Approximate multipliers for MAC

Verstoep, B. (s1009966)

April 18, 2018

Abstract

Approximate computing techniques reduce the cost (in terms of among others area and power consumption) of computing units in exchange for a reduced accuracy. These techniques are not optimized for Multiply-Accumulate (MAC) processing elements. This leaves a lot of room for improvement as the integrator part of a MAC allows for error balancing.
In this work, designs for an 8×8 bit MAC are sought that have optimal quality compared to their area cost for FPGA. To achieve this, different error balancing techniques are considered and combined with existing approximate computing techniques. An algorithm is proposed to perform an exhaustive search for the optimal designs, using an error balancing technique within a multiplier to achieve an average error close to 0. The designs found by the algorithm have a much higher quality compared to conventional approximate computing techniques for a small increase in area on the FPGA and the overall quality-cost tradeoff is improved.

Contents

Introduction

Multiply-Accumulate (MAC) circuits are a type of circuit that calculates the dot product of two input vectors. MAC circuits are widely used in many different applications. One use of MAC circuits is for example radio astronomy[1].
Figure 1 shows a diagram of a MAC processing element. The elements of the two input vectors are multiplied, and the results added together using an integrator. The output of the MAC is given in equation (1). Here O is the output of the MAC. M is the number of elements in the input vectors. A_n and B_n are the nth elements of the input vectors $\vec{A}$ and $\vec{B}$.

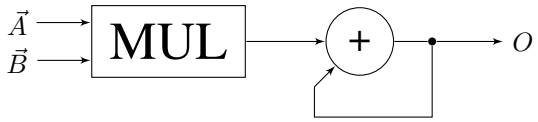


Figure 1: MAC processing element diagram

$$O = \vec{A} \cdot \vec{B} = \sum_{n=1}^{M} (A_n * B_n) \tag{1}$$

In this work the use of approximate computing techniques[2][3] is explored to reduce the cost, in terms of area for FPGA, while keeping the accuracy of the computation as high as possible. The multiplier of the MAC can be replaced by an approximate multiplier and the integrator can make use of approximate adders. In this work the adders will be kept accurate and the focus will lie on approximating the multipliers efficiently. The inputs are assumed to be uncorrelated. The goal is to find an approximate design of an 8 bit MAC processing element which has the lowest cost for a given quality or the best accuracy for a given cost. The cost considered is the area used on an FPGA.
In the first chapter, *Approximate multipliers*, a known method of creating approximate multipliers is discussed. Next in chapter 2 the difference the integrator part of a MAC operation makes for the approximate multiplier is explained and options to use these differences are explored. In chapter 3 a Matlab model is introduced to calculate the quality of a given design and a method of computing the area of the designs using Quartus is discussed. The 4th chapter explains the design space and an algorithm is proposed to explore it. In the final chapter, chapter 6, the algorithm is used to find designs and checked using the methods discussed in chapter 3. The results will be discussed and a few recommendations for future work are made.

Chapter 1

Approximate multipliers

In this chapter existing techniques for creating an approximate multiplier are introduced. Also the method to calculate the average error of a multiplier is discussed.

1.1 Creating a multiplier

An existing technique of creating approximate multipliers is to make a small and efficient 2×2 bit approximate multiplier and use multiple of them to create a larger $n \times n$ multiplier[4]. To create a 4×4 bit multiplier, the two 4 bit inputs, A and B, are divided into two 2 bit parts each. These are called A_H, A_L, B_H and B_L. The H indicates the most significant part of the inputs and L the least significant part. To calculate the 8 bit output of the 4×4 bit multiplier, $O_{4\times4}$, the input parts will first be multiplied using 2×2 bit multipliers. The 2 bit partial inputs from A are then multiplied with the 2 bit partial inputs of B in all possible combinations. The resulting four outputs are then shifted, where a more significant input means more shifting for the output. This process is shown in equation (1.1) and illustrated in Figure 1.1.

$$O_{4\times4} = 16A_HB_H + 4A_HB_L + 4A_LB_H + A_LB_L \tag{1.1}$$

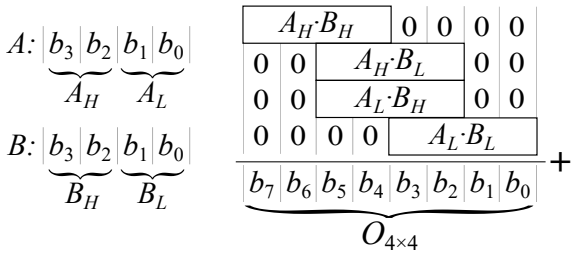


Figure 1.1: A 4×4 bit multiplier using 2×2 bit multiplier elements

This process can be repeated to create a 8×8 bit multiplier using four of the created 4×4 bit multipliers. This way the 8×8 bit multiplier is made up entirely of adders and 2×2 bit multipliers and can easily be made approximate by replacing some or all 2×2 elements with approximate versions. The equation of the 8×8 bit multiplier is shown as (1.2) and the diagram in Figure 1.2. This process can be repeated again to create larger multipliers. For each doubling of input bits the needed 2×2 bit multipliers is increased with

a factor of 4. For a $n \times n$ multiplier, where $n = 2^k$, there are 4^{k-1} of the 2×2 multipliers needed.

$$\begin{aligned} O_{8\times8} = {} & 4096A_{HH}B_{HH} + 1024(A_{HL}B_{HH} + A_{HH}B_{HL}) \\ & + 256(A_{HL}B_{HL} + A_{HH}B_{LH} + A_{LH}B_{HH}) \\ & + 64(A_{HH}B_{LL} + A_{HL}B_{LH} + A_{LH}B_{HL} + A_{LL}B_{HH}) \\ & + 16(A_{LL}B_{HL} + A_{HL}B_{LL} + A_{LH}B_{LH}) \\ & + 4(A_{LH}B_{LL} + A_{LL}B_{LH}) \\ & + A_{LL}B_{LL} \end{aligned} \tag{1.2}$$

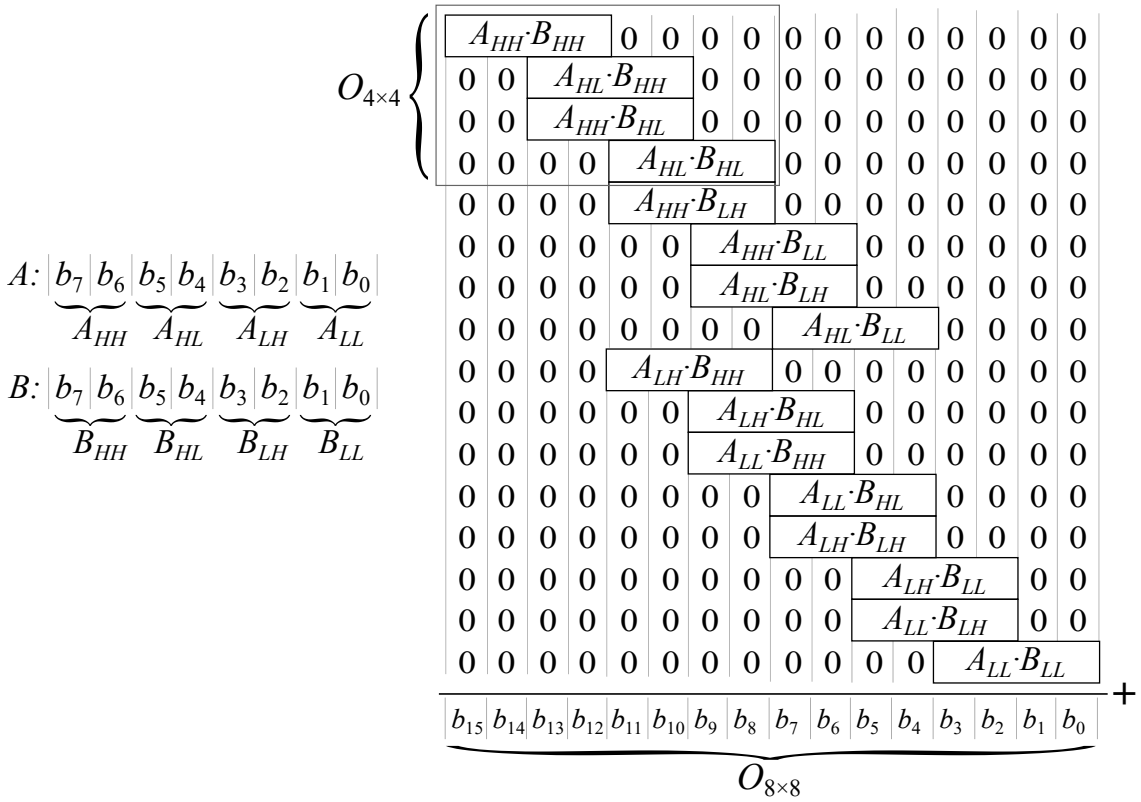


Figure 1.2: An 8×8 bit multiplier using 2×2 bit multiplier elements

1.2 Existing 2×2 bit multiplier elements

As discussed in section 1.1, to create an 8×8 bit approximate multiplier, 16 approximate 2×2 bit multiplier elements are needed. An accurate design of a 2×2 multiplier is shown in Figure 1.3 and the corresponding truth table in Table 1.1. In Figure 1.4 existing approximate designs[4][5] of 2×2 multipliers are shown and their corresponding truth tables in Table 1.2. The errors in the truth table are indicated by the coloured cell. The design in Figure 1.4(a) is the multiplier introduced in [4]. This design does not calculate the least significant bit and makes its output equal to the most significant bit. This creates a multiplier that has three errors with a magnitude of $+1$ as shown in the truth table 1.2(a). In Figure 1.4(b) the state of the art approximate design of [5] is shown. This design does not calculate the most significant bit, resulting in a much smaller multiplier having only a single error when calculating $3 * 3$. The multiplier then outputs 7 instead of 9. This means the multiplier has only one error with a magnitude of -2.

