Designing 8-bit Approximate Multipliers for FPGA using Internal Self-Healing

Rienk van der Wijk
Bachelor student Electrical Engineering
r.h.vanderwijk@student.utwente.nl

***Abstract*—This paper uses a design methodology for 8-bit approximate multipliers for FPGA using the Internal Self-Healing (ISH) methodology from a reduced design space of 4-bit approximate multipliers. Approximate computing is an effective method to reduce power consumption or chip area at the cost of output quality. However, optimizations made for ASIC may not translate entirely to FPGA, due to architectural differences. This paper emphasizes the importance of designing specifically for FPGA systems and presents a methodology that reduces the design space, while still allowing for the discovery of high-quality designs. The results show that optimized 8-bit multipliers for area outperform some of the state-of-the-art designs. Although power-optimized designs encountered some issues, the paper proposes possible multipliers to challenge the current state-of-the-art. Area-optimized designs are discussed which challenge the state-of-the-art. This paper demonstrates the effectiveness of the ISH methodology for designing efficient and high-quality 8-bit approximate multipliers for FPGA, with potential applications in error-resilient computing.**

I. Introduction

Approximate computing is the process of making approximations in basic computational modules, such as adders and multipliers, for an improved efficiency, often at the cost of quality. This means that some parts are intentionally simplified or removed in order to reduce for example the power consumption or the chip area that is used. This however leads to the output quality decreasing, such that it is not entirely accurate anymore. It has been shown to be a great option for error-resilient applications, such as image processing and neural networks [1]–[3]. This is because approximate computing can offer the required processing power with a high efficiency with regard to power and chip area.

Most research done for approximate computing is done with Application-Specific Integrated Circuits (ASIC) in mind, with FPGA lagging behind. Because of the architectural differences between ASIC and FPGA, any optimizations done on ASIC might not entirely translate to FPGA [2]–[4]. Since FPGA systems typically consume more power and require more area it is also crucial that more optimizations are done for these types of systems, since more gains could possibly be made here. It is shown that when designing specifically for FPGAs, designs can be found that outperform designs made for ASIC adapted to fit to FPGA [2], [3], [5].

The Internal Self-Healing methodology (ISH)[1] has been shown to work for ASIC for 4-bit, 8-bit and 16-bit approximate multiplier designs[1] and for FPGA for approximate multiplier 4-bit systems[5]. 8-bit approximate multiplier designs have also been made for FPGA[2], [3]. However, it is shown that the design space for 8-bit is quite large[1] which can take quite some time to explore[5]. Due to this large computational time it is often not possible to do a review of the entire design space. It is however still interesting to do reviews of at least a part of the design space, since designs could still be found with a good efficiency-quality trade-off. The ISH-methodology has also not been implemented yet on FPGA for 8-bit designs. This is why the main goal of this paper will be to design 8-bit approximate multipliers for FPGA from a reduced design space of 4-bit approximate multipliers while using the ISH-methodology. The 8-bit multipliers are optimized for power consumption or chip area, while minimzing error.

The paper layout is as follows: in section II preliminary knowledge and previous research will be discussed. This will also include which approximate multipliers have been considered to include in the design space. In section III the design methodology will be discussed including which multipliers are considered for the final design space. It also includes the tool flow and experimental setup. Section IV presents the results of the experiment and compares the results with the state-of-the-art. Section V provides a conclusion after which section VI gives some possibilities for future research.

II. Background

When designing approximate multipliers the conventional method often only considers systems that are *fail-small*, *fail-rare* [6] or *fail-moderate* [7]. These are systems with either small error magnitudes with higher error rates (*fail-small*), low error rates with high error magnitudes (*fail-rare*) or a combination with a moderate error magnitude and rate (*fail-moderate*). When working in this manner, however, not all possible designs are considered, which could result in designs with a good efficiency-quality trade-off being missed. This is why [1] discusses the self-healing methodology and improves on this to make the Internal Self-Healing (ISH) methodology. This design methodology allows systems with higher error rates and magnitudes to be considered, which leads to a larger design space.

To allow for Internal Self-Healing [1] uses recursive multipliers, which are multipliers where any $2n \times 2n$ multiplier is made of four $n \times\ n$ multipliers. Since these $n \times n$ multipliers can be four different ones, internally any errors made by one can be (partially) cancelled out by one of the other multipliers. In this way, the advantages of the $n \times n$ multipliers can also be

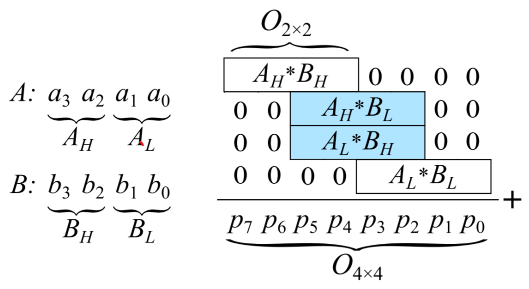


Fig. 1. 4×4 recursive multiplier which requires four 2×2 multipliers (taken from [1])

combined to create multipliers which suit the requirements of the $2n \times 2n$ multiplier. The smallest binary multipliers used in this paper are 2-bit multipliers, which multiply two 2-bit inputs to create a 4-bit output. These 4-bit multipliers are constructed as shown in Figure 1. The 4-bit inputs are split into the highest and lowest bits. These split inputs are given to the 2-bit multipliers shown resulting in 4-bit outputs, which afterwards have to be multiplied by a certain factor depending on which inputs were given. This is shown by the leading and trailing zeros. This can be seen as multiplying the output of the 2-bit multiplier by a factor, after which these outputs are added together to get the output of the 4-bit multiplier. The resulting equation can be seen in Formula 1:

$$Output_{4\times4} = 16(A_H \times B_H) + 4(A_H \times B_L) + 4(A_L \times B_H) + A_L \times B_L \; [1] \tag{1}$$

Sets of 2-bit multipliers were used in previous research to create 4×4 multipliers for FPGA [5]and 4×4, 8×8 and 16×16 multipliers for ASIC [1]. The designs of the 2-bit multipliers and the naming convention can be found in Appendix A.

