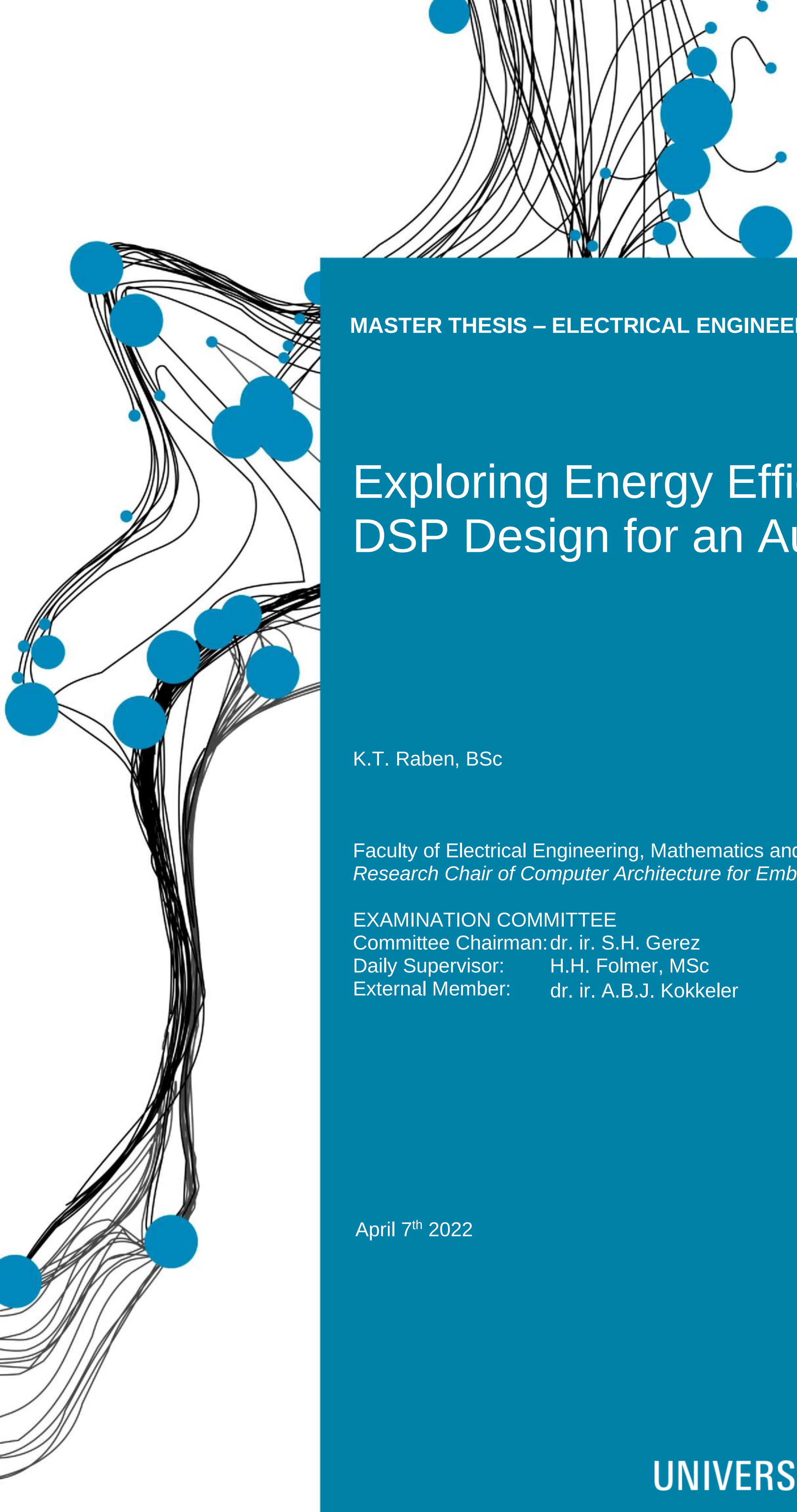


MASTER THESIS – ELECTRICAL ENGINEERING

Exploring Energy Efficient DSP Design for an Audio ASIC

K.T. Raben, BSc

Faculty of Electrical Engineering, Mathematics and Computer Science
Research Chair of Computer Architecture for Embedded Systems (CAES)

EXAMINATION COMMITTEE

Committee Chairman:	dr. ir. S.H. Gerez
Daily Supervisor:	H.H. Folmer, MSc
External Member:	dr. ir. A.B.J. Kokkeler

April 7th 2022

UNIVERSITY OF TWENTE.

Abstract

With more and more consumer audio products becoming portable, maximizing battery lifetime has become extremely important. This drives the need for more energy-efficient Application Specific Integrated Circuits (ASIC) with a focus on audio. Often these chips have a Digital Signal Processor (DSP) section in their design to manipulate the audio signal before being amplified. Such a DSP typically implements various signal processing algorithms. This thesis explores how certain architectural choices influence the energy efficiency of such a DSP. A focus is laid on how the time-area trade off affects the energy consumption of a Finite Impulse Response (FIR) filter implementation. Through the use of Clash[1], a hardware description language with strong abstraction mechanisms, a method is presented where the amount of parallelism in a processor-like architecture can be controlled by one argument, creating a ”knob” to generate designs which possess the same functionality but do so in a varying number of clock cycles. Through industry standard tools all the created designs are synthesized and the energy consumption is simulated. The energy efficiency of the processor designs generated with the variable parallelism are compared to a fully dedicated FIR filter implementation.

The results show that with a a fully parallel design a 85% reduction in energy consumption can be achieved with respect to a design with only a single multiplier, but at the cost of 8 times the chip area needed. By turning the ”knob” results are obtained for designs with more than one multiplier, but less multipliers than the fully parallel design. From these results it can be seen that a significant decreases in energy consumption occur for small increases in number of multipliers. A design with 2 multipliers already reduces the energy consumption by 35%, while only increasing the chip area needed by 2%. The results also show however that further increasing the amount of multipliers does not always result in a decrease in energy consumption.

With obtaining the results it is also shown how the presented method can help an IC designer gain more insight in navigating the design space from an energy consumption perspective. Using the proposed method, the effect a specific architectural choice has on energy efficiency can be investigated.

CONTENTS

1 INTRODUCTION

With the increase in popularity of wireless headphones and true wireless in-ear monitors there comes a drive to create audio products that can function a long time on small batteries. To achieve this long battery lifetime the energy consumption of the product must be brought down. This requirement results in a desire for audio ICs with low energy consumption.

In audio IC's, before a bitstream is converted to an analog signal, a manipulation of the stream is often desired. High-performance audio ICs typically have specific digital hardware to handle a set of possible manipulations. This hardware is required to be fast enough to handle the data-rates but should also possess the flexibility such that it can be configured to perform several different signal processing functions. Flexible hardware that serves such a purpose is called a Digital Signal Processor (DSP). It lies between a traditional processor and dedicated hardware in terms of performance, energy efficiency and programmability.

The DSP section of the chip can have a large impact on this total energy consumption and should therefore be designed with energy efficiency in mind. In designing this flexible hardware many architectural choices have to be made which all have an influence on the total energy consumption of the hardware. Gaining insight into the exact influence of a specific architectural choice is however a difficult task. This work delves into the aspects of what makes an energy efficient DSP design and proposes a method to easily make high-level architectural choices and observe the effect this choice has on energy efficiency.

In particular, this work investigates the architectural choice of how many calculations are done simultaneously and how that influences the energy efficiency of a DSP design. By using the proposed method, more exact measurements can be made of how much efficiency is gained by performing more calculations at the same time and what the costs are in terms of chip area needed. This information can guide the IC designer in creating an efficient, cost-effective hardware design.

Collaboration with Axign

This work is developed in collaboration with Axign, a fabless semiconductor company from Enschede. Axign develops state-of-the-art class-D audio amplifier controllers. For future products, Axign would like to include a energy efficient DSP as part of their amplifier controller IC.

Thesis Layout

The structure of this thesis is as follows: First, in chapter 2 the global research topic of energy efficient DSP design will be refined in a concrete research question. Chapter 3 will elaborate on the theory involved in creating energy efficient digital hardware. Keeping this knowledge in mind a method of creating multiple design architectures is presented in chapter 4. For each design, the energy consumption is determined through the method described in chapter 5. An overview of the results for the different designs is given in chapter 6, after which a conclusion is presented in chapter 7.

2 PROBLEM DEFINITION

2.1 DSP algorithms for audio

The functionalities that an audio DSP should support naturally differ for each product, but common functions are: Channel mixing, volume control, equalization, and interpolation/decimation of signals.
Most of the above-mentioned functionality is some combination of multiplication and addition of a set of (delayed) samples. The algorithms behind many of these implementations are often Finite Impulse Response (FIR) filters, Infinite Impulse Response (IIR) filters, a Least Mean Squares (LMS) calculation, or the Fast Fourier Transform (FFT).

2.2 Implemented Algorithm

To limit the design space that needs to be explored this research is narrowed down to a single algorithm to be implemented. Of the algorithms mentioned in section 2.1, the FIR algorithm can be used to perform, among other things, interpolation, decimation, and equalization. With respect to the IIR algorithm it is less complex, as it is a system without feedback. The FFT and LMS algorithms are less often used. For these reasons, the research focuses on creating an FIR filter implementation.

2.3 Time-area trade-off

To further limit the scope, the question is approached from a digital IC architecture perspective. All DSP algorithms mentioned above are dominated by multiplier operations and memory access [2]. Therefore this research will mainly revolve around the chosen structure of digital building blocks that implement these operations, such as memory elements, multipliers and adders. For a FIR implementation changing the number and structure of memory elements and multipliers results in varying the time-area trade-off. Designs with a different time-area trade-off will take a different number of clock cycles to finish the operation and can be implemented with a different amount of chip area. This thesis focuses on how that trade-off affects energy efficiency.

2.4 Research Question

After constraining the larger question of how to design an energy efficient DSP the following research question is formed:
"How does the time-area trade-off affect the energy efficiency of a streaming FIR filter ASIC implementation for audio purposes?"

3 ENERGY EFFICIENT DESIGN

3.1 Introduction

Energy efficient design can be approached from many different levels of abstraction. The total energy-efficiency will depend on the choices made on all levels. But especially on how well the different levels are attuned to each other. For instance, circuit level analysis may give insight into how the power consumption of memory access is in proportion to that of a multiplier. This information can be used to find a good combination of an algorithm and an architecture that can optimize for multiplications and memory access in an energy-efficient manner.

This chapter will discuss the different levels mentioned above in more detail. The relevant aspects that need to be considered in the DSP design are elaborated upon and aspects that fall outside of this research scope are addressed.

3.2 Energy Efficiency on a Circuit Level

At the basis of digital CMOS circuitry lies the inverter shown in Figure 3.1. There are three major sources of power dissipation in these circuits: A switching component P_s, a short-circuit component P_{sc} and a leakage component P_l. Equation 3.1 gives a description of these sources of dissipation[3].

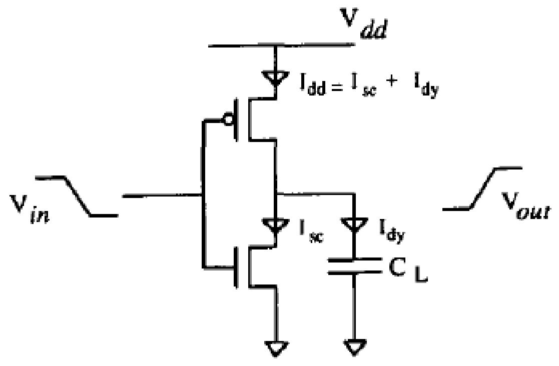


Figure 3.1: CMOS circuitry for digital inverter [4]

$$\begin{aligned} P_{\text{total}} &= P_s + P_{sc} + P_l \\ &= p_t \left(C_L \cdot V_{dd}^2 \cdot f_{\text{clk}} \right) + I_{sc} \cdot V_{dd} + I_{\text{leakage}} \cdot V_{dd} \end{aligned} \tag{3.1}$$

Here $I_{leakage}$ is the leakage current and mostly dependent on the chip process technology[5]. I_{sc} is the short-circuit current that occurs when switching. C_L is the load capacitance that needs to be driven. p_t is the probability of a switch occurring between zero to one or vice versa.

For modern CMOS technologies, the contributions of P_l and P_{sc} are generally well below that of P_s[5][6][7]. Therefore, the focus of this work is mainly laid on the switching power dissipation.

3.2.1 Voltage Scaling

Looking at equation 3.1, a very effective method to reduce the switching power would be to lower the supply voltage, since the power scales down quadratically with V_{dd}. However, decreasing the voltage will increase the delay of the circuit [3]. Architectures that are tolerant to this increase in delay have been an active area of research [8],[9]. Improvements in chip process technology further enable the reduction of supply voltages.

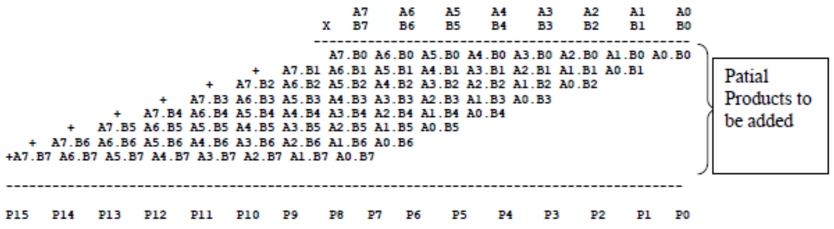
3.2.2 Reducing the Average Switched Capacitance

When the supply voltage, the leakage power and the short-circuit power remain constant, power reduction can then be achieved by reducing the amount of capacitance that is being switched on average. In equation 3.1 this is the term $p_t \cdot C_L \cdot f_{clk}$. C_L consists of the gate capacitance of subsequent inputs attached to the output of the inverter, capacitance of the interconnect wires and diffusion capacitance on the drains of the transistors of the inverter[10].

The gate and diffusion capacitance are determined by the chip process technology. Length of the interconnect wires and the number of gates is highly dependent on the chosen architecture. How often these switch (p_t) is then dependant on the executed algorithm (and of course the input signal).

3.3 Energy Efficient Multiplication

In almost all functionality of a DSP, multiplication plays a crucial role. The energy efficiency of the multiplier is therefore significant in the efficiency of the total DSP. There are many multiplier architectures, and which architecture is most efficient differs per situation. Important factors that determine which architecture is optimal are the bit-sizes of the input arguments and the required speed of the multiplier. To give a brief introduction into multipliers and how they can differ two architectures are discussed below.
The most common multiplication method is the ”add and shift” algorithm, which is illustrated in figure 3.2. The total multiplication can be divided into two sets of operations: partial product generation and addition.



A7 A6 A5 A4 A3 A2 A1 A0
X B7 B6 B5 B4 B3 B2 B1 B0
A7.B0 A6.B0 A5.B0 A4.B0 A3.B0 A2.B0 A1.B0 A0.B0
+ A7.B1 A6.B1 A5.B1 A4.B1 A3.B1 A2.B1 A1.B1 A0.B1
+ A7.B2 A6.B2 A5.B2 A4.B2 A3.B2 A2.B2 A1.B2 A0.B2
+ A7.B3 A6.B3 A5.B3 A4.B3 A3.B3 A2.B3 A1.B3 A0.B3
+ A7.B4 A6.B4 A5.B4 A4.B4 A3.B4 A2.B4 A1.B4 A0.B4
+ A7.B5 A6.B5 A5.B5 A4.B5 A3.B5 A2.B5 A1.B5 A0.B5
+ A7.B6 A6.B6 A5.B6 A4.B6 A3.B6 A2.B6 A1.B6 A0.B6
+A7.B7 A6.B7 A5.B7 A4.B7 A3.B7 A2.B7 A1.B7 A0.B7
P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0
Patial Products to be added

Figure 3.2: ”Add and shift” multiplication algorithm.[11]

3.3.1 Partial Product Generation

The partial product generation of a classic ”add and shift” multiplier is rather straightforward. In figure 3.3 the structure can be seen. There is the multiplicand A and the multiplier B. The structure illustrates that for a M-bits multiplicand and a N-bits multiplier, N partial products are generated and M·N AND gates are needed.

Modified Booth Encoding A popular method that reduces the number of partial products that need to be added is Modified Booth encoding. This method uses the phenomena in bitwise operations that doubling and sign inversion require less logic and switching with respect to pure addition. In the classical case only one bit of the multiplier is evaluated and determines whether the partial product that is generated is $1 \cdot multiplicand$ or $0 \cdot multiplicand$. This is also illustrated in table 3.1. Here A partial products are created.

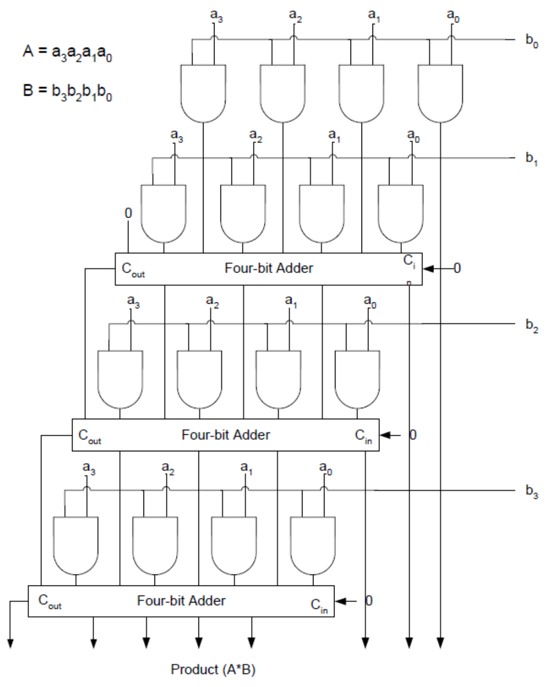


Figure 3.3: Classic "add shift" multiplier structure [11]

A_i	**Partial Product**
0	$0 \cdot multiplicand$
1	$1 \cdot multiplicand$

Table 3.1: Partial Product generation for a classical "add and shift" multiplier.

The process of Radix4 Booth encoding is illustrated in figure 3.4 for an 8 bit multiplier argument. The 8 bits are divided in overlapping sets of 3 bits. For the most right encoding a zero is padded.

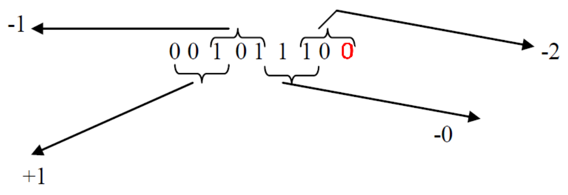


Figure 3.4: Radix 4 Booth encoding of an 8 bit number. A zero is padded to the right[11].

The 8 bit representation has now been encoded to 4 3-bit representations. The partial product generation from these 3-bit representations can be seen in table 3.2. Using this method multiplications with an N-bits multiplier result in a maximum of $\lfloor\frac{N+2}{2}\rfloor = \lfloor\frac{N}{2}+1\rfloor$ partial products for the Radix 4 case.
Instead of two possible partial products there are now five variants which complicates the generation of these products. Multiplying the multiplicand with 2 and -1 are tasks that require less switching with respect to multiple additions. Therefore the modified Booth encoding enables a significant reduction of the partial products to be added while replacing the effort needed with operations that are more efficient, such as the shifting of bits.

Applying a higher radix further reduces the number of partial products that need to be added. However, it does introduce the need to produce odd multiples, which are less efficient to generate.

A_{i+1}	A_i	A_{i-1}	**Partial Product**
0	0	0	$0 \cdot multiplicand$
0	0	1	$+1 \cdot multiplicand$
0	1	0	$+1 \cdot multiplicand$
0	1	1	$+2 \cdot multiplicand$
1	0	0	$-2 \cdot multiplicand$
1	0	1	$-1 \cdot multiplicand$
1	1	0	$-1 \cdot multiplicand$
1	1	1	$0 \cdot multiplicand$

Table 3.2: Partial Product generation for a Radix 4 Booth encoded "add and shift" multiplier.

3.4 Power Analysis of Memory

On chip addressable memory typically consists of an array of SRAM cells. The structure of a single bit SRAM cell can be seen in figure 3.5. When the word line (WL) is low the data in the cross-coupled inverters is held. When the word line is high the cell can be read or written to. Both reading and writing are performed through the bit lines[12].

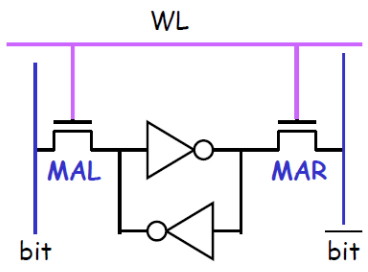


Figure 3.5: Schematic layout of a single bit SRAM cell.[12]

Placing these cells in an array and adding control logic results in a structure as shown in figure 3.6. [13] categorizes the energy consumption of such a structure in four categories:

- **Input line dissipation:** Caused by driving transitions on the input lines and input latches. Increases with the word width of the memory.
- **Word line dissipation:** Caused by driving both the word select wires that enable a read/write and the gates of the transistors in the cells. Increases with the word width of the memory.
- **Bit line dissipation:** Caused by driving the bit lines during a write or read. There are two bit lines per column of SRAM cells. The total bit line dissipation increases with word width but also with the amount of words in the memory as the wires will be longer and thus have a higher capacitance that needs to be driven.
- **Output line dissipation:** Caused by driving transitions on the output lines. Increases with the word width of the memory.

The address decoding is not taken into account with these categories. The total dissipation of demultiplexing an address line also increases with the number of words in the memory.
An important realization that comes from this analysis is that increasing the number of words in the memory can increase the energy needed per read/write significantly. This is also shown in related work [15],[16]. Both adding rows and columns increases the energy consumption per access, but only the column increase delivers more bits per access.

These facts motivate the choice to keep the memories used as small as possible.

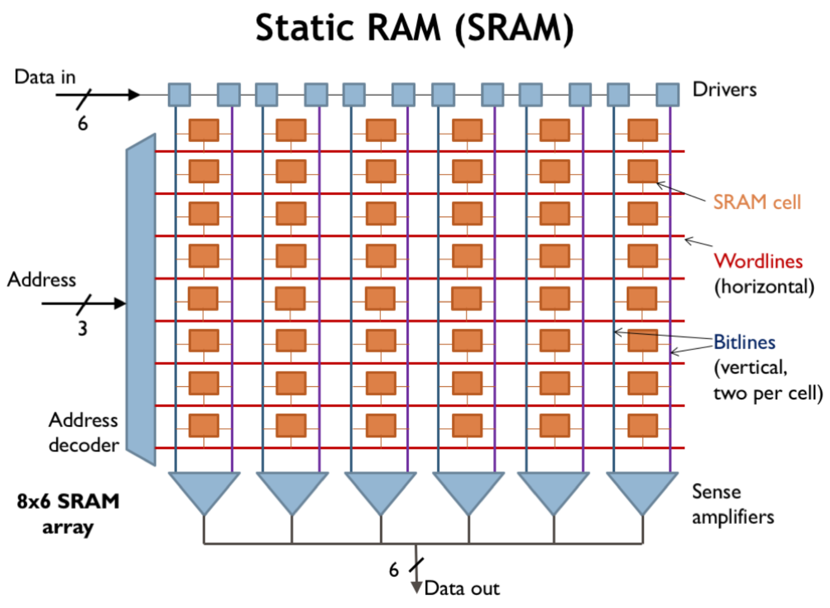


Figure 3.6: Schematic of a memory layout with 8 words of 6 bits. [14]

3.5 Energy Efficiency on an Algorithmic Level

The algorithmic level concerns the algorithms that are chosen to obtain the desired signal manipulation. For example, when the goal is to lower the energy of low frequency content in a signal (a high pass filter), different mathematical algorithms can be chosen to achieve that goal. The FIR and IIR algorithm are examples of such algorithms, each with their own performance and mathematical description. As discussed before, an important technique for reducing energy consumption is reducing the amount of switching in the system. The algorithm that is executed can often be manipulated to better fit the hardware architecture and improve results. Re-ordering of multiplications and additions can for instance reduce the number of operations that have to be executed[17]. Additionally, the possibility to execute operations in parallel is highly dependent on the algorithm, especially when feedback is present[2].

3.6 Energy Efficiency on Architectural Level

Just as an algorithm can be adjusted to fit the architecture, this architecture can be adjusted to better fit the implemented algorithm. When analyzing an architecture from an energy perspective focus is laid on which operation is performed by which piece of hardware and how the different functional blocks interact. Examples of choices that can be made are for instance different levels of parallelism, where intermediate results are stored and how different areas of the design communicate with each other. The goal is again to reduce the total amount of capacitance that is switched [18]. Two relevant aspects of an architecture are spatial locality and regularity which are described below in more detail.

3.6.1 Spatial Locality

Spatial locality is a term which describes the degree that an algorithm has isolated clusters of operations with few interconnections between these clusters[17]. This effect is described in figure 3.7. Here a three-stage biquad IIR is laid out with the local clusters encircled. [17] argues that to improve energy efficiency, resource sharing between the local clusters should be avoided. This to keep the amount of interconnects and bus multiplexing to a minimum and reduce the amount of switching.

This example focuses on the interconnects between computational units. Perhaps even more important is to optimize an architecture with memories such that the majority of the data transfers take place within the local clusters[6]. Global buses and large memories incur a large amount of capacitance to be switched and thus typically result in a higher energy consumption.