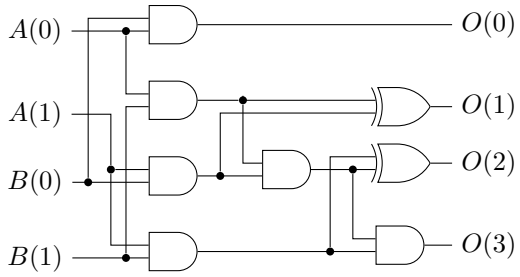
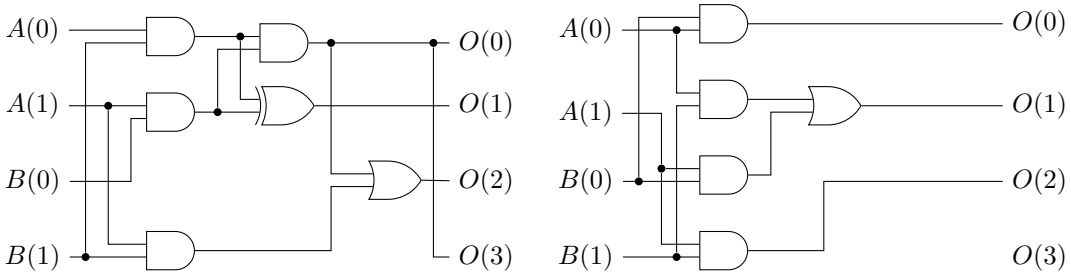


Figure 1.3: Accurate 2×2 bit multiplier design

Table 1.1: Accurate 2×2 multiplier design truth table

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1001



(a) $M1$ (b) $M2$

Figure 1.4: Approximate 2×2 bit Multiplier Designs

Table 1.2: Approximate 2×2 multiplier designs truth tables

(a) M1

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0000	0010	0010
10	0000	0010	0100	0110
11	0000	0010	0110	1001

(b) M2

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0111

1.3 Calculating the average error

A figure of the quality of an approximate multiplier can be the average error of the multiplier. To get the maximum quality, the average error should be as small as possible. The average error of the multiplier can be calculated by multiplying the probability of an error occurring with the weighted error magnitude. Here the weighted magnitude is the error magnitude of the 2×2 multiplier, multiplied with the shift due to the location of the 2×2 multiplier as shown in (1.2). Each of the 16 approximate 2×2 multipliers (for 8×8 bit) has its own error probability and weighted magnitude. For example the multiplier calculating $A_{HH} * B_{HH}$ using $M2$ has a much bigger weighted error magnitude ($|4096 * -2| = 8192$) than for example $A_{LL}B_{LL}$ ($|1 * -2| = 2$). The probability of the error occuring at each multiplier is dependent on the input distribution. For a uniform distribution the probability of each input is equally likely and therefore the probability for en error in every multiplier using $M2$ is $1/16$ which is the amount of errors divided by the number of (equally likely) options in the truth table in Figure 1.4(b). For other distributions, calculating the probability is much harder. For example with a normal distribution, if the probability for the highest numbers is much lower, the most significant bits of the input are more likely to be 0 and therefore the probability of the 2×2 bit calculation being $3 * 3$, where the error occurs for $M2$, is much lower.
To calculate the average error, the probability needs to be multiplied by the weighted magnitude of the error for each of the 16 multipliers and added up. This can be generalised for a $n \times n$ multiplier. This is shown in equation (1.3). Here $\overline{E}$ is the average error of the whole multiplier. S_i is the shift of the output of the 2×2 multiplier seen in equation (1.2). E_i is the error magnitude and $P(E)_i$ the probability of an error occuring for the ith 2×2 multiplier.

$$\overline{E} = \sum_{i=1}^{4^{k-1}} (S_i * |E_i| * P(E)_i) \tag{1.3}$$

Chapter 2

Approximate multipliers for MAC

There is a distinct difference between creating an approximate multiplier for a MAC as opposed to just an approximate multiplier in general. When calculating the outcome of a multiplication every result counts. It does not matter if the errors made are sometimes negative and sometimes positive. For a MAC however the multiplier gets followed up by an integrator which sums all the results of the multiplier. The individual multiplications do not matter as much as the end result of the addition. If in the multiplications sometimes a negative error is made and sometimes a positive error, the errors add up in the integrator and compensate eachother, resulting in a lower error of the total MAC operation.

2.1 Average error for MAC

To calculate the average error of the MAC, equations (1) and (2.1) are used to create equation (2.2). Note that equation (2.1) is a slight variation on equation (1.3). This is because for the calculation of the average error of a MAC the sign of the error does matter. Therefore the absolute operation is removed and the new variable is called $\overline{E'}$

$$O = \vec{A} \cdot \vec{B} = \sum_{n=1}^{M} (A_n * B_n) \tag{1 revisited}$$

$$\overline{E'} = \sum_{i=1}^{4^{k-1}} (S_i * E_i * P(E)_i) \tag{2.1}$$

$$\begin{aligned} \overline{E}_{MAC} &= \left| \sum_{n=1}^{M} \overline{E'} \right| \\ &= \sum_{n=1}^{M} \left| \sum_{i=1}^{4^{k-1}} (S_i * E_i * P(E)_i) \right| \\ &= M \left| \sum_{i=1}^{4^{k-1}} (S_i * E_i * P(E)_i) \right| \end{aligned} \tag{2.2}$$

Because the errors may cancel each other, the absolute value is taken after the addition of the errors of each of the 2×2 multipliers instead of before addition.

2.2 Error balancing methods

To get the best quality the average error should be as low as possible. This can be done in a couple of ways. One way is to balance a single 8×8 bit multiplier using a combination of different 2×2 bit elements.

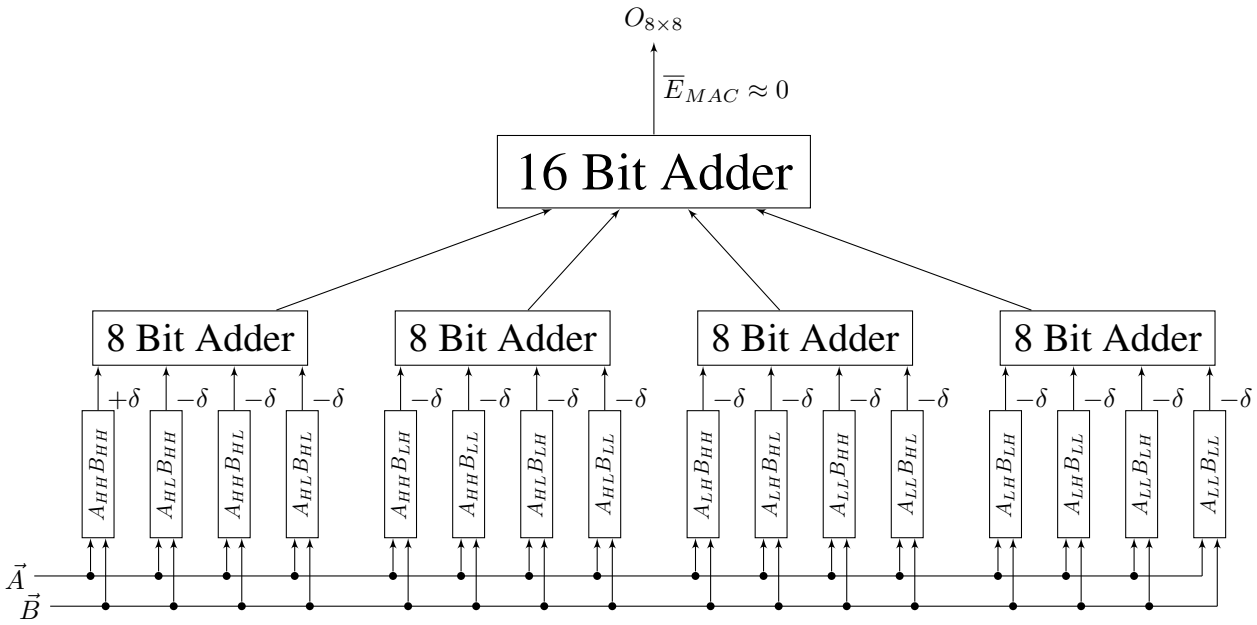
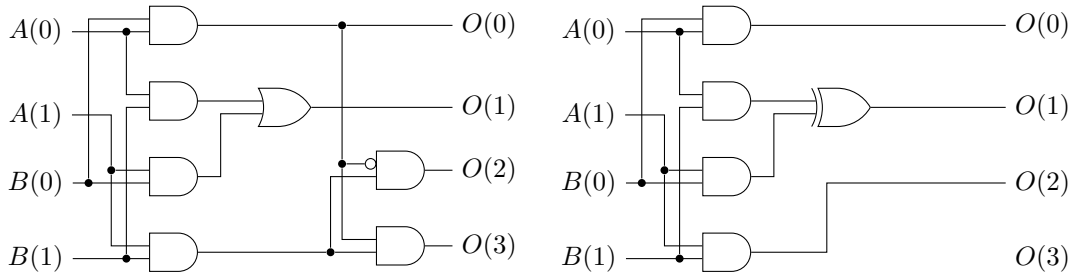


Figure 2.1: Internal error balancing of an 8×8 bit multiplier



(a) $M3$ (b) $M4$

Figure 2.2: 2×2 bit Multiplier Designs for error balancing purposes

An example is shown in Figure 2.1. The $+\delta$ and $-\delta$ are the errors of the 2×2 multiplier elements. $+\delta$ indicates an overall positive error and $-\delta$ a negative error. For the purpose of creating a balanced multiplier two new 2×2 multipliers are introduced in Figure 2.2. Their truth tables can be found in Table 2.1.

$M3$ in Figure 2.2(a) is a multiplier made to directly balance $M2$. It has the same error probability but the opposite error magnitude. To more precisely balance the multiplier to get an average error closer to 0, $M4$ is introduced. The only difference between this multiplier and $M2$ is that an OR-gate is replaced by an XOR-gate resulting in a larger error and simular area for FPGA as will be shown in chapter 3. These multipliers can used in conjunction with the ones introduced in chapter 1 to create a single set of 16 multipliers creating both negative and positive errors which cancel each other out as close to 0 as possible.

Another way of reducing the average error is to work with a *mirror pair*. For example, two multipliers with the same error probability and magnitude but opposing signs, like $M2$ and $M3$, can be used to create two 8×8 bit multipliers. When the output of these multipliers are added up as shown in Figure 2.3 the average errors add up to become exactly 0. This does double the area requirements, as it uses two multipliers as well as additional adders but it also doubles the throughput and therefore is acceptable in a lot of cases.

