University of Twente.

Bachelor Thesis

Approximate signed multipliers for multiply-accumulate circuits

CAES - Computer Architecture for Embedded Systems

Author:
Johan Oedzes

Committee:
S.G.A. Gillani
dr.ir. A.B.J. Kokkeler
dr.ir. M.S. Oude Alink

June 29, 2018

Abstract

The field of approximate computing studies the trade-off between cost and quality in approximate computation or storage circuits, where computational quality is traded for reduced cost in hardware. Approximate computing is particularly interesting for MAC circuits, as the accumulation allows for error balancing. This work investigates an 8 bit approximate hybrid signed multiplier design targeted at application in MAC circuits. Both an accurate design and the approximate design are modeled in both matlab and VHDL. The VHDL model is synthesized by Quartus for the Cyclone IV E FPGA, and the synthesis results are used for cost estimation in terms of the amount of logic elements. Different error metrics, for both uncorrelated and correlated input distributions are considered for quality analysis, using uniform and normal distributions. It appears that the approximate design only saves about 6% in hardware compared to an accurate design, but has zero average error for different kinds of input distributions.

Contents

1 Introduction

The field of approximate computing studies the trade-off between cost and quality in approximate computation or storage circuits [1]. By modifying circuits in a smart way and making them smaller, the actual computation is approximated, which for some applications is acceptable given that the error stays within certain bounds. One of the fields where approximate computing can be used might be radio astronomy, given that the average error can be tuned to a zero average. One structure used a lot in radio astronomy is a Multiply-Accumulate circuit, or MAC [2].

A MAC is a circuit that calculates the dot product of two input vectors. This means that for two input vectors of N elements, a total of N multiplications and $N-1$ additions are executed to get to the final answer. This MAC structure is an interesting structure to explore in approximate computing, as it gives the opportunity to tune an error into a certain direction, as the accumulation of the products gives space for averaging the error of the final product. When an approximate multiplier with zero average error is used, one could expect that the final answer of a MAC output will also have zero average error. When looking at the MAC structure however, another configuration can be explored. If one would have an approximate multiplier with a given positive average error, and another approximate multiplier with a given negative average error, a certain combination of those multipliers could be used in a MAC. If tuned correctly, the average error of the final answer might go to zero again (i.e., after accumulation of the products).

Already quite some work has been done on unsigned multipliers [1][3]. In this report an approximate signed multiplier design will be explored, with a MAC as the targeted application. The savings in hardware will be explored for an FPGA design. The error of the multiplier will be explored by using different error metrics on both uncorrelated and correlated input distributions. The goal is to model an 8 bit signed multiplier and see if it is suitable to be used in a MAC. A MAC structure will not be modeled. The approximate multiplier will be constructed according to a design algorithm proposed by [4]. The paper proposes a hybrid design with both an accurate and an approximate part.

Section 2 will discuss the theory behind a signed multiplier, and how the multiplier works. Also the structure for the approximate multiplier will be discussed. Lastly, the way in which errors are calculated is discussed. In Section 3 the constructed models in both Matlab and VHDL will be discussed, and their behavior will be validated. In Section 4 the cost and quality of the approximate multiplier will be discussed. Error analysis will be done on both uncorrelated and correlated input distributions. The last Sections of the report discuss the results and their significance towards an application in a MAC, draw a final conclusion and present recommendations for future work.

2 Multiplier analysis

2.1 Accurate multiplier

Considering two numbers A and B in 2's complement notation which have to be multiplied, one needs a multiplier which is able to multiply signed numbers. A lot of signed multipliers use the so-called modified booth algorithm to calculate the product of two input numbers in 2's complement notation. The modified booth algorithm differentiates one input of the multiplier as the *multiplicand* and the other input as the *multiplier*. To avoid confusion, the *multiplicand* and the *multiplier* (both being inputs) will be represented in italics, while the multiplier (being the computational circuit) is denoted simply as multiplier. Based on how the bits are arranged in the *multiplicand*, a decision is made on how the *multiplier* should be added to the final answer. A modified booth multiplier is often called a Radix 4 (R4) multiplier. The way of implementing such an R4 multiplier is discussed in the following Sections.

2.1.1 R4 encoding

Given an input size of 8 bits, a total of four R4 bit triples are considered on the *multiplicand*, in an arrangement represented by (1). The 8 bits of the input number are represented by $b_7 ... b_0$, where b_7 represents the MSB and b_0 the LSB of the number. Each triple is represented by $R4_i$, with i being the index of the triple.

$$\underbrace{b_7 b_6}_{R4_3} \overbrace{b_5 b_4}^{R4_2} \underbrace{b_3 b_2}_{R4_1} \overbrace{b_1 b_0 b_{-1}}^{R4_0} \tag{1}$$

As can be seen, $R4_0$ contains b_{-1} which is not an actual bit present in the number. Consequently, this bit will be set to 0. Each triple of the *multiplicand* will be used in the generation of a partial product, thus resulting in a total of 4 partial products. Each partial product will have a different weight, depending on the position of the bit triple that is used in the generation. Triples which are more on the MSB side have a higher weight, and their partial products will be shifted left. This is demonstrated in Section 2.1.3. The partial products are generated using an encoding scheme as represented in table 1. The encoding scheme describes the encoding bits, and which multiplication factor (mf_i) should be used on the *multiplier* to get to the partial product. The 'sf_i' column represents the 'sign factor' which will be elaborated on later.

Table 1: R4 encoding

Input triple			Encoding				
b_{2i+1}	b_{2i}	b_{2i-1}	$sign_i$	$\times 2_i$	$\times 1_i$	sf_i	mf_i
0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	1
0	1	0	0	0	1	0	1
0	1	1	0	1	0	0	2
1	0	0	1	1	0	1	-2
1	0	1	1	0	1	1	-1
1	1	0	1	0	1	1	-1
1	1	1	1	0	0	0	0

The boolean expressions representing the truth table can be seen below. These expressions can be translated to hardware directly.

$$\begin{aligned} sign_i &= b_{2i+1} \\ \times 1_i &= b_{2i-1} \oplus b_{2i} \\ \times 2_i &= (b_{2i+1} \oplus b_{2i}) \cdot \overline{b_{2i-1} \oplus b_{2i}} \end{aligned} \tag{2}$$

2.1.2 R4 partial product generation

The actual partial product generation can now be done by the use of AND-gates on the encoding bits as shown by (2) and the *multiplier* bits. This encoding needs to be done bit by bit, as for each bit position of the partial product, the encoding bits will determine the place of each bit of the *multiplier*. For example, in the case that only the encoding bit $\times 1$ is high, meaning that the partial product will be the same as the *multiplier*, each *multiplier* bit should propagate to the output on the same bit position. So, when saying that the partial product for *multiplicand* i on bit position j is pp_{ij} and the *multiplier* on bit position j is A_j, and the encoding bit $\times 1_i$ is high, then the result should be $pp_{ij} = A_j$. When the the encoding bit $\times 2_i$ is high, this would result in $pp_{ij} = A_{j-1}$, indicating a bit shift to the left. When the *sign* encoding bit is high, the result should be flipped so a 1's complement form is acquired. The hardware that can achieve this operation is shown in the following figure:

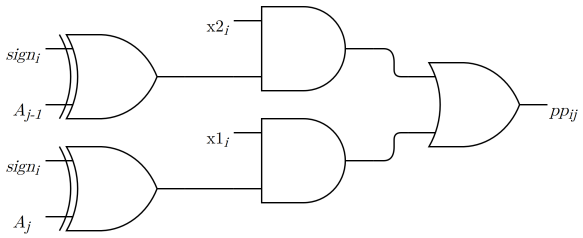


Figure 1: Bitwise R4 partial product (pp_{ij}) generator

The index j as mentioned above goes over all of the partial product bits, which are 9 in total. This extra bit compared to the original length of 8 bits is due to the highest multiplication factor of 2, resulting in a bit shift of 1 bit to the left. To still represent all the possible numbers, an extra bit should be added to accommodate for this bit shift. j then has the following range: $0 \leq j \leq 8$. This range results in some interesting cases for the index of A, as A_{-1} and A_8 can both occur given the range of j, but those do not represent actual digits of the 8 bit *multiplier*. Whenever the index j on A is negative, a non existent part of the *multiplier* is addressed, so all of those cases are 0. Whenever the index j on A is bigger than 7, a part of the *multiplier* is addressed which cannot be represented by the actual *multiplier*. However, binary numbers can be made longer. In the case of unsigned numbers this can simply be done by adding extra zeros on the MSB side of the number. In the case of signed numbers however, this can be done by sign extending the number. Sign extension can be done by taking the value of the MSB, and using that value for the extra bits required on the MSB side. So, When index j on A is bigger than 7, index 7 should be used instead. As an example, in the case where $pp_{i8} = A_8$ should be presented, $pp_{i8} = A_7$ is used instead.
It should be noted that the structure as shown in Fig. 1 generates a 1's complement representation of the number, where a 2's complement notation is required. This is where the sign factor included in table 1 comes in. Whenever a negative partial product is generated, the sign factor will be 1. When the sign factor is added to its corresponding partial product, a 2's complement notation is achieved. The sign factor can be found with the following boolean expression, using the encoding bits as shown in (2).

$$sf_i = sign_i \cdot (\times 1_i + \times 2_i) \tag{3}$$

2.1.3 R4 partial product addition

With the structure of Fig. 1 each partial product bit should be generated for each of the 4 *multiplicand* triples. The resulting partial products will then need to be added together to get to the final result. To do this in a hardware efficient way, a Wallace tree can be used. By the use of a Wallace tree, it is not necessary to completely sign extend each partial product to the full length of the final answer (16 bits). Instead, the Wallace tree uses a smart technique to limit the need of sign extension. Furthermore, the Wallace tree uses carry save adders to add the partial products together, with only one ripple carry adder at the end (to accommodate the carry propagation), resulting in an overall faster addition structure. To make use of the Wallace tree, the partial products and their sign factors should be arranged as follows (where c indicates each bit of the final answer):

					$\overline{pp_{08}}$	pp_{08}	pp_{08}	pp_{07}	pp_{06}	pp_{05}	pp_{04}	pp_{03}	pp_{02}	pp_{01}	pp_{00}
				1	$\overline{pp_{18}}$	pp_{17}	pp_{16}	pp_{15}	pp_{14}	pp_{13}	pp_{12}	pp_{11}	pp_{10}		sf_0
		1	$\overline{pp_{28}}$	pp_{27}	pp_{26}	pp_{25}	pp_{24}	pp_{23}	pp_{22}	pp_{21}	pp_{20}		sf_1		
1	$\overline{pp_{38}}$	pp_{37}	pp_{36}	pp_{35}	pp_{34}	pp_{33}	pp_{32}	pp_{31}	pp_{30}		sf_2				
									sf_3						
c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Figure 2: R4 partial product arrangement for each partial product pp_i, or each partial product bit pp_{ij}, and the sign factors sf_i. Bitwise addition results in the final result c

In the structure of Fig. 2 each column should be added bitwise to get to the final result c.

2.2 Approximate multiplier

The accurate design as discussed in Section 2.1 works well, but it requires a lot of hardware, especially for adding all the partial products. Multiplier designs with higher radix result in less groups of bits on the *multiplicand*. Instead of 4 groups of *multiplicand* bits, as demonstrated in (1), one could have a lot less groups, resulting in less partial products. Going to a higher radix multiplier introduces new problems however. Where a radix 4 multiplier has partial product multiplications by factors of either 0, 1 or 2, a radix 8 multiplier has partial product multiplications by factors of either 0, 1, 2, 3 or 4. The multiplication factor of 3 is a problem here, as such a multiplication cannot easily be done by bit-shifting into a certain direction like with powers of 2, and therefore requires bigger, more complicated hardware to achieve. If a higher radix multiplier would only use multiplication factors which are powers of 2, this would result in less partial products, where the generation of the partial products can still be done in a simple way. The work of [4] proposes the use of a hybrid multiplier structure: where most of the partial products will be generated by accurate radix 4, and one partial product by an approximate higher radix structure. The idea is that the multiplication factors of the higher radix part will all be rounded to the highest four powers of 2, so the non 2-power multiplication factors will no longer form a problem. Also, the design proposed by the paper is scalable. Where in the paper a 16 bit multiplier is tested with hybrid R4/R64, R4/R256 and R4/R1024 structures, the actual proposed design is scalable by a factor k, resulting in a general expression for the used structure as R4/R2^k, where k is the scale factor, and $k \geq 4$. In this case the design will be applied to an 8 bit multiplier, with a scale factor of $k = 4$, resulting in a hybrid R4/R16 structure, which according to the paper is the smallest design possible. The way of implementing the design is discussed in the following Sections. The encoding and generation of the R4 partial products is done in exactly the same way as discussed in Section 2.1.

2.2.1 R16 encoding

The hybrid R4/R16 cases results in 3 groups of bits on the *multiplicand* as demonstrated in (4) below. As can be seen, this structure will result in only 3 partial products compared to the 4 partial products of the R4 only case. The R16 part is represented by $R16_0$, and takes into account 4 bits of the *multiplicand*, and again one extra bit denoted as b_{-1} which is set to 0.

$$\underbrace{b_7 b_6}_{R4_2}\overbrace{b_5 b_4 b_3}^{R4_1}\underbrace{b_2 b_1 b_0 b_{-1}}_{R16_0} \tag{4}$$

The design in [4] proposes the boolean expressions required for getting the encoding as scalable expressions by the aforementioned scaling factor k. When taking $k = 4$, the following encoding expressions are acquired:

$$\begin{aligned} sign =& b_3 \\ \times 1 =& (\bar{b}_2 \cdot \bar{b}_1 \cdot \bar{b}_0 + b_2 \cdot b_1 \cdot b_0) \cdot (b_0 \oplus b_{\text{-}1}) \\ \times 2 =& \bar{b}_3 \cdot \bar{b}_2 \cdot (\bar{b}_1 \cdot b_0 \cdot b_{\text{-}1} + b_1 \cdot \bar{b}_0) + b_3 \cdot b_2 \cdot (b_1 \cdot \bar{b}_0 \cdot \bar{b}_{\text{-}1} + \bar{b}_1 \cdot b_0) \\ \times 4 =& \bar{b}_2 \cdot b_1 \cdot (b_3 + b_0) + b_2 \cdot \bar{b}_1 \cdot (\bar{b}_3 + \bar{b}_0) \\ \times 8 =& \bar{b}_3 \cdot b_2 \cdot b_1 + b_3 \cdot \bar{b}_2 \cdot \bar{b}_1 \end{aligned} \tag{5}$$

Using these expressions, a new encoding truth table for the approximate R16 encoding can be constructed, using the bits of the $R16_0$ group as input. The table is shown below.