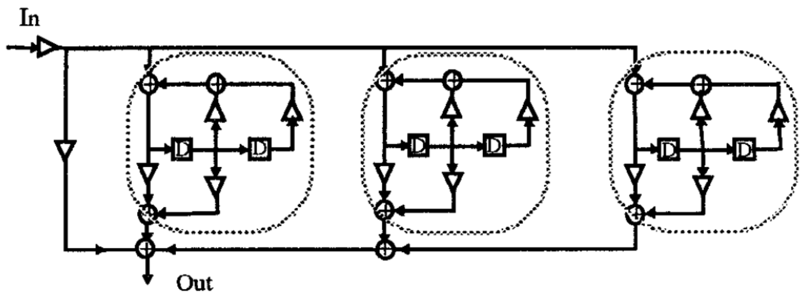


Figure 3.7: Architecture for a three-stage biquad IIR filter. Three local clusters can be identified where there are little interconnections with other clusters. [17]

3.6.2 Regularity

Regularity is the degree to which common patterns appear in an algorithm. Regularity enables the design of less complex architectures. This reduces the amount of control overhead and thus additional switching [17]. By identifying regularity in an algorithm the architecture that it is mapped on can be optimized to reduce the amount of interconnections and multiplexers necessary[19].

3.7 Conclusion

Designing for low energy consumption is a multi-dimensional problem with many facets. The desired DSP algorithm for this work (FIR) relies heavily on multiplication, addition, and memory access. These operations and what aspects influences their energy usage give insight in how to optimize the architecture for low energy consumption.

4 FIR DESIGNS WITH DIFFERENT TIME-AREA TRADE-OFFS

4.1 Introduction

To investigate the energy efficiency for FIR architectures with different time-area balances multiple designs are created. This chapter will first elaborate on the constraints and functionality that all these designs will have to adhere to. Then different designs are presented which range in parallelism: First, a fully parallel variable-tap FIR design. Then a design that uses only a single multiplier. Lastly, a method is proposed which can systematically create designs with an arbitrary number of multipliers. Thus creating the possibility to easily produce a range of designs. The result is a set of designs that computes the output of an n-taps FIR filter in 1 clock cycle, 2 clock cycles etc. up until n clock cycles.

4.2 System Constraints

Because of the collaboration with Axign, many of the system constraints for this research are based on the chip environment which is present in their products. A simplified overview of the audio system can be seen in Figure 4.1. In this figure the system constraints listed below in Section 4.2.1 are also displayed. Additionally, to reduce complexity a supply voltage is taken that is typical for the used IC technology. Therefore the possible benefits of voltage scaling as mentioned in section 3.2.1 are not explored.

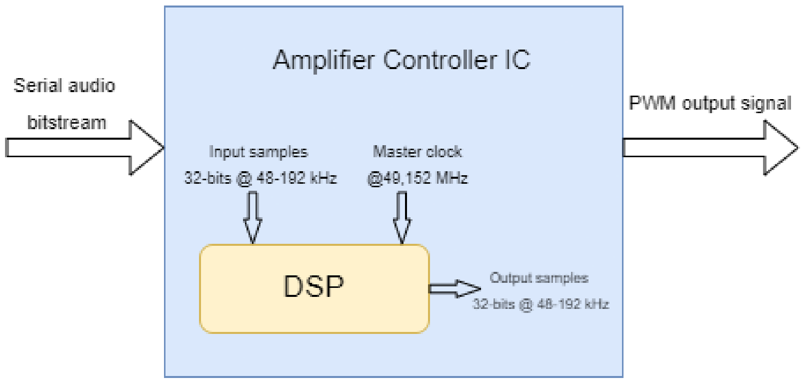


Figure 4.1: Simplified overview of the amplifier controller IC with the DSP as subsection. The controller IC provides 32 bit wide samples and a clock frequency to the DSP.

4.2.1 List of Constraints

- The input sample frequency, f_s is between $48kHz$ and $192kHz$
- The DSP core runs on a master clock frequency of $f_{clk} = 1024 * f_s(48kHz) = 49,152MHz$
- The input samples are 32-bit wide.
- The TSMC 55nm process is taken as a guideline for the IC technology used.
- A supply voltage is used that is typical for that IC technology.

4.2.2 Required FIR functionality

The FIR algorithm that needs to be executed is given in Equation 4.1. The output is computed by multiplying N delayed samples ($x[n-k]$) of the input with N coefficients (c_k) and summing the results.

$$u[n] = \sum_{k=0}^{N-1} c_k \cdot x[n-k] \tag{4.1}$$

One of the goals of the audio IC is to convert a digital bitstream to an analog signal with little latency, such that the chip can stream the audio near real-time. This goal also translates to the DSP, meaning that in many applications the DSP must finish processing the latest sample before a new sample arrives. With the constraints on system clock frequency and sample frequency imposed by the Axign system the simple timing diagram in Figure 4.2 is constructed. When a new sample arrives the system has a budget of 256 clock cycles in which a new output must be computed. How much of that budget is used differs per design. As typical FIR filter use case within the Axign IC a filter with around 100 delayed samples, or taps is considered. This filter can for instance be part of an interpolation setup. However, more exotic applications might require more taps. Being able to configure the number of taps used is a requirement for all designs.

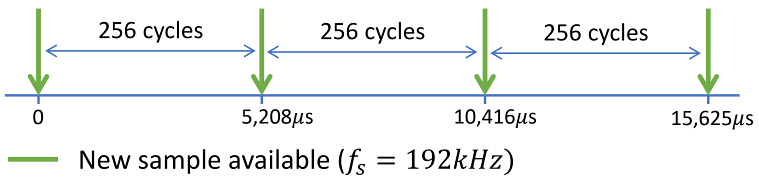


Figure 4.2: Timing illustration of the system displayed for the first three samples. New samples arrive at a maximum frequency of 192 kHz, while the system clock frequency is 49,152 MHz. Since an output sample must be available before the new sample arrives 256 clock cycles are available for the design to do all the computations.

4.2.3 Sample and Coefficient bit-width

Nowadays, a 24-bit resolution has become a standard for hi-fi audio sources. For DSP operations without significant quality loss, 32-bit arithmetic is typically required to have enough headroom[20]. Therefore all proposed designs use 24-bit samples and 24-bit coefficients casted into 32 bit registers. Both samples and coefficients are in a fixed point format with 1 integer bit and 31 fractional bits. A choice often made in audio since the output sample is then a fraction of the maximum output amplitude the amplifier can supply[20]. To reduce complexity, using floating-point arithmetic instead of fixed-point is a design choice that is not considered in this work. Energy efficiency in a floating-point DSP is a good subject for future research.

4.3 Fully Parallel FIR Design

A fully parallel design exploits the regularity aspect of an algorithm as mentioned in Section 3.6.2. By taking the regularity of the FIR algorithm into account an implementation is created with the least amount of overhead.
This design is rather straightforward as it consists of a delay line with a tap after each register. This tap feeds into a multiplier where the delayed sample is multiplied by a coefficient. All results are then accumulated by an adder tree and the output sample is ready. Since the design is fully parallel it needs only a single clock cycle to do all computations for that sample. After that cycle the design will remain idle until a new sample is fed in, which will trigger the delay line and move all the delayed samples one register further down the line.
The fully parallel design is a dedicated piece of hardware, with almost no flexibility to perform any other operation than FIR filtering.

4.3.1 Variable Tap FIR Design

The desired number of taps is not known beforehand. A maximum of 256 taps is chosen, as this number is a power of two and enough for many applications. The design is made configurable by adding the possibility to inject the input sample at any tap and disabling the registers before that tap. This effectively shortens the delay line that is operational to the desired length. This design is illustrated in Figure 4.3.

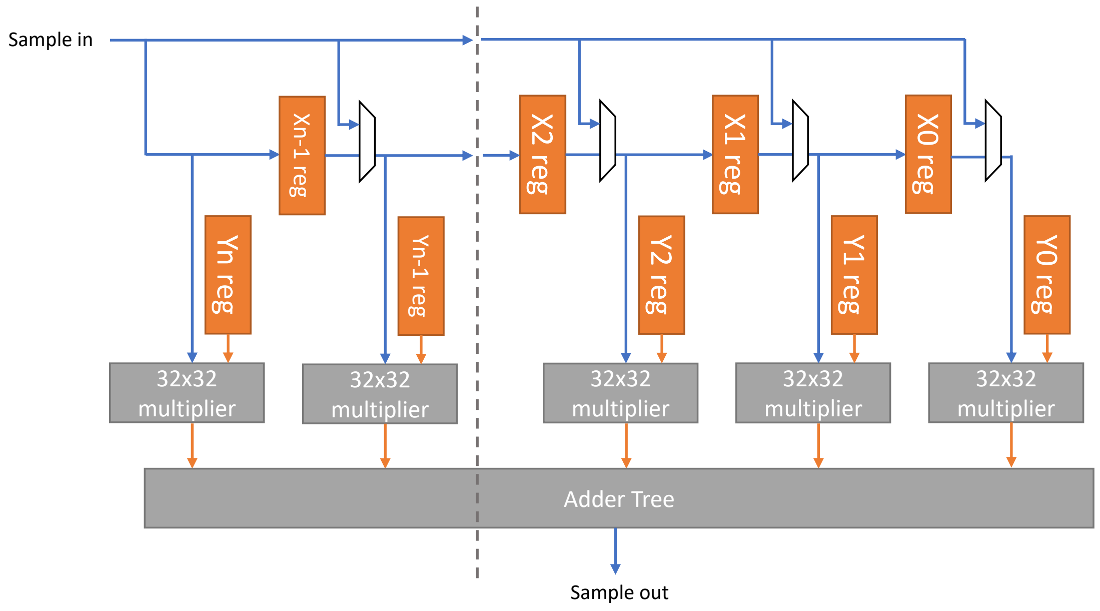


Figure 4.3: Schematic overview of the proposed parallel FIR filter implementation with a variable number of effective taps. The multiplexers enable the input sample to be injected at one of the taps. All registers to the left of that point are not enabled, preventing unwanted data outputs.

4.3.2 Hardware Description of Fully Parallel FIR Design

For the translation of a design concept to a format the synthesis tool can interpret, Clash is chosen. Clash (CAES Language for Synchronous Hardware) is a functional hardware description language that heavily borrows its semantics from Haskell. Clash can generate both VHDL/Verilog RTL designs while providing powerful abstraction mechanisms[1]. These abstraction mechanisms provide the ability to easily make adjustments to the time/area trade-off in a design.

The first aspects that are defined in the Clash design are the data types. Below a code snippet is shown defining the input samples, coefficients and output samples to be 32-bit wide in a signed fixed point format with 31 fractional bits.

```haskell
-- Define samples and coefficients to be 32 bit signed fixed point values with 31 fractional
    bits
type Data_reg = SFixed 1 31
type Coeff_reg = SFixed 1 31
type Sin = SFixed 1 31
type Sout = SFixed 1 31
-- Define number of taps input
type Num_taps = Int
-- Define coefficient and data memory banks chosen to be 256 32-bit registers
type Mem_coeff = Vec 256 Coeff_reg
type Mem_data = Vec 256 Data_reg
```

With the created data types defined, the next step is to define the arithmetic from Figure 4.3. The code below describes how the design should compute the new output and new state from the input and current state. The fully parallel FIR design checks every clock cycle whether the externally supplied frame trigger is high; When this is the case the new sample is injected into the tapped delay line and the taps and coefficients are multiplied and added together, creating the output value. When the frame trigger is low, the new state remains the same as the old state, effectively putting the design in an 'idle' mode.

```haskell
-- Top level function. Upon an external frame trigger (frame) the delay line is triggered and
    a new sample (sin) is injected at the tap which is indicated by the Num_taps input
fir_acc :: (Mem_data, Mem_coeff) -> (Sin,Bool,Mem_coeff,Bool,Num_taps) -> ((Mem_data,
    Mem_coeff), Sout)
fir_acc (regs, coeff_regs) (sin,tick,coeff,frame,num_taps) = case frame of
    True -> ((regs', coeff), sout)
    False -> ((regs, coeff), sout)
    where
        -- The regs' function shifts the delay line and inserts the new input at the correct
    place
        regs' =  replace (length(regs)-num_taps) sin (fst(shiftOutFromN d1 regs))
        -- The calc_out function is called upon to produce an output
        sout = calc_out num_taps coeff_regs regs sin

-- The output is calculated with entire delay line and coefficient bank, but enable signals
    should prevent unneccessary calculations from being done.
calc_out :: Num_taps -> Mem_coeff -> Mem_data -> Sin -> Sout
calc_out num_taps coeff regs sin = sum $ zipWith (*) coeff $ replace (length(regs)-num_taps-1)
     sin regs
```

To transfer data from one clock cycle into the next a function is defined that implements a mealy machine functionality. This function will take a clock signal, reset signal, enable signal and the arithmetic function "fir_acc" from the snippet above. The function will create the registers as described in Figure 4.3. 256 registers are always implemented, but by disabling a subset of these registers a FIR filter with a variable number of taps between 0 and 256 can be configured.

```haskell
-- An adjusted version of the clash mealy machine function. mm and its subfunctions xregs and
    yregs implement registers and define the default state, how the new state is calculated
    and how the enable signals (cVec) connect to the registers.
mm clk rst f cVec i = o
    where
      (s', o) = unbundle $ (f <$> s <*> i)
      (s1',s4') = unbundle s'
      s = bundle (s1,s4)
      s1 = xregs clk rst cVec s1'
      s4 = yregs clk rst cVec s4'

-- the xregs function defines the X-registers used, it is made a seperate function to be able
    to use the NOINLINE pragma, which will ensure that it becomes a seperate verilog entity
xregs clk rst cVec s1' = bundle $ zipWith (\en inp -> exposeClockResetEnable register clk rst
    en 0 inp) cVec (unbundle s1')

-- the yregs function defines the Y-registers used, it is made a seperate function to be able
    to use the NOINLINE pragma, which will ensure that it becomes a seperate verilog entity
yregs clk rst cVec s4' = bundle $ zipWith (\en inp -> exposeClockResetEnable register clk rst
    en 0 inp) cVec (unbundle s4')
```

The full clash code can be found in Appendix A.1.

4.4 Single Multiplier Design

In contrast to the fully parallel design, where the time-area trade-off is set completely to minimal time and maximum area, an architecture which is on the other side of the spectrum is also developed. This architecture is more similar to that of a classic CPU and a simple schematic can be found in Figure 4.4. Three memories are implemented. The "X" memory for all the samples, the "Y" memory for the coefficients and the "P" memory for the instructions that control the system. All memories are 256 words deep. The X and Y memory output their data in a register, after which that data is multiplied and accumulated in a register. Only a single sample and coefficient can be multiplied per clock cycle. After all the multiplications and accumulations are completed the value in the accumulation register can be propagated to an output register and becomes available at the output of the system.

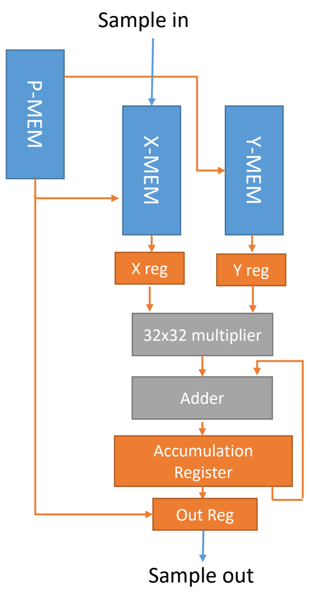


Figure 4.4: Basic schematic overview of the single multiplier design. Samples from the X-Mem and coefficients from the Y-Mem get multiplied and accumulated. When all the needed multiplications are accumulated, the data from the accumulation register is passed to the output register. Addressing and output register propagation is controlled by instructions stored in the P-Mem.

4.4.1 X Memory Adressing

To select the correct delayed sample and coefficient combination the correct address needs to be selected for the X and Y memories. For convolution algorithms such as FIR this combination is different for every multiplication. Figure 4.5 illustrates the address combinations needed for the first 3 output samples of a FIR filter with 5 taps. From this illustration two problems already become apparent when a FIR program needs to run for an indefinite amount of time: 1. The writing of new samples into the memory needs to be handled adequately, as the memory space is finite. 2. The number of unique X and Y address combinations needed to run the FIR algorithm rises quadratically with the number of taps. Meaning that if every instruction in the P-MEM stores one unique X and Y address combination, for a 100 taps FIR filter $100^2 = 10000$ instructions are needed, making the program memory very large and energy inefficient.

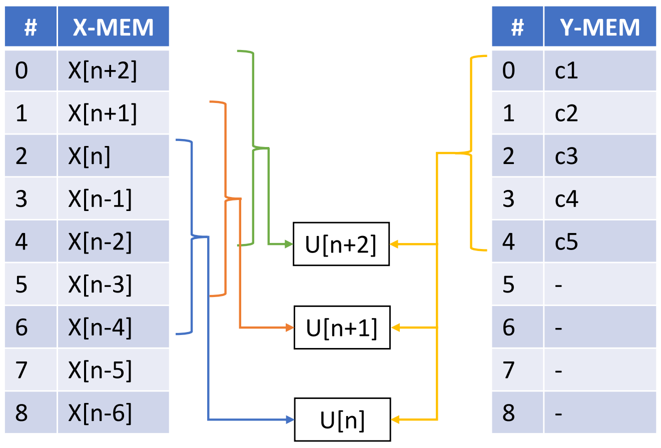


Figure 4.5: Illustration of the memory accesses needed to compute three outputs of a 5-tap FIR filter. The convolving nature of the algorithm can be seen, as for every new output the samples needed from the X-Mem shift one address upwards, but the Y-Mem addresses stay the same.

The solution for reducing the large number of instructions is a common technique used in processors

and uses the regularity of the FIR algorithm. All samples that are needed to calculate an output are always successively stored in the memory. Therefore calculating the absolute address from adding a base address and a relative address is a good solution for the data memory (X-MEM). The base address is stored in registers in the design and can be increased when a new output sample should be calculated. The relative address is still stored in the instruction. By applying this technique the number of needed instructions for a continuous N-taps FIR filter is reduced to N. At the end of the N cycles the base address is increased and the same program can be run again.
Another memory management technique applied to the X-MEM to enable efficient streaming behavior is that a so called circular buffer is created. This means that when a the calculated absolute address would fall outside of the possible range for the memory, say -1, the addressing wraps around and starts again at the back of the memory.
For writing an incoming sample to the X-MEM, a write address is stored in a register. This address can be increased by an instruction and wraps around when the address limit is reached.

4.4.2 Instruction Set

Aside from specifying the relative memory address discussed in the previous section the proposed instruction set can control the propagation of certain registers. A breakdown of a instruction can be found in Table 4.1. The instruction set enables the correct handling of inputs and outputs at the correct clock cycles and provides more flexibility than absolutely necessary for the FIR operation. For instance, programs can be written where consecutive instructions jump more than one address. This flexibility is not needed for a FIR design as all memory locations needed for the tapped delay line are stored consecutively and only a jump of a single address is used. This means that the design posses flexibility that is not exploited in this work but can be interesting for future work where slightly other algorithms are mapped onto the architecture.

# of bits	Function
1	Increase base address of X memory
1	Write data at the input to the X memory
1	End of program
log2(memory size)	Relative memory pointer
1	Add multiplier output to accumulator
1	Propagate accumulator value to output register

Table 4.1: Breakdown of all the fields in the DSP instruction. The instruction set is build to enable a straightforward FIR filter implementation, but by keeping the ability to specify any read address at any clock cycle more complex operations are possible in future work.

When the end of program bit is high, the program counter will stop and the design will remain idle until an external port signals that a new sample is available. This ”frame clock” will reset the program counter to zero and the program will run again.
Incorporating the information of the adressing and instruction set section a more elaborate architecture schematic can be made. This schematic can be found in Figure 4.6.

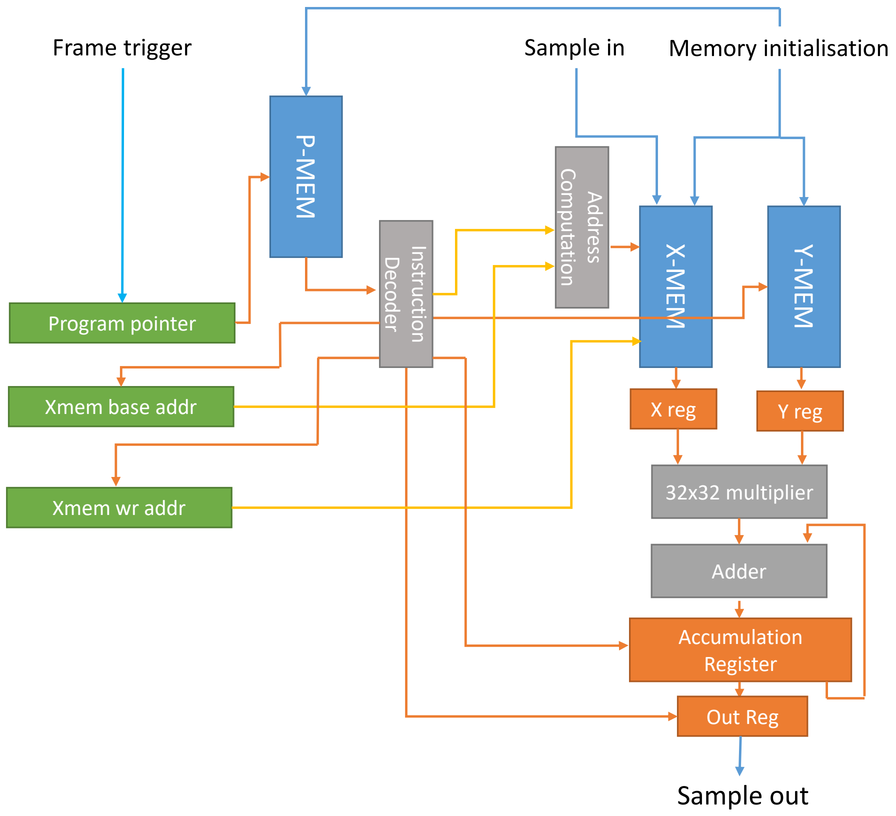


Figure 4.6: A more elaborate schematic of the single multiplier design. Here additional registers that enable relative addressing are shown in green. Using the base address and relative address information from the instruction absolute addresses are decoded. The frame trigger port can set the program pointer to 0, restarting the program. The memories can also be written through an external port, enabling programming of the design.

4.4.3 Hardware Description of Single Multiplier Design

The complete Clash code for the single multiplier design can be found in Appendix A.2. In this section the most important sections are elaborated upon.

Data Types

The single multiplier design requires more complex Clash code as there is much more control hardware needed. As with the fully parallel Clash description, first the data types are defined. The code snippet below shows again the sample and coefficient formats, but now also the memory addresses and additional registers needed to implement the design shown in Figure 4.6.

```
-- Define samples and coefficients to be 32 bit signed fixed point values with 31 fractional
    bits
type Sin = SFixed 1 31
type Sout = SFixed 1 31
type Coeff = SFixed 1 31

-- Set types for internal registers
type Xreg = Sin
type Yreg = Coeff
type Acc_reg = SFixed 1 31
type Out_reg = Acc_reg

-- Base memory pointer
type XMem_addr = Unsigned 8
-- Y memory address
```

```haskell
type YMem_addr = Unsigned 8
-- Program memory address, in this design this is equivalent to the program counter.
type PMem_addr = Unsigned 8
```

Instruction Set

The instruction set as described in section 4.4.2 is defined in the Clash design by the code below:

```haskell
---- INSTRUCTION SET -----
-- Base memory pointer increment instruction (+1 or +0)
type Xbase_inc = Bool
-- Write enable for Xmem
type Xwr_en = Bool
-- End of Program Boolean
type Prog_jump = Bool
-- Memory pointer (points to absolute Ymem location and relative Xmem location)
type Mem_pnt = Unsigned 8
-- Boolean that determines whether the data in the accumulation register will be passed to the
    output
type Outp_instr = Bool
-- Boolean that determines whether the accumulation register should be updated
type Acc_en = Bool
-- Construction of the total instruction
type Instr = (Xbase_inc,Xwr_en,Prog_jump,Mem_pnt,Outp_instr,Acc_en)
```

Top Level Function

With the data types defined the desired functionality can be implemented. The ”dsp” function in the code below is the top level entity of the design. Unlike the code for the fully parallel design the single multiplier code does not use a mealy machine function but implements the registers in the same function as the arithmetic. This can clearly be seen in the structure of the ”dsp6” function; It is a collection of registers preceded by the functionality needed to compute the new value that will be clocked into that register. To prevent cluttering, a substantial amount of the arithmetic has been defined in handling functions which are called upon by the ”dsp” function. Several of these handling functions are elaborated in the sections below.

```haskell
-- Top function, here all subfunctions are combined into a single entity
dsp :: Clk -> Rst -> En -> En -> Sig Sin -> Sig Frame_trig -> Sig (Maybe (PMem_addr, Instr))
    -> Sig (Maybe (YMem_addr, Coeff)) -> Sig (Maybe XMem_addr) -> Sig Sout
dsp clk rst en en_mac sin frame_trig p_in y_in ext_wr = sout
    where
        --Program counter section
        prog_cnt' = prog_cnt_handle <$> prog_cnt <*> (get3rd <$> instr) <*> frame_trig
        prog_cnt = (exposeClockResetEnable register clk rst en_mac) (0 :: PMem_addr) prog_cnt'
        --X memory base pointer section
        xbase_rd' = xbase_rd_handle <$> xbase_rd <*> (get1st <$> instr)
        xbase_rd = (exposeClockResetEnable register clk rst en_mac) (0 :: XMem_addr) xbase_rd'
        --X memory write pointer section
        xpnt_wr' = xpnt_wr_handle <$> xpnt_wr <*> (get2nd <$> instr)
        xpnt_wr = (exposeClockResetEnable register clk rst en_mac) (0 :: XMem_addr) xpnt_wr'
        --Instruction is fetched from program memory
        instr = pmem clk rst en prog_cnt p_in
        --X register section, the xmem function handles both writing to and reading from the X
     memory
        xreg' = xmem clk rst en xbase_rd xpnt_wr instr sin ext_wr
        xreg = (exposeClockResetEnable register clk rst en_mac) (0 :: Xreg) xreg'
        --Y register section, the ymem function handles both writing to and reading from the Y
     memory
        yreg' = ymem clk rst en instr y_in
        yreg = (exposeClockResetEnable register clk rst en_mac) (0 :: Yreg) yreg'
        --Accumulator register section. Depending on the instruction the values in xreg and
    yreg are multiplied and added to the accumulation.
        acc_reg' = acc_handle <$> xreg <*> yreg <*> acc_reg <*> (get6th <$> instr) <*> (get5th
     <$> instr)
        acc_reg = (exposeClockResetEnable register clk rst en_mac) (0 :: Acc_reg) acc_reg'
        --Output register section. Depending on the instruction the value in the accumulator
    register is passed to the output register or not.
```

```
        sout = (exposeClockResetEnable register clk rst en_mac) (0 :: Out_reg) sout'
        sout' = calc_out <$> (get5th <$> instr) <*> sout <*> acc_reg
```

Program Counter

The program counter is used to control what instruction from the program memory is executed when. Other than increasing every clock cycle the program counter is also manipulated to start or stop the entire design from computing outputs. When the external frame trigger signal is high, indicating the arrival of a new sample, the program counter is set to zero, effectively restarting the program. When the end of program (prog_jump) bit of the instruction is high, the program counter is set to the last address of the program memory. At this address an instruction is present which disables the accumulation and output register from updating. When the last address of the program memory is reached the program counter will not update until a new frame trigger is observed, effectively putting the hardware in an idle mode.