When designing approximate multipliers there are also other ways to create approximate multipliers. When designing for ASIC a conventional way is to target transistor or gate-level truncations, but for FPGA this is not an option since the base blocks are Look-Up Tables (LUTs), as has been analysed by [2]. This means that any optimizations made for ASIC might not transfer properly to FPGA. This is why when designing approximate multipliers for FPGA one of the most effective methods is to consider what each Look-Up Table (LUT) does and what the underlying logic is that is required of the circuit. Since the LUTs are the base block of FPGA's the usage of these needs to be reduced in order to gain efficiency. Reducing the number of LUTs and the amount of LUT usage can be done in multiple ways of which a few will be discussed. The partial products and carrier signals can be approximated [3], in which the amount of approximations is dependent on how many LUTs can be removed. It is also possible to predetermine which partial products can be combined within a single LUT when looking at the required inputs, for which a few inputs are then ignored [8]. More on [8], including the design, can be seen in Appendix B. Both of these methods are quite intensive however, since all the LUTs need to gain specific mapping to get the correct outputs. This is why these methods are more difficult to use to design new small approximate multipliers.

III. Methodology

To design 8-bit multipliers the recursive multiplier method will be used, where four 4-bit multipliers will be combined. When combining multipliers with this method the 8-bit inputs are split into the 4 highest and 4 lowest bits, in the same manner as with four multipliers in a 4-bit multiplier discussed in section II. The outputs of the 4-bit multipliers in an 8-bit multiplier are combined as shown in Formula 2:

$$Output_{8\times8} = 256(A_H \times B_H) + 16(A_H \times B_L) + 16(A_L \times B_H) + A_L \times B_L \tag{2}$$

To reduce the design space specific 4-bit multipliers need to be selected. Since the 8-bit multipliers made will be optimized for FPGA, the 4-bit multipliers will only be selected from methods which also optimize for FPGA. In [5], a large number of multipliers optimized for FPGA are discussed and compared. However, before deciding which 4-bit multiplier will be in the design space, the metrics for the quality and efficiency by which these will be compared should be decided.

To determine the error the difference between the correct output and the inexact output will be used, so $(y_i - x_i)$. The self-healing methodology is used multipliers so both positive and negative errors should be present in order to cancel out. This is why multipliers which have some positive and some negative errors are beneficial. This is why no absolute value of the error will be taken at this point. The mean has to be taken over all inputs, after which the absolute for this value has to be taken, to be able to quantize and compare multipliers. Then the *Absolute Mean Error (AME)* metric is formed, which is shown in Formula 3:

$$AME = \left|\frac{\sum_{\forall i}(y_i - x_i)}{N}\right| \tag{3}$$

To be able to compare multipliers of different widths this value can be normalized by dividing over 2^{2n}, with n being the input size. This leads to the *Normalized Absolute Mean Error (NAME)* being formed, which is shown in Formula 4:

$$NAME = \frac{AME}{2^{2n}} \tag{4}$$

This error metric will be used to compare for quality of output of the multipliers to determine which are suitable to use in the design space.

To compare multipliers the efficiency-quality trade-off has to be taken into account. The quality part is already covered in the error metric. Since the designs will be optimized for either power or chip area a method needs to be determined for both of these. When considering the large design space estimations need to be done, since calculating accurate values needs a considerably larger processing time.

The power consumption of an 8-bit multiplier can be estimated by adding together the power of the four 4-bit multipliers it is made of. This estimation differs a bit from reality in that the switching activity for a specific input distribution needs to be used to determine the power consumption. This value will differ slightly for a 4-bit multiplier if it is part of a bigger multiplier compared to when it itself is the highest order, since the input distribution may differ slightly. This method still gives a reasonable estimation though, which can be used to select a possible set of multipliers that may be interesting. To estimate area, the area of the four 4-bit multipliers can be added together. With this estimation the circuit for adding together the outputs of the 4-bit multipliers is not taken into account. This estimation is still viable however, since the circuit required for adding together the outputs will have approximately the same size for each combination of 4-bit multipliers. Due to this, the estimation can be used to compare the combinations, although the designs should not be compared with complete 8-bit designs which already include this circuitry.

Research has been done on which 4-bit approximate multipliers are most suitable for FPGA[5]. Based on this a selection of 4-bit approximate multipliers is found which can be considered as suitable. The multipliers need to be verified to determine whether they are suitable, both in quality and efficiency, which can be done as discussed beforehand. Afterwards, the Pareto front of 4-bit multipliers can be used to estimate the Pareto front of 8-bit approximate multipliers, which can then be verified as well.

A. *Design Methodology*

The final design methodology will be to combine selected 4-bit multipliers to make 8-bit approximate multipliers. All combinations of multipliers will be compared on estimated efficiency (either area or power) against calculated output quality. The output quality is determined with the NAME metric, which makes sure that the ISH-methodology is incorporated.

B. *Toolflow and Experimental Setup*

To verify the 4-bit multipliers MATLAB was used to determine the error and Vivado was used to determine the power and chip area. In MATLAB behavioural models of the multipliers were used, while in Vivado the multipliers were described in VHDL. To determine the error in MATLAB normally distributed inputs were used. To determine the power in Vivado the same normally distributed inputs were used to determine the switching activity. This is done because Vivado assumes a uniform-like distribution where each input is on for 50% of the time and switches 12.5% of the time [9], which means that the calculated power can differ from reality. More information on using switching activity to determine can be found in Appendix C. To determine the area described in LUTs used, Vivado gives a standard report which is accurate. The selected verified 4-bit multiplier can be used to make the 8-bit multipliers. MATLAB is first used to determine the error and estimate the power and chip area. For this, the verified power and area of the 4-bit multipliers are used to make an estimation. The resulting estimated Pareto front is then verified in Vivado. The total tool flow can be seen in Figure 2.