Table 2: Approximate R16 encoding

Input					Encoding						
b_3	b_2	b_1	b_0	$b_{\text{-}1}$	$sign$	$\times 8$	$\times 4$	$\times 2$	$\times 1$	mf	sf
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0	2	0
0	0	1	1	0	0	0	1	0	0	4	0
0	1	0	0	0	0	0	1	0	0	4	0
0	1	0	1	0	0	0	1	0	0	4	0
0	1	1	0	0	0	1	0	0	0	8	0
0	1	1	1	0	0	1	0	0	1	?	0
1	0	0	0	0	1	1	0	0	0	-8	1
1	0	0	1	0	1	1	0	0	0	-8	1
1	0	1	0	0	1	0	1	0	0	-4	1
1	0	1	1	0	1	0	1	0	0	-4	1
1	1	0	0	0	1	0	1	0	0	-4	1
1	1	0	1	0	1	0	0	1	0	-2	1
1	1	1	0	0	1	0	0	1	0	-2	1
1	1	1	1	0	1	0	0	0	1	-1	1

When looking the at the table, the row with red digits stands out. On this particular input case, both the $\times 1$ and $\times 8$ cases get triggered, which would result in an unknown multiplication factor. The $\times 1$ bit should not have been triggered on this particular input. This is due to a mistake in the encoding statements from [4]. To fix this problem, a different encoding expression for $\times 1$ has been found. The following alternative expression for $\times 1$ can be used:

$$\times 1 = b_3 \cdot b_2 \cdot b_1 \cdot b_0 \tag{6}$$

The new expressions for $\times 1$ takes care of the problem and realizes the following encoding truth table:

Table 3: Better approximate R16 encoding

Input					Encoding						
b_3	b_2	b_1	b_0	b_{-1}	$sign$	$\times 8$	$\times 4$	$\times 2$	$\times 1$	mf	sf
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0	2	0
0	0	1	1	0	0	0	1	0	0	4	0
0	1	0	0	0	0	0	1	0	0	4	0
0	1	0	1	0	0	0	1	0	0	4	0
0	1	1	0	0	0	1	0	0	0	8	0
0	1	1	1	0	0	1	0	0	0	8	0
1	0	0	0	0	1	1	0	0	0	-8	1
1	0	0	1	0	1	1	0	0	0	-8	1
1	0	1	0	0	1	0	1	0	0	-4	1
1	0	1	1	0	1	0	1	0	0	-4	1
1	1	0	0	0	1	0	1	0	0	-4	1
1	1	0	1	0	1	0	0	1	0	-2	1
1	1	1	0	0	1	0	0	1	0	-2	1
1	1	1	1	0	1	0	0	0	1	-1	1

2.2.2 R16 partial product generation

The partial product generation for the approximate R16 case can be done much in the same way as the accurate R4 partial product generation as discussed in Section 2.1.2. The R16 case has 2 more encoding bits compared to the R4 case, resulting in a partial product generator of roughly double the size of the one in Fig. 1. The hardware structure for the R16 partial product generation can be seen in the figure below.

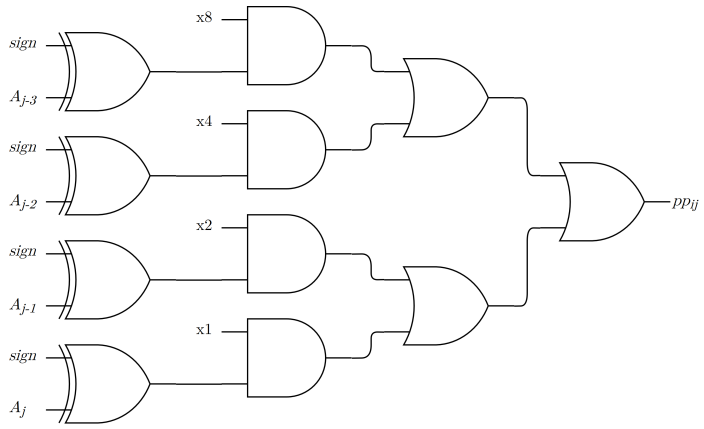


Figure 3: Bitwise R16 partial product (pp_{ij}) generator

In this case the partial product itself consists of 11 bits. This is because the highest multiplication that can be performed on the *multiplier* is by a factor of 8, which means that 3 extra bits are required as compared to the input to be able to represent all the possible outputs. This means that j in Fig. 3 has the following range: $0 \leq j \leq 10$. Again, if the indices j of pp_{ij} or A_j are negative, a 0 will be used as input, and whenever the index j of A_j is larger than 7, an index of 7 will be used instead (resulting in sign extension). The sign factor for the R16 encoding can be found by the following expression:

$$sf_i = sign \cdot (\times 1 + \times 2 + \times 4 + \times 8) \tag{7}$$

2.2.3 R4/R16 partial product addition

Just like the accurate R4 case, also for the R4/R16 the partial products need to be added together. Now there are only two R4 bit triples and one approximate R16 bit group thus resulting in three partial products total, in an arrangement shown by Fig. 4 below. Here, c indicates each bit of the final answer.

			$\overline{pp_{010}}$	pp_{010}	pp_{010}	pp_{09}	pp_{08}	pp_{07}	pp_{06}	pp_{05}	pp_{04}	pp_{03}	pp_{02}	pp_{01}	pp_{00}
		1	$\overline{pp_{18}}$	pp_{17}	pp_{16}	pp_{15}	pp_{14}	pp_{13}	pp_{12}	pp_{11}	pp_{10}				sf_0
1	$\overline{pp_{28}}$	pp_{27}	pp_{26}	pp_{25}	pp_{24}	pp_{23}	pp_{22}	pp_{21}	pp_{20}		sf_1				
									sf_2						
c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Figure 4: R4/R16 partial product arrangement for each partial product pp_i, or each partial product bit pp_{ij}, and the sign factors sf_i. Bitwise addition results in the final result c

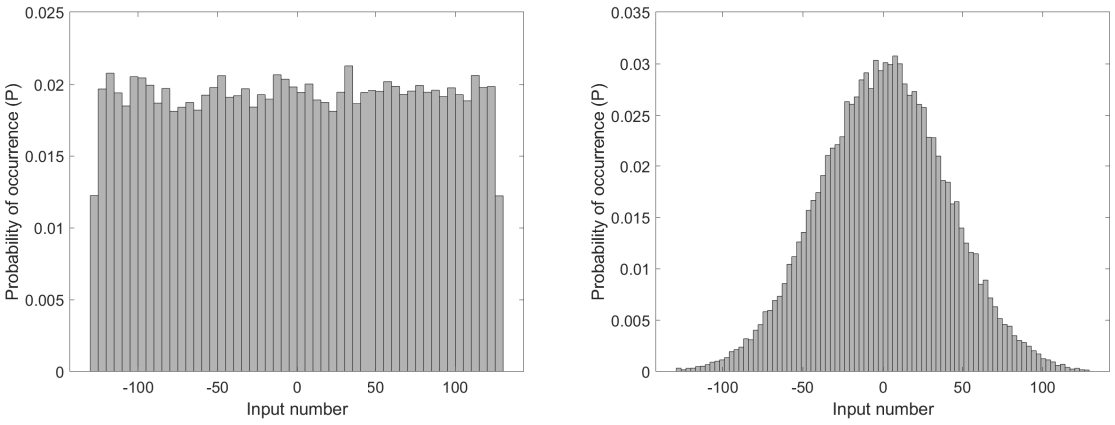
Fig. 4 clearly shows that there is one partial product less as compared to the accurate R4 case. Again each column of bits is added bitwise to get to the final result c.

2.3 Error analysis

An error analysis can be performed on the design to get insight into the quality and cost trade-off which is made by approximating the multiplier. In the following two Sections, different input distributions and error metrics are discussed.

2.3.1 Input distributions

Different input distributions can be used as input for the multiplier. A distribution size of $2^{16} = 65536$ samples is used, as this is a large set of numbers, while still enabling reasonable simulation time. The multiplier is then tested with both a uniform and a normal distribution. For both of the multiplier inputs, a separate distribution is generated.



(a) Uniform distribution

(b) Normal distribution ($\mu = 0, \sigma = 40$)

Figure 5: Uncorrelated input distributions

When looking at the different distributions, it becomes clear that in terms of probability the uniform distributions yields a set of numbers where all of the numbers have equal probability of occurring, where in the normal distribution especially numbers with lower magnitude as compared to all of the numbers in the set, have a higher probability of occurring than numbers with higher magnitude (given $\mu = 0$). After generating a normal distribution with $\mu = 0$ and $\sigma = 40$, a small amount of the generated values will be out of the input bounds of the multiplier: $-128 \leq \text{input} \leq 127$. All numbers of the normal distribution that appear out of this range will be distributed uniformly over the range. As this only holds for about 0.15% of the numbers in the set, the influence will be negligible. Furthermore, it

should be noted that the uniform and normal distribution have a different σ. The range for the uniform distribution is chosen as $-128 \leq \text{input} \leq 127$, so all of the possible input numbers can be represented. This results in a σ for the uniform distributions: $\sigma = \sqrt{\frac{1}{12}(b-a)^2} = \sqrt{\frac{1}{12}(127-(-128))^2} \approx 74$, whereas the normal distribution has a σ of 40.

The multiplier design can be tested by giving two uniform distributions, or two normal distributions as inputs. In the case of Fig. 5 the distributions are uncorrelated, meaning the generation of the two input distributions for the multiplier happens separately and the distributions do not influence each other.

Keeping in mind the original focus of the multiplier, namely a MAC, it is not uncommon for the two multiplier inputs to be correlated. An example of a MAC application where the inputs are correlated is radio astronomy. A MAC is often used to combine two antenna inputs, where a common signal is received by both of the antennas, but noise is added at both sides. The two inputs to the MAC will be partially correlated, with a correlation factor depending on the magnitude of the actual signal compared to the added noise. The influence of correlated inputs on the multiplier can be investigated by generating correlated input distributions, and looking at the error for different correlations. Fig. 6 proposes a way of generating two correlated distributions.

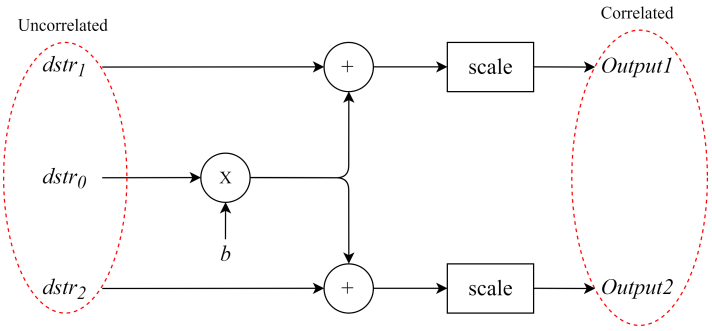


Figure 6: Generation of correlated distributions

First, three uncorrelated distributions should be generated, denoted in the figure as $dstr_0$, $dstr_1$ and $dstr_2$. A scaled version of $dstr_0$ is added to the other distributions, as a common component for realizing the correlation. $dstr_0$ is scaled by a factor b, which is a factor depending on correlation factor ρ as demonstrated in (8). As can be seen, a higher correlation factor results in a larger scaling factor b. Also, as ρ approaches 1, b approaches ∞ resulting in two equal output distributions (after the distributions are scaled back). One thing to note is that an 8 bit 2's complement number n has the following range: $-128 \leq n \leq 127$. The structure as shown in Fig. 6 shows an addition of distributions, which could result in out of bounds values. Therefore the 'scale' block is responsible for scaling the distribution back to a range of $-128 \leq n \leq 127$.

$$b = \sqrt{\frac{\rho}{1-\rho}} \tag{8}$$

2.3.2 Error metrics

Different error metrics are used for evaluating the quality of the multiplier. The most straightforward metric is the mean error (ME),which can be calculated using (9), where N is the size of the set of numbers from the input distributions and X and Y are the two input distributions. The computation which is performed by the approximate multiplier is denoted by $\widetilde{*}$.

$$ME = \frac{1}{N}\sum_{k=0}^{N-1} X_k \cdot Y_k - X_k \mathbin{\widetilde{*}} Y_k \tag{9}$$

The mean error is an interesting error to look at for a MAC application, as positive and negative errors will cancel each other, resulting in a lower ME.

An error metric which is used more commonly is the mean squared error (MSE). This is a more often used error metric, and can therefore by used as a number to compare different designs. As the error will be squared and therefore positive, no error cancellation by the summation of both positive and negative errors can occur.

$$MSE = \frac{1}{N} \sum_{k=0}^{N-1} (X_k \cdot Y_k - X_k \mathbin{\tilde{*}} Y_k)^2 \tag{10}$$

Another interesting metric is the mean absolute percentage error (MAPE). Here, each error is divided by the accurate value to get the relative difference with respect to the accurate value. The absolute value of each relative difference is taken, so error cancellation cannot occur, and this metric will therefore be a good measure of the relative error. This metric gives a better insight into the relative error than the MSE, as the MSE will go to a higher magnitude due to squaring of the errors.

$$MAPE = \frac{100\%}{N} \sum_{k=0}^{N-1} \left| \frac{X_k \cdot Y_k - X_k \mathbin{\tilde{*}} Y_k}{X_k \cdot Y_k} \right| \tag{11}$$

One thing to keep in mind while calculating the MAPE is that each instance where the accurate value is equal to 0 cannot be calculated, as a division by 0 would occur. In this particular application however, the answer for those cases can be taken as 0, as all of the cases where the accurate multiplier yields 0, the approximate multiplier yields the same result, giving an error of 0.

3 Model implementation

To test the design as demonstrated in Section 2.2 both a Matlab model and a VHDL model were constructed to test both the quality and the cost of the design.

