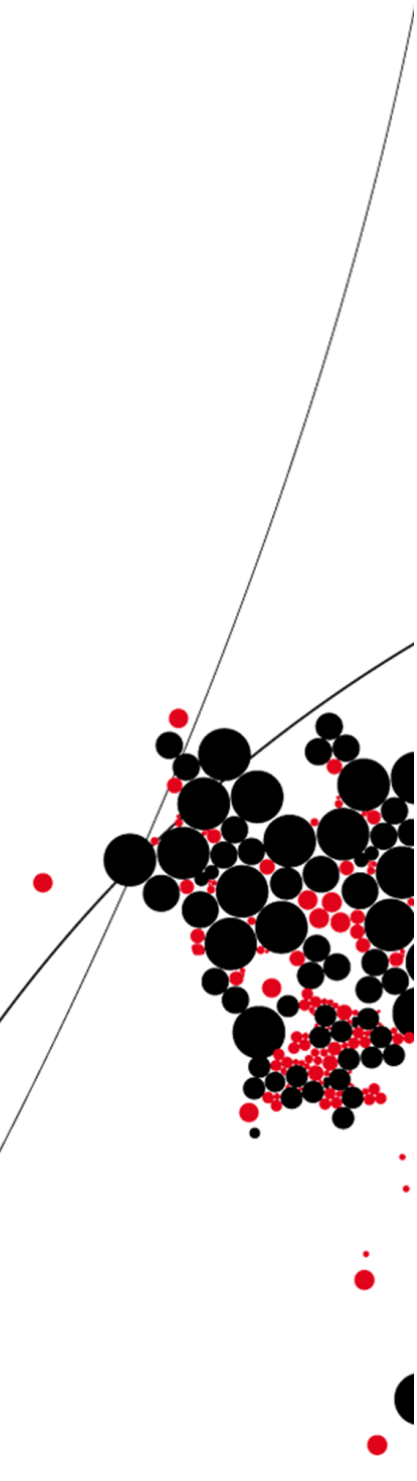
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Investigating Approximate FPGA Multiplication for Increased Power-Efficiency

Rick van Loo

M.Sc. Thesis
December 2022



Supervisors:
dr. ir. N. Alachiotis
dr. ir. S.G.A. Gillani
dr. ir. A.B.J. Kokkeler

Computer Architecture for Embedded Systems (CAES)
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Contents

Chapter 1

Introduction

Opposed to general-purpose computer processing units (CPU), hardware accelerators are specifically designed digital hardware to perform the same functionality at a higher performance. Fields such as Radio Astronomy require highly power-efficient digital computer systems to process big streams of data. Neural Networks algorithms are being progressively more used, but demand high computational power. Due to increasing complexity of these signal processing applications and demand for performance, such hardware accelerators are commonplace.

For these error-resilient applications, "Approximate Computing" studies the field of further improving performance at a loss of quality for digital systems. Commonly, a reduction in chip-area or power consumption is investigated by reducing accuracy of the signal processing system. For these fields, such a loss in accuracy has to be acceptable for the approximate accelerator to be of any use. Many of these accelerators are based upon common digital arithmetic structures, such as multipliers and adders. Therefore, these approximate modules are commonly studied within literature. For example, Gillani et al.[18] show in their work that building recursive multipliers using a set of smaller components, can create approximate multiply-accumulators with a near-to-zero mean error. In their case study, they apply these multiply-accumulators to an accelerator for a Radio Astronomy Calibration Processing algorithm and get favourable results in terms of power-efficiency. Other papers, such as one by Rehman et al.[29] explore a vast design space of multipliers and adders for ASIC 45nm technology using their automated tool-flow.

However, these aforementioned studies limit themselves to fine-grained architectures such as ASICs technology. Papers studying these multipliers for FPGA technology come to the conclusion that due to the architectural differ-

ences, comparable performance cannot be archived and they introduce new approximate multiplier designs for FPGA architectures. [19]. While these newly explored Approximate Multiplier configurations show mayor improvements in area-efficiency, they show minimum to no improvement in terms of power-efficiency. [19][32]. In conclusion, research aiming to improve power-efficiency for approximate multipliers on the FPGA is sparse.

The objective of this thesis is to explore Approximate Multiplier models on the FPGA and compare them on basis of power consumption and error quality. Instead of putting the focus on reducing area, the internal architecture of FPGA technology and their properties are investigated. The properties of this architecture are related to the resulting resource-area, power-consumption and latency from implemented multiplier models found in literature, and a resulting approximate multiplier implementation for low power-consumption is shown.

The main contributions this thesis are:

- A detailed design space exploration of 4-bit recursive multipliers on the FPGA, using designs from Gillani et al.[18]
- An automated tool-flow, called 'Approxy', is developed. This tool-flow automates verification, synthesis, implementation and analysis for Approximate Multiplier models on Xilinx FPGAs.

1.1 Outline

Chapter 2: Background will focus on the background of Approximate Computing, FPGA technology and its resources. Furthermore, it will discuss system requirements and error-resilient applications for approximate computing. In Chapter 3: Methodology, a design workflow and experimental setup is introduced that is used for comparing various Approximate Multiplier models. After which in Chapter 4: Implementation of Multiplier Models, various multipliers models are discussed and implemented. The best performing models are compared to other models found within literature in Chapter 5: Evaluation. Finally, the thesis concludes in Chapter 6: Conclusion and future work is suggested in Chapter 7: Future Work.

Chapter 2

Background

This Chapter will go through some general Approximate Computing techniques in Section 2.1, after which an introduction to basic Digital Multiplication is given in 2.2. From here on in Section 2.3, FPGA technology and its Design Flow is explained. Section 2.4 "System Requirements" explains the relevant (FPGA) resources to quantify the performance of Approximate Multiplication of the FPGA. In Section 2.5, a handful of fields where these Approximate Multipliers could be applied to are investigated. Finally, in Section 2.6 an introduction is given to the Go programming language that is later used in the thesis to design CAD tooling.

2.1 Approximate Computing

Approximate Computing is an emerging field of research that aims to improve energy-efficiency and/or performance of a digital system by introducing an acceptable loss of quality to the overall result. Various methods and techniques can be used to achieve this loss of quality, by for instance returning an inaccurate result instead of the expected accurate one.

Generally, Approximate Computing techniques can be classified as applicable on three layers, the Software/Program level, the Architectural level and the Hardware/Circuit level. [1] In terms of software Approximate Computing, these can range from techniques that reduce computational time to techniques that reduce memory footprint. For instance, the technique 'loop-perforation' reduces computational time by only executing a subset of iterations within an iterative algorithm. This comes at a cost of general accuracy. [2] A more classic computing technique that can be used for approximate computing is called 'Memoization', coined by Donald Michie

in 1968. [3] Generally, this method is used by saving outputs of functions for given inputs, to be reusable at a later time. This improves the computational time of the function, at the cost of higher memory utilization. However, 'Memoization' can also be used to apply approximate computing. If a function produces similar outputs for similar inputs, an earlier calculated output could potentially be used for a range of inputs. [1]

Within Architectural Approximation, more recent research shows that neural networks can be used as a basis to replace approximable code segments within systems to increase execution time performance by reducing accuracy. Furthermore, Neural Networks itself "can be architected and optimized in various ways to maximize energy efficiency[4]". Another technique that can be either applied within both software and architecture is the reduction of data precision. By reducing the precision of data within an application, accuracy is reduced. Despite this, memory footprint and energy consumption is reduced. Within architectural design on hardware level, this generally also means lower footprint by for instance requiring smaller adder chains. [1]

Lastly, Approximation can be achieved on a hardware level by introducing faulty or inaccurate hardware. For instance by removing the carry chain of an adder: the area, delay and power consumption can be reduced. This means however that carry bits within addition get truncated, leading to a non-accurate result. Similar results can be achieved within multiplication, where within an NxN-multiplier, a collection of inputs result in non-accurate results. Furthermore, approximate adders can be used within a multiplier to sum up accurate partial products. Other commonly investigated operations within approximate hardware is the Square(-Accumulate) operation [34] and the Logarithmic Multiplier [26]. Both operations will be shortly discussed in Section 2.5.1 and Section 2.5.3. This thesis however will only investigate Approximate Computing within Digital Multiplication.

2.2 Digital Multiplication

Almost universally, when representing numbers within a digital system, a radix-2 or binary system is chosen. The individual digits within such a number system are called 'bits'. Our general number representation is called the decimal system or radix-10. When for instance representing the decimal integer number '342' both in a radix-10 and radix-2 system, we can use the

following equation:

$$A_b = \sum_{i=0}^{n-1} x_i b^i \tag{2.1}$$

Here A is the number, x_i the individual digits of this number, and b is the base. By setting the base for both sides of the equation differently, e.g. 342_{10} and $b = 2$, the binary representation of the number 342_{10} can be found:

$$342_{10} = 3 * 10^2 + 4 * 10^1 + 2 * 10^0 \tag{2.2}$$

$$342_{10} = 101010110_2 \tag{2.3}$$

$$101010110_2 = 1*2^8+0*2^7+1*2^6+0*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0 \tag{2.4}$$

This number can thus be represented using 9 bits. Which means that to multiply this integer, at least a 9-bit or 9x9 multiplier is needed. This simple binary integer representation is also called 'unsigned', since it does not represent negative numbers. Given that the rest of this thesis will only focus on unsigned digital multipliers, going into detail in the various ways of how negative number representations (sign-magnitude, ones' complement, two's complement) work is out of scope.

When looking at pure combinational logic multipliers, unlike sequential multipliers that take multiple clock cycles for one calculation, one of the most basic methods to multiply binary numbers is shown in Listing 2.1. Essentially it is very similar to standard decimal 'long multiplication', whereas for multiplying two four-bit input values, sixteen partial products are created. Each individual digit within the partial products can be realized with an "AND" gate, which means this multiplier can be realized with just AND-gates and adders. However, there are various ways outside this multiplication algorithm to design digital multiplications. The rest of this thesis will cover more types, and apply approximations to it.

2.3 FPGA and Design Flow

One example of a technology where digital multiplication can be applied to are FPGAs. This thesis will heavily focus on FPGA technology.

FPGAs are Field Programmable Gate Arrays. Effectively, these are integrated circuits of inter-connectable 'logic blocks' that can be programmed

```
1                                          X3      X2      X1      X0
2                                   *      Y3      Y2      Y1      Y0
3                                          ---------------------------
4                                          X3Y0    X2Y0    X1Y0    X0Y0
5  +                        X3Y1           X2Y1    X1Y1    X0Y1
6  +                X3Y2    X2Y2           X1Y2    X0Y2
7  +        X3Y3    X2Y3    X1Y3           X0Y3
8  ---------------------------------------------------------------------
9  Z7       Z6      Z5      Z4      Z3      Z2      Z1      Z0
```

Listing 2.1: Binary long multiplication of two 4-bit numbers, showing all partial products

after manufacturing of the chip. Commonly, these ICs are seen as an alternative to ASICs (application-specific integrated circuits), a fine-coarse integrated circuit technology, where gates are the smallest building block. While ASICs generally perform better in terms of capacity, performance and cost per chip, FPGAs time-to-market and upfront production costs are substantially lower. Furthermore, the reprogrammability of FPGA introduces abilities as in-system programming or in-field updating of existing hardware, that ASICs lack. [6]

2.3.1 VHDL

Similar as to designing ASICs, FPGAs are generally programmed using Hardware Description Languages such as VHDL or Verilog. For instance, VHDL describes a piece of hardware using an 'Entity' and the 'Architecture'. A basic example of a 2-bit VHDL multiplier is shown in Listing 2.2. Here, the Entity describes the multiplier by assigning it two inputs (a, b) and an output (prod), with datatype 'STD_LOGIC_VECTOR'. These are essentially vectors or arrays of individual bits. The generic 'word_size' is used to describe the size of the multiplier by changing the width of the logic vectors.

In the Architecture, here named 'Behavioral', the functionality of the multiplier is described. The outputs are converted to unsigned integers, multiplied and the outcome is converted again to a logic vector. This vector is assigned to the output vector 'prod'. The multiplication operator '*' is defined in the 'IEEE.NUMERIC_STD' library, and thus is not part of the default VHDL specification. This way of describing the multiplier is very abstract . Besides the in- and output ports, this piece of VHDL code does not describe any structural aspects of the multiplier and leaves the 'synthesizing' tool free to implement the multiplication operator in any way possible. Section 4.2 will

