error caused by the ADC for various numbers of bits and for peak voltages of the magnetic sensor

3.2 Simulation

For the system to be simulated, a simulation of the AMR needs to be created first. The resulting output voltages can then be fed into the solution to the system of equations created in section 3.1.1. When the behaviour of this system is clear, an ADC can be added.

3.2.1 Simulation of the AMR

To be able to test the algorithm in Matlab, a script, amr_generate (see appendix A.3.1), has been created. This script generates AMR output values for various angles with and without an offset. It also allows the creation of non-ideal results. The following parameters have been used:

Symbol	Parameter	Min	Typ	Max	Unit
Vcc	Supply voltage	–	5.00	–	V
Roff	Resistor value	2992.5	3000.0	3002.5	Ω
Amrr	AMR influence	96.6	99.0	101.4	Ω

Table 3.2: Parameters used for AMR simulation

3.2.2 Implementation of the mathematics

After the generation, the offset calculation can be done. This calculation is implemented in offset_calculation. The code is an implementation of the mathematics discussed in the previous section.

First the output (V_{out_n}) and summed offset voltages ($off_{\Sigma n}$) are calculated:

```
Vout = Vp - Vn;
```

```
offS = Vn(:,1) + Vp(:,1) - Vcc;
```

After this, the squared voltages (V_{sqr_n}) and the voltage pairs (V_x) are calculated:

```
Vsqr = [ Vout(1,2)^2 + Vout(2,2)^2 - Vout(1,1)^2 - Vout(2,1)^2 ;...
         Vout(1,3)^2 + Vout(2,3)^2 - Vout(1,1)^2 - Vout(2,1)^2 ];
```

```
Vx = [[ Vout(1,2)-Vout(1,1), Vout(1,3)-Vout(1,1) ];
      [ Vout(2,2)-Vout(2,1), Vout(2,3)-Vout(2,1) ]];
```

The denominator of both fractions can now be calculated:

```
denom = 2*(Vx(1,2)*Vx(2,1) - Vx(1,1)*Vx(2,2));
```

After which the offsets (off_n) can be calculated:

```
offs(1) = (Vx(2,1)*Vsqr(2) - Vx(2,2)*Vsqr(1))/denom;
offs(2) = (Vx(1,2)*Vsqr(1) - Vx(1,1)*Vsqr(2))/denom;
```

From this, the corrected AMR values can be calculated:

```
amr = Vout(:,1) - offs;
```

But before the corrected AMR values are calculated, the result is validated:

```
if abs(mean(offs)) == Inf | isnan(mean(offs))
    offs = zeros(2,1);
end
```

Invalid values result in an offset of zero, ensuring no extra error is introduced.

3.2.3 Testing of the algorithm without ADC

The implementation is tested for four scenarios: (i) without non-idealities, (ii) with non-ideal AMR values, (iii) with non-ideal resistors and (iv) with non-ideal resistors and AMR values. When using non-ideal values, random offsets have been generated one thousand times and the average is shown. An example of one such situation is shown in figure 3.4. The results can be found in table 3.4.

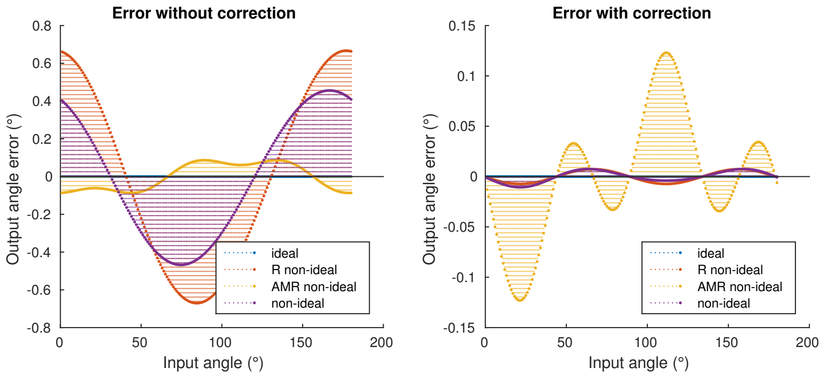

Figure 3.4: Example of the resulting error before and after correction by the algorithm

Table 3.3: Offset errors before and after correction in several scenarios

Ideal		Average offset (μV)		
R	AMR	Before	After	Improvement
yes	yes	0	1.430×10^{-24}	0
yes	no	518.8	2.122	244.5
no	yes	3.310×10^{3}	143.8×10^{-12}	23.02×10^{12}
no	no	3.356×10^{3}	2.181	1.539×10^{3}

Ideal		Mean ($\times 10^{-3}$ °)			Max ($\times 10^{-3}$ °)		
R	AMR	Before	After	Improvement	Before	After	Improvement
yes	yes	0	7.413×10^{-9}	0	0	18.46×10^{-9}	0
yes	no	82.90	824.5×10^{-3}	100.6	129.5	2.122	61.02
no	yes	582.0	3.984	146.1	918.4	6.257	146.8
no	no	591.3	4.107	144.0	933.0	8.439	110.6

Table 3.4: Angular error before and after correction in several scenarios

It is clear that in all scenarios the average error is improved or basically the same. The maximum error, however, is higher in the case of the random variations of the AMR effect.

3.2.4 Implementation with an ADC

For testing the influence of an ADC, it has to be simulated in Matlab as well. The principle for *n* bits is simple: first the fraction between a reference voltage (usually equal to V_{cc}) is calculated. This is then converted into a binary fixed point number with the point before the most significant byte. This result is then multiplied with the reference voltage to get the final result. In Matlab this looks like:

```
% Cast to n bit unsigned fixed point
Vref = Vcc;
voutp = fi(voutp./Vref,0,adc_bit,adc_bit);
voutn = fi(voutn./Vref,0,adc_bit,adc_bit);

% Cast to floating points
voutp = single(voutp).*Vref;
voutn = single(voutn).*Vref;
```

The result of the rounding of bits is that errors can be introduced because values are not sufficiently different. Simply adding the code above results in many errors, which can be seen in figure 3.5.

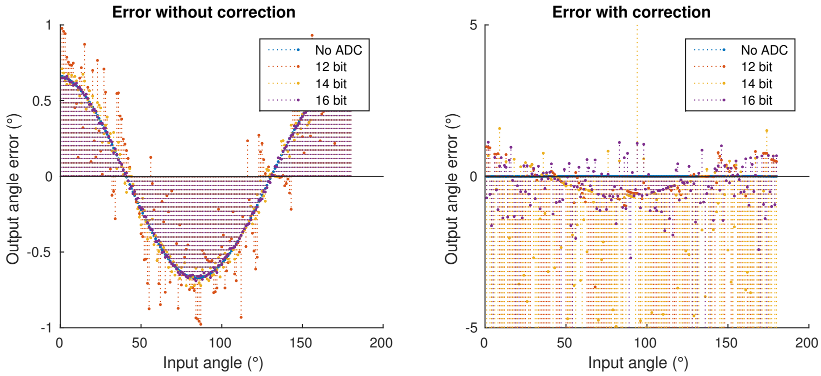

Figure 3.5: Example of the resulting error before and after correction by the algorithm when adding an ADC

To mitigate this, some filtering is added. The filtering exists of two steps:

1. The calculation is only done after the output values have changed significantly. This means the difference is larger then a value V_b.
2. Calculated offset values are passed through a discrete low-pass filter. Mathematically, this works as follows:

$$off_k = \frac{off_{k-1} \cdot (weight - 1) + off_{calc}}{weight}$$

   Where off_{calc} is the value received from the calculation of the offset with V_{out}.

Both V_b and *weight* can be configured to tune the filter and increase accuracy.

The decision making process for every newly measured value is now a lot more complicated, therefore a flow chart representing the process is shown in figure 3.6. This process has been implemented in MATLAB, where optimisation of the parameters can be done.

The optimisation process consists of sweeping the parameters over a wide range of values. For V_b, this is done between 0 and V_{peak} (or about 100 mV). The ideal value is dependent on several factors, mainly the number of bits of the ADC. The results of the sweeps in MATLAB (figure 3.7) clearly show this: without an ADC, a V_b of 0 is good enough, but with an ADC it improves results severely. When V_b approaches V_{peak}, the error increases gradually until it is equal to the error of the original signal.

The optimisation of the weight of the filter is done in the same way (figure 3.8). A consideration that should be taken into account is that it is important for the filter to still be fast. Otherwise a large value should be better. In MATLAB the simulation is done for a limited number of angles so that this is reflected.

Through this optimisation process, the optimal parameters can be obtained. They have been displayed in table 3.5.