3.1 Matlab model

The Matlab model was targeted on simulating the approximate multiplier computations in a fast way, while still resembling the hardware in its code as much as possible. This close resemblance to the hardware makes it easier to change the model later on, if for example a new design should be tested. Furthermore, this close resemblance helps with the construction of the VHDL model, as in this case Matlab can be used as reference to the hardware layout. As Matlab does not have a lot of low level functionality, a lot of functionality needed to be coded from scratch. During the model design, the accurate model was constructed first, so all the functionality could be debugged more easily if necessary (an approximate structure with incorrect answers is not easy to debug). After completion of the accurate model, the approximate model was constructed in the same manner. Instead of the Wallace tree structure as mentioned in Section 2.1.3, a regular ripple carry adder structure was modeled in Matlab, as a ripple carry adder is modeled more easily, and the end result of computation yield the same numbers. The code for the Matlab model can be found in Appendix C.

3.2 VHDL model

The VHDL model is targeted on getting insight into the size of the multiplier in terms of the amount of logic elements. Both the accurate R4 and the approximate hybrid R4/R16 multiplier are modeled in VHDL, so the two can be compared for a measure of the actual reduction of hardware in the approximate case. The amount of logic elements used in the models is found after synthesis of the designs by Quartus. The synthesis is done for an FPGA in the Cyclone IV E family. Both the boolean expressions and the partial product structures from Section 2 and the Matlab model were used as reference for the construction of the VHDL model. The code for both the accurate and approximate multiplier models in VHDL can be found in Appendix D.

The model of the accurate R4 multiplier can be seen in Fig. 7 below. Most blocks in this figure represent full adders and half adders, which make up the Wallace tree used for the addition of the partial products. The figure contains 4 blocks responsible for the generation of partial products. The internals of this block are shown in Fig. 8. This part is made up mostly of the partial product structures, which can be recognized by the AND gates and OR gates on the right side.

Figure 7: RTL view of accurate R4 multiplier after synthesis in Quartus

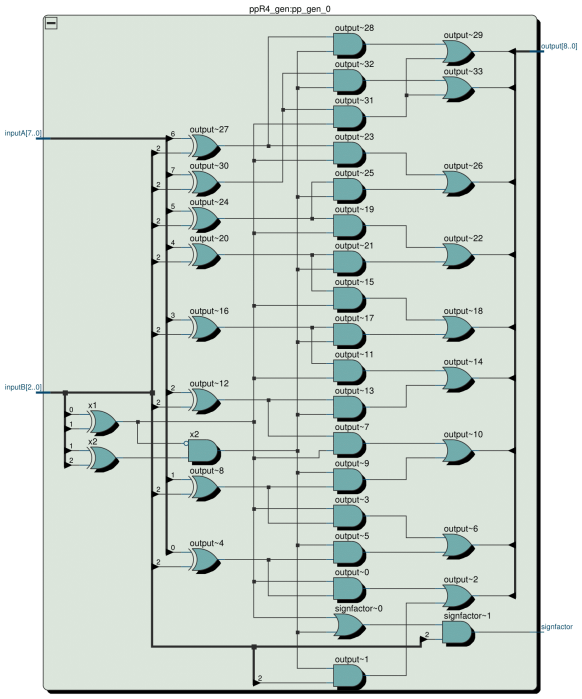


Figure 8: RTL view of accurate R4 encoding and partial product generation after synthesis in Quartus

The model of the approximate hybrid R4/R16 multiplier can be seen in Fig. 9. When comparing this structure to the accurate R4 one, it can be seen that the addition tree is smaller by a full addition stage, as the approximate structure has one less partial product. This structure has 3 blocks for the

partial product generation, one of which contains the approximate R16 encoding. The internals of this block can be seen in Fig. 10. The partial product generation by the approximate R16 encoding proves to be a lot bigger compared to the R4 case. This is an expected result, as the R16 encoding has more encoding bits, each consisting of larger boolean expressions. Also, the partial product generated by the approximate R16 encoding is bigger compared to the R4 case.

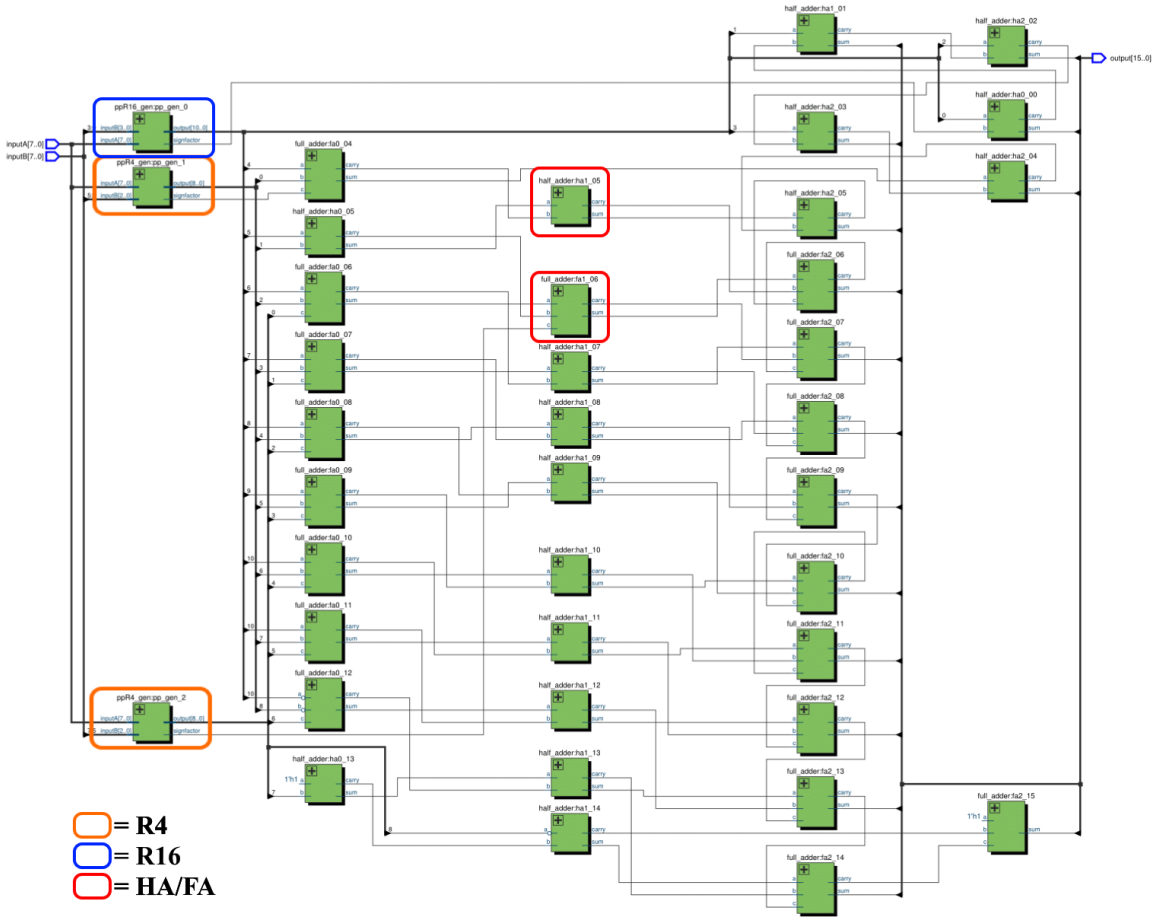


Figure 9: RTL view of hybrid R4/R16 multiplier after synthesis in Quartus

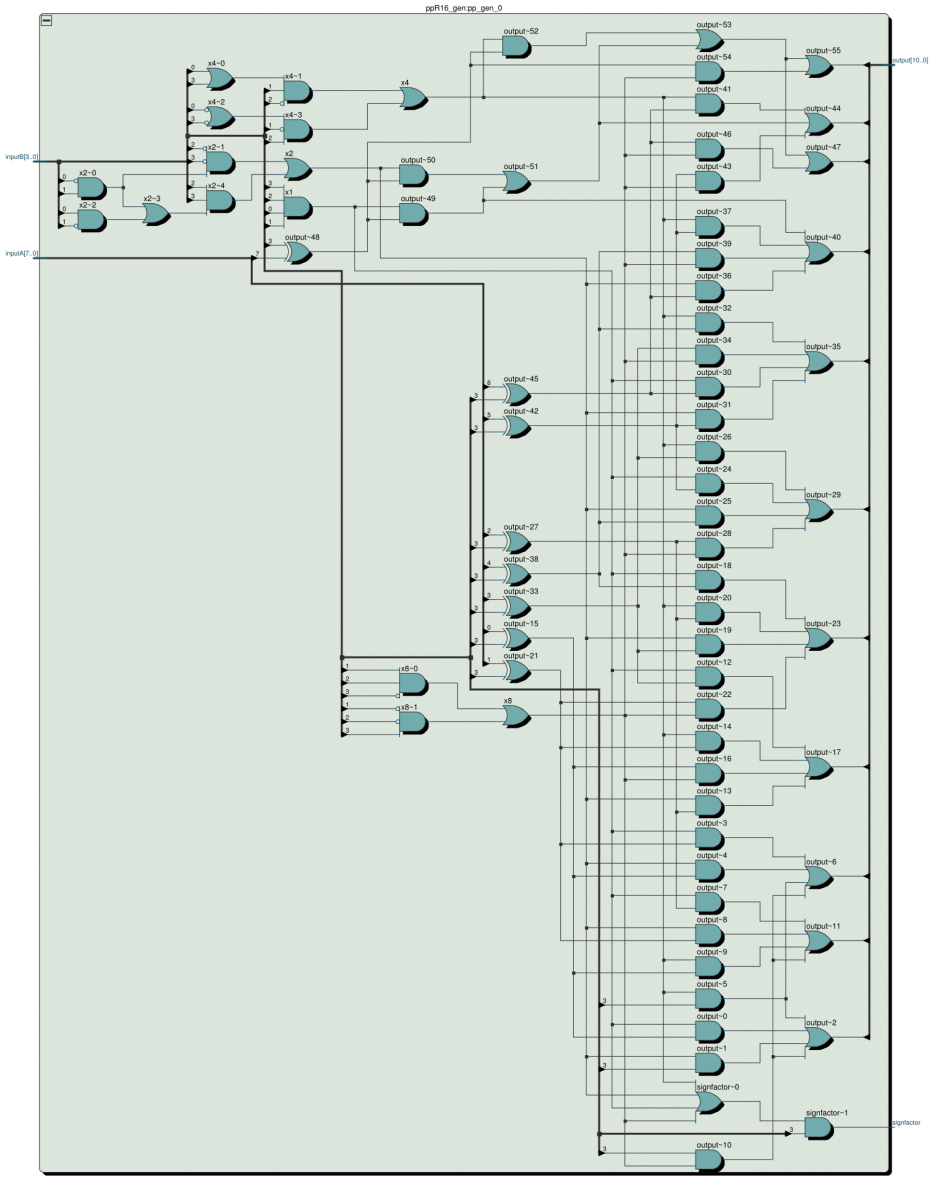


Figure 10: RTL view of approximate R16 encoding and partial product generation after synthesis in Quartus

3.3 Model validation

The VHDL model and the Matlab model should behave in exactly the same way. To validate that this is actually the case, a test was performed on both of the models, with the same numbers as inputs to the multiplier, where the outputs are compared, and should yield exactly the same result. To perform this test, a Matlab script was written to generate a large set of random numbers (within the range of 8 bit 2's complement numbers). Given the set of numbers, the Matlab script used the approximate multiplier structure to calculate the output numbers, and also generated a 'do' file containing the same input numbers as used in Matlab. The multiplier model in VHDL was simulated using Modelsim, where inputs can be forced on the multiplier using the 'do' file generated by Matlab. A snapshot of both the Modelsim simulation environment and the console output from Matlab can be found in Appendix A, both showing 20 equal input cases. Both simulations yielded exactly the same results, indicating equal behavior of the model.

4 Results

4.1 Hardware cost

The amount of logic elements taken by the VHDL models as described in section 3.2 is a measure of the size of the multiplier. By approximating the design, the number of logic elements should go down, where a bigger reduction means better results. The number of logic elements required for the accurate R4 and the approximate R4/R16 designs are shown in table 4 below.

Table 4: Hardware cost in terms of Logic Elements (LE)

	#LE
R4	135
R4/R16	127

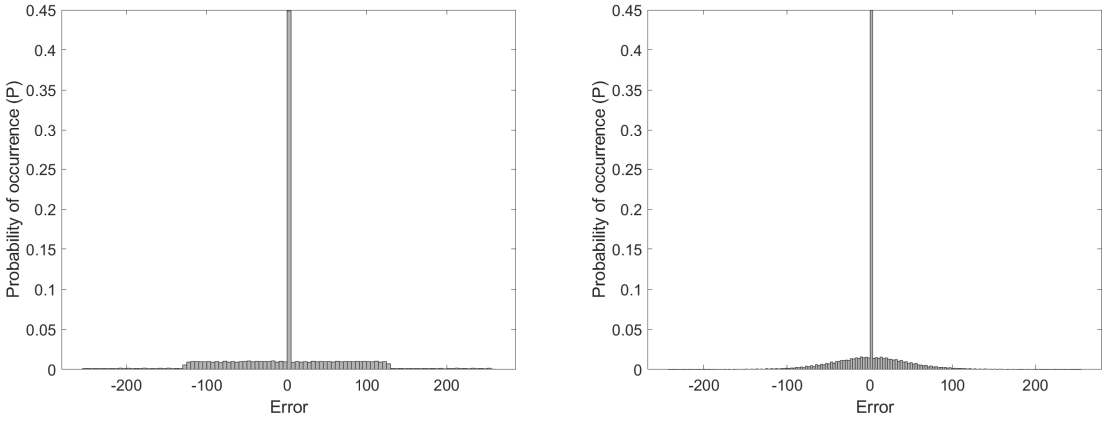
The numbers from the table are indicated by Quartus after synthesis of the VHDL model (for an FPGA from the Cyclone IV E family). As can be seen, this particular approximate design only reduces the amount of logic elements by 8 gates, which is a reduction of roughly 6% with respect to the accurate design. The reduction of 6% is not very significant, but given this particular design, that could be expected.
The partial product addition tree makes up a significant part of the hardware, and the tree itself is used in its accurate form. The only reduction in hardware acquired when going from the accurate case to the approximate case consists of one full stage of the addition tree, as the approximate case has one less partial product. Still, one would expect a bigger hardware reduction when a complete stage of the partial product addition can be skipped. However, one should also take into account the partial product encoding and generation. The approximate encoding results in bigger encoding hardware, which becomes clear when comparing (2) and (5) from Section 2.