These two methods can also be combined. A design which is internally balanced towards a positive

Table 2.1: Approximate 2×2 multiplier designs truth tables

(a) $M3$

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1011

(b) $M4$

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0101

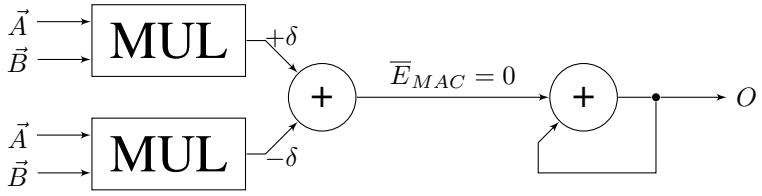


Figure 2.3: Two 8×8 bit multipliers used as mirror pair in a MAC

error of A can be *mirrored* with the second method, using a design balanced towards $-A$. For this work however the focus will be on balancing a single multiplier towards an average error of 0.

Chapter 3

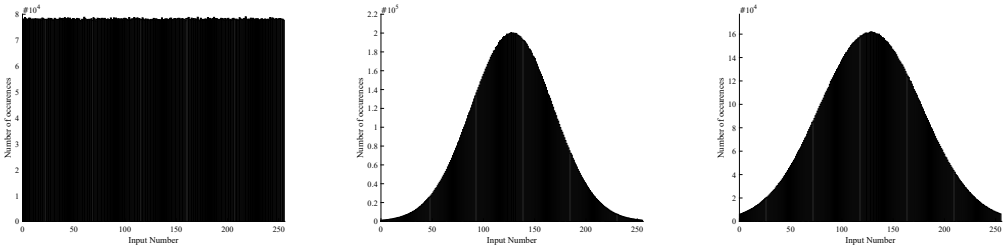
Quality and computational cost analysis

In this chapter the methods of calculating the quality and computational cost of the designs is discussed. The quality is calculated using a Matlab model and the computational cost is calculated using Quartus.

3.1 Matlab model of a MAC

The code for the Matlab model of a MAC can be found in Appendix A. The model calculates the accurate and the approximate outcomes of a generated set of inputs.
Three sets of random inputs with different input distributions are generated using Matlab. The inputs range from 0 to 255 (8 bit). The input vector size of the MAC, M, will be chosen as 10000 and the result of the MAC will be computed 1000 times. One input set is a uniform distribution, generated using the Matlab function $randi$. The other two sets are normal distributions with an average of 128 and a standard deviation of 40 and 50 respectively. The resulting distributions are shown in the histograms in Figure 3.1.

The accurate result of the MAC is calculated using the built-in dot function of Matlab. The approximate version is calculated by first separating the 8 bit inputs into the 2 bit inputs of each of the 16 2×2 bit multiplier elements. Next, the accurate products of those 2 bit inputs are calculated. Dependent on which multiplier is used for which of the inputs, the 4 bit outputs are adjusted to include the errors. For example, for multiplier $M2$ every 9 in the output is replaced with a 7. The results are summed using equation (1.2) from chapter 2 to get the output of the total approximate multiplier and finally summed to get the result of the MAC.



(a) Uniform Distribution (b) Normal Distribution $\sigma = 40$ (c) Normal Distribution $\sigma = 50$

Figure 3.1: Histrograms of the generated inputs

Table 3.1: MSE and MAE for different distributions for 8×8 MAC with a single type of 2×2 multiplier each

Distribution / Multiplier	Uniform		Normal $\sigma = 40$		Normal $\sigma = 50$	
	MSE	MPE	MSE	MPE	MSE	MPE
Accurate	0.00	0.00%	0.00	0.00%	0.00	0.00%
$M1$	$1.83 * 10^{14}$	8.33%	$2.95 * 10^{14}$	10.48%	$2.79 * 10^{14}$	10.21%
$M2$	$8.16 * 10^{13}$	5.55%	$2.41 * 10^{12}$	0.95%	$6.94 * 10^{12}$	1.61%
$M3$	$8.16 * 10^{13}$	5.55%	$2.41 * 10^{12}$	0.95%	$6.94 * 10^{12}$	1.61%
$M4$	$3.26 * 10^{14}$	11.1%	$9.65 * 10^{12}$	1.89%	$2.78 * 10^{13}$	3.22%

3.2 Quality analysis using the Matlab model

The resulting MAC outputs are compared to get a figure of quality. A commenly used metric of quality is the Mean Square Error[6][7]. The Mean Square Error (MSE) is calculated by calculating the square of the difference (or error) between each of the 1000 accurate and approximate MAC results. That result is divided by the total amount of MAC results, in this case 1000, to get the mean. This is shown in equation (3.1). Here α is the result of the accurate MAC calculation and β the result of the approximate. n is the amount of calculations.

$$\text{MSE} = \frac{(\alpha_1 - \beta_1)^2 + (\alpha_2 - \beta_2)^2 \ldots + (\alpha_n - \beta_n)^2}{n} \tag{3.1}$$

The MSE can be used to compare different designs with eachother, but the values for MSE do not mean much on their own. The values are dependent on the actual outcome of the MAC and since we have a large input vector ($M = 10000$) the values for MSE will become very large. To get a better idea of the actual meaning of the error, a second metric is used. The Mean Percentage Error (MPE) is a relative error calculated as shown in equation (3.2). Instead of calculating the square of the error, the absolute value is taken and is divided by the accurate result to get a relative indication of the error.

$$\text{MPE} = 100\frac{\frac{|\alpha_1-\beta_1|}{\alpha_1} + \frac{|\alpha_2-\beta_2|}{\alpha_2} \ldots + \frac{|\alpha_n-\beta_n|}{\alpha_n}}{n} \tag{3.2}$$

A few examples of resulting values for MSE and MPE are shown in Table 3.1. These are the values for MSE and MPE for each of the input sets when all sixteen 2×2 bit elements are the same. The MSE and MPE values for $M2$ and $M3$ are identical as expected since they are a mirror pair where the only difference is the sign of the error. The error of $M4$ is relatively big. This does not matter as it is not made to be a multiplier on its own but rather to compensate the positive errors of other multipliers.

Table 3.2: Area Cost of 2×2 bit elements and MAC using a single type of 2×2 bit element

Multiplier Used	Area 2×2 [LE]	Area MAC [LE]
Accurate	4	174
$M1$	3	166
$M2$	3	136
$M3$	4	175
$M4$	3	136

3.3 Cost analysis for FPGA using Quartus

To calculate the area cost of the designs on an FPGA, Quartus is used. The used VHDL code can be seen in appendix B. Quartus is used for synthesis for FPGA. An area cost is expressed for the designs as the number of Logic Elements (LE) used in the FPGA. In appendix C the register transfer level (RTL) view of the synthesis of the MAC is shown. Table 3.2 shows the computed area of the individual 2×2 bit elements and the complete MAC made using only a single type of 2×2 bit multiplier each.

In Table 3.2 the cost result for $M3$ stands out as it uses a larger area than the accurate one. Since this multiplier makes large positive errors, the output does not always fit within 16 bits but will overflow into a 17th bit. This overflow also happens with the intermediate 4×4 bit calculations in the multiplier. Larger adders are needed to account for this which makes the multiplier a lot bigger. Not all cases allow for a 17th bit to be output. This makes a multiplier made solely of $M3$ elements inefficient. The $M3$ 2×2 multiplier is only used for partial products where it does not result in overflow.

Chapter 4

Design space exploration of approximate multipliers for MAC

In this chapter the complexity of the design space for approximate multipliers for a MAC operation is explained. Then an algorithm to explore this design space is proposed and discussed.

4.1 Complexity of the design space

For an 8×8 bit multiplier sixteen 2×2 bit multipliers are needed. This means that even with only a few options of 2×2 bit multipliers the design space to explore gets large really fast. For example when only using three different 2×2 bit elements the number of possible designs (permutations with repetition) is already $3^{16} = 43046721$.

4.2 Algorithm for design space exploration

To explore this design space an algorithm (Appendix D) is proposed. The algorithm computes the average error of each of the designs and estimates the cost. The cost and error of each of the designs are compared and the optimal designs are chosen. A flowchart of the algorithm can be seen in Figure 4.1.

Input

The algorithm has 3 inputs: Input data for a MAC in the wanted distribution, the error magnitudes and cost estimations for each of the 2×2 bit multipliers.

Error probability computation

Using the input data the probability of an error occurring is calculated. The algorithm only includes $M2$, $M3$ and $M4$ of the aforementioned multipliers which means the probability of an error occurring in the 2×2 bit multiplier is always equal to the probability the inputs of that multiplier are both 3. The probability of each of the inputs being 3 is computed with equation (4.1). The probability the input for the given 2×2 bit multiplier is 3 is the amount of times it was 3 in the distribution sample ($M_{A=3}$) divided by the total amount of generated numbers (M_{total}). Then to get the probability of an error occurring for each multiplier the correct input probabilities are multiplied as shown in equation (4.2). This is done for each of the multipliers and a vector containing the 16 values for the error probability is output to the next step.