look into this way of describing multipliers in more detail. VHDL is not limited to only this way of describing multipliers. It is perfectly possible to describe a multiplier like in Section 2.2 consisting of AND-gates and adders instead.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4
5  entity mult is
6  generic (word_size: integer:= 2;
7  Port (
8  a : in  STD_LOGIC_VECTOR (word_size-1 downto 0);
9  b : in  STD_LOGIC_VECTOR (word_size-1 downto 0);
10 prod: out STD_LOGIC_VECTOR (word_size * 2 - 1 downto 0));
11 end mult;
12
13 architecture Behavioral of mult is
14 begin
15 prod <= STD_LOGIC_VECTOR(unsigned(a) * unsigned(b));
16 end Behavioral;
```

Listing 2.2: Accurate Multiplier VHDL

2.3.2 Design Flow

After designing the digital circuit, generally a commonly used design flow is adopted before it is implemented on the actual FPGA hardware. Generally after designing a digital circuit, it is necessary to verify the result. In VHDL this is done my making testbenches. This testbench is generally a non-implementable VHDL file that 'connects' to the VHDL entity that is about to be verified. By supplying the Entity with known input/output combinations, the architecture can be tested.

After the design is verified, the synthesis tool needs to convert the HDL into logic. For ASICs this means gate-level representation, however FPGAs are not such a fine-grained architecture. Combinational logic is mostly limited to be applied to Lookup-table structures or LUTs with defined inputs and outputs. Depending on the FPGA manufacturer or FPGA series, extra functionality can be added such as internal MUX structures, fast chain-adders or DSP slices. Sequential logic is implemented using flip-flops. During the implementation stage, or also called Place and Route (P+R), the synthesis design is implemented using the logic primitives available on the FPGA and paths between them are routed. After the implementation is completed, a

bitstream can be generated to program the FPGA hardware. After each step within this design flow, is possible to validate functionality, power-estimation, area-estimation and timing of the design. However, the reliability of of these validations is dependent on the step in the design process. Early estimations for power-consumption do not have the necessary information available to make an accurate guess, and thus are only relevant very early in the design process.

An example of such an FPGA Design Flow is seen in Figure 2.1. Here external tooling is presented in the form of debugging tools, VHDL coverage and MATLAB/Simulink models. The design flow presented in this thesis, and further explained in Section 3.2, is based upon this flow but other external tooling is used. The tooling "Approxy" goes through the same steps towards implementation for the chip, and allows simulation at all steps of the process. However in this thesis, post-synthesis simulation is not being done. Behavioral Simulation is applied to early detect any mismatches between the VHDL and the multiplier model, without having to wait on the time-intensive steps of Synthesis and P+R. Post-Implementation Simulation is used for the most accurate simulation of power consumption. Here also timing values and utilization will be analyzed. These requirements will be discussed in the following section.

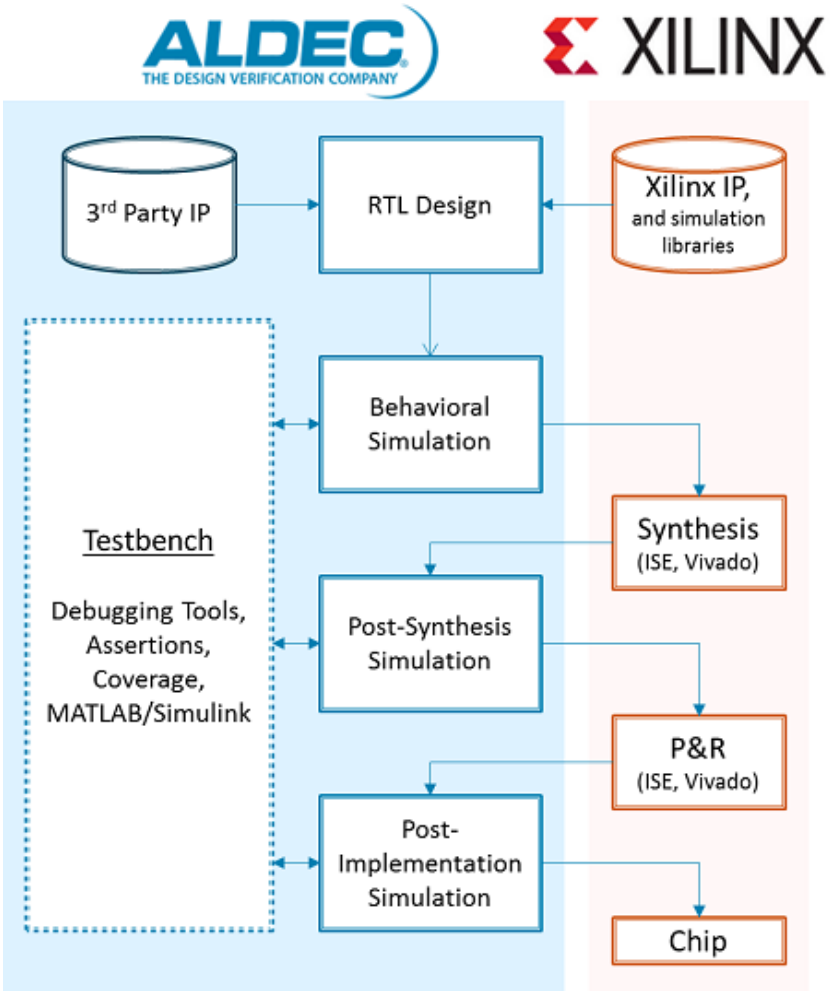


Figure 2.1: Default Xilinx Design Flow [7]

2.4 System Requirements

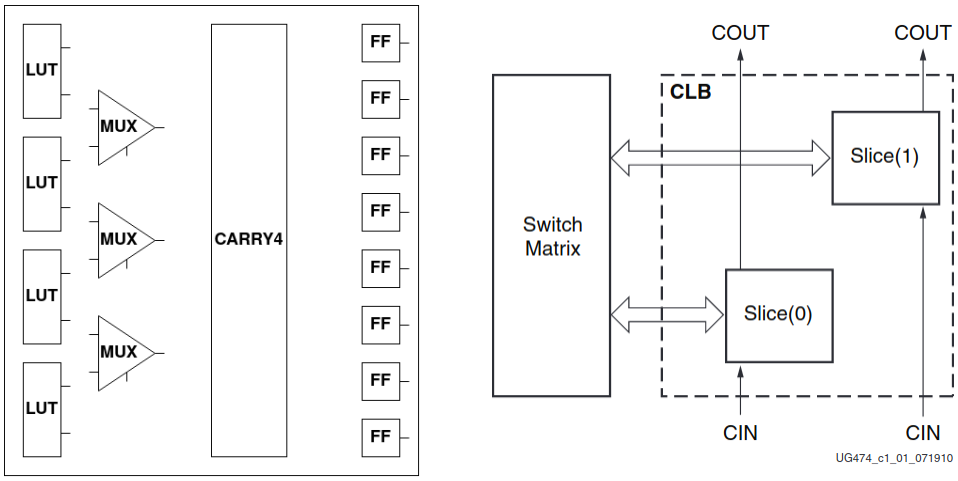
In this Section, the System Requirements are discussed that are applicable to Approximate Computing on the FPGA.

2.4.1 Slices and Chip-area

System requirements in terms of Chip-area for FPGAs are often expressed in terms of Slices, LUTs, CARRY4 adders and Flip-flops. These are essentially the programmable buildings blocks of the FPGA, on which the designed logic gets implemented. The programmable logic of a Xilinx FPGA consists of 'Configurable Logic Blocks' or CLB for short. The Xilinx CLBs consist of two identical 'Slices'.

In Figure 2.2a, a simplified overview of the Xilinx 7-Series Slice is shown. One slice is an interconnected collection of four look-up tables, three multiplexers, one CARRY4 adder and eight flip-flops. Figure 2.2b shows the arrangement of the slices within one CLB. Inputs and outputs are generally routed via the Switch Matrix. One exception being the CARRY4 adders that can be chained via their CIN/COUT pins. The internal connections between elements within the slices, and how the slices connect to each other constraint the design possibilities. For instance, the multiplexers do not accept inputs from the Switch Matrix, given they are directly connected to the LUT outputs. Two of the multiplexers are called 'MUX F7', these are directly connected to two LUT outputs. The other multiplexer, 'MUX F8', connects directly to the output of the two 'F7' multiplexers. LUTs within these FPGAs have 6 inputs and 2 outputs. A LUT like this can implement a 6-input logic function. Combining these with the internal multiplexers, logic functions up to 27-bit wide can be achieved within a single slice. More in-detail limitations of the LUTs used within the Xilinx 7-Series, the LUT6_2, is shown in Section 4.1. The CARRY4 adders, flip-flops and the LUTs accept inputs from the Switch Matrix. It has to be noted that internal latency is lower than data that flows over the Switch Matrix, i.e. a flip-flop connected to a LUT within its own slice experiences less delay than a flip-flop input connected to a LUT from another slice. Hence, the slices have 2:1 ratio of flip-flops to LUT.

The number of LUTs, FFs and Slices are physically limited. Improving a design in terms of area can make a substantial difference. On FPGAs that share multiple designs, a reduction in area can make the difference between a design that fits or does not. Reducing the area of the design in a pre-



(a) Basic Overview Xilinx 7-series slice (b) Arrangement slices within the CLB[8]

Figure 2.2: Overview Xilinx 7-series FPGA

production stage could also mean that a tinier FPGA package can be chosen, resulting in a less costly bill of materials. But FPGA utilization is also a factor in the overall power consumption of the design.

While FPGA area might not directly correlate to lower power consumption, bigger designs tend to consume more power than smaller ones. To fairly compare different approximate multipliers and see what factors affect power consumption, FPGA area should be investigated as well.

2.4.2 Latency

Another factor that should be considered is latency. Latency is caused by a combination of factors. LUTs, for instance, cannot instantly produce the correct result when the inputs change. Although FPGAs are digital, the internal hardware still has analogue behavior. This time before these combinational logic elements settle on the correct output, is called the logic delay. The other factor is routing delay. In Section 2.4.1 it was already established that the Switch Matrix within the FPGA introduces delay. Often this is minimized by correctly placing and routing the design, so paths are generally small, but nevertheless this is a contributing factor to the experienced latency of the design. Why this delay matters can be seen in Figure 2.3. Here a constant clock signal is seen, but the input signal at D changes within the period. The flip-flop 'holds' the signal D at the rising edge of the clock cycle and outputs this to Q. Any changes at D within a clock cycle, for instance due

to combinational logic settling on new input are simply ignored. If the total delay exceeds the clock cycle, outputs will be erroneous since the 'holding' happens during the settling of the logic and the design should simply operate at a lower clock speed.

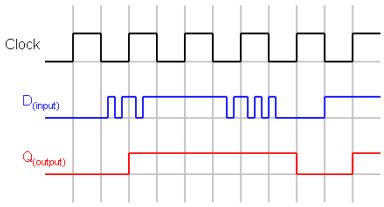


Figure 2.3: Example of Flip-flop timing [9]

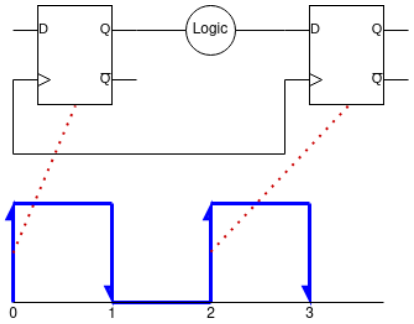


Figure 2.4: Example of Register-to-Register path

A more concrete example can be seen in Figure 2.4, showing a Register-to-Register path. Here, the first flip-flop launches the data on the first rising-edge. Data transfers through the logic, and gets captured on the next rising-edge in the last flip-flop. The time period till the logic has settled, and thus is 'ready' to be captured by the next flip-flop is called the 'Data Arrival Time'.

Definition 1 ***Data Arrival Time (setup)*** *= Launch Edge Time + Source Clock Path Delay + Data-path Delay [10]*

The requirement within the system is dependent on when the second flip-flop captures the data from the combinational logic. This is called the 'Data Required Time', and is besides the clock speed dependent on any delays within the clock path, uncertainty and physical limitations of the flip-flop.

Definition 2 ***Data Required Time (Setup)*** *= Capture Edge Time + Destination Clock Path delay - Clock Uncertainty - Setup Time [10]*

The difference between these two time values is called the 'slack'. Within latency analysis, this is an often used value. Essentially, this is the 'leeway' or 'clearance' of a digital system.

Definition 3 ***Setup Slack*** *= Data Required Time - Data Arrival Time [10]*

When Vivado implements a design, the slack of each path can be calculated. The 'Worst Negative Slack' (WNS) corresponds to the critical path of the design. If every path within a digital system produces a positive slack, the timing holds. A negative slack or WNS directly results in a setup violation. Design alterations or simply a slower clock speed are necessary for the correct functionality of the design. However, if the WNS is positive this means that the 'Data Required Time' is sufficient for the Arrival Time for every internal path. This means there is leeway within the constrained design to improve clock speeds.

Definition 4 ***Worst Negative Slack (WNS)*** *"This value corresponds to the worst slack of all the timing paths for max delay analysis. It can be positive or negative." [10]*

$$t_{min} = t_{constr} - WNS \tag{2.5}$$

$$F_{max} = (t_{min})^{-1} \tag{2.6}$$

Given that the functions of determining 'WNS' require a set clock frequency, first a 'dummy' frequency has to be set: t_{constr}. For instance, a design is synthesized using clock constraints. This clock constraint is set at 100MHz ($t_{constr} = 10ns$). After synthesis and implementation, a WNS of $+2ns$ in seen, which is the leeway of the design. To reach a WNS of 0ns, $t_{min} = 8ns$ or $F_{max} = 125MHz$ can be used for this design without causing a setup violation. This is the maximum frequency that still holds for this design.

Combinational Logic

Within purely combinational logic, it is difficult to properly investigate latency and relate this value to something like a maximum frequency, given any required data like 'Clock Path delays' or 'Setup times' are simply non-existent. Vivado often interprets combinational logic, as logic being directly connected to the input/output pins of the FPGA package. For small combinational logic designs this works, the in/out ports in the VHDL Entity are simply connected to an internal buffer which is physically connected to one

of the package pins. For bigger designs, this quickly becomes a problem since the pins are physically limited. Out-of-Context (OOC) synthesis[10] removes the connection to the input/output buffers, while keeping the entity ports labelled as so. This means however another limitation to the accuracy of the latency analysis. Generally, designs are placed closely to input/output buffers to reduce path delays. This is not the case for OOC designs, given this constraint does not exist. Another design choice to work around the limitation of limited pins, is to design sequential logic around these bigger combinational designs. For any actual design scenarios, this is most likely the correct choice. However, for analysis of pure combinational logic on FPGA technology this would add a design-specific factor that will influence latency statistics, area usage and power consumption.

For combinational logic, Vivado provides functionality in terms of 'timing constraints', to be able to model the timing properties. Within the timing constraints, the designer can set up a virtual clock speed. By constraining the input- and output-delay relating to this virtual clock, the designer can simply replace the non-existent timing data from the flip-flops with a specified value. By creating a model that has ideal flip-flops, no clock-path delays or clock uncertainty, these delays can be set to $0ns$. This results in the following model:

Model 1 ***Data Arrival Time (setup)*** *= Data-path Delay*

Model 2 ***Data Required Time (Setup)*** *= Capture Edge Time (Virtual Clock)*

Model 3 ***Setup Slack*** *= Capture Edge Time (Virtual Clock) - Data-path Delay*

Model 4 ***WNS*** *= Capture Edge Time (Virtual Clock) -* ***Worst*** *Data-path Delay*

$$t_{min} = t_{constr} - WNS = t_{constr} - (t_{constr} - t_{delay}) \tag{2.7}$$

$$F_{max} = (t_{min})^{-1} = (t_{delay})^{-1} \tag{2.8}$$

Model 5 *Combinational Logic Model:* $F_{max} = (t_{delay})^{-1}$

This model creates a latency-model for combinational logic that is only dependent on internal data-path delays of the design, specifically the delay of

the 'critical path'. The path with the highest logic delay will determine the maximum achievable frequency. While this model will not show expected maximum frequencies achievable for combinational logic designs used within other bigger sequential designs, it does create a model where latency can be compared between different configurations of combinational logic, such as different approximate multiplier designs. During analysis within Vivado, it's not necessary to constraint the virtual clock, or set up the port delays. The function report_timing −nworst 1 −path_type end will report the input-output path delay of the critical path.

2.4.3 Power Consumption

To be able to understand FPGA Power Consumption, an investigation to the actual hardware is needed. FPGAs, just like most digital technology, is designed on basis of CMOS. CMOS power-consumption is well understood and can be described using the following general formula:

$$P_{avg} = P_{switching} + P_{short-circuit} + P_{leakage}[11] \tag{2.9}$$

$$P_{avg} = \alpha_{0\to1} C_L V_{dd}^2 f_{clk} + I_{SC} V_{dd} + I_{leak} V_{dd}[11] \tag{2.10}$$

An example of a CMOS inverter, consisting of a PMOS and NMOS transistor can be seen in Figure 2.5a. Out of these equations, a *Static* and *Dynamic* contribution to the average power consumption can be seen. The *Static* power consumption is due to short-circuit current that flows from V_{dd} to ground at moments where both the PMOS and NMOS transistors within a CMOS circuit are active. Leakage current is due to sub-threshold effects which is a result of design consideration during fabrication of the chip. In Equation 2.11, the *dynamic* or switching power consumption for a CMOS gate is shown.

$$P_{switch} = \alpha_{0\to1} C_L V_{dd}^2 f_{clk} \tag{2.11}$$

This power consumption is dependent on the clock frequency f_{clk}, voltage V_{dd}, load capacitance C_L and $\alpha_{0\to1}$ which is "the average number of times the node makes a power consuming transition in one clock period." [11] The dynamic power happens due to the capacitive load of the CMOS circuit being charged when a transition from 0 to V_{dd} happens. If this happens every clock cycle, the switching power is simply $C_L V_{dd}^2 f_{clk}$, however it is fairly unlikely within a circuit that every node makes this transition every cycle. This transition can be modelled by using the logic function that the CMOS gate performs. Given a known distribution of inputs connected to an AND-gate,

for example, the probability that a $0 \rightarrow 1$ transition happens can be calculated. The possible inputs are $(a, b) = (00), (01), (10), (11)$. Using a uniform distribution, these inputs are equally likely. For an AND-gate, the chance that the output is **0** is $\frac{3}{4}$ making the probability $\alpha_{0\rightarrow1} = p(0)p(1) = \frac{3}{4}\frac{1}{4} = \frac{3}{16}$. When using a non-uniform distribution for the input-values, which is common for various applications, this probability does not hold. [11]

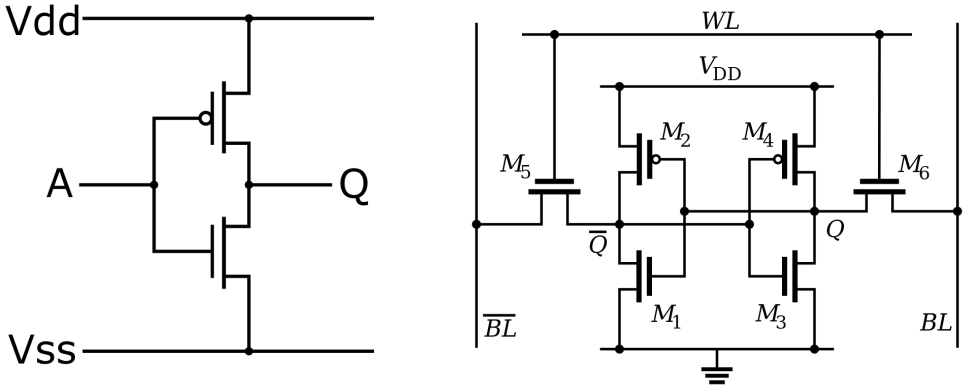
FPGAs however, do not have the flexibility that CMOS designs such as ASICs hold and have a predetermined structure. Most FPGAs are SRAM-based, seen in Figure 2.5b. One Static Random-Access Memory structure consisting of 6 MOSFETs can hold one bit, and are generally in FPGA used within the LUT as seen in Figure 2.6. The Figure shows a 4-bit LUT with a single output, any 4-bit input logic function can be executed by this LUT. When programming the FPGA, a bit-mask gets written to the SRAM circuits within the LUT, whereas the inputs of the FPGA are connected to the various MUX structures to select the desired output during operation. For instance, the input of "0000", produces the result of the top SRAM structure within the Figure. Every combination of input bits connects to a single SRAM source. Given that the FPGA has many of these general purpose structures, the dynamic power consumption for a single FPGA can be described using a different formula:

$$P_{switch} = V_{dd}^2 f_{clk} \sum C_i U_i S_i [12] \tag{2.12}$$

Here the factor $\alpha_{0\rightarrow1} C_L$ is replaced by a summation over all FPGA resources consisting of the effective capacitance C_i, the utilization for each resource U_i and the switching activity S_i. [12] When estimating power consumption for an FPGA design, it is possible to do a post-placement analysis to determine this switching behavior. By writing a test bench that supplies the design with accurate real-life data with the correct distribution, the tooling can estimate power consumption for production. Together with the proprietary power model used by Xilinx Vivado, the power consumption of the FPGA can be estimated.

2.4.4 Error Metrics

To be able to signify the quality of an error-prone system, various error metrics have been developed within literature. The field of approximate computer therefore also uses a set of error metrics to be able to compare designs.



(a) Basic CMOS Inverter[13] (b) SRAM Circuit Diagram[14]

Figure 2.5: Circuit Diagrams of CMOS and SRAM

These metrics can be application-specific such as Peak Signal-to-Noise Ratios, where approximate computing modules are compared within a system, such as image compression. Alternatively, various metrics exist within literature that depend only on the digital design. [1]

In the article by Han et al. [15], about various Approximate Computing paradigms, they investigate in Chapter 4, various error metrics used in literature. The *Error Rate* shows the frequency of incorrect outputs and the *Error Magnitude* shows the maximum magnitude of the erroneous outputs within an approximate design. One quite simple approximate multiplier, that will also be investigated in this thesis, is M1[16]. This multiplier is like an accurate 2-bit multiplier with one modification: $3 * 3 \rightarrow 7$. Given there are $2^{2n} = 16$ possible outcomes, the *Error Rate* is $1/16$, the *Error Magnitude* is 2.

Another way to define these errors is the *Error Distance*, which is simply the distance between the correct output and the inexact output: $|y_i - x_i|$. For M1, the *Error Distance* of $3 * 3 \rightarrow 7$ is simply 2. By taking the mean over this distance for various pairs of inputs and outputs, the *Mean Error Distance* or *Mean Absolute Error* metric is formed:

$$MAE = \frac{\sum_{\forall i} |y_i - x_i|}{N} \tag{2.13}$$

This metric is useful to define the accuracy of an approximate multiplier without having to define every individual inaccuracy. Often within literature[17], the *Mean Absolute Error* is generalized to a form where the mean is taken over all input/output pairs, and therefore the MAE over a uniform distribu-

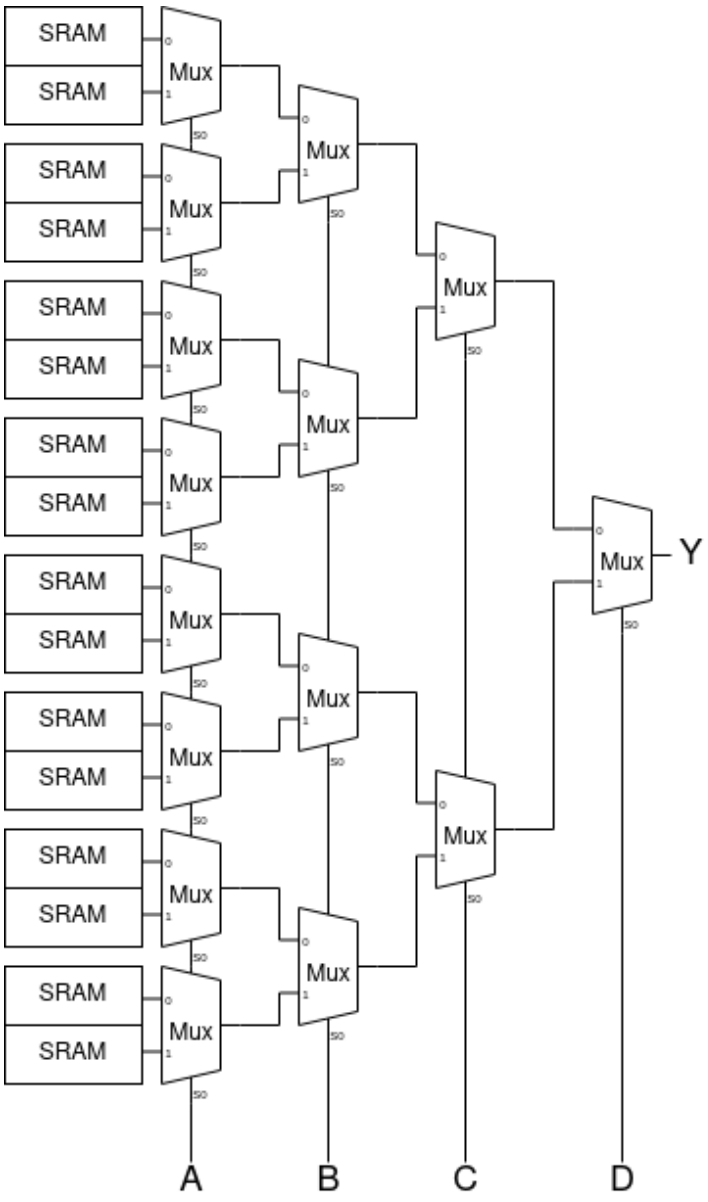


Figure 2.6: Circuit Diagram of SRAM-based 4-bit LUT

tion:

$$MAE_{uni} = \frac{\sum_{\forall i} |y_i - x_i|}{2^{2n}} \tag{2.14}$$

Where n is the width of the multipliers operands. For example, a 4-bit multiplier has 2^8 possible outputs. This generalization is however not necessary, and any distribution of input/output pairs can be chosen. Various ways exist to normalize this mean, such as normalizing by using the mean of the accurate values, or over the range. This is the *Normalized Mean Absolute Error* that is used in this thesis, and has as a feature that the normalization removes the factor of the multipliers' width from the metric, making comparisons of approximate multipliers of different sizes possible:

$$NMAE = \frac{MAE}{2^{2n}} \tag{2.15}$$

Another used metric by various papers such as the paper by Ullah et al. investigating an FPGA library of multipliers[19], is the *Average Relative Error* or also named *Mean Relative Error*[17]:

$$ARE = \frac{1}{N} \sum_{i=1}^{N} |\frac{y_i - x_i}{x_i}| \tag{2.16}$$

This metric is essentially the same as the MAE normalized over the mean of the accurate values. Another used metric is the *Mean Squared Error*. Given that it squares the *Error Distance*, bigger but incidental errors within an approximate multiplier are heavier punished.

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - x_i)^2 \tag{2.17}$$

2.5 Applications for Approximate Multipliers

In Section 2.1, a short overview was given to the various Approximate Computing techniques which were generalized as aiming "to improve energy-efficiency and/or performance of a digital system by introducing a loss of quality to the overall result". In the last section, various metrics have been discussed to signify this loss of quality in terms of accuracy. Approximate Computing cannot be applied to every generic application; the loss of quality needs to be acceptable within the system. This section will discuss various applications within literature where this intrinsic imprecision of approximate computing can create acceptable results, in exchange for lower power consumption or FPGA resource use.

2.5.1 Radio Astronomy

Radio Astronomy is a field that is dominated by power-intensive digital systems, and input signals can be regarded as Gaussian noise, making it an attractive field for the application of Approximate Computing. For instance, the power consumption of a modern computer processing system utilized by the SKA1-MID is 7.2MW [20]. In the PhD Thesis from Gillani[21], a case study is presented for the Radio Astronomy Calibration Algorithm 'StEFCal', an iterative algorithm that estimates the individual gains of the antenna array. This calibration algorithm makes use of the Least-Squares (LS) algorithm. Here, an accelerator is presented where an Accurate and Approximate Core is used, where the algorithm uses the Approximate Core for the initial iterations.

One common used structure within general signal processing is the Multiply-Accumulate (MAC) function as described in the next equation:

$$z(t) = a(t)b(t) + z(t-1) \tag{2.18}$$

MAC operations are commonly used within applications that use mathematical functions like the Dot Product, Matrix Multiplication or Convolutions. For floating point numbers, often devices such as CPUs or GPUs have their own dedicated solution. The Least-Squares algorithm makes usage of such MAC operations. In Figure 2.7, part of the calibration signal-flow is shown, where the yellow square represents the MAC operation. Next to the MAC op-

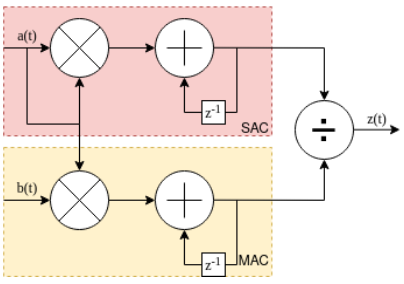


Figure 2.7: Signal Flow Graph of Least Squares Operation

eration, the Least-square algorithm also makes use of the Square-Accumulate (SAC) operation. While the SAC operation could potentially also be implemented using general multipliers, more power-efficient implementations that purely calculate the square are available. Radio Astronomy signals are represented by complex numbers, expanding general multiplication to:

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc) \tag{2.19}$$

This means that the MAC operation for complex numbers requires four multipliers. For comparison the square operation only requires two square functions, making the Least Square operation dominated by the MAC operation:

$$(a + ib)(a - ib) = a^2 + iab - iab + b^2 = a^2 + b^2 \tag{2.20}$$

In the thesis, an LS accelerator is proposed where these four multipliers and two square operation are approximated. For the LS approximate core they claim a power reduction of 41% resulting in an efficiency gain of 23.4% of the whole LS accelerator, while reaching satisfactory results. [21]

2.5.2 Image Processing

Another field where Approximate Computing can be used, due to its inherent error-tolerance, is Image Processing. In the paper by Ullah et al. [19], demonstrating a library of approximate FPGA multipliers, a Gaussian noise removal filter is applied to an image to analyze the effect of the approximate multipliers. Here the PSNR (Peak Signal-to-Noise Ratio) is used, an application-specific error metric. Similarly, in a paper by Niemann et al. [22] a Gaussian smoothing filter is applied. Both papers report favorable results, but unfortunately do not report on the overall FPGA filter architecture and simulate the filtering operation externally using the behavioral model of the multipliers.

However, to explain how approximate multipliers can help increase power efficiency for applying filters, we can look at the convolution integral and its discrete version:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau \tag{2.21}$$

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n - m] \tag{2.22}$$

This integral can be extended to a two-dimensional version, suitable for the application of filter kernels to 2D images:

$$r(i, j) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(x - i, y - i)H(i, j) \tag{2.23}$$

Where $r(i, j)$ is the output image, $I(x, y)$ is the original image and $H(i, j)$ is the filter kernel. From this 2D Convolution operation, it can be quickly seen that these types of image filtering applications make heavy usage of

multipliers and adders.

In the journal article "Implementation of a Fixed-Point 2D Gaussian Filter for Image Processing based on FPGA" by Cabello et al. a possible implementation is shown, where a [512 x 512] image is convolved with an [3 x 3] kernel, shown in Figure 2.8. This [512 x 512] image is extracted into a one dimensional vector, after which each element is multiplied with kernel image w using a sliding window. The circuit makes heavy usage of MAC operations.

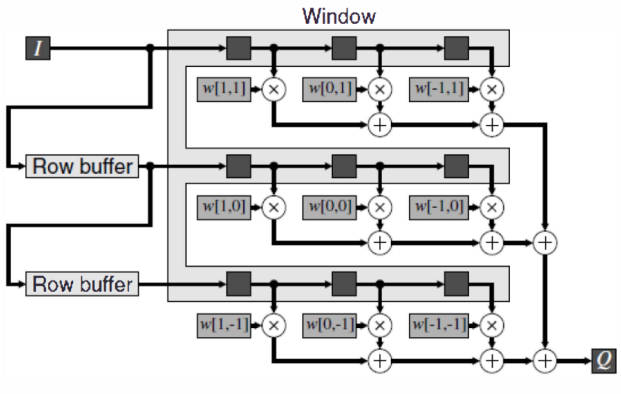


Figure 2.8: "Extraction of a [3 x 3] sub-image and MAC operations"[23]

2.5.3 Neural Networks

The last field this Section will be looking into is Artificial Intelligence. In a journal paper by Torres-Huitzil et al. [24] a study is done to review the Error, Fault and Failure tolerance of Artificial Neural Networks (ANN), where they claim that while ANNs have no inherent Fault Tolerance unless properly designed, the Error Resilience of Neural Networks can be used to implement Approximate Computing: "The tolerance to approximation, for instance, can be leveraged for substantial performance and energy gains through the design of custom low-precision neural accelerators that operate on sensory input streams[24]".

At its basis, a Neural Network is a designed network of elementary computing units called 'Neurons'. A Figure of such a single 'Neuron' can be seen in Figure 2.9, which implements the following function:

$$O_i = \Phi(\sum_i w_i x_i + b_i) \tag{2.24}$$

Where x_i are the inputs, w_i the weights, b_i the bias, and Φ is an Activation Function, such as the Sigmoid function. Using a learning algorithm, the weights and biases of such a network can be calculated. A state-of-the-art Convolutional Neural Network (CNN) is highly complex, and could be using millions of Multiply-Accumulate operations to process. [25]

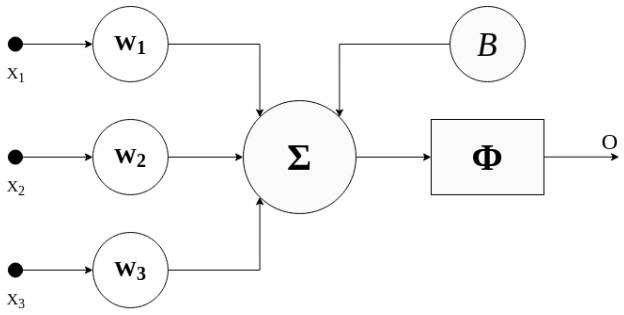


Figure 2.9: Artificial Neural Network(ANN) Neuron

In a conference paper by Wang et al.[25] an "Approximate Multiply-Accumulate Array" architecture is presented to implement a Convolutional Neural Network (CNN) on an FPGA. In this design, they replace the MAC units with their own "Approximate Multiply-Accumulators" based upon a logarithmic multiplier algorithm by J. Mitchell [26] which introduces error. The multiplier is iterative, meaning the accuracy of the multiplier can be increased by increasing the amount of iterative steps taken. For a two-iteration implementation, they claim an improvement of F_{max} in Post-Synthesis Analysis of 10.7% compared to using exact multipliers while having a loss of 1.6% in terms of accuracy for 8-bit, and 0.001% for 12-bit. [25]

2.6 Go Programming Language

Part of the tooling used in this thesis, described in Section 3.2, is written in the programming language Go. In this section, some key elements of the language Go are explained to get an understanding of the framework 'Approxy'. Go is a statically typed, compiled language influenced by C but with an emphasis on simplicity. A distinct difference between classic Object-Oriented Programming languages such as C++/Java, and Go can be seen in the way inheritance is defined. While C++/Java define inheritance by classes that can 'inherit' the methods and thus the functionality of other classes, Go does not use classes or this form of inheritance.

In Go, methods are added to types. A similar structure like a Class would then be implemented in Go by using a 'Struct'. These Structs look very similar to a C struct. An example of such a Struct and a method is seen in Listing 2.3. When defining such a Struct, only the fields are explicitly defined when declaring the type. Methods can afterwards be added to the Struct. Inheritance is applied to these Structs by using Interfaces. These interfaces describe a collection of methods. Every Struct that implements these collections of methods implicitly inferences this Interface. An example is shown in Listing 2.4.

When looking at both the Struct 'Recursive4' in Listing 2.3 and the Interface 'Multiplier' in Listing 2.4: If 'Recursive4' would next to the *Overflow* method also implement *ReturnVal* and *MeanAbsoluteError*, it would inherit the 'Multiplier' type interface implicitly. This would mean that for any function that requests the 'Multiplier' type as input, 'Recursive4' can be used as well.

Another clear difference between Go and a lot of other programming languages is it sizeable standard library. 'Approxy' is only using this standard library, and does not import any online libraries from third-parties. For 'Approxy' to be able to generate VHDL, it uses the templating engine provided by the standard library.

The templating package from Go is often used for serving dynamic HTML within web-services by importing the 'html/template' library, but Go also provides a 'text/template' library that can be used for VHDL and TCL files. This library provides a templating system with its own functions and syntax to generate files. An Example is shown in Listing 2.5. This VHDL template shows a simple VHDL multiplier using the IEEE NUMERIC_STD Library. All the templating syntax is encapsulated using the "{}" accolades or curly brackets. Given the simplicity of this multiplier, the template only allows the *EntityName* and the *BitSize* to be modified. These variable names are preceded by a 'dot' to denote a field. When a Go Struct is passed to the templating engine, the value of these fields are filled into the template. Furthermore, the templating system also provides methods to apply simple boolean logic within if/else structures. It is also possible to range over Go arrays and thus generate blocks of VHDL code over the length of the array. Calls to Go functions can also be made. [27]