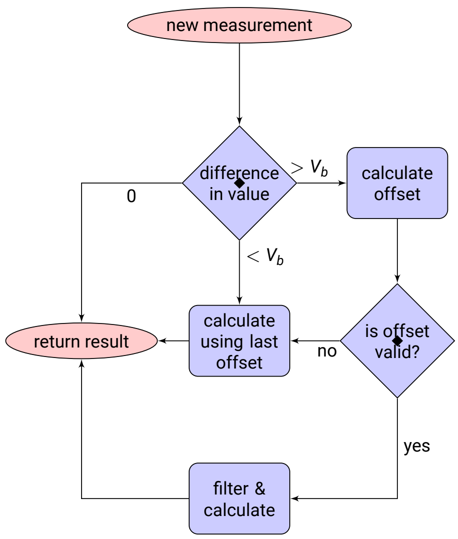

Figure 3.6: Flow chart of the actions for every measured value

ADC	V_b (mV)	*weight*
none	23.0	3.7
16 bit	40.5	4.6
14 bit	48.5	5.7
12 bit	62.5	8.1

Table 3.5: Optimal values of the filter parameters for different ADCs

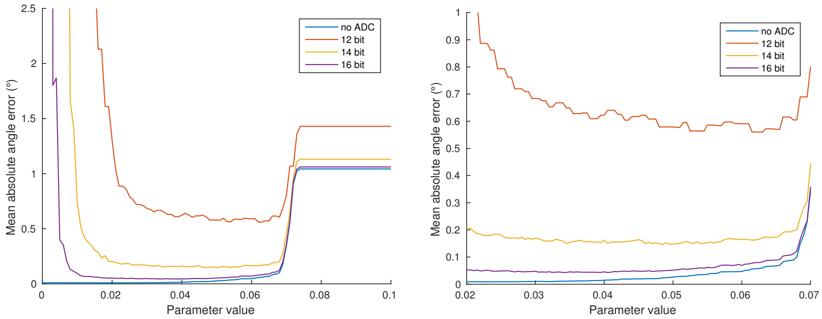

Figure 3.7: Angular error for different values of V_b

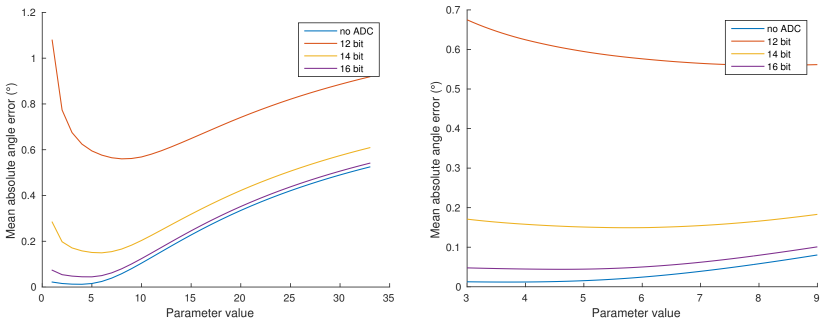

Figure 3.8: Angular error for different values of *weight*

3.2.5 Simulation with an ADC

This implementation is again tested for four scenarios: (i) without non-idealities, (ii) with non-ideal AMR values, (iii) with non-ideal resistors and (iv) with non-ideal resistors and AMR values. When using non-ideal values, random offsets have been generated one thousand times and the average is shown. An example of one such situation is shown in figure 3.9.

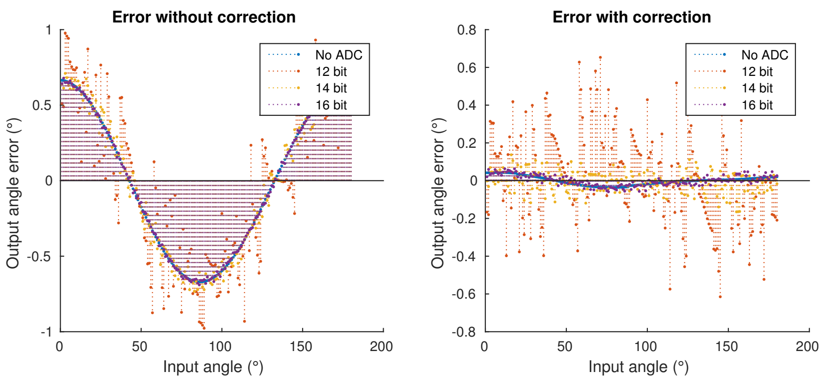

Figure 3.9: Example of the resulting error before and after correction by the algorithm

In table 3.7, the statistics are shown for all situations and with several configurations of the ADC. In this table, the optimised values from table 3.5 have been used.

From the results it is clear that the only number of bits that reliably results in less than 0.06° error is 16. It can also be seen that, in ideal conditions, the performance of the system is less than in the most non-ideal situation. The relative improvement is, of course, correlated with the number of bits. This means that the largest improvement can be seen with the highest number of bits. From this simulation and the analysis done in section 3.1.2 a 16 bit ADC is recommended.

Ideal R	Ideal AMR	Bits	Average offset (μV) Before	After	Improvement
yes	yes	none	0	135×10^{-12}	0
		16	404×10^{-9}	11.1	36.3×10^{-9}
		14	1.41×10^{-6}	174	8.14×10^{-9}
		12	8.73×10^{-6}	662	13.2×10^{-9}
yes	no	none	519	2.38	218
		16	519	13.4	38.8
		14	519	56.0	9.28
		12	587	314	1.87
no	yes	none	3.31×10^{3}	119×10^{-3}	27.7×10^{3}
		16	3.31×10^{3}	12.3	270
		14	3.31×10^{3}	53.6	61.8
		12	3.30×10^{3}	260	12.7
no	no	none	3.36×10^{3}	2.45	1.37×10^{3}
		16	3.36×10^{3}	13.3	252
		14	3.36×10^{3}	57.6	58.3
		12	3.35×10^{3}	241	13.9

Table 3.6: Offset error before and after correction in several scenarios

Ideal R	Ideal AMR	Bits	Mean ($\times 10^{-3}$ °) Before	After	Improvement	Max ($\times 10^{-3}$ °) Before	After	Improvement
yes	yes	none	0	119×10^{-12}	0	0	384×10^{-12}	0
		16	15.5	17.9	863×10^{-3}	33.3	46.8	712×10^{-3}
		14	59.3	77.9	762×10^{-3}	112	217	514×10^{-3}
		12	232	254	913×10^{-3}	508	739	688×10^{-3}
yes	no	none	82.9	928×10^{-3}	89.3	130	2.16	60.1
		16	83.7	11.7	7.15	148	39.5	3.75
		14	92.6	42.0	2.21	211	137	1.55
		12	191	174	1.09	513	529	969×10^{-3}
no	yes	none	582	6.07	95.9	918	10.8	85.2
		16	582	13.3	43.7	933	43.5	21.4
		14	583	41.7	14.0	989	134	7.38
		12	598	157	3.80	1.23×10^{3}	481	2.57
no	no	none	591	6.22	95.0	933	12.7	73.7
		16	591	13.4	44.2	948	43.7	21.7
		14	593	41.9	14.1	1.01×10^{3}	138	7.32
		12	608	155	3.91	1.26×10^{3}	491	2.57

Table 3.7: Angular error before and after correction in several scenarios

3.3 Implementation in CλaSH

CλaSH has been chosen for the implementation of the algorithm. The decision to use CλaSH over plain VHDL was made because CλaSH claims to be made for rapid prototyping, which is the main goal of this project.

3.3.1 Mathematics

The simplest part of the implementation in CλaSH are the mathematics themselves. For this purpose, a function—`offCalc`—has been created:

```
offCalc (o11,o12,o13) (o21,o22,o23) = (offs1, offs2)
    where
        -- Calculate squares
        sqr0 = square o11 + square o21
        sqr1 = square o12 + square o22 - sqr0
        sqr2 = square o13 + square o23 - sqr0

        -- Calculate differential pairs
        x1 = o12 - o11
        x2 = o13 - o11
        x3 = o22 - o21
        x4 = o23 - o21

        -- Calculate the denominator
        denom = 2 * (x2 * x3 - x1 * x4)

        -- Calculate offset
        divzero = denom .==. 0
        offs1 = mux divzero 0 ( (x3 * sqr2 - x4 * sqr1)/denom )
        offs2 = mux divzero 0 ( (x2 * sqr1 - x1 * sqr2)/denom )
```

The most important thing is that this function returns 0 when the denominator is 0 and the result would be infinity. This means a result is always usable.

Another function—`reCalc`—makes sure the three latest output values of each AMR are available and changed:

```
reCalc (out1, out2) = (amr1, amr2, off1, off2)
    where
        -- Shift the values
        (o11,o12,o13) = (out1, register 0 o11, register 0 o12)
        (o21,o22,o23) = (out2, register 0 o21, register 0 o22)

        -- Calculate the offset
        (offs1,offs2) = offCalc (o11,o12,o13) (o21,o22,o23)

        -- Filter the offset
        maxOffset  = (max (abs offs1) (abs offs2))
        boundCheck = maxOffset .>. 0 .&&. maxOffset .<. vcc

        off1 = mux boundCheck (offFilter off1 offs1) (register 0
             off1)
        off2 = mux boundCheck (offFilter off2 offs2) (register 0
             off2)

        -- Calculate the correct AMR values
        amr1 = out1 - off1
        amr2 = out2 - off2
```

Depending on the returned values `offCalc`, the offset value is filtered by `offFilter` and updated or not updated at all. From the offset the corrected AMR values are calculated.

These functions are only called if the input signal differs enough. This decision is made in the `amrCalc` function:

```
amrCalc (vp1,vn1,vp2,vn2) = (amr1, amr2, off1, off2)
    where
        -- Calculate output
        out1 = vp1 - vn1
        out2 = vp2 - vn2

        -- Calculate difference for comparison
        diff = max (abs (out1 - o11)) (abs (out2 - o21))

        -- Calculate AMR values
        (amr1, amr2, off1, off2) =
            muxT4 ( diff .>. filter_bound )
                ( reCalc (out1, out2) )
                ( out1 + off1, out2 + off2, register 0 off1,
                    register 0 off2 )
```

The input values are the individual voltages in the bridges. With these inputs the output voltages are calculated and consequently the maximum difference is calculated. This is done by taking the maximum of the absolute difference between the current and the previous value. If the difference is large enough the calculation is done again. Otherwise the new voltage is corrected by the current offset.

3.3.2 Filtering

The filtering is done in a separate function—`offFilter`—which calculates the offset based on the previous value and a predefined weight:

```
offFilter offset offs = offset
    where
        offset' = ( offset * (filter_weight - 1) + offs )/
            filter_weight
        offset  = register 0 offset'
```

This filtering, however, is slightly problematic. Because of the way signals work in CλaSH, the filter value is updated every clock tick. In Matlab the filter value was updated only when a new measurement was given. The consequence of this is that the resulting offset values, and by extension the AMR values, differ from the results obtained by Matlab. A problem this creates is that an inaccurate offset can have a greater influence on the result, thereby decreasing the accuracy of the algorithm.