$$P(A = 3) = \frac{M_{A=3}}{M_{total}} \tag{4.1}$$

Figure 4.1: Flowchart of the Design Space Exploration algorithm

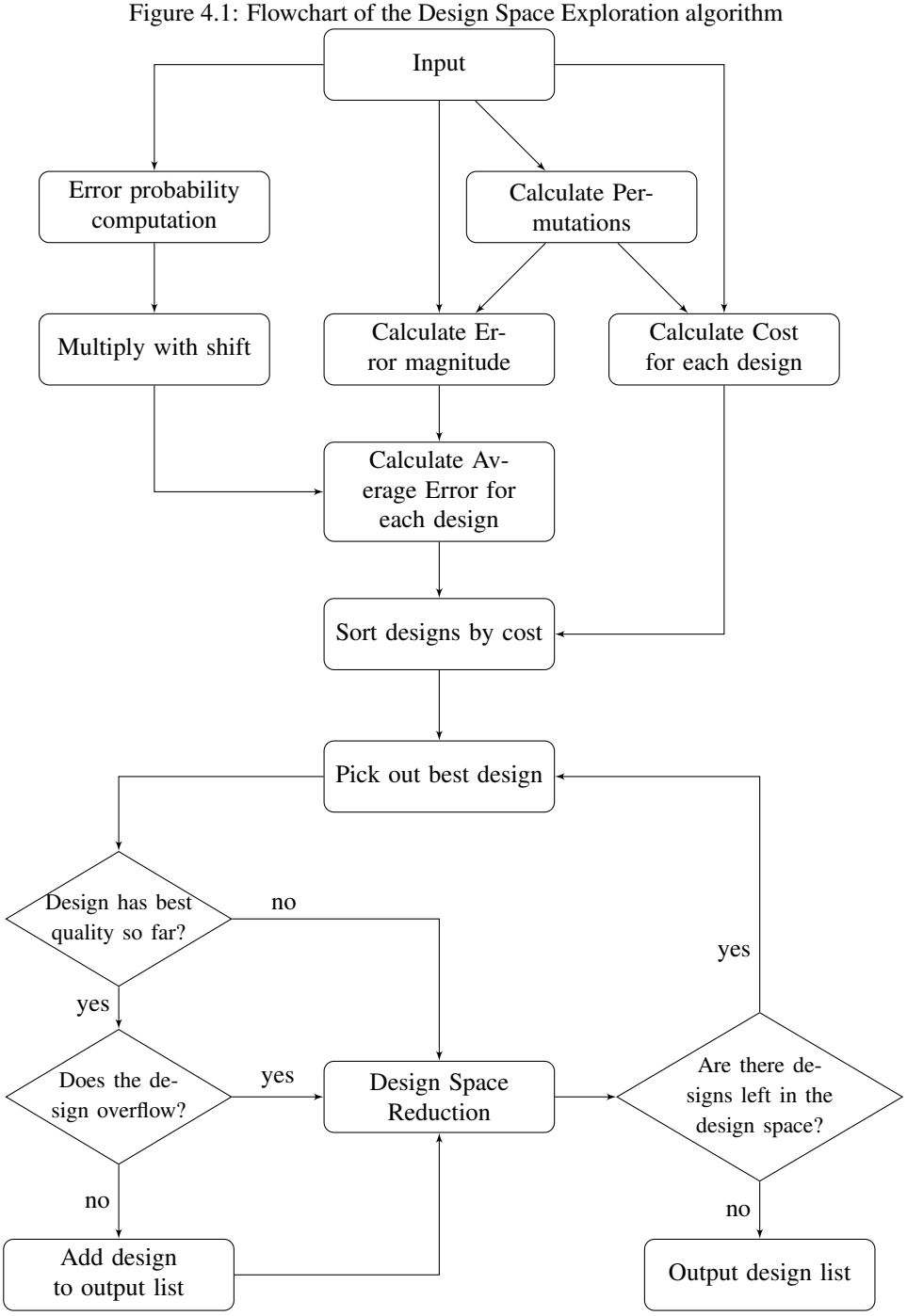


Table 4.1: Example of a few sets of permutations

design	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X1$	$M2$	$M2$	$M2$	$M2$	$M3$	$M4$	$M2$	$M2$	$M3$	$M2$	$M2$	$M2$	$M3$	$M2$	$M3$	$M2$
$X2$	$M2$	$M2$	$M2$	$M2$	$M3$	$M4$	$M2$	$M2$	$M3$	$M2$	$M2$	$M2$	$M3$	$M2$	$M3$	$M3$
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..

Table 4.2: Example of a few sets of error magnitudes

design	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X1$	-2	-2	-2	-2	+2	-4	-2	-2	+2	-2	-2	-2	+2	-2	+2	-2
$X2$	-2	-2	-2	-2	+2	-4	-2	-2	+2	-2	-2	-2	+2	-2	+2	+2
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..

$$P(A = 3 \text{ and } B = 3) = \frac{M_{A=3} * M_{B=3}}{M_{total}^2} \tag{4.2}$$

Multiply with shift

The 16 probability values are multiplied with the needed shift for the outputs of the 2×2 multipliers shown in equation (1.2).The shifted probability values, $S_i * P(E)_i$ in equation (2.1), are the output.

Calculate Permutations

The number of different 2×2 bit multipliers is used to generate all different design permutations. In the current configuration it calculates for 3 different multipliers. They are permutations with repetition which results as mentioned in $3^{16} = 43046721$ different designs. This block outputs 43 million sets of 16 numbers representing each multiplier in each design. Table 4.1 shows an example of a few of those 43 million sets. Here is $X1$ the index of the designs and the numbers in the top row represent the sixteen 2×2 multiplier locations in the MAC.

Calculate Error magnitude

The numbers representing the multipliers are replaced with the error magnitude of each of the multipliers resulting in 43 million sets of 16 error magnitudes. Table 4.2 shows an example of a few of those 43 million sets.

Calculate Average Error for each design

Each set of 16 error magnitudes is multiplied with the shifted probability ($S_i * P(E)_i$) to get the average error each of the multipliers contributes to the whole 8×8 bit multiplier. These are then added up to get the average error of the whole multiplier. The output is a vector with an average error for each of the designs.

Calculate Cost for each design

The generated numbers representing all design permutations are replaced with the estimations for cost. These costs are added up to get an estimated cost for each of the 43 million designs.

Sort designs by cost

With two lists available, one with the average error for all designs and one with all costs, the optimal designs need to be picked out. To do this first both lists are sorted based on the costs. The output of this block contains the sorted lists of the average error and cost of the designs.

Pick out best design

From the sorted lists the designs with both the lowest cost and best quality (lowest average error) are taken and output to be used in the next steps.

Design has best quality so far?

In this block a check is done if the chosen optimal designs have the best quality so far. The algorithm checks the design space in order, from lowest to the highest cost. If the new design has a lower quality it means that both the cost is higher and the quality worse than it's predecessors and it can be removed from the design space.

Does the design overflow?

As discussed in chapter 3, the $M3$ multiplier makes positive errors which can cause the output to exceed 16 bits. This can also happen with the designs containing some $M3$ multipliers. This is not wanted and these designs will be removed from the design space. Normally the overflow can be checked by calculating $255 * 255$ for this is the largest number and will contain all the positive errors. However because $M4$ has such a large error, the biggest number is actually $2 * 3 = 6$ instead of $3 * 3 = 5$. This means there is a chance the multiplier will not overflow calculating $255 * 255$ but will overflow calculating a lower sum. This makes checking for overflow a lot more complicated. It can be checked by just checking all possible 8×8 multiplications. However when this has to be done for a lot of designs it will take a lot of time. To speed up the algorithm, a few logic steps are done first, specific to the multipliers used in this work. For example the 4×4 bit multiplications will never overflow if the most significant multiplier is not the $M3$ multiplier. The other logic steps can be seen in the algorithm in appendix D. The last few designs that did not get filtered out using these logic steps are tested by calculating all possibilities.

Add design to output list

When the designs do not overflow and have the best quality so far they are added to the output list of the algorithm.

Design Space Reduction

If the design overflows, that specific design will be removed from the design space. Otherwise all designs with the same cost as that design will be removed from the design space.

Are there designs left in the design space?

If there are designs left in the design space the algorithm loops back to find the optimal designs again. Otherwise, the algorithm outputs the sorted design list containing designs with ascending cost and quality. The designs have the lowest cost for each quality and the highest quality for each cost.

Chapter 5

Results

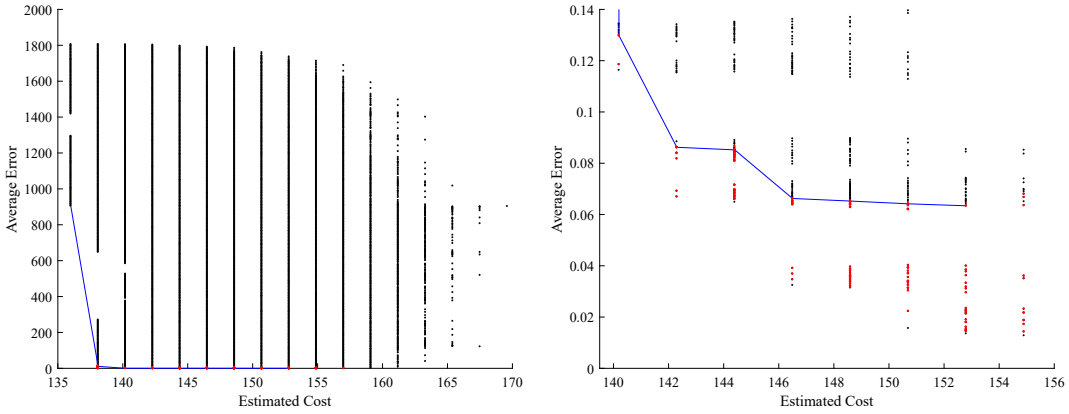
In this chapter the design space exploration algorithm is run and the results are discussed. Then a few reccomendations for future work are made.

5.1 Results of design space exploration

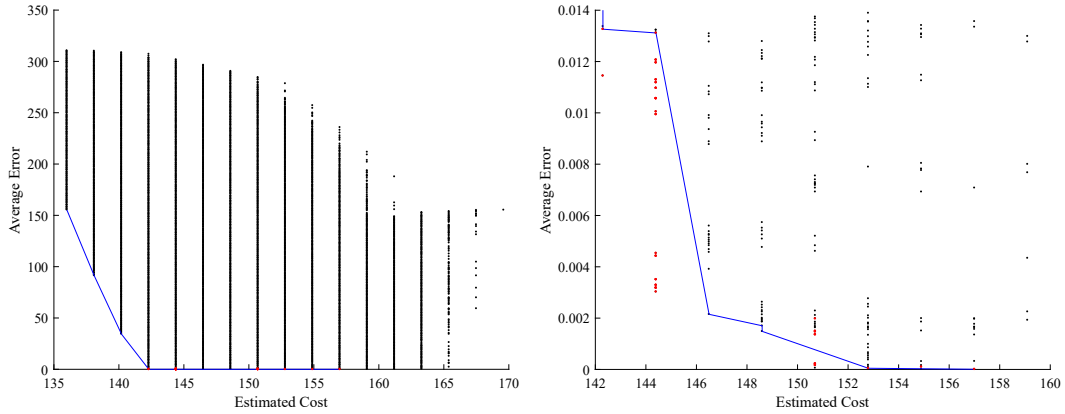
The algorithm discussed in the last chapter was used to find the optimal designs using the 2×2 bit elements $M2$, $M3$ and $M4$. The corresponding error magnitudes are -2, $+2$ and -4 respectively. The estimations of the costs are based on the values in Table 3.2 in chapter 3. The values are $\frac{136}{16} = 8.5, \frac{170}{16} = 10.6$ and $\frac{136}{16} = 8.5$. Here the value for the $M3$ multiplier differs from the one gained in the cost chapter because the 17th bit is not taken into account. The algorithm removes designs with overflow so this will not be a problem. The algorithm was run for the uniform distribution, a normal distribution where $\sigma = 40$ and a normal distribution with $\sigma = 50$. The results are shown in Figure 5.1. The left side shows the total explored design space and the right side a zoomed version that is focused on the part with the lowest average errors. The black dots represent the $43 * 10^6$ designs and their calculated average errors and cost estimations. The red dots represent the designs removed by the overflow handling of the algorithm. Finally, the blue line connects the chosen designs with optimal average error for each cost.