```
type Recursive4 struct {
    EntityName     string
    BitSize        uint                        //Default to 4
    OutputSize     uint                        //Default to 8
    LUTArray       [4]VHDLEntityMultiplier //Size of 4
    VHDLFile       string
    TestFile       string
    OverflowError bool
}

func (r4 *Recursive4) Overflow() bool {
    return r4.OverflowError
}
```

Listing 2.3: Recursive4 Struct, and Overflow() method

```
type Multiplier interface {
    ReturnVal(uint, uint) uint
    Overflow() bool
    MeanAbsoluteError() float64
}
```

Listing 2.4: Multiplier Interface

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity {{.EntityName}} is
generic (word_size: integer:={{.BitSize}});
Port (
a : in  STD_LOGIC_VECTOR (word_size-1 downto 0);
b : in  STD_LOGIC_VECTOR (word_size-1 downto 0);
prod: out STD_LOGIC_VECTOR (word_size * 2 - 1 downto 0));
end {{.EntityName}};

architecture Behavioral of {{.EntityName}} is
begin
prod <= STD_LOGIC_VECTOR(unsigned(a) * unsigned(b));
end Behavioral;
```

Listing 2.5: Accurate Multiplier Template

Chapter 3

Methodology

In this chapter, the method of comparing FPGA multiplier models is discussed. In Section 3.1, the current state of affairs of available automated computer tooling for Xilinx FPGAs is discussed. Furthermore, a couple articles are discussed that developed their own tooling to perform analysis on a sizeable design space. In Section 3.2 an overview to the Approxy framework is given. Section 3.3 till Section 3.6, explain the general Approxy multiplier model, how a model is verified, implemented and analyzed. Finally, in Section 3.7, the full workflow to analyze a recursive multiplier is shown in terms of code and analysis data.

3.1 Computer-Aided Design

Chapter 4 explains various multiplier models to investigate on the FPGA. While some are fairly simple, and require one generic VHDL file, some designs greatly expand the design-space to a point where a manual workflow is not feasible anymore. Software such as Vivado has the ability to automate this workflow using TCL. This scripting language, when executed by Vivado, has the possibility to automatize steps described in Section 2.3 in a way that the FPGA workflow can be integrated into the developers' environment. Furthermore, it is possible to extract information such as used FPGA resources, timing and power consumption. Next to this 'batch' option, Vivado also has an interactive command-line interface where the designer can execute these TCL commands, without the need of a GUI.

In projects where the scope expands beyond a couple of Vivado project files however, it might be interesting to design a completely automated workflow. In a paper by Ullah et al.[19], a design-space of 545 '8x8' multipliers are

explored using their own automated tool flow which they have made publicly available. Unfortunately it has to be noted that the publicly available tooling differs from the tool flow described in their paper. In their paper they describe using MATLAB for their behavioral modeling, in the online tooling this behavioral module is a generic script in Python that uses supplied text files describing all the possible input/output combinations of the individual multiplier. The hardware is described in VHDL, but these files are generated using a Python script utilizing a collection of for-loops and string manipulation. Neither do they supply the tooling that is used for creating the 545 multipliers, only their final designs are supplied. Applying this tooling to other models described in the rest of this thesis would require an extensive overhaul of the code and the generation of new text files describing the multipliers. Furthermore the Synthesis and P+R is not automated.

In the article "MACISH: Designing Approximate MAC Accelerators With Internal-Self-Healing" by Gillani et al.[18], a similar setup is introduced where ASIC synthesis is done in Synopsis on basis of VHDL Models, however verification is implemented between the Questasim simulation and the MATLAB behavioral models.

Due to the lack of available generic open-source tooling that help with the design-exploration of multiplier models, tooling has been developed, aimed at designing FPGA multipliers, which generates VHDL models, combines the behavioral modelling and FPGA workflow into a single workflow and has integrated verification. The previous two pointed out articles have been an inspiration to the general workflow. This tooling is explained in Section 3.2.

3.2 Approxy

This section and the following discuss the framework 'Approxy' written in Go that has been developed for this thesis to further investigate approximate multipliers on the FPGA. The project can be downloaded at `https://github.com/RickvanLoo/Approxy`.
The following features are currently implemented:

- Using models written in Go:
  - Pre-synthesis behavioral analysis of multipliers models
  - Generation of VHDL Multipliers
  - Verification of VHDL Files

- Automated Synthesis/P+R using Xilinx Vivado
- Automated Analysis Post-Placement+Route

Given the various existing models and parameters going into Approximate Multiplier design, Approxy has been designed to develop an automated approach to compare these various implementations in VHDL in a controlled and identical setup.

Using a collection of created libraries and interfaces, Approxy provides an approach for the automated design and analysis of approximate multipliers on Xilinx FPGAs. In Figure 3.1, an overview of the Approxy design flow is seen. Every multiplier requires an user-provided multiplier Approxy Model and VHDL template, shown with a light-blue background in the Design Flow. All other steps of the design flow has been taken in account for by the provided tooling within Approxy, 'xsim' and Xilinx Vivado.

Approxy has been written in Go. While this programming language is fairly new compared to programming languages most FPGA engineers could be familiar with, its syntax should be familiar to people having worked with languages such as C(++) or Java. Go is a statically typed language, but compiling times are negligibly short. This means that many programming errors, relating to mismatched types, while using 'Approxy' can already be caught quickly before runtime. Another attractive feature is its extensible standard library. Approxy only builds on its own libraries and the Go standard library, making it independent of quickly-changing third-party libraries. The only existing main Go version, v1.x, guarantees library compatibility, ensuring the functionality of this software in the future. The main reason for writing it in Go, has been the 'text/template' library. Syntax for this library is encapsulated using the "{}" accolades, which VHDL does not use itself, separating the VHDL syntax from the syntax used for Approxy VHDL generation. This means that during runtime, this library can be used to generate VHDL files on basis of Go functions and data structures.The Go compiler is available for many different operating systems and computing architectures, making Vivado itself the limiting factor on where this application can be used. A disadvantage to the language is its class inheritance system in the form of "Interfaces". While still being an "Object-Oriented Programming Language", Go's inheritance system is nothing like traditional OOP languages. This might make it confusing for programmers newly using Go. This system has been explained in Section 2.6.

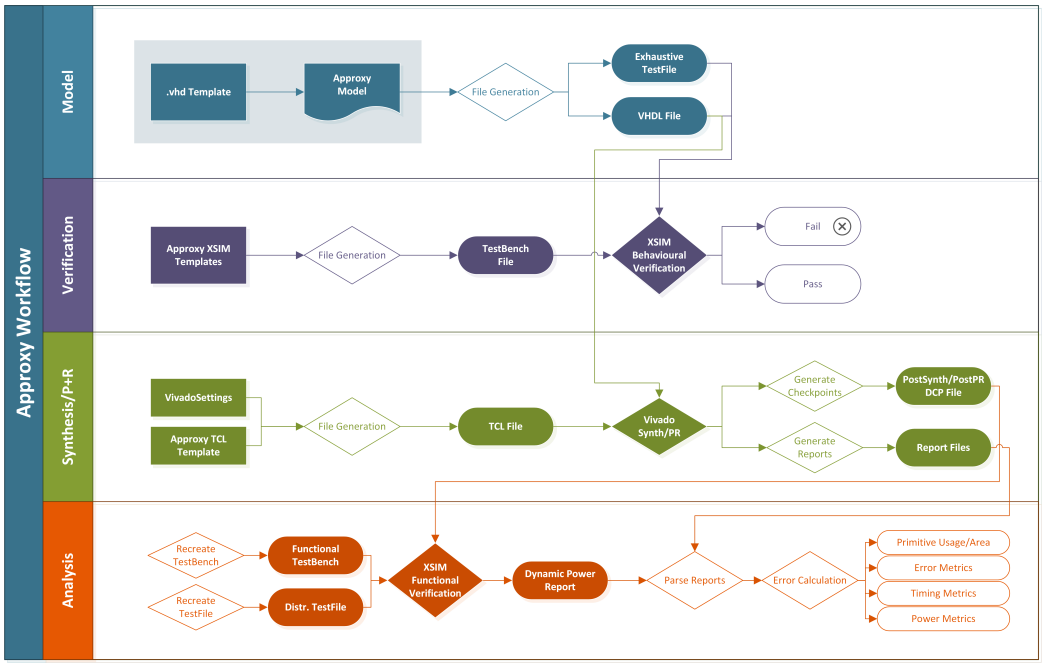


Figure 3.1: Approxy Design Flow

The next paragraphs will discuss the various implemented stages of the Approxy workflow.

3.3 Approxy Model

As described in the previous Section, the Approxy Design Flow requires two user-provided inputs. The Approxy Model is a Go file, that has a main datatype or 'struct' containing information for generation of the VHDL Multiplier. It has general methods that for instance return error metrics, a function that models the multiplication or provided functions that generate VHDL files. These will be later in detail described. Together with this model a VHDL Template has to be supplied. This is a non-synthesizable VHDL file describing either the behavior or architecture of the digital multiplier but provides data-inputs within the template using the 'text/template' syntax. These can be 'filled in' by the Approxy model, generating a synthesizable design using data-driven VHDL templates. There is no real limit in how a Approxy model is described. It is possible for instance to create a purely architectural Approxy model that incorporates other Approxy models to get a final multiplier result. One exception to these earlier described Approxy models that use VHDL templates, is the Approxy model called 'External'.