3.3.3 ADC Conversion

The main problem in the CλaSH code is located in the conversion from an ADC signal (unsigned fixed point with 0 bits before the point) to a usable signal for calculations (signed fixed point). Because the result of the ADC is relative to V_{cc}, it also has to be multiplied with that amount[1]. The function `adcConv` attempts to do this conversion:

[1] the correction also works correctly without this multiplication, but the resulting AMR value would differ with the expected value by a factor of 5.

```
adcConv (vp1,vn1,vp2,vn2) = (amr1, amr2, off1, off2)
    where
        -- Convert to right number of bits
        cvp1 = vp1 `times` (5 :: Signal (UFixed 7 9)) :: Signal
            (UFixed 7 25)
        cvn1 = vn1 `times` (5 :: Signal (UFixed 7 9)) :: Signal
            (UFixed 7 25)
        cvp2 = vp2 `times` (5 :: Signal (UFixed 7 9)) :: Signal
            (UFixed 7 25)
```

```
        cvn2 = vn2 `times` (5 :: Signal (UFixed 7 9)) :: Signal
            (UFixed 7 25)

        -- Convert to signed
        svp1 = bitCoerce1 (shiftR1 cvp1) :: Signal (SFixed 8 56)
        svn1 = bitCoerce1 (shiftR1 cvn1) :: Signal (SFixed 8 56)
        svp2 = bitCoerce1 (shiftR1 cvp2) :: Signal (SFixed 8 56)
        svn2 = bitCoerce1 (shiftR1 cvn2) :: Signal (SFixed 8 56)

        amr1 = svp1 - svn1
        amr2 = svp2 - svn2
        (amr1, amr2, off1, off2) = amrCalc (cvp1,cvn1,cvp2,cvn2)
```

The main problem in this code is that there is no `bitCoerce1` function in CλaSH, meaning it is very hard to change the way a signal's bits are interpreted. The result of this is that a proper test with the correct number of input bits is not possible with the current code.

3.3.4 Testbench

With some changes to the Matlab scripts with expected input and output values (`TestInput.hs` and `TestOutput.hs`) have been generated. These values can then be used in the native testbench functionality of CλaSH. This functionality can be used in CλaSH using the `testbench` function.

The main problem with the testbench is that Matlab and CλaSHdo not seem to agree on simple calculated values. Because of this some rounding has to be done. For the testbench this is done by using a simple bit shift, so the least significant bits are dropped.

Because of problems mentioned above, however, the testbench never passes. Results can be obtained by changing the input signal to the same type as the one that is used for the calculations.

To get a view on the performance of the system, the inputs of the system can be set to a fixed number of bits in Matlab. Because of the way the simulation is set up, the number of bits of the calculation can easily be changed. The calculation can therefore be done using a fixed point 32 bit (`SFixed 8 24`) and 64 bit (`SFixed 8 56`) signal.

RESULTS

Using the results obtained from the previously explained testbench, the performance of the system can be evaluated. The performance has been evaluated for a single test case, with several levels of bit-rounding.

4.1 Used parameters

Input and output values have been simulated by MatLab with the simulations described in section 3.2. The parameters of the script where:

- Ideal resistance: no
- Ideal AMR: no
- Number of ADC bits: 12, 16 and 24
- V_b: 62.5 mV
- *weight*: 8

The 24 bit value for the ADC has been added in order to simulate the 'none' situation described previously.

The values obtained from MatLab have been entered into CλaSH and used as data for the simulation. The result of the testbench was saved as a comma separated value.

The testbench was done twice: once with 32 bit fixed point signals and once with 64 bit fixed point signals.

4.2 Differences

In figure 4.10 and figure 4.12 the original and corrected values are shown. These graphs clearly show the inadequate performance of the implemented algorithm. The source of this is the calculation of the offset (shown in figure 4.11 and figure 4.13). The offset is orders of magnitude worse than the simulated one by Matlab.

The best simulated case, by far, is the 24 bit ADC. This is mainly because for the 32 bit implementation the offset was zero the whole time. In the 64-bit implementation it performed a lot better, as can be seen in table 4.8. This implementation is, on average, the only implementation that can be considered working. It's performance in degrees is not consistent everywhere though.

The correction applied for 16 and 12 bit implementations was too large in both the 32 and 64 bit implementation, which means there performance in correcting the angular error is poor.

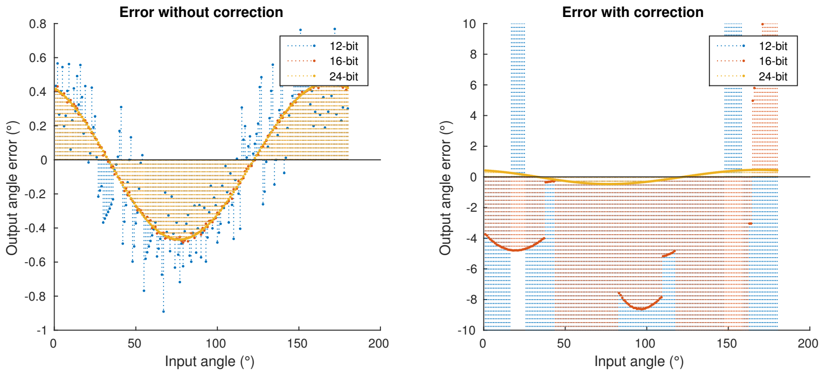

Figure 4.10: Offset errors before and after correction in several scenarios by the 32 bit CλaSH implementation.

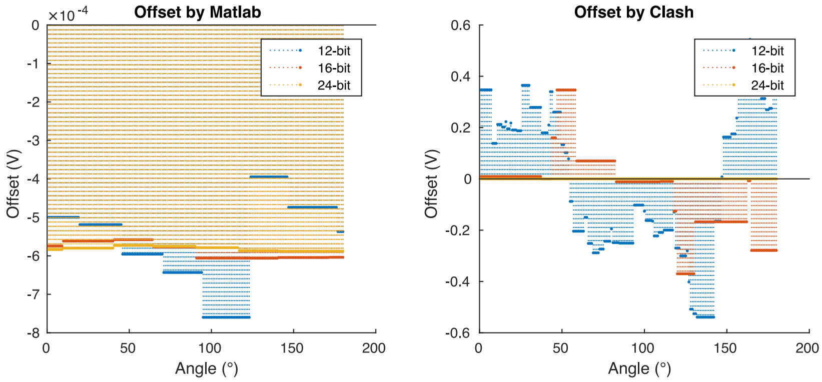

Figure 4.11: Offset before and after correction in several scenarios by the 32 bit CλaSH implementation.

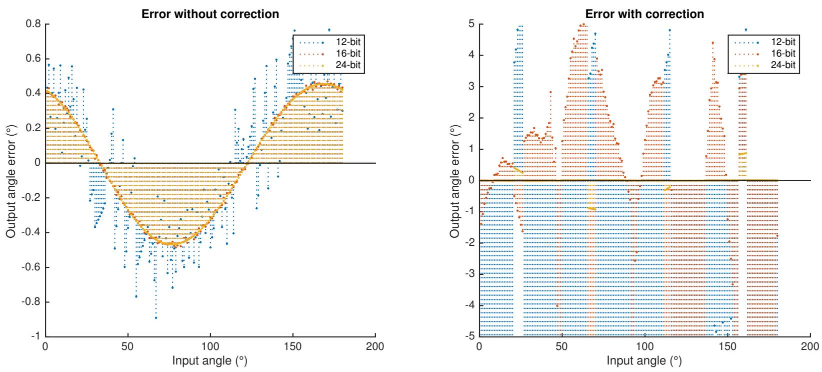

Figure 4.12: Offset errors before and after correction in several scenarios by the 64 bit CλaSH implementation.

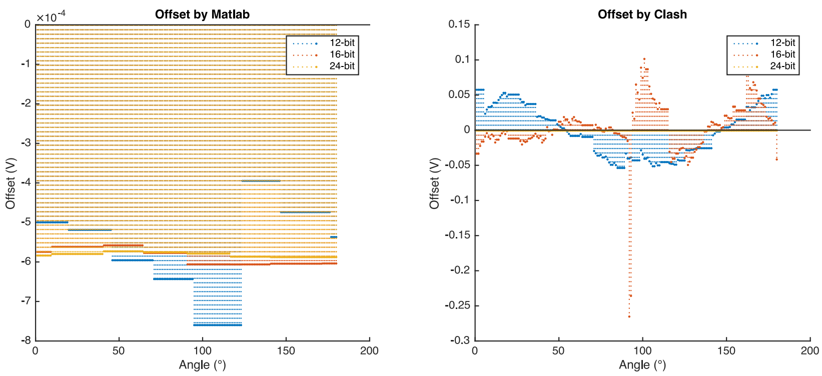

Figure 4.13: Offset before and after correction in several scenarios by the 64 bit CλaSH implementation.

Bits	AMR Off. (μV)	Matlab Off. (μV)	Matlab Improv.	CλaSH Off. (μV)	CλaSH Improv.
24	880	51.3	17.2	52.4	16.8
16	882	55.9	15.8	31.4×10^3	28.1×10^{-3}
12	969	450	2.16	31.5×10^3	30.7×10^{-3}

Table 4.8: Offset error before and after correction in several scenarios by the 64 bit CλaSH implementation.

DISCUSSION

The performance of the implementation in CλaSH was clearly inadequate. This can be contributed to multiple sources, of which the most significant one is the implementation itself.

5.1 Implementation goal

The goal of this implementation has always been the implementation in an ADC. This, however, should not have been a given: an alternative is implementing the calculations in the software receiving the measurements: the calculations can be simplified to require only the two AMR output voltages. This means a library with tunable parameters (depending on application) and software can then be published for inclusion in projects using this sensor. The current implementation in Matlab forms a good basis for such a library (and also provides values for bench-testing the performance), which can be implemented in any programming language. The downside of this is that it does not result in an easy to use embedded implementation and an ADC is still required.

5.2 Implementation in CλaSH

The decision to implement the project in CλaSH for rapid prototyping turned out to to be not so rapid. The main cause of this is inexperience on the author's part. It also presented extra challenges in the implementation of the filters, which needed to be dynamically updated. The lack of floating point variables might also have played a part in the inaccuracy of the calculations, though 24 bit fixed point (an LSB of 60 pV) should have been more than enough. Additional time, and probably training or experience, or the choice for a different language (eg: VHDL) might have alleviated these issues.