The resulting optimal designs were checked with the methods discussed in chapter 3. The results can be found in Figure 5.2.

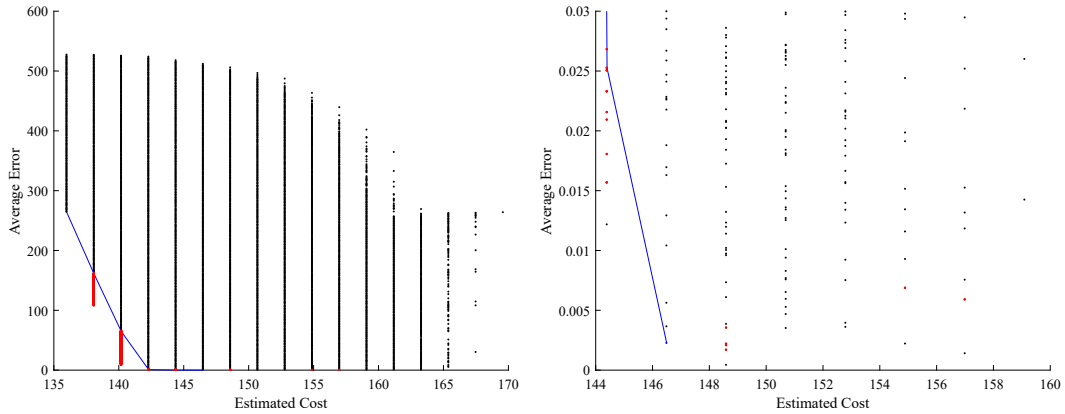
The designs found by the algorithm can be seen in Table 5.1 and the corresponding cost and quality values are shown in Table 5.2. The cost and quality are in ascending order.



(a) Uniform Distribution (left: Total design space, right: Zoomed version at low average errors)

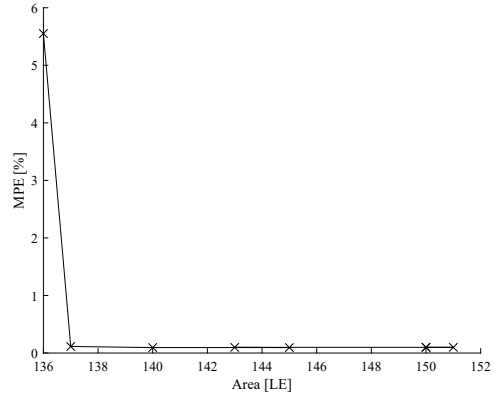


(b) Normal Distribution $\sigma = 40$ (left: Total design space, right: Zoomed version at low average errors)

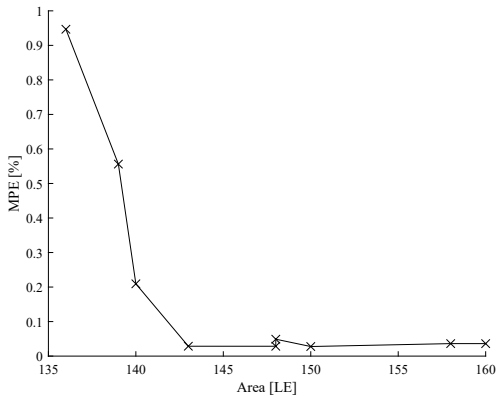


(c) Normal Distribution $\sigma = 50$ (left: Total design space, right: Zoomed version at low average errors)

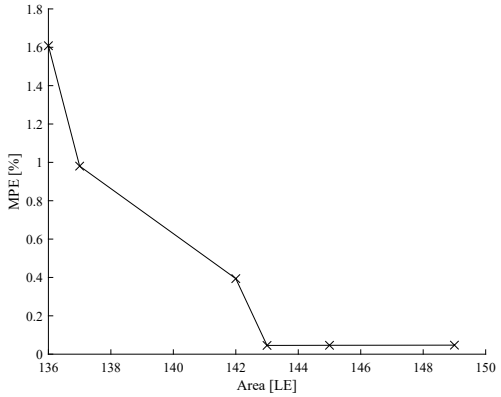
Figure 5.1: Design space exploration results



(a) Uniform Distribution



(b) Normal Distribution $\sigma = 40$



(c) Normal Distribution $\sigma = 50$

Figure 5.2: Cost and Quality of optimal designs found by the algorithm

Table 5.1: Found Designs

(a) Uniform Distribution

designs	$D1$	$D2$	$D3$	$D4$	$D5$	$D6$	$D7$	$D8$	$D9$
$A_{HH}B_{HH}$	$M2$	$M3$	$M3$	$M3$	$M3$	$M3$	$M3$	$M3$	$M3$
$A_{HL}B_{HH}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M4$	$M4$	$M4$	$M4$
$A_{HH}B_{HL}$	$M2$	$M4$	$M4$	$M4$	$M4$	$M4$	$M4$	$M4$	$M4$
$A_{HL}B_{HL}$	$M2$	$M2$	$M3$	$M2$	$M2$	$M3$	$M3$	$M3$	$M3$
$A_{HH}B_{LH}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M2$	$M2$	$M2$	$M3$
$A_{HH}B_{LL}$	$M2$	$M2$	$M4$	$M2$	$M2$	$M2$	$M2$	$M3$	$M2$
$A_{HL}B_{LH}$	$M2$	$M2$	$M2$	$M3$	$M3$	$M2$	$M2$	$M4$	$M4$
$A_{HL}B_{LL}$	$M2$	$M2$	$M2$	$M2$	$M4$	$M2$	$M4$	$M3$	$M4$
$A_{LH}B_{HH}$	$M2$	$M2$	$M4$	$M2$	$M2$	$M3$	$M3$	$M3$	$M2$
$A_{LH}B_{HL}$	$M2$	$M2$	$M4$	$M4$	$M4$	$M4$	$M4$	$M4$	$M4$
$A_{LL}B_{HH}$	$M2$	$M2$	$M4$	$M2$	$M2$	$M3$	$M3$	$M2$	$M3$
$A_{LL}B_{HL}$	$M2$	$M4$	$M2$	$M2$	$M4$	$M2$	$M4$	$M2$	$M3$
$A_{LH}B_{LH}$	$M2$	$M4$	$M2$	$M2$	$M3$	$M2$	$M3$	$M3$	$M3$
$A_{LH}B_{LL}$	$M2$	$M2$	$M4$	$M2$	$M4$	$M4$	$M4$	$M4$	$M3$
$A_{LL}B_{LH}$	$M2$	$M2$	$M4$	$M2$	$M4$	$M4$	$M4$	$M4$	$M2$
$A_{LL}B_{LL}$	$M2$	$M2$	$M2$	$M3$	$M3$	$M3$	$M3$	$M3$	$M3$

(b) Normal Distribution $\sigma = 40$

designs	$D1$	$D2$	$D3$	$D4$	$D5$	$D6$	$D7$	$D8$	$D9$
$A_{HH}B_{HH}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M4$	$M2$	$M2$	$M2$
$A_{HL}B_{HH}$	$M2$	$M2$	$M2$	$M3$	$M3$	$M3$	$M2$	$M3$	$M3$
$A_{HH}B_{HL}$	$M2$	$M2$	$M3$	$M3$	$M3$	$M3$	$M3$	$M4$	$M4$
$A_{HL}B_{HL}$	$M2$	$M3$	$M3$	$M3$	$M3$	$M3$	$M3$	$M2$	$M3$
$A_{HH}B_{LH}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M2$	$M3$	$M3$	$M3$
$A_{HH}B_{LL}$	$M2$	$M2$	$M2$	$M4$	$M4$	$M2$	$M3$	$M2$	$M2$
$A_{HL}B_{LH}$	$M2$	$M2$	$M2$	$M4$	$M4$	$M3$	$M2$	$M3$	$M3$
$A_{HL}B_{LL}$	$M2$	$M2$	$M2$	$M2$	$M4$	$M4$	$M3$	$M4$	$M4$
$A_{LH}B_{HH}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M4$	$M3$	$M3$	$M3$
$A_{LH}B_{HL}$	$M2$	$M2$	$M2$	$M4$	$M4$	$M2$	$M2$	$M3$	$M3$
$A_{LL}B_{HH}$	$M2$	$M2$	$M2$	$M4$	$M4$	$M2$	$M4$	$M2$	$M4$
$A_{LL}B_{HL}$	$M2$	$M2$	$M2$	$M4$	$M4$	$M4$	$M2$	$M3$	$M3$
$A_{LH}B_{LH}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M4$	$M2$	$M2$	$M4$
$A_{LH}B_{LL}$	$M2$	$M2$	$M2$	$M4$	$M2$	$M2$	$M2$	$M2$	$M3$
$A_{LL}B_{LH}$	$M2$	$M2$	$M2$	$M4$	$M3$	$M4$	$M2$	$M2$	$M3$
$A_{LL}B_{LL}$	$M2$	$M2$	$M2$	$M4$	$M4$	$M3$	$M2$	$M2$	$M3$

(c) Normal Distribution $\sigma = 50$

designs	$D1$	$D2$	$D3$	$D4$	$D5$	$D6$
$A_{HH}B_{HH}$	$M2$	$M3$	$M2$	$M2$	$M2$	$M2$
$A_{HL}B_{HH}$	$M2$	$M4$	$M3$	$M3$	$M3$	$M3$
$A_{HH}B_{HL}$	$M2$	$M2$	$M3$	$M3$	$M3$	$M3$
$A_{HL}B_{HL}$	$M2$	$M2$	$M2$	$M3$	$M3$	$M3$
$A_{HH}B_{LH}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M4$
$A_{HH}B_{LL}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M2$
$A_{HL}B_{LH}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M3$
$A_{HL}B_{LL}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M2$
$A_{LH}B_{HH}$	$M2$	$M2$	$M2$	$M2$	$M4$	$M4$
$A_{LH}B_{HL}$	$M2$	$M2$	$M2$	$M2$	$M3$	$M3$
$A_{LL}B_{HH}$	$M2$	$M4$	$M2$	$M2$	$M2$	$M4$
$A_{LL}B_{HL}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M2$
$A_{LH}B_{LH}$	$M2$	$M2$	$M2$	$M2$	$M4$	$M4$
$A_{LH}B_{LL}$	$M2$	$M2$	$M2$	$M2$	$M2$	$M4$
$A_{LL}B_{LH}$	$M2$	$M2$	$M2$	$M2$	$M4$	$M4$
$A_{LL}B_{LL}$	$M2$	$M2$	$M2$	$M2$	$M4$	$M2$