X, Y and P Memory

As mentioned in section 3.4, an SRAM design with latch cells is an implementation often used in the industry. Often designs are available that have been optimized for minimal energy consumption. This IP is typically supplied as a hard macro, which means the chip layout has already been fully determined and optimized. Unfortunately, industry standard optimized SRAM IP blocks were not readily available. To model the SRAM behaviour in this work a design is used constructed of cells from the standard library of the used IC technology. This IP constructed from standard library cells is implemented through the Synopsys Designware tool.
To incorporate this Designware SRAM IP in the Clash design a so-called primitive is constructed. This primitive is a description of how the Clash tool should generate a Verilog description when a certain function is called. In this case, the primitive is used for the asyncRAM function, resulting in a Verilog instantiation of the Designware SRAM IP. The primitive description can be found in Appendix B.1.
The Clash code instantiating the memories can be seen below. The asyncRAM function, which is a native Clash function, requires an input that indicates the address which should be read from, whether there is a write input and if so, what the write address and value of that write input is. In the description for the X memory (xmem) the calculation of the absolute read address from the base address and memory pointer can be seen. In the single multiplier design underflow of the 8-bit unsigned address type is used to make the address wrap around.

```
-- Instantiation of the program memory, the asyncRamPow2 function is used. This function is
    native to Clash and implements an asynchronous read, synchronous write RAM.
-- The bit-depth of the program memory automatically scales with the defined instruction set.
    The word depth is dependant on the size of the PMem_addr, which leads to a program memory
    of 256 words.
pmem :: Clk -> Rst -> En -> Sig PMem_addr -> Sig (Maybe (PMem_addr,Instr)) -> Sig Instr
pmem clk rst en prog_cnt p_in = (exposeClockResetEnable asyncRamPow2 clk rst en) prog_cnt p_in

-- Instantiation of the X memory, here the absolute memory address for reading is derived from
     the base address and the memory pointer section of the instruction.
  -- The write handle function is called to obtain the absolute write address when a value
    needs to be written to the X memory
xmem :: Clk -> Rst -> En -> Sig XMem_addr -> Sig XMem_addr -> Sig Instr -> Sig Sin -> Sig (
    Maybe XMem_addr) -> Sig Xreg
xmem clk rst en baddr xpnt_wr instr sin ext_wr = (exposeClockResetEnable asyncRamPow2 clk rst
    en) raddr wrinp
  where
        raddr = (-) <$> baddr <*> (get4th <$> instr) -- Relies on underflow to wrap around and
     start at the bottom address when (baddr - mem_pnt) becomes negative
        wrinp = wr_handle <$> (get2nd <$> instr) <*> xpnt_wr <*> sin <*> ext_wr -- writes sin
    to memory when instruction tells it to or when external write is a (Just addr)

-- Instantiation of the Y memory. Like the X memory, the Y memory can be written to via an
    external write port. That is however the only method to write to the Y memory.
-- The Y memory is not circular and does not use relative memory pointers, so the read address
     is simply the memory pointer section from the instruction.
ymem :: Clk -> Rst -> En -> Sig Instr -> Sig (Maybe (YMem_addr, Coeff)) -> Sig Yreg
ymem clk rst en instr y_in = (exposeClockResetEnable asyncRamPow2 clk rst en) raddr y_in
    where
       raddr = get4th <$> instr
```

X Memory Base Pointer and Write Pointer

The base pointer contains the information of where a program should start in the X memory and can be increased by the base_inc section of the instruction. Additionally, the pointer should wrap back to zero when the address limit is reached. The write pointer indicates where a new sample should be written to in the X memory and has a nearly identical behaviour to the base pointer. The only difference is that the write pointer is increased when the Xwr_en section of the instruction is high instead.

Multiply-Accumulate Function

The computation of the new value in the accumulator register is given in the code snippet below:

```
-- MAC function:
  -- When the Outp_instr part of the instruction is True the output of the accumulator is sent
     to the output port and the accumulation register is set to 0.
  -- When the acc_en section of the instruction is True, multiply the values in the X and Y
    registers and add the result to the accumulator register
  -- When the acc_en section of the instruction is False, do not update the value in the
    accumulator register
acc_handle :: Xreg -> Yreg -> Acc_reg -> Acc_en -> Outp_instr -> Acc_reg
acc_handle xreg yreg _ _ True = (0 :: Acc_reg)
acc_handle xreg yreg acc_reg True _ = xreg * yreg + acc_reg
acc_handle _ _ acc_reg False _ = acc_reg
```

There are three possible situations depending on the instruction. When the Outp_instr section of the instruction is True the value in the accumulator is propagated to the output register and the accumulator register is reset to zero. When the accumulation is enabled by the Acc_en section of the instruction the output of the multiplication of the values in the X and Y registers is added to the existing value in the accumulation register. If the Acc_en section of the instruction is False the accumulation register retains its previous value.

4.5 n-multipliers Design

Analyzing the energy efficiency of the fully parallel and single multiplier designs gives information about two edge cases of the time-area trade off for a FIR filter design. This work also explores the design space in between these two cases. An adjustable architecture is defined that can generate a design with $1 < n < 256$ multipliers. A design with n multipliers also consists out of n sets of X and Y memories. Additional controlling hardware and a barrel shifter are added to ensure the correct matching of samples and coefficients and calculate the correct result. A schematic can be found in Figure 4.7. Effort has been made to keep the control mechanisms in the designs close to the single multiplier architecture. For instance, there is still only one memory pointer that now addresses n sets of memories instead of a single set. This means that when n increases the design becomes more dedicated, where one instruction always instructs n multiplications and programming other behaviour then a FIR filter becomes more difficult.

4.5.1 Data and Coefficient memories

Instead of all samples and coefficients each being stored in a 256 word memory, now all the data is divided over n memories with $\frac{256}{n}$ words each. The process of matching the right sample in the X memories with the right coefficient in the Y memories becomes more intricate. Also the placing of new input samples in the correct memory needs to be managed. The choice is made to store all the data interleaved in the memories as is illustrated in Figure 4.8.

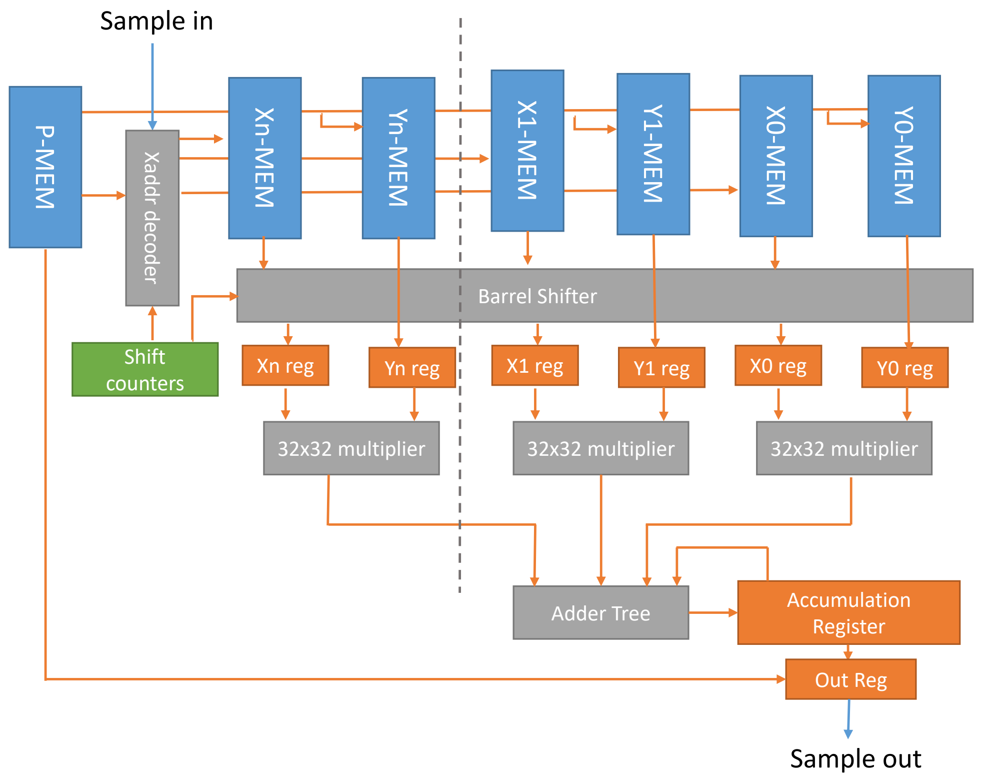


Figure 4.7: A schematic overview of the n-multipliers design. By adding an internal counter and a barrel shifter that scale with the number of memories and multipliers the correct output is calculated. The generalized template enables quick creation of many designs, each with a different level of parallelism.

#	X0-MEM
0	s_0
1	s_{n+1}
2	s_{2n+1}
3	s_{3n+1}

#	X1-MEM
0	s_1
1	s_{n+2}
2	s_{2n+2}
3	s_{3n+2}

#	X2-MEM
0	s_2
1	s_{n+3}
2	s_{2n+3}
3	s_{3n+3}

#	Xn-MEM
0	s_n
1	s_{2n}
2	s_{3n}
3	s_{4n}

#	Y0-MEM
0	c_0
1	c_{n+1}
2	c_{2n+1}
3	c_{3n+1}

#	Y1-MEM
0	c_1
1	c_{n+2}
2	c_{2n+2}
3	c_{3n+2}

#	Y2-MEM
0	c_2
1	c_{n+3}
2	c_{2n+3}
3	c_{3n+3}

#	Yn-MEM
0	c_n
1	c_{2n}
2	c_{3n}
3	c_{4n}

Figure 4.8: Illustration of the interleaved sample and coefficient storage over multiple memories. For simplicity only the first 4 addresses of each memory are shown. In practice each memory contains a number of words equal to $\frac{256}{n}$, rounded upwards.

4.5.2 Matching sample and coefficient data

With the data stored interleaved in the memories and only a single memory pointer, additional control is needed to propagate the matching sample and coefficient data to the input of the multipliers. Again making use of the regularity of the FIR algorithm, a combination of a counter and a barrel shifter implement the correct behavior. These components can be seen placed in Figure 4.7. The barrel shifter shifts entire 32-bit samples from the X memories to the left until the correct combination of sample and coefficient is made.

4.5.3 Hardware Description of n Multiplier Designs

For generating a range of designs with a different amount of multipliers a shell script is written. This script creates separate Clash files for each design and ensures that every design has a unique name. This script can be found in Appendix C.2. The generated Clash code for $n = 2$ can be found in Appendix A.3 and is elaborated upon in this section. Aside from the names of the modules and the type definition ”Num_mem” given in the code snippet below the Clash code for $n = 2$ is identical to that of every other n.

```
type Num_mem=2
```

By making the memory sizes, amount of MAC hardware, memory pointers and barrel shifter size dependant on this Num_mem type definition the ”knob” is created with which the time-area trade-off is controlled.

Data Types

The type system of Clash enables the creation of types that have a dependency on other types. The data types used for the n-multiplier designs are very similar to that of the single multiplier design, but the memory addresses are now defined to fit $\lceil \frac{256}{n} \rceil$ addresses. The definition is shown in the code snippet below, where the Index data type is used. When generating Verilog, Clash will translate the Index data type to an unsigned number using the least amount of bits necessary. The program memory remains the same size for every design, containing 256 words.

```
-- Base memory pointer, each memory will have 256/num_mem addresses, rounded upwards
type XMem_addr = Index (DivRU 256 Num_mem)
-- Y memory address, each memory will have 256/num_mem addresses, rounded upwards
type YMem_addr = Index (DivRU 256 Num_mem)
-- Program memory address, for every design a 256 word program memory is used.
type PMem_addr = Unsigned 8
```

Having variable-size memory pointers also means the memory pointer in the instruction becomes variable, as defined in the code snippet below.

```
---- INSTRUCTION SET -----
-- Base memory pointer increment instruction (+1 or +0)
type Xbase_inc = Bool
-- Write enable for Xmem
type Xwr_en = Bool
-- End of Program Boolean
type Prog_jump = Bool
-- Memory pointer (points to absolute Ymem location and relative Xmem location)
type Mem_pnt = Index (DivRU 256 Num_mem)
-- Boolean that determines whether the data in the accumulation register will be passed to the
     output
type Outp_instr = Bool
-- Boolean that determines whether the accumulation register should be updated
type Acc_en = Bool
-- Construction of the total instruction
type Instr = (Xbase_inc,Xwr_en,Prog_jump,Mem_pnt,Outp_instr,Acc_en)
```

Reading the X memories

With multiple X memories but still one base address and one memory pointer addressing the correct addresses in the X memories becomes more complex. As mentioned in Section 4.5.1, the data is stored interleaved in the X memories. Because of this mapping and the delay line nature of the FIR algorithm the required absolute addresses are always either $baseaddress - memorypointer$ or $baseaddress -$

$memorypointer - 1$. This fact is also shown in Figure 4.9, where the memory usage is shown for a design with 3 memories and a 6 tap FIR filter implementation. To implement the desired behaviour a bit vector of n bits is created. Each bit in this bit vector indicates whether that corresponding X memory should be addressed with $baseaddress - memorypointer$ or $baseaddress - memorypointer - 1$. The computation of that bit vector is performed by the Clash code below. Every time a full FIR cycle is performed and an output sample is computed a new memory select bit vector is computed.

```haskell
--Handling function for the X memory address offset. The resulting vector indicates which
    memories should be read at (base address - mem pointer) and which should be read at (base
    address - mem pointer - 1).
  -- The vector only changes when the Xbase_inc section of the instruction is True
  -- Bitwise operations are used to incrementally shift in 0's until all bits in the vector
    are 0's, then the vector resets to all 1's except for the first bit. example for num_mem =
     4:
  -- cycle1: (0 1 1 1,F)
  -- cycle2: (0 0 1 1,F)
  -- cycle3: (0 0 0 1,F)
  -- cycle4: (0 0 0 0,T) -> By setting the second argument of the tuple to true the base
    pointer is increased for next cycle
  -- cycle5: (0 1 1 1,F) -> Cycle starts again but base pointer is now increased
f_mem_select_rd ::  Vec Num_mem Bit -> Xbase_inc -> (Vec Num_mem Bit,Bool)
f_mem_select_rd  vb_in (False) = (vb_in,False)
f_mem_select_rd  vb_in (True)  = (((0 :> Nil) ++ (tail vb_new)),(bitToBool (head vb_new)))
    where
        vb_new = map (or## (complement## (last vb_in))) shift
        shift = fst(shiftOutFromN d1 vb_in)
```

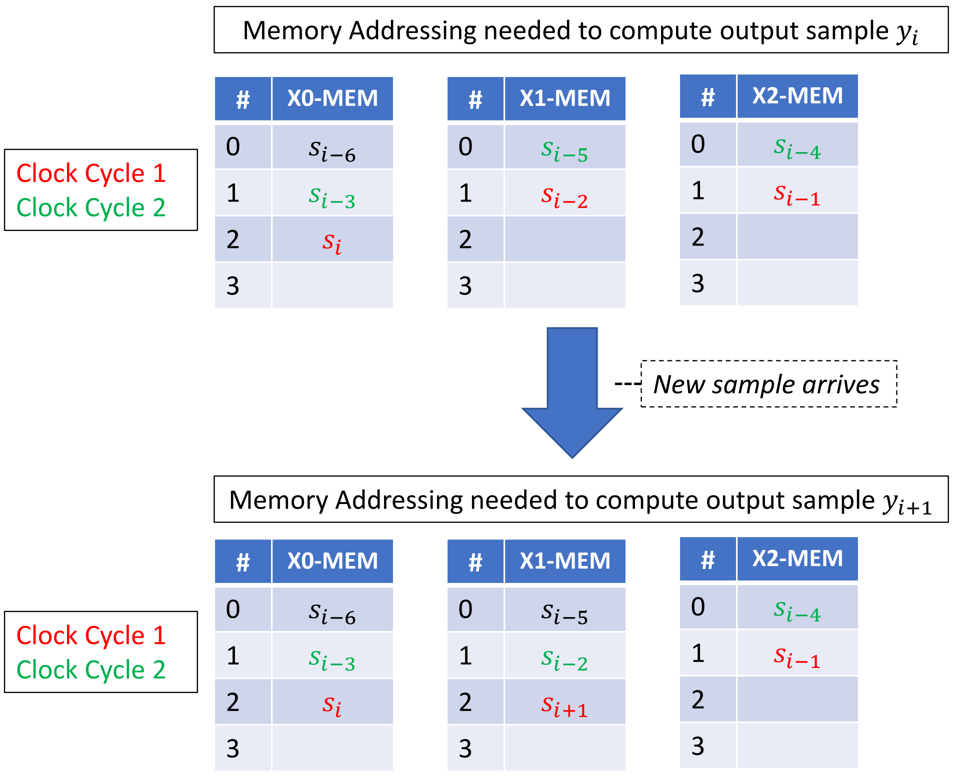


Figure 4.9: Illustration of the memory addressing needed to compute two output samples of a 6 taps FIR filter using a 3-multiplier design. In the figure it can be seen that every clock cycle the samples are taken from a specific row (a) or the row before that ($a-1$). Which of the three memories should be addressed by row a or $a-1$ changes every time a new sample is added to the memories. In this example, for the computation of output sample y_i two clock cycles are needed where the addresses per memory for each clock cycle are a for memory X0, $(a-1)$ for memory X1 and $(a-1)$ for memory X2. When a new sample arrives and output sample y_{i+1} needs to be computed, the addresses per memory change to a for memory X0, a for memory X1 and $(a-1)$ for memory X2. To obtain the addressing behaviour as described in this figure a decoding step is introduced which computes the correct read address for each X memory from the base address, the memory pointer and a newly introduced counter.

The X memory base address is increased every n output samples. This is in contrast with the single multiplier design, where this would happen every output sample.
With the memory select vector, the base address and the memory pointer section of the instruction the absolute read addresses for the X memories can be computed. This is done in the Clash code below.

```
--Handling function used to generate absolute addresses for the X memories using the base
    address, instruction and memory select bitvector
  -- The f_add_base_mem_select handling function is mapped over the bitvector creating a
    vector of Num_mem absolute adresses
  -- The f_sub_base_mem_pnt handling function calculates base_addr - mem_pnt with overflow
    handling
x_rd_addrs_handle :: XMem_addr -> Vec Num_mem Bit -> Instr -> Vec Num_mem XMem_addr
x_rd_addrs_handle xbase_rd mem_select instr = map (f_add_base_mem_select a) mem_select
    where
      a = f_sub_base_mem_pnt xbase_rd mem_pnt
      mem_pnt = get4th instr

--Handling function to calculate absolute memory address for one memory by either passing
    through (base_addr - mem_pnt) when the bit from the bitvector is zero or (base_addr -
    mem_pnt - 1) when the bit in the bitvector is one. Additionally, handle proper wrap around
     behaviour when (base address - mem_pnt) is zero.
f_add_base_mem_select :: XMem_addr -> Bit -> XMem_addr
f_add_base_mem_select xbase_rd 0 = xbase_rd
f_add_base_mem_select 0 1 = (maxBound::XMem_addr)
f_add_base_mem_select xbase_rd 1 = xbase_rd-1

--Handling function to calculate base address - memory pointer with overflow handling.
  -- Subtracts the memory pointer part of the instruction from the base address and ensures
    proper wrap around behaviour.
f_sub_base_mem_pnt :: XMem_addr -> Mem_pnt -> XMem_addr
f_sub_base_mem_pnt baddr mem_pnt
    | baddr < mem_pnt = (maxBound :: XMem_addr) - (mem_pnt-baddr - 1)
    | otherwise = baddr - mem_pnt
```

Writing to the X memory

A new sample that arrives must be written to one of the X memories. An internal counter is defined that indicates which of the n X memories should be written to. This counter is increased every write operation and will wrap around to zero once the maximum is reached. Additionally, when the maximum of the internal counter is reached, the write pointer is increased. By implementing the writing in this fashion the interleaved sample storage as shown in Figure 4.8 is achieved. The corresponding Clash code for this behaviour is given below:

```
-- Write select counter which will indicate which memory to write new samples to
type Wr_sel = Index Num_mem
```

```
--Handling function for the computation of the X memory write select variable
-- When the Xwr_en section of the instruction is False, do not update write select variable
-- When the Xwr_en section of the instruction is True, increase the write select variable and
    when the maximum is reached, set write select variable to zero and set the boolean in the
    second part of the tuple to True to indicate that the write pointer should be increased.
f_mem_select_wr :: Wr_sel -> Xwr_en -> (Wr_sel,Bool)
f_mem_select_wr wr_sel False = (wr_sel,False)
f_mem_select_wr wr_sel True
      | wr_sel == (maxBound::Wr_sel) = ((0 :: Wr_sel),True)
      | otherwise = (wr_sel + 1,False)

--Handling function which creates a vector of Write instructions in the format required by the
     asyncRam function.
  -- When the Xwr_en section of the instruction is True create a num_mem wide vector with 1
    element filled with the write address and new sample.
  -- Which element in the vector contains the new sample is determined by the Wr_sel value,
    using an imap to replace the value in the vector at a certain index.
x_wr_inps_handle :: XMem_addr -> Wr_sel -> Sin -> Xwr_en -> Vec Num_mem (Maybe (XMem_addr,Sin)
    )
x_wr_inps_handle _ _ _ False = replicate (SNat @ Num_mem) Nothing
x_wr_inps_handle wr_base wr_sel sin True = f_rep_ind wr_sel (Just (wr_base,sin)) (replicate (
    SNat @ Num_mem) Nothing)
    where
      f_rep_ind wr_sel a vec = imap (\i vec -> if i == wr_sel then a else Nothing) vec
```

Barrel Shifter

A barrel shifter is implemented to shift the samples that are obtained from the X memories with the correct coefficients from the Y memories. After every computation of an output sample and thus after every full FIR cycle a counter is increased. This counter indicates how many ”places” the barrel shifter should shift the samples from the X memories to match the correct coefficients.
The barrel shifter is implemented by multiple shifter stages and a shift counter. The structure can be seen in Figure 4.10. Every stage shifts the samples by a power of two and is turned on or off by the shift counter.

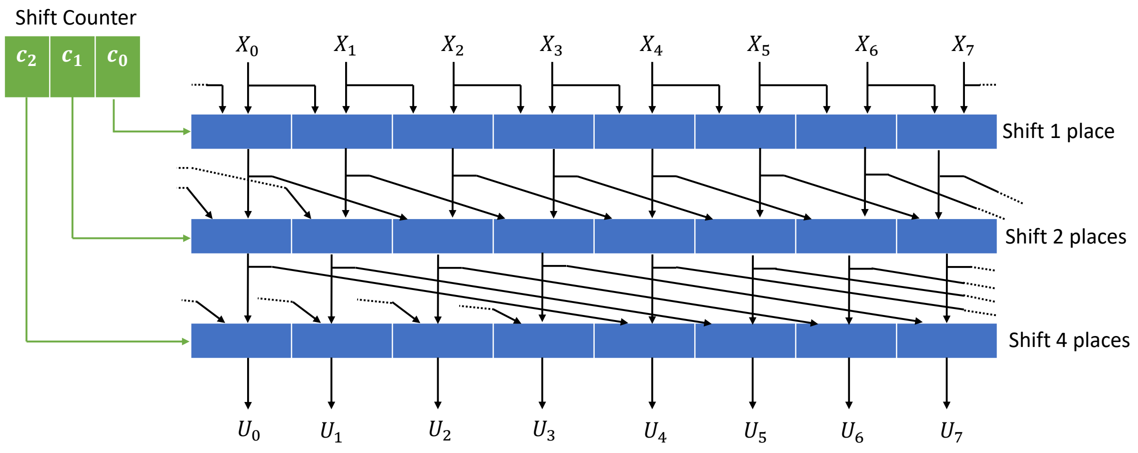


Figure 4.10: Illustration of the barrel shifter implementation. Each shifter stage shifts the input sample a power of two places to the right. The shift counter controls which of the stages perform a shift or not. This structure enables 2^{stages} possible shifts.