CONCLUSION

The performance of the implementation of the AMR compensation algorithm could not be evaluated because of an unfinished and extremely inaccurate implementation in CλaSH. In simulation, the performance of the algorithm turned out to be quite good. The simulated performance, however, greatly diminished when an ADC was introduced.

While no physical implementation was realized, the performance requirements can still be reviewed in some capacity:

- *Accuracy improvement of* 90 % *(factor of 10)*
  In simulation an accuracy improvement of 98.6 % was realised. This was, however greatly diminished by the ADC: while a 16 bit ADC still resulted in an improvement of 95 %, both a 14 and 12 bit ADC resulted in an improvement of less than 90 % (86 % and 61 % respectively).
- *Uses an ADC of less than 16 bits.*
  From the simulation results it became apparent that an ADC of at least 16 bits was required to achieve the accuracy requirement. This is technically not less than 16 bits, so this requirement cannot be achieved with the current implementation. Some better filtering might solve some of this by achieving acceptable results with a 14 bit ADC.
- *Can do at least 10 calculations per second.*
  This is mainly dependent on the ADC, and because no physical realisation was realised this could not be tested.

The conclusion of this project is that while the algorithm is sound and can certainly be used to correct variation in an AMR sensor, implementing it on an FPGA can be challenging.

MATLAB CODE

A.1 Scripts

A.1.1 adc_error

```
% Calculates maximum error due to the ADC
addpath('functions');

% AMR parameters
Vpeak = [73,81,89]*1e-3;
Vdd = 5;

% Array of bits
bits = 12:.1:24;
LSBs = Vdd./2.^(bits);

% Error
degs = 0:0.1:180;
errorcalc = @(lsb,Vp) max(abs( errorcast( atan2d(sind(2*degs).*Vp+lsb,cosd(2*degs).*Vp+lsb)/2 - degs) ));
error = arrayfun(errorcalc, repmat(LSBs,size(Vpeak,2),1), repmat(Vpeak',1,size(bits,2)));

% Plot
figure(3);
clf(3);
hold on;
plot(bits,error);
legend('Min␣(73␣mV)','Typ␣(81␣mV)','Max␣(89␣mV)');
xlabel('Number␣of␣bits');
ylabel('Error␣(°)');

% Output results
[bits(mod(bits,1) == 0); error(:,mod(bits,1) == 0)]
```

A.1.2 amr_test

```
% Test the properties of the AMR generation

tests = 10000;
random = true;
amrandom = true;

error = zeros(1,tests);

for i=1:tests;
    if mod(i,tests/100) == 0
        disp(i/tests*100);
    end

    rng(i);
    run('includes/amr_generate');

    amr_angles_none  = anglecast( voutnone(2,:), voutnone(1,:) );
    amr_angles_wrong = anglecast( vout(2,:), vout(1,:) );

    angle_error = errorcast(amr_angles_none - amr_angles_wrong);
```

```
    error(i) = max(abs(angle_error));
end;

hist(error,tests/20)
```

A.1.3 clash_results

```
clash12 = csvread('data/testbench_12.csv')';
clash16 = csvread('data/testbench_16.csv')';
clash24 = csvread('data/testbench_24.csv')';

clash12(5:8,:) = clash12(5:8,:);
clash16(5:8,:) = clash16(5:8,:);
clash24(5:8,:) = clash24(5:8,:);

range = 1080-180+1:1080;

amr12  = clash12(5:6, range);
amr16  = clash16(5:6, range);
amr24  = clash24(5:6, range);

vout12 = (clash12(1:2, range) + clash12(3:4, range));
vout16 = (clash16(1:2, range) + clash16(3:4, range));
vout24 = (clash24(1:2, range) + clash24(3:4, range));

off12i = clash12(3:4, range);
off16i = clash16(3:4, range);
off24i = clash24(3:4, range);

off12 = clash12(7:8, range);
off16 = clash16(7:8, range);
off24 = clash24(7:8, range);

%% Run AMR simulation
rng(1);
random = true;
amrandom = true;
batchmode = true;
filter_bound = 62.5e-3;
filter_weight = 8;

%% ADC simulation
figure(1);
figure(2);
clf(1);
clf(2);
for adc_bit = [12, 16, 24]
adc_ena = true;
adc_met = 2;

simulation_single_stripped;
voutnonex = repmat(voutnone,1,3);

amr  = eval(['amr' num2str(adc_bit)]);
vout = eval(['vout' num2str(adc_bit)]);
offi = eval(['off' num2str(adc_bit) 'i']);
off  = eval(['off' num2str(adc_bit)]);
offo = vout - voutnonex(:,range);

angle_pictures_layout
angle_pictures
%angle_results

disp(adc_bit)

off_orig = 1e6 * ((meanabs(offo)));
off_mat  = 1e6 * ((meanabs(offo - offi)));
off_clash= 1e6 * ((meanabs(offo - off)));

disp('Offsets (orig, matlab, clash)')
disp(off_orig)
disp(off_mat)
disp(off_clash)
```

```
disp('Improvement␣(matlab,␣clash)')
disp(off_orig/off_mat)
disp(off_orig/off_clash)

set(fig, 'Position', [80, 450, 900, 350])
subplot(1,2,1)
legend('12-bit', '16-bit', '24-bit');
subplot(1,2,2)
ylim([-5 5]);
legend('12-bit', '16-bit', '24-bit');

fig2 = figure(2);
set(fig2, 'Position', [80, 0, 900, 350])
% Unfixed angles
subplot(1,2,1)
hold on;
stem(offi(1,:),':.');
title('Offset␣by␣Matlab')
ylabel('Offset␣(V)')
xlabel('Angle␣(°)')
legend('12-bit', '16-bit', '24-bit');

% Fixed angles
subplot(1,2,2)
hold on;
stem(off(1,:),':.');
title('Offset␣by␣Clash')
ylabel('Offset␣(V)')
xlabel('Angle␣(°)')
legend('12-bit', '16-bit', '24-bit');

end
```

A.1.4 offset_rewrite

```
% This is a rework of the model provided in 'equation_system_offset_compensation.m'. It simulates a noon-ideal AMR and runs the calculation to compensate.

%% Initiate values for modeling
angs = 184;                     % Number of angles
Ro   = 3000;                     % Offset resistance.
amr = [84,83,83,83];             % Order 3, 4, 1, 2.
Vcc = 5;                         % Vcc of AMR.
Roff = [[2,9,13,9];[2,12,3,9]]; % Mismatched

%% Generate values of the AMR-sensor for 180°

% Generate angles
angles = 1:angs;
angles = arrayfun(@(n) [...
    [ cosd(n-1)^2;...
      cosd(n-1+45)^2 ],...
    [ sind(n-1)^2;...
      sind(n-1+45)^2]...
], angles, 'UniformOutput', false);

% Generate resistances
R = cellfun(@(c) repmat(c,1,2) .* repmat(amr,2,1) + Ro + Roff, angles, 'UniformOutput', false);
Rnone = cellfun(@(c) repmat(c,1,2) .* repmat(amr,2,1) + Ro, angles, 'UniformOutput', false);

% Generate vout
voutp = cellfun(@(r) [...
    r(1,1)/(r(1,1)+r(1,4)); ...
    r(2,1)/(r(2,1)+r(2,4)) ...
], R, 'UniformOutput', false);

voutn = cellfun(@(r) [...
    r(1,2)/(r(1,3)+r(1,2)); ...
    r(2,2)/(r(2,3)+r(2,2)) ...
], R, 'UniformOutput', false);

% Generate vout without offset
voutpnone = cellfun(@(r) [...
```

```
    r(1,1)/(r(1,1)+r(1,4)); ...
    r(2,1)/(r(2,1)+r(2,4)) ...
], Rnone, 'UniformOutput', false);

voutnnone = cellfun(@(r) [...
    r(1,2)/(r(1,3)+r(1,2)); ...
    r(2,2)/(r(2,3)+r(2,2)) ...
], Rnone, 'UniformOutput', false);

%resistances without offset
%Rnone = Ro + [amr(3)  * coseno(1), amr(4) * seno(1), amr(2) * coseno(1), amr(1) * seno(1)];

% Convert back to regular matrices
voutp = cell2mat(voutp) * Vcc;
voutn = cell2mat(voutn) * Vcc;

voutpnone = cell2mat(voutpnone) * Vcc;
voutnnone = cell2mat(voutnnone) * Vcc;

% Calculate actual output
voutnone = voutpnone - voutnnone;

%% ADC simulation
% Cast to 8 bit signed fixed point
voutp = fi(voutp,0,32);
voutn = fi(voutn,0,32);

% Calculate vout
vout = double(voutp - voutn);

%% Value calculation
off  = zeros(2,angs-3);
offp = off;
offn = off;
amr  = off;
%for i = 1:angs-3 % Positive calculation
%    % Get three values of vout
%    Vp = voutp(:,i:i+2);
%    Vn = voutn(:,i:i+2);
 for i = 3:angs % Negative calculation
    % Get three values of vout
    Vp = fliplr(voutp(:,i-2:i));
    Vn = fliplr(voutn(:,i-2:i));
    Vo = Vp - Vn;

    % Calculate offset summed (offp + offn)
    Vd = voutn(:,i) + voutp(:,i) - Vcc;

    % Fixed to float
    Vo = double(Vo);
    Vd = double(Vd);

    % Calculate squares
    Vsqr = [ Vo(1,2)^2 + Vo(2,2)^2 - Vo(1,1)^2 - Vo(2,1)^2 ;...
             Vo(1,3)^2 + Vo(2,3)^2 - Vo(1,1)^2 - Vo(2,1)^2 ];

    % Calculate differential voltage pairs
    Vx = [[ Vo(1,2)-Vo(1,1), Vo(1,3)-Vo(1,1) ];
          [ Vo(2,2)-Vo(2,1), Vo(2,3)-Vo(2,1) ]];

    % Subexpressions from final answers
    cse(1) = 2*(Vx(1,2)*Vx(2,1) - Vx(1,1)*Vx(2,2));

    % Calculate offsets
    offs = zeros(2,1);
    offs(1) = (Vx(2,1)*Vsqr(2) - Vx(2,2)*Vsqr(1))/cse(1);
    offs(2) = (Vx(1,2)*Vsqr(1) - Vx(1,1)*Vsqr(2))/cse(1);

    % Calculate actual offsets
    offp1 = (offs(1)+Vd(1))/2;
    offp2 = (offs(2)+Vd(2))/2;
    offn1 = Vd(1) - offp1;
    offn2 = Vd(2) - offp2;