Table 5.2: Design results

(a) Uniform Distribution

designs	Average Error	Estimated Cost [LE]	MSE	MPE	Cost [LE]
$D1$	$9.03 * 10^{2}$	136	$8.16 * 10^{13}$	5.55%	136
$D2$	$1.12 * 10^{1}$	138.1	$5.35 * 10^{10}$	0.11%	137
$D3$	$1.30 * 10^{-1}$	140.2	$3.80 * 10^{10}$	0.10%	140
$D4$	$8.62 * 10^{-2}$	142.3	$3.79 * 10^{10}$	0.09%	145
$D5$	$8.52 * 10^{-2}$	144.4	$3.79 * 10^{10}$	0.10%	143
$D6$	$6.63 * 10^{-2}$	146.5	$4.07 * 10^{10}$	0.10%	150
$D7$	$6.52 * 10^{-2}$	148.6	$4.08 * 10^{10}$	0.10%	151
$D8$	$6.42 * 10^{-2}$	150.7	$4.09 * 10^{10}$	0.10%	150
$D9$	$6.34 * 10^{-2}$	152.8	$4.08 * 10^{10}$	0.10%	153

(b) Normal Distribution $\sigma = 40$

designs	Average Error	Estimated Cost [LE]	MSE	MPE	Cost [LE]
$D1$	$1.55 * 10^{2}$	136	$2.41 * 10^{12}$	0.95%	136
$D2$	$9.12 * 10^{1}$	138.1	$8.34 * 10^{11}$	0.56%	139
$D3$	$3.43 * 10^{1}$	140.2	$1.21 * 10^{11}$	0.21%	140
$D4$	$1.33 * 10^{-2}$	142.3	$3.41 * 10^{9}$	0.03%	143
$D5$	$1.31 * 10^{-2}$	144.4	$3.40 * 10^{9}$	0.03%	148
$D6$	$2.15 * 10^{-3}$	146.5	$1.00 * 10^{10}$	0.05%	148
$D7$	$1.70 * 10^{-3}$	148.6	$3.29 * 10^{9}$	0.03%	150
$D8$	$4.56 * 10^{-5}$	152.8	$5.43 * 10^{9}$	0.04%	158
$D9$	$7.60 * 10^{-6}$	157.0	$5.42 * 10^{9}$	0.04%	160

(c) Normal Distribution $\sigma = 50$

designs	Average Error	Estimated Cost [LE]	MSE	MPE	Cost [LE]
$D1$	$2.63 * 10^{2}$	136	$6.94 * 10^{12}$	1.61%	136
$D2$	$1.61 * 10^{2}$	138.1	$2.59 * 10^{12}$	0.98%	137
$D3$	$6.44 * 10^{1}$	140.2	$4.24 * 10^{11}$	0.39%	142
$D4$	$8.18 * 10^{-1}$	142.3	$8.55 * 10^{9}$	0.05%	143
$D5$	$2.52 * 10^{-2}$	144.4	$8.77 * 10^{9}$	0.05%	145
$D6$	$2.25 * 10^{-3}$	146.5	$9.12 * 10^{9}$	0.05%	149

5.2 Conclusion and discussion

The goal of this work is to find an approximate design of an 8 bit MAC which has the lowest cost for a given quality or the best accuracy for a given cost. The algorithm achieves this and outputs the optimal designs for each cost. Compared to conventional multipliers like $M1$ and $M2$, the designs found by the algorithm have a much lower error for a small increase in cost and the overall quality-cost tradeoff is improved. For example, the uniform distribution has a value of MPE that is 5.44% lower for an increase of only 1 logic element in area ($D2$ in Table 5.2(a)) over the multiplier made with 2×2 bit element $M2$ (Table 3.1 and Table 3.2).
When taking a closer look at the results and comparing the results from the algorithm with the result of the Quality and Cost Analysis tests, the algorithm seems to come reasonably close with its estimations. When looking at the zoomed version of the normal distribution with $\sigma = 50$ in Figure 5.1(c) the line connecting the chosen optimal designs suddenly stops. This means the algorithm did not find any designs with lower average error at a higher cost.
The values of MPE and MSE in Table 5.2 are mostly in descending order since the designs are sorted on the calculated average error of the algorithm. There are a few values that stand out however, for they are not in order like the rest. For example $D4$ of the uniform designs has the lowest error according to the Quality Analysis while not at the lowest spot. In the same way $D6$ from the normal distribution with $\sigma = 40$ has a higher error. The error is really small however and these differences are most likely the result of estimation in the algorithm. Also the inputs used to test the quality are random and therefore the errors not always perfectly cancel eachother.
When comparing the estimated and actual cost of the designs the values seem to come close. However a important part of the goal is that the actual cost should be in ascending order like the estimated costs. When looking at $D5$ and $D8$ of the uniform distribution it can be seen that this is not always the case. They have a lower cost than the designs $D4$ and $D7$ respectively and are therefore (theoretically) objectively better designs as they have a lower error and lower cost. This indicates that the area is not only dependent on the used multipliers and that the estimation based on this assumption is not accurate enough. The difference in cost is most likely caused by the fact that $M2$ and $M4$ only output 3 bits instead of 4. This can change the size of the adders needed within each 4×4 bit multiplier segment depending on the location of $M2$ or $M4$. This makes the total area of the MAC differ between two multiplier designs using the same multipliers but in a different configuration. The algorithm is reasonably fast and completes a single run in a few minutes. When the algorithm is adjusted to take into account 4 instead of 3 different multipliers the needed memory to run the algorithm increases exponentially. This is the result of the algorithm calculating the quality and cost for every permutation of which there are $4^{16} = 4.3$ billion for 4 multipliers instead of the 43 million with 3 multipliers. The algorithm is therefore not suited for a large variety of 2×2 multipliers.

5.3 Future work

- As discussed in the *Discussion*, the cost estimation is inaccurate. The effect of the configuration of the multipliers on the area should be investigated and the results can be implemented in the estimation of the cost. The algorithm should then be able to more accurately find the optimal designs.
- The cost estimation of the algorithm can also be rewritten to express something completely different, for example the energy consumption of the multiplier. These different properties of the multiplier can also be combined to form an abstract cost value to find optimal designs.
- The algorithm only works with 2×2 bit multiplier elements creating an error when calculating 3×3. This can be changed to add in, for example, multiplier $M1$ by rewriting the probability calculation and error magnitude parts of the algorithm.
- The algorithm is not suited for a high number of approximate 2×2 multiplier variants. This is because all permutations are considered. An algorithm can be written which can in a more intelligent way search for optimal designs. Since this algorithm does not have to consider all permutations it would need a lot less memory and therefore be able to handle more multiplier elements.

Bibliography

[1] Larry R. D′Addario and Douglas Wang. An integrated circuit for radio astronomy correlators supporting large arrays of antennas. *Journal of Astronomical Instrumentation*, 05(02):1650002, 2016.

[2] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel. Invited: Cross-layer approximate computing: From logic to architectures. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[3] Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate computing: A survey. *IEEE Design Test*, 33(1):8–22, Feb 2016.

[4] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Architectural-space exploration of approximate multipliers. In Frank Liu, editor, *ICCAD*, page 80. ACM, 2016.

[5] Parag Kulkarni, Puneet Gupta, and Milos D. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *VLSI Design*, pages 346–351. IEEE Computer Society, 2011.

[6] S. S. Shalom and J. Tabrikian. Efficient computation of mse lower bounds via matching pursuit. *IEEE Signal Processing Letters*, 24(12):1798–1802, Dec 2017.

[7] Beayna Grigorian and Glenn Reinman. Improving coverage and reliability in approximate computing using application-specific, light-weight checks. 2014.

Appendices

Appendix A

Matlab code to model a MAC and test for quality

A.1 MAC Model

A.1.1 MacSpeed

```
function [ res ] = macSpeed(I1,I2,mul)
%This function first calculates the product using the FullMulSpeed function and
%then adds the results.
       res = sum(FullMulSpeed(I1,I2,mul),2);

end
```

A.1.2 FullMulSpeed

```
function [ r ] = FullMulSpeed( I1, I2, mul )
%FULLMULSPEED Matlab model of a 8x8 bit Multiplier. I1 and I2 are the input vectors and
%mul is a 16 element vector describing which 2x2 bit multiplier to use in
%which location. The 2x2 bit multipliers are defined in TwoBitMulSpeed

d1 = I1;
d2 = I2;

a1 = mod(d1,4); %mod will give the leftovers of the input divided by 4 which ranges from
     0 - 3
a2 = mod(d2,4); %this is the least significant 2 bits of the input.

d1 = d1 - a1; %the leftovers are substracted from the inputs
d2 = d2 - a2;

b1 = mod(d1,16); %this gives the next 2 bits of the input
b2 = mod(d2,16);

d1 = d1 - b1;
d2 = d2 - b2;

c1 = mod(d1,64);
c2 = mod(d2,64);

d1 = (d1 - c1)/64; %The output is divided by the shift of the multiplier
d2 = (d2 - c2)/64; %this is to get all the 2bit inputs in a range of 0-3
c1 = c1/16;
c2 = c2/16;
b1 = b1/4;
b2 = b2/4;
```

```
x1 = TwoBitMulSpeed(a1,a2,mul(1));      %the 16 multiplications are done using
    TwoBitMulSpeed
x2 = TwoBitMulSpeed(a1,b2,mul(2)).*4;  %the input mul(x) defines the correct 2x2 bit
    multiplier used
x3 = TwoBitMulSpeed(b1,a2,mul(3)).*4;  %The output is multiplied with the needed shift.
x4 = TwoBitMulSpeed(b1,b2,mul(4)).*16;

x5 = TwoBitMulSpeed(a1,c2,mul(5)).*16;
x6 = TwoBitMulSpeed(a1,d2,mul(6)).*64;
x7 = TwoBitMulSpeed(b1,c2,mul(7)).*64;
x8 = TwoBitMulSpeed(b1,d2,mul(8)).*256;

x9 =  TwoBitMulSpeed(c1,a2,mul(9)).*16;
x10 = TwoBitMulSpeed(c1,b2,mul(10)).*64;
x11 = TwoBitMulSpeed(d1,a2,mul(11)).*64;
x12 = TwoBitMulSpeed(d1,b2,mul(12)).*256;

x13 = TwoBitMulSpeed(c1,c2,mul(13)).*256;
x14 = TwoBitMulSpeed(c1,d2,mul(14)).*1024;
x15 = TwoBitMulSpeed(d1,c2,mul(15)).*1024;
x16 = TwoBitMulSpeed(d1,d2,mul(16)).*4096;


r = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 + x12 + x13 + x14 + x15 + x16
    ; % the outputs of all multiplications are added together to get the total output.

end
```