This model does not use templates, but instead uses synthesizable VHDL files as a basis to create a behavioral model. More about this is explained in Section 5.1.

The general 'Approxy' multiplier model is based around the idea that in its most simple form, the multiplier is nothing more than a combinational entity that performs the function $c = a * b$, whereas the bit-width of the output is twice as large as the input. A simple extension which expands this model to be usable for approximate computing is the deterministic function: $c = f(a, b)$. The usage of this model results in a general VHDL Entity that is usable for automated verification and integration, shown in Listing 3.1.

```
1 entity {{.EntityName}} is
2 generic (word_size: integer:={{.BitSize}});
3 Port (
4 A : in  STD_LOGIC_VECTOR (word_size-1 downto 0);
5 B : in  STD_LOGIC_VECTOR (word_size-1 downto 0);
6 prod: out STD_LOGIC_VECTOR (word_size * 2 - 1 downto 0));
7 end {{.EntityName}};
```

Listing 3.1: Required VHDL Entity for VHDL Template of Approxy Model

Every multiplier model within Approxy inherits the VHDLEntityMultiplier interface. This interface again inherits both the VHDLEntity and the Multiplier interface. An overview of all generic Approxy types and interfaces can be seen in Listing 3.3. Every Approxy Model should provide the methods seen under the VHDLEntity and Multiplier interface.

The VHDLEntity interface encapsulates all methods for a generic VHDL model. ReturnData() returns a struct containing basic information such as the VHDL Entity Name, some VHDL Generics like bit-width and Filenames. This is the minimum necessary information to be able to verify and synthesize the model, and should be in every Approxy model. Method GenerateVHDLEntityArray() is available to return information about port-mapped VHDL Entities. If a certain VHDLEntity makes use of an Entity Port map in their template, essentially importing external VHDL files, XSIM has to be able to verify and load these VHDL files. The returned array has to be an array of all EntityData structs that are necessary to perform VHDL verification. Functions GenerateVHDL(string) and GenerateTestData(string) generate VHDL for synthesizing and a TestData file for verification in the given project path. These interfaces can be individually extended with methods or fields to be able to provide extra information to the VHDL Template, however they are not publicly accessible.

The model function is encapsulated within the method ReturnVal(uint, uint). Since certain multiplier models can (internally) overflow, a method is implemented to provided to show if a model shows this behavior. Other functionality within this interface is dedicated to the functional analysis of the multiplier, in terms of error-analysis.

A special case is the Multiply-Accumulate(MAC) operation, which is the following sequential set of functions:

$$c(t+1) = c(t) + f(a,b) \tag{3.1}$$

$$c(0) = 0 \tag{3.2}$$

When comparing this in VHDL, the MAC operation is essentially a sequential extension of the earlier shown combinational model. This MAC model is already implemented within Approxy, and thus does not require to be supplied by the user. The implemented MAC model within Approxy essentially 'inherits' a general non-sequential Approxy model as earlier described. However it uses a different testbench, analysis is different and some methods have different functionality. The required VHDL Entity for this MAC model is shown in Listing 3.2.

```
1  entity {{.EntityName}} is
2  generic (word_size: integer:={{.BitSize}};
3  output_size: integer:={{.OutputSize}});
4  Port (
5  clk : in std_logic;
6  rst : in std_logic;
7  A : in  STD_LOGIC_VECTOR (word_size-1 downto 0);
8  B : in  STD_LOGIC_VECTOR (word_size-1 downto 0);
9  prod: out STD_LOGIC_VECTOR (output_size-1 downto 0));
10 end {{.EntityName}};
```

Listing 3.2: Required VHDL Entity for VHDL Template of Approxy MAC Model

While the standard Approxy Model has a pure function that returns always the same output on basis of two inputs. The MAC ReturnVal(uint, uint) function in this case is an impure function. A call to this function, increases time by one, so a subsequent call will return a different value. A Reset() method is added to set $t = 0$.

Next to being able to implement single VHDL models, Approxy provides a Scaler model to linearly scale VHDLEntityMultiplier interfaces or in other words: other Approxy multiplier models. This scaler can be used to increase

FPGA area coverage by implementing an N amount of (approximate) multipliers to accurately model the dynamic power consumption. This is being done by using a VHDL package including new types that create arrays of the input and output vectors of a single multiplier. In combination with the VHDL Generate statement, a positive integer amount of independent multipliers can be realized.

Finally, when implementing an Approxy model one has to adhere to the restrictions within the framework. This means that every newly created model must have the same VHDL entity description in its template and within Go implement the required interface functions. Given that a lot of code can be reused, and general functions are available, this means that integrating a new model is quite a quick task.

```
1  type VHDLEntityMultiplier interface {
2      VHDLEntity
3      Multiplier
4  }
5
6  type VHDLEntity interface {
7      ReturnData() *EntityData
8      GenerateVHDL(string)
9      GenerateTestData(string)
10     GenerateVHDLEntityArray() []VHDLEntity
11     String() string //MSB -> LSB
12 }
13
14 type Multiplier interface {
15     ReturnVal(uint, uint) uint
16     Overflow() bool
17     MeanAbsoluteError() float64
18 }
19
20 type EntityData struct {
21     EntityName string
22     BitSize    uint
23     OutputSize uint
24     VHDLFile   string
25     TestFile   string
26 }
```

Listing 3.3: Approxy Types

3.4 Verification

The verification system is quite simple, and supports both the earlier described standardized Approxy model and the MAC model. The VHDL test bench loads the generated test file as specified in the EntityData struct. The TestData file is generated in binary format. Every line is a new singular test requiring input a, b and output c separated by a white space. The models within Approxy export for verification the entire set of input and output values possible to verify behavior for a single multiplier.

Matrix representations are shown in Equation 3.3, with the left most matrix representing the verification data for an ideal multiplier, the middle representing an approximate multiplier, and the right matrix showing a model for testing the multiply-accumulate(MAC) operation.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 0 \\ \vdots & \vdots & \vdots \\ 2^n-1 & 2^n-1 & (2^n-1)^2 \end{bmatrix}, \begin{bmatrix} 0 & 0 & c_0 \\ 0 & 1 & c_1 \\ 0 & 2 & c_2 \\ \vdots & \vdots & \vdots \\ 2^n-1 & 2^n-1 & c_{2^{2n-1}} \end{bmatrix}, \begin{bmatrix} \mathbb{R}_{>0} & \mathbb{R}_{>0} & c(0) \\ \mathbb{R}_{>0} & \mathbb{R}_{>0} & c(1) \\ \mathbb{R}_{>0} & \mathbb{R}_{>0} & c(2) \\ \vdots & \vdots & \vdots \\ \mathbb{R}_{>0} & \mathbb{R}_{>0} & c(N) \end{bmatrix} \tag{3.3}$$

While for an n-bit multiplier, 2^{2n} evaluations are needed to fully compare and verify the Go model to the VHDL file, this is not the case for the MAC unit. It would not be computational efficient to exhaust all possible input and output combinations of the MAC. Since the MAC unit is an extension of an existing (approximate) multiplier, the multiplier can be separately verified using the non-sequential verification model. The sequential evaluation model can be used to verify the accumulative behavior. The MAC model verification within 'Approxy' keeps the inputs constant for the highest output value, e.g. $A = B = 2^n - 1$ and extend the clock cycles N beyond what the MAC-unit is designed for. By extending it, without resetting the MAC, the expected accumulation and overflow behavior within the model can be verified.

The template for the benchmark asserts if the expected result and the actual result are the same. If this is not the case, an error will be exported to the XSIM log file. Approxy provides functionality to parse this log file to be able to handle this fault during runtime to prevent further program execution and further Synthesis for a faulty model. If this fault does not happen, this means that the Approxy model accurately describes the embedded VHDL template behaviorally, and the program can safely continue.

3.5 Synthesis Design-Flow

Approxy provides a VivadoSetting type to be able to set various parameters to adjust the Vivado project flow as seen in Listing 3.4. This in combination with the TCL template, automatically generates a script for Vivado to execute a project from Synthesis, Place and Route to generating reports for analysis. These reports show the utilization of the FPGA primitives and the critical timing path. The various booleans within the section simply turn on and off features. For instance setting the 'NO_DSP' parameter to 'True', forces Vivado to not use any DSP Slices and places the combinational logic fully on Look-up tables, multiplexers and carry adders. The 'OOC' parameter, synthesizes the multiplier(s) out-of-context. This removes IO buffers in the design, thus not connecting any input and output ports to the physical pins of the FPGA chip. While this might not always be the most realistic way to compare VHDL designs, FPGA pins are limited, giving the opportunity for 'OOC'-designs to be bigger without having to introduce overhead to handle the IO connectivity. The other booleans simply enable or disable various steps in the project flow.

In Listing 3.5, an extensive example of such a project script is seen. This example links the design to a certain FPGA part number, after which it synthesizes, places, and routes the design. For functional analysis, a new VHDL file is created by Vivado, the CARRY primitive count is exported, and various Vivado reports are written to a file.

```
type VivadoTCLSettings struct {
    PartName          string
    OOC               bool
    NO_DSP            bool
    WriteCheckpoint   bool
    Placement         bool
    Route             bool
    Funcsim           bool
    Utilization       bool
    Hierarchical      bool
    Clk               bool
    Timing            bool
}
```

Listing 3.4: Approxy Vivado Settings

```
1  link_design -part Xc7z030fbg676-3
2  read_vhdl [glob *.vhd]
3  synth_design -mode out_of_context -max_dsp 0 -top
       recacc8_0_scaler
4
5  write_checkpoint -force recacc8_0_scaler_postsynth.dcp
6  place_design
7  route_design
8  write_vhdl -mode funcsim recacc8_0_scaler_funcsim.vhd
9  write_checkpoint -force recacc8_0_scaler_postplace.dcp
10 report_utilization -hierarchical -file
       recacc8_0_scaler_post_place_ult.rpt
11 set fo [open recacc8_0_scaler_primitive.rpt a]
12 puts $fo [llength [get_cells -hier -filter {PRIMITIVE_GROUP ==
       CARRY}]]
13 close $fo
14
15 report_timing -nworst 1 -path_type end -file
       recacc8_0_scaler_post_place_time.rpt
16
17 close_project
```

Listing 3.5: Example of a generated Project File

3.6 Analysis

Analysis within Approxy can be divided into static and dynamic analysis. After the Synthesis/P+R stage, all reports relating to primitive usage, area usage and timing are available. Approxy provides functionality to be able to parse these static reports. This is not possible for the power measurement, which requires functional analysis post P+R. This means that a simulation of the exported 'funcsim' VHDL file has to be performed after placement and route.

Given that in cases such as scaled VHDLEntityMultiplier interfaces, the test bench from the behavioral analysis cannot be directly used for the functional analysis due to the automatic unrolling of user-provided VHDL array types, Approxy provides extra functionality on top of the Go template library to be able to use the same template file. Functionally this is similar to the #ifdef preprocessor in C.

To be able to simulate real-life scenarios, the test-file from the validation-stage can be regenerated to provide a distribution of input values for the simulation, instead of an exhaustive analysis. Approxy can generate a test-

file consisting of i random values with either a Uniform or a Normal distribution. These values are used to simulate the multiplier using the testbench within XSIM, and a SAIF (Switching Activity Interchange Format) file is generated. The earlier created project checkpoint is opened, and a Vivado power report is generated using the SAIF file.

After the static and dynamic analysis, all reports are parsed, and a summary is exported in the JSON format for easy parsing and graphing in MATLAB/Anaconda. An example is shown in Listing 3.6. This single report shows an accurate 8-bit multiplier that is linearly scaled($N = 500$) using the Scaling interface, $i = 1000$ input values are used in post-placement analysis to determine dynamic power consumption. The JSON format does not contain units, but uses similar units as Vivado reports use. Power is in W, timing in ns and frequency in MHz

To compare the influence of i on the reported power consumption, an Approxy Run is started to report the power consumption of an accurate 8-bit multiplier 50 times, using new random normal distribution input values each time, with $N = 500$ for $i = \{10, 100, 1000\}$. This results in a run exported to a JSON file, consisting of 50 reports. The reports are parsed externally in a Python script. A probability density graph of the normal distribution of the results are shown in Figure 3.2. By increasing the amount of input values, the mean shifts but the standard deviation gets smaller. Out of this data can be concluded that while this analysis cannot directly say anything about real-life accuracy in terms of power consumption, for high enough value i, the precision of the measurements is high enough for comparative analysis as long as N, i and the bit width stays the same. For instance, $N = 500, i = 1000$ for the 8-bit multiplier has a standard deviation of $\approx 5.7\mu W$. Making the results fall 95% of the time fall in the range of $\mu \pm 11.4\ \mu W$.

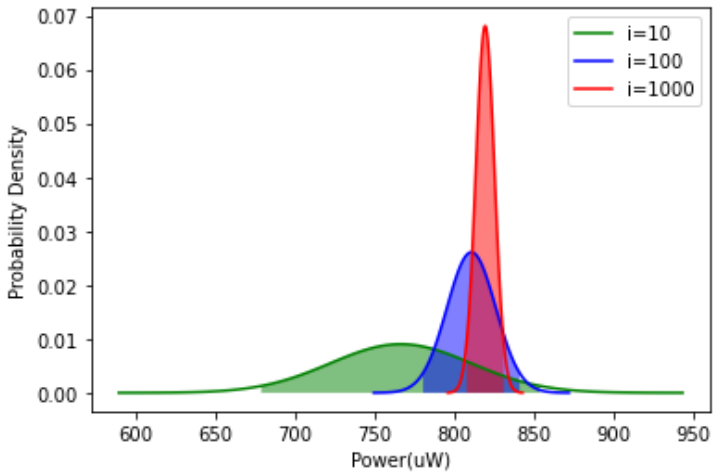


Figure 3.2: Probability Density, Normal Distribution of 50 times N=500 8-bit accurate multipliers

```
{
    "Name": "ErrorRun_500_1000",
    "Reports": [
    {
        "EntityName": "Run0",
        "Util": {
            "TotalLUT": 35000,
            "LogicLUT": 35000,
            "LUTRAMs": 0,
            "SRLs": 0,
            "FFs": 0,
            "RAMB36": 0,
            "RAMB18": 0,
            "DSP": 0,
            "CARRY": 5000
        },
        "Power": {
            "Total_Power": 0.408,
            "Dynamic_Power": 0.286,
            "Static_Power": 0.122,
            "Confidence_Level": "High"
        },
        "Timing": {
            "EndPoint": "prod[71][15]",
            "WorstPath": 4.3,
            "MaxFreq": 232.5581395348837
        },
        "Other": [
        {
            "Key": "Error",
            "Value": "0"
        },
        {
            "Key": "ElapsedTime",
            "Value": "16m7.081299435s"
        }
        ]
    }
    ],
    "Other": [
    {
        "Key": "Disc",
        "Value": "Running 500 accurate 8-bit Multipliers to determine power error, i=1000"
    }
    ]
}
```

Listing 3.6: Example of an Approxy run consisting of a single report.

3.7 Case Study

A code example that shows the full workflow of Approxy can be seen in Listing A.1, as part of Appendix A.1. Here a Recursive Multiplier $\{M_1; M_2; M_3; M_4\}$, as described in Section 4.4, is investigated. The environment used is Xilinx Vivado "v.2021.1 (lin64) Build 3247384" and Vivado Simulator v2021.1 (XSIM). The device running Approxy and Vivado is a desktop running Manjaro Linux, Linux Kernel: Linux 5.15.65-1-MANJARO, using an "Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz (June 2013)" and 8 GB of RAM. The FPGA targeted is the Xilinx Kintex-7 FPGA: Xc7z030fbg676−3.

First a "Run" is generated and cleared. This refers to the generation of a report JSON file to the "ReportPath". Output data is expected to be in the "OutputPath".

A 4-bit Recursive Multiplier object is created on basis of $\{M_1; M_2; M_3; M_4\}$, after which validation test-data and a VHDL file for the Recursive Multiplier are generated and exported to the "OutputPath". This latter function also generates all individual VHDL files for the behavioral models of M_1, M_2, M_3 and M_4.

Next step is the optional verification, a testbench named "prePR" is created using the template for a standard multiplier. This testbench is executed using 'XSIM' and parsed. If any validation errors are seen, the execution of the run is terminated and logged to the command-line.

A scaler object is created to be able to synthesize multiple of these multipliers. N is set to 1000. New VHDL and test data is exported, after which "main.tcl" is created. On basis of the global "VivadoSettings", a TCL is exported to synthesize, place and route the Scaler Object using Vivado. This TCL is executed.

After P+R, a new test bench object is created, named "postPR" for the Scaler Object. The template needs to be set to the one for this Scaling object, and a special PostPR testbench has to be created using the function "CreateFile". $i = 1000$ Normal Distributed values are created and exported after which an XSIM Post-Placement simulation is run. "PowerPostPlacementGeneration" opens the project again in Vivado, uses simulation data from the simulation to generate the required data for System Requirement analysis as explained in Section 2.4. A report is created, which includes timing, resource and power data. Manually, the Mean Absolute Error for the

Uniform distribution and the Normal distribution is added, next to a boolean marker to show if the multiplier internally overflows. These error metrics are not normalized, this needs to be processed afterwards. The report is added to the "Run".

In Listing 3.7, the final resulting report can be seen. The resulting data is divided between a section "Util", listing Primitive usage, a section "Power" listing all power statistics for the FPGA and a summary listing "Timing" details. All manually added statistics are "Key/Value" combinations, listed under "Other". An analysis, showing the execution time of the Approxy "Run" is shown in Table 3.1. Here can be seen that Approxy adds 5% overhead to the workflow, and most execution time is due to processing power needed for Vivado and XSIM.

Time	**Approxy**	**Vivado/XSIM**	**Time**
Start till Analysis	**x**		0.003s
Analysis		**x**	36s
Analysis till SynthPR	**x**		0.004s
SynthPR		**x**	178s
SynthPR till Func	**x**		0.046s
FunctionAnalysis		**x**	282s
Report Generation	**x**	**x**	25s
			8m41s or 521.053s

Table 3.1: Time Overview of Approxy workflow for 4-bit Rec4, N=1000, i=1000

```
{
    "Name": "Rec_1234",
    "Reports": [
    {
        "EntityName": "Rec1234_scaler",
        "Util": {
            "TotalLUT": 22659,
            "LogicLUT": 22659,
            "LUTRAMs": 0,
            "SRLs": 0,
            "FFs": 0,
            "RAMB36": 0,
            "RAMB18": 0,
            "DSP": 0,
            "CARRY": 0
        },
        "Power": {
            "Total_Power": 0.162,
            "Dynamic_Power": 0.041,
            "Static_Power": 0.121,
            "Confidence_Level": "High"
        },
        "Timing": {
            "EndPoint": "prod[444][2]",
```

```
                    "WorstPath": 1.698,
                    "MaxFreq": 588.9281507656065
                },
                "Other": [
                {
                    "Key": "MAE_Uniform",
                    "Value": "3.09375E+00"
                },
                {
                    "Key": "MAE_Normal_1000",
                    "Value": "1.4000000000000001E+00"
                },
                {
                    "Key": "Overflow",
                    "Value": "false"
                }
                ]
            }
            ],
            "Other": null
        }
```


Listing 3.7: JSON Report for Rec1234_scaler

Chapter 4

Implementation of Multiplier Models

In this chapter various VHDL and Approxy models are shown, explained and compared on the FPGA for accurate and approximate multiplier design. First, in Section 4.1, the RTL design of multipliers is compared to see what factors change the size of the multipliers. When describing these multipliers, a design approach is used where instead of describing multipliers by their behavior, the synthesis step is essentially bypassed by describing the design on the register-transfer level(RTL). Secondly a numeric benchmark is shown in Section 4.2 to compare future approximate models to. These multipliers are called 'Numeric' within this thesis, because it makes usage of the IEEE Numeric VHDL library. Multipliers are simply described using the multiplication operator, so the tooling has full control over the implementation of the multiplier. Afterwards in Section 4.3, behavioral models are introduced that cover some smaller 2-bit approximate designs. What makes these models 'behavioral', is that the VHDL only describes the expected output values without describing any structure to the digital design. As long as the expected behavior holds, the tooling is free to implement the design in any way. Finally, Recursive Multipliers are looked into in Section 4.4 and extensively investigated. Here architectural structure is added to the designs.