4.2 Error analysis

When using the distributions as described in Section 2.3.1 as inputs for the model, the error metrics as described in Section 2.3.2 can be acquired. The two Sections below show results for both uncorrelated and correlated input distributions.

4.2.1 Uncorrelated inputs

Both a uniform and a normal distribution were applied as inputs to the multiplier, and the error of the approximation was calculated. A histogram of the errors for both the uniform and normal distribution can be seen in Fig. 11. When looking at the figures, the large peak at an error of 0 stands out for both of the cases, and indicates that just under half of the given inputs resulted in 0 error. The rest of the errors in the histogram take a shape which is similar to their input distribution.



(a) By uniform input distribution

(b) By normal input distribution ($\mu = 0, \sigma = 40$)

Figure 11: Uncorrelated error histograms

The error metrics which were calculated for both of the input distributions can be seen in table 5 below. Both distributions result in a near-zero mean error. Also, when running the Matlab model multiple times (i.e., generating new distributions, and applying those to the model of the approximate multiplier) it appears that the ME is sometimes negative and sometimes positive for both of the input distributions, indicating a mean error of zero.

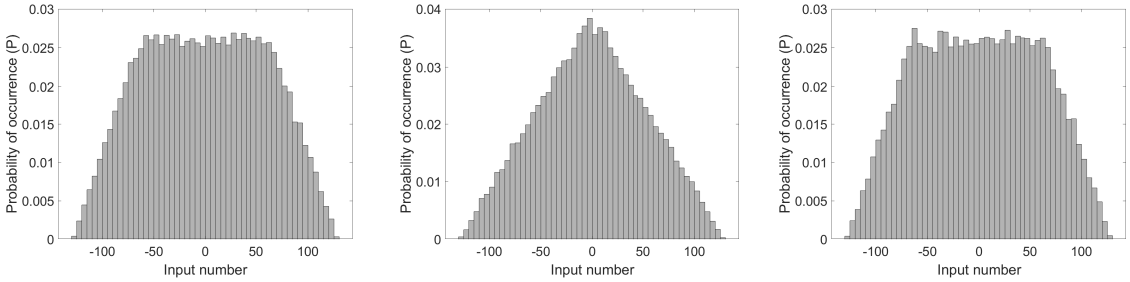
Table 5: Error metrics for uniform and normal input distribution

	Uniform	**Normal**
ME	0.392	-0.172
MSE	5110.0	1503.6
MAPE	2.71	5.25

The other two error metrics, the MSE and the MAPE, prove to be more stable over multiple runs of the Matlab model: they stay within the same order of magnitude. An interesting thing to note is that the uniform input distribution results in a higher MSE compared to the normal input distribution, but in a lower MAPE compared to the normal input distribution. When looking at the error histograms in Fig. 11 it can be seen that both input distributions result in approximately the same amount of errors, as the graph indicates about the same probability for 0 error. Then it can be expected that the MSE in the case of uniform input distribution is higher, as the histogram shows a fairly even spread of the errors around the different error values, where the error histogram of the normal input distributions shows more errors closer to zero, and less errors towards higher error values.
The graphs do not provide enough information to be able to say something about the different MAPE, as the accurate value should be known for that as well, which is not indicated in the graphs.
Furthermore, it should be noted that especially the σ in both the uniform and the normal distribution has a lot of influence on the outcomes of the MSE and the MAPE. The shapes of the distributions differ for different sigma, resulting in different values for the MSE and MAPE, when following the same reasoning for the MSE as above.

4.2.2 Correlated inputs - uniform

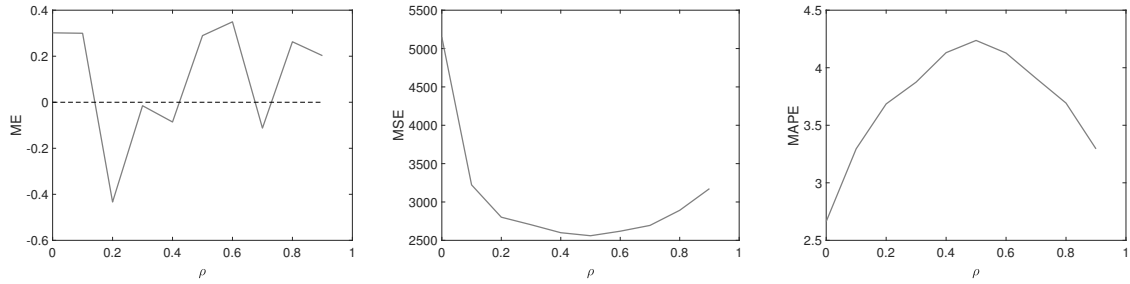
For the correlated case the errors are evaluated for different correlation factors ρ. Several shapes of uniform input distributions for different ρ values can be seen in Fig. 12.



(a) $\rho = 0.1$ (b) $\rho = 0.5$ (c) $\rho = 0.9$

Figure 12: Correlated uniform input distributions

When looking at the figures it becomes clear that already for a correlation factor of $\rho = 0.1$ the shape of the input distribution starts to change. As ρ increases, the sides of the input distributions grow steeper, until $\rho = 0.5$ where the input distributions has reached a triangular shape, with more values around the zero point when compared to a regular uniform distribution. When increasing ρ even more from this point on, the shape of the input distributions slowly goes back to its original shape. This can also be seen, as the input distributions for $\rho = 0.1$ and $\rho = 0.9$ are shaped similarly.
This similar shape can be explained by the way of correlating the numbers. When calculating the scale factor b using (8) as demonstrated in Section 2.3.1, $\rho = 0.5$ results in $b = 1$, meaning that the two distributions used to make one of the correlated output distributions have equal weight, and when passing this point, the distribution used to correlate the two distributions will start weighing heavier. Using $\rho = 0.1$ and $\rho = 0.9$ as an example, their resulting scale factors b will be $\frac{1}{3}$ and 3 respectively. In the case $\rho = 0.9$, one could say that the common distribution has weight 3, and the other distribution weight 1. Where using $\rho = 0.1$, the common distribution has weight $\frac{1}{3}$, and the other distribution weight 1. Instead one could also say for the case of $\rho = 0.1$, the common distribution has weight 1, and the other distribution weight 3, showing the same weights as compared to the case of $\rho = 0.9$, only flipped for the two distributions, resulting in two equal looking distributions in Fig. 12.



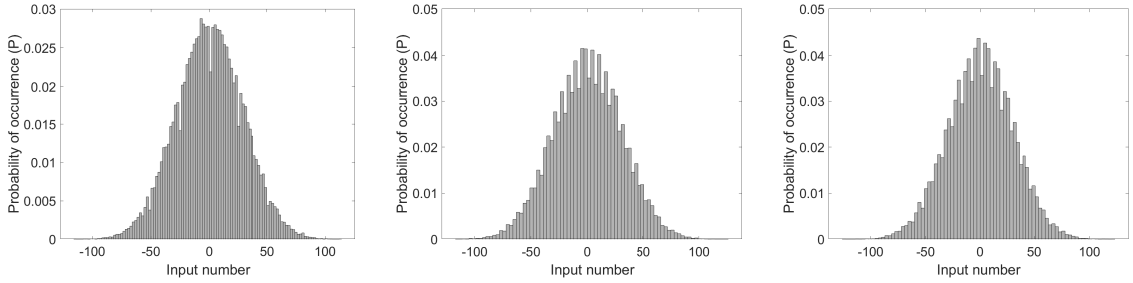
(a) ME (b) MSE (c) MAPE

Figure 13: Errors for uniform input distributions for different ρ

The errors for different correlation factors ρ are shown in the figures above. It looks like the ME has no dependency on ρ, as it is both positive and negative for different ρ, without showing any recognizable pattern. The plots for the MSE and MAPE look more interesting. The MSE appears to be lowest at $\rho = 0.5$ at the center, and higher when increasing or decreasing ρ from that point. For the MAPE seems to be the other way around.

4.2.3 Correlated inputs - normal

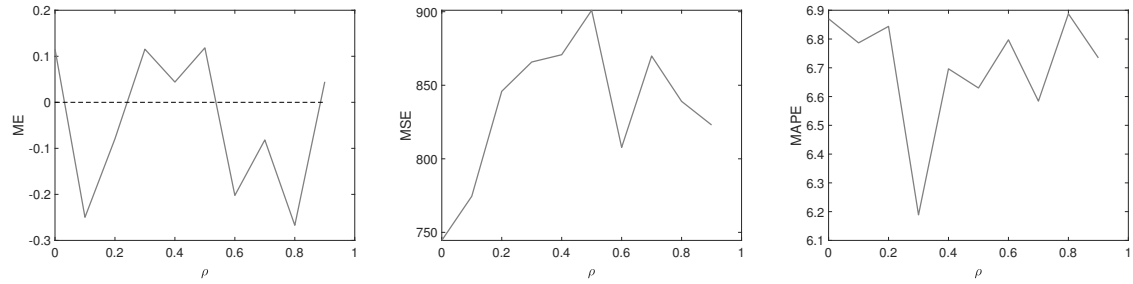
Several shapes of normal input distributions for different ρ values can be seen in Fig. 14. In contrast to the uniform input distributions, the normal input distributions seem to have equal shapes for different ρ.



(a) $\rho = 0.1$ (b) $\rho = 0.5$ (c) $\rho = 0.9$

Figure 14: Correlated normal input distributions

The errors for different correlation factors ρ are shown in the figures below. Again, it looks like the ME has no dependency on ρ, as it is both positive and negative for different ρ, without showing any recognizable pattern. In contrast to the case of the uniform input distributions, the MSE and MAPE also seem to have no dependency on ρ, as their shape appears random without any recognizable pattern.



(a) ME (b) MSE (c) MAPE

Figure 15: Errors for normal input distributions for different ρ

5 Discussion on quality and cost trade-off

Looking at the reduction in hardware as shown in Section 4.1, the decrease in hardware usage of about 6% does not seem very significant, and raises the question if this reduction in hardware is worth the degraded quality of the multiplier. The fact that only a very small amount of hardware is saved by this specific approximate design, is due to the scaling of the design proposed in [4]. Originally it seemed that this particular approximate multiplier design would yield larger decrease of hardware. However, this was actually only indicated for higher radix designs, which was one of the reasons to investigate this particular design specifically. Even though that it was indicated that the design could be scaled to as low as R4/R16, there are only slight gains in hardware size. The higher radix designs have way more drastic hardware decreases, as the size of the encoding hardware does not increase when going to higher radix designs. The paper proposed a design with boolean expressions scalable by a factor k, and these expressions do not change for higher radices, except for the index used on the input bits in those statements. So going to a higher radix design will result in more hardware savings, as in those cases more stages of the partial product accumulation tree can be skipped, resulting in way more significant hardware decreases.

Also, the design does not seem to be very well optimized for the R4/R16 case. The encoding expressions as shown in (5) in Section 2.2.1 clearly show the presence of the $b_{\text{-}1}$ digit, which should always be set to 0, as it is a non existing bit of the multiplicand. Given the fact that it is always set to 0, the encoding expressions can be revisited and further simplified by leaving out the $b_{\text{-}1}$ digit. This will most likely lead to some further decrease in hardware.

The actual quality results of the multiplier seem to be like originally expected, with a near-zero average error. Different input distributions, or different correlations between the input distributions also seem to have no effect on the mean error. This is due to the fact that all of the tested input distributions were generated to be symmetric around 0 ($\mu = 0$). It should be noted that the ME is dependent on the size of the distributions. When taking larger distributions, the error will most likely get closer to zero, and average out a bit lower than 0. This is due to how the encoding is done as proposed by table 3 in section 2.2.1. As can be seen the mf shows a behavior which is almost symmetric around 0, but leans slightly more towards a negative mf. When the size of the set of input numbers increases, the line of the ME as shown in Fig. 13a will most likely fluctuate less, and settle around an ME just under 0.

An interesting quality result that stands out can be seen in Fig. 13b and 13c. Here a specific influence of the correlation factor on both the means squared error and the mean absolute percentage error can be seen. The MSE is at its lowest at a correlation factor of 0.5, while the MAPE is at its highest at the same point. The reason for this can most likely be explained by the shapes of the error distributions which can be found in Appendix B. As can be seen, each of the different correlation factors, the peak at 0 error is approximately equally high in all the plots. This means that about the same amount of errors are generated from the different input distributions. However, it can be seen that in the case of input distributions correlated by a factor of 0.5, the errors are more centered around 0, while in the case of a correlation of 0.1 and 0.9 for example, the errors are more spread out, and therefore more errors with higher magnitudes will occur, which explains the shape of Fig. 13b. The shape of Fig. 13c however, is harder to explain. As the actual accurate value is used for normalization of the number, one cannot exactly explain the shape of Fig. 13c, as information about the accurate values as compared to the error is not shown there.

6 Conclusion

The goal of this work was to investigate the behavior of the signed multiplier design as proposed by [4] when scaled down, and if the design is suitable for application in a MAC circuit. It should be noted that a definitive verdict on whether this is a suitable multiplier for MAC can not be given, as this report does not show any actual MAC results, but only results from the multiplier which is to be used in such a MAC circuit. Keeping this in mind, and looking at the mean error of the multiplier, it can be said that this multiplier would be well suited for a MAC application, as it has a near-zero average error, for all of the different distributions demonstrated in Section 4: both uniform and normal, and both uncorrelated and correlated. In a MAC setting, this near-zero average error might get even closer to zero, due to the accumulation of the products inside a MAC. However, the approximation of the multiplier results in hardware savings of only 6%, which is not very significant, also when considering that the size of the accurate multiplier is only 135 logic elements. By saving only the small amount of 8 logic elements, a noticeable error is introduced. Therefore this specific approximate R4/R16 design does not seem very well suited for a MAC application.
However, it does not mean that this multiplier design is not worth exploring more for the MAC application. Now that all of the structures are set up, and proper VHDL and Matlab models are constructed, more time can be spent looking into the workings of the design, an potentially trying to change parts in the design for further hardware reduction. Potential ideas on future work are discussed in Section 7.

7 Recommendations for future work

The design discussed in this report leaves a lot of room for further investigation and improvement. Several investigation options to further increase quality or decrease cost are listed below.