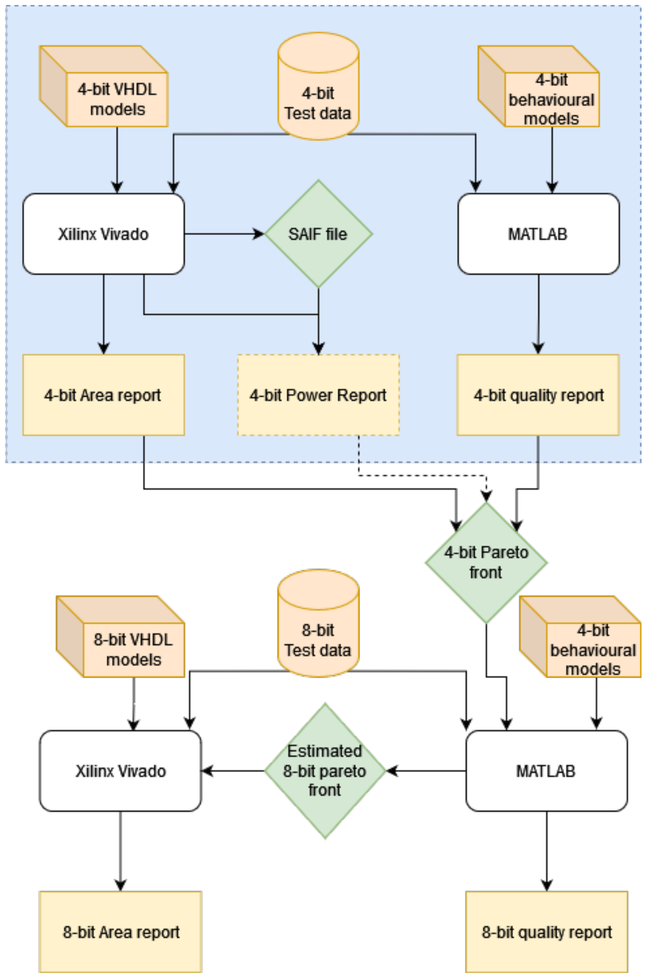


Fig. 2. Toolflow chart, the block in blue is preprocessing and verification done to select 4-bit multipliers. The dashed outlines of the power report indicate what has not been implemented in this thesis, for more info see section IV-A

When determining the error normally distributed inputs were used, with the following mean (μ), standard deviation (σ) and size of input set (N): for 4-bit ($\mu = 8,\ \sigma = 1.5,\ N = 1000$) and for 8-bit ($\mu = 128,\ \sigma = 22.5,\ N = 100\ 000$). For simulating the behavioural models MATLAB version 2022b was used. For verifying the designs Xilinx Vivado version 2020.1 was used. For this, a part from the Kintex-7 series "xc7k70tfbg484-2" was used.

The first step of verifying 4-bit multipliers leads to the sets used to make the 8-bit multipliers. The set which will be used to make 8-bit multipliers optimized for power can be seen in table I. It should be noted that the power stated in this table was not verified. Although the approximate multiplier R_{4335} has a higher power and error than multiplier R_{1315} and should thus not be included in the Pareto front, it will still be included in the design space since the design space is not large yet for the power-optimized multipliers. This means that the processing time will not increase too greatly from including it.

The set of 4-bit multipliers which will be used to make

TABLE I
SELECTED 4-BIT POWER OPTIMIZED MULTIPLIERS [5], THE 2-BIT MULTIPLIERS USED IN THESE 4-BIT MULTIPLIERS ARE SHOWN FROM MOST SIGNIFICANT MULTIPLIER (MSM) TO LEAST SIGNIFICANT MULTIPLIER (LSM).

Name	MSM	→		LSM	Power(μW)[5]	NAME (Normal)
R_{1311}[5]	M_1	M_3	M_1	M_1	129	$6.02E-4$
R_{1315}[5]	M_1	M_3	M_1	M_5	134	$1.25E-4$
R_{4335}[5]	M_4	M_3	M_3	M_5	137	$1.88E-4$
R_{1555}[5]	M_1	M_5	M_5	M_5	138	$0.00E+0$

TABLE II
SELECTED 4-BIT AREA-OPTIMIZED RECURSIVE MULTIPLIERS [5], THE 2-BIT MULTIPLIERS USED IN THESE 4-BIT MULTIPLIERS ARE SHOWN FROM MOST SIGNIFICANT MULTIPLIER (MSM) TO LEAST SIGNIFICANT MULTIPLIER (LSM)

Name	MSM	→		LSM	LUT/m	NAME (Normal)
R_{5421}[5]	M_5	M_4	M_2	M_1	13	$4.55E-3$
R_{3511}[5]	M_3	M_5	M_1	M_1	14	$5.76E-4$
R_{3311}[5]	M_3	M_3	M_1	M_1	16	$5.35E-4$
R_{5155}[5]	M_5	M_1	M_5	M_5	17	$4.05E-5$

TABLE III
SELECTED 4-BIT MULTIPLIERS FROM SMAPPROXLIB[8]

Name	LUT/m	NAME (Normal)	Shorthand
SMApproxLib Approx2[8]	7	$3.24E-2$	$SMA2_{4b}$
SMApproxLib Approx3[8]	8	$2.52E-2$	$SMA3_{4b}$
SMApproxLib Accurate[8]	12	$0.00E+0$	$SMA4_{4b}$

8-bit multipliers optimized for area can be seen in Table II and Table III. The 4-bit multipliers in Table V were selected from a verified set from [5]. These are recursive multipliers made from 2-bit multipliers, for which the design and naming convention can be found in Appendix A. The 4-bit multipliers in Table III were selected from a verified set from [8]. More on these designs can be found in Appendix B and the naming convention can be found in Table III. Although the approximate multipliers from Table V have a higher area and error than the accurate multiplier from Table III, these will still be included in the design space. They will not appear often in the Pareto front but it is interesting to test whether they have no value at all or if they still add something.

IV. EVALUATION

A. 8-bit Power Optimized Recursive Multipliers

The behavioural models of the approximate multipliers shown in table I were used to determine the estimated quality-efficiency trade-off plot shown in Figure 3. The best multipliers of the Pareto front from Figure 3 can be seen in table IV. The power could not be verified for the 4-bit and 8-bit approximate multipliers due to the limitations of Vivado which does not report power lower than 1 mW. Since the approximate multipliers differ from each other with only a few μW, this does not suffice to do a proper comparison.

B. 8-bit Area Optimized Recursive Multipliers

The behavioural models of the approximate multipliers shown in tables II and III were used to determine the estimated

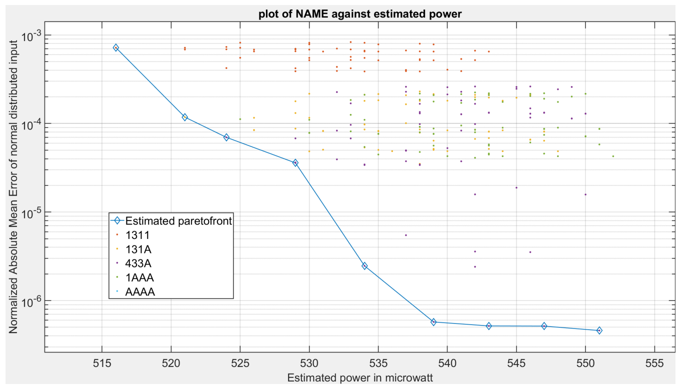


Fig. 3. Estimated quality-efficiency trade-off of 8-bit approximate multipliers optimized for power. The dot colour indicates the Most Significant Multiplier

TABLE IV
ESTIMATED PARETO FRONT MULTIPLIERS OF 8-BIT POWER OPTIMIZED MULTIPLIERS, *IT SHOULD BE NOTED THAT THE POWER IS AN ESTIMATION AND COULD NOT BE VERIFIED BY SYNTHESIS.