4.1 RTL Multiplier Design

To create smaller multipliers, and thus by a some extents create more power efficient multipliers, it might be helpful to look at the building blocks of the FPGA to see how to make smaller multipliers from a bottom-up approach. In Figure 4.1, the most fine-grain building block of a modern Xilinx FPGA

is shown, the 'LUT6_2'. This two-output LUT has six inputs of which five are shared. This results in the ability to "act as a dual asynchronous 32-bit ROM (with 5-bit addressing), implementing any two 5-input logic functions with shared inputs, or implementing a 6-input logic function and a 5-input logic function with shared inputs and shared logic values."[28] Extending combinational logic beyond the capability of the LUT means that the synthesis tools has to place the logic on extra LUT units, while less complicated combinational logic may not show improvements in terms of LUT usage.

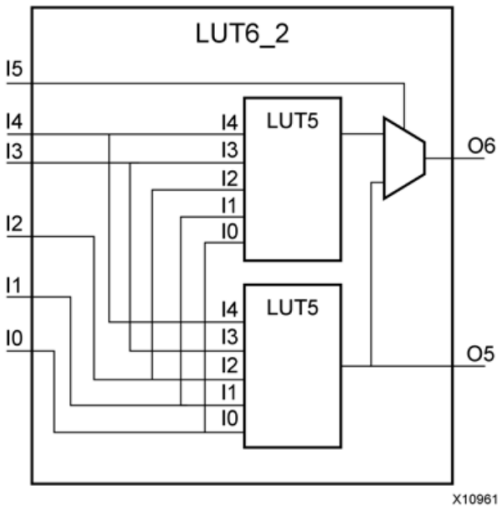


Figure 4.1: LUT6_2 [28]

An interesting example to inspect are the 2-bit multiplier and two approximate designs (M1[16] and M2[29]), as seen in Figure 4.2. Both seen by Equation 4.1 and the RTL drawing, there are four outputs. A two bit multiplier being only dependent on four independent inputs, means that due to the design of the LUT6_2, one multiplier can be placed upon two of these LUTs. A single modification ($3 * 3 \implies 7$) to this multiplier, creates M1 in Equation 4.2. This reduces the amount of boolean functions, and thus the output ports to three. A possible design placement would be to have o_0 and o_1 on a single LUT, whereas the second LUT is shared by the o_2 of two M1 multipliers. This effectively reduces the theoretical minimum LUTs per multiplier from 2 to 1.5.

Another different version, M2, designed for an ASIC using Synopsis on a

TSMC 45nm process in the paper by Rehman et al. shows an area improvement of 28.21%[29] compared to an accurate design. This design also shows a reduction in the amount of boolean functions; o_3 can simply be directly routed to o_0. However, the same placement approach as M1 is here not applicable. Boolean function o_2 is dependent on four variables, meaning that for two of these functions on one LUT, eight independent inputs would be needed. M2 still has a minimum LUT usage of two per multiplier due to the coarser granularity possible on an FPGA compared to an ASIC. This design in itself shows no theoretical improvement in terms of LUT usage to the accurate multiplier, while introducing an error.

Using the Xilinx UNISIM Library[28] it is possible to design multiplier architectures using the building blocks of an FPGA. This however bypasses the algorithms used by the synthesizing tool, thus extra care has to be taken to not erroneously under design a multiplier. If for example the designer assigns two LUT6_2 structures to one M1 multiplier, one reaches a LUT usage of 2 per multiplier: the synthesizer will not merge partly unused look-up tables if not explicitly stated in the VHDL or Verilog code.

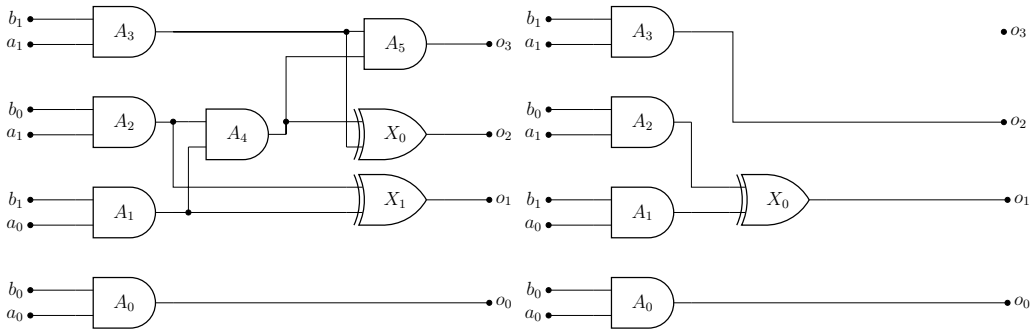
Furthermore, an issue is that using the Xilinx UNISIM Library creates vendor lock-in. Designs that are centered around using these 'primitives' might not show the hoped-for improvements on a different or future line of products, even less so at direct competitors. For example, the comparable logic unit like the 'LUT6_2' on the Intel Stratix V[30], called the 'ALM', has the possibility to compute two 4-input boolean functions meaning that M2 would expectantly show an improvement in terms of area usage that M2 on the Xilinx FPGA does not.

Because of these reasons these models are not implemented using 'Approxy'. There are however some papers that choose this approach ([19], [31], [32]) to essentially create an approximate multiplier by hand and get favorable results for their specific Xilinx architecture.

$$Acc_{n=2}\begin{cases} o_0 = a_0 \wedge b_0 = f(a_0, b_0) \\ o_1 = (a_0 \wedge b_1) \veebar (a_1 \wedge b_0) = f(a_0, b_0, a_1, b_1) \\ o_2 = ((a_0 \wedge b_1) \wedge (a_1 \wedge b_0)) \veebar (a_1 \wedge b_1) = f(a_0, b_0, a_1, b_1) \\ o_3 = ((a_0 \wedge b_1) \wedge (a_1 \wedge b_0)) \wedge (a_1 \wedge b_1) = f(a_0, b_0, a_1, b_1) \end{cases} \tag{4.1}$$

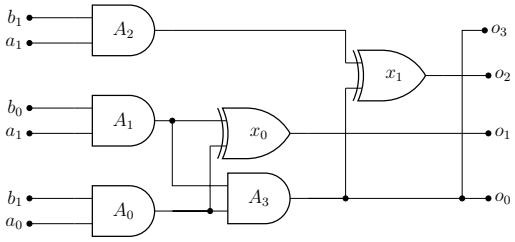
$$M1 \begin{cases} o_0 = a_0 \wedge b_0 = f(a_0, b_0) \\ o_1 = (a_0 \wedge b_1) \veebar (a_1 \wedge b_0) = f(a_0, b_0, a_1, b_1) \\ o_2 = a_1 \wedge b_1 = f(a_1, b_1) \\ o_3 = 0 \end{cases} \tag{4.2}$$

$$M2 \begin{cases} o_0 = (a_0 \wedge b_1) \wedge (a_1 \wedge b_0) = f(a_0, a_1, b_0, b_1) \\ o_1 = (a_0 \wedge b_1) \veebar (a_1 \wedge b_0) = f(a_0, a_1, b_0, b_1) \\ o_2 = ((a_0 \wedge b_1) \wedge (a_1 \wedge b_0)) \veebar (a_1 \wedge b_1) = f(a_0, a_1, b_0, b_1) \\ o_3 = o_0 \end{cases} \tag{4.3}$$



(a) Accurate 2-Bit Multiplier

(b) Approximate M1 2-Bit Multiplier



(c) Approximate M2 2-Bit Multiplier

Figure 4.2: RTL Design Comparison

4.2 Numeric Multipliers

Numeric Multipliers are based upon the availability within the IEEE VHDL specification to be able to describe arithmetic using mathematical operators, in this case the '*'. These are naturally the simplest way to describe since it

creates an abstraction where only the multiplication of typed integer numbers is apparent: the inner functionality of the multiplication algorithm is abstracted away. Its biggest limitation is that it cannot be used to describe approximate multiplication. However, a positive outcome of this model is that it gives full control to the synthesis tool on how to synthesize the multiplication, given that the design is not constricted in any way by the designer. This makes it a great benchmark to compare approximate models to in terms of area, delay and power usage.

These multipliers can also be designed on non-LUT based architecture. For instance, the Xilinx UltraScale architecture provides DSP Slices[33], that are ASIC structures within the FPGA architecture that can be used to compute certain (high-width) functionality often used in DSP applications. The functionality of these slices can differ between product line and vendor, but in the case of the DSP48E2 Slice, as shown in Figure 4.3, it is perfectly capable of implementing Multiplier/Multiply-Accumulate structures.

To see the effect of DSP slices on Multiplier design, n-bit sized multipliers are synthesized out-of-context on a Xilinx Kintex-7 FPGA using Vivado 2021.1. In Figure 4.4, the area usage in terms of look-up tables is compared on the same device by either giving full control to the synthesis tool(4.4b), or by disallowing the usage of DSP slices(4.4a). From the results, it can be seen that the extent of area reduction due to the introduction of DSP Slices especially gets prominent when investigating higher width multipliers. Till $n = 10$, the synthesis tool does not use any DSP slices. From $n = 19$ on, LUT instances are again being placed due to the width of a single Xilinx DSP slice not being sufficient anymore. More interestingly, the LUT-based multipliers do not simply see a continuous increase in LUT usage when increasing the multiplier width, increasing the size at various instances causes either stagnation or even reduction in terms of LUT area. This shows that the synthesis tool also considers the usage of other primitives, such as CARRY4, in the synthesis of multipliers and thus can have a considerable effect on the LUT usage.

4.3 Behavioral Multipliers

A very basic multiplier model for the realization of approximate multipliers is made by modelling the multiplier as a simple two dimension integer slice in Go. The model for a 4x4-size array is shown in Table 4.1. Over this model it is possible to superimpose the possible output values of an accurate 2-bit

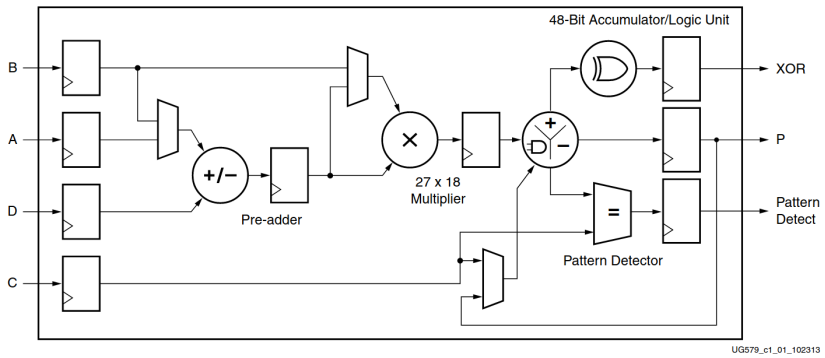
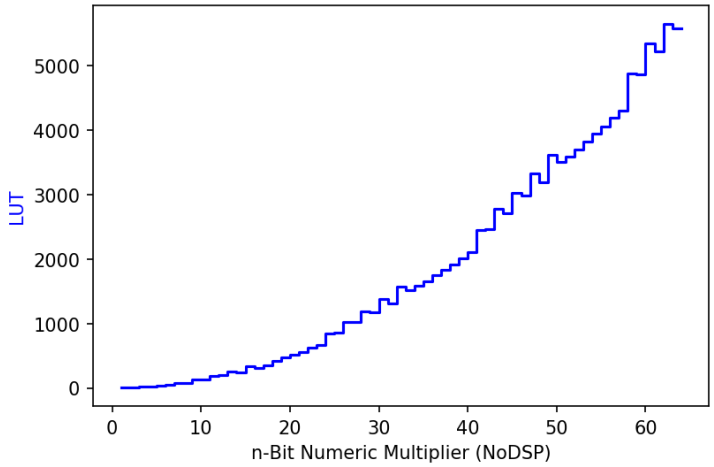
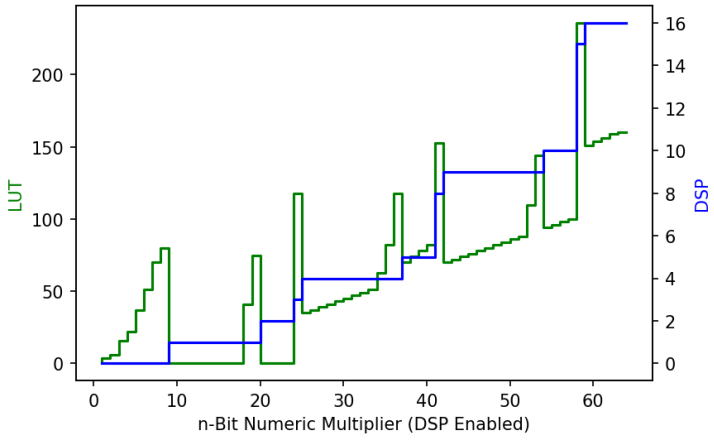


Figure 4.3: Xilinx UltraScale DSP48E2 Slice [33]



(a) Disabling DSP Slices



(b) Using DSP Slices

Figure 4.4: Area usage of Accurate Numeric Multipliers

multiplier, as shown in Table 4.2.

In the accompanying VHDL template the choice has been made to implement this model with a Process statement to describe the combinational behavior of the multiplier as seen in Listing 4.1. Here the templating model of Go is extensively used to unroll the two-dimensional unsigned integer array field LUT into a nested VHDL Case structure, where Case A responds to the array row, and Case B to its column. Functions such as indexconv and valconv are used within the templating pipeline to provide a conversion between Go's decimal integers and VHDL's STD_LOGIC_VECTOR.

```
1  process(A,B) is
2  begin
3      case A is
4          {{- range $rowindex, $row := .LUT}}
5          when "{{$rowindex | indexconv}}" =>
6          case B is
7              {{- range $columnindex, $val := $row}}
8              when "{{$columnindex | indexconv}}" =>
9                  prod <= "{{$val | valconv}}";
10             {{- end}}
11             when others =>
12                 prod <= (others => 'X');
13         end case;
14         {{- end}}
15         when others =>
16             prod <= (others => 'X');
17     end case;
18 end process;
```

Listing 4.1: Process Statement of the behavioral Multiplier Template

Since the resulting behavioral model in VHDL does not use the multiplication operator, the synthesis tool might not optimize the design correctly, thus the resulting overhead in terms of LUT usage is compared to the Numeric Multiplier of the previous section. In Figure 4.5 the results of this comparison can be seen. While for a 2-bit multiplier there is no difference between either accurate models, a slight increase is seen for $n = 4, 16 \rightarrow 22$. For 8-bit multipliers, the behavioral model is completely useless, given the almost factor 100 increase of used LUT primitives. This behavioral model is therefore not applicable beyond 2-bit multipliers.

In Table 4.7, four approximate 2-bit multipliers are shown that are further investigated. The approximate multipliers are synthesized on the Xilinx Kintex-7 FPGA, using part number "Xc7z030fbg676-3", and compared to a

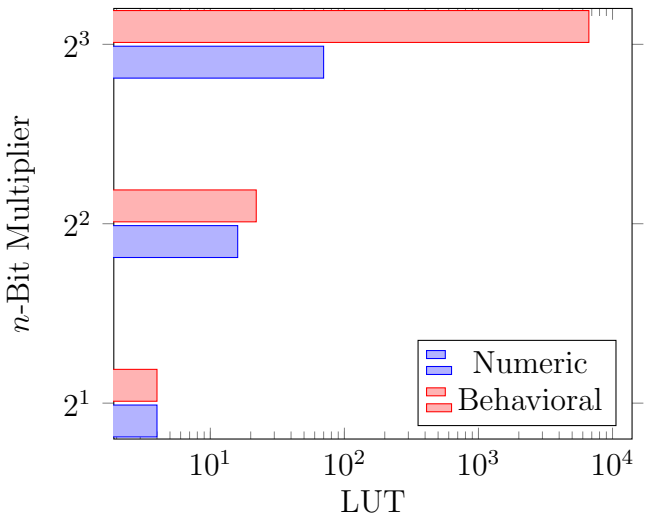


Figure 4.5: Comparison Accurate Multiplier Models

behavioral accurate multiplier. As seen in Table 4.8 the behavioral model for an accurate 2-bit multiplier performs at the theoretical smallest area limit described in Section 4.1 of 2 LUT/m.

This particular model of the Kintex-7 FPGA has 78600 available LUTs. For these measurements a total LUT-usage of 80% is taken as a baseline to account for any issues that may arise due to possible lax optimizing for designs that barely occupy any area. The multipliers are simply linearly scaled, Out-of-Context, using the following equation:

$$N = 0.8 * \frac{LUT_{Part}}{LUT_{Mult}} = 0.8 * \frac{78600}{2} = 31440 \tag{4.4}$$

As predicted earlier, M2 does not show any improvements in terms of LUT area. This is also the case for M3. M1 and M4 only show a slight overhead compared to their theoretical RTL design equivalents of 1.73 percentage points.

	Col0	**Col1**	**Col2**	**Col3**
Row0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
Row1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
Row2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
Row3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Table 4.1: 2D Array Model

	0	**1**	**2**	**3**
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	9

Table 4.2: Accurate 2-bit Multiplier

	0	**1**	**2**	**3**
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	7

Table 4.3: M1 [16]

	0	**1**	**2**	**3**
0	0	0	0	0
1	0	0	2	2
2	0	2	4	6
3	0	2	6	9

Table 4.4: M2 [29]

	0	**1**	**2**	**3**
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	11

Table 4.5: M3 [34]

	0	**1**	**2**	**3**
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	5

Table 4.6: M4 [18]

Table 4.7: Behavioural Models for 2-bit Multipliers

	LUT	**LUT(%)**	**LUT/m**	**ERate**	**EMag**	**MAE (Uniform)**
Acc	62880	80	2	0	0	0
M1	48518	61.73	1.54	1/16	2	0.125
M2	62880	80	2	3/16	1	0.1875
M3	62880	80	2	1/16	2	0.125
M4	48460	61.65	1.54	1/16	4	0.25

Table 4.8: Results for the behavioral Multipliers, N=31440,

4.4 Recursive Multipliers

By shifting the output of various smaller sized multipliers and adding the results, a larger sized multiplier can be created[29]. An overview is shown in Figure 4.6. When multiplying A with B, with bit-size $2W$, the input values are split in A_H, A_L, B_H and B_L with size W. The next formula calculates the output of a 4-bit multiplier[18]:

$$C_{n=4} = (A_L * B_L) + 4(A_L * B_H) + 4(A_H * B_L) + 16(A_H * B_H) \quad (4.5)$$

Naturally, the multiplier calculating $A_H B_H$ has the biggest influence on the

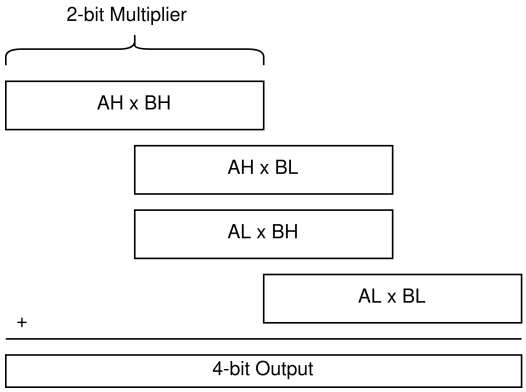


Figure 4.6: Recursive Multipliers

Mean Error, which is also reflective in the formula that determines the total Mean Error:

$$ME_4 = ME_{A_L B_L} + 2^2 ME_{A_L B_H} + 2^2 ME_{A_H B_L} + 2^4 ME_{A_H B_H}[18] \quad (4.6)$$

In the Approxy framework, the Recursive Multiplier is simply defined as a 4-length sized array of Approxy models in the form of VHDLEntityMultiplier interfaces. The VHDL Template is simply a VHDL file that port maps the four multipliers and uses the Numeric IEEE Library to perform addition using the '+' operator.