```

```
    % Save solutions
    off(:,i)  = offs;
    offp(:,i) = [offp1;offp2];
    offn(:,i) = [offn1;offn2];

    % Correct amr values
    amr(:,i) = Vo(:,1) - offs;
end

% Check results
load('values_andreina.mat');

a_off_mean
off_mean = mean(abs(off),2)

a_offv
off_v    = (max(vout,[],2) + min(vout,[],2))/2

a_off_amr
off_amr  = (max(amr,[],2) + min(amr,[],2))/2



% Result meaning
amr_angles_andr  = -.5 * atan2d( a_amr_b(2,:), a_amr_b(1,:) );
amr_angles_none  = -.5 * atan2d( voutnone(2,:), voutnone(1,:) );
amr_angles_wrong = -.5 * atan2d( vout(2,:), vout(1,:) );
amr_angles_fixed = -.5 * atan2d( amr(2,:), amr(1,:) );

% Fix 180 deg errors
amr_angles_wrong(91:93) = amr_angles_wrong(91:93) - 180;
amr_angles_fixed(91) = amr_angles_fixed(91) - 180;

% Uncorrected error
angle_error = meanabs(amr_angles_none - amr_angles_wrong)

% Corrected error
angle_error_fixed = meanabs(amr_angles_none(3:183) - amr_angles_fixed(3:183))
```

A.1.5 simulation_multi

```
% This is a model of the implementation of an AMR correction algorithm on an embedded system.

%% Setup simulation environment
addpath('includes', 'functions');

% Check if this script is running in batch mode
if exist('batchmode','var') && batchmode
    showplots = false;
else
    rng(1);
    batchmode = false;
    showplots = true;

    % Method 2 parameters
    filter_bound = 62.5e-3;
    filter_weight = 8;
end

%% Run AMR simulation
random = true;
amrandom = false;
amr_generate;

%% Some setup
angs = 180;
sample_range = 1:angs;
adc_met = 2;

%% ADC simulation

% Use single precision from now on
feature('SetPrecision', 24)
```

```

voutpdouble = voutp;
voutndouble = voutn;

bits = [12,14,16];

tests = size(bits, 2) + 1;
test_bits    = zeros(tests,1);
angles_none  = zeros(tests,180);
errors_none  = zeros(tests,180);
angles_fixed = zeros(tests,180);
errors_fixed = zeros(tests,180);

% Clear figure
% Create general figure
if showplots
    figure(1);
    clf(1);
    legenditems = {};
end

for adc_ena = [0,1]
    for adc_bit = bits
        if adc_ena == 0 & adc_bit == 16 | adc_ena
            % Progress...
            if ~batchmode
                disp([num2str(adc_ena) '-' num2str(adc_met) '-' num2str(adc_bit) ])
            end

            % ADC simulation
            if adc_ena
                % Cast to n bit unsigned fixed point
                Vref = Vcc;
                voutp = fi(voutpdouble./Vref,0,adc_bit,adc_bit);
                voutn = fi(voutndouble./Vref,0,adc_bit,adc_bit);

                % Vcc is reference voltage
                %Vcc = fi(Vcc,0,adc_bit,adc_bit);

                % Cast to floating points
                voutp = single(voutp).*Vref;
                voutn = single(voutn).*Vref;
                vout = voutp - voutn;
                Vcc = single(Vcc);
            end

            %% Value calculation
            off  = zeros(2,max(sample_range));
            offp = off;
            offn = off;
            amr  = off;

            % Run the calculation method
            run(['method_' num2str(adc_met)])

            % Save data
            test_bits   = circshift(test_bits,-1,1);
            angles_none  = circshift(angles_none,-1,1);
            errors_none  = circshift(errors_none,-1,1);
            angles_fixed = circshift(angles_fixed,-1,1);
            errors_fixed = circshift(errors_fixed,-1,1);

            test_bits(end)      = adc_ena * adc_bit;
            angles_none(end,:)  = amr_angles_wrong(sample_range);
            errors_none(end,:)  = angle_error;
            angles_fixed(end,:) = amr_angles_fixed(sample_range);
            errors_fixed(end,:) = angle_error_fixed;
        end
    end
end

% Create general figure
if showplots
    run('angle_pictures_layout');
```

```

    if exist('showresponse','var') && showresponse
        fig2 = figure(2);
        clf(2);
        hold on;

        title('ADC␣caused␣error');
        xlabel('Number␣of␣ADC␣bits');
        ylabel('Maximum␣error␣(°)');

        stem(test_bits(2:end), max(abs(errors_none(2:end,:)),[],2), ':o' );
        stem(test_bits(2:end), max(abs(errors_fixed(2:end,:)),[],2), ':o' );
        stem(test_bits(2:end), max(abs(errors_fixed(2:end,:)),[],2) - max(abs(errors_none(2:end,:)
            ),[],2), ':o' );

        legend('Before␣correction', 'After␣correction', 'Difference');
    end
end
```

A.1.6 simulation_single

```
% This is a model of the implementation of an AMR correction algorithm on an embedded system.

%% Setup simulation environment
addpath('includes', 'functions');

% Check if this script is running in batch mode
if exist('batchmode','var') && batchmode
    showplots = false;
else
    rng(1);
    batchmode = false;
    showplots = true;

    % Method 2 parameters
    filter_bound = 62.5e-3;
    filter_weight = 8;
end

%% Run AMR simulation
random = true;
amrandom = true;
run('amr_generate');

%% Some setup
angs = 180;
sample_range = 1:angs;

%% ADC simulation
adc_ena = true;
adc_bit = 24;
adc_met = 2;

% Use single precision from now on
feature('SetPrecision', 24)

if adc_ena
    % Cast to n bit unsigned fixed point
    Vref = Vcc;
    voutp = fi(voutp./Vref,0,adc_bit,adc_bit);
    voutn = fi(voutn./Vref,0,adc_bit,adc_bit);

    % Vcc is reference voltage
    %Vcc = fi(Vcc,0,adc_bit,adc_bit);

    % Cast to floating points
    voutp = single(voutp).*Vref;
    voutn = single(voutn).*Vref;
    vout = voutp - voutn;
    Vcc = single(Vcc);
end

%% Value calculation
```

```
off  = zeros(2,max(sample_range));
offp = off;
offn = off;
amr  = off;

% Create general figure
if showplots
    figure(1);
    clf(1);
    legenditems = {};
    run('angle_pictures_layout');
end

% Run the calculation method
run(['method_' num2str(adc_met)])

% Get results
%pause(10);
```

A.1.7 simulation_single_stripped

```
% This is a model of the implementation of an AMR correction algorithm on an embedded system.

%% Setup simulation environment
addpath('includes', 'functions');

% Check if this script is running in batch mode
if exist('batchmode','var') && batchmode
    showplots = false;
else
    %rng(1);
    batchmode = false;
    showplots = true;

    % Method 2 parameters
    filter_bound = 62.5e-3;
    filter_weight = 8;
end

%% Run AMR simulation
run('amr_generate');

%% Some setup
angs = 180;
sample_range = 1:angs;

% Use single precision from now on
feature('SetPrecision', 24)

if adc_ena
    % Cast to n bit unsigned fixed point
    Vref = Vcc;
    voutp = fi(voutp./Vref,0,adc_bit,adc_bit);
    voutn = fi(voutn./Vref,0,adc_bit,adc_bit);

    % Vcc is reference voltage
    %Vcc = fi(Vcc,0,adc_bit,adc_bit);

    % Cast to floating points
    voutp = single(voutp).*Vref;
    voutn = single(voutn).*Vref;
    vout = voutp - voutn;
    Vcc = single(Vcc);
end

%% Value calculation
off  = zeros(2,max(sample_range));
offp = off;
offn = off;
amr  = off;

% Run the calculation method
run(['method_' num2str(adc_met)])
```

A.1.8 testInputGen

```
fulltestdata = testdata;%(cellfun(@(x) size(x,1) > 0,testdata));

inputs = size(fulltestdata,2);

infile = ['../Clash/TestInput_' num2str(adc_bit) '.hs'];
outfile = ['../Clash/TestOutput_' num2str(adc_bit) '.hs'];

type1 = ['(␣' strjoin(repmat({'SFixed␣8␣24'},1,4)',',␣') ')'];
type2 = ['(␣' strjoin(repmat({'SFixed␣8␣24'},1,4)',',␣') ')'];

testInput = zeros(3,4);
testOutput = zeros(3,4);

for c = 1:inputs
    d = fulltestdata{c};
    testInput(c,:) = [ d(1,1) d(1,2) d(2,1) d(2,2) ]/Vcc;
    testOutput(c,:) = [ d(1,4) d(2,4) d(1,5) d(2,5) ];
end

delete(infile)
diary(infile)

disp(['module␣TestInput_'  num2str(adc_bit) '␣where'])
disp('import␣CLaSH.Prelude')
disp(['testInput␣::␣Signal' type1 ])
disp('testInput␣=␣stimuliGenerator␣$(v␣[')

for i = 1:size(testInput,1)
    a = arrayfun(@(e) num2str(e,50), testInput(i,:), 'Un', 0);
    string = ['␣␣␣␣(' strjoin(a, ',␣') ')'];
    if i == 1
        disp([string '␣::␣' type1 ','])
    elseif i == size(testInput,1)
        disp([string '])'])
    else
        disp([string ','])
    end
end

diary off;
delete(outfile)
diary(outfile)

disp(['module␣TestOutput_'  num2str(adc_bit) '␣where'])
disp('import␣CLaSH.Prelude')
disp(['expectedOutput␣::␣Signal␣' type2 '␣->␣Signal␣Bool'])
disp('expectedOutput␣=␣outputVerifier␣expectedData')
disp('expectedData␣␣␣=␣$(v␣[')

for i = 1:size(testOutput,1)
    a = arrayfun(@(e) num2str(e,50), testOutput(i,:), 'Un', 0);
    string = ['␣␣␣␣(' strjoin(a, ',␣') ')'];
    if i == 1
        disp([string '␣::␣' type2 ','])
    elseif i == size(testOutput,1)
        disp([string '])'])
    else
        disp([string ','])
    end
end

diary off;

dlmwrite('testInput.csv',testInput,'delimiter',',','Precision',50);
dlmwrite('testOutput.csv',testOutput,'delimiter',',','Precision',50);
```