MSM	→		LSM	Power(μW)*	NAME (Normal)
R_{1311}	R_{1311}	R_{1311}	R_{1311}	516	$7.19E-4$
R_{1315}	R_{1311}	R_{1311}	R_{1311}	521	$1.18E-4$
R_{4335}	R_{1315}	R_{1311}	R_{1311}	529	$3.47E-5$
R_{4335}	R_{1315}	R_{1315}	R_{1311}	534	$2.47E-6$
R_{4335}	R_{1315}	R_{1315}	R_{1315}	539	$5.74E-7$

quality-efficiency trade-off plot shown in Figure 4. In this Figure, the estimated Pareto front can be seen on the blue line. Since this is only an estimation the estimated area has a standard bias of approximately 9 LUTs, due to the method of making the estimation. These multipliers can however still be properly compared using the estimated area since all multipliers show this approximately the same bias. The estimated Pareto front was verified by implementing in Xilinx Vivado which resulted in the plot seen in Figure 5. As can be observed the area differs from Figure 4. Most notable is the multipliers at 47 LUTs, since synthesis showed that, although they were estimated to have a different area, in reality the area is the same, thus the multipliers with a higher error can be discarded. The best multipliers from Figure 5 can be seen in Table V. As can be observed from this Table the multipliers from Table III are more prevalent in these multipliers. Only the multiplier with an area of 55 LUTs has a multiplier from Table II included in its structure. This is likely due to the higher area of the multipliers from Table II which apparently does not outweigh the lower error in comparison with the approximate multipliers from Table III. It could also be that if the recursive 4-bit multipliers were predefined within the HDL like SMApprox (see Appendix B), they might decrease in area and would appear more in the Pareto front of 8-bit multipliers.

C. Comparison

When comparing the proposed approximate multipliers with [1] some observations can be done without needing to implement the other multipliers on FPGA. This is because those

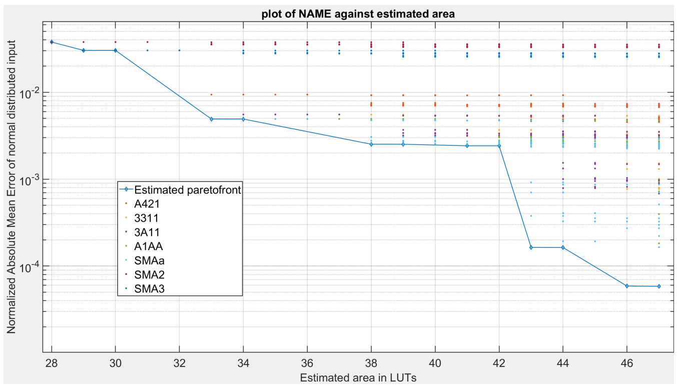


Fig. 4. Estimated quality-efficiency trade-off of 8-bit approximate multipliers optimized for area. The dot colour indicates the Most Significant Multiplier. The blue line indicates the estimated Pareto front.

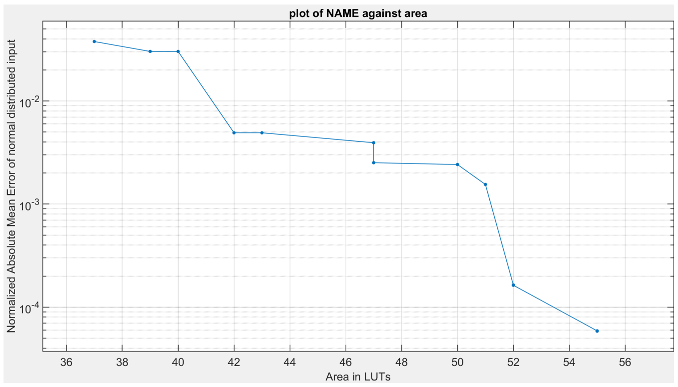


Fig. 5. Implemented quality-efficiency trade-off of 8-bit approximate multipliers optimized for area. The multipliers from the estimated Pareto front from figure 4 are shown.

TABLE V
PARETO FRONT OF 8-BIT AREA OPTIMIZED MULTIPLIERS

MSM	$\rightarrow$		LSM	LUT/m	NAME (Normal)
$SMA2_{4b}$	$SMA2_{4b}$	$SMA2_{4b}$	$SMA2_{4b}$	37	$3.78E-2$
$SMA4_{4b}$	$SMA2_{4b}$	$SMA2_{4b}$	$SMA2_{4b}$	42	$4.92E-3$
$SMA4_{4b}$	$SMA2_{4b}$	$SMA4_{4b}$	$SMA3_{4b}$	47	$2.52E-3$
$SMA4_{4b}$	$SMA4_{4b}$	$SMA4_{4b}$	$SMA3_{4b}$	52	$1.63E-4$
$SMA4_{4b}$	$SMA4_{4b}$	R_{3311}	$SMA3_{4b}$	55	$5.84E-5$

multipliers are made in the same recursive manner as the 4-bit recursive multipliers made for FGPA [5]. Since the 4-bit recursive multipliers used in this paper are optimized for FPGA and those multipliers do not match with the 4-bit multipliers from [1], it can be concluded that any 8-bit Pareto front made will outperform or at least be of the same level. Thus, since the proposed multipliers are designed with FPGA in mind, it is likely that they outperform on FPGA. This should however be tested when implementing the method to verify the power.

The 8-bit approximate multipliers proposed by [8] can be seen in table VI. Their designs can be found in Appendix B. These multipliers were verified in the same manner as the proposed 8-bit area-optimized multipliers. Since the proposed multipliers also use a set from this paper while including more

TABLE VI
8-BIT MULTIPLIERS FROM SMAPPROXLIB[8]

Name	LUT/m	NAME (Normal)
SMApproxLib Approx1[8]	55	$1.83E-3$
SMApproxLib Approx2[8]	47	$2.95E-3$
SMApproxLib Approx3[8]	46	$2.92E-3$