Overflow handling is performed in the same way as in the MACISH paper by Gillani et al. ([18]) Recursively for every $n/2$-sized multiplier within a recursive multiplier, and the n-sized recursive multiplier itself, the following equation must hold.

$$\forall i \in \{0, ..., 2^n - 1\}, \forall j \in \{0, ..., 2^n - 1\}, f(i, j) < 2^{2n} \quad (4.7)$$

This function is within Approxy only implemented as best-effort by checking if any overflow behavior is detected when a single ReturnVal(uint, uint) is called. However, by-design the whole output space gets calculated for generating the exhaustive test-file. As long as the verification step is not being omitted, the overflow handling can be guaranteed.

4.4.1 Design Space Exploration, 4-bit

For exploring the Recursive Multiplier based upon the results of the behavioral multipliers, the set $\boldsymbol{S} := \{M_1, M_2, M_3, M_4, M_{Acc}\}$ is used to create a 4-bit Recursive Multiplier. By taking the Cartesian Power $\boldsymbol{S}^4$, the entire set of 4-bit Recursive Multipliers using $\boldsymbol{S}$ can be defined. Since the size of $\boldsymbol{S}$ is 5, the size of the design space of the 4-bit Recursive Multiplier using this set equals $5^4 = 625$. This would mean that for an 8-bit Recursive Multiplier, set $\boldsymbol{S}$ would have a size of 625, making the Design Space $\boldsymbol{S}^4$ considerably bigger. The general formula for the size of the design space is the following equation, where m equals the size of the set of initial set $\boldsymbol{S}$, and n equals the size of the recursive multiplier:

$$s = m^{(n/2)^2}[18] \tag{4.8}$$

Given the Time Execution overview of the case study in Table 3.1, an estimate for the full-analysis of a recursive multiplier design ($N = 1000$, $i = 1000$) is 8m41s. A full exploration of all 4-bit multipliers, using the earlier described set, would take an estimate 3.8 days. For 8-bit exploration the estimate is 2.5 million years, making it clearly non-viable.

Both the article by Gillani et al. [18] and the SMApproxLib library[19] use a methodology to substantially reduce the computational power of synthesizing these multipliers by estimating the 'cost' of the recursive multipliers by using the resource information of their smaller $N/2$ building blocks. It has to be noted though, that [18] applies these multipliers to "TSMC 40nm Low Power(TCBN40LP) technology" which behaves differently than FPGA technology. However, the SMApproxLib article summarizes their FPGA LUT usage similarly:

$$LUT^{N*N} = \sum_{i=1}^{4} LUT_i^{\frac{N}{2}*\frac{N}{2}} + LUT^{Adder}[19] \tag{4.9}$$

In Section 4.1 it was already discussed that Vivado does not optimize RTL designs and treats these VHDL entities as 'blackboxes', making this additive LUT formula hold for RTL designs like SMApproxLib. When synthesizing behavioral models and flattening their hierarchy, however, recursive

blocks can be optimized by the synthesis tooling and placed upon shared resources transforming the formula into a higher-bound estimate instead. This makes optimizing the design space exploration for recursive multipliers on the FPGA, by selecting on these functions, quite risky given unexpected synthesis optimizations might go undetected.

Given the extent of the 8-bit multiplier design space, only the 4-bit space is fully explored. The following general notation is used to define the 4-bit recursive multiplier from here on:

$$\boldsymbol{R}_n := \{A_H B_H; A_H B_L; A_L B_H; A_L B_L\} \tag{4.10}$$

An example of such a multiplier is:

$$\boldsymbol{R}_{Acc4} := \{M_{Acc}; M_{Acc}; M_{Acc}; M_{Acc}\} \tag{4.11}$$

This is a 4-bit Recursive Multiplier constructed using four 2-bit Accurate behavioral Multipliers. In the previous section it has been already shown that there is no significant difference in terms of LUT usage between behavioral and numerical 2-bit multipliers. Synthesizing the Accurate 4-bit Recursive Multiplier $\{M_{Acc}; M_{Acc}; M_{Acc}; M_{Acc}\}$ shows an average LUT usage per multiplier of 16.963. While this is higher than the 16 of the numerical 4-bit multiplier, the overhead of this model is quite low. Comparing this to the purely behavioral 4-bit model shown in Section 4.3 , which showed a LUT usage of 22, the difference is clear. Adding the recursive structure to the design which uses four 2-bit behavioral multipliers, outperforms a 4-bit design that is only described behaviorally.

4.4.2 Results

Using Vivado 2021.1 for a Xilinx Kintex-7 FPGA "Xc7z030fbg676-3", an Approxy Run is created for the whole 4-bit Recursive Multiplier Design Space using the 2-bit behavioral multiplier set $\boldsymbol{S} := \{M_1, M_2, M_3, M_4, M_{Acc}\}$. As earlier described, every permutation where repetition is possible is investigated by creating the fourth Cartesian power, $\boldsymbol{S}^4$. All individual multipliers are scaled with $N = 1000$. The post-placement simulation processes $i = 1000$ normal distributed ($\mu = 8, \sigma = 1.5$) input values on a clock speed of 50MHz. These means that all power simulations use a normally distributed input vector. Additive results like LUT usage and power consumption are again divided by $N = 1000$ to get the results per multiplier. The Mean Absolute Error is normalized. Unless stated otherwise, the Mean Absolute Error has the same normal distribution as used in the post-placement simulation.

Similar to the analysis of the influence of i and N on the power consumption results in Section 3.6, a run is created consisting of 50 reports for the benchmark 4-bit numeric multiplier. The results of this run is: $\mu = 172.26\mu W, \sigma = 0.795\mu W$. Showing a small but insignificant possible deviation of the following power results.

Figure 4.7 shows the box plots of the Mean Absolute Error of the set. The distribution of the input values is uniform. The first distinction is made for the most significant partial product $A_H B_H$, each showing a different box plot. The color of the dots represent the second most significant partial product $A_H B_L$. Every single dot represent a single configuration of $\boldsymbol{S}^4$. As expected, the lowest MAE is seen for configurations that use an accurate multiplier for the $A_H B_H$ partial product. Similarly to the behavioral models, M_4 shows the highest mean error, after M_2. M_3 and M_1 are comparable. In Figure

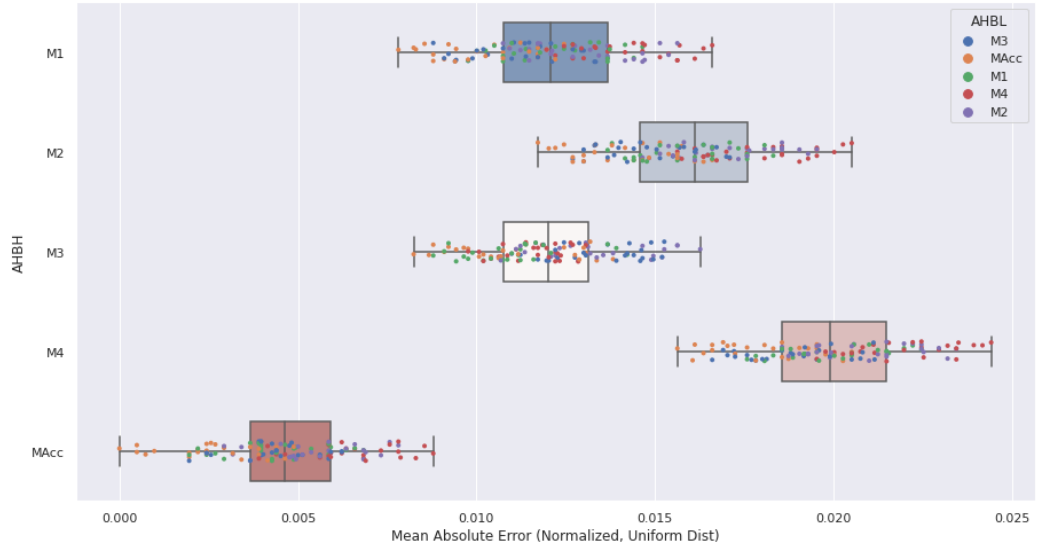


Figure 4.7: Comparison of Normalized Uniform Mean Absolute Error of 4 x 4 Recursive Multipliers on basis of most-significant partial product.

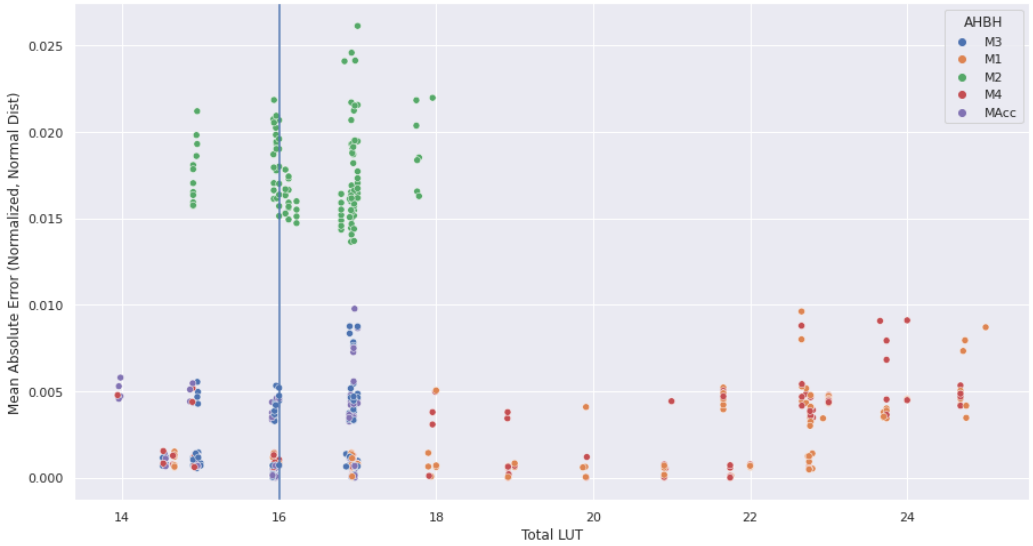


Figure 4.8: Quality-Area trade-off of 4 x 4 Recursive Multipliers

4.8 the LUT-MAE trade-off is shown for the entire set. Using different hues, the configurations are distinguished by their most significant partial product $A_H B_H$. The vertical line shown is the LUT usage of the numeric accurate multiplier. Again seen in this graph, the lowest MAE is seen for configurations that use an accurate multiplier for the $A_H B_H$ partial product, and almost fully dominate the Pareto Frontier. However a bigger spread in terms of LUT usage is seen for configurations $A_H B_H = M_1 \vee M_4$. When for instance looking at the biggest (25 LUT/m) configuration $\{M_1; M_2, M_2, M_2\}$, the Vivado Synthesis issues a relatively high amount of LUT6 primitives, which cannot share a LUT6_2 placement. To limit the amount of points in the Pareto Frontier, the LUT/m is rounded off to one decimal digit, and shown in Table 4.9. The smallest configuration in terms of LUT area is 13.9, which is a 13.125% decrease compared to the numerical benchmark. An argument could as well be made for the second multiplier in the table. While barely increasing in size, the normal MAE is a bit lower. Furthermore, its uniform MAE is significantly lower, making it more versatile for different use cases.

In Figure 4.9 the Power-MAE trade-off is shown for the set. It has to be noted that for the whole set the Static Power consumption is 121-122 μW. While this dominates the total power consumption, differences between static power consumption are negligible to non-existent. This makes the actual differences between total power consumption fully dominated by the differences in dynamic power consumption. Furthermore, no direct positive correlation

is seen between LUT area and power consumption of the FPGA; a lower LUT/m does not necessarily mean lower power consumption given the shift of the M_{Acc} configurations. This is further substantiated in the Pearson correlation coefficient heat map of Figure 4.11, showing even a negative correlation between LUT area and Power consumption for this run. In the Quality-Power trade-off, configurations with $A_H B_H = M_1 \vee M_4$ are showing both the highest and lowest power consumption of the set. Other configurations mostly perform worse than the numeric benchmark. The Pareto Points are shown in Table 4.10. The point with the lowest power consumption shows a 25% decrease compared to the numeric benchmark.

Figure 4.9: Quality-Power trade-off of 4 x 4 Recursive Multipliers

The worst paths delay in the designs are compared, otherwise called the critical path, which results in the maximum frequency $(T)^{-1}$(Hz) shown in Figure 4.10. Similar to the Power/MAE trade-off, most spread is seen for the M_1 and M_4 multipliers, where most other multipliers perform worse than the numeric benchmark. The configuration with the highest maximum possible frequency is also the configuration that performs best in terms of power consumption. This design shows a lot of MUX primitive implementations where the select input is triggered by actual input instead of internal signals, which means the capacity of a single slice is efficiently used, and the clocked paths are short. Looking at the Maximum Frequency and the Pearson Correlation Coefficients in Figure 4.11, the biggest correlations are seen. A higher maximum frequency in general means a higher amount of look-up tables but

smaller power consumption.

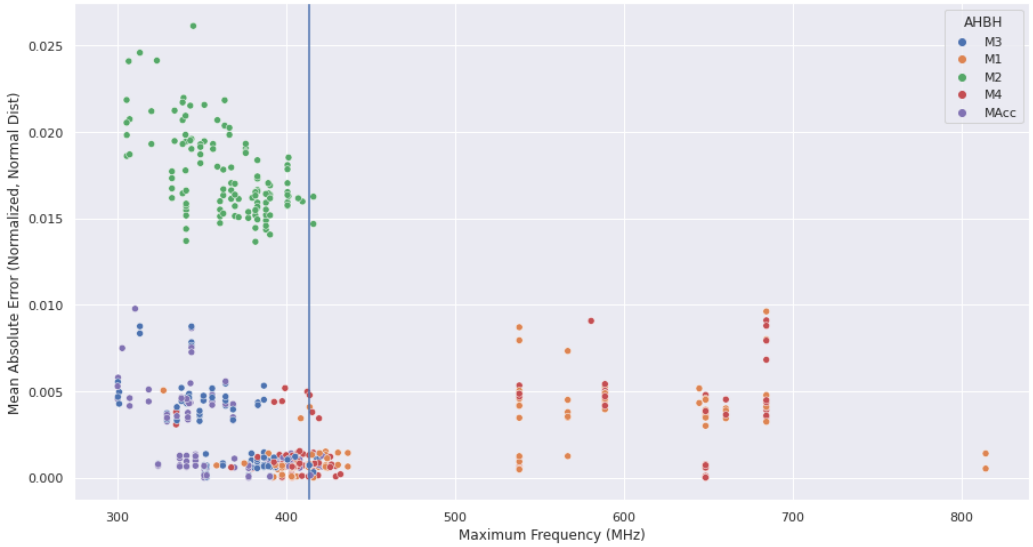


Figure 4.10: Quality-Latency trade-off of 4 x 4 Recursive Multipliers

For some configurations, Vivado decides to use 2 CARRY4 primitives per multiplier to take care of the adder instead of LUT/MUX logic. A slight negative correlation is seen in relation to LUT area, but this does not seem to carry into higher maximum frequencies or lower power consumption.

LUT/m	Power(μW)	MAE (Normal)	Max Freq (MHz)	A_HB_H	A_HB_L	A_LB_H	A_LB_L	CARRY4/m	MAE (Uniform)
13.9	199	4.7656e-3	414.08	M4	M4	M2	M4	0	2.3438e-2
14	184	4.5547e-3	300.39	MAcc	M4	M2	M1	2	7.3242e-3
14.5	174	6.7969e-4	419.82	M3	M4	M4	M1	0	1.2207e-2
14.6	177	6.4844e-4	401.93	MAcc	M1	M4	M1	0	6.3477e-3
14.7	175	5.8594e-4	423.55	M1	M4	M1	M1	0	1.4160e-2
14.9	177	5.8594e-4	379.65	M3	M3	M1	M1	2	1.0986e-2
15	177	5.3906e-4	383.00	M3	MAcc	M1	M1	2	9.2163e-3
15.9	174	0.0000e+0	353.11	MAcc	M1	MAcc	MAcc	2	1.9531e-3

Table 4.9: Pareto Frontier of 4 x 4 set, minimizing LUT/m and MAE

LUT/m	Power(μW)	MAE (Norm)	Max Freq (MHz)	A_HB_H	A_HB_L	A_LB_H	A_LB_L	CARRY4/m	MAE (Uniform)
22.8	129	5.1563e-4	814.33	M1	M3	M1	M1	0	1.1047e-2
21.8	134	9.3800e-5	648.51	M1	M3	M1	MAcc	0	1.0742e-2
21.7	137	6.2500e-5	648.51	M4	M3	M3	MAcc	0	1.7578e-2
21.7	138	0.0000e+0	648.51	M1	MAcc	MAcc	MAcc	0	7.8125e-3

Table 4.10: Pareto Frontier of 4 x 4 set, minimizing Power/m and MAE

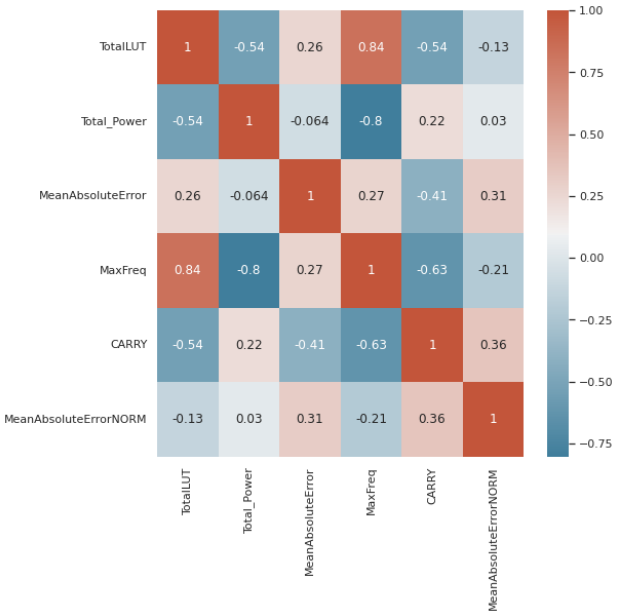


Figure 4.11: Pearson Correlation Coefficient Heatmap of Data features of 4 x 4 Recursive Multiplier set

Chapter 5

Evaluation

In this Chapter, the model to compare multipliers from literature is discussed in Section 5.1. The first library of multipliers discussed is SMApproxLib in Section 5.2, these multipliers are optimized for low LUT resource usage on the FPGA. In Section 5.3, multipliers from other articles and a commonly used benchmark for ASIC designs is investigated. The results of Recursive Multipliers from Section 4.4 are compared to the earlier described multipliers from literature in Section 5.4. Finally, Section 5.5 shows the implementation of the most power-efficient multiplier of the Results.