A.1.9 test_2

```
tests = 1000;

```

```
batchmode = true;

filter_bound = 23.0e-3;
filter_weight = 4;

mean_fails = 0;
max_fails = 0;

orig_mean = zeros(1,tests);
orig_med = zeros(1,tests);
orig_max = zeros(1,tests);
fix_mean = zeros(1,tests);
fix_med = zeros(1,tests);
fix_max = zeros(1,tests);

for s=1:tests
    disp(s/tests * 100)
    rng(s);
    run('simulation_single');
    run('angle_errors');

    orig_mean(s) = angle_error_mean;
    orig_med(s) = median(abs(angle_error));
    orig_max(s) = max(abs(angle_error));

    fix_mean(s) = angle_error_fixed_mean;
    fix_med(s) = median(abs(angle_error_fixed));
    fix_max(s) = max(abs(angle_error_fixed));

    if fix_mean(s) > orig_mean(s)
        %disp(['Mean error: rng=' num2str(s)])
        mean_fails = mean_fails + 1;

    elseif fix_max(s) > orig_max(s)
        %disp(['Max errorr: rng=' num2str(s)])
        max_fails = max_fails + 1;
    end
end

% Stats
mean_fails
max_fails
[mean(orig_mean) mean(fix_mean) mean(orig_max)  mean(fix_max)]

% LaTeX output
disp(['\num{' num2str(mean(orig_mean)*1e3) '} & \num{' num2str(mean(fix_mean)*1e3) '} & \num{'
    num2str(mean(orig_max)*1e3) '} & \num{' num2str(mean(fix_max)*1e3) '} \\'])


batchmode = false;
```

A.1.10 test_2_sweep

```
% Sweep over a parameter using 'simulation_single'.
% Test type
sweep = 'bound';
zoom = false;

% Number of tests per parameter value
tests = 100;

% Numer of parameter values (-1)
kmax = 100;

% Default values
filter_bound = 26.7e-3;
filter_weight = 8;

% Script options
batchmode = true;

% Result arrays
test_x = zeros(1,kmax+1);
```

```matlab
test_y = zeros(2,kmax+1);
test_data = cell(4,kmax+1);

% Parallel computing
parpool(4);

% Run simulation
for k = 1:kmax+1
    % Progress
    disp((k-1)/kmax*100);

    % Parameter sweep values
    if strcmp(sweep,'bound')
        if zoom
            filter_bound = (k-1) * 0.045/kmax + 0.01;
        else
            filter_bound = (k-1) * .1/kmax;
        end
        test_x(k) = filter_bound;
    elseif strcmp(sweep,'weight')
        if zoom
            filter_weight = (k-1) * 12/kmax + 4;
        else
            filter_weight = k;
        end
        test_x(k) = filter_weight;
    end

    % Data arrays
    mean_fails = 0;
    max_fails = 0;

    orig_mean = zeros(1,tests);
    orig_max = zeros(1,tests);
    fix_mean = zeros(1,tests);
    fix_max = zeros(1,tests);

    % Run tests
    parfor s=1:tests
        rng(s);
        batchmode('simulation_single');
        batchmode('angle_errors');

        orig_mean(s) = angle_error_mean;
        orig_max(s) = max(abs(angle_error));

        fix_mean(s) = angle_error_fixed_mean;
        fix_max(s) = max(abs(angle_error_fixed));

        if fix_mean(s) > orig_mean(s)
            %disp(['Mean error: rng=' num2str(s)])
            mean_fails = mean_fails + 1;

        elseif fix_max(s) > orig_max(s)
            %disp(['Max errorr: rng=' num2str(s)])
            max_fails = max_fails + 1;
        end
    end

    % Store results
    test_y(:,k) = [mean_fails; max_fails];
    test_data(:,k) = {orig_mean, orig_max, fix_mean, fix_max};
end

% Plot results
figure(1);
clf(1);
hold on;
% Number of errors
stem(test_x,test_y(1,:),':');
stem(test_x,test_y(2,:),':');

% More
test_y(3,:) = cellfun(@meanabs, test_data(3,:));
```

```
test_y(4,:) = cellfun(@meanabs, test_data(4,:));

stem(test_x, test_y(3,:),':');
stem(test_x, test_y(4,:),':');

% Legend
legend('Number␣of␣worse␣means', 'Number␣of␣worse␣maxima', 'Sum␣of␣means', 'Sum␣of␣maxima');

% No batch after this
batchmode = false;
delete(gcp);
```

A.1.11 test_2_sweep_multi

```
% Sweep over a parameter using 'simulation_multi'.
% Test type
sweep = 'bound';
zoom = false;

% Number of tests per parameter value
loops = 5;

% Numer of parameter values (-1)
kmax = 10;

% Default values
filter_bound = 62.5e-3;
filter_weight = 8;

% Script options
batchmode = true;

% Run once for params
run('simulation_multi');
legenditems = {};

% Result arrays
test_x = zeros(1,kmax+1);
test_y = zeros(tests,kmax+1);
test_data = cell(4,kmax+1);

% Run simulation
for k = 1:kmax+1
    % Progress
    disp((k-1)/kmax*100);

    % Parameter sweep values
    if strcmp(sweep,'bound')
        if zoom
            filter_bound = (k-1) * 0.05/kmax + 0.02;
        else
            filter_bound = (k-1) * .2/kmax;
        end
        test_x(k) = filter_bound;
    elseif strcmp(sweep,'weight')
        if zoom
            filter_weight = (k-1) * 6/kmax + 3;
        else
            filter_weight = k;
        end
        test_x(k) = filter_weight;
    end

    orig_mean = zeros(tests,loops);
    orig_max = zeros(tests,loops);
    fix_mean = zeros(tests,loops);
    fix_max = zeros(tests,loops);

    % Run tests
    for s=1:loops
        rng(s);
        run('simulation_multi');

```

```
        orig_mean(:,s) = mean(abs(errors_none(1:end,:)),2);
        orig_max(:,s) = max(abs(errors_none(1:end,:)),[],2);

        fix_mean(:,s) = mean(abs(errors_fixed(1:end,:)),2);
        fix_max(:,s) = max(abs(errors_fixed(1:end,:)),[],2);
    end

    % Store results
    test_data(:,k) = {orig_mean, orig_max, fix_mean, fix_max};
end

% Plot results
figure(1);
clf(1);
test_y = cell2mat(cellfun(@(x) mean(x,2), test_data(4,:),'Un',0));
stem(test_x, test_y',':');

% Legend
legend(legenditems(1:4));

% No batch after this
batchmode = false;
```

A.1.12 test_performance

```

adc_met = 1;

tests = 1000;

batchmode = true;

filter_bound = 23.0e-3;
filter_weight = 4;

bitlevels   = [0];
%bitlevels   = [0, 16, 14, 12];
boundlevels = [23, 40.5, 48.5, 62.5]*1e-3;
weightlevels = [4, 5, 6, 8];

performance_data = zeros(1,6);


for ideal=0:3

    if ideal==1 | ideal==3
        amrandom = true;
    else
        amrandom = false;
    end

    if ideal==2 | ideal==3
        random = true;
    else
        random = false;
    end

    for bittest = 1:1
        adc_bit = bitlevels(bittest);
        filter_bound = boundlevels(bittest);
        filter_weight = weightlevels(bittest);

        if adc_bit == 0
            adc_ena = false;
        else
            adc_ena = true;
        end

        orig_mean = zeros(1,tests);
        orig_med = zeros(1,tests);
        orig_max = zeros(1,tests);
        fix_mean = zeros(1,tests);
        fix_med = zeros(1,tests);
```

```
        fix_max = zeros(1,tests);


        for s=1:tests
            disp((s+(ideal*size(bitlevels,2)+bittest-1)*tests)/(tests * 4 * size(bitlevels,2)) *
                100)
            rng(s);
            run('simulation_single_stripped');
            run('angle_errors');

            orig_mean(s) = angle_error_mean;
            orig_med(s) = median(abs(angle_error));
            orig_max(s) = max(abs(angle_error));
            orig_off(s) = offset_error_mean_sum;

            fix_mean(s) = angle_error_fixed_mean;
            fix_med(s) = median(abs(angle_error_fixed));
            fix_max(s) = max(abs(angle_error_fixed));
            fix_off(s) = offset_error_fixed_mean_sum;
        end

        performance_data(bittest + ideal * size(bitlevels,2), :) = ...
            [mean(orig_mean) mean(fix_mean) mean(orig_max)  mean(fix_max) mean(orig_off) mean(
                fix_off)];

    end
end

% Improvement
performance_data(:,end+1) = performance_data(:,1)./performance_data(:,2);
performance_data(:,end+1) = performance_data(:,3)./performance_data(:,4);
performance_data(:,end+1) = performance_data(:,5)./performance_data(:,6);

batchmode = false;
```

A.2 Functions

A.2.1 anglecast

```
function [ a ] = anglecast( X, Y )
%ANGLECAST Summary of this function goes here
%   Detailed explanation goes here
    a = -.5 * atan2d(X,Y);
    a(a < 0) = a(a < 0) + 180;
    a(isnan(a)) = 0;
end
```