A.1.3 TwoBitMulSpeed

```
function [ r ] = TwoBitMulSpeed( I1, I2, mul )
%TWOBITMULSPEED This function calculates I1 * I2 and then changes the
%output to match the error made by the approximate 2x2 multiplier defined
%in mul.

    switch(mul)
        case 1      %accurate
            r = I1.*I2;
        case 2      %M1
            r = I1.*I2;  %calculate the exact result of 2x2 multiplier
            r(r==3) = 2; %add the errors in
            r(r==1) = 0;
        case 3      %M2
            r = I1.*I2;
            r(r==9) = 7;
        case 4      %M3
            r = I1.*I2;
            r(r==9) = 11;
        case 5      %M4
            r = I1.*I2;
            r(r==9) = 5;
    end

end
```

A.2 Quality test

A.2.1 QualityCheck

```
function [ Q ] = QualityCheck( I1,I2,di )
%QUALITYCHECK This function calculates both the MPE and MSE for given input
%samples I1 and I2. di is a matrix containing a list of designs with 16 values for 2x2
    multipliers for each design.

mpe = [];
mse = [];
```

```
h = waitbar(0,'Calculating Quality'); %progress bar
Accdot = dot(I1,I2,2); %calculating accurate dot product.
for i=1:size(di,1) %for each design in the list di

    waitbar(i/size(di,1),h); %update progress bar
    mul = di(i,:); %select ith design in the list

    Macdot = macSpeed(I1,I2,mul); %perform inaccurate mac operation

    mpe = [mpe; MPE(Accdot,Macdot)]; %calculate MPE
    mse = [mse; MSE(Accdot,Macdot)]; %calculate MSE

end
Q = [mse mpe]; %output list
delete(h)
clearvars h

end
```

A.2.2 MSE

```
function [ Error ] = MSE( Acc, Ax )
%MSE This function calculates the Mean Square Error. Acc are the accurate MAC
%results and Ax the approximate results.
Error = sum((Acc-Ax).^2)/size(Acc,1);


end
```

A.2.3 MPE

```
function [ Error ] = MPE(Acc, Ax)
%MPE This function calculates the Mean Percentage Error. Acc are the accurate MAC
%results and Ax the approximate results.
Error = 100*sum(abs(Acc-Ax)./Acc)/size(Acc,1);
end
```

Appendix B

VHDL code of the MAC

B.1 MAC

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.ALL;

entity AccMAC is
        port(   i1, i2:                 in unsigned(7 downto 0);
                        CLK:                    in std_logic;
                        result:                 out unsigned(22 downto 0)
        );
end AccMAC;

architecture bhv of AccMAC is
        COMPONENT eightbitmultiplier is
                port(   i1, i2: in std_logic_vector(7 downto 0);
                                result: out std_logic_vector(15 downto 0)
                );
        end COMPONENT;
        signal mul: std_logic_vector(15 DOWNTO 0);
        signal total, Rtotal: unsigned(22 downto 0) := (others => '0');

begin

        --eight bit multiplier:
        Mult: eightbitmultiplier PORT MAP(i1 => std_logic_vector(i1) , i2 =>
            std_logic_vector(i2) , result => mul);

        --sum up all inputs:
        total <= Rtotal + resize(unsigned(mul), 23);


PROCESS(CLK)
BEGIN
IF rising_edge(CLK) THEN
Rtotal <= total;                --next clock cycle update output
END IF;
END PROCESS;
result <= Rtotal;
end architecture;
```

B.2 eightbitmultiplier

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

```
USE IEEE.numeric_std.ALL;
USE work.ALL;

entity eightbitmultiplier is
        port(   i1, i2:                 in std_logic_vector(7 downto 0);
                        result:                 out std_logic_vector(15 downto 0)
        );
end eightbitmultiplier;

architecture eighttofour of eightbitmultiplier is


        COMPONENT fourbitmultiplier is
                port(   i1, i2: in std_logic_vector(3 downto 0);
                                result: out std_logic_vector(7 downto 0)
                );
        end COMPONENT;

        signal temp1, temp2, temp3, temp4: std_logic_vector(7 downto 0);
begin



        mul1: fourbitmultiplier PORT MAP(i1 => i1(3 downto 0) , i2 => i2(3 downto 0) ,
            result => temp1);        --LSB
        mul2: fourbitmultiplier PORT MAP(i1 => i1(3 downto 0) , i2 => i2(7 downto 4) ,
            result => temp2);        --MidSB
        mul3: fourbitmultiplier PORT MAP(i1 => i1(7 downto 4) , i2 => i2(3 downto 0) ,
            result => temp3);        --MidSB
        mul4: fourbitmultiplier PORT MAP(i1 => i1(7 downto 4) , i2 => i2(7 downto 4) ,
            result => temp4);        --MSB


        result <= std_logic_vector(resize(unsigned(temp1), 16) + shift_left(resize(
            unsigned(temp2), 16),4) + shift_left(resize(unsigned(temp3), 16),4) +
            shift_left(resize(unsigned(temp4), 16),8)); --shift the results and add up

end architecture;
```

B.3 fourbitmultiplier

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.ALL;

entity fourbitmultiplier is
        port(   i1, i2:                 in std_logic_vector(3 downto 0);
                        result:                 out std_logic_vector(7 downto 0)
        );
end fourbitmultiplier;

architecture fourtotwo of fourbitmultiplier is


        COMPONENT twobitmultiplier is
                port(   i1, i2: in std_logic_vector(1 downto 0);
                                result: out std_logic_vector(3 downto 0)
                );
        end COMPONENT;

        signal temp1, temp2, temp3, temp4: std_logic_vector(3 downto 0);
begin
```

```
        mul1: twobitmultiplier PORT MAP(i1 => i1(1 downto 0) , i2 => i2(1 downto 0) ,
            result => temp1); --LSB
        mul2: twobitmultiplier PORT MAP(i1 => i1(1 downto 0) , i2 => i2(3 downto 2) ,
            result => temp2); --MidSB
        mul3: twobitmultiplier PORT MAP(i1 => i1(3 downto 2) , i2 => i2(1 downto 0) ,
            result => temp3); --MidSB
        mul4: twobitmultiplier PORT MAP(i1 => i1(3 downto 2) , i2 => i2(3 downto 2) ,
            result => temp4); --MSB


        result <= std_logic_vector(resize(unsigned(temp1), 8) + shift_left(resize(
            unsigned(temp2), 8),2) + shift_left(resize(unsigned(temp3), 8),2) +
            shift_left(resize(unsigned(temp4), 8),4)); --shift the results and add up

end architecture;
```

B.4 Accurate twobitmultiplier

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity twobitmultiplier is
        port(   i1, i2:         in std_logic_vector(1 downto 0);
                        result:         out std_logic_vector(3 downto 0)
        );
end twobitmultiplier;

architecture accurate of twobitmultiplier is
        signal temp: std_logic_vector(3 downto 0);

begin

        temp(0) <= i1(0) and i2(1);
        temp(1) <= i1(1) and i2(0);
        temp(2) <= i1(1) and i2(1);
        temp(3) <= temp(0) and temp(1);

        result <= (temp(3) and temp(2)) & (temp(3) xor temp(2)) & (temp(0) xor temp(1))
            & (i1(0) and i2(0));  --gate logic of the 2x2 multiplier

end architecture;
```

B.5 M1 twobitmultiplier

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity AM1twobitmultiplier is
        port(   i1, i2:         in std_logic_vector(1 downto 0);
                        result:         out std_logic_vector(3 downto 0)
        );
end AM1twobitmultiplier;

architecture approx1 of AM1twobitmultiplier is
        signal temp: std_logic_vector(3 downto 0);

begin

        temp(0) <= i1(0) and i2(1);
        temp(1) <= i1(1) and i2(0);
        temp(2) <= temp(0) and temp(1);
        temp(3) <= i1(1) and i2(1);
```

```
        result <= temp(2) & (temp(2) xor temp(3)) & (temp(0) xor temp(1)) & temp(2); --
            gate logic of the 2x2 multiplier

end architecture;
```

B.6 M2 twobitmultiplier

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity AM2twobitmultiplier is
        port(   i1, i2:         in std_logic_vector(1 downto 0);
                        result:         out std_logic_vector(3 downto 0)
        );
end AM2twobitmultiplier;

architecture approx1 of AM2twobitmultiplier is
        signal temp: std_logic_vector(1 downto 0);

begin

        temp(0) <= i1(0) and i2(1);
        temp(1) <= i1(1) and i2(0);


        result <= '0' & (i1(1) and i2(1)) & (temp(0) or temp(1)) & (i1(0) and i2(0)); --
            gate logic of the 2x2 multiplier

end architecture;
```

B.7 M3 twobitmultiplier

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity AM3twobitmultiplier is
        port(   i1, i2:         in std_logic_vector(1 downto 0);
                        result:         out std_logic_vector(3 downto 0)
        );
end AM3twobitmultiplier;

architecture approx1 of AM3twobitmultiplier is
        signal temp: std_logic_vector(3 downto 0);

begin

        temp(0) <= i1(0) and i2(0);
        temp(1) <= i1(0) and i2(1);
        temp(2) <= i1(1) and i2(0);
        temp(3) <= i1(1) and i2(1);


        result <= (temp(3) and temp(0)) & (temp(3) and (not temp(0))) & (temp(1) or temp
            (2)) & temp(0); --gate logic of the 2x2 multiplier

end architecture;
```