For an n-multiplier design a barrel shifter with $\log_2(n)$ stages are needed and thus designs with a smaller n only need fewer stages. To implement the structure in Clash a description is made for 8 barrel shifter stages. With 8 stages the barrel shifter structure can be created for $n \leq 256$. The description for stage 1 through 3 is given below.

```
-- Barrel shifter functions for DSP implementations with multiple memories. To be able to
    combine different static barrel shifters multiples of the same function description are
    necessary
-- but each one is called with a different SNat for the amount rotated. Since this requires a
    different type for each function unique function names are needed.
-- the functions are instantiated by a line in a DSP_program file generated by Matlab
    depending on the amount of barrel shifting needed.

-- Up to 8 unique barrel shifters are supported

vec_shft_1 :: KnownNat n => Bool -> (Vec n a1,SNat 0) -> (Vec n a1,SNat 1)
vec_shft_1 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_1 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d1

vec_shft_2 :: KnownNat n => Bool -> (Vec n a1,SNat 1) -> (Vec n a1,SNat 2)
vec_shft_2 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_2 True (vec_in,pow)  = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d2

vec_shft_3 :: KnownNat n => Bool -> (Vec n a1,SNat 2) -> (Vec n a1,SNat 3)
vec_shft_3 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_3 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d4
```

Since not all stages are needed for every design, a unique instantiation function is generated for every n-multiplier design. The instantiation function for $n = 2$ is given below. This function can take the outputs of the X memories and the shift counter variable and will produce a vector with the correctly shifted samples.

```
-- Barrel shifter function generated by MATLAB, with log2(Num_mem) stages.
```

```
barrel_shift :: Vec Num_mem Xreg -> Shft_cnt -> Vec Num_mem Xreg
barrel_shift vec_in cnt = fst $ ((vec_shft_1 (shft_en !! 0))) (vec_in,d0)
    where
        shft_en = reverse (to_bool_vec cnt)

--Barrel shifter help function
to_bool_vec :: Shft_cnt -> Vec (CLog 2 Num_mem) Bool
to_bool_vec cnt = unpack (pack cnt) :: Vec (CLog 2 Num_mem) Bool
```

5 METHOD OF ANALYSIS

To evaluate all proposed designs an analysis method is set up which simulates each design and records the energy consumption. A high level overview of this method is given in Figure 5.1. The goal is to obtain an estimate of a design's energy usage if this design would be manufactured on an IC.
The Clash hardware descriptions as shown in sections 4.4.3, 4.3.2 and 4.5.3 are synthesized to create a Verilog description of the eventual IC hardware. A testbench is created to shape an environment comparable to the on-chip situation and provide representative inputs to the design. The design's switching behaviour is recorded and used to create an estimate of the power usage over time, from which the energy consumption is derived.

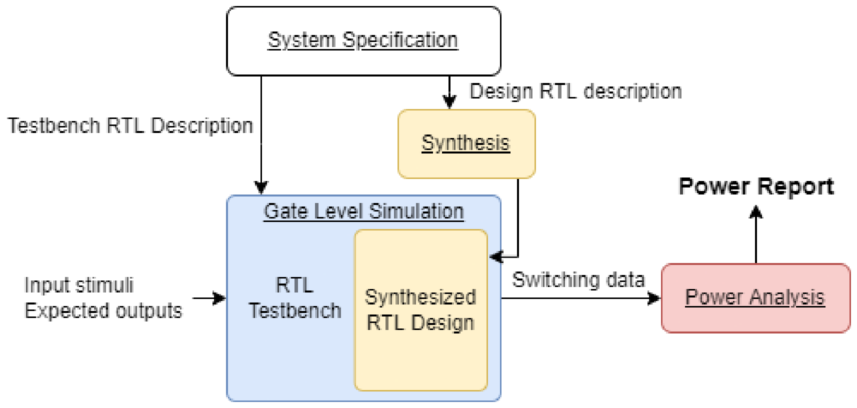


Figure 5.1: High level overview of the method of analysis.

The following sections will elaborate on the different steps of the analysis process and how they are implemented in the tools. This will lead to Figure 5.3 where a schematic overview of the used software tools and how they interact is given.

5.1 Generation of Hardware Description

The Clash tool is instructed to generate Verilog HDL from the code described in sections 4.4.3, 4.3.2 and 4.5.3. Important to note here is that the primitive from appendix B.1 ensures that when the Verilog RTL for the X,Y and P memories are generated the Designware memory IP is instantiated. This is the memory constructed from standard library cells that is used to model SRAM in the designs.

5.2 Logic Synthesis

The synthesis process is the next step in creating an actual chip. For this, the synthesis tool requires the Verilog RTL descriptions produced by Clash and a library file that contains all the possible components that are available in that specific IC technology. The tool is controlled by a script that can set design constraints. The tool maps the functionality described in the RTL description to a netlist consisting of cells that are possible to fabricate in that specific IC technology. This netlist is a more accurate description of the eventual IC and is used as an improved model to perform power analysis.
Through the Europractice initiative the Design Compiler of Synopsys[21] is available. The Synopsys toolset is industry proven software and a very large player in the IC EDA tool industry.

Preferably a library for the 55nm TSMC technology would be used, as this is the technology used by Axign. This library was however not readily available and thus the 65 UMC library is used. The assumption is made that a representative comparison between the different designs can still be made. This assumption is made because the 55nm technology is presented as a half node shrink of the 65nm process. This implies that there are few changes aside from the size of the transistors. The assumption is made that the lower energy consumption of the 55nm technology will apply to all designs equally. The exact difference is a good subject for future studies.
Aside from delivering a more accurate IC model of the desired design, the synthesis tool also gives a preliminary chip area report. A more exact area indication would be available after the netlist would have been placed and routed, but the preliminary report already gives a good indication of the size.

5.2.1 SRAM implementation

As discussed in section 3.4, on-chip SRAM is an extremely popular form of addressable memory and is often highly optimized for minimal area and power usage. The SRAM is often provided in the form of a hard macro. This means that this specific piece of the chip is already fully laid out and considered a ”black box” by the synthesis tools.
Because the SRAM is such a delicate and optimized design it is not included as a standard cell in the UMC65nm library. Unfortunately, no representable SRAM IP was easily available. As an alternative a piece of IP is used provided by the Synopsys DesignWare library. This IP generates a design from standard library components that is functionally equivalent to typical SRAM, but benefits less from the area and power optimizations. The consequence is that the results presented in this work are a less accurate representation of industrial IC performance, but still relevant. A more detailed indication of the resulting difference in energy and chip area is difficult to obtain with the limited time available, as energy figures for the many different sizes of SRAM in the correct IC technology are not easily available, and differ significantly not only per IC technology, but also per supplier.

5.2.2 Multiplier Generation

Section 3.3 discussed digital hardware architectures for performing multiplication. In this work, the choice and exact implementation of the multipliers is left to the Synopsys Design Compiler. The synthesis tool builds the 32-bit multipliers from a combination of smaller multipliers and adders that are present in the standard cell library. The tool performs several optimization runs to ensure timing is met and the least amount of hardware is used. In future work, investigations could be conducted to gain insight in how these multiplier optimization runs effect the energy efficiency.

5.3 RTL Testbench

A schematic of the testbench functionality can be seen in Figure 5.2.

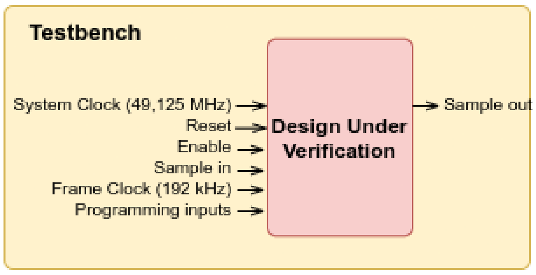


Figure 5.2: Overview of the RTL testbench.

The RTL testbench must provide the FIR design with all the stimuli that it needs for correct operation. Additionally, the testbench will ”program” the design when necessary such that coefficients and initial memory values can be set. The testbench also compares the output samples of the design to expected values, thus verifying the functional correctness of the design under verification. To obtain a fair comparison between different implementations the testbench is kept almost identical for each design.

The testbench is designed in Clash, which is then compiled into verilog RTL. Both the design and the testbench are loaded into ModelSim, the HDL simulator developed by Mentor Graphics[22]. ModelSim will simulate the design in the testbench and monitor the switching behavior of the design under verification. This switching behaviour will be written to a value change dump (vcd) file. To not take startup behaviour into account the switching behavior is not recorded until the program, all coefficients and initial values are loaded into the design.

5.4 Samples, Coefficients, Programs and Expected Output Generation

For every testbench/design input stimuli are generated, programs are compiled and expected outputs are computed. In this work Matlab is chosen for this task because of the authors familiarity with the tool. For the input signal random values are used to represent an audio bitstream. As mentioned in section 4.2.3 values between -1 and $1 - 2^{-24}$ are generated with a precision of 24 bits. These values are used for the input sample stream and coefficients. In the Matlab script the number of FIR taps and the number of memories can be configured. The script then generates a set of instructions. All the generated data is written to a haskell file in the correct format such that the Clash tool can interpret the data and create a testbench.

5.5 Power Analysis

When the switching behavior of the design is obtained, Synopsys PrimeTime[23] is used to generate a power report. The 65nm UMC technology library includes power models of all the used cells in the synthesized netlist. This information, together with the switching behavior enables the tool to create a report on the power usage. This report contains the average power consumption over multiple samples and not the peak power consumption. When the total energy consumption (in Joules) over a certain time is desired, the reported power consumption can then be obtained by multiplying the power figure with that certain time. Similar to the Synopsys Design Compiler, this tool is controlled with a constraint file.

5.6 Tool Overview

In Figure 5.3 a schematic overview can be seen of all the tools used to obtain the reports on area and power consumption of one design. The annotated arrows indicate what files each tool produces or consumes.

5.7 Automatically Generating and Evaluating n-multiplier Designs

Using the generalized Clash design, input generation, synthesis script and power analysis scripts an overarching script is written which will automatically generate the needed files for all desired n-multiplier designs. This enables the user to set a range of parallel multipliers used, say 2 to 32, and perform the power analysis method described above on all designs. All generalized scripts and the Clash design can be found in Appendix C.

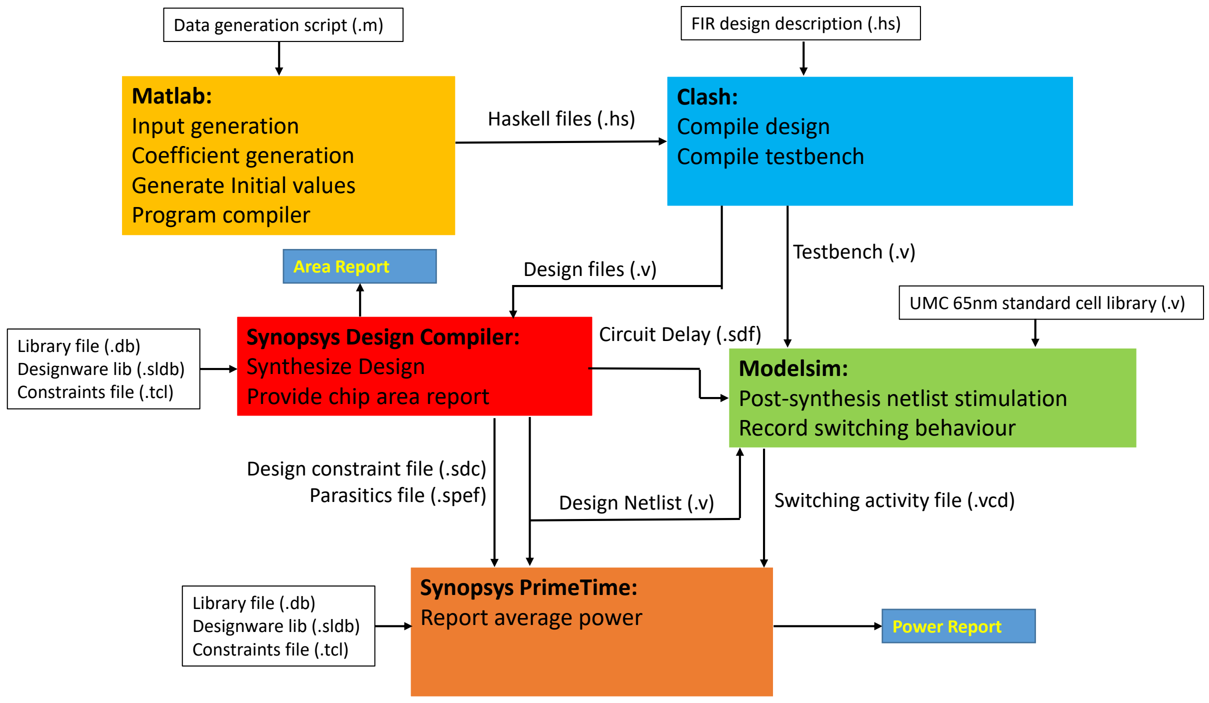


Figure 5.3: Schematic overview of the tools used to obtain area and power consumption reports. The arrow annotations indicate the type of file that is consumed by the corresponding tool.

6 RESULTS

The analysis method presented in section 5 is used to obtain the average power consumption of the designs presented in section 4. All designs are stimulated by a testbench that initializes the designs and configures a FIR filter with 100 taps. The average power consumption is determined over at least 100 consecutive input samples. The input samples arrive at a frequency of 192 KHz and the average power consumption is computed over the entire runtime of the testbench, thus including both the consumption when a design is actually computing a new output and when the design is idle.
By synthesizing a design an estimation of the needed chip area is also obtained. To gain further insight into the energy consumption per function categories are defined in the Clash description. By annotating sub-functions in the Clash description a structure is created that remains in place throughout the verilog generation, the synthesis process, and power analysis. These categories are then also present in the generated power reports and thus the energy consumption per category is determined.

6.1 Fully Parallel Design

The categories that are constructed in the fully parallel design are shown in Figure 6.1. By creating the categories in this fashion an idea can be obtained for the energy consumption ratio between arithmetic operations and memory access.

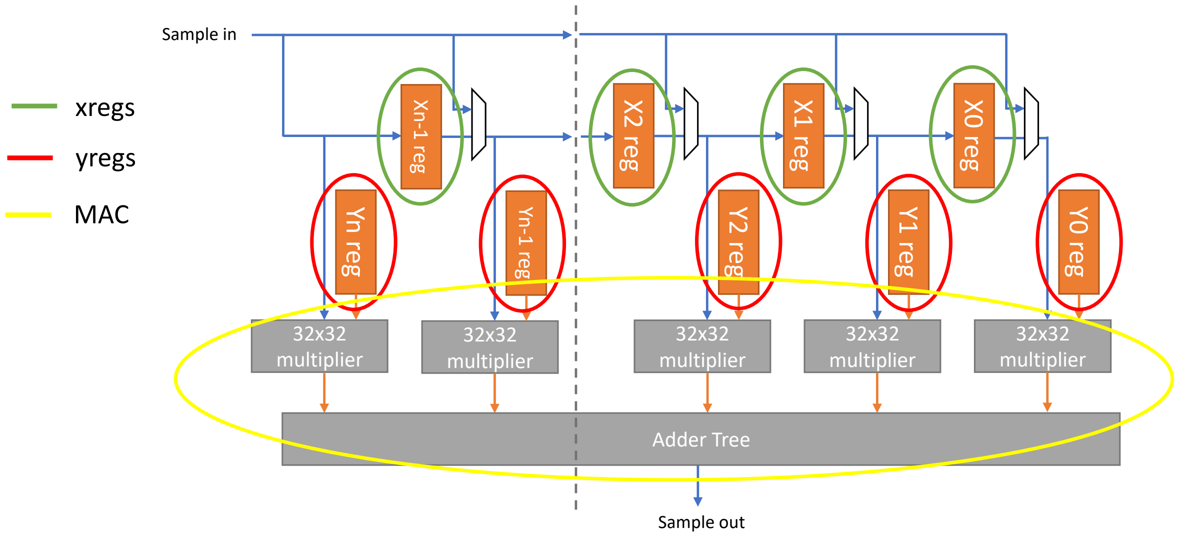


Figure 6.1: A schematic overview of the fully parallel design with the analysis categories annotated.

The chip area results can be seen in Figure 6.2. From this bar graph, the dominant area contribution of the MAC hardware can clearly be seen. This hardware encompasses all the 256 32-bit multipliers and the adder tree that sums all the results. The power results shown in Figure 6.3 show that even though the MAC category dominates the area figure, this is not the case for the energy consumption. Here it can be seen that the memory elements have a more significant contribution. Interesting to note is that the memory of the X registers and Y registers is almost equal, while the Y values do not change and the X registers change every sample clock. This would imply that moving the data between registers is insignificant with respect to driving the output from a power perspective.

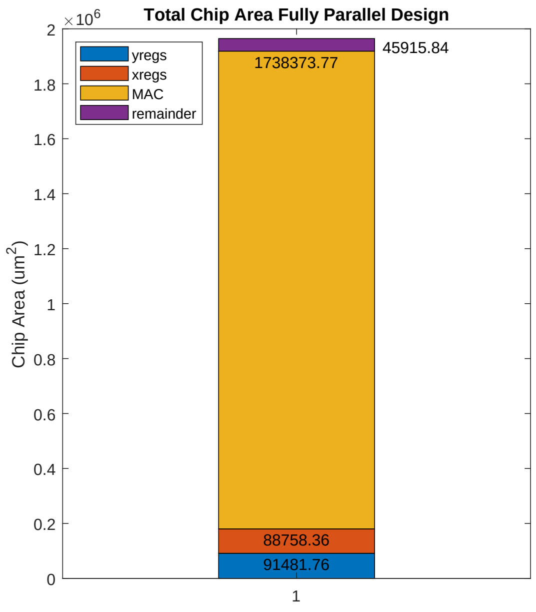


Figure 6.2: Bar graph showing the total cell area of the fully parallel design. The different colours indicate the contribution of specified subsections of the design. It can be seen that the total chip area is dominated by the MAC hardware.

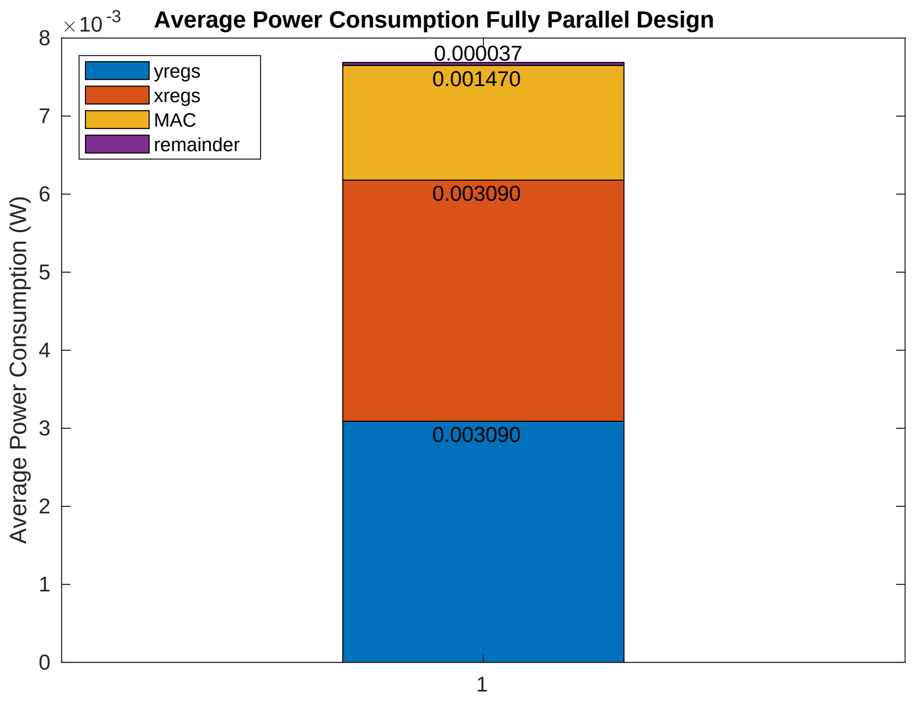


Figure 6.3: Bar graph showing the average power usage of the fully parallel design. The average power is computed over 120 samples. The design is configured for a 100 tap FIR filter and thus only 100 of the 256 multipliers are used effectively. The different colours indicate the contribution of specified subsections of the design.

6.2 Single Multiplier Design

For the results of the single multiplier design more categories are defined. These categories are annotated in the schematic in Figure 6.4. The program memory is a new addition to the categories.

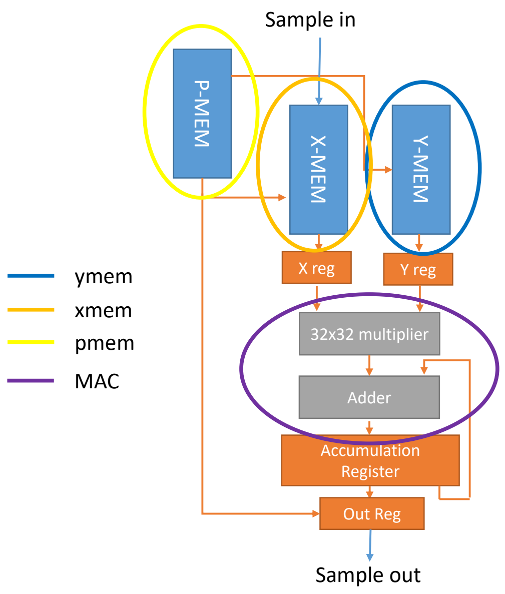


Figure 6.4: Basic schematic overview of the single multiplier design as shown in section 4.4, now annotated with the analysis categories.

First the chip area results are shown in Figure 6.5. In this figure there are no real surprises. Since there is only one multiplier the contribution of the memories is relatively larger.

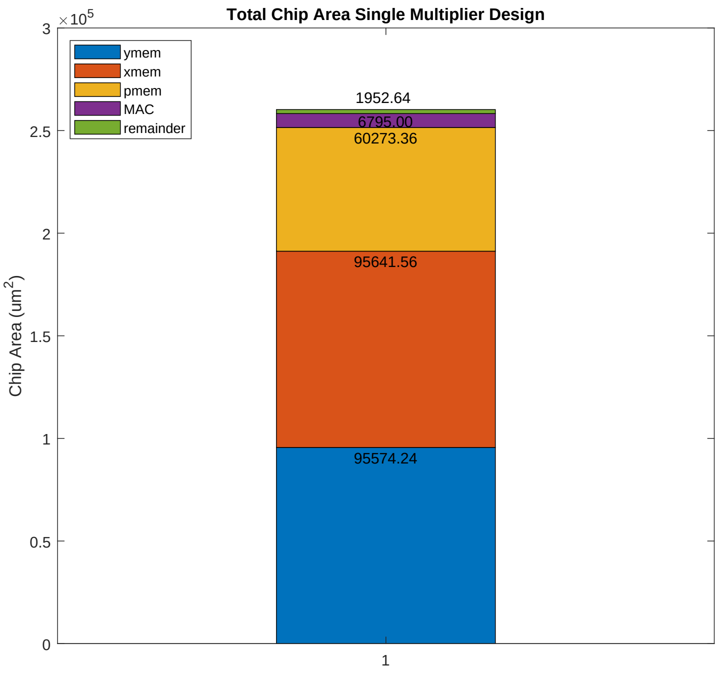


Figure 6.5: Bar graph showing the total cell area of the single multiplier design. The different colours indicate the contribution of specified subsections of the design.

The average power consumption can be seen in Figure 6.6. From the figure it can be seen that the chosen categories encompass the energy hungry sections quite well, as the ”remainder” contribution is small. This shows that little energy consumption is unaccounted for. The second observation made is that the average power is heavily dominated by the memories. This is not unexpected according to the literature mentioned in section 3.4. However, as discussed in section 5.2.1 the ”SRAM” used in the designs is not fully representative for industry standard SRAM. In a more accurate representation, the energy consumption of the memories would likely be lower.

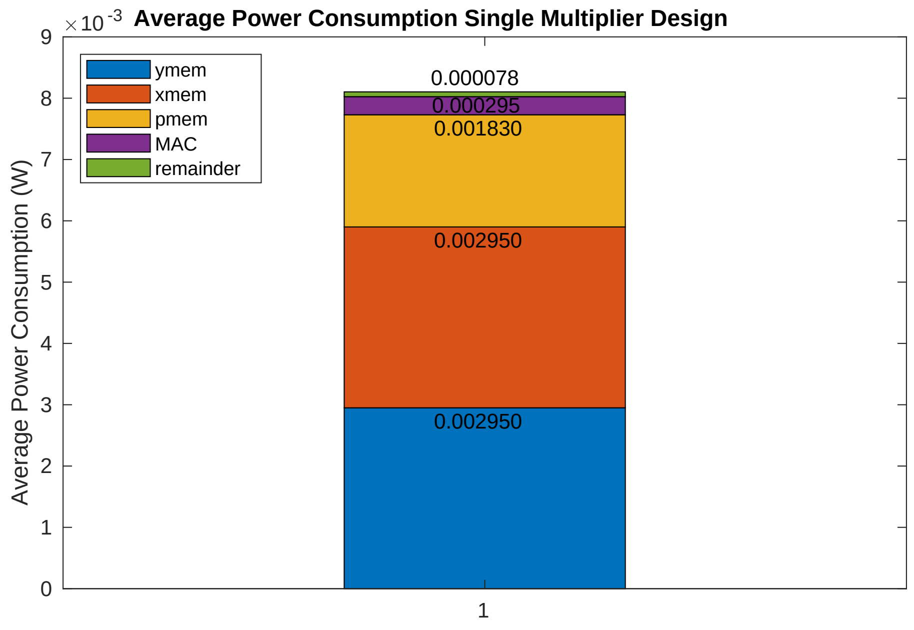


Figure 6.6: Bar graph showing the average power usage of the single multiplier design over 300 samples. The different colours indicate the contribution of specified subsections of the design.