A.2.2 errorcast

```
function [ e ] = errorcast( e )
%ERRORCAST Summary of this function goes here
%   Detailed explanation goes here
    for i = 1:2
        e(e < -90) = e(e < -90) + 180;
        e(e > 90) = e(e > 90) - 180;
    end
end
```

A.3 Includes

A.3.1 amr_generate

```
% This is a rework of the model provided in 'equation_system_offset_compensation.m'. It simulates
    a non-ideal AMR.

%% Initiate values for modeling
angs = 360;                         % Number of angles
Ro   = 3000;                        % Offset resistance.
Ra   = 99;                          % AMR resistance
amrr = zeros(1,4) + Ra;             % Order 3, 4, 1, 2.
amrrideal = amrr;
Vcc  = 5;                           % Vcc of AMR.
Roff = [[2,9,13,9];[2,12,3,9]]; % Mismatched
Roff = zeros(2,4) + Ro;             % Ideal

% Generation of non-idealities
% Documentation works out to maximum of 2.69 degrees error
% This works out to sigma = 2.5 ohm
% Random Roff
if random
    Roff = randn(2,4) * 2.5 + Ro;
end

% Sigma is approximated for four AMRs and sigma3 = 32/27 * Ra
if amrandom
   amrr = amrr + randn(1,4) * (8/81) * Ra/12;
end

%% Generate values of the AMR-sensor for 180 degrees

% Generate angles
angles = 1:angs;
angles = arrayfun(@(n) [...
    [ cosd(n-1)^2;...
      cosd(n-1+45)^2 ],...
    [ sind(n-1)^2;...
      sind(n-1+45)^2]...
], angles, 'UniformOutput', false);

% Generate resistances
R = cellfun(@(c) repmat(c,1,2) .* repmat(amrr,2,1) + Roff, angles, 'UniformOutput', false);
Rnone = cellfun(@(c) repmat(c,1,2) .* repmat(amrrideal,2,1) + Ro, angles, 'UniformOutput', false);

% Generate vout
voutp = cellfun(@(r) [...
    r(1,1)/(r(1,1)+r(1,4)); ...
    r(2,1)/(r(2,1)+r(2,4)) ...
], R, 'UniformOutput', false);

voutn = cellfun(@(r) [...
    r(1,2)/(r(1,3)+r(1,2)); ...
    r(2,2)/(r(2,3)+r(2,2)) ...
], R, 'UniformOutput', false);

% Generate vout without offset
voutpnone = cellfun(@(r) [...
    r(1,1)/(r(1,1)+r(1,4)); ...
    r(2,1)/(r(2,1)+r(2,4)) ...
], Rnone, 'UniformOutput', false);

voutnnone = cellfun(@(r) [...
    r(1,2)/(r(1,3)+r(1,2)); ...
    r(2,2)/(r(2,3)+r(2,2)) ...
], Rnone, 'UniformOutput', false);

% Convert back to regular matrices
voutp = cell2mat(voutp) * Vcc;
voutn = cell2mat(voutn) * Vcc;

voutpnone = cell2mat(voutpnone) * Vcc;
```

```
voutnnone = cell2mat(voutnnone) * Vcc;

% Calculate actual output
voutnone = voutpnone - voutnnone;
vout = voutp - voutn;
```

A.3.2 angle_errors

```
% Calculate all angle errors
% amr_angles_andr  = anglecast( a_amr_b(2,:), a_amr_b(1,:) );
amr_angles_none  = anglecast( voutnone(2,:), voutnone(1,:) );
amr_angles_wrong = anglecast( vout(2,:), vout(1,:) );
amr_angles_fixed = anglecast( amr(2,:), amr(1,:) );

% Uncorrected error
angle_error = errorcast(amr_angles_none(sample_range) - amr_angles_wrong(sample_range));
angle_error_mean = meanabs(angle_error);
angle_error_std  = std(angle_error);
angle_error_max  = max(abs(angle_error));

% Corrected error
angle_error_fixed = errorcast(amr_angles_none(sample_range) - amr_angles_fixed(sample_range));
angle_error_fixed_mean = meanabs(angle_error_fixed);
angle_error_fixed_std  = std(angle_error_fixed);
angle_error_fixed_max  = max(abs(angle_error_fixed));

% Offsets
offset_error = vout(:,sample_range) - voutnone(:,sample_range);
offset_error_mean = mean(offset_error,2);
offset_error_mean_sum  = sumabs(offset_error_mean);

offset_error_fixed = offset_error - offlog(:,sample_range);
offset_error_fixed_mean  = mean(offset_error_fixed,2);
offset_error_fixed_mean_sum  = sumabs(offset_error_fixed_mean);
```

A.3.3 angle_pictures

```
% Generate plots of data
% Load data
angle_errors;

% Create figure
figure(1);

% Unfixed angles
subplot(1,2,1)
hold on;
stem(angle_error,':.');

% Fixed angles
subplot(1,2,2)
hold on;
stem(angle_error_fixed,':.');

% Export
filename = '../Report/graphics/generated/fpga_simulation_method_';
filename = [ filename num2str(adc_met) '_'];
if adc_ena
    filename = [filename 'adc_' num2str(adc_bit)];
else
    filename = [filename 'ideal'];
end

% hgexport(1,[filename '.eps']);
```

A.3.4 angle_pictures_layout

```
% Create figure
fig = figure(1);
%set(fig, 'Position', [1920, 0, 900, 350])
%set(fig, 'Position', [50, 0, 900, 350])
```

```

subplot(1,2,1)
hold on;
title('Error without correction')
xlabel('Input angle (°)')
ylabel('Output angle error (°)')
if exist('legenditems','var')
    legend(legenditems);
end

subplot(1,2,2)
hold on;
title('Error with correction')
xlabel('Input angle (°)')
ylabel('Output angle error (°)')
if exist('legenditems','var')
    legend(legenditems);
end

% limit axis
angmax = 5;
if max(abs(ylim)) > angmax
    ylim([-angmax angmax]);
end

% Export
filename = '../Report/graphics/generated/fpga_simulation_method_';
filename = [ filename num2str(adc_met) '_overview'];
hgexport(1,[filename '.eps']);
```

A.3.5 angle_results

```
% Output results
run('angle_pictures');

angle_error_mean
angle_error_fixed_mean

angle_error_std
angle_error_fixed_std

angle_error_max
angle_error_fixed_max
```

A.3.6 method_1

```
% Method 1: take three values

% Some setup
sample_step = 1;
sample_offset = 2 * sample_step + 1;
sample_range = sample_offset:angs+sample_offset-1;

for i = sample_range % Negative calculation
    % Get three values of vout
    Vp = [voutp(:,i) voutp(:,i-sample_step) voutp(:,i-2*sample_step)];
    Vn = [voutn(:,i) voutn(:,i-sample_step) voutn(:,i-2*sample_step)];

    % Calculate data
    [amr(:,i), off(:,i), offp(:,i), offn(:,i) ] = offset_calculation(Vp, Vn, Vcc);
end

% Create legend string
if exist('legenditems','var')
    if adc_ena
        legenditems(end+1) = {[num2str(adc_bit) ' bit']};
    else
        legenditems(end+1) = {'No ADC'};
    end
end

% Show plots
```

```
if exist('showplots','var') && showplots
    run('angle_results');

    if exist('showresponse','var') && showresponse
        % Plots of the response of both offsets
        figure(2);
        clf(2);
        subplot(2,1,1);
        hold on;
        %plot(repmat(-a_off_b(1,1:angs),1,samples/angs+1))
        plot(offslog(1,1:samples+angs));
        plot(offlog(1,1:samples+angs));

        subplot(2,1,2);
        hold on;
        %plot(repmat(-a_off_b(2,1:angs),1,samples/angs+1))
        plot(offslog(2,1:samples+angs));
        plot(offlog(2,1:samples+angs));
    end
end

offlog = off;
```

A.3.7 method_2

```
% Method 2: look behind and filter

% Double the matrix size
reps = 3;
samples = size(voutp,2) * (reps-1);
voutprep = repmat(voutp,1,reps);
voutnrep = repmat(voutn,1,reps);

% Create stacks
off = zeros(2,1);
offlog = zeros(2,1);
offslog = zeros(2,1);

Vp = zeros(2,3);
Vn = zeros(2,3);

testdata = {};

for i = 1:size(voutprep,2)

    % Check if top of stack is different
    % Check in Vout
    Diff = max(abs(...
        ( voutprep(:,i) - voutnrep(:,i) ) - ( Vp(:,1)-Vn(:,1) )...
    ));
    % Check in raw values
    Pdiff = voutprep(:,i) - Vp(:,1);
    Ndiff = voutnrep(:,i) - Vn(:,1);
    Adiff = abs([Pdiff(:); Ndiff(:)]);

    % Debugging
    %diff(i) = Diff;
    %adiff(i) = max(Adiff);

    % What to do when stack is different
    %if max(Adiff) > filter_bound
    if Diff > filter_bound
        % Shift the stacks
        Vp = circshift(Vp,1,2);
        Vn = circshift(Vn,1,2);
        % Change the top value
        Vp(:,1) = voutprep(:,i);
        Vn(:,1) = voutnrep(:,i);

        % Do the calculation
        [~, offs, ~, ~ ] = offset_calculation(Vp, Vn, Vcc);
        offslog(:,i) = offs;