- A higher radix design can be explored, so a bigger gain in hardware cost is acquired, but most likely at a significantly lower computational quality. The higher radix design is interesting, as going to a higher radix design does not need any bigger encoding hardware for the higher radix part, while still the amount of partial products, and therefore also the hardware, is decreased.
- To get more insight in the actual performance in a MAC circuit of the multiplier design as discussed in this report, the Matlab model can be expanded with functionality to simulate a MAC circuit. This way the behavior of the errors for different input distributions and the influence of correlation factors can be explored in a setting closer to the actual purpose of the multiplier.
- An attempt can be made to rewrite the encoding statements, targeting less hardware usage, and tuning the mean error to a positive side for one design, and the negative side for an other design, so the errors of both design could potentially cancel each other in a MAC circuit.

References

[1] 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)*, pp. 1–6, June 2016.

[2] L. R. D'Addario and D. Wang, "An integrated circuit for radio astronomy correlators supporting large arrays of antennas," *Journal of Astronomical Instrumentation*, p. 1650002, 03 2016.

[3] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, J. Henkel, and J. Henkel, "Architectural-space exploration of approximate multipliers," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, Nov 2016.

[4] V. Leon, G. Zervakis, D. Soudris, and K. Pekmestzi, "Approximate hybrid high radix encoding for energy-efficient inexact multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, pp. 421–430, March 2018.

Appendices

A Model validation

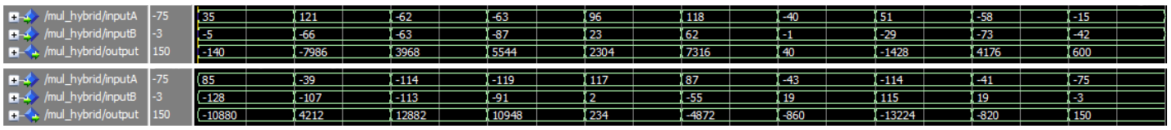
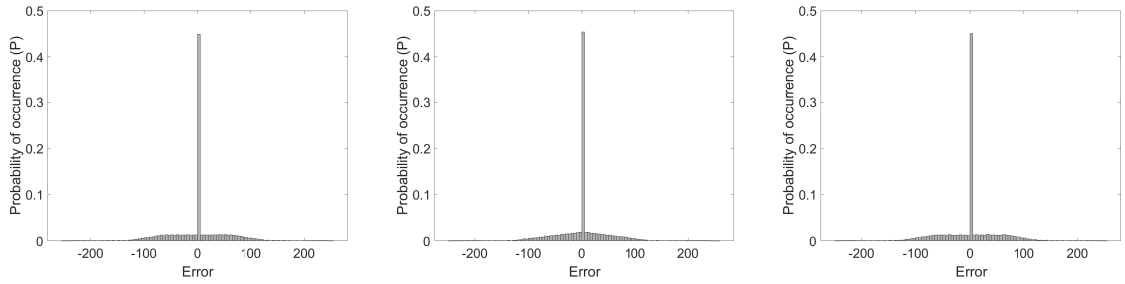


Figure 16: Modelsim validation simulation wave output

```
>> model_validation
35 * -5 = -140
121 * -66 = -7986
-62 * -63 = 3968
-63 * -87 = 5544
96 * 23 = 2304
118 * 62 = 7316
-40 * -1 = 40
51 * -29 = -1428
-58 * -73 = 4176
-15 * -42 = 600
85 * -128 = -10880
-39 * -107 = 4212
-114 * -113 = 12882
-119 * -91 = 10948
117 * 2 = 234
87 * -55 = -4872
-43 * 19 = -860
-114 * 115 = -13224
-41 * 19 = -820
-75 * -3 = 150
```

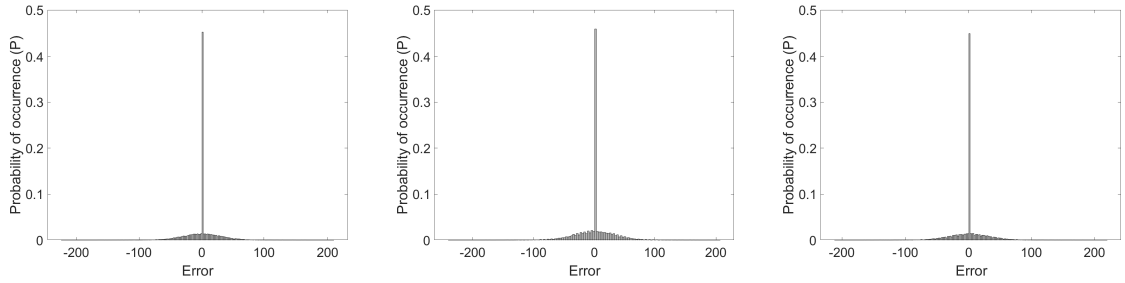
Figure 17: Matlab validation command window output

B Error histograms for correlated inputs



(a) $\rho = 0.1$ (b) $\rho = 0.5$ (c) $\rho = 0.9$

Figure 18: Error histograms for correlated uniform input distributions



(a) $\rho = 0.1$ (b) $\rho = 0.5$ (c) $\rho = 0.9$

Figure 19: Error histograms for correlated normal input distributions

C Matlab code

Matlab main

```
%% INITIALIZE
clear;
close all;

global font_size; %font size used for plot axis
font_size = 20;

%if the following variables are 1,
%error metrics/plots will be printed/plotted
unif_cor = 0; %correlated uniform input distributions
norm_cor = 1; %correlated normal input distributions
inpt_dtr = 0; %uncorrelated perfect uniform/uniform/normal distributions

%% CORRELATED UNIFORM DISTRIBUTION
if unif_cor

    %initialize arrays
    me = zeros(1,10);
    mse = zeros(1,10);
    mape = zeros(1,10);

    for i = 0:9
        %multiplications will be done on the data in dat_container
        dat_container = uniform_distr_correlated(i/10);     %get uniform distr
        result_array = compute_result_array(dat_container); %compute approx results

        %create histograms showing the distribution of inputs and errors
        input_histogram(dat_container,['Correlated uniformly distributed input,'...
            ' \rho = ', num2str(i/10)]);
        error_histogram(result_array(3,:),['Error histogram correlated uniform'...
            ' distribution, \rho = ', num2str(i/10)]);

        %print error metrics
        text = ['CORRELATED UNIFORM DISTRIBUTION, rho = ', num2str(i/10), '\n'];
        fprintf(text);
        [me(i+1),mse(i+1),mape(i+1)] = print_errors(result_array);
    end

    %plot ME against rho
    figure('Name','Mean error for different rho','NumberTitle','off');
    plot(0:.1:.9,me,'Color',RGB);
    hold on;
    plot(get(gca, 'XLim'), [0 0], '--k')  %add dashed reference line at 0
    title('Mean error for different \rho');
    xlabel('\rho');
    ylabel('ME');
    set(gcf, 'Position', [100, 100, 800, 600]);
    set(findobj(gca, 'Type', 'Line', 'Linestyle', '-'), 'LineWidth', 2);
    set(gca,'fontsize', font_size);

    %plot MSE against rho
    figure('Name','Mean squared error for different rho','NumberTitle','off');
    plot(0:.1:.9,mse,'Color',RGB);
    title('Mean squared error for different \rho');
    xlabel('\rho');
    ylabel('MSE');
    set(gcf, 'Position', [100, 100, 800, 600]);
    set(findobj(gca, 'Type', 'Line', 'Linestyle', '-'), 'LineWidth', 2);
    set(gca,'fontsize', font_size);

    %plot MAPE against rho
    figure('Name','Mean absolute percentage error for different rho','NumberTitle','off');
    plot(0:.1:.9,mape,'Color',RGB);
    title('Mean absolute percentage error for different \rho');
    xlabel('\rho');
    ylabel('MAPE');
```

```
67      set(gcf, 'Position', [100, 100, 800, 600]);
68      set(findobj(gca, 'Type', 'Line', 'Linestyle', '-'), 'LineWidth', 2);
69      set(gca,'fontsize',font_size);
70
71  end
72  %% CORRELATED NORMAL DISTRIBUTION
73  if norm_cor
74
75      %initialize arrays
76      me = zeros(1,10);
77      mse = zeros(1,10);
78      mape = zeros(1,10);
79
80      for i = 0:9
81          %multiplications will be done on the data in dat_container
82          dat_container = normal_distr_correlated(i/10);      %get normal distr
83          result_array = compute_result_array(dat_container); %compute approx results
84
85
86          %create histograms showing the distribution of inputs and errors
87          input_histogram(dat_container,['Correlated normally distributed input,'...
88              ' \rho = ', num2str(i/10)]);
89          error_histogram(result_array(3,:),['Error histogram correlated normal'...
90              ' distribution, \rho = ', num2str(i/10)]);
91
92          %print error metrics
93          text = ['CORRELATED NORMAL DISTRIBUTION, rho = ', num2str(i/10), '\n'];
94          fprintf(text);
95          [me(i+1),mse(i+1),mape(i+1)] = print_errors(result_array);
96      end
97
98      %plot ME against rho
99      figure('Name','Mean error for different rho','NumberTitle','off');
100     plot(0:.1:.9,me,'Color',RGB);
101     hold on;
102     plot(get(gca, 'XLim'), [0 0], '--k')  %add dashed reference line at 0
103         title('Mean error for different \rho');
104     xlabel('\rho');
105     ylabel('ME');
106     set(gcf, 'Position', [100, 100, 800, 600]);
107     set(findobj(gca, 'Type', 'Line', 'Linestyle', '-'), 'LineWidth', 2);
108     set(gca,'fontsize',font_size);
109
110     %plot MSE against rho
111     figure('Name','Mean squared error for different rho','NumberTitle','off');
112     plot(0:.1:.9,mse,'Color',RGB);
113     title('Mean squared error for different \rho');
114     xlabel('\rho');
115     ylabel('MSE');
116     set(gcf, 'Position', [100, 100, 800, 600]);
117     set(findobj(gca, 'Type', 'Line', 'Linestyle', '-'), 'LineWidth', 2);
118     set(gca,'fontsize',font_size);
119
120     %plot MAPE against rho
121     figure('Name','Mean absolute percentage error for different rho','NumberTitle','off');
122     plot(0:.1:.9,mape,'Color',RGB);
123     title('Mean absolute percentage error for different \rho');
124     xlabel('\rho');
125     ylabel('MAPE');
126     set(gcf, 'Position', [100, 100, 800, 600]);
127     set(findobj(gca, 'Type', 'Line', 'Linestyle', '-'), 'LineWidth', 2);
128     set(gca,'fontsize',font_size);
129
130 end
131 %% PERFECT UNIFORM DISTRIBUTION
132 if inpt_dtr
133
134     %multiplications will be done on the data in dat_container
135     dat_container = perfect_uniform_distr();
136     result_array = compute_result_array(dat_container);
137
```

```
138     %create histograms showing the distribution of inputs and errors
139     input_histogram(dat_container,'Perfect uniformly distributed input');
140     error_histogram(result_array(3,:),'Error histogram perfect uniform distribution');
141 
142     %print error metrics
143     fprintf('PERFECT UNIFORM DISTRIBUTION\n');
144     print_errors(result_array);
145 
146     %% UNIFORM DISTRIBUTION
147 
148     %multiplications will be done on the data in dat_container
149     dat_container = uniform_distr();
150     result_array = compute_result_array(dat_container);
151 
152     %create histogram showing the distribution of errors
153     input_histogram(dat_container,'Uniformly distributed input');
154     error_histogram(result_array(3,:),'Error histogram uniform distribution');
155 
156     %print error metrics
157     fprintf('UNIFORM DISTRIBUTION\n');
158     print_errors(result_array);
159 
160     %% NORMAL DISTRIBUTION
161 
162     %multiplications will be done on the data in dat_container
163     dat_container = normal_distr();
164     result_array = compute_result_array(dat_container);
165 
166     %create histogram showing the distribution of errors
167     input_histogram(dat_container,'Normally distributed input');
168     error_histogram(result_array(3,:),'Error histogram normal distribution (\sigma = 40)');
169 
170     %print error metrics
171     fprintf('NORMAL DISTRIBUTION\n');
172     print_errors(result_array);
173     %}
174 
175 end
176 %% FUNCTIONS
177 
178 %this function sets the RGB code for the color to be used in plots
179 function output = RGB
180     output = [.5,.5,.5]; %grey
181 end
182 
183 %this functions returns an array with on index:
184 %1:accurate result
185 %2:approximate result
186 %3:error
187 %of the given inputs
188 function output = compute_result_array(distr)
189     return_array = zeros(3,2^16);
190     for i = 1:2^16
191         %accurate result
192         return_array(1,i) = distr(1,i) * distr(2,i);
193         %approximate result
194         return_array(2,i) = mul_approx(distr(1,i),distr(2,i));
195         %error
196         return_array(3,i) = return_array(1,i) - return_array(2,i);
197     end
198     output = return_array;
199 end
200 
201 %function for plotting an error histogram of the given error data
202 function error_histogram(func_dat,func_name)
203     %create histogram showing the distribution of errors
204     figure('Name',func_name,'NumberTitle','off');
205     h = histogram(func_dat,'Normalization','probability');
206     h.FaceColor = RGB;
207     %title(func_name);
208     xlabel('Error');
```

```
    ylabel('Probability of occurrence (P)');
    %set(get(gca,'ylabel'),'rotation',0);
    set(gcf, 'Position', [100, 100, 800, 600]);
    global font_size;
    set(gca,'fontsize',font_size);
end

%functino for plotting input histrograms of the given input data
function input_histogram(func_dat,func_name)
    %create histogram showing the distribution of errors
    figure('Name',func_name,'NumberTitle','off');
    h = histogram(func_dat(1,:),'Normalization','probability');
    h.FaceColor = RGB;
    %title(func_name);
    xlabel('Input number');
    ylabel('Probability of occurrence (P)');
    %set(get(gca,'ylabel'),'rotation',0);
    set(gcf, 'Position', [100, 100, 800, 600]);
    global font_size;
    set(gca,'fontsize',font_size)
end

%function for generating a 'perfect' uniform distribution
%with where ever possible combination of inputs occurs exactly once
function output = perfect_uniform_distr()
    data = zeros(2,2^16);
    cnt = 0; %used for indexing the two datasets
    for i = -128:127
        for j = -128:127
            cnt = cnt + 1;
            data(1,cnt) = i;
            data(2,cnt) = j;
        end
    end
    output = data;
end

%function for calcuatling error metrics, printing them to the command
%window, and returning the ME, MSE, and MAPE
function [mean_error, mean_sq_error,mean_abs_perc_error] = print_errors(dat_array)
    mean_error = sum(dat_array(3,:))/2^16;          %calculate ME
    mean_abs_error = sum(abs(dat_array(3,:)))/2^16; %calculate MAE
    mean_sq_error = sum(dat_array(3,:).^2)/2^16;    %calculate MSE
    mean_perc_error = 0;                            %initialize MPE
    mean_abs_perc_error = 0;                        %initialize MAPE
        for i = 1:2^16
        if(dat_array(1,i) ~= 0 && dat_array(3,i) ~= 0)
            %calcualte PE and APE (no mean yet)
            mean_perc_error = mean_perc_error + (dat_array(3,i)/dat_array(1,i))*100;
            mean_abs_perc_error = mean_abs_perc_error + abs(dat_array(3,i)/dat_array(1,i))*100;
        end
        end
    mean_perc_error = mean_perc_error/2^16;         %find mean of the set
    mean_abs_perc_error = mean_abs_perc_error/2^16; %find mean of the set

    %print all errors to the command window
    fprintf('Mean error                   : %.5f\n', mean_error);
    fprintf('Mean absolute error          : %.5f\n', mean_abs_error);
    fprintf('Mean squared error           : %.5f\n', mean_sq_error);
    fprintf('Mean relative error          : %.5f\n', mean_perc_error);
    fprintf('Mean absolute relative error : %.5f\n', mean_abs_perc_error);
    fprintf('\n');
end
```