6.3 Comparison Fully Parallel - Single Multiplier

In Figure 6.7 the area results of the fully parallel and single multiplier design are compared. The fully dedicated design is much larger, as is to be expected. From the figure it can be seen that both the designs have roughly the same area occupied by memory. The difference is in the MAC section of the fully parallel design, which is much larger.

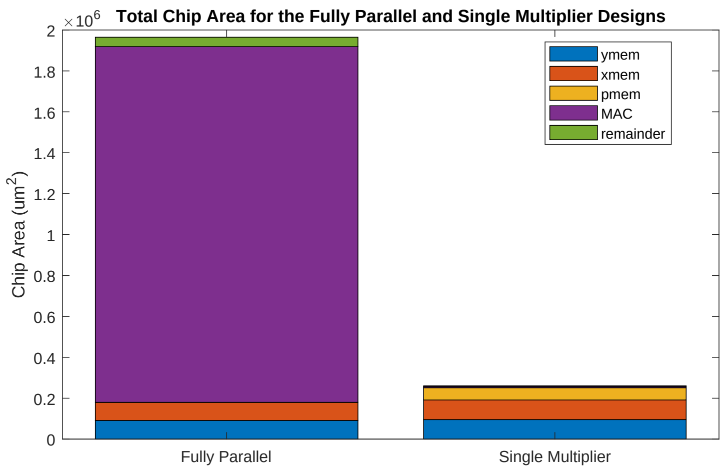


Figure 6.7: Bar graph showing the needed chip area for both the fully parallel and single multiplier designs. The MAC section of the fully parallel requires much more area, as the fully parallel design has 256 multipliers compared to only one that is used in the single multiplier design.

Comparing the average power results in Figure 6.8 it can be seen that the total energy consumption is quite similar. The single multiplier design consumes 5.4% more. Looking at the different categories an interesting difference can be seen. While the memory elements consume roughly the same the average MAC consumption is much smaller for the single multiplier design even though the amount of multiplication operations is the same for both designs. This decrease in MAC consumption is however more than fully offset by the consumption of the program memory.

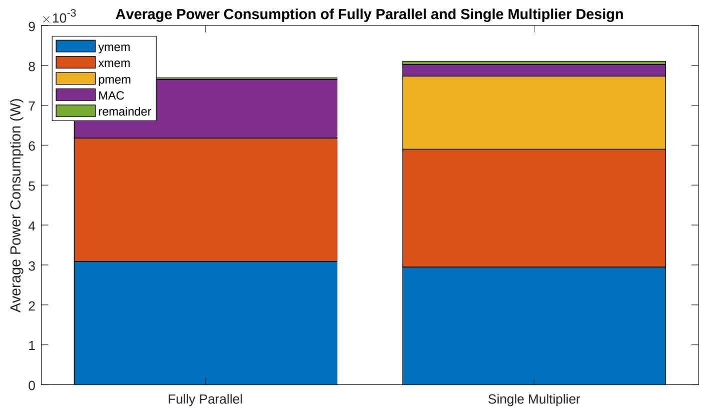


Figure 6.8

6.4 N Multiplier Design

For the analysis of the N-multiplier designs the categories are annotated in Figure 6.9. For the single multiplier design, categories are added for the Barrel Shifter, the X registers and Y registers.

Average Power Consumption

A bar graph for the average power for 32 designs with $2 \geq n \geq 33$ can be seen in Figure 6.10. From this bar graph a number of observations can be made.

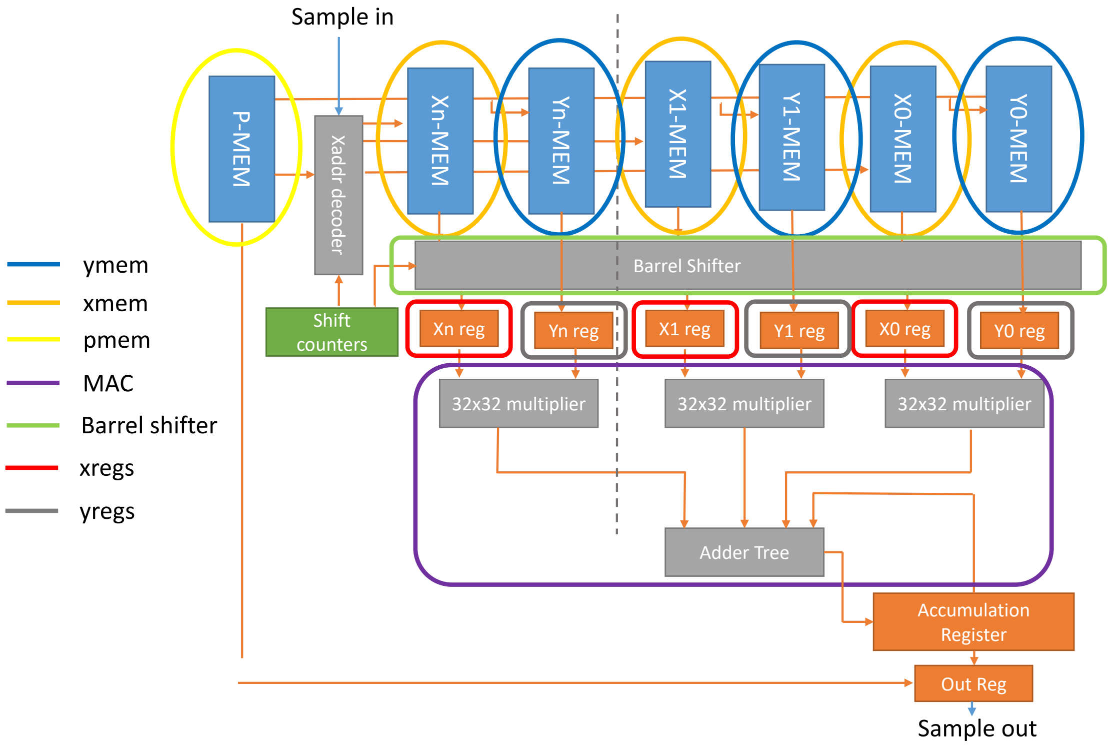


Figure 6.9: A schematic overview of the n-multipliers design with the analysis categories annotated.

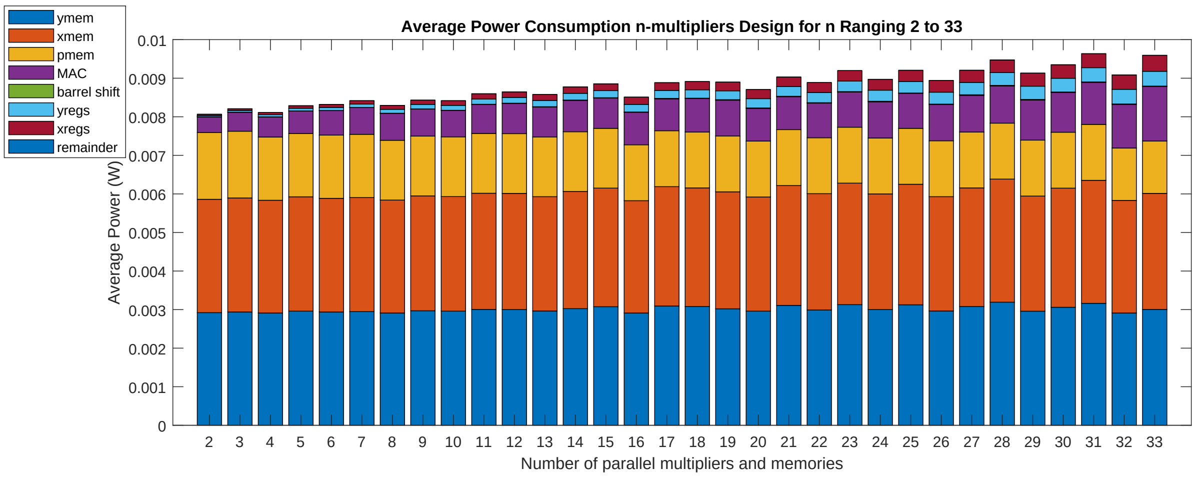


Figure 6.10: Bar graph showing the average power usage of the the N-multipliers designs for $2 \geq n \geq 33$.

First and foremost, the average power of the complete designs increases with n. For $n = 2$ the average power consumption is very close to that of only a single multiplier and for $n = 33$ the consumption has increased with around 10% with respect to the single multiplier design.

Second, the energy consumption of the X and Y memories shows no clear trend upward or downward. Every design has nearly the same amount of total storage space, but from section 3.4 the expectation rose that the energy consumption should decrease when the amount of words in each individual memory would go down. This deviation from expectations could be explained by the standard-cell implementation of the SRAM instead of a hard macro.
Third, the rise in total energy consumption can be attributed to the MAC, X registers and Y registers. This trend is also clearly visible in Figure 6.11. In this figure the difference in energy consumption of the different categories with respect to the single multiplier design is shown. This figure shows the trend of whether a category increases or decreases its consumption with the increase of n. The increases

shown are curious, because while the amount of MAC hardware and the amount of registers increases, the number of multiplications, additions and register updates is the same for all designs. In all cases a 100 taps FIR filter is implemented. For the MAC hardware the cause of this behavior may lie in the fact that all multipliers feed into one adder tree. When an output of even one multiplier changes this causes switching all throughout the adder tree. When a lot of multipliers change output at slightly different times this will result in a lot of unnecessary updates in the adder tree, causing switching losses. This hypothesis has not been verified. For the X and Y registers the cause may lie in the switching losses that still occur even when the new register value is the same as the old value. As the amount of registers increases, each register will have less new values, but the ”idle” switching loss may still be there. This hypothesis drives the choice to implement clock gating, which is elaborated upon in sections 6.6 and 6.7.

Program Memory

Figure 6.11 also shows the decrease in energy consumption of the program memory. This decrease can be explained by two factors: The first and most significant cause is that when memories with fewer words are used, less bits are necessary for the memory pointer. This is why the drops in consumption occur at powers of two, as that is the point where one bit less is needed. The P-Mem then becomes smaller and the average power decreases. The second effect which is less visible but still present is the decrease in amount of instructions needed to complete the program. The reducing program size can be seen in Figure 6.12, where the 100 FIR tap program can be seen for different n. This behavior shows that (for this standard cell ”SRAM” implementation of) the P-Mem word width affects the energy consumption much more than the amount of memory accesses needed to run the program.

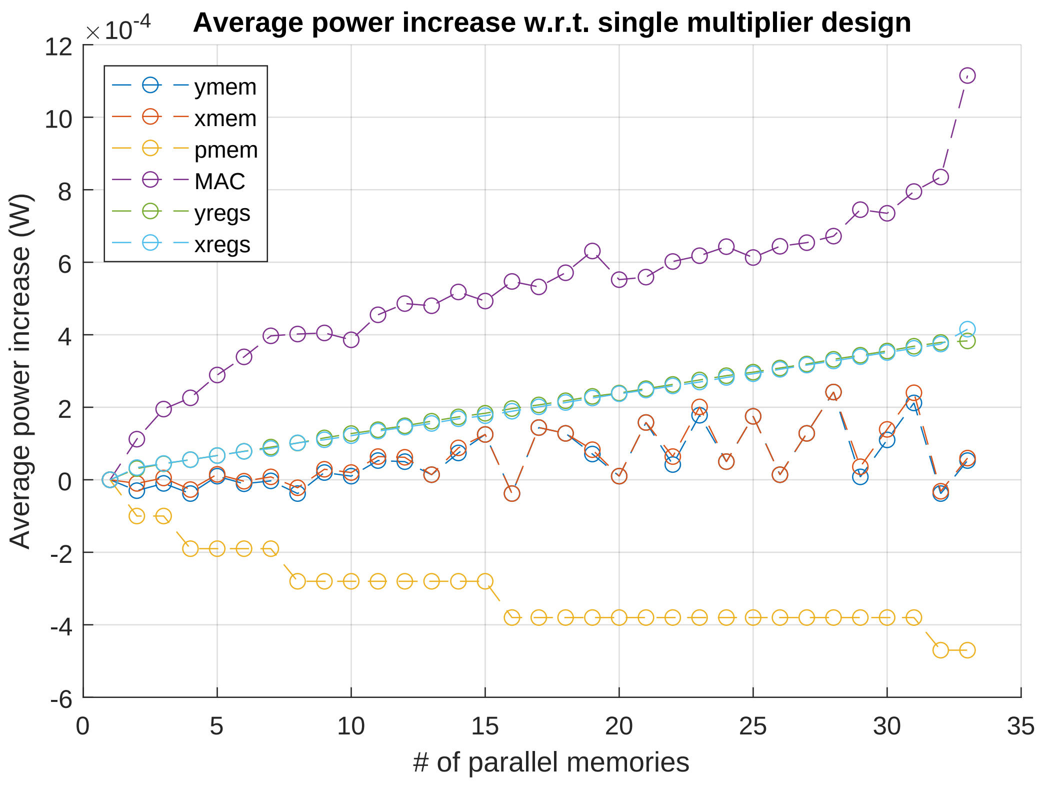


Figure 6.11: Plot showing the increase/decrease in average power for four categories with respect to the single multiplier design. From the plot the increase in consumption of the MAC hardware and decrease for the P-Mem can clearly be seen.

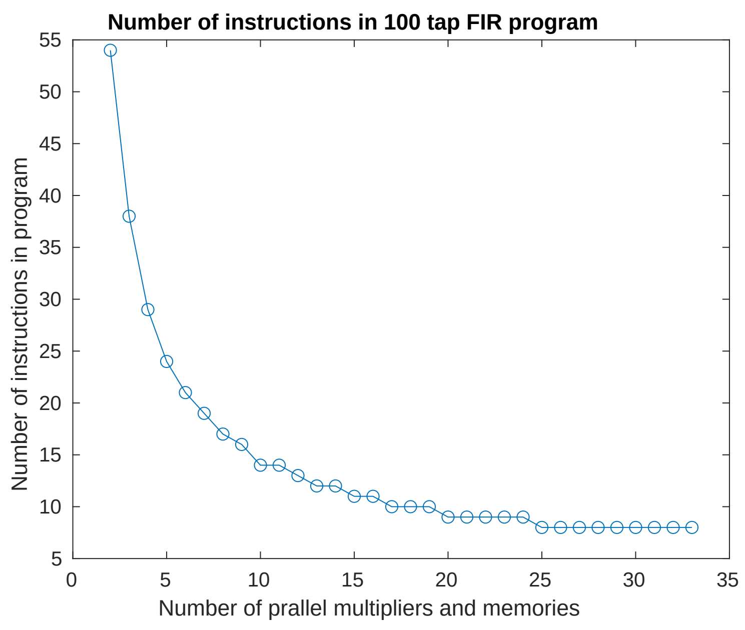


Figure 6.12: Plot showing the amount of instructions needed to execute a 100 tap FIR filter program for $2 \geq n \geq 33$.

Area Results

The total cell area for the same designs are shown in Figure 6.13. The area results show little surprises. As expected, the amount of area used by the MAC, X and Y registers increases. The total area of the X and Y memory differs for every design and is lowest when n is a power of two. This is explained by the choice to round the number of addresses of each X and Y memory upwards, as discussed in section 4.5.1. When n is a power of two, $\frac{256}{n}$ becomes an integer value and no rounding is necessary, thus producing less total memory space and consequently, less area. In that same fashion, when n is a power of two the amount of addresses needed per memory precisely fit in the amount of bits used for the address lines. Thus, less overhead is needed to address all memories.

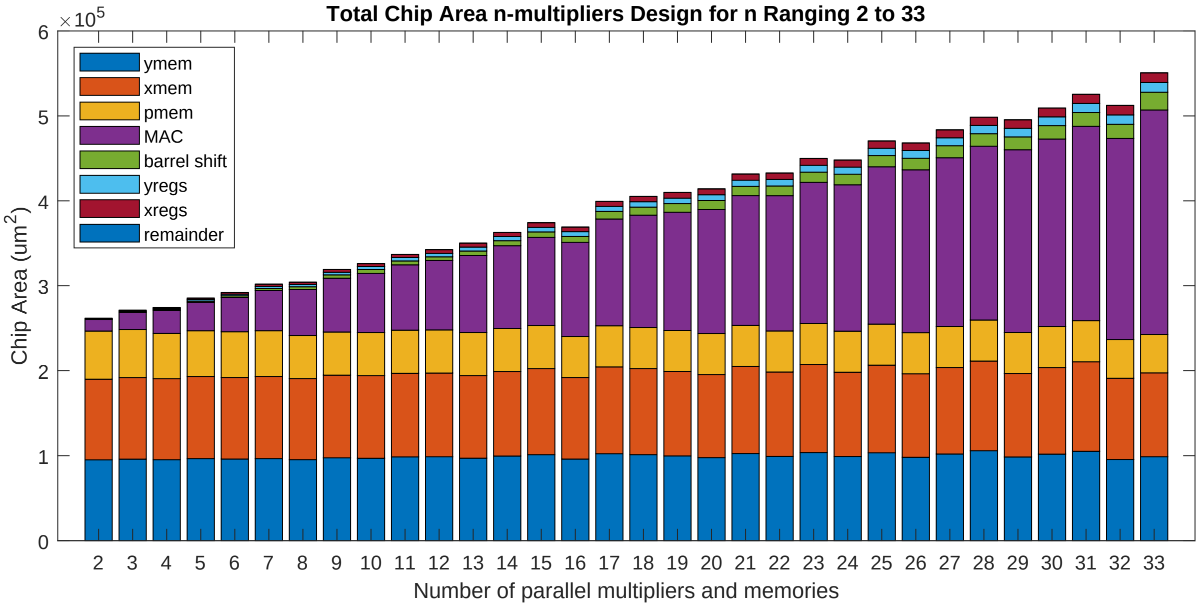


Figure 6.13: Bar graph showing the total cell area of the the N-multipliers designs for $2 \geq n \geq 33$.

6.5 Comparison of Sinlge Multiplier, N-Multiplier and Fully Parallel Designs

Plotting all the area and power results of the previous sections in one bar graph Figure 6.14 and Figure 6.15 are created. The trend seen in the area results leaves little surprises. With increasing n, the total area needed for the memory stays roughly the same while mainly the MAC hardware increases. Extrapolating this for beyond $n = 33$, the expectation is that at $n = 256$ the total area will be roughly that of the fully parallel design.

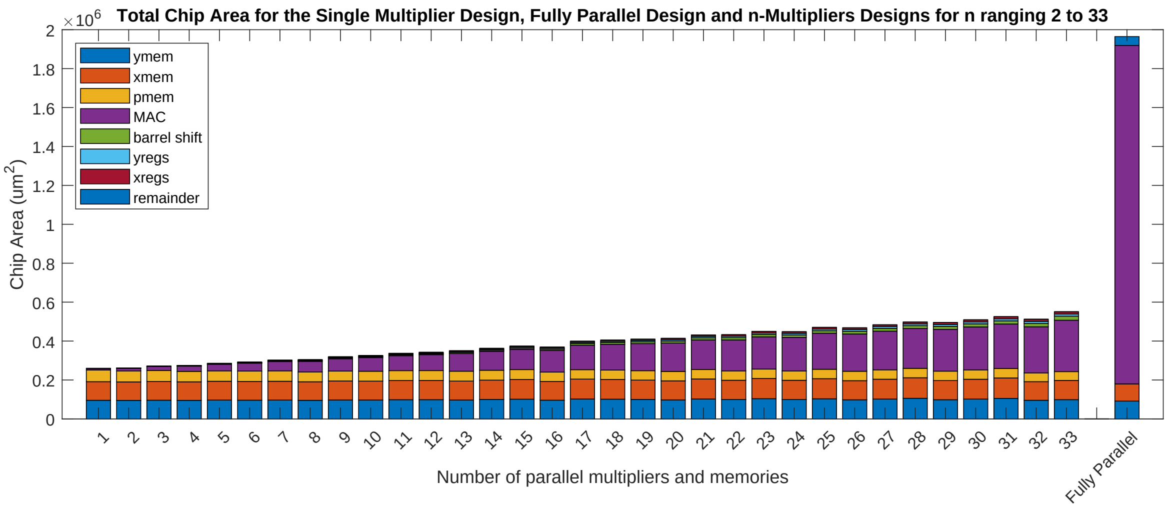


Figure 6.14: Bar graph showing the total cell area of the single multiplier design, the fully parallel design and the n-multipliers design for $2 \geq n \geq 33$. A table with the values of all the bars is given in appendix D.1

For the average power results however the trend does not seem to advance towards the fully parallel design. While the energy consumption of the memories stays roughly the same the energy consumption of the MAC at $n = 33$ is already roughly the same as the fully parallel case. Extrapolating this graph for a higher n will likely result in even more power consumption.

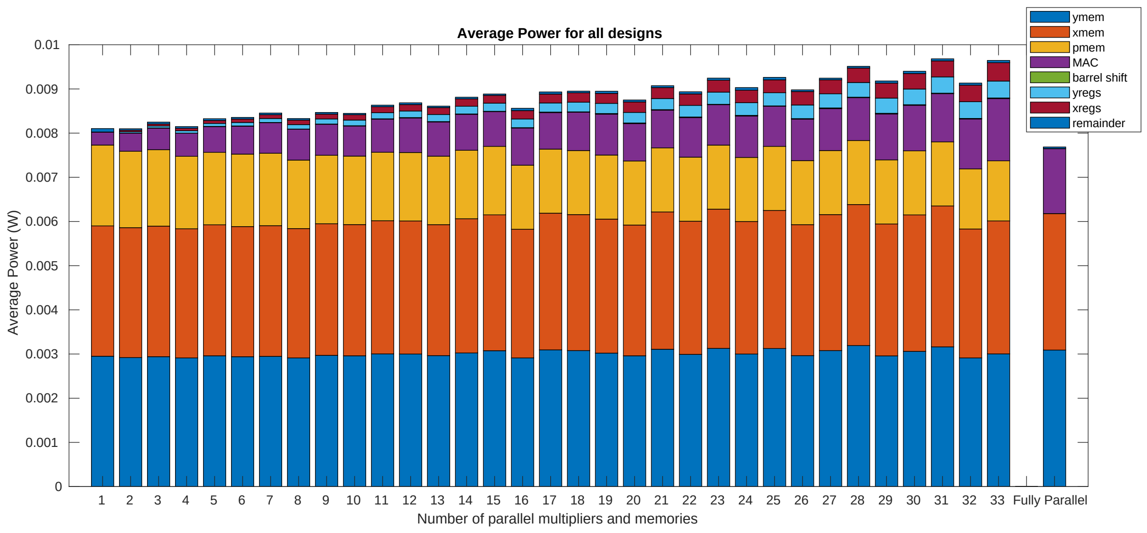


Figure 6.15: Bar graph showing the average power consumption of the single multiplier design, the fully parallel design and the n-multipliers design for $2 \geq n \geq 33$. A table with the values of all the subsection is given in appendix D.1

6.6 Clock Gating

As mentioned in section 6.4, disabling the clock input for hardware that does not need to run can result in a reduction in energy consumption. This clock gating is a well known technique that is widely used in digital ASIC hardware. The basic principle is shown in Figure 6.16. When the input of a register is preceded by a multiplexer that either passes a new value or the registers own output, a transformation can be made. Instead of setting the previous output to the input of the register, the same control signal for the multiplexer can be used to choose whether the clock to the register is passed or not. If not, the register will not update and the output will remain the same. Only now, no energy is used to update the register values. Synopsys Design Compiler can recognize the above mentioned situation and automatically implement clock gating. An attempt was made to apply this functionality to the n-multiplier designs but unfortunately this did not result in functioning designs. As an alternative the clock was manipulated within the RTL design. A primitive was created that instructs Clash how to generate the desired Verilog RTL. The specific primitive that is created simply takes a control signal and a clock signal as inputs to an AND gate and outputs the "gated" clock signal. The primitive description can be found in Appendix B.2. The designs in this work possess the property that during most clock cycles, either all of the hardware needs to be enabled or all the hardware can be disabled. Therefore a single clock gate is implemented that can disable the entire design except for the control logic of the clock gate itself. The specific control signal originates from a register in which the value is set to true by the external frame trigger and set to false by reaching the end of a program. This results in a gated clock that turns on when a new sample arrives, remains on for the duration of the program, and turns off directly after the last instruction of the program is executed.

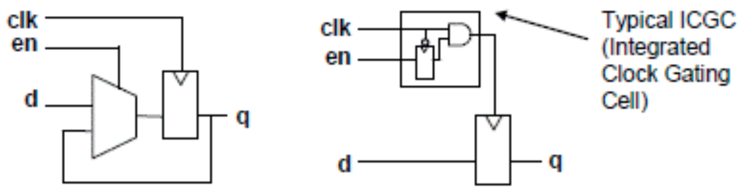


Figure 6.16: RTL schematic illustrating the clock gating transformation[24].

6.7 Results Clock Gated Designs

The average power results for n-multiplier designs with clock gating ranging from $n = 1$ till $n = 33$ can be seen in Figure 6.17. The chip area results for the clock gated designs are not shown as they are very similar to the results shown for the non clock gated designs in Figure 6.14. This is the case since only the single clock gate is added, which uses very little chip area.