5.1 Comparing Literature Designs

To compare external results to the earlier explorer multiplier models, an Approxy model called 'External' is created. Similar to other Approxy models, it revolves around a VHDLEntityMultiplier interface. However, given the multiplier comes from an external source it often lacks a behavioral model to validate results similarly than earlier described models. For these models, the verification step in the Approxy flow is altered to build a behavioral model around the provided VHDL file. Instead of using the Approxy model as a baseline for verification data, the verification is in reverse. All possible inputs are simulated within XSIM using the external VHDL file and the results are written to a file. These output values are then parsed and encapsulated within the interface. Finally, an Approxy behavioral model exist that functions the same as all other models, based upon a VHDL model from external source, and the rest of the flow can be exactly executed in the same manner.

5.2 SMApproxLib

In the paper "SMApproxLib: Library of FPGA-based Approximate Multipliers" by Ullah et al. [19] one of the first extensive methodologies is shown to explore approximate multipliers on 6-input LUT FPGA systems. Specifically, in this paper they try to recreate an open-source collection such as the *EvoApprox8b*[17] library. This latter library presents a collection of 8-bit approximate adders and multipliers for design and benchmarking. However, in the paper by Ullah et al. [19] they recognize that the library is not optimized towards FPGA implementation, given that the original library is aimed towards ASIC implementations, and introduce one accurate and three approximate multiplier modules to compare 4-bit and 8-bit approximate multipliers to the ones from *EvoApprox8b*.

The multiplier models described within the paper are based upon RTL structures, comparable to the method of VHDL description as shown in Section 4.1. Instead of giving freedom to the synthesizing tool, they completely structurally design the multiplier using HDL primitives for the Xilinx FPGAs. Using a pre-defined set of three different LUT6_2 configurations and the CARRY4 primitive, they design an FPGA optimized accurate multiplier by taking the general approach of an NxN multiplier seen in Figure 5.1. Compared to Binary long multiplication, as shown in Listing 2.1, consisting of AND gates and adders, it can be seen here that they combine the first two rows of partial products using the 'T-1' LUT6_2 configurations. The last rows use 'T-2' configurations, and 'T-3' to generate P_0 and P_{N+1}.

5.2.1 Approximate Multipliers

On basis of the earlier described accurate multiplier, they describe three proposed approximate designs that focus on the approximate addition of the partial products. Their first proposed alternative (Approx1) is to remove one of the carry chains that performs the addition of partial products and replace it with LUT-based approximate addition. Although the paper describes the INIT values for the LUT6_2 primitives, they unfortunately do not explain in detail how this approximate addition is done. 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. An overview is seen in Figure 5.2 and Figure 5.3. The Approx3 implementation tries to improve upon the accuracy of Approx2 by trying to predict the carry out of the preceding partial product groups.

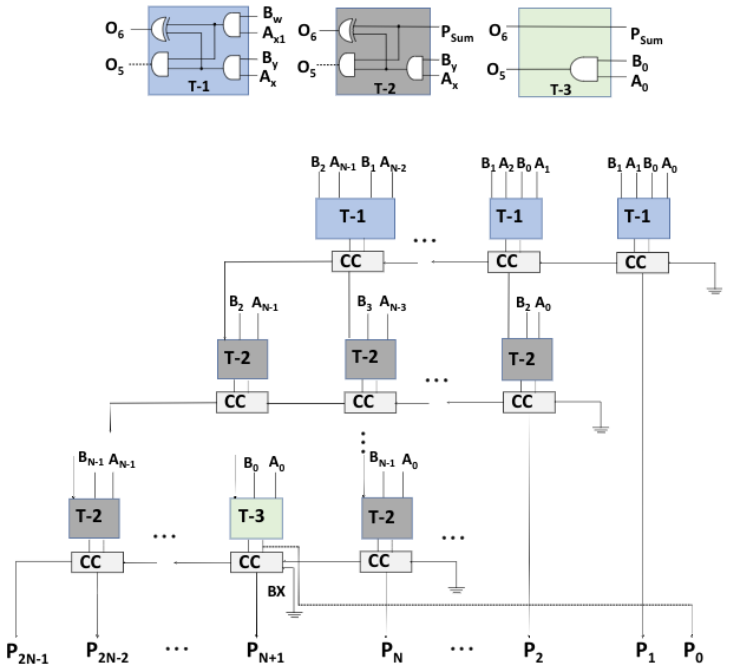


Figure 5.1: Implementation of SMApproxLib Multipliers [19]

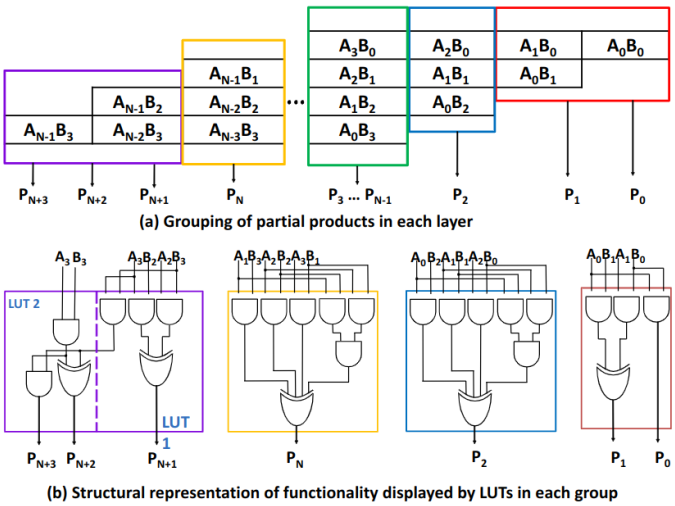


Figure 5.2: Implementation of Approx2 and Approx3 [19]

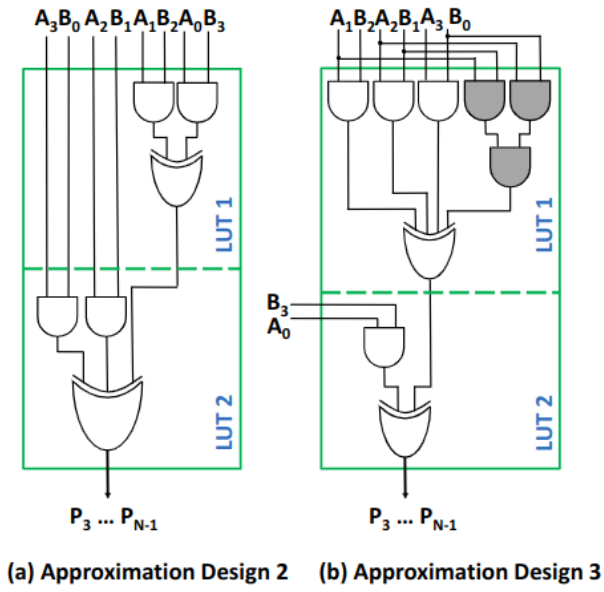


Figure 5.3: Implementation of green 'layer' in Approx2 and Approx3 [19]

5.2.2 SMApproxLib Evaluation

The SMApproxLib multipliers are available as open-source VHDL files consisting of interconnected RTL primitives. An 'External' Approxy model is created for these files, as mentioned in Section 5.1. Similarly to Recursive Multiplier Design Space Exploration, an Approxy Run is created for the SMApproxLib multipliers using Vivado 2021.1 on a Xilinx Kintex-7 FPGA. The multiplier is linearly scaled with $N = 1000$, and post-placement simulation uses the same random number generation as earlier shown results ($i = 1000$, $\mu = 8$, $\sigma = 1.5$). The results of these simulations are shown in Table 5.1.

Comparing the results, it is clear that the SMApproxLib multipliers perform well in terms of LUT usage. This is very clear when looking at the accurate SMApproxLib multiplier. When synthesizing for a Xilinx Kintex-7 FPGA with 78.6K LUTs and 19650 Slices, 6550 instances can fit on the FPGA. The amount of CARRY4 adders needed would be 19650, which means that all slice carry adders are utilized. This is not the case for the numeric multiplier, where only 4912 instances can be initialized resulting in 9824 carry adders.

All these multipliers outperform in terms of LUT/m compared to even the smallest Recursive Multipliers. However, these come at a cost; In the previ-

Name	LUT/m	Power(μW)	MAE (Norm)	Max Freq (MHz)	CARRY4/m	MAE (Uniform)
Acc	12	267	0.0000e+0	319.28	3	0.0000e+0
Approx1	11	268	2.9781e-2	393.24	2	3.1982e-2
Approx2	7	158	3.4031e-2	556.17	0	4.1992e-2
Approx3	8	189	2.6281e-2	547.05	0	4.1992e-2

Table 5.1: Results of SMApproxLib designs[19], 4-bit, analyzed using Approxy

ous chapter, it is already shown that a smaller footprint does not automatically mean lower dynamical power consumption for normal distributed input. Both the Accurate and Approx1 multiplier have a higher power consumption than the highest power consuming Recursive Multiplier $\{M_4; M_{Acc}; M_{Acc}; M_4\}$ of 222 μW. The normalized MAE(Norm) of $\{M_2; M_2; M_2; M_2\}$, the most inaccurate recursive multiplier is ≈ 0.03, which is still less than all the approximate SMApproxLib multipliers.

5.3 lpACLib and Others

Another paper from Ullah et al.[35] before the creation of their SMApproxLib, creates a single 4x4 multiplier design using two approximate 4x2 multipliers. In addition, two carry chains are used for summation. They propose an approximate design where they reduce the CARRY4 primitives to a single one, and optimize their LUT structure by generating carry propagation within. The result is a 4x4 multiplier using 12 LUT6_2 primitives and 1 CARRY4 primitive, with only 6 erroneous outputs with a difference of 8. The normalized MAE(1.0938×10^{-3}) for normally distributed inputs is fairly low. In later results, this multiplier has the Class '4x4' to be able to reference it.

The paper by Rehman et al. [29] is used as a basis of this thesis and works by Gillani et al. [18] as the 'M2' 2-bit multiplier, but this work also introduces it own set of multipliers using a similar approach as this thesis. On basis of various 2-bit multipliers, with the addition of various approximate 1-bit adder designs and using the same recursive multiplier approach, they create a set of various larger sized multipliers. The major difference is that this paper is targeting ASIC 45nm technology instead of FPGAs. The resulting 4-bit multipliers that they publicize in their 'lpACLib' library are therefore Pareto-optimum for this technology and therefore not directly comparable to FPGA technology. The versions of these multipliers that use accurate adders are classified as 'XMAA'. Furthermore they show published configurable multi-

Name	LUT/m	Power(μW)	MAE (Norm)	Max Freq (MHz)	CARRY4/m	MAE (Uniform)
4x4[35]	12	186	1.0938e-3	448.83	1	7.3200e-4
XMAA Accurate[29]	18	176	0.0000e+0	366.57	2	0.0000e+0
XMAA Approx Lit[29]	15	176	5.6250e-4	423.55	0	1.2207e-2
XMAA Approx V1[29]	16	179	2.3566e-2	352.61	2	1.8311e-2
XMAA Config Lit (Acc)[29]	18	177	0.0000e+0	366.57	2	0.0000e+0
XMAA Config V1 (Acc)[29]	18	179	0.0000e+0	349.65	2	0.0000e+0
ConfigXA Lit+Third (Full Approx)[36]	15	175	5.7812e-4	424.45	0	1.8311e-2
ConfigXA Lit+Third (Full Accurate)[36]	22	139	0.0000e+0	648.51	0	0.0000e+0
ConfigXA V1+First (Full Approx)[36]	23	222	2.5262e-2	324.68	0	1.8311e-2
ConfigXA V1+First (Full Accurate)[36]	21	220	0.0000e+0	346.26	0	0.0000e+0

Table 5.2: Results of other designs, 4-bit, analyzed using Approxy

pliers using approximate adders[36] which are classified 'ConfigXA'. Of these configurable multipliers, the individual 2-bit multipliers can be configured between an approximate and accurate configuration. For this comparison, these are configured into a fully accurate design and a fully approximate design. The verified results of these papers[35][29][36] using Approxy are shown in Table 5.2.

5.4 Comparison of Investigated 4-bit Multipliers

This section shows the results of the multipliers from other papers, compared to the Pareto front of the Recursive Multipliers. Table 5.3 shows the Pareto front looking at minimum resource utilization. Table 5.4 shows the Pareto front for the designs with the lowest power consumption.

Figure 5.4 shows the Quality-Area trade-off. While the Recursive Multipliers optimized for LUT-usage show improvement over the numerical benchmark, it is the SMApproxLib that has the best results in terms of low area usage. This comes at a cost however, the three smallest SMApproxLib multipliers have a significantly higher MAE. However, they provide an accurate multiplier that has a smaller footprint than any of the other investigated designs.

In Figure 5.5, the power consumption is compared to the normalized Mean Absolute Error. All 'RecPower' configurations, consisting of the optimized Recursive Multiplier configurations for power consumption here perform favorably and dominate the Pareto front entirely. However, directly after this front is an accurate multiplier 'ConfigXA Lit+Third (Full Accurate)' that also has a uniform MAE of zero, while only consuming 1 μW more than $\boldsymbol{R}_{1555}$. Interestingly there is also an SMApproxLib configuration, that be-

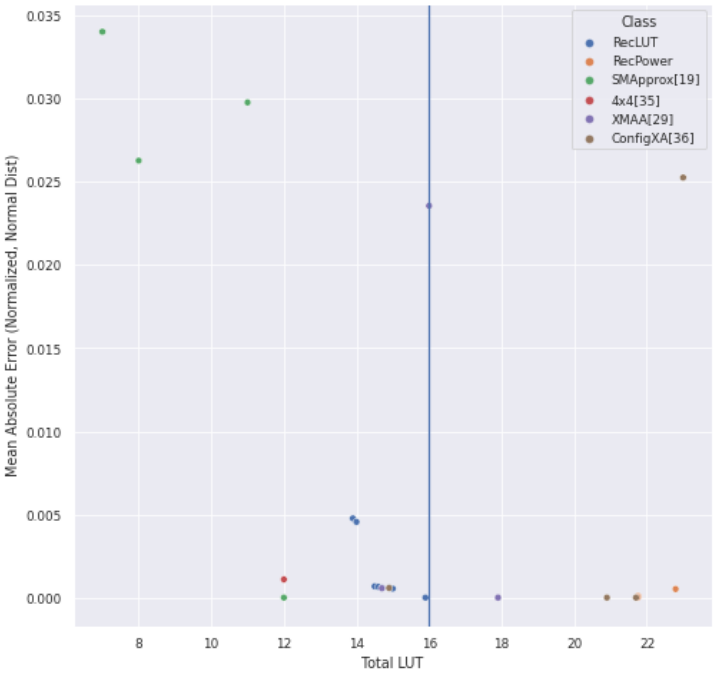


Figure 5.4: Quality-Area trade-off of investigated 4x4 Multipliers

sides being low in area, also outperforms the numeric benchmark. All other configurations perform worse than an accurate numeric multiplier.

In earlier results a strong correlation between latency and power consumption has already been shown, the results in Figure 5.6 show a similar picture. Where it is the 'RecPower' configurations that have the shortest latency, and thus the highest possible frequency. A difference here is that more configurations here show a better latency than the benchmark, such as some 'RecLUT' configurations and the '4x4'. Similarly, another 'SMApproxLib' that in the previous figure compared worse in power consumption than the benchmark, outperform here in terms of latency.

LUT/m	Power(μW)	MAE (Normal)	Max Freq (MHz)	Name	CARRY4/m	MAE (Uniform)
7	159	3.4031e-2	556.17	SMApproxLib Approx2[19]	0	4.1992e-2
8	186	2.6281e-2	547.05	SMApproxLib Approx3[19]	0	4.1992e-2
11	272	2.9781e-2	393.24	SMApproxLib Approx1[19]	2	3.1982e-2
12	267	0.0000e+0	319.28	SMApproxLib Accurate[19]	3	0.0000e+0

Table 5.3: Pareto Frontier of 4 x 4 set, minimizing LUT/m and MAE

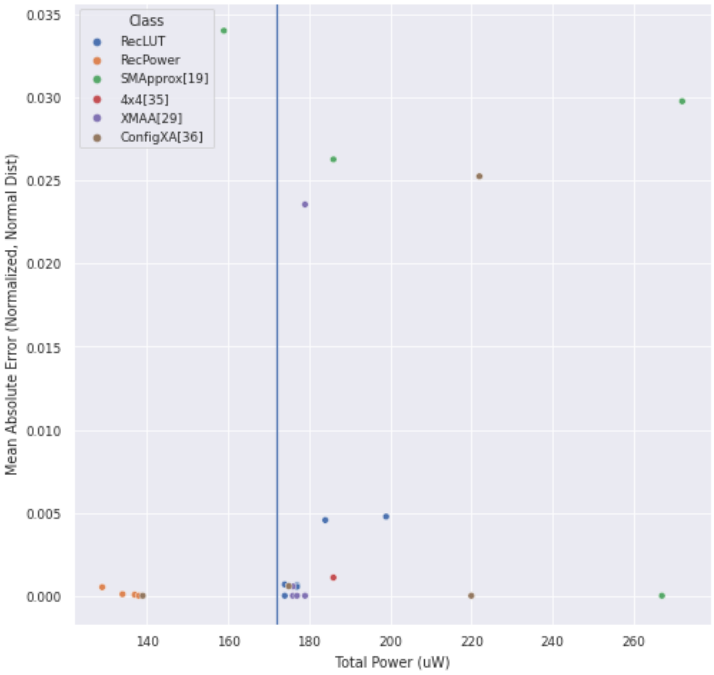


Figure 5.5: Quality-Power trade-off of investigated 4x4 Multipliers

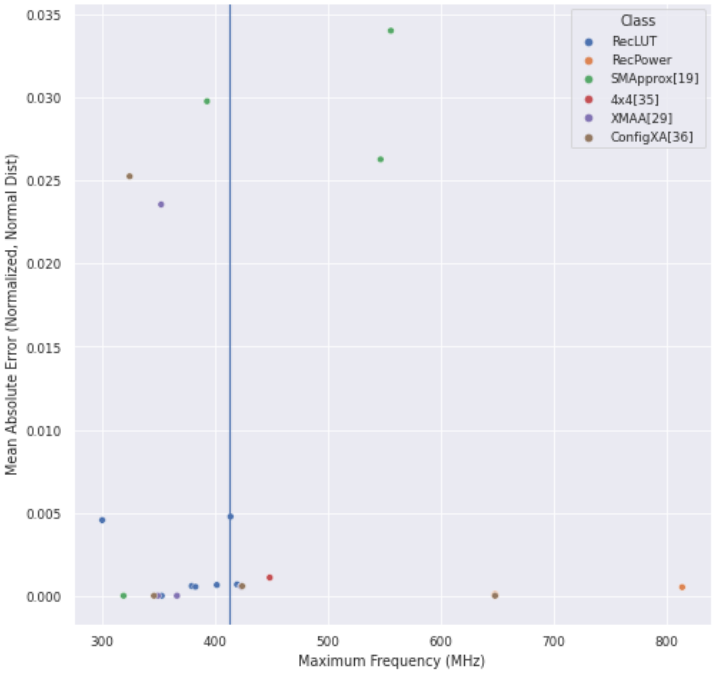


Figure 5.6: Quality-Latency trade-off of investigated 4x4 Multipliers

LUT/m	Power(μW)	MAE (Normal)	Max Freq (MHz)	Name	CARRY4/m	MAE (Uniform)
22.8	129	5.1563e-4	814.33	$\boldsymbol{R}_{1311}$ (RecPower)	0	1.1047e-2
21.8	134	9.3800e-5	648.51	$\boldsymbol{R}_{1315}$ (RecPower)	0	1.0742e-2
21.7	137	6.2500e-5	648.51	$\boldsymbol{R}_{4335}$ (RecPower)	0	1.7578e-2
21.7	138	0.0000e+0	648.51	$\boldsymbol{R}_{1555}$ (RecPower)	0	7.8125e-3

Table 5.4: Pareto Frontier of 4 x 4 set, minimizing Power/m and MAE

5.5 Final Design

In Chapter 4, FPGA multipliers models are discussed, resulting in the design-space exploration of 4-bit Recursive Multipliers. An exhaustive analysis is done for all Recursive Multipliers using the set of 2-bit behavioral models: $\boldsymbol{S} := \{M_1, M_2, M_3, M_4, M_{Acc}\}$. Two Pareto Frontiers are created, one that optimizes for low area, and one for lower power consumption. From the results could be seen that optimizing for low resource usage does not automatically mean lower power consumption. These two frontiers have been compared in this Chapter to available open-source approximate multiplier models published in literature. The multiplier with the lowest power consumption is the Recursive Multiplier $\boldsymbol{R}_{1311} := \{M_1; M_3; M_1; M_1\}$, having a Normalized Mean Absolute Error for a Normal Distribution($i = 1000, \mu = 8, \sigma = 1.5$) of 0.000515625.