Accurate R4 encoding and partial product generation

```
%function for the accurate R4 partial product generation
function [output,signfac] = ppR4_gen(a,b,j)
    %prepare the 3 bits for the R4 case with a special case of j==0
    if(j == 0)
```

```
        b_input = [0,b(1),b(2)];
    else
        b_input = [b(2*j),b(2*j+1),b(2*j+2)];
    end

    %calculate multiplicaton parameters
    sign = b_input(3);
    x1 = times1(b_input);
    x2 = times2(b_input);

    %sign factor
    signfac = and(sign,or(x1,x2));

    %prepare a partial product array
    pp = zeros(1,9);

    %calculate each partial product bit
    pp(1) = ppi_structure([0,a(1)], sign, x1, x2);
    pp(2) = ppi_structure([a(1),a(2)], sign, x1, x2);
    pp(3) = ppi_structure([a(2),a(3)], sign, x1, x2);
    pp(4) = ppi_structure([a(3),a(4)], sign, x1, x2);
    pp(5) = ppi_structure([a(4),a(5)], sign, x1, x2);
    pp(6) = ppi_structure([a(5),a(6)], sign, x1, x2);
    pp(7) = ppi_structure([a(6),a(7)], sign, x1, x2);
    pp(8) = ppi_structure([a(7),a(8)], sign, x1, x2);
    pp(9) = ppi_structure([a(8),a(8)], sign, x1, x2);

    %return
    output = pp;
end

%function for the partial product generation
function output = ppi_structure(a, sign, x1, x2)
    Ax1 = xor(sign,a(2));
    Ax2 = xor(sign,a(1));
    output = or(and(x2,Ax2),and(x1,Ax1));
end

%encoding bit x1
function output = times1(b)
    output = xor(b(2),b(1));
end

%encoding bit x2
function output = times2(b)
    output = and(xor(b(3),b(2)),not(xor(b(1),b(2))));
end
```

Approximate R16 encoding and partial product generation

```
%function for the approximate R16 partial product generation
function [output, signfac] = pp_gen(a,b)
    %for the R16 case the LSB is zero
    b_input = [0,b(1),b(2),b(3),b(4)];

    %calculate multiplicaton parameters
    sign = b_input(5);
    x1 = times1(b_input);
    x2 = times2(b_input);
    x4 = times4(b_input);
    x8 = times8(b_input);

    %sign factor
    signfac = and(sign,or(or(x1,x2),or(x4,x8)));

    %prepare a partial product array
    pp = zeros(1,11);

    %calculate each partial product bit
    pp(1) = ppi_structure([0,0,0,a(1)], sign, x1, x2, x4, x8);
```

```
    pp(2) = ppi_structure([0,0,a(1),a(2)], sign, x1, x2, x4, x8);
    pp(3) = ppi_structure([0,a(1),a(2),a(3)], sign, x1, x2, x4, x8);
    pp(4) = ppi_structure([a(1),a(2),a(3),a(4)], sign, x1, x2, x4, x8);
    pp(5) = ppi_structure([a(2),a(3),a(4),a(5)], sign, x1, x2, x4, x8);
    pp(6) = ppi_structure([a(3),a(4),a(5),a(6)], sign, x1, x2, x4, x8);
    pp(7) = ppi_structure([a(4),a(5),a(6),a(7)], sign, x1, x2, x4, x8);
    pp(8) = ppi_structure([a(5),a(6),a(7),a(8)], sign, x1, x2, x4, x8);
    pp(9) = ppi_structure([a(6),a(7),a(8),a(8)], sign, x1, x2, x4, x8);
    pp(10) = ppi_structure([a(7),a(8),a(8),a(8)], sign, x1, x2, x4, x8);
    pp(11) = ppi_structure([a(8),a(8),a(8),a(8)], sign, x1, x2, x4, x8);

    %return
    output = pp;
end

%function for the partial product generation
function output = ppi_structure(a, sign, x1, x2, x4, x8)
    Ax1 = xor(sign,a(4));
    Ax2 = xor(sign,a(3));
    Ax4 = xor(sign,a(2));
    Ax8 = xor(sign,a(1));
    output = or(or(and(Ax1,x1),and(Ax2,x2)),or(and(Ax4,x4),and(Ax8,x8)));
end

%encoding bit x1
function output = times1(b)
        output = and(and(b(2),b(3)),and(b(4),b(5)));
end

%encoding bit x2
function output = times2(b)
    p1 = and(and(not(b(5)),not(b(4))),or(and(and(not(b(3)),b(2)),b(1)),and(b(3),not(b(2)))));
    p2 = and(and(b(5),b(4)),or(and(and(b(3),not(b(2))),not(b(1))),and(not(b(3)),b(2))));
    output = or(p1,p2);
end

%encoding bit x4
function output = times4(b)
    p1 = and(and(not(b(4)),b(3)),or(b(5),b(2)));
    p2 = and(and(b(4),not(b(3))),or(not(b(5)),not(b(2))));
    output = or(p1,p2);
end

%encoding bit x8
function output = times8(b)
    p1 = and(and(not(b(5)),b(4)),b(3));
    p2 = and(and(b(5),not(b(4))),not(b(3)));
    output = or(p1,p2);
end
```

Accurate R4 multiplier

```
%function representing the accurate R4 multiplier structure
function output = mul_acc(input1, input2)
    numbits = 8;

    %convert input numbers to their binary signed equivalent
    varA = de2bisi(input1, numbits);
    varB = de2bisi(input2, numbits);

    %partial product calculation
    [ppE0,sf_0] = ppR4_gen(varA,varB,0);
    [ppE1,sf_1] = ppR4_gen(varA,varB,1);
    [ppE2,sf_2] = ppR4_gen(varA,varB,2);
    [ppE3,sf_3] = ppR4_gen(varA,varB,3);

    %partial product extension
    ppE0 = [ppE0, ppE0(9), ppE0(9), ppE0(9), ppE0(9), ppE0(9), ppE0(9), ppE0(9)];
    ppE1 = [sf_0, 0, ppE1, ppE1(9), ppE1(9), ppE1(9), ppE1(9), ppE1(9)];
    ppE2 = [0, 0, sf_1, 0, ppE2, ppE2(9), ppE2(9), ppE2(9)];
```

```
    ppE3 = [0, 0, 0, 0, sf_2, 0, ppE3, ppE3(9)];
    ppE4 = zeros(1,16);
    ppE4(7) = sf_3;

    %add pp's together for the accurate R4 case
    addR4_1 = bitadd(ppE0, ppE1, 0);
    addR4_2 = bitadd(addR4_1, ppE2, 0);
    addR4_3 = bitadd(addR4_2, ppE3, 0);
    final_R4 = bitadd(addR4_3, ppE4, 0);

    %convert the signed binary numbers back to decimal
    answerR4 = bi2desi(final_R4);

    %return
    output = answerR4;
end
```

Approximate hybrid R4/R16 multiplier

```
%function representing the approximate R4/R16 multiplier structure
function output = mul_approx(input1, input2)
    numbits = 8;

    %convert input numbers to their binary signed equivalent
    varA = de2bisi(input1, numbits);
    varB = de2bisi(input2, numbits);

    %partial product calculation
    [ppA, sf_A] = pp_gen(varA,varB);
    [ppE2,sf_2] = ppR4_gen(varA,varB,2);
    [ppE3,sf_3] = ppR4_gen(varA,varB,3);

    %partial product extension
    ppA = [ppA, ppA(11), ppA(11), ppA(11), ppA(11), ppA(11)];
    ppE2 = [sf_A, 0, 0, 0, ppE2, ppE2(9), ppE2(9), ppE2(9)];
    ppE3 = [0, 0, 0, 0, sf_2, 0, ppE3, ppE3(9)];
    ppE4 = zeros(1,16);
    ppE4(7) = sf_3;

    %add pp's together for the approx hybrid case (R4 and R16)
    add = bitadd(ppA, ppE2, 0);
    add2 = bitadd(add, ppE3, 0);
    final = bitadd(add2, ppE4, 0);

    %convert the signed binary numbers back to decimal
    answer = bi2desi(final);

    %return
    output = answer;
end
```

Binary to decimal signed

```
%function performs the same as bi2de, but then with signed notation
function output = bi2desi(a)
    [check,length] = size(a);
    if(check ~= 1)
        error('size problem, vector must be of size 1xA');
    end
    neg = a(length);   %check sign bit
    if(neg)
        a = twocom(a); %if negative number, find two's complement
    end
    number = bi2de(a); %find the decimal number
    if(neg)
        %make the number negative if the sign bit was high
        number = number * (-1);
    end
```

```
    output = number;
end
```

Decimal to binary signed

```
%function performs the same as de2bi, but then with signed notation
function output = de2bisi(a, numbits)
    neg = 0;          %initialize a check for negative numbers
    if(a < 0)         %if the number is negative
        a = a * (-1); %make the number positive
        neg = 1;      %and indicate the presence of a negative number
    end
    a = de2bi(a);     %then find the binary representation (only pos numbers)
    if(neg)
        a = twocom(a);%if the number was negative, now take the 2's complement
    end

    %if the number is not as long as the specified length, extend it
    [~,length] = size(a);
    for i = length+1:numbits
        a(i) = neg;
    end
    output = a;
end
```

Bitwise binary adder

```
%function for performing a bitwise addition on an array with 1's/0's
function output = bitadd(a,b,select)
    [l1,l2] = size(a);
    [l3,l4] = size(b);

    %perform checks on the length of the input vectors, so now most
    %cases should be properly handled
    if(l1 == 1 && l2 == 1 && l3 == 1 && l4 == 1)
        output = xor(a,b);
        return;
    elseif(l1 ~= l3 || l2 ~= l4)
        error('input sizes not equal');
    elseif(l1 == 1)
        siz = l2;
    elseif(l2 == 1)
        siz = l1;
    else
        error('size should be 1xA or Ax1');
    end

    output = zeros(1,siz); %initialize output array
    c_in = 0;              %intialize the first carry in

    %for loop to iterate over all of the bits
    for i = 1:siz
        %the switch statement selects a type of full adder
        %some approximate full adders are included as well
        switch(select)
            case 0                                      %accurate adder
                [s,c] = fulladder(a(i),b(i),c_in);
            case 1                                      %approx adder
                [s,c] = fulladder_A1(a(i),b(i),c_in);
            case 2                                      %approx adder
                [s,c] = fulladder_A2(a(i),b(i),c_in);
            case 3                                      %approx adder
                [s,c] = fulladder_A3(a(i),b(i),c_in);
            otherwise                                   %no case specified
                error('no appropriate case specified');
        end
        c_in = c;
        output(i) = s; %set output bits
```

```
    end
end

%function representing an accurate 1 bit full-adder
function [s_o,c_o] = fulladder(a,b,c)
    s_o = xor(xor(a,b),c);              %calculate sum out
    c_o = or(and(xor(a,b),c),and(a,b)); %calculate carry out
end

%function representing a state of the art approximate 1 bit full-adder
function [s_o,c_o] = fulladder_A1(a,b,c)
          s_o = not(or(and(a,b),and(or(a,b),c))); %calculate sum out
    c_o = or(and(a,b),and(or(a,b),c));      %calculate carry out
end

%function representing a state of the art approximate 1 bit full-adder
function [s_o,c_o] = fulladder_A2(a,b,c)
          s_o = and(not(b),or(not(a),not(c)));      %calculate sum out
    c_o = not(and(not(b),or(not(a),not(c)))); %calculate carry out
end

%function representing a state of the art approximate 1 bit full-adder
function [s_o,c_o] = fulladder_A3(a,b,~)
          s_o = b; %calculate sum out
    c_o = a; %calculate carry out
end
```

Uniform distribution

```
%function for generating a uniform distribution
%with values only in the range of -128 to 127
function output = uniform_distr()
    data = randi([-128 127],2,2^16);
    output = data;
end
```

Uniform distribution correlated

```
%function for generating correlated uniform distributions
%with values only in the range of -128 to 127, and a correlation factor
function output = uniform_distr_correlated(relation)
    if(relation < 0 || relation >= 1)
       error('Correlation parameter must be >= 0 and < 1');
    end
    %get uniform distribution
    data = randi([-128 127],2,2^16);

    %get a uniform distribution for the correlation
    cor_data = randi([-128 127],1,2^16)*sqrt(relation/(1-relation));

    %correlate the data
    data = data + cor_data;

    %scale data back to a range of -128 to 127
    data = round((127/max(data(:)))*data);

    %check if the numbers are really in those bounds
    if(max(data(:)) > 127 || min(data(:)) < -128)
        %try again when the max numbers are out of bounds
        output = uniform_distr_correlated(relation);
    else
        output = data;
    end
end
```

Normal distribution

```
%function for generating a normal distribution
%with values only in the range of -128 to 127
function output = normal_distr()
    dstr_mean = 0;   %normal distr mean
    dstr_sigma = 40; %normal distr standard deviation
    data = round(normrnd(dstr_mean,dstr_sigma,2,2^16));
    for i = 1:2^16   %loop over the data to find out of bounds values
        if(data(1,i) > 127 || data(1,i) < -128)
            %redistribute the out of bounds value uniformly
            data(1,i) = randi([-128,127]);
        end
        if(data(2,i) > 127 || data(2,i) < -128)
            %redistribute the out of bounds value uniformly
            data(2,i) = randi([-128,127]);
        end
    end
    output = data;
end
```