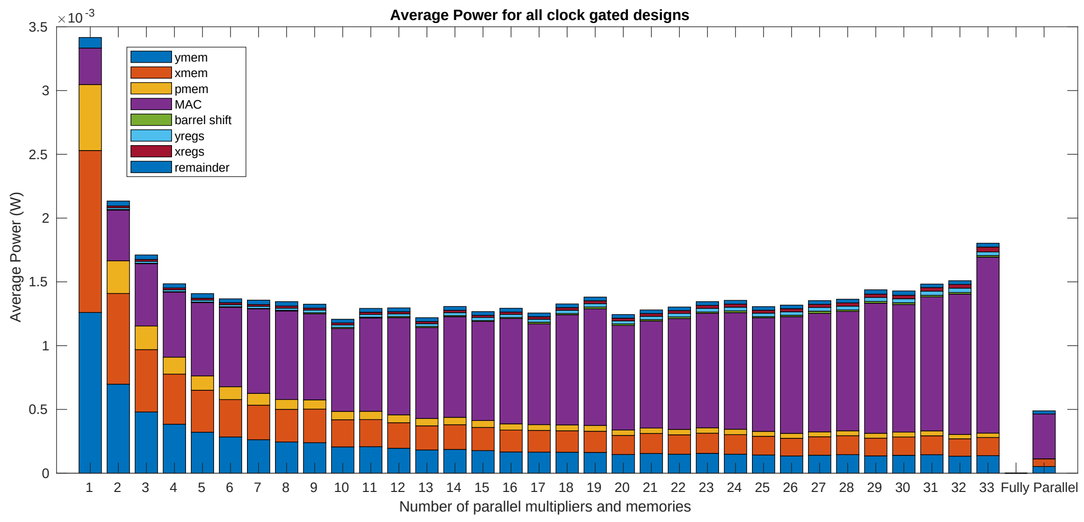


Figure 6.17: Bar graph showing the average power consumption of the single multiplier design, the fully parallel design and the n-multipliers design for $2 \geq n \geq 33$. Clock gating is applied to all designs in this figure. A table with the values of all the subsection is given in appendix D.2

The first observation that is made is that there is a clear reduction in consumption for all designs. The designs in figure 6.17 range from 3.4 mW to 0.49 mW while the designs without clock gating as shown in figure 6.15 never drop below 7.68 mW.
The second observation is that there are significant reductions in the average power consumption of the X and Y memories where this was not the case for the designs without clock gating. Figure 6.18 shows again the increase or decrease in energy consumption of different categories with respect to the single multiplier design. A logarithmic trend can be seen in the X and Y memory categories. This trend is not present in the designs without clock gating in Figure 6.14. This difference in the results shows that using fewer words is not necessarily beneficial in the used DesignWare memory IP. The decrease in energy consumption likely flows from the fact that all registers inside the X and Y memory are clocked while the data of only one register is needed. Using more memories for fewer clock cycles is very beneficial since the cumulative amount of register clocking is reduced.

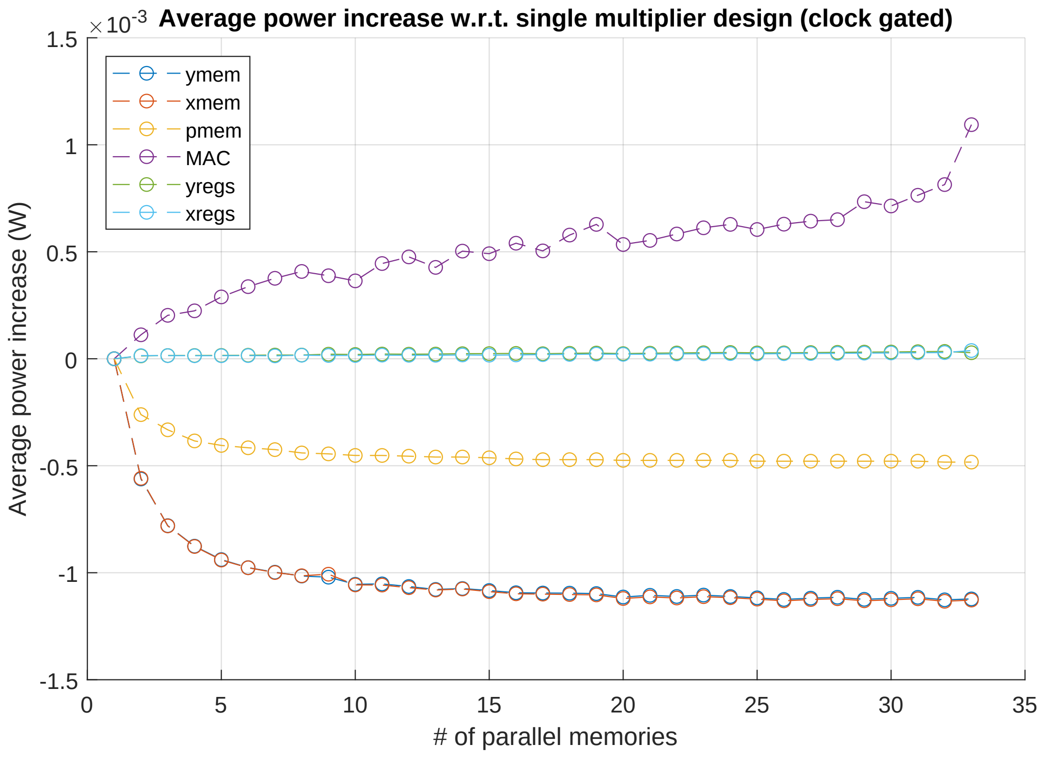


Figure 6.18: Plot showing the increase/decrease in average power for four categories with respect to the single multiplier design. All designs have clock gating applied.

Figure 6.18 also shows that the increasing consumption of the X and Y registers that was discussed in section 6.4 is no longer present. Additionally, the consumption of the program memory is no longer dominated by the instruction bit-width but by the program length.
The only category that remains virtually unchanged is that of the MAC hardware. Since the MAC category consists purely of combinational logic and no clocked hardware the lack of change by the introduction of clock gating is not surprising.

6.8 MAC Energy Consumption for Highly Parallel Designs

To further investigate the energy consumption behaviour of the MAC category the average power of designs for $1 \geq n \geq 110, 128$ and 255 multipliers is given in Figure 6.19.

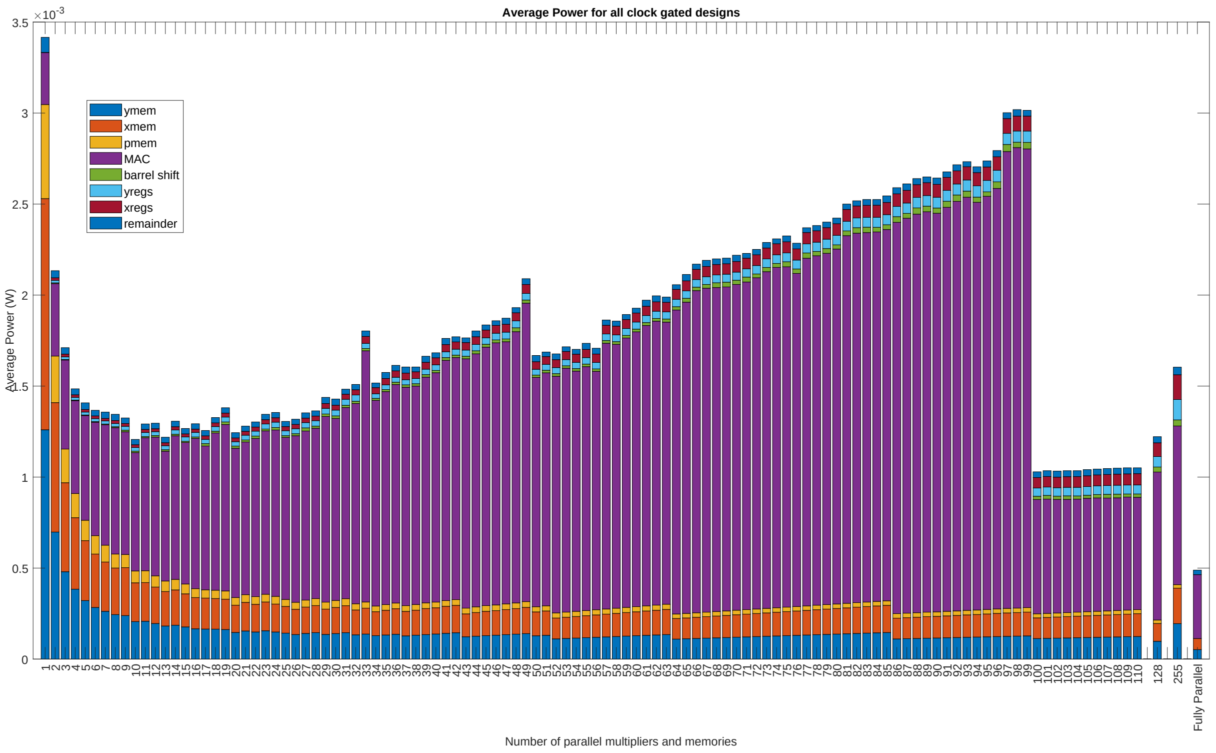


Figure 6.19: Bar graph showing the average power consumption for designs with $1 \geq n \geq 110, 128$ and 255 multipliers and the fully parallel design. All designs have clock gating applied.

The first observation made is the steep decrease in energy consumption at $n = 100$. This decrease can partly be explained by the fact that a FIR filter with exactly 100 taps is executed. When the number of multipliers in the designs is equal to or exceeds the number of taps that is executed the MAC hardware is only used once. However, the decrease in energy consumption of the MAC category between $n = 99$ and $n = 100$ is much more significant with respect to the decrease between $n = 49$ and $n = 50$. In both cases the number of cycles the MAC hardware is operational is reduced, between $n = 49$ and $n = 50$ it is reduced from 3 cycles to 2, while between $n = 99$ and $n = 100$ it is reduced from 2 cycles to 1. The exact cause for this specific deviation is not found. In future work an attempt can be made to further investigate the behaviour by defining separate categories for the multipliers and adders or even looking at the energy consumption of individual multipliers and adders.

Another observation that can be made is that there is a decrease in energy consumption at every n by which 100 is divisible, such as $n = 10, n = 20, n = 25, n = 50$ and $n = 100$ as mentioned above. This decrease can be explained by the fact that in the designs containing these number of multipliers all multipliers always contribute to the output and there are no cycles where only a fraction of the multipliers are used. This effect of excess multipliers can also be seen at $n = 128$ and $n = 255$, where the number of multipliers exceeds the number of taps in the implemented filter.

Unfortunately obtaining average power results for more designs did not deliver a better explanation for the large increase in energy consumption of the MAC hardware with the increase in number of multipliers. Also Figure 6.20, which shows the area results for the designs discussed, does not give any new insights. The area results show a steady increase in MAC hardware, as is to be expected, but the rise in energy consumption can not be attributed to increasing leakage dissipation of the larger hardware, as then the decrease in energy consumption at $n = 100$ in Figure 6.19 would not be present. Figure 6.20 also shows that the n-multiplier architecture with $n = 255$ would require more chip area compared to the fully parallel design. This additional hardware is caused by the large amount of control hardware needed to address all the memories, the large amount of X and Y registers needed and the large barrel shifter section.

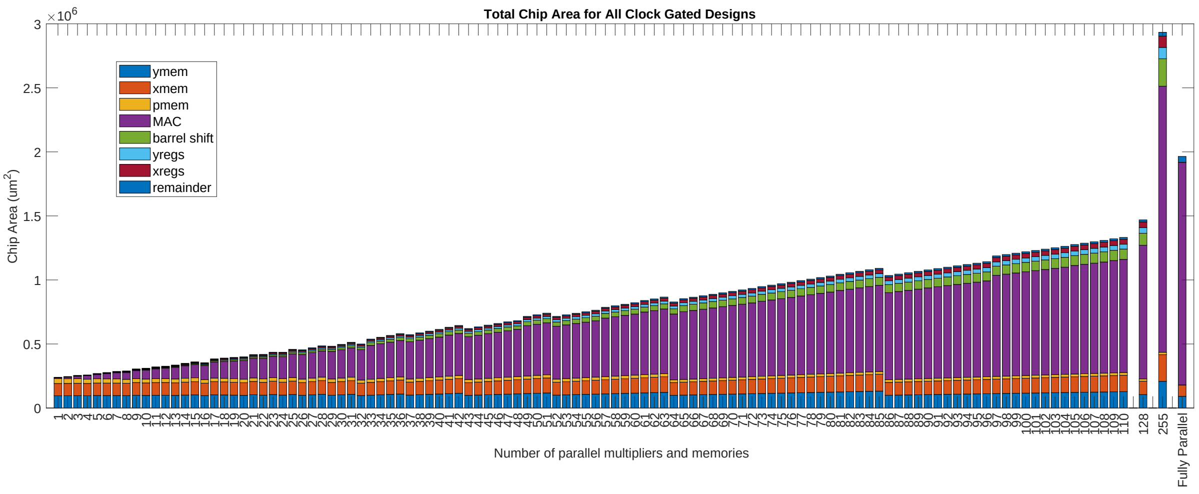


Figure 6.20: Bar graph showing the total cell area for designs with $1 \geq n \geq 110, 128$ *and* 255 *multipliers and the fully parallel design. All designs have clock gating applied.*

7 CONCLUSION

This work aimed to gain more insight into how architectural choices affect the energy-efficiency of digital ASIC hardware in an audio setting. The focus was laid on the time-area trade-off in FIR implementations. In section 3 an overview of relevant theory and literature on power consumption in digital hardware is shown. In sections 4 a method was presented to systematically create designs with a different time-area trade-off, introducing a single ”knob” that would adjust the amount of parallelism in a design. Analyzing these designs with the method elaborated upon in section 5 the time-area design space can be explored while focusing on energy efficiency. Section 4 also presented two designs for comparison: One with a very flexible ”processor like” architecture with only a single multiplier and a fully parallel, fully dedicated design of an FIR filter.

The first conclusion that can be drawn is that the proposed analysis method can give additional insight into the energy consumption behaviour of a design. This is illustrated in section 6. Here it was identified that when the amount of X and Y registers in the design was increased, the dynamic energy consumption also increased even though the total amount of data passing through all X and Y registers was the same for all designs. This identified the potential for using clock gating to reduce the dynamic power consumption. Being able to investigate the energy consumption over multiple designs gave insight into the trend and subsequently sparked the question of what might have caused this trend. The design change following from this question improved the energy efficiency of all designs. This example shows that the proposed method can be a tool for aiding the digital IC designer in navigating the energy efficiency design space.

From the results presented in sections 6.7 and 6.8, where the energy consumption and required chip area of designs with clock gating applied are discussed, an answer to the research question *”How does the time-area trade-off affect the energy efficiency of a streaming FIR filter ASIC implementation for audio purposes?”* can be formulated. Starting with the edge cases of the trade-off, the fully parallel design and the design with only a single multiplier, the beneficial effect of parallelism on energy efficiency can clearly be seen. The fully parallel design has an energy consumption that is more than 85% lower compared to the single multiplier design. This gain in energy efficiency comes at a cost of flexibility and chip area, as the fully parallel design is 8 times as large. All designs with an $2 \geq n \geq 255$ number of multipliers are within these edge cases in terms of energy consumption. The largest gains in energy efficiency compared to the single multiplier design are obtained for designs with a small n. At $n = 2$ a 37% decrease in energy consumption is obtained while only increasing the total chip area by 2%. As n increases the gain in energy efficiency reduces and for $n > 20$ the energy consumption often even increases again. The cause of this increase can be attributed to the multipliers and adders, but the exact reason for the energy consumption increase of this specific part of the design has not been found.

Important to note is that the results are obtained through simulations only and actual silicon performance may differ. Especially the SRAM memory model constructed of standard library cells used in the designs can differ significantly from the SRAM that is found in chips in industry.

8 DISCUSSION AND FUTURE WORK

In this work a number of assumptions and simplifications are made to benefit the research process. Often these choices are a result of either the limited availability of time or resources such as design tool software. The assumptions and simplifications with the most expected influence on the results are discussed in this chapter.

By narrowing the research question in section 2 the assumption is made that the results for a FIR implementation give insight for a more complex DSP with more functionality. More complex DSP algorithms still mainly consists of multiplications, additions and memory access operations. But the relative contribution to energy consumption of these operations will vary from the FIR case. For instance, the introduction of feedback will put more strain on the memories, as outputs need to be written back. Therefore, applying the conclusions drawn in this work on other algorithms should be done with care.

The absence of more accurate SRAM models can have a great influence on the results presented in this thesis as the energy consumption of most designs is dominated by that of the memories. Currently a memory implementation using registers from the standard-cell library is used to model memory behaviour. Optimized hard-macro SRAM IP as is the industry standard is likely much more energy efficient. Additionally, the change in energy consumption with respect to memory size will likely be different.

In section 4 the choice is made to explore the time-area trade-off by making the single multiplier ”processor like” design more parallel. This resulted in that all designs kept the addressable memory layout. An interesting topic for future research would be to ”fold” the fully parallel design, which uses a tapped delay line, in time instead. Both techniques will result in designs that explore the time-area trade off, but the ”folded” designs will use tapped delay lines instead of addressable memory.

Using a more abstract hardware description language such as Clash instead of low-level VHDL or Verilog can introduce a lack of understanding of exactly what hardware specifically is generated. This may have introduced illogical and inefficient hardware structures which while functionally correct may hinder optimal power performance.

Because of limited available libraries the 65nm UMC IC technology library was used instead of the 55nm TSMC as is used by Axign. This changes the power figures as a smaller processing node generally has less capacitance and thus less energy dissipation. Additionally, using the 65nm library from UMC instead of the 65nm library from TSMC might have introduced more changes to the power behavior.

The synthesis toolset is proprietary software which introduces many optimization steps of which the exact functionality is not well known. Additionally, there is limited available documentation. Tutorials and guides for working with the tools are hard to come by. The synthesis process is an important step to obtain an accurate model for the power behaviour if the design were to be produced on an IC. The difficulties in working with and understanding the tools may have resulted in unnecessary inaccuracies in the results presented in this work.

All power analysis and area results are extracted from a netlist that has not been through the layout phase of IC design. The layout phase can introduce deviations in the power behavior because of specific placing, lengths of traces, clock tree layout etc.

Future Work

Below a list can be found of recommended subjects to investigate in future work, listed from highest priority to lowest.

- The effect of using industry standard, optimized, hard-macro SRAM.
- The cause for the increase in MAC energy consumption for n-multiplier designs with a high n.
- The effect of ”folding” the tapped delay line to vary parallelism instead of expanding the processor architecture with more multipliers.
- The expansion of the analysis method to other DSP functionalities aside from FIR filtering.
- The effect of using the 65 nm UMC IC technology instead of the 55 nm TSMC technology.
- The use of floating point samples/coefficients instead of the fixed point format.
- The energy efficiency of the multipliers and adder trees generated by the synthesis tools and how this may be improved.

REFERENCES

[1] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, “ClaSH: Structural Descriptions of Synchronous Hardware Using Haskell,” in *13th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2010, 1-3 September 2010, Lille, France* (S. López, ed.), pp. 714–721, IEEE Computer Society, 2010.

[2] W. Suntiamorntut, N. Gupta, L. E. M. Brackenbury, and J. Garside, *Energy Efficient Functional Unit for a Parallel Asynchronous DSP*. PhD thesis, University of Manchester, 2005.

[3] A. Chandrakasan, S. Sheng, and R. Brodersen, “Low-power CMOS digital design,” *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, 1992.

[4] T. Arslan, A. T. Erdogan, and D. H. Horrocks, “Low power design for DSP: methodologies and techniques,” *Microelectronics Journal*, vol. 27, no. 8, pp. 731–744, 1996.

[5] H. Veendrick, *Nanometer cmos ics*. New York, NY: Springer Science+Business Media, LLC, 2017.

[6] P. J. Havinga and G. J. M. Smit, “Design techniques for low-power systems,” *Journal of systems architecture*, vol. 46, no. 1, pp. 1–21, 2000. Publisher: Elsevier.

[7] H. Veendrick, “Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits,” *IEEE Journal of Solid-State Circuits*, vol. 19, pp. 468–473, Aug. 1984. Conference Name: IEEE Journal of Solid-State Circuits.

[8] S. Jain, L. Lin, and M. Alioto, “Dynamically Adaptable Pipeline for Energy-Efficient Microarchitectures Under Wide Voltage Scaling,” *IEEE Journal of Solid-State Circuits*, vol. 53, pp. 632–641, Feb. 2018. Conference Name: IEEE Journal of Solid-State Circuits.

[9] N. Ickes, G. Gammie, M. E. Sinangil, R. Rithe, J. Gu, A. Wang, H. Mair, S. Datla, B. Rong, S. Honnavara-Prasad, L. Ho, G. Baldwin, D. Buss, A. P. Chandrakasan, and U. Ko, “A 28 nm 0.6 V Low Power DSP for Mobile Applications,” *IEEE Journal of Solid-State Circuits*, vol. 47, pp. 35–46, Jan. 2012. Conference Name: IEEE Journal of Solid-State Circuits.

[10] T. D. Burd, “Energy-efficient processor system design,” Tech. Rep. UCB/ERL M01/13, EECS Department, University of California, Berkeley, Mar. 2001.

[11] “ELCT706 Microelectronics: course notes on multipliers,” 2014. Published: eee.guc.edu.eg.

[12] A. Mason, “Memory Basics, Lecture Notes Ch 13.”

[13] M. Kamble and K. Ghose, “Analytical energy dissipation models for low power caches,” in *Proceedings of 1997 International Symposium on Low Power Electronics and Design*, pp. 143–148, Aug. 1997.

[14] S. o. M.I.T. 6.004, “Lecture notes for Computation Structures,” 2017.

[15] R. Evans and P. Franzon, “Energy consumption modeling and optimization for SRAM’s,” *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 571–579, May 1995. Conference Name: IEEE Journal of Solid-State Circuits.

[16] A. Azizi-Mazreah, M. T. M. Shalmani, H. Barati, and A. Barati, “Delay and energy consumption analysis of conventional SRAM,” in *Proc. Of World Academy of Science, Engineering and Technology*, vol. 27, Citeseer, 2008.

[17] J. Rabaey, L. Guerra, and R. Mehra, "Design guidance in the power dimension," in *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. 2837–2840 vol.5, May 1995. ISSN: 1520-6149.

[18] A. Abnous and J. Rabaey, "Ultra-low-power domain-specific multimedia processors," in *VLSI Signal Processing, IX*, pp. 461–470, Oct. 1996.

[19] R. Mehra and J. Rabaey, "Exploiting regularity for low-power design," in *Proceedings of International Conference on Computer Aided Design*, pp. 166–172, Nov. 1996.

[20] J. Tomarakos, "Relationship of Data Word Size to Dynamic Range and Signal Quality in Digital Audio Processing Applications," tech. rep., Analog Devices.

[21] "Synopsys design compiler." `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html`. Accessed on 2022-02-10.

[22] "Modelsim hdl simulator." `https://eda.sw.siemens.com/en-US/ic/modelsim/`. Accessed on 2022-02-10.

[23] "Synopsys primetime." `https://www.synopsys.com/support/training/signoff/primetime1-fcd.html`. Accessed on 2022-02-10.

[24] N. Koduri and K. Vittal, "Power analysis of clock gating at RTL." https://www.design-reuse.com/articles/23701/power-analysis-clock-gating-rtl.html. Accessed on 2022-02-10.