```

```
        % Validate and filter offset
        if offs ~= zeros(2,1) & max(abs(offs)) < Vcc/2 & sum(Vp(:,3) + Vn(:,3)) > 0
            % Discrete low-pass filter
            off = (off.*(filter_weight-1) + offs)/filter_weight;
        end

        % Calculate amr, with tracking
        Vo = Vp - Vn;
        offlog(:,i) = off;
        amr(:,i) = Vo(:,1) - off; %mean(off,2);
    elseif max(Adiff) > 0
        % Calculate with old offset
        amr(:,i) = voutprep(:,i) - voutnrep(:,i) - off;
        offlog(:,i) = offlog(:,max(1,i-1));
        offslog(:,i) = offslog(:,max(1,i-1));
    else
        % Nothing is different
        offlog(:,i) = offlog(:,max(1,i-1));
        offslog(:,i) = offslog(:,max(1,i-1));
        amr(:,i) = amr(:,max(1,i-1));
    end

    testdata(i) = {[voutprep(:,i) voutnrep(:,i) (voutprep(:,i)-voutnrep(:,i)) amr(:,i) off]};
end

% Halve the matrix size
amr = amr(:,samples+1:samples+angs);

% Calculate errors
run('angle_errors.m');

% Create legend string
if exist('legenditems','var') && size(legenditems,2) < 100
    if adc_ena
        legenditems(end+1) = {[num2str(adc_bit) ' bit']};
    else
        legenditems(end+1) = {'No ADC'};
    end
end

% Show plots
if exist('showplots','var') && showplots
    run('angle_results');

    if exist('showresponse','var') && showresponse
        % Plots of the response of both offsets
        figure(2);
        clf(2);
        subplot(2,1,1);
        hold on;
        %plot(repmat(-a_off_b(1,1:angs),1,samples/angs+1))
        plot(offslog(1,1:samples+angs));
        plot(offlog(1,1:samples+angs));

        subplot(2,1,2);
        hold on;
        %plot(repmat(-a_off_b(2,1:angs),1,samples/angs+1))
        plot(offslog(2,1:samples+angs));
        plot(offlog(2,1:samples+angs));
    end
end
```

A.3.8 offset_calculation

```
function [ amr, offs, offp, offn ] = offset_calculation( Vp, Vn, Vcc )
%OFFSET_CALCULATION Calculates the offset given Vp and Vn
%   Detailed explanation goes here

    % Calculate Vout
    Vout = Vp - Vn;

    % Calculate offset summed (offp + offn)
    offS = Vn(:,1) + Vp(:,1) - Vcc;
```

```

    % Calculate squares
    Vsqr = [ Vout(1,2)^2 + Vout(2,2)^2 - Vout(1,1)^2 - Vout(2,1)^2 ;...
             Vout(1,3)^2 + Vout(2,3)^2 - Vout(1,1)^2 - Vout(2,1)^2 ];

    % Calculate differential voltage pairs
    Vx = [[ Vout(1,2)-Vout(1,1), Vout(1,3)-Vout(1,1) ];
          [ Vout(2,2)-Vout(2,1), Vout(2,3)-Vout(2,1) ]];

    % Subexpressions from final answers
    denom = 2*(Vx(1,2)*Vx(2,1) - Vx(1,1)*Vx(2,2));

    % Calculate offsets
    offs = zeros(2,1);
    offs(1) = (Vx(2,1)*Vsqr(2) - Vx(2,2)*Vsqr(1))/denom;
    offs(2) = (Vx(1,2)*Vsqr(1) - Vx(1,1)*Vsqr(2))/denom;

    % Calculate actual offsets
    offp1 = (offs(1)+offS(1))/2;
    offp2 = (offs(2)+offS(2))/2;
    offn1 = offS(1) - offp1;
    offn2 = offS(2) - offp2;

    % Validate offset
    if abs(mean(offs)) == Inf | isnan(mean(offs))
        offs = zeros(2,1);
    end

    % Save solutions
    offp = [offp1;offp2];
    offn = [offn1;offn2];

    % Correct amr values
    amr = Vout(:,1) - offs;
end
```

A.3.9 sweep_save

```
% Save plot
label = sweep;
if zoom
    label = [sweep '_zoom'];
end
%errormat = [error_x',error_mean_ideal',error_mean_12',error_mean_16',error_mean_24',error_mean_32
    '];

hgexport(1,['data/sweep_plot_' label '.eps']);
save(['data/sweep_data_' label '.mat'], 'test_x', 'test_y', 'test_data');
```

FPGA CODE

B.1 amr

```
module AMR where

import CLaSH.Prelude

import TestInput
import TestOutput

-- Settings
filter_weight = 8
filter_bound  = 62.5e-3
vcc = 5

-- Global signals
(o11, o12, o13) = (0,0,0)
(o21, o22, o23) = (0,0,0)
(off1, off2) = (0,0)

-- These shifters shift a fixed number of places, this is a kind of rounding
r = 0
shifter (x1,x2,x3,x4) = (shiftR x1 r, shiftR x2 r, shiftR x3 r, shiftR x4 r)
shifter1 (x1,x2,x3,x4) = (shiftR1 x1 r, shiftR1 x2 r, shiftR1 x3 r, shiftR1 x4 r)

-- A simple testbench command for n samples
testbench n = sampleN n $ expectedRoundedOutput (testBundler testInput)

-- Rounded expectations
expectedRoundedOutput :: Signal ( SFixed 8 56, SFixed 8 56, SFixed 8 56, SFixed 8 56) -> Signal
    Bool
expectedRoundedOutput = outputVerifier (map shifter expectedData)

-- For rounding you need to shift the contents of the output of the top entity.
testBundler input = bundle ( shifter1 ( topEntity (unbundle input) ) )

-- This tester just calculates the normal AMR-value.
amrNormal (vp1,vn1,vp2,vn2) = (vp1 - vn1, vp2 - vn2)

-- Top Entity
topEntity
  :: (Signal (UFixed 0 16),
      Signal (UFixed 0 16),
      Signal (UFixed 0 16),
      Signal (UFixed 0 16))
     -> (Signal (SFixed 8 56),
         Signal (SFixed 8 56),
         Signal (SFixed 8 56),
         Signal (SFixed 8 56))
topEntity = adcConv

--
bitCoerce1 :: (Bits a, Applicative f) => f a -> f Int -> f a
bitCoerce1 = liftA2 bitCoerce

-- Convert ADC-values
adcConv (vp1,vn1,vp2,vn2) = (amr1, amr2, off1, off2)
    where
```

```haskell
        -- Convert to right number of bits
        cvp1 = vp1 `times` (5 :: Signal (UFixed 7 9)) :: Signal (UFixed 7 25)
        cvn1 = vn1 `times` (5 :: Signal (UFixed 7 9)) :: Signal (UFixed 7 25)
        cvp2 = vp2 `times` (5 :: Signal (UFixed 7 9)) :: Signal (UFixed 7 25)
        cvn2 = vn2 `times` (5 :: Signal (UFixed 7 9)) :: Signal (UFixed 7 25)

        -- Convert to signed
        svp1 = bitCoerce1 (shiftR1 cvp1) :: Signal (SFixed 8 56)
        svn1 = bitCoerce1 (shiftR1 cvn1) :: Signal (SFixed 8 56)
        svp2 = bitCoerce1 (shiftR1 cvp2) :: Signal (SFixed 8 56)
        svn2 = bitCoerce1 (shiftR1 cvn2) :: Signal (SFixed 8 56)

        amr1 = svp1 - svn1
        amr2 = svp2 - svn2
        (amr1, amr2, off1, off2) = amrCalc (cvp1,cvn1,cvp2,cvn2)

-- Calculate AMR values
amrCalc (vp1,vn1,vp2,vn2) = (amr1, amr2, off1, off2)
    where
        -- Calculate output
        out1 = vp1 - vn1
        out2 = vp2 - vn2

        -- Calculate difference for comparison
        diff = max (abs (out1 - o11)) (abs (out2 - o21))

        -- Calculate AMR values
        (amr1, amr2, off1, off2) =
            muxT4 ( diff .>. filter_bound )
                ( reCalc (out1, out2) )
                ( out1 + off1, out2 + off2, register 0 off1, register 0 off2 )

-- Redo the calculation and filter the offset
reCalc
    :: (Signal (SFixed 8 56),
        Signal (SFixed 8 56))
        -> (Signal (SFixed 8 56),
            Signal (SFixed 8 56),
            Signal (SFixed 8 56),
            Signal (SFixed 8 56))
reCalc (out1, out2) = (amr1, amr2, off1, off2)
    where
        -- Shift the values
        (o11,o12,o13) = (out1, register 0 o11, register 0 o12)
        (o21,o22,o23) = (out2, register 0 o21, register 0 o22)

        -- Calculate the offset
        (offs1,offs2) = offCalc (o11,o12,o13) (o21,o22,o23)

        -- Filter the offset
        maxOffset  = (max (abs offs1) (abs offs2))
        boundCheck = maxOffset .>. 0 .&&. maxOffset .<. vcc

        off1 = mux boundCheck (offFilter off1 offs1) (register 0 off1)
        off2 = mux boundCheck (offFilter off2 offs2) (register 0 off2)

        -- Calculate the correct AMR values
        amr1 = out1 - off1
        amr2 = out2 - off2

-- Calculate offset values
offCalc (o11,o12,o13) (o21,o22,o23) = (offs1, offs2)
    where
        -- Calculate squares
        sqr0 = square o11 + square o21
        sqr1 = square o12 + square o22 - sqr0
        sqr2 = square o13 + square o23 - sqr0

        -- Calculate differential pairs
        x1 = o12 - o11
        x2 = o13 - o11
        x3 = o22 - o21
        x4 = o23 - o21

```

```
        -- Calculate the denominator
        denom = 2 * (x2 * x3 - x1 * x4)

        -- Calculate offset
        divzero = denom .==. 0
        offs1 = mux divzero 0 ( (x3 * sqr2 - x4 * sqr1)/denom )
        offs2 = mux divzero 0 ( (x2 * sqr1 - x1 * sqr2)/denom )

-- Filter the offset
offFilter offset offs = offset
    where
        offset' = ( offset * (filter_weight - 1) + offs )/filter_weight
        offset  = register 0 offset'

-- Calculate the square of a number
square x = x * x

-- Run mux for a 4-tuple
muxT4 z (ta,tb,tc,td) (fa,fb,fc,fd) =
    ( mux z ta fa ,
      mux z tb fb ,
      mux z tc fc ,
      mux z td fd )
```