Normal distribution correlated

```
%function for generating correlated normal distributions
%with values only in the range of -128 to 127, and a correlation factor
function output = normal_distr_correlated(relation)
    if(relation < 0 || relation >= 1)
       error('Correlation parameter must be >= 0 and < 1');
    end

    %generate independent dstr first
    data = normal_distr();
    cor_data = normal_distr();
    cor_data = cor_data(1,:); %take only one dstr used for correlation
    data = data + cor_data;   %correlate the data

    %scale data back to a range of -128 to 127
    data = round((127/max(data(:)))*data);

    %check if the numbers are really in those bounds
    if(max(data(:)) > 127 || min(data(:)) < -128)
        %try again when the max numbers are out of bounds
        output = normal_distr_correlated(relation);
    else
        output = data;
    end
end
```

Find 2's complement

```
%function for finding 2's complement of array with 1's/0's
function output = twocom(a)
    [~,length] = size(a);
    a = ~a;                 %invert a for 1's complement
    b = zeros(1,length);
    b(1) = 1;
    output = bitadd(a,b,0); %add one for 2's complement
end
```

Model validation

```
%create two sets of 20 random samples
dat = randi([-128 127],2,20);

%use the approximate multiplier on the data, and print the results
```

```
5  for i = 1:20
6      fprintf('%d * %d = %d\n', dat(1,i),dat(2,i),...
7          mul_approx(dat(1,i),dat(2,i)));
8  end
9  fprintf('\n');
10
11 %print commands which can be used in a .do file for modelsim,
12 %using the same numbers as above, so the models can be compared
13 for i = 1:20
14     dat1 = de2bisi(dat(1,i),8);
15     dat2 = de2bisi(dat(2,i),8);
16     fprintf('force inputA %d%d%d%d%d%d%d%d\n',dat1(8),dat1(7),...
17         dat1(6),dat1(5),dat1(4),dat1(3),dat1(2),dat1(1));
18     fprintf('force inputB %d%d%d%d%d%d%d%d\n',dat2(8),dat2(7),...
19         dat2(6),dat2(5),dat2(4),dat2(3),dat2(2),dat2(1));
20     fprintf('run 10 ns\n\n');
21 end
```

D VHDL code

Accurate R4 encoding and partial product generation

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  ENTITY ppR4_gen IS
6          GENERIC(
7                  numbits : integer := 8
8          );
9          PORT(
10                 inputA     : IN std_logic_vector (numbits-1 downto 0);
11                 inputB     : IN std_logic_vector (2 downto 0);
12                 output     : OUT std_logic_vector (numbits downto 0);
13                 signfactor : OUT std_logic
14         );
15 END ppR4_gen;
16
17 ARCHITECTURE arch_ppR4_gen OF ppR4_gen IS
18         --encoding bits
19         SIGNAL x1 : std_logic := '0';
20         SIGNAL x2 : std_logic := '0';
21         SIGNAL sign : std_logic := '0';
22 BEGIN
23         --multiplication factors (encoding bits)
24         x1 <= inputB(0) XOR inputB(1);
25         x2 <= (inputB(1) XOR inputB(2)) AND (NOT (inputB(0) XOR inputB(1)));
26         sign <= inputB(2);
27
28         --sign factor (to get to the 2's complement)
29         signfactor <= sign AND (x1 OR x2);
30
31         --bitwise partial product generation (1's complement)
32         output(0) <= (x1 AND (sign XOR inputA(0))) OR (x2 AND (sign XOR '0'));
33         output(1) <= (x1 AND (sign XOR inputA(1))) OR (x2 AND (sign XOR inputA(0)));
34         output(2) <= (x1 AND (sign XOR inputA(2))) OR (x2 AND (sign XOR inputA(1)));
35         output(3) <= (x1 AND (sign XOR inputA(3))) OR (x2 AND (sign XOR inputA(2)));
36         output(4) <= (x1 AND (sign XOR inputA(4))) OR (x2 AND (sign XOR inputA(3)));
37         output(5) <= (x1 AND (sign XOR inputA(5))) OR (x2 AND (sign XOR inputA(4)));
38         output(6) <= (x1 AND (sign XOR inputA(6))) OR (x2 AND (sign XOR inputA(5)));
39         output(7) <= (x1 AND (sign XOR inputA(7))) OR (x2 AND (sign XOR inputA(6)));
40         output(8) <= (x1 AND (sign XOR inputA(7))) OR (x2 AND (sign XOR inputA(7)));
41
42 END arch_ppR4_gen;
```

Approximate R16 encoding and partial product generation

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY ppR16_gen IS
        GENERIC(
                numbits : integer := 8
        );
        PORT(
                inputA      : IN std_logic_vector (numbits-1 downto 0);
                inputB      : IN std_logic_vector (3 downto 0);
                output      : OUT std_logic_vector (numbits+2 downto 0);
                signfactor  : OUT std_logic
        );
END ppR16_gen;

ARCHITECTURE arch_ppR16_gen OF ppR16_gen IS
        --encoding bits
        SIGNAL x1 : std_logic;
        SIGNAL x2 : std_logic;
        SIGNAL x4 : std_logic;
        SIGNAL x8 : std_logic;
        SIGNAL sign : std_logic;
BEGIN
        --multiplication factors (encoding bits)
        x1 <= inputB(0) AND inputB(1) AND inputB(2) AND inputB(3);
        x2 <= (NOT inputB(3) AND NOT inputB(2) AND (( NOT inputB(1) AND inputB(0) AND
               '0') OR (inputB(1) AND NOT inputB(0)))) OR (inputB(3) AND inputB(2) AND
             ((inputB(1) AND NOT inputB(0) AND '1') OR (NOT inputB(1) AND inputB(0))));
        x4 <= (NOT inputB(2) AND inputB(1) AND (inputB(3) OR inputB(0))) OR (inputB(2)
               AND NOT inputB(1) AND (NOT inputB(3) OR NOT inputB(0)));
        x8 <= (NOT inputB(3) AND inputB(2) AND inputB(1)) OR (inputB(3) AND NOT inputB(2)
               AND NOT inputB(1));
        sign <= inputB(3);

        --sign factor (to get to the 2's complement)
        signfactor <= sign AND (x1 OR x2 OR x4 OR x8);

        --bitwise partial product generation (1's complement)
        output(0) <= (x1 AND (sign XOR inputA(0))) OR (x2 AND (sign XOR '0'))
                      OR (x4 AND (sign XOR '0')) OR (x8 AND (sign XOR '0'));
        output(1) <= (x1 AND (sign XOR inputA(1))) OR (x2 AND (sign XOR inputA(0)))
                      OR (x4 AND (sign XOR '0')) OR (x8 AND (sign XOR '0'));
        output(2) <= (x1 AND (sign XOR inputA(2))) OR (x2 AND (sign XOR inputA(1)))
                      OR (x4 AND (sign XOR inputA(0))) OR (x8 AND (sign XOR '0'));
        output(3) <= (x1 AND (sign XOR inputA(3))) OR (x2 AND (sign XOR inputA(2)))
                      OR (x4 AND (sign XOR inputA(1))) OR (x8 AND (sign XOR inputA(0)));
        output(4) <= (x1 AND (sign XOR inputA(4))) OR (x2 AND (sign XOR inputA(3)))
                      OR (x4 AND (sign XOR inputA(2))) OR (x8 AND (sign XOR inputA(1)));
        output(5) <= (x1 AND (sign XOR inputA(5))) OR (x2 AND (sign XOR inputA(4)))
                      OR (x4 AND (sign XOR inputA(3))) OR (x8 AND (sign XOR inputA(2)));
        output(6) <= (x1 AND (sign XOR inputA(6))) OR (x2 AND (sign XOR inputA(5)))
                      OR (x4 AND (sign XOR inputA(4))) OR (x8 AND (sign XOR inputA(3)));
        output(7) <= (x1 AND (sign XOR inputA(7))) OR (x2 AND (sign XOR inputA(6)))
                      OR (x4 AND (sign XOR inputA(5))) OR (x8 AND (sign XOR inputA(4)));
        output(8) <= (x1 AND (sign XOR inputA(7))) OR (x2 AND (sign XOR inputA(7)))
                      OR (x4 AND (sign XOR inputA(6))) OR (x8 AND (sign XOR inputA(5)));
        output(9) <= (x1 AND (sign XOR inputA(7))) OR (x2 AND (sign XOR inputA(7)))
                      OR (x4 AND (sign XOR inputA(7))) OR (x8 AND (sign XOR inputA(6)));
        output(10) <= (x1 AND (sign XOR inputA(7))) OR (x2 AND (sign XOR inputA(7)))
                       OR (x4 AND (sign XOR inputA(7))) OR (x8 AND (sign XOR inputA(7)));

END arch_ppR16_gen;
```

Accurate R4 multiplier with Wallace tree

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  ENTITY mul_hybrid IS
6          GENERIC(
7                  numbits : integer := 8
8          );
9          PORT(
10                 inputA : IN  std_logic_vector (numbits-1 downto 0);
11                 inputB : IN  std_logic_vector (numbits-1 downto 0);
12                 output : OUT  std_logic_vector ((2*numbits)-1 downto 0)
13         );
14 END mul_hybrid;
15
16 ARCHITECTURE arc_mul_hybrid OF mul_hybrid IS
17
18 COMPONENT full_adder IS
19         PORT(
20                 a     : IN  std_logic;
21                 b     : IN  std_logic;
22                 c     : IN  std_logic;
23                 sum   : OUT  std_logic;
24                 carry : OUT  std_logic
25         );
26 END COMPONENT;
27
28 COMPONENT half_adder IS
29         PORT(
30                 a     : IN  std_logic;
31                 b     : IN  std_logic;
32                 sum   : OUT  std_logic;
33                 carry : OUT  std_logic
34         );
35 END COMPONENT;
36
37 COMPONENT ppR4_gen IS
38         PORT(
39                 inputA     : IN std_logic_vector (numbits-1 downto 0);
40                 inputB     : IN std_logic_vector (2 downto 0);
41                 output     : OUT std_logic_vector (numbits downto 0);
42                 signfactor : OUT std_logic
43         );
44 END COMPONENT;
45 --partial product signals
46 SIGNAL pp0,pp1,pp2,pp3 : std_logic_vector (numbits downto 0); --partial products
47 SIGNAL sf0,sf1,sf2,sf3 : std_logic; --sign factors
48
49 --partial producht inverse terms
50 SIGNAL pp0_inv,pp1_inv,pp2_inv,pp3_inv : std_logic;
51
52 --first partial product input vector
53 SIGNAL pp_gen_0_inputB : std_logic_vector(2 downto 0);
54
55 --stage 0 sums and carries
56 SIGNAL s0_00,s0_02,s0_03,s0_04,s0_05,s0_06,s0_07,s0_08,s0_09,s0_10,s0_11 : std_logic;
57 SIGNAL c0_00,c0_02,c0_03,c0_04,c0_05,c0_06,c0_07,c0_08,c0_09,c0_10,c0_11 : std_logic;
58
59 --stage 1 sums and carries
60 SIGNAL s1_01,s1_03,s1_04,s1_05,s1_06,s1_07,s1_08,s1_09,s1_10,s1_11,s1_12,s1_13 : std_logic;
61 SIGNAL c1_01,c1_03,c1_04,c1_05,c1_06,c1_07,c1_08,c1_09,c1_10,c1_11,c1_12,c1_13 : std_logic;
62
63 --stage 2 sums and carries
64 SIGNAL s2_02,s2_04,s2_05,s2_06,s2_07,s2_08,s2_09,s2_10,s2_11,s2_12,s2_13,s2_14 : std_logic;
65 SIGNAL c2_02,c2_04,c2_05,c2_06,c2_07,c2_08,c2_09,c2_10,c2_11,c2_12,c2_13,c2_14 : std_logic;
66
67 --stage 3 sums and carries
68 SIGNAL s3_03,s3_04,s3_05,s3_06,s3_07,s3_08,s3_09,s3_10,s3_11,s3_12,s3_13,s3_14,s3_15 : std_logic;
69 SIGNAL c3_03,c3_04,c3_05,c3_06,c3_07,c3_08,c3_09,c3_10,c3_11,c3_12,c3_13,c3_14,c3_15 : std_logic;
```