The implementation can be seen in Figure 5.7. Noteworthy is that for the implementation of $\boldsymbol{R}_{1311}$, there is zero routing between FPGA Slices. The calculation of O_7 till O_3 (PROD[7] till PROD[3]) is each done on a single slice, consisting of four LUT6_2 Primitives, two MUXF7 primitives controlled by A_2 and one MUXF8 Primitive controlled by B_1 creating a combinational logic function calculation the output bit without extra delays due to routing. The lower significant output bits are done using only LUTs. For O_2, a 6-input logic function is implemented. O_1 is simply two AND-gates and one OR-gate implemented using a single LUT. The first bit is simply one LUT implementing: $O_0 = A_0 \& B_0$. The LUT6_2 primitives implementing O_0 and O_1 could be implemented using a single LUT, explaining the non-integer LUT/m (22.792) results for implementing $N = 1000$ $\boldsymbol{R}_{1311}$ multipliers on a single FPGA. Next to the lack of any slice-to-slice routing that would have added to the dynamic power consumption, the inputs A_2 and B_1 are only connected to the multiplexers, causing minimal cascading switching behavior.

When compared to an accurate Numeric Multiplier, as described in 4.2, which has higher power consumption but requires less LUT resources, the distinction can be seen. The implementation of this multiplier is shown in

Figure 5.8, Here the placement of resources and their connectivity is shown. Multiple routing connections can be seen between the slices. Besides the CARRY route between the two CARRY4 primitives, which are physically connected within the CLB, these are routed over the routing matrix of the FPGA, increasing power consumption and latency. No MUX primitives are used, and some LUT outputs connect to other LUT inputs, causing a cascading path with increased internal dynamic power switching. The final results, compared to the numeric 4-bit benchmark, is as follows:

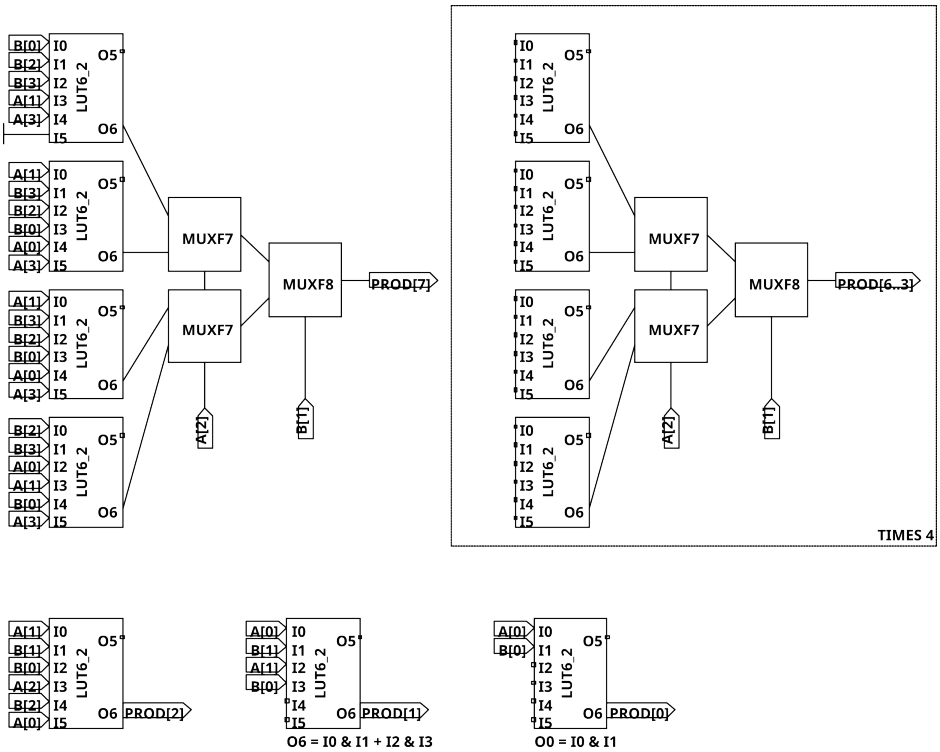


Figure 5.7: Implementation of $\boldsymbol{R}_{1311}$ on Xilinx 7-series using Approxy

- Name: $\boldsymbol{R}_{1311}$
- LUT/m: 22.792(+6.792)
- Power: $129\mu W(-43\mu W)$
- MaxFreq: $817.332MHz(+403.772MHz)$
- CARRY4: 0 (-2)
- MAE (Uniform): 0.01104736328125

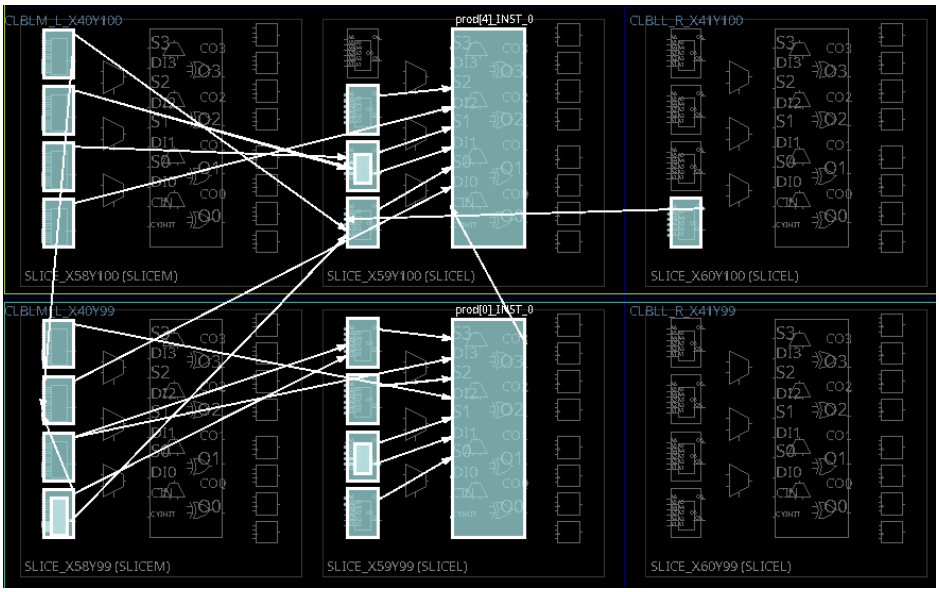


Figure 5.8: Placement of an Accurate 4-bit Numeric Multiplier on Xilinx 7-series using Approxy

- Error Rate (Uniform): 49
- Error Magnitude (Uniform): 40
- MAE (Normal) : 0.000515625

Chapter 6

Conclusion

This thesis shows that applying Recursive Multiplier techniques for ASIC technology can be used to achieve reduction in terms of power consumption and reduced latency on the FPGA. The highest power gains are reached in the 4-bit multiplier $\boldsymbol{R}_{1311}$, reaching a reduction of 25% in power consumption. The Pareto Front also shows configurations reaching 19.77% power reduction for a near-zero Mean Absolute Error for a normal distributed input, making the application of Approximate Computing to FPGA signal processing a viable approach. State-of-the-art libraries applying Approximate Computing to FPGA multipliers generally optimize for FPGA resource allocation. Comparatively, to these available open-source VHDL libraries, the investigated multipliers do not show any interesting gains in terms of chip-area, thus are not part of this Pareto Frontier. They however do outperform them in terms of power consumption. For the investigated multipliers, the general approach within FPGA Power Optimizing, in which reduction in area results in reduced power consumption does not seem to hold. Bigger multipliers, in terms of chip utilization, that mostly keep switching propagation within the slice use less power than significantly smaller multipliers that make more usage of the FPGAs routing network.

In Chapter 3, 'Approxy', an extendable open-source framework is presented to automate the Design Exploration of various multiplier models. By leveraging the generic model applicable to all (approximate) multipliers, 'Approxy Runs' can be created to automate FPGA experiments in a controlled environment. Due to the architectural connection between the behavioral modelling in the imperative programming language 'Go' and the automatic generation of VHDL code, models can be validated against each other by using the automated testing functionality. Time execution experiments show that 'Approxy' has negligible overhead compared to a classic workflow using Vivado

and VHDL Simulators only. Nevertheless, the exploration of 8-bit Recursive Multipliers and beyond is still unfeasible without further extending of the methodology by 'pruning' the available Design Space.

Chapter 7

Future Work

7.1 Multiply-Accumulate and Applications

While in Chapter 3, the initial work for investigating the Multiply-Accumulate operation in terms the model and validation methodology have been investigated, the Design Space Exploration has not been executed yet. Given the behavioral approach of the implementation of the MAC, the synthesis tool is free to implement the Accumulator without any structural constraints besides the multiplier. Therefore, it is unknown still if extending the 4-bit Recursive Multiplier Design Space to a MAC Design Space by applying the Accumulator will show similar results or show different Pareto Fronts. What is known however is that the current comparison favors designs with a near-zero Mean Absolute Error. If this near-zero MAE is actually a positive or negative mean error in disguise, accumulation will decrease the quality of the system. Multiply-Accumulators using the MACISH approach by Gillani et al. [18], using the same Recursive Multipliers, implement Internal-Self-Healing by having 'mirrored' errors, which cancel each other out. Investigating this effect for MAC operations on the FPGA might show different results. Furthermore, the results can be further validated by implementing an application using the approximate multipliers. This will make it possible to show application-specific error metrics, showing the trade-of in terms of loss-of-quality to power reduction within real-life applications.

7.2 Expanding to 8-bit and beyond

Many applications require data representation with a higher resolution than four bits. A quick calculation within the Design Space Exploration, given the results of the time execution analysis, showed while 4-bit exploration

is feasible, 8-bit and beyond is not viable using the current methodology. Other papers([18][19]) use the resource consumption of $\frac{N}{2}$ multipliers to estimate the 'cost' of higher resolution multipliers. Exploration of the 4-bit Recursive Multipliers have showed that using the data of the 2-bit Behavioral Multipliers gives no proper estimate for resource allocation and power consumption, because of the freedom that the synthesis tool has to optimize designs. However, the results of the Pareto Fronts can be used as inspiration for structural modelling as an RTL Design, as described in Section 4.1. This way the synthesis tool does not apply optimizations to the overall design, making the earlier described 'cost' estimates valid.

Other approaches to Design Space Exploration can be found in the Error Metric analysis. By ignoring the 'Verification' step of the Approxy workflow, the calculation of the Mean Absolute Error of the Design Space can be greatly accelerated. For instance, the exploration of all MAE-values for normal distributed input for the 4-bit Recursive Multipliers takes without optimizations one second. While linearly extrapolating this to the 8-bit Design Space makes the exploration still infeasible, Equation 4.6 and further software optimizations can be used to exclude subsets of multipliers with high mean-errors from the exploration.

Bibliography

[1] G. Rodrigues, F. L. Kastensmidt, and A. Bosio, "Survey on approximate computing and its intrinsic fault tolerance," *Electronics 2020, Vol. 9, Page 557*, vol. 9, p. 557, 3 2020.

[2] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, *Managing Performance vs. Accuracy Trade-Offs with Loop Perforation*. ESEC/FSE '11, New York, NY, USA: Association for Computing Machinery, 2011.

[3] D. O. N. A. L. D. MICHIE, ""memo" functions and machine learning," *Nature*, vol. 218, no. 5136, pp. 19–22.

[4] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design and Test*, vol. 33, pp. 8–22, 2 2016.

[5] J. Qian and J. Wang, *A 4-bit array multiplier design by reversible logic*. Taylor & Francis Group, 10 2014.

[6] S. M. S. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology: This paper reflects on how moore's law has driven the design of FPGAs through three epochs: the age of invention, the age of expansion, and the age of accumulation," *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, 2018.

[7] Aldec, "Xilinx design flow," 2022. `https://www.aldec.com/en/solutions/fpga_design/fpga_vendors_support/xilinx--xilinx-fpga-design-flow` [Accessed 1 Sept 2022].

[8] Xilinx and Inc, "7 series fpgas configurable logic block user guide (ug474)." `https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB`, 2016.

[9] Hwang, "D flipflop timing." `http://hades.mech.northwestern.edu/index.php/File:D_flipflop_timing.gif`, 6 2006.

[10] Xilinx and Inc, "Vivado design suite user guide, design analysis and closure techniques, ug906 (v2017.3)." `https://docs.xilinx.com/v/u/2017.3-English/ug906-vivado-design-analysis`, 2017.

[11] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital cmos circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[12] A. Amara, F. Amiel, and T. Ea, "Fpga vs. asic for low power applications," *Microelectronics journal*, vol. 37, no. 8, pp. 669–677, 2006.

[13] Wikimedia Commons, "Layout of nmos and pmos components in an inverter (not gate).," 2006.

[14] Wikimedia Commons, "Circuit diagram of an sram cell, built with six mosfets.," 2009.

[15] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18TH IEEE EUROPEAN TEST SYMPOSIUM (ETS)*, IEEE.

[16] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *2011 24th Internatioal Conference on VLSI Design*, pp. 346–351, IEEE, IEEE, 2011.

[17] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "Evoapproxsb: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 258–261, Institute of Electrical and Electronics Engineers Inc., 5 2017.

[18] 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. 77142–77160, 2019.

[19] 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)*, pp. 1–6, Institute of Electrical and Electronics Engineers (IEEE), 9 2018.

[20] G. A. Gillani and A. B. J. Kokkeler, "Poster: Go green radio astronomy: Approximate computing perspective: Opportunities and challenges," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, vol. 2, pp. 300–301, Association for Computing Machinery, Inc, 4 2019.

[21] S. G. A. Gillani, *Exploiting error resilience of iterative and accumulation based algorithms for hardware efficiency.* PhD thesis, 7 2020.

[22] C. Niemann, M. Rethfeldt, and D. Timmermann, "Approximate multipliers for optimal utilization of fpga resources," *Proceedings - 2021 24th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2021*, pp. 23–28, 4 2021.

[23] F. Cabello, J. Leon, Y. Iano, and R. Arthur, "Implementation of a fixed-point 2d gaussian filter for image processing based on fpga," in *2015 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, vol. 2015-December, pp. 28–33, IEEE Computer Society, 12 2015.

[24] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17322–17341, 8 2017.

[25] Z. Wang, M. A. Trefzer, S. J. Bale, and A. M. Tyrrell, "Approximate multiply-accumulate array for convolutional neural networks on fpga," in *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 35–42, Institute of Electrical and Electronics Engineers Inc., 7 2019.

[26] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, 1962.

[27] Go, "template package - text/template - version: go1.18rc1." `https://pkg.go.dev/text/template@go1.18rc1`. Accessed: 2022-03-01.

[28] Xilinx and Inc, "Ultrascale architecture libraries guide (ug974)." `https://docs.xilinx.com/v/u/2018.1-English/ug974-vivado-ultrascale-libraries`, 2018.

[29] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, and J. Henkel, "Architectural-space exploration of approximate multipliers," in *Proceedings of the 35th International Conference on Computer-Aided Design*, vol. 07-10-November-2016, Institute of Electrical and Electronics Engineers Inc., 11 2016.

[30] Intel, "Stratix v device handbook: Volume 1: Device interfaces and integration , 1.2.1. Normal Mode." `https://www.intel.com/content/www/us/en/docs/programmable/683665/current/normal-mode.html`, Last accessed on 2022-03-31, 2022.

[31] 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, 2021.

[32] 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)*, vol. 2020-January, pp. 599–604, Institute of Electrical and Electronics Engineers Inc., 1 2020.

[33] Xilinx and Inc, "Ultrascale architecture dsp slice (ug579)." `https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp`, 2013.

[34] 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. 49112–49128, 2018.

[35] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique, and A. Kumar, "Area-optimized low-latency approximate multipliers for fpga-based hardware accelerators," in *Proceedings of the 55th annual design automation conference*, pp. 1–6, IEEE, 2018.

[36] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013.

Appendix

A.1 Approxy Case Study

```
1         func PaperExample() {
2             //Start of Approxy Run
3             CurrentRun := Viv.StartRun(ReportPath, OutputPath, "
                  Rec_1234")
4             CurrentRun.ClearData()
5
6             //Generation of Recursive Multipler
7             Rec_1234 := VHDL.NewRecursive4("Rec1234", [4]VHDL.
                  VHDLEntityMultiplier{M1, M2, M3, M4})
8             Rec_1234.GenerateTestData(OutputPath)
9             Rec_1234.GenerateVHDL(OutputPath)
10
11            //Optional Verification of Recursive4
12            verify := Viv.CreateXSIM(OutputPath, "prePR",
                  Rec_1234.GenerateVHDLEntityArray())
13            verify.SetTemplateMultiplier()
14            verify.Exec()
15            err := Viv.ParseXSIMReport(OutputPath, Rec_1234)
16            if err != nil {
17                log.Fatalln(err)
18            }
19
20            //Generation of N=1000 Scaler of Rec1234
21            Rec_1234_scaler := VHDL.New2DScaler(Rec_1234, 1000)
22            Rec_1234_scaler.GenerateTestData(OutputPath)
23            Rec_1234_scaler.GenerateVHDL(OutputPath)
24
25            //Synth + Place + Route
26            viv := Viv.CreateVivadoTCL(OutputPath, "main.tcl",
                  Rec_1234_scaler, VivadoSettings)
27            viv.Exec()
28            //PostSynthesisAnalysis
29            post_analysis := Viv.CreateXSIM(OutputPath, "postPR"
                  , Rec_1234_scaler.GenerateVHDLEntityArray())
30            post_analysis.SetTemplateScaler(1000)
```

```
31              post_analysis.CreateFile(true)
                                        //Create PostPR Testbench
32              VHDL.NormalTestData(Rec_1234_scaler, OutputPath,
                    1000) //Create i=1000 Normal Test Data for 4-bit
33              post_analysis.Funcsim()
34              viv.PowerPostPlacementGeneration()
35
36              //Create Report
37              Report := Viv.CreateReport(OutputPath,
                    Rec_1234_scaler)
38              Report.AddData("MAE_Uniform", strconv.FormatFloat(
                    Rec_1234.MeanAbsoluteError(), 'E', -1, 64))
39              Report.AddData("MAE_Normal_1000", strconv.
                    FormatFloat(Rec_1234.MeanAbsoluteErrorNormalDist
                    (1000), 'E', -1, 64))
40              Report.AddData("Overflow", strconv.FormatBool(
                    Rec_1234.Overflow()))
41              CurrentRun.AddReport(*Report)
42          }
```

Listing A.1: Example of Approxy run