A CLASH CODE

A.1 Fully Parallel Design

A.1.1 Data Types

```haskell
module Data_types_FIR_acc_10 where
import Clash.Prelude
import Clash.Explicit.Testbench

-- Define a clock domain with a 20 ns clock period (50 MHz)
createDomain vSystem{vName="Twenty", vPeriod=20000}

-- Define clock, reset, enable and data signals in the 50 MHz clock domain
type Clk = Clock "Twenty"
type Rst = Reset "Twenty"
type Sig = Signal "Twenty"
type En = Enable "Twenty"

-- Define samples and coefficients to be 32 bit signed fixed point values with 31 fractional
    bits
type Data_reg = SFixed 1 31
type Coeff_reg = SFixed 1 31
type Sin = SFixed 1 31
type Sout = SFixed 1 31
-- Define number of taps input
type Num_taps = Int
-- Define coefficient and data memory banks chosen to be 256 32-bit registers
type Mem_coeff = Vec 256 Coeff_reg
type Mem_data = Vec 256 Data_reg
```

A.1.2 Design and Testbench

```haskell
-- Code for fully parallel FIR design, version 10.
module FIR10 where
import Clash.Prelude
import Clash.Explicit.Testbench
import qualified Data.List as L
import Data_types_FIR_acc_10
import FIR_accelerator_10_test_vectors

-- Top level function. Upon an external frame trigger (frame) the delay line is triggered and
    a new sample (sin) is injected at the tap which is indicated by the Num_taps input
fir_acc :: (Mem_data, Mem_coeff) -> (Sin,Bool,Mem_coeff,Bool,Num_taps) -> ((Mem_data,
    Mem_coeff), Sout)
fir_acc (regs, coeff_regs) (sin,tick,coeff,frame,num_taps) = case frame of
    True -> ((regs', coeff), sout)
    False -> ((regs, coeff), sout)
    where
        -- The regs' function shifts the delay line and inserts the new input at the correct
    place
        regs' =  replace (length(regs)-num_taps) sin (fst(shiftOutFromN d1 regs))
        -- The calc_out function is called upon to produce an output
        sout = calc_out num_taps coeff_regs regs sin

-- The output is calculated with entire delay line and coefficient bank, but enable signals
    should prevent unneccessary calculations from being done.
calc_out :: Num_taps -> Mem_coeff -> Mem_data -> Sin -> Sout
calc_out num_taps coeff regs sin = sum $ zipWith (*) coeff $ replace (length(regs)-num_taps-1)
     sin regs

-- An adjusted version of the clash mealy machine function. mm and its subfunctions xregs and
    yregs implement registers and define the default state, how the new state is calculated
    and how the enable signals (cVec) connect to the registers.
mm clk rst f cVec i = o
    where
      (s', o) = unbundle $ (f <$> s <*> i)
      (s1',s4') = unbundle s'
      s = bundle (s1,s4)
      s1 = xregs clk rst cVec s1'
      s4 = yregs clk rst cVec s4'

-- the xregs function defines the X-registers used, it is made a seperate function to be able
    to use the NOINLINE pragma, which will ensure that it becomes a seperate verilog entity
xregs clk rst cVec s1' = bundle $ zipWith (\en inp -> exposeClockResetEnable register clk rst
    en 0 inp) cVec (unbundle s1')

-- the yregs function defines the Y-registers used, it is made a seperate function to be able
    to use the NOINLINE pragma, which will ensure that it becomes a seperate verilog entity
yregs clk rst cVec s4' = bundle $ zipWith (\en inp -> exposeClockResetEnable register clk rst
    en 0 inp) cVec (unbundle s4')

-- Defining the topEntity, combining the sequential part as defined by the mm function with
    the combinational functionality as is defined in fir_acc
topEntity :: Clk -> Rst -> Vec 256 En -> Sig (Sin,Bool,Mem_coeff,Bool,Num_taps) -> Sig (Sout)
topEntity clk rst = mm clk rst fir_acc

--By using the NOINLINE pragma, seperate verilog entities are created for the top entity, the
    x registers, y regsiters and MAC hardware. This enables seperate power results per entity,
     creating categories.
{-# NOINLINE xregs #-}
{-# NOINLINE yregs #-}
{-# NOINLINE calc_out #-}
{-# NOINLINE topEntity #-}

--To automate the synthesis process more easily, the port names and entity name of the
    topEntity are explicitly defined
{-# ANN topEntity
  (Synthesize
    { t_name   = "FIR10"
    , t_inputs = [ PortName "clk"
                 , PortName "rst"
                 , PortName "en"
                 , PortName "inp" ]
    , t_output = PortName "outp"
    })
  #-}

-- TESTBENCH SECTION --
-- The code below this line is for the testbench generation and therefore will not be
    synthesized by the Synopsys tooling. The code below uses the test inputs as generated by
    matlab and inclued in the  file FIR_accelerator_10_test_vectors.hs

-- Appending generated input vectors with zeroes to fill up registers
coeff_vec_full = (replicate d156 0) ++ coeff_vec

-- Combining all inputs into one quintuple, as the mealy function expects only one argument as
     input
in_vec_comb = zip_n <$> (map (f_comb_inp coeff_vec_full (99 :: Num_taps)) in_vec) <*>
    frame_trig_vec
    where
      f_comb_inp x q (y,z) = (y,z,x,q)
      zip_n (y,z,x,q) w = (y,z,x,w,q)

-- Desired output generation, uses the test vectors generated by MATLAB.
f_out_vec regs_init coeff sins = souts
      where
          sout regs = foldl1 (+) (zipWith (*) regs coeff)
          regs' regs sin = fst(shiftInAt0 regs (sin:>Nil))
          delay_line = scanl (regs') regs_init sins
          souts = map sout (drop d1 delay_line)
```

```haskell

-- Calculates the expected samples on the rate fs (fs = fclk/256)
out_vec_fs = f_out_vec (reverse (take d100 in_vec_hs)) coeff_vec (drop d100 in_vec_hs)

-- Function to upsample the expected outputs at fs to fclk, basically repeating every output
    256 clock cycles.
f_out_vec_fclk out_vec_fs = (concat (map fs_to_fclk out_vec_fs))
    where
        fs_to_fclk sout = (replicate d256 (sout))

-- Calculating the expected output at fclk
out_vec_fclk = f_out_vec_fclk out_vec_fs

-- The testbench function which takes the inputs and expected output and compares that to the
    results of the actual FIR accelerator.
testBench :: Signal "Twenty" Bool
testBench = done
  where
    testInput    = stimuliGenerator clk rst in_vec_comb
    expectOutput = outputVerifier' clk rst ((replicate d100 0) ++ (0:>Nil) ++ (drop d1
    out_vec_fclk))
    done         = expectOutput (topEntity clk rst en testInput)
    en           = ((replicate d156 (toEnable (stimuliGenerator clk rst (False:>Nil)))) ++ (
    replicate d100 (toEnable (stimuliGenerator clk rst (True:>Nil)))))
    clk          = tbClockGen @"Twenty" (not <$> done)
    rst          = resetGen @"Twenty"

{-# ANN topEntity
  (TestBench 'testBench)
  #-}
```

A.2 Single Multiplier Design

A.2.1 Data Types

```haskell
module Data_types_DSP6 where
import Clash.Prelude
import Clash.Explicit.Testbench
import qualified Data.List as L

-- Define a clock domain with a 20 ns clock period (50 MHz)
createDomain vSystem{vName="Twenty", vPeriod=20000}

-- Define clock, reset, enable and data signals in the 50 MHz clock domain
type Clk = Clock "Twenty"
type Rst = Reset "Twenty"
type Sig = Signal "Twenty"
type En = Enable "Twenty"

-- Define boolean for indicating a new frame (sample) has arrived to reset program clock
type Frame_trig = Bool

-- Define samples and coefficients to be 32 bit signed fixed point values with 31 fractional
    bits
type Sin = SFixed 1 31
type Sout = SFixed 1 31
type Coeff = SFixed 1 31

-- Set types for internal registers
type Xreg = Sin
type Yreg = Coeff
type Acc_reg = SFixed 1 31
type Out_reg = Acc_reg

-- Base memory pointer
type XMem_addr = Unsigned 8
-- Y memory address
type YMem_addr = Unsigned 8
-- Program memory address, in this design this is equivalent to the program counter.
type PMem_addr = Unsigned 8

```

```
---- INSTRUCTION SET -----
-- Base memory pointer increment instruction (+1 or +0)
type Xbase_inc = Bool
-- Write enable for Xmem
type Xwr_en = Bool
-- End of Program Boolean
type Prog_jump = Bool
-- Memory pointer (points to absolute Ymem location and relative Xmem location)
type Mem_pnt = Unsigned 8
-- Boolean that determines whether the data in the accumulation register will be passed to the
     output
type Outp_instr = Bool
-- Boolean that determines whether the accumulation register should be updated
type Acc_en = Bool
-- Construction of the total instruction
type Instr = (Xbase_inc,Xwr_en,Prog_jump,Mem_pnt,Outp_instr,Acc_en)
```

A.2.2 Design and Testbench

```
-- code for Single Multiplier DSP design, version 6
module DSP6 where
import Clash.Prelude
import Clash.Explicit.Testbench
import qualified Data.List as L
import Data_types_DSP6
import DSP6_program1

-- Top function, here all subfunctions are combined into a single entity
dsp :: Clk -> Rst -> En -> En -> Sig Sin -> Sig Frame_trig -> Sig (Maybe (PMem_addr, Instr))
    -> Sig (Maybe (YMem_addr, Coeff)) -> Sig (Maybe XMem_addr) -> Sig Sout
dsp clk rst en en_mac sin frame_trig p_in y_in ext_wr = sout
    where
        --Program counter section
        prog_cnt' = prog_cnt_handle <$> prog_cnt <*> (get3rd <$> instr) <*> frame_trig
        prog_cnt = (exposeClockResetEnable register clk rst en_mac) (0 :: PMem_addr) prog_cnt'
        --X memory base pointer section
        xbase_rd' = xbase_rd_handle <$> xbase_rd <*> (get1st <$> instr)
        xbase_rd = (exposeClockResetEnable register clk rst en_mac) (0 :: XMem_addr) xbase_rd'
        --X memory write pointer section
        xpnt_wr' = xpnt_wr_handle <$> xpnt_wr <*> (get2nd <$> instr)
        xpnt_wr = (exposeClockResetEnable register clk rst en_mac) (0 :: XMem_addr) xpnt_wr'
        --Instruction is fetched from program memory
        instr = pmem clk rst en prog_cnt p_in
        --X register section, the xmem function handles both writing to and reading from the X
     memory
        xreg' = xmem clk rst en xbase_rd xpnt_wr instr sin ext_wr
        xreg = (exposeClockResetEnable register clk rst en_mac) (0 :: Xreg) xreg'
        --Y register section, the ymem function handles both writing to and reading from the Y
     memory
        yreg' = ymem clk rst en instr y_in
        yreg = (exposeClockResetEnable register clk rst en_mac) (0 :: Yreg) yreg'
        --Accumulator register section. Depending on the instruction the values in xreg and
    yreg are multiplied and added to the accumulation.
        acc_reg' = acc_handle <$> xreg <*> yreg <*> acc_reg <*> (get6th <$> instr) <*> (get5th
     <$> instr)
        acc_reg = (exposeClockResetEnable register clk rst en_mac) (0 :: Acc_reg) acc_reg'
        --Output register section. Depending on the instruction the value in the accumulator
    register is passed to the output register or not.
        sout = (exposeClockResetEnable register clk rst en_mac) (0 :: Out_reg) sout'
        sout' = calc_out <$> (get5th <$> instr) <*> sout <*> acc_reg

-- Handling function for the program counter:
  -- Set program counter to 0 when the frame trigger is high
  -- Set program counter to last address of program memory (255) when the end of program (
    Prog_jump) bit in the instruction is high
  -- Once end of program is triggered and program counter is set to 255, keep program counter
     at 255 to prevent further data manipulation until the new frame trigger arrives
prog_cnt_handle :: PMem_addr -> Prog_jump -> Frame_trig -> PMem_addr
prog_cnt_handle _ _ True = (0 :: PMem_addr)
prog_cnt_handle _ True False = (255 :: PMem_addr)
prog_cnt_handle 255 _ False = (255 :: PMem_addr)
prog_cnt_handle prog_cnt False False = prog_cnt + 1
```

```haskell

-- Handling function for the X memory base pointer:
  -- Increase base pointer when Xbase_inc in the instruction is True
  -- When the base address reaches the maximum address, circle back to 0
xbase_rd_handle :: XMem_addr -> Bool -> XMem_addr
xbase_rd_handle baddr False = baddr
xbase_rd_handle baddr True
      | baddr == (255 :: XMem_addr) = (0 :: XMem_addr)
      | otherwise = baddr + 1

-- Handling function for the X memory write pointer, which has the identical behaviour as the handling for the base pointer, but triggers on the Xwr_en part of the instruction instead
xpnt_wr_handle :: XMem_addr -> Bool -> XMem_addr
xpnt_wr_handle addr False = addr
xpnt_wr_handle addr True
      | addr == (255 :: XMem_addr) = (0 :: XMem_addr)
      | otherwise = addr + 1

-- Instantiation of the program memory, the asyncRamPow2 function is used. This function is native to Clash and implements an asynchronous read, synchronous write RAM.
-- The bit-depth of the program memory automatically scales with the defined instruction set. The word depth is dependant on the size of the PMem_addr, which leads to a program memory of 256 words.
pmem :: Clk -> Rst -> En -> Sig PMem_addr -> Sig (Maybe (PMem_addr,Instr)) -> Sig Instr
pmem clk rst en prog_cnt p_in = (exposeClockResetEnable asyncRamPow2 clk rst en) prog_cnt p_in

-- Instantiation of the X memory, here the absolute memory address for reading is derived from the base address and the memory pointer section of the instruction.
  -- The write handle function is called to obtain the absolute write address when a value needs to be written to the X memory
xmem :: Clk -> Rst -> En -> Sig XMem_addr -> Sig XMem_addr -> Sig Instr -> Sig Sin -> Sig (Maybe XMem_addr) -> Sig Xreg
xmem clk rst en baddr xpnt_wr instr sin ext_wr = (exposeClockResetEnable asyncRamPow2 clk rst en) raddr wrinp
  where
        raddr = (-) <$> baddr <*> (get4th <$> instr) -- Relies on underflow to wrap around and start at the bottom address when (baddr - mem_pnt) becomes negative
        wrinp = wr_handle <$> (get2nd <$> instr) <*> xpnt_wr <*> sin <*> ext_wr -- writes sin to memory when instruction tells it to or when external write is a (Just addr)

-- Instantiation of the Y memory. Like the X memory, the Y memory can be written to via an external write port. That is however the only method to write to the Y memory.
-- The Y memory is not circular and does not use relative memory pointers, so the read address is simply the memory pointer section from the instruction.
ymem :: Clk -> Rst -> En -> Sig Instr -> Sig (Maybe (YMem_addr, Coeff)) -> Sig Yreg
ymem clk rst en instr y_in = (exposeClockResetEnable asyncRamPow2 clk rst en) raddr y_in
    where
       raddr = get4th <$> instr


-- Write handling function which:
  -- Instructs X memory to write to the address specified by the external write port, which is used when filling the memory at startup
  -- When the Xwr_en part of the instruction is True, instruct X memory to write data from the sin port to the write address pointer
  -- When the above two scenarios do not apply, instruct the memory to not write
wr_handle :: Bool -> XMem_addr -> Sin -> Maybe XMem_addr -> Maybe (XMem_addr,Sin)
wr_handle _ _ sin (Just ext_addr) = Just (ext_addr,sin)
wr_handle False _ _ Nothing = Nothing
wr_handle True wr_addr sin Nothing = Just (wr_addr,sin)

-- MAC function:
  -- When the Outp_instr part of the instruction is True the output of the accumulator is sent to the output port and the accumulation register is set to 0.
  -- When the acc_en section of the instruction is True, multiply the values in the X and Y registers and add the result to the accumulator register
  -- When the acc_en section of the instruction is False, do not update the value in the accumulator register
acc_handle :: Xreg -> Yreg -> Acc_reg -> Acc_en -> Outp_instr -> Acc_reg
acc_handle xreg yreg _ _ True = (0 :: Acc_reg)
acc_handle xreg yreg acc_reg True _ = xreg * yreg + acc_reg
acc_handle _ _ acc_reg False _ = acc_reg

```

```haskell
-- Handling function of output register:
  -- When the Outp_instr section of the instruction is True, update the value in the output
    register with the value in the accumulation register
  -- When the Outp_instr section of the instruction is False, do not update the value of the
    output register
calc_out :: Outp_instr -> Sout -> Acc_reg -> Sout
calc_out True _ acc_reg = acc_reg
calc_out False sout _ = sout

-- Help functions to extract specific sections from the instruction sextuple
get1st (a,_,_,_,_,_) = a
get2nd (_,a,_,_,_,_) = a
get3rd (_,_,a,_,_,_) = a
get4th (_,_,_,a,_,_) = a
get5th (_,_,_,_,a,_) = a
get6th (_,_,_,_,_,a) = a

get1st3 (a,_,_) = a

-- define dsp as topEntity
topEntity = dsp

--To automate the synthesis process more easily, the port names and entity name of the
    topEntity are explicitly defined
{-# ANN topEntity
  (Synthesize
    { t_name   = "DSP6"
    , t_inputs = [ PortName "clk"
                 , PortName "rst"
                 , PortName "en"
                 , PortName "en_mac"
                 , PortName "inp"
                 , PortName "frame_trig"
                 , PortName "p_in"
                 , PortName "y_in"
                 , PortName "ext_wr"
                  ]
    , t_output = PortName "outp"
    })
  #-}

-- No inline certain functions to analyse power of those pieces seperately
{-# NOINLINE xmem #-}
{-# NOINLINE ymem #-}
{-# NOINLINE pmem #-}
{-# NOINLINE acc_handle #-}
{-# NOINLINE topEntity #-}

-- TESTBENCH SECTION --
-- The code below this line is for the testbench generation and therefore will not be
    synthesized by the Synopsys tooling. The code below uses the test inputs as generated by
    matlab and inclued in the  file DSP6_program1.hs

-- Cast programming vectors generated by matlab in the correct format
p_in_vec = Just <$> (zip (iterate d256 (+1) (0 :: PMem_addr)) program1)
coeff_in_vec = Just <$> (zip (iterate d100 (+1) (0 :: YMem_addr)) coeff_vec)

-- Desired output generation, uses the test vectors generated by MATLAB.
f_out_vec regs_init coeff sins = souts
      where
          sout regs = foldl1 (+) (zipWith (*) regs coeff)
          regs' regs sin = replace 0 sin (fst(shiftOutFromN d1 regs))
          delay_line = scanl (regs') regs_init sins
          souts = map sout (drop d1 delay_line)
out_vec_fs = f_out_vec regs_init coeff_vec in_vec_hs

-- Function to upsample the expected outputs at fs to fclk, basically repeating every output
    256 clock cycles.
f_out_vec_fclk out_vec_fs = (replicate output_offset (0 :: Sout)) ++ (concat (map fs_to_fclk
    out_vec_fs))
    where
        fs_to_fclk sout = (replicate d256 (sout))

```

```haskell
-- Calculates the expected samples on the rate fs (fs = fclk/256)
out_vec_fclk = f_out_vec_fclk out_vec_fs

testBench :: Signal "Twenty" Bool
testBench = done
  where
      done      = outputVerifier' clk rst out_vec_fclk (ignoreFor clk rst en d1 0 (topEntity
    clk rst en en_mac testInput frame_trig p_in y_in ext_wr))
      clk       = tbClockGen @"Twenty" (not <$> done)
      rst       = resetGen @"Twenty"
      en        = tbEnableGen
      en_mac    = toEnable (stimuliGenerator clk rst ((replicate d256 False) ++ (True:>Nil)))
      testInput = stimuliGenerator clk rst in_vec
      p_in      = stimuliGenerator clk rst p_in_vec
      y_in      = stimuliGenerator clk rst coeff_in_vec
      ext_wr    = stimuliGenerator clk rst ((Just <$> (iterate d256 (+1) (0 :: XMem_addr))) ++
     (Nothing :> Nil))
      frame_trig= stimuliGenerator clk rst frame_trig_vec


{-# ANN topEntity
  (TestBench 'testBench)
  #-}
```

A.3 n-Multiplier Design

A.3.1 Data Types

```haskell
module Data_types_DSP13_num_mem_2 where
import Clash.Prelude
import Clash.Explicit.Testbench
import Clash.Promoted.Nat

--The Num_mem type is the "knob" that determines the amount of memories and multipliers.
type Num_mem=2

-- Define a clock domain with a 20 ns clock period (50 MHz)
createDomain vSystem{vName="Twenty", vPeriod=20000}

-- Define clock, reset, enable and data signals in the 50 MHz clock domain
type Clk = Clock "Twenty"
type Rst = Reset "Twenty"
type Sig = Signal "Twenty"
type En = Enable "Twenty"

-- Define boolean for indicating a new frame (sample) has arrived to reset program clock
type Frame_trig = Bool

-- Define samples and coefficients to be 32 bit signed fixed point values with 31 fractional
    bits
type Sin = SFixed 1 31
type Sout = SFixed 1 31
type Coeff = SFixed 1 31

-- Set types for internal registers
type Xreg = Sin
type Yreg = Coeff
type Acc_reg = SFixed 1 31
type Out_reg = Acc_reg

-- Base memory pointer, each memory will have 256/num_mem addresses, rounded upwards
type XMem_addr = Index (DivRU 256 Num_mem)
-- Y memory address, each memory will have 256/num_mem addresses, rounded upwards
type YMem_addr = Index (DivRU 256 Num_mem)
-- Program memory address, for every design a 256 word program memory is used.
type PMem_addr = Unsigned 8

-- Shift counter which will instruct the barrel shifter how to match the correct sample and
    coefficient
type Shft_cnt = Index Num_mem
-- Write select counter which will indicate which memory to write new samples to
```

```haskell
type Wr_sel = Index Num_mem

---- INSTRUCTION SET -----
-- Base memory pointer increment instruction (+1 or +0)
type Xbase_inc = Bool
-- Write enable for Xmem
type Xwr_en = Bool
-- End of Program Boolean
type Prog_jump = Bool
-- Memory pointer (points to absolute Ymem location and relative Xmem location)
type Mem_pnt = Index (DivRU 256 Num_mem)
-- Boolean that determines whether the data in the accumulation register will be passed to the
     output
type Outp_instr = Bool
-- Boolean that determines whether the accumulation register should be updated
type Acc_en = Bool
-- Construction of the total instruction
type Instr = (Xbase_inc,Xwr_en,Prog_jump,Mem_pnt,Outp_instr,Acc_en)
```