```
70
71  BEGIN
72  --partial product generation
73  pp_gen_0_inputB(0) <= '0';
74  pp_gen_0_inputB(1) <= inputB(0);
75  pp_gen_0_inputB(2) <= inputB(1);
76  pp_gen_0 : ppR4_gen port map(inputA,pp_gen_0_inputB,pp0,sf0);
77  pp_gen_1 : ppR4_gen port map(inputA,inputB(3 downto 1),pp1,sf1);
78  pp_gen_2 : ppR4_gen port map(inputA,inputB(5 downto 3),pp2,sf2);
79  pp_gen_3 : ppR4_gen port map(inputA,inputB(7 downto 5),pp3,sf3);
80
81  pp0_inv <= NOT pp0(8);
82  pp1_inv <= NOT pp1(8);
83  pp2_inv <= NOT pp2(8);
84  pp3_inv <= NOT pp3(8);
85
86  ----------------------------
87  ---stages of wallace tree---
88  ----------------------------
89
90  ---stage 0---
91  ha0_00 : half_adder port map(pp0(0),sf0,s0_00,c0_00);
92  --0_01 :
93  fa0_02 : full_adder port map(pp0(2),pp1(0),sf1,s0_02,c0_02);
94  ha0_03 : half_adder port map(pp0(3),pp1(1),s0_03,c0_03);
95  fa0_04 : full_adder port map(pp0(4),pp1(2),pp2(0),s0_04,c0_04);
96  fa0_05 : full_adder port map(pp0(5),pp1(3),pp2(1),s0_05,c0_05);
97  fa0_06 : full_adder port map(pp0(6),pp1(4),pp2(2),s0_06,c0_06);
98  fa0_07 : full_adder port map(pp0(7),pp1(5),pp2(3),s0_07,c0_07);
99  fa0_08 : full_adder port map(pp0(8),pp1(6),pp2(4),s0_08,c0_08);
100 fa0_09 : full_adder port map(pp0(8),pp1(7),pp2(5),s0_09,c0_09);
101 fa0_10 : full_adder port map(pp0_inv,pp1_inv,pp2(6),s0_10,c0_10);
102 ha0_11 : half_adder port map('1',pp2(7),s0_11,c0_11);
103 --0_12 :
104 --0_13 :
105 --0_14 :
106 --0_15 :
107
108 ---stage 1---
109 --1_00 :
110 ha1_01 : half_adder port map(pp0(1),c0_00,s1_01,c1_01);
111 --1_02 :
112 ha1_03 : half_adder port map(s0_03,c0_02,s1_03,c1_03);
113 fa1_04 : full_adder port map(s0_04,c0_03,sf2,s1_04,c1_04);
114 ha1_05 : half_adder port map(s0_05,c0_04,s1_05,c1_05);
115 fa1_06 : full_adder port map(s0_06,c0_05,pp3(0),s1_06,c1_06);
116 fa1_07 : full_adder port map(s0_07,c0_06,pp3(1),s1_07,c1_07);
117 fa1_08 : full_adder port map(s0_08,c0_07,pp3(2),s1_08,c1_08);
118 fa1_09 : full_adder port map(s0_09,c0_08,pp3(3),s1_09,c1_09);
119 fa1_10 : full_adder port map(s0_10,c0_09,pp3(4),s1_10,c1_10);
120 fa1_11 : full_adder port map(s0_11,c0_10,pp3(5),s1_11,c1_11);
121 fa1_12 : full_adder port map(pp2_inv,c0_11,pp3(6),s1_12,c1_12);
122 ha1_13 : half_adder port map('1',pp3(7),s1_13,c1_13);
123 --1_14 :
124 --1_15 :
125
126 ---stage 2---
127 --2_00 :
128 --2_01 :
129 ha2_02 : half_adder port map(s0_02,c1_01,s2_02,c2_02);
130 --2_03 :
131 ha2_04 : half_adder port map(s1_04,c1_03,s2_04,c2_04);
132 ha2_05 : half_adder port map(s1_05,c1_04,s2_05,c2_05);
133 fa2_06 : full_adder port map(s1_06,c1_05,sf3,s2_06,c2_06);
134 ha2_07 : half_adder port map(s1_07,c1_06,s2_07,c2_07);
135 ha2_08 : half_adder port map(s1_08,c1_07,s2_08,c2_08);
136 ha2_09 : half_adder port map(s1_09,c1_08,s2_09,c2_09);
137 ha2_10 : half_adder port map(s1_10,c1_09,s2_10,c2_10);
138 ha2_11 : half_adder port map(s1_11,c1_10,s2_11,c2_11);
139 ha2_12 : half_adder port map(s1_12,c1_11,s2_12,c2_12);
140 ha2_13 : half_adder port map(s1_13,c1_12,s2_13,c2_13);
```

```
141  ha2_14 : half_adder port map(pp3_inv,c1_13,s2_14,c2_14);
142  --2_15 :
143
144  ---stage 3---
145  --3_00 :
146  --3_01 :
147  --3_02 :
148  ha3_03 : half_adder port map(s1_03,c2_02,s3_03,c3_03);
149  ha3_04 : half_adder port map(s2_04,c3_03,s3_04,c3_04);
150  fa3_05 : full_adder port map(s2_05,c2_04,c3_04,s3_05,c3_05);
151  fa3_06 : full_adder port map(s2_06,c2_05,c3_05,s3_06,c3_06);
152  fa3_07 : full_adder port map(s2_07,c2_06,c3_06,s3_07,c3_07);
153  fa3_08 : full_adder port map(s2_08,c2_07,c3_07,s3_08,c3_08);
154  fa3_09 : full_adder port map(s2_09,c2_08,c3_08,s3_09,c3_09);
155  fa3_10 : full_adder port map(s2_10,c2_09,c3_09,s3_10,c3_10);
156  fa3_11 : full_adder port map(s2_11,c2_10,c3_10,s3_11,c3_11);
157  fa3_12 : full_adder port map(s2_12,c2_11,c3_11,s3_12,c3_12);
158  fa3_13 : full_adder port map(s2_13,c2_12,c3_12,s3_13,c3_13);
159  fa3_14 : full_adder port map(s2_14,c2_13,c3_13,s3_14,c3_14);
160  fa3_15 : full_adder port map('1',c2_14,c3_14,s3_15,c3_15);
161
162  ---result---
163  output(0) <= s0_00;
164  output(1) <= s1_01;
165  output(2) <= s2_02;
166  output(3) <= s3_03;
167  output(4) <= s3_04;
168  output(5) <= s3_05;
169  output(6) <= s3_06;
170  output(7) <= s3_07;
171  output(8) <= s3_08;
172  output(9) <= s3_09;
173  output(10) <= s3_10;
174  output(11) <= s3_11;
175  output(12) <= s3_12;
176  output(13) <= s3_13;
177  output(14) <= s3_14;
178  output(15) <= s3_15;
179
180  END;
```

Approximate hybrid R4/R16 multiplier with Wallace tree

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  ENTITY mul_hybrid IS
6          GENERIC(
7                  numbits : integer := 8
8          );
9          PORT(
10                 inputA : IN  std_logic_vector (numbits-1 downto 0);
11                 inputB : IN  std_logic_vector (numbits-1 downto 0);
12                 output : OUT  std_logic_vector ((2*numbits)-1 downto 0)
13         );
14 END mul_hybrid;
15
16 ARCHITECTURE arc_mul_hybrid OF mul_hybrid IS
17
18 COMPONENT full_adder IS
19         PORT(
20                 a     : IN  std_logic;
21                 b     : IN  std_logic;
22                 c     : IN  std_logic;
23                 sum   : OUT  std_logic;
24                 carry : OUT  std_logic
25         );
26 END COMPONENT;
27
28 COMPONENT half_adder IS
29         PORT(
30                 a     : IN  std_logic;
31                 b     : IN  std_logic;
32                 sum   : OUT  std_logic;
33                 carry : OUT  std_logic
34         );
35 END COMPONENT;
36
37 COMPONENT ppR4_gen IS
38         PORT(
39                 inputA     : IN std_logic_vector (numbits-1 downto 0);
40                 inputB     : IN std_logic_vector (2 downto 0);
41                 output     : OUT std_logic_vector (numbits downto 0);
42                 signfactor : OUT std_logic
43         );
44 END COMPONENT;
45
46 COMPONENT ppR16_gen IS
47         PORT(
48                 inputA     : IN std_logic_vector (numbits-1 downto 0);
49                 inputB     : IN std_logic_vector (3 downto 0);
50                 output     : OUT std_logic_vector (numbits+2 downto 0);
51                 signfactor : OUT std_logic
52         );
53 END COMPONENT;
54
55 --partial product signals
56 SIGNAL pp0 : std_logic_vector (numbits+2 downto 0); --approximate R16 partial product
57 SIGNAL pp1,pp2 : std_logic_vector (numbits downto 0); --accurate R4 partial products
58 SIGNAL sf0,sf1,sf2 : std_logic; --sign factors
59
60 --partial producht inverse terms
61 SIGNAL pp0_inv,pp1_inv,pp2_inv : std_logic;
62
63 --stage 0 sums and carries
64 SIGNAL s0_00,s0_04,s0_05,s0_06,s0_07,s0_08,s0_09,s0_10,s0_11,s0_12, s0_13 : std_logic;
65 SIGNAL c0_00,c0_04,c0_05,c0_06,c0_07,c0_08,c0_09,c0_10,c0_11,c0_12, c0_13 : std_logic;
66
67 --stage 1 sums and carries
68 SIGNAL s1_01,s1_05,s1_06,s1_07,s1_08,s1_09,s1_10,s1_11,s1_12,s1_13, s1_14 : std_logic;
69 SIGNAL c1_01,c1_05,c1_06,c1_07,c1_08,c1_09,c1_10,c1_11,c1_12,c1_13, c1_14 : std_logic;
```

```
70
71  --stage 2 sums and carries
72  SIGNAL s2_02,s2_03,s2_04,s2_05,s2_06,s2_07,s2_08,s2_09,s2_10,s2_11,s2_12,s2_13,s2_14,s2_15 : std_logic;
73  SIGNAL c2_02,c2_03,c2_04,c2_05,c2_06,c2_07,c2_08,c2_09,c2_10,c2_11,c2_12,c2_13,c2_14,c2_15 : std_logic;
74
75  BEGIN
76  --partial product generation
77  pp_gen_0 : ppR16_gen port map(inputA,inputB(3 downto 0),pp0,sf0);
78  pp_gen_1 : ppR4_gen port map(inputA,inputB(5 downto 3),pp1,sf1);
79  pp_gen_2 : ppR4_gen port map(inputA,inputB(7 downto 5),pp2,sf2);
80
81  pp0_inv <= NOT pp0(10);
82  pp1_inv <= NOT pp1(8);
83  pp2_inv <= NOT pp2(8);
84
85  ----------------------------
86  ---stages of wallace tree---
87  ----------------------------
88
89  ---stage 0---
90  ha0_00 : half_adder port map(pp0(0),sf0,s0_00,c0_00);
91  --0_01 :
92  --0_02 :
93  --0_03 :
94  fa0_04 : full_adder port map(pp0(4),pp1(0),sf1,s0_04,c0_04);
95  ha0_05 : half_adder port map(pp0(5),pp1(1),s0_05,c0_05);
96  fa0_06 : full_adder port map(pp0(6),pp1(2),pp2(0),s0_06,c0_06);
97  fa0_07 : full_adder port map(pp0(7),pp1(3),pp2(1),s0_07,c0_07);
98  fa0_08 : full_adder port map(pp0(8),pp1(4),pp2(2),s0_08,c0_08);
99  fa0_09 : full_adder port map(pp0(9),pp1(5),pp2(3),s0_09,c0_09);
100 fa0_10 : full_adder port map(pp0(10),pp1(6),pp2(4),s0_10,c0_10);
101 fa0_11 : full_adder port map(pp0(10),pp1(7),pp2(5),s0_11,c0_11);
102 fa0_12 : full_adder port map(pp0_inv,pp1_inv,pp2(6),s0_12,c0_12);
103 ha0_13 : half_adder port map('1', pp2(7),s0_13,c0_13);
104 --0_14 :
105 --0_15 :
106
107 ---stage 1---
108 --1_00 :
109 ha1_01 : half_adder port map(pp0(1),c0_00,s1_01,c1_01);
110 --1_02 :
111 --1_03 :
112 --1_04 :
113 ha1_05 : half_adder port map(s0_05,c0_04,s1_05,c1_05);
114 fa1_06 : full_adder port map(s0_06,c0_05,sf2,s1_06,c1_06);
115 ha1_07 : half_adder port map(s0_07,c0_06,s1_07,c1_07);
116 ha1_08 : half_adder port map(s0_08,c0_07,s1_08,c1_08);
117 ha1_09 : half_adder port map(s0_09,c0_08,s1_09,c1_09);
118 ha1_10 : half_adder port map(s0_10,c0_09,s1_10,c1_10);
119 ha1_11 : half_adder port map(s0_11,c0_10,s1_11,c1_11);
120 ha1_12 : half_adder port map(s0_12,c0_11,s1_12,c1_12);
121 ha1_13 : half_adder port map(s0_13,c0_12,s1_13,c1_13);
122 ha1_14 : half_adder port map(pp2_inv,c0_13,s1_14,c1_14);
123 --1_15 :
124
125 ---stage 2---
126 --2_00 :
127 --2_01 :
128 ha2_02 : half_adder port map(pp0(2),c1_01,s2_02,c2_02);
129 ha2_03 : half_adder port map(pp0(3),c2_02,s2_03,c2_03);
130 ha2_04 : half_adder port map(s0_04,c2_03,s2_04,c2_04);
131 ha2_05 : half_adder port map(s1_05,c2_04,s2_05,c2_05);
132 fa2_06 : full_adder port map(s1_06,c1_05,c2_05,s2_06,c2_06);
133 fa2_07 : full_adder port map(s1_07,c1_06,c2_06,s2_07,c2_07);
134 fa2_08 : full_adder port map(s1_08,c1_07,c2_07,s2_08,c2_08);
135 fa2_09 : full_adder port map(s1_09,c1_08,c2_08,s2_09,c2_09);
136 fa2_10 : full_adder port map(s1_10,c1_09,c2_09,s2_10,c2_10);
137 fa2_11 : full_adder port map(s1_11,c1_10,c2_10,s2_11,c2_11);
138 fa2_12 : full_adder port map(s1_12,c1_11,c2_11,s2_12,c2_12);
139 fa2_13 : full_adder port map(s1_13,c1_12,c2_12,s2_13,c2_13);
140 fa2_14 : full_adder port map(s1_14,c1_13,c2_13,s2_14,c2_14);
```

```
141  fa2_15 : full_adder port map('1',c1_14,c2_14,s2_15,c2_15);
142
143  ---result---
144  output(0) <= s0_00;
145  output(1) <= s1_01;
146  output(2) <= s2_02;
147  output(3) <= s2_03;
148  output(4) <= s2_04;
149  output(5) <= s2_05;
150  output(6) <= s2_06;
151  output(7) <= s2_07;
152  output(8) <= s2_08;
153  output(9) <= s2_09;
154  output(10) <= s2_10;
155  output(11) <= s2_11;
156  output(12) <= s2_12;
157  output(13) <= s2_13;
158  output(14) <= s2_14;
159  output(15) <= s2_15;
160
161  END;
```

Full adder

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY full_adder  IS
        PORT(
                a     : IN  std_logic;
                b     : IN  std_logic;
                c     : IN  std_logic;
                sum   : OUT  std_logic;
                carry : OUT  std_logic
        );
END full_adder;

ARCHITECTURE arch_full_adder OF full_adder  IS

BEGIN
--full adder operation
sum <= (a XOR b XOR c);
carry <= (a AND b) OR (c AND (a XOR b));

END arch_full_adder;
```

Half adder

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY half_adder  IS
        PORT(
                a     : IN  std_logic;
                b     : IN  std_logic;
                sum   : OUT  std_logic;
                carry : OUT  std_logic
        );
END half_adder;

ARCHITECTURE arch_half_adder OF half_adder  IS

BEGIN
--half adder operation
sum <= a XOR b;
carry <= a AND b;

END arch_half_adder;
```