B.8 M4 twobitmultiplier

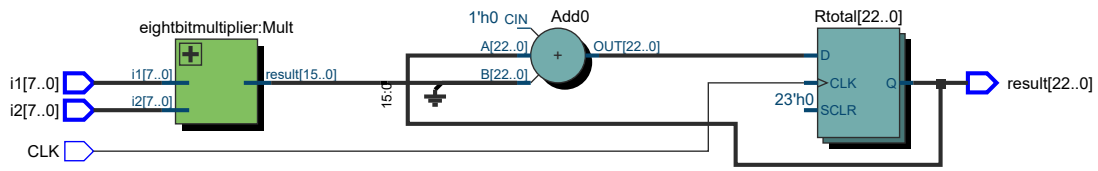
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

```
USE IEEE.numeric_std.ALL;

entity AM4twobitmultiplier is
        port(   i1, i2:         in std_logic_vector(1 downto 0);
                        result:         out std_logic_vector(3 downto 0)
        );
end AM4twobitmultiplier;

architecture approx1 of AM4twobitmultiplier is
        signal temp: std_logic_vector(1 downto 0);

begin

        temp(0) <= i1(0) and i2(1);
        temp(1) <= i1(1) and i2(0);


        result <= '0' & (i1(1) and i2(1)) & (temp(0) xor temp(1)) & (i1(0) and i2(0));
            --gate logic of the 2x2 multiplier

end architecture;
```

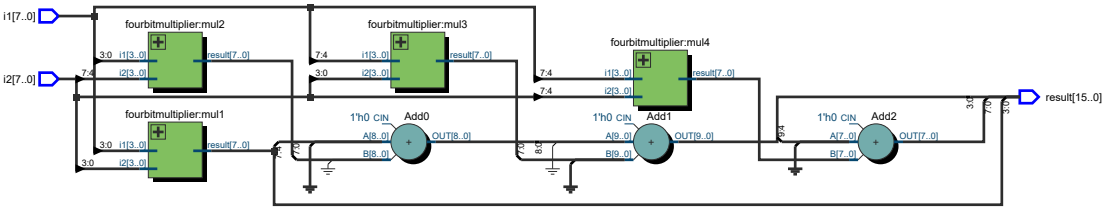
Appendix C

RTL view of the MAC synthesised by Quartus

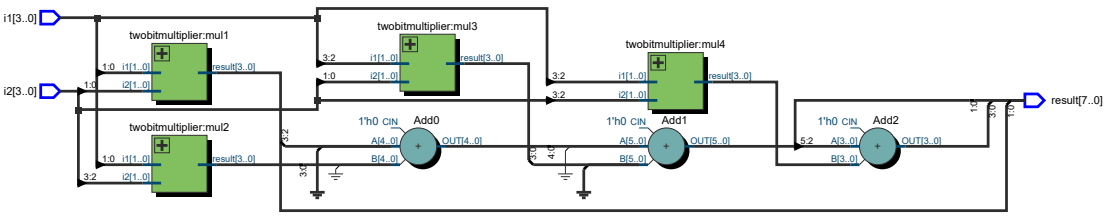
C.1 MAC



C.2 8 bit multiplier



C.3 4 bit multiplier



Appendix D

Design space exploration Matlab algorithm

D.1 FindDesign

```
function [de,Era,Ba,Co] = FindDesign( I1,I2, E, C)
%FINDDESIGN This function finds the optimal design for given input
%distributions, error magnitudes and cost estimations.
%I1 and I2 are the input samples. E is the error magnitude of the 2x2
%multiplier and C is the estimation for cost for each multiplier.
%outputs are: de - a list of the designs
%             Era - the removed designs due to overflow
%             Ba,Co - a list of all average errors and cost estimations

h = waitbar(0,'Calculating Weight and Chance of each multiplier'); %progress bar
W = FindWeight(I1,I2); %function for calculating weighted chance. (chance of error times
     the shift of the multiplier location.

waitbar(0.115,h,'Loading permutations');
load('../Variables/Per.mat'); %loading the permutations variable.

waitbar(0.123,h,'Calculating Cost');
Co = sum(C(Per(1:20000000,:)),2); %Calculating cost estimations by summing up costs for
    each design.
Co = [Co; sum(C(Per(20000001:end,:)),2)]; %Calculation divided in 2 for better memory
    management (some memory troubles occured)

waitbar(0.246,h,'Calculating Balance');
Er = int8(E); %making sure the error magnitudes are in int8 format to prevent memory
    overload.
Er = Er(Per); %inputing the errors into the permutations matrix
clearvars Per %clearing Per variable to clear up some memory

waitbar(0.38,h);
Ba = abs(double(Er)*W'); %calculating the average error for each design by multiplying
    and adding up with the weighted chance.

de = [];
Era = [];
c = unique(Co); %make a list of all different costs present
bd = 1000; %value of the current lowest average error, initialised at a high value
    (1000)

waitbar(0.5,h,'Sorting and Overflow handling');
for i=1:size(c,1)   %for each unique cost
    waitbar((i/size(c,1))/2+.5,h);
    a = find (Co==c(i));    %get all designs wih the smallest cost
```

```
    b = Ba(a);              %get the corresponding average errors of those designs
    if min(b) < bd          %if the lowest average error of those designs is lower then
        the lowest average error so far
        e = Er(a,:);        %Make a list of the designs
    end
    while min (b) < bd      %while the lowest average error of the list is still lower
        then the previous lowest
        o = find(b == min(b));  %find the designs with the lowest average error
        K = [];     %empty list K - a list of designs that do not overflow
        R = [];     %empty list R - a list of designs that do overflow
        [K,R] = FindOverflowFast(e(o,:)); %fill lists K and R
        if isempty(K) ==0   %if list K is not empty
            e=e(o,:);       %get a list of the designs
            de = [de;double(e(K,:)) ones(size(K,1),1)*c(i) ones(size(K,1),1)*min(b)]; %
                add the designs and their cost and average error to the output list
            bd=min(b);  %update new lowest average error
        else
            b(o) = [];  %remove designs from the list.
            e(o,:)= [];
        end
        if isempty(R) ==0  %if list R is not empty
            Era = [Era;min(b) c(i)]; %add designs to the list of designs that overflow
        end


     end
end

waitbar(1,h,'Done');
delete(h)
end
```

D.2 FindWeight

```
function [ W ] = FindWeight( I1,I2 )
%FINDWEIGHT This function finds the weighted chance of an error occuring
%I1 and I2 are the input samples.

d1 = I1;
d2 = I2;

a1 = mod(d1,4); %mod will give the leftovers of the input divided by 4 which ranges from
     0 - 3
a2 = mod(d2,4); %this is the least significant 2 bits of the input.

d1 = d1 - a1; %the leftovers are substracted from the inputs
d2 = d2 - a2;

b1 = mod(d1,16); %this gives the next 2 bits of the input
b2 = mod(d2,16);

d1 = d1 - b1;
d2 = d2 - b2;

c1 = mod(d1,64);
c2 = mod(d2,64);

d1 = (d1 - c1)/64; %The output is divided by the shift of the multiplier
d2 = (d2 - c2)/64; %this is to get all the 2bit inputs in a range of 0-3
c1 = c1/16;
c2 = c2/16;
b1 = b1/4;
b2 = b2/4;

A = (size(a1(a1==3),1)/(size(a1,1)*size(a1,2))  + size(a2(a2==3),1)/(size(a2,1)*size(a2
    ,2)))/2;  %for each 2bit part of the 8bit input the chance is calculated that it is
    3.
```

```
B = (size(b1(b1==3),1)/(size(b1,1)*size(b1,2))  + size(b2(b2==3),1)/(size(b2,1)*size(b2
    ,2)))/2;
C = (size(c1(c1==3),1)/(size(c1,1)*size(c1,2))  + size(c2(c2==3),1)/(size(c2,1)*size(c2
    ,2)))/2;
D = (size(d1(d1==3),1)/(size(d1,1)*size(d1,2))  + size(d2(d2==3),1)/(size(d2,1)*size(d2
    ,2)))/2;

S=[1 4 4 16 16 64 64 256 16 64 64 256 256 1024 1024 4096]; %a list of the shifts needed
W= S.*[A^2 A*B A*B B^2  A*C A*D B*C B*D   A*C B*C A*D B*D   C^2 C*D C*D D^2]; %the
    shifts are multiplied with the chance an error occurs(3*3 happens).
end
```

D.3 FindOverflowFast

```
function [ O, E ] = FindOverFlowFast( D )
%FINDOVERFLOW This function uses some logic steps to speed up the overflow
%check. This only works with the multipliers with error -2 +2 and -4 !!
%   D are the designs that need to be checked O is the list that does not
%   overflow and E the ist of designs that does.
O = [];
E = [];
for i=1:size(D,1) %for all designs in list D
    o=0;    %start with the assumption the design does not overflow
    for j=1:4 %for each set of 4 multipliers
        if D(i,4*j)==2 %check if the most significant is M3 (3*3=11)
            if nnz(D(i,(4*(j-1)+1):(4*j))==2)>2 %check if there are more then 2 M3 in
                the set of 4
                if (D(i,4*(j-1)+2) + D(i,4*(j-1)+2)) > (-2) %check if sum of specific
                    locations is higher then -2
                    o=1;    %if all these statements are true the individual 4x4
                        multiplier does overflow.
                end  %otherwise the 4x4 multiplier does not overflow
            end
        end
    end

    %It is possible that no individual 4x4 multipliers overflow but the 8x8
    %does. The following code deals with that
    if o==0     %if the 4x4 already overflows there is no need to check
        if D(i,16) == 2     %if the most significant 2x2 multiplier is M3 (3*3=11)
            if nnz([D(i,12:15) D(i,8)] == -4) == 0  %if there are no M4 multipliers
                present the design overflows.
                o=1;
            else
                o=FindOverflowSlow(D(i,:));  %if there are the rest is checked with an
                    exhaustive check
            end
        end
    end

    if o==0 %if it does not overflow
        O = [O;i]; %add it to the list
    else
        E = [E;i]; %else add it to the overflowing list
    end
end
end
```

D.4 FindOverflowslow

```
function [ o ] = FindOverflowSlow(D)
%FINDOVERFLOWSLOW this function does an exhaustive search on a design to see if it
    overflows
%D is the design to check, o is 1 if it overflows and 0 otherwise
a = D;
```

```
a(a==-4)=5;     %change the designs indication from error magnitude (as worked with in
    FindDesign)
a(a==2)=4;      %to a number (as worked with in the MAC model)
a(a==-2)=3;


k = [0:1:255];      %creating a matrix with all inputs
o = ones(1,256);
i1 = k;
i2 = zeros(1,256);

for i = [1:255]
    i1 = [i1 k];
    i2 = [i2 i*o];
end

R = FullMulSpeed(i1,i2,a);  %calculate all outputs with all inputs using the design D

o=0;        %assume design does not overflow
if max(R)> 65535    %change it to overflow if the output exceeds 65535 (16 bits)
o=1;
end


end
```