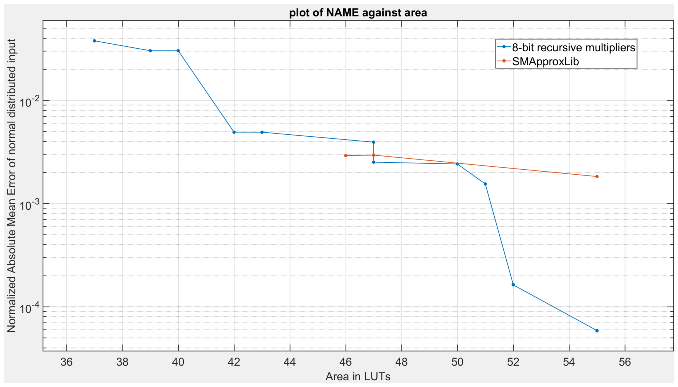


Fig. 6. Quality-efficiency trade-off of 8-bit approximate multipliers optimized for area against the multipliers from SMApproxLib (blue = proposed multipliers, red = SMApproxLib)

options, they will likely outperform. This can also be seen in Figure 6. It can be observed that the proposed multipliers indeed outperform for the largest part of the quality-efficiency trade-off. This could be attributed to the ISH-methodology being employed, allowing for more design options. The multiplier SMApproxLib Approx3[8] however still outperforms the multipliers which have approximately the same area, by having only a slightly higher error while having saving on a LUT in comparison with the 47 LUT multiplier from Table V. The other multipliers from [8] however are outmatched by the multipliers designed within this paper. The 8-bit multiplier from Table V with an area of 55 LUTs has a 30× lower error than the multiplier from VI with the same area. The 8-bit multiplier from Table V with an area of 42 LUTs has 4 LUTs less than the 46 LUT multiplier from VI while having an error of approximately the same magnitude.

From this, it can be concluded that the Pareto front for 8-bit will be as seen in table VII. Four multipliers designed in this paper are included and one 8-bit multiplier from [8] is also included. In the 8-bit multipliers designed within this paper, one 4-bit multiplier from Table II is included. It can be observed that the 8-bit multipliers designed in this paper outperform the state-of-the-art of [8] for some of their designs. For power a final Pareto front can not be concluded, since the power can only be estimated.

V. CONCLUSION

The aim was to design 8-bit approximate multipliers for FPGA using the Internal Self-Healing methodology from a reduced design space of 4-bit approximate multipliers. The paper discussed the importance of approximate computing and the value of designing specifically for FPGA, since ASIC

TABLE VII
PARETO FRONT OF 8-BIT AREA OPTIMIZED MULTIPLIERS

MSM	$\rightarrow$		LSM	LUT/m	NAME (Normal)
$SMA2_{4b}$	$SMA2_{4b}$	$SMA2_{4b}$	$SMA2_{4b}$	37	$3.78E-2$
$SMA4_{4b}$	$SMA2_{4b}$	$SMA2_{4b}$	$SMA2_{4b}$	42	$4.92E-3$
SMApproxLib Approx3 [8]				46	$2.92E-3$
$SMA4_{4b}$	$SMA4_{4b}$	$SMA4_{4b}$	$SMA3_{4b}$	52	$1.63E-4$
$SMA4_{4b}$	$SMA4_{4b}$	R_{3311}	$SMA3_{4b}$	55	$5.84E-5$

does not translate entirely well due to architectural differences. A design methodology was proposed where 4-bit multipliers are selected beforehand in order to reduce the design space while making it possible to find good options. 8-bit approximate multipliers optimized for area were proposed which in some cases improved the quality of output by approximately 30$\times$ for a similar area in comparison with SMApproxLib Approx1[8] or showed approximately 10% decrease in LUT usage while having a similar error magnitude in comparison with SMApproxLib Approx3[8]. Overall the 8-bit approximate multipliers showed to be a good addition to the state-of-the-art. Although problems were encountered with multipliers optimized for power, estimations were done to propose possible multipliers to challenge the current state-of-the-art. Overall, an effective methodology has been proposed employing the ISH-methodology to design recursive approximate multipliers for FPGA.

VI. Future Work

Since the power-optimized multipliers could not be verified, it would be interesting to find a method of verification. Based on these results further research could be conducted to determine the optimal 4-bit and 8-bit approximate multipliers to incorporate into larger designs.

For the area-optimized multipliers, a set of twelve 4-bit multipliers was used to make the 8-bit multipliers. It would be interesting to expand this set by incorporating more 4-bit multipliers to explore a larger design space. To ensure that the processing time does not become too great, any 4-bit multipliers that did not make it into the 8-bit multipliers from the Pareto front could be excluded from the set.

The designed 8-bit multipliers could be applied to real applications or data to determine if they still offer the same advantages. Radio astronomy processing could be a potential application to be tested on. Based on the results, the design methodology could be adapted to incorporate different test data or estimations during the design space reduction step.

References

[1] G. A. Gillani, M. A. Hanif, B. Verstoep, S. H. Gerez, M. Shafique, and A. B. J. Kokkeler, "Macish: Designing approximate mac accelerators with internal-self-healing," *IEEE Access*, vol. 7, pp. 77 142–77 160, 2019.

[2] B. S. Prabakaran, S. Rehman, M. A. Hanif, *et al.*, "Demas: An efficient design methodology for building approximate adders for fpga-based systems," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 917–920.

[3] Y. Guo, H. Sun, and S. Kimura, "Small-area and low-power fpga-based multipliers using approximate elementary modules," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020, pp. 599–604.

[4] S. Ullah, S. Rehman, M. Shafique, and A. Kumar, "High-performance accurate and approximate multipliers for fpga-based hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 211–224, 2022. DOI: 10.1109/TCAD.2021.3056337.

[5] R. van Loo, *Investigating approximate fpga multiplication for increased power-efficiency*, Nov. 2022. [Online]. Available: http://essay.utwente.nl/93756/.

[6] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–9. DOI: 10.1145/2463209.2488873.

[7] E. Nogues, D. Menard, and M. Pelcat, "Algorithmic-level approximate computing applied to energy efficient hevc decoding," *IEEE Transactions on Emerging Topics in Computing*, vol. 7, no. 1, pp. 5–17, 2019. DOI: 10.1109/TETC.2016.2593644.

[8] S. Ullah, S. S. Murthy, and A. Kumar, "Smapproxlib: Library of fpga-based approximate multipliers," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465845.

[9] Xilinx, *Vivado design suite user guide: Power analysis and optimization*, Jun. 2020. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2020_1/ug997-vivado-power-analysis-optimization-tutorial.pdf.

[10] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *2011 24th Internatioal Conference on VLSI Design*, 2011, pp. 346–351. DOI: 10.1109/VLSID.2011.51.

[11] 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)*, 2016, pp. 1–8. DOI: 10.1145/2966986.2967005.

[12] G. A. Gillani, M. A. Hanif, M. Krone, S. H. Gerez, M. Shafique, and A. B. J. Kokkeler, "Squash: Approximate square-accumulate with self-healing," *IEEE Access*, vol. 6, pp. 49 112–49 128, 2018. DOI: 10.1109/ACCESS.2018.2868036.

Appendix

A. 2-bit approximate multipliers

For the design of the 8-bit multipliers two different types of multipliers were used. The 4-bit recursive multipliers were made from 2-bit (approximate) multipliers. The naming convention for a 4-bit recursive multiplier is R_{abcd}, with $a, b, candd$ being the 2-bit multipliers it is made of. An example can be seen in R_{4335}, which is the combination of the 2-bit multipliers M4, M3 twice and M5, from most significant (in this case M4) to least significant multiplier (in this case M5). The behaviour of each 2-bit multiplier can be seen in table VIII-XII and the logic of each 2-bit multiplier can be seen in figure 7-11. In the tables with behaviour the miscalculations are shown in grey.

TABLE VIII
BEHAVIOUR OF APPROXIMATE MULTIPLIER M1 [10] (TAKEN FROM[5])

B \ A	**00 (0)**	**01 (1)**	**10 (2)**	**11 (3)**
00 (0)	0000 (0)	0000 (0)	0000 (0)	0000 (0)
01 (1)	0000 (0)	0001 (1)	0010 (2)	0011 (3)
10 (2)	0000 (0)	0010 (2)	0100 (4)	0110 (6)
11 (3)	0000 (0)	0011 (3)	0110 (6)	0111 (7)

TABLE IX
BEHAVIOUR OF APPROXIMATE MULTIPLIER M2 [11] (TAKEN FROM [5])

B \ A	**00 (0)**	**01 (1)**	**10 (2)**	**11 (3)**
00 (0)	0000 (0)	0000 (0)	0000 (0)	0000 (0)
01 (1)	0000 (0)	0000 (0)	0010 (2)	0010 (2)
10 (2)	0000 (0)	0010 (2)	0100 (4)	0110 (6)
11 (3)	0000 (0)	0010 (2)	0110 (6)	1001 (9)

TABLE X
BEHAVIOUR OF APPROXIMATE MULTIPLIER M3 [12] (TAKEN FROM [5])

B \ A	**00 (0)**	**01 (1)**	**10 (2)**	**11 (3)**
00 (0)	0000 (0)	0000 (0)	0000 (0)	0000 (0)
01 (1)	0000 (0)	0001 (1)	0010 (2)	0011 (3)
10 (2)	0000 (0)	0010 (2)	0100 (4)	0110 (6)
11 (3)	0000 (0)	0011 (3)	0110 (6)	1011 (11)

TABLE XI
BEHAVIOUR OF APPROXIMATE MULTIPLIER M4 [1] (TAKEN FROM [5])

B \ A	**00 (0)**	**01 (1)**	**10 (2)**	**11 (3)**
00 (0)	0000 (0)	0000 (0)	0000 (0)	0000 (0)
01 (1)	0000 (0)	0001 (1)	0010 (2)	0011 (3)
10 (2)	0000 (0)	0010 (2)	0100 (4)	0110 (6)
11 (3)	0000 (0)	0011 (3)	0110 (6)	0101 (5)

TABLE XII
BEHAVIOUR OF ACCURATE MULTIPLIER M5 (TAKEN FROM [5])

B \ A	**00 (0)**	**01 (1)**	**10 (2)**	**11 (3)**
00 (0)	0000 (0)	0000 (0)	0000 (0)	0000 (0)
01 (1)	0000 (0)	0001 (1)	0010 (2)	0011 (3)
10 (2)	0000 (0)	0010 (2)	0100 (4)	0110 (6)
11 (3)	0000 (0)	0011 (3)	0110 (6)	1001 (9)

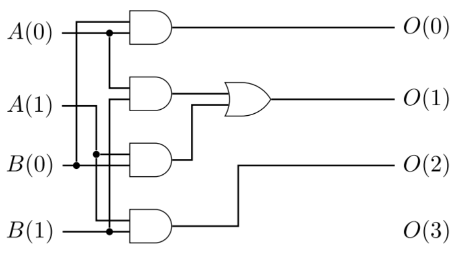


Fig. 7. Logic of approximate multiplier M1 [10] (taken from [1])

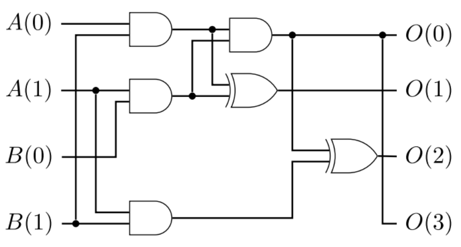


Fig. 8. Logic of approximate multiplier M2 [11] (taken from [1])

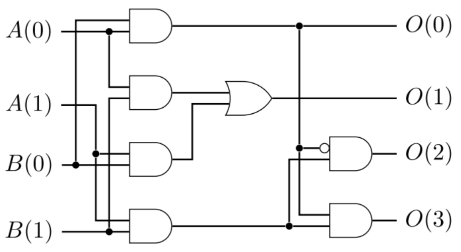


Fig. 9. Logic of approximate multiplier M3 [12] (taken from [1])

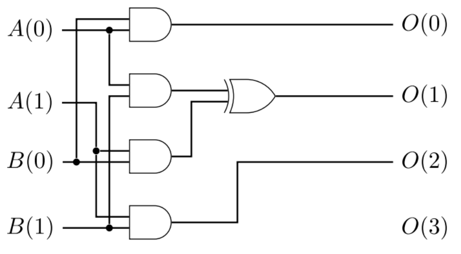


Fig. 10. Logic of approximate multiplier M4 [1] (taken from [1])

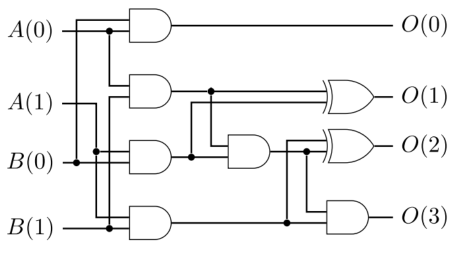


Fig. 11. Logic of accurate multiplier M5 (taken from [1])

B. SMApproxLib Multipliers [8]

For the design of the 8-bit multipliers two different types of multipliers were used. The multipliers from [8] are based on not allowing the synthesizing tool any freedom, but instead designing the entire multiplier using HDL primitives. This means that the LUTs are predefined within the HDL, for which three different configurations are predefined. These are then combined to create a general $n \times n$ multiplier which can be seen in Figure 12.

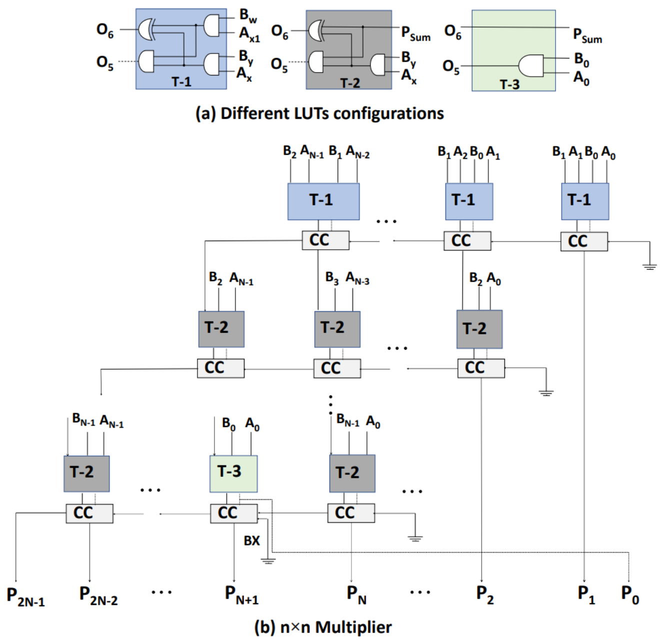
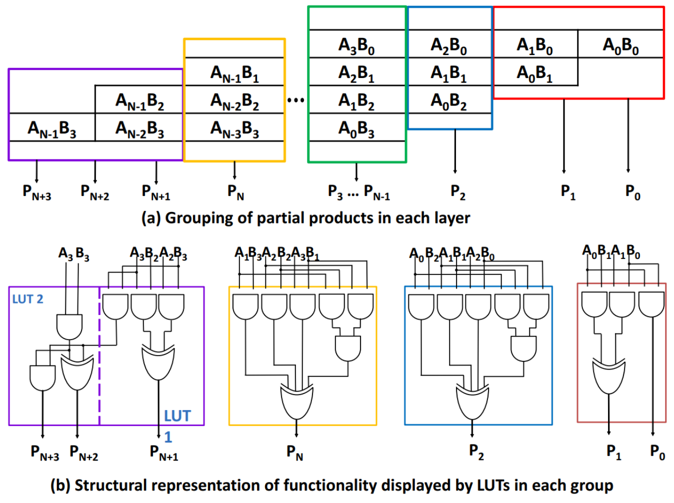


Fig. 12. Implementation of SMApproxLib Multipliers [8] (taken from [8])

The basic accurate multiplier described above is adapted in a few ways to make approximate multipliers. ”For Approx2 and Approx3 they propose to not use any chain adders at all, but group partial products together into single layers that are either implemented using one or two LUT6 2 primitives”[5]. This can be seen in Figure 13 and Figure 14. Approx3 differs from Approx2 in that it uses a view extra inputs for the middle groups to increase the accuracy.

Since the designs can be used for any $n \times n$ size multiplier, the structure of a 4-bit and an 8-bit design should be discussed. The structure for the 4-bit design is similar to what is shown in Figure 13a, with N being 4. This means that only one green layer is required for a 4-bit Approx2 or Approx3 multiplier. For 8-bit the array multiplier can be seen in Figure 15. This shows that to implement an 8-bit Approx2 or Approx 3 multiplier, two structures as shown in Figure 13a are required, with five green layers per structure.



(a) Grouping of partial products in each layer
(b) Structural representation of functionality displayed by LUTs in each group
LUT 2
LUT 1

Fig. 13. Implementation of Approx2 and Approx3 [8] (taken from [8])

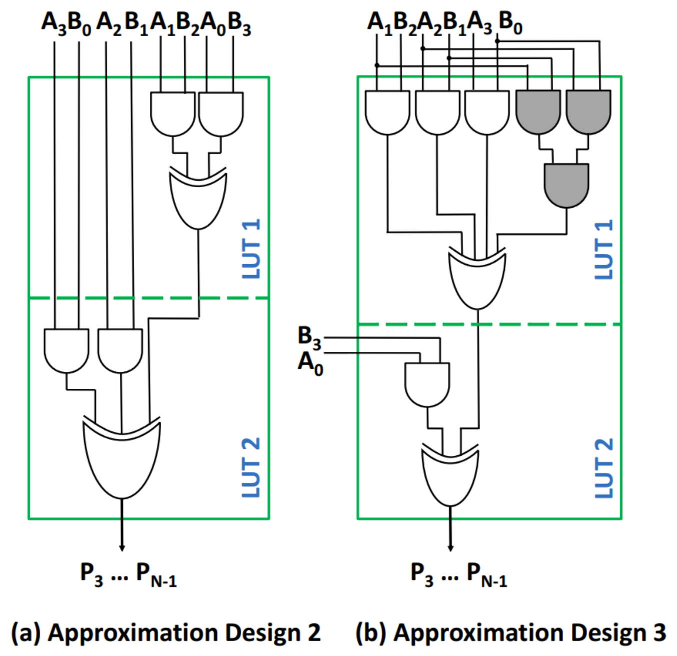


Fig. 14. Implementation of the green part in Approx2 and Approx3 [8] (taken from [8])

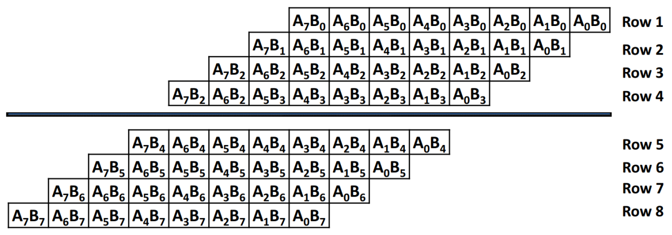


Fig. 15. 8×8 array multiplier (taken from [8])

C. *Determining power for specified switching activity*

Xilinx Vivado should be used, with a design which has been confirmed to be functioning as desired. In this tutorial Xilinx Vivado version 2020.1 is used. Xilinx Vivado can be used by first going to X2Go (https://caesdoc.ewi.utwente.nl/x2go). Here in the terminal Vivado can be loaded in by running the following commands:

1) ”module avail” can be run to check which version of Vivado are available.
2) ”module load module” needs to be run to load in a specific application. In this case Xilinx Vivado v2020.1 is used, so the command is ”module load xilinx/vivado/2020.1”.
3) ”vivado” needs to be run as command to start up Xilinx Vivado. This will start up an instance of Xilinx Vivado.

After Vivado has been started up and your design project is loaded in the following steps should be followed:

1) Synthesize and implement your design. Resolve any encountered errors.
2) In Vivado under ”Flow > Settings > Simulation Settings > Simulation” go to the simulation tab, as shown in figures 16 and 17. Change ”xsim.simulate.saif_scope*” to the module for which you want to know the power as shown in figure 17. This will most often be the unit under test (UUT).
3) Below this set “xsim.simulate.saif*” to the name under which you want the output SAIF file to have, as shown in figure 18.
4) Simulate your design to output a SAIF file.
5) Read out the SAIF file by running ”read_saif {file_name}” in the TCL console, with {file_name} being the location and name of your previously named SAIF file. An example is shown in figure 19.
6) Report the power by running “report_power” in the TCL console, as shown in figure 20.
7) You will now get a report of your power in the TCL Console, which can be used to see your power usage for a specific input.

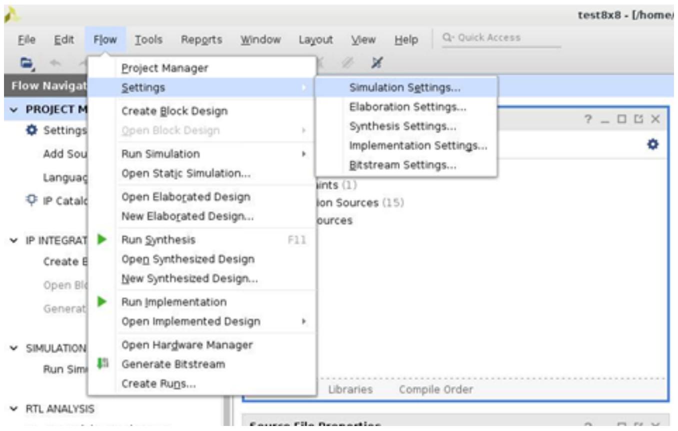
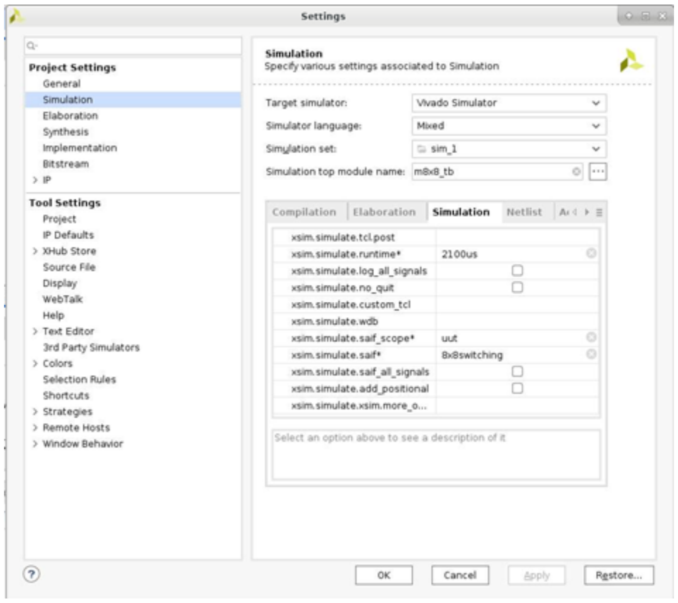


Fig. 16. Location of simulation settings option



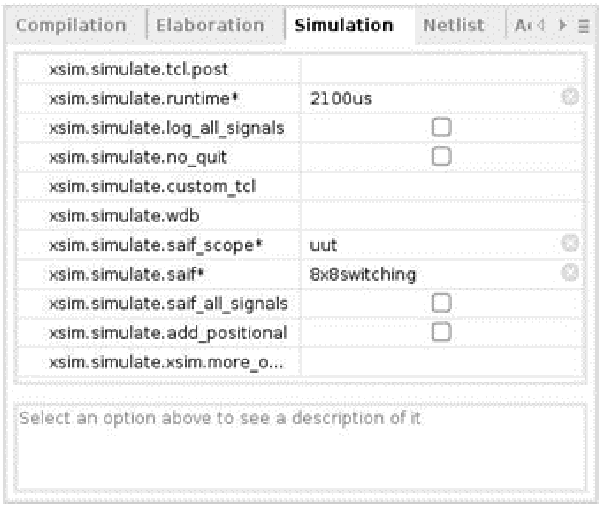
Simulation
Specify various settings associated to Simulation

Target simulator: Vivado Simulator
Simulator language: Mixed
Simulation set: sim_1
Simulation top module name: m8x8_tb

xsim.simulate.tcl.post	
xsim.simulate.runtime*	2100us
xsim.simulate.log_all_signals	☐
xsim.simulate.no_quit	☐
xsim.simulate.custom_tcl	
xsim.simulate.wdb	
xsim.simulate.saif_scope*	uut
xsim.simulate.saif*	8x8switching
xsim.simulate.saif_all_signals	☐
xsim.simulate.add_positional	☐
xsim.simulate.xsim.more_o...	

Select an option above to see a description of it

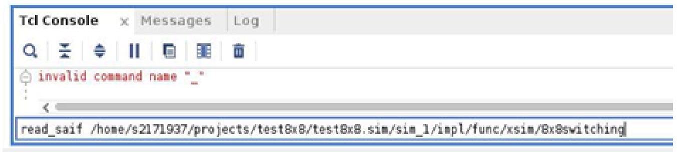
Fig. 17. Location of the Simulation tab within the Simulation Settings



xsim.simulate.tcl.post	
xsim.simulate.runtime*	2100us
xsim.simulate.log_all_signals	☐
xsim.simulate.no_quit	☐
xsim.simulate.custom_tcl	
xsim.simulate.wdb	
xsim.simulate.saif_scope*	uut
xsim.simulate.saif*	8x8switching
xsim.simulate.saif_all_signals	☐
xsim.simulate.add_positional	☐
xsim.simulate.xsim.more_o...	

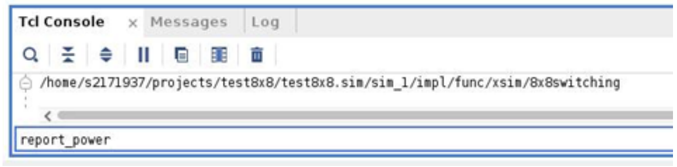
Select an option above to see a description of it

Fig. 18. Example names of ”xsim.simulate.saif_scope*” and ”xsim.simulate.saif*” shown within Simulation tab



```
invalid command name "_"
read_saif /home/s2171937/projects/test8x8/test8x8.sim/sim_1/impl/func/xsim/8x8switching
```

Fig. 19. Example command line to read out a SAIF file



```
/home/s2171937/projects/test8x8/test8x8.sim/sim_1/impl/func/xsim/8x8switching
report_power
```

Fig. 20. Example command line to report power