A.3.2 Top Design and Testbench

```haskell
import Clash.Prelude
import DSP13_program1_num_mem_2
import Data_types_DSP13_num_mem_2
import Clash.Explicit.Testbench
import Clash.Sized.Internal.BitVector
import Clash.Sized.Internal.Index
import qualified Data.List as L
import DSP_barrel_shifters
import qualified Clash.Explicit.Signal as E

-- Top function, here all subfunctions are combined into a single entity
dsp13 :: Clk -> Rst -> En -> En -> Sig Sin -> Sig Frame_trig -> Sig (Maybe (PMem_addr, Instr))
     -> (Vec Num_mem (Sig (Maybe (YMem_addr, Coeff)))) -> Sig (Vec Num_mem (Maybe (XMem_addr,
    Sin))) -> Sig Sout
dsp13 clk rst en en_mac sin frame_trig p_in y_ins ext_wr = sout
  where
    --Program counter section
    prog_cnt' = prog_cnt_handle <$> prog_cnt <*> (get3rd <$> instr) <*> frame_trig
    prog_cnt = (exposeClockResetEnable register clk rst en_mac) (0 :: PMem_addr) prog_cnt'
    --Instruction is fetched from program memory
    instr = pmem clk rst en prog_cnt p_in
    --mem_slect_rd is a register containing num_mem amount of bits to indicate which memory
    should be read at (base_address + mem_pnt) and which should be read at (base_address +
    mem_pnt - 1)
    mem_select_rd' = unbundle $ f_mem_select_rd <$> mem_select_rd <*> (get1st <$> instr)
    mem_select_rd = (exposeClockResetEnable register clk rst en_mac) (fst(f_mem_select_rd (
    replicate (SNat @Num_mem) (0 :: Bit)) True)) (fst mem_select_rd')
    --xbase_rd contains the base address for the x memory, every num_mem FIR cycles the base
    address in increased by one.
    --  (snd mem_select_rd') is the boolean that indicates when the base address should be
    increased
    xbase_rd' = xaddr_handle <$> xbase_rd <*> (snd mem_select_rd')
    xbase_rd = (exposeClockResetEnable register clk rst en_mac) (0 :: XMem_addr) xbase_rd'
    --mem_select_wr contains the information which of the num_mem X memories should be written
     to when a new sample arrives
    -- mem_select_wr increases by one every time a sample is written and when num_mem is
    reached it is reset to zero
    mem_select_wr' = unbundle $ f_mem_select_wr <$> mem_select_wr <*> (get2nd <$> instr)
    mem_select_wr = (exposeClockResetEnable register clk rst en_mac) (0 :: Wr_sel) (fst
    mem_select_wr')
    --xpnt_wr contains the absolute write address where in a X memory a new sample should be
    written
    -- xpnt_wr is increased by one every num_mem times a sample is written to the x memories
    xpnt_wr' = xaddr_handle <$> xpnt_wr <*> (snd mem_select_wr')
    xpnt_wr = (exposeClockResetEnable register clk rst en_mac) (0 :: XMem_addr) xpnt_wr'
    --x_rd_addrs is a vector of absolute X memory addresses. These addresses are either (base
    address + memory pointer) or (base address + memory pointer + 1)
    x_rd_addrs = unbundle $ x_rd_addrs_handle <$> xbase_rd <*> mem_select_rd <*> instr
    --x_wr_inps is a vector of write instructions for the X memories. This vector contains the
     information which X memory is written to and at what address.
    x_wr_inps = x_wr_inps_handle <$> xpnt_wr <*> mem_select_wr <*> sin <*> (get2nd <$> instr)
    --shft_cnt is a shift counter that is used to instruct the barrel shifter how to match the
     samples from the X memory to the correct coefficients of the Y register
    -- shft_cnt is increased when the base read address is increased, which happens every full
     FIR cycle. When the maximum value is reached the value wraps around to zero.
    shft_cnt' = f_shft_cnt_inc <$> shft_cnt <*> (get1st <$> instr)
    shft_cnt = (exposeClockResetEnable register clk rst en_mac) (1 :: Shft_cnt) shft_cnt'
    --The xmem_outs function interfaces with all the X memories. Mapping the read and write
    instructions constructed in x_rd_addrs, x_wr_inps and ext_wr_handle to an absolute read/
    write instruction per X memory.
    xmem_outs = (\x_rd_addr x_wr_inp -> (xmem clk rst en) x_rd_addr x_wr_inp) <$> x_rd_addrs
    <*> (unbundle (ext_wr_handle <$> x_wr_inps <*> ext_wr))
    --xregs computes the new value in the x registers by taking the outputs of the X memory (
    xmem_outs) and shifting the samples with the barrel shifter. Now the samples in the X
    registers match with the coefficients in the Y registers.
    xregs' = unbundle $ reverse <$> (barrel_shift <$> (bundle xmem_outs) <*> shft_cnt)
    --The xregs function creates the X registers, it is made a seperate function to enable
    seperate power analysis on that specific hardware.
    xregs_i = xregs clk rst en_mac xregs'
    --The Y registers are filled with values directly from the Y memory, without any barrel
    shifting of the samples.
    yregs' = (\yin -> ymem clk rst en (get4th <$> instr) yin) <$> y_ins
    --The yregs function creates the Y registers, it is made a seperate function to enable
    seperate power analysis on that specific hardware.
    yregs_i = yregs clk rst en_mac yregs'
    --acc_handle computes the output value, creating the num_mem multipliers and adder tree
    hardware. The output of the MAC is stored in the acc_reg register.
    -- Depending on the acc_en section of the instruction the output is propagated to the
    accumulation register. When the outp_instr section is high, the accumulator register is
    set to zero.
    acc_reg' = acc_handle <$> (bundle xregs_i) <*> (bundle yregs_i) <*> acc_reg <*> (get6th <$
    > instr) <*> (get5th <$> instr)
    acc_reg = (exposeClockResetEnable register clk rst en_mac) (0 :: Acc_reg) acc_reg'
    --Output register section. Depending on the instruction the value in the accumulator
    register is passed to the output register or not.
    sout = (exposeClockResetEnable register clk rst en_mac) (0 :: Out_reg) sout'
    sout' = calc_out <$> (get5th <$> instr) <*> sout <*> acc_reg

-- Handling function for the program counter:
  -- Set program counter to 0 when the frame trigger is high
  -- Set program counter to last address of program memory (255) when the end of program (
    Prog_jump) bit in the instruction is high
  -- Once end of program is triggered and program counter is set to 255, keep program counter
    at 255 to prevent further data manipulation until the new frame trigger arrives
prog_cnt_handle :: PMem_addr -> Prog_jump -> Frame_trig -> PMem_addr
prog_cnt_handle _ _ True = (0 :: PMem_addr)
prog_cnt_handle _ True False = (255 :: PMem_addr)
prog_cnt_handle 255 _ False = (255 :: PMem_addr)
prog_cnt_handle prog_cnt False False = prog_cnt + 1

--Creation of the vector of X registers by mapping the xreg function, through this hierarchy
    of seperate functions power categories can be formed for both all the x registers together
     and for every x register seperately.
xregs :: Clk -> Rst -> En -> Vec Num_mem (Sig Xreg) -> Vec Num_mem (Sig Xreg)
xregs clk rst en_mac xregs' = (xreg clk rst en_mac) <$> xregs'

--Creation of the vector of Y registers by mapping the yreg function, through this hierarchy
    of seperate functions power categories can be formed for both all the y registers together
     and for every y register seperately.
yregs :: Clk -> Rst -> En -> Vec Num_mem (Sig Yreg) -> Vec Num_mem (Sig Yreg)
yregs clk rst en_mac yregs' = (yreg clk rst en_mac) <$> yregs'

--Creation of a single X register
xreg :: Clk -> Rst -> En -> Sig Xreg -> Sig Xreg
xreg clk rst en_mac xreg' = (exposeClockResetEnable register clk rst en_mac) (0 :: Xreg) xreg'

--Creation of a single Y register
yreg :: Clk -> Rst -> En -> Sig Yreg -> Sig Yreg
yreg clk rst en_mac yreg' = (exposeClockResetEnable register clk rst en_mac) (0 :: Yreg) yreg'

--Handling function for the external write functionality of the X memory
  -- When an external write input is present (when the design is programmed) set the write
    input to the memories to that external input
  -- When the external write input is not present (a vector of nothings), set the write input
```

```
to the memories to the computed write inputs. This is the behaviour during normal operation
ext_wr_handle :: Vec Num_mem (Maybe (XMem_addr,Sin)) -> Vec Num_mem (Maybe (XMem_addr,Sin)) -> Vec Num_mem (Maybe (XMem_addr,Sin))
ext_wr_handle x_wr_inps ext_wr
        | ext_wr == (replicate (SNat @ Num_mem) Nothing) = x_wr_inps
        | otherwise = ext_wr

--Handling function for the X memory address offset. The resulting vector indicates which memories should be read at (base address - mem pointer) and which should be read at (base address - mem pointer - 1).
  -- The vector only changes when the Xbase_inc section of the instruction is True
  -- Bitwise operations are used to incrementally shift in 0's until all bits in the vector are 0's, then the vector resets to all 1's except for the first bit. example for num_mem = 4:
  -- cycle1: (0 1 1 1,F)
  -- cycle2: (0 0 1 1,F)
  -- cycle3: (0 0 0 1,F)
  -- cycle4: (0 0 0 0,T) -> By setting the second argument of the tuple to true the base pointer is increased for next cycle
  -- cycle5: (0 1 1 1,F) -> Cycle starts again but base pointer is now increased
f_mem_select_rd ::  Vec Num_mem Bit -> Xbase_inc -> (Vec Num_mem Bit,Bool)
f_mem_select_rd  vb_in (False) = (vb_in,False)
f_mem_select_rd  vb_in (True)  = (((0 :> Nil) ++ (tail vb_new)),(bitToBool (head vb_new)))
    where
        vb_new = map (or## (complement## (last vb_in))) shift
        shift = fst(shiftOutFromN d1 vb_in)

--Handling function for the shift counter used by the barrel shifter
-- When the Xbase_inc section of the instruction is True increase the shift counter, when the maximum is reached reset shift counter to zero.
f_shft_cnt_inc :: Shft_cnt -> Xbase_inc -> Shft_cnt
f_shft_cnt_inc shft_cnt False = shft_cnt
f_shft_cnt_inc shft_cnt True
      | shft_cnt == (maxBound::Shft_cnt) = (0 :: Shft_cnt)
      | otherwise = shft_cnt + 1

--Handling function used for both the X memory base read pointer and the X memory write pointer. Increase when the relevant part of the instruction is True and handle overflow.
xaddr_handle :: XMem_addr -> Bool -> XMem_addr
xaddr_handle addr False = addr
xaddr_handle addr True
      | addr == (maxBound::XMem_addr) = (0 :: XMem_addr)
      | otherwise = addr + 1

--Handling function used to generate absolute addresses for the X memories using the base address, instruction and memory select bitvector
  -- The f_add_base_mem_select handling function is mapped over the bitvector creating a vector of Num_mem absolute adresses
  -- The f_sub_base_mem_pnt handling function calculates base_addr - mem_pnt with overflow handling
x_rd_addrs_handle :: XMem_addr -> Vec Num_mem Bit -> Instr -> Vec Num_mem XMem_addr
x_rd_addrs_handle xbase_rd mem_select instr = map (f_add_base_mem_select a) mem_select
    where
      a = f_sub_base_mem_pnt xbase_rd mem_pnt
      mem_pnt = get4th instr

--Handling function to calculate absolute memory address for one memory by either passing through (base_addr - mem_pnt) when the bit from the bitvector is zero or (base_addr - mem_pnt - 1) when the bit in the bitvector is one. Additionally, handle proper wrap around behaviour when (base address - mem_pnt) is zero.
f_add_base_mem_select :: XMem_addr -> Bit -> XMem_addr
f_add_base_mem_select xbase_rd 0 = xbase_rd
f_add_base_mem_select 0 1 = (maxBound::XMem_addr)
f_add_base_mem_select xbase_rd 1 = xbase_rd-1

--Handling function to calculate base address - memory pointer with overflow handling.
  -- Subtracts the memory pointer part of the instruction from the base address and ensures proper wrap around behaviour.
f_sub_base_mem_pnt :: XMem_addr -> Mem_pnt -> XMem_addr
f_sub_base_mem_pnt baddr mem_pnt
    | baddr < mem_pnt = (maxBound :: XMem_addr) - (mem_pnt-baddr - 1)
    | otherwise = baddr - mem_pnt
```

```haskell

--Handling function for the computation of the X memory write select variable
-- When the Xwr_en section of the instruction is False, do not update write select variable
-- When the Xwr_en section of the instruction is True, increase the write select variable and when the maximum is reached, set write select variable to zero and set the boolean in the second part of the tuple to True to indicate that the write pointer should be increased.
f_mem_select_wr :: Wr_sel -> Xwr_en -> (Wr_sel,Bool)
f_mem_select_wr wr_sel False = (wr_sel,False)
f_mem_select_wr wr_sel True
      | wr_sel == (maxBound::Wr_sel) = ((0 :: Wr_sel),True)
      | otherwise = (wr_sel + 1,False)

--Handling function which creates a vector of Write instructions in the format required by the asyncRam function.
  -- When the Xwr_en section of the instruction is True create a num_mem wide vector with 1 element filled with the write address and new sample.
  -- Which element in the vector contains the new sample is determined by the Wr_sel value, using an imap to replace the value in the vector at a certain index.
x_wr_inps_handle :: XMem_addr -> Wr_sel -> Sin -> Xwr_en -> Vec Num_mem (Maybe (XMem_addr,Sin))
x_wr_inps_handle _ _ _ False = replicate (SNat @ Num_mem) Nothing
x_wr_inps_handle wr_base wr_sel sin True = f_rep_ind wr_sel (Just (wr_base,sin)) (replicate (SNat @ Num_mem) Nothing)
    where
      f_rep_ind wr_sel a vec = imap (\i vec -> if i == wr_sel then a else Nothing) vec

-- Instantiation of the program memory, the asyncRamPow2 function is used. This function is native to Clash and implements an asynchronous read, synchronous write RAM.
-- The bit-depth of the program memory automatically scales with the defined instruction set. The word depth is dependant on the size of the PMem_addr, which leads to a program memory of 256 words.
pmem :: Clk -> Rst -> En -> Sig PMem_addr -> Sig (Maybe (PMem_addr,Instr)) -> Sig Instr
pmem clk rst en prog_cnt p_in = (exposeClockResetEnable asyncRamPow2 clk rst en) prog_cnt p_in

xmem :: Clk -> Rst -> En -> Sig XMem_addr -> Sig (Maybe (XMem_addr,Sin)) -> Sig Xreg
xmem clk rst en raddr wrinp = (exposeClockResetEnable asyncRam clk rst en (SNat @ (DivRU 256 Num_mem))) raddr wrinp

ymem :: Clk -> Rst -> En -> Sig Mem_pnt -> Sig (Maybe (YMem_addr, Coeff)) -> Sig Yreg
ymem clk rst en raddr y_in = (exposeClockResetEnable asyncRam clk rst en (SNat @ (DivRU 256 Num_mem))) raddr y_in

-- MAC function:
  -- When the Outp_instr part of the instruction is True the output of the accumulator is sent to the output port and the accumulation register is set to 0.
  -- When the acc_en section of the instruction is True, multiply the values of each corresponding X and Y register and sum all the results, then add this to the accumulation register
  -- When the acc_en section of the instruction is False, do not update the value in the accumulator register
acc_handle :: Vec Num_mem Xreg -> Vec Num_mem Yreg -> Acc_reg -> Acc_en -> Outp_instr -> Acc_reg
acc_handle xregs yregs _ _ True = (0 :: Acc_reg)
acc_handle xregs yregs acc_reg True _ = (sum (zipWith (*) xregs yregs)) + acc_reg
acc_handle _ _ acc_reg False _ = acc_reg


-- Handling function of output register:
  -- When the Outp_instr section of the instruction is True, update the value in the output register with the value in the accumulation register
  -- When the Outp_instr section of the instruction is False, do not update the value of the output register
calc_out :: Outp_instr -> Sout -> Acc_reg -> Sout
calc_out True _ acc_reg = acc_reg
calc_out False sout _ = sout

-- Barrel shifter function generated by MATLAB, with log2(Num_mem) stages.
barrel_shift :: Vec Num_mem Xreg -> Shft_cnt -> Vec Num_mem Xreg
barrel_shift vec_in cnt = fst $ ((vec_shft_1 (shft_en !! 0))) (vec_in,d0)
    where
        shft_en = reverse (to_bool_vec cnt)

--Barrel shifter help function
```

```haskell
to_bool_vec :: Shft_cnt -> Vec (CLog 2 Num_mem) Bool
to_bool_vec cnt = unpack (pack cnt) :: Vec (CLog 2 Num_mem) Bool

-- Help functions to extract specific sections from the instruction sextuple
get1st (a,_,_,_,_,_) = a
get2nd (_,a,_,_,_,_) = a
get3rd (_,_,a,_,_,_) = a
get4th (_,_,_,a,_,_) = a
get5th (_,_,_,_,a,_) = a
get6th (_,_,_,_,_,a) = a

get1st3 (a,_,_) = a

-- define dsp13 as topEntity
topEntity = dsp13

--To automate the synthesis process more easily, the port names and entity name of the
    topEntity are explicitly defined
{-# ANN topEntity
    (Synthesize
    { t_name   = "DSP13_num_mem_2"
    , t_inputs = [ PortName "clk"
                 , PortName "rst"
                 , PortName "en"
                 , PortName "en_mac"
                 , PortName "inp"
                 , PortName "frame_trig"
                 , PortName "p_in"
                 , PortName "y_in"
                 , PortName "ext_wr"
                  ]
    , t_output = PortName "outp"
    })
  #-}

-- No inline certain functions to analyse power of those pieces seperately
{-# NOINLINE xmem #-}
{-# NOINLINE ymem #-}
{-# NOINLINE pmem #-}
{-# NOINLINE acc_handle #-}
{-# NOINLINE xreg #-}
{-# NOINLINE yreg #-}
{-# NOINLINE x_rd_addrs_handle #-}
{-# NOINLINE x_wr_inps_handle #-}
{-# NOINLINE barrel_shift #-}
{-# NOINLINE topEntity #-}

-- TESTBENCH SECTION --
-- The code below this line is for the testbench generation and therefore will not be
    synthesized by the Synopsys tooling. The code below uses the test inputs as generated by
    matlab and inclcuded in the  file DSP6_program1_num_mem_2.hs


-- Cast programming vectors generated by matlab in the correct format
p_in_vec = Just <$> (zip (iterate d256 (+1) (0 :: PMem_addr)) program1)

-- Desired output generation, uses the test vectors generated by MATLAB.
f_out_vec regs_init coeff sins = souts
      where
          sout regs = foldl1 (+) (zipWith (*) regs coeff)
          regs' regs sin = replace 0 sin (fst(shiftOutFromN d1 regs))
          delay_line = scanl (regs') regs_init sins
          souts = map sout (drop d1 delay_line)
out_vec_fs = f_out_vec regs_init coeff_vec_hs in_vec_hs

-- Function to upsample the expected outputs at fs to fclk, basically repeating every output
    256 clock cycles.
f_out_vec_fclk out_vec_fs = (replicate output_offset (0 :: Sout)) ++ (concat (map fs_to_fclk
    out_vec_fs))
    where
        fs_to_fclk sout = (replicate d256 (sout))

-- Calculates the expected samples on the rate fs (fs = fclk/256)
```

```haskell
out_vec_fclk = f_out_vec_fclk out_vec_fs

testBench :: Signal "Twenty" Bool
testBench = done
  where
      done       = outputVerifier' clk rst out_vec_fclk (ignoreFor clk rst en d1 0 (topEntity
    clk rst en en_mac testInput frame_trig p_in y_in ext_wr))
      clk        = tbClockGen @"Twenty" (not <$> done)
      rst        = resetGen @"Twenty"
      en         = tbEnableGen
      en_mac     = toEnable (stimuliGenerator clk rst ((replicate d259 False) ++ (True:>Nil)))
      testInput = stimuliGenerator clk rst in_vec
      p_in       = stimuliGenerator clk rst ((Nothing :> Nil) ++ p_in_vec ++ (Nothing :> Nil))
      y_in       = unbundle $ stimuliGenerator clk rst (((replicate (SNat @Num_mem) Nothing):>
    Nil) ++ coeff_vec ++ ((replicate (SNat @Num_mem) Nothing):> Nil))       ext_wr     =
    stimuliGenerator clk rst (((replicate (SNat @Num_mem) Nothing):> Nil) ++ ext_wr_vec ++ ((
    replicate (SNat @Num_mem) Nothing):> Nil))
      frame_trig= stimuliGenerator clk rst frame_trig_vec

{-# ANN topEntity
   (TestBench 'testBench)
  #-}
```

A.3.3 Barrel Shifter Stages

```haskell
module DSP_barrel_shifters where
import Clash.Prelude
-- Barrel shifter functions for DSP implementations with multiple memories. To be able to
    combine different static barrel shifters multiples of the same function description are
    necessary
-- but each one is called with a different SNat for the amount rotated. Since this requires a
    different type for each function unique function names are needed.
-- the functions are instantiated by a line in a DSP_program file generated by Matlab
    depending on the amount of barrel shifting needed.

-- Up to 8 unique barrel shifters are supported

vec_shft_1 :: KnownNat n => Bool -> (Vec n a1,SNat 0) -> (Vec n a1,SNat 1)
vec_shft_1 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_1 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d1

vec_shft_2 :: KnownNat n => Bool -> (Vec n a1,SNat 1) -> (Vec n a1,SNat 2)
vec_shft_2 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_2 True (vec_in,pow)  = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d2

vec_shft_3 :: KnownNat n => Bool -> (Vec n a1,SNat 2) -> (Vec n a1,SNat 3)
vec_shft_3 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_3 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d4

vec_shft_4 :: KnownNat n => Bool -> (Vec n a1,SNat 3) -> (Vec n a1,SNat 4)
vec_shft_4 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_4 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d8

vec_shft_5 :: KnownNat n => Bool -> (Vec n a1,SNat 4) -> (Vec n a1,SNat 5)
vec_shft_5 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_5 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d16

vec_shft_6 :: KnownNat n => Bool -> (Vec n a1,SNat 5) -> (Vec n a1,SNat 6)
vec_shft_6 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_6 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d32
```

```

vec_shft_7 :: KnownNat n => Bool -> (Vec n a1,SNat 6) -> (Vec n a1,SNat 7)
vec_shft_7 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_7 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d64

vec_shft_8 :: KnownNat n => Bool -> (Vec n a1,SNat 7) -> (Vec n a1,SNat 8)
vec_shft_8 False (vec_in,pow) = (vec_in,addSNat pow d1)
vec_shft_8 True (vec_in,pow) = (rotateLeftS vec_in num_shft, addSNat pow d1)
    where
        num_shft = d128
```

B CLASH PRIMITIVES

Primitives are instructions for Clash on how to implement a certain function in a HDL (Verilog in the cases below). The primitive declaration at the top of a file indicates the arguments that are given to the Clash function and how they correspond to Verilog arguments.

B.1 SRAM Primitive

```
[ { "BlackBox" :
    { "name" : "Clash.Explicit.RAM.asyncRam#"
    , "kind" : "Declaration"
    , "type" :
"asyncRam#
  :: ( HasCallStack                -- ARG[0]
     , KnownDomain wdom wconf      -- ARG[1]
     , KnownDomain rdom rconf )    -- ARG[2]
  => Clock wdom                    -- ^ wclk, ARG[3]
  -> Clock rdom                    -- ^ rclk, ARG[4]
  -> Enable wdom                   -- ^ wen,  ARG[5]
  -> SNat n                        -- ^ sz,   ARG[6]
  -> Signal rdom Int               -- ^ rd,   ARG[7]
  -> Signal wdom Bool              -- ^ en,   ARG[8]
  -> Signal wdom Int               -- ^ wr,   ARG[9]
  -> Signal wdom a                 -- ^ din,  ARG[10]
  -> Signal rdom a"
    , "template" :
"// DesignWare SRAM instantiation
wire [$clog2( LIT[6])-1:0]  GENSYM[rd_addr_int][0];
wire [$clog2( LIT[6])-1:0]  GENSYM[wr_addr_int][1];
wire signed [63:0]  GENSYM[rd_addr_int_big][2];
wire signed [63:0]  GENSYM[wr_addr_int_big][3];

assign  SYM[2] =  ARG[7];
assign  SYM[3] =  ARG[9];

assign  SYM[0] =  SYM[2][$clog2( LIT[6])-1:0];
assign  SYM[1] =  SYM[3][$clog2( LIT[6])-1:0];
DW_ram_r_w_s_dff #( SIZE[ TYP0],  LIT[6])
 GENSYM[U][4] (.clk( ARG[3]), .rst_n(!rst), .cs_n(1'b0), .wr_n(! ARG[8]), .rd_addr( SYM[0]), .
    wr_addr( SYM[1]), .data_in( ARG[10]), .data_out( RESULT));
// DesignWare SRAM end"
    }
  }
]
```

B.2 Clock Multiplexer Inline Primitive

```
--
    ---------------------------------------------------------------------------------------------------------------
{-# LANGUAGE QuasiQuotes #-}
module InlinePrimitive where

import              Clash.Annotations.Primitive
import              Clash.Prelude
import              Data.String.Interpolate      (i)
import            Data.String.Interpolate.Util (unindent)

{-# ANN clkMux (InlinePrimitive [Verilog] $ unindent [i|
  [ { "BlackBox" :
      { "name" : "InlinePrimitive.clkMux"
      , "kind": "Declaration"
      , "template" :
  "// begin InlinePrimitive clkMux:
  assign  RESULT =  ARG[0] &  ARG[1];
  // end InlinePrimitive clkMux"
      }
    }
  ]
  |]) #-}
{-# NOINLINE clkMux #-}
clkMux :: Signal dom Bool -> Clock dom -> Clock dom
clkMux _ (clk) = clk

--
    ------------------------------------------------------------------------------------------------------------
```

C SCRIPTS FOR STIMULI GENERATION, SYNTHESIS AND POWER ANALYSIS

In this chapter scripts for generating programs, input stimuli, synthesize configurations and power analysis configurations can be found. The scripts are given for the n-multiplier designs and are representative for all designs discussed in this work.

C.1 Matlab Stimuli Generation and Program Compiler

C.1.1 n-Multiplier Design

```
addpath(genpath('programs_DSP13'));
%Program compiler DSP13 with larger inputs.

%% Create fixed point object for correct handling
clear all;
F = fimath('RoundingMethod', 'Floor', ...
    'OverflowAction', 'Saturate', ...
    'ProductMode', 'FullPrecision', ...
    'SumMode', 'FullPrecision');

%% Set parameters
figIdx = 1;
enablePlots = false;
enableFileout = true;

%Set maximum memory size
mem_size = 256;
%Set number of taps for the FIR filter
num_taps = 100;
%Set number of samples that will be fed to the design
num_inputs = 300;
%When True, create random values with which to initialise the X memories
fill_xmem_rand = true;
%Set bit-width of samples
sample_size = 32;
%Set how many bits of the samples are maximally filled.
filled_bits_sin = 24;
%Set bit-width of samples
coeff_size = 32;
%Set how many bits of the coefficients are maximally filled.
filled_bits_coeff = 24;

%% Generate coefficients and inputs
%To prevent the maximum of 32 bit registers of being reached in the adder
%tree, do not set range to full possible range (which would be 1.0)
range_coeff = 0.0625;

%Uncomment next two lines to generate new coefficients
    %coeff = -range_coeff + (range_coeff+range_coeff)*rand(num_taps,1);
    %coeff_file = coeff;
%load coefficients from file if no new coefficients are generated
load('coeff_file_23_2_22.mat');
coeff = coeff_file;

% Generate input
```

```
range_sin = 0.0625;
%Uncomment next two lines to generate new samples
    %sin = -range_sin + (range_sin+range_sin)*rand(num_inputs,1);
    %sin_file = sin;
%load samples from file if no new samples are generated
load('sin_file_23_2_22.mat');
sin = sin_file;

%Uncomment next two lines to generate new initial values for the X memories
    %sin_xmem_init_base = -range_sin + (range_sin+range_sin)*rand(mem_size,1);
    %sin_xmem_init_base_file = sin_xmem_init_base;
%load initial X memory values from file if no new initial samples are generated
load('sin_xmem_init_base_file_23_2_22.mat');
sin_xmem_init_base = sin_xmem_init_base_file;

%Set range of designs to generate programs for
for num_mem = 2 : 110

%When mem_size/num_mem is not a round number the amount of generated memory
%is rounded upwards, this extra memory is filled with zeros.
sin_xmem_init = [sin_xmem_init_base' zeros(1,(ceil(mem_size/num_mem))*num_mem-mem_size)];

%% Convert to fixed point
sin = fi(sin,1,32,31,F);
sin_xmem_init = fi(sin_xmem_init,1,32,31,F);
coeff = fi(coeff,1,32,31,F);

%Create program index
prog_cnt = 1;

%% Adjusting constants and splitting up coefficients for multiple memory implementation
mem_pnt_max = ceil(num_taps/num_mem);
coeff_temp = [coeff; fi(zeros(num_mem-mod(num_taps,num_mem),1),1,32,31,F)];
coeff_split = reshape(coeff_temp,num_mem,[]);

sin_xmem_init_temp = sin_xmem_init';
sin_xmem_init_split = reshape(sin_xmem_init_temp,num_mem,[]);

%% Writing the program
%write new sample (preamble)
    Xbase_inc(prog_cnt) = "False";
    Xwr_en(prog_cnt) = "True";
    %Prog_jump(prog_cnt) = prog_jump;
    Mem_pnt(prog_cnt) = (0);
    Outp(prog_cnt) = "False";
    Acc_en(prog_cnt) = "False";
prog_cnt = prog_cnt + 1;
%Load Xreg and Yreg, do not accumulate
    Xbase_inc(prog_cnt) = "False";
    Xwr_en(prog_cnt) = "False";
    %Prog_jump(prog_cnt) = prog_jump;
    Mem_pnt(prog_cnt) = (0);
    Outp(prog_cnt) = "False";
    Acc_en(prog_cnt) = "False";
prog_cnt = prog_cnt + 1;
preamble_length = prog_cnt;

%main loop
for i = prog_cnt : (preamble_length+mem_pnt_max-2)
    Xbase_inc(prog_cnt) = "False";
    Xwr_en(prog_cnt) = "False";
    Mem_pnt(prog_cnt) = (i-preamble_length + 1);
    Outp(prog_cnt) = "False";
    Acc_en(prog_cnt) = "True";
prog_cnt = prog_cnt + 1;
end

%last accumulation
    Xbase_inc(prog_cnt) = "False";
    Xwr_en(prog_cnt) = "False";
    Mem_pnt(prog_cnt) = (0);
    Outp(prog_cnt) = "False";
    Acc_en(prog_cnt) = "True";
```

```
prog_cnt = prog_cnt + 1;
%Writing to output register and increasing base pointer
    Xbase_inc(prog_cnt) = "True";
    Xwr_en(prog_cnt) = "False";
    Mem_pnt(prog_cnt) = (0);
    Outp(prog_cnt) = "True";
    Acc_en(prog_cnt) = "False";
prog_cnt = prog_cnt + 1;

% Pad rest of program memory with "idle" instruction
for i = prog_cnt : mem_size
    Xbase_inc(i) = "False";
    Xwr_en(i) = "False";
    %Prog_jump(i) = prog_jump;
    Mem_pnt(i) = 0;
    Outp(i) = "False";
    Acc_en(i) = "False";
end

%Set End of Program bit, that instruction will be repeated in the design
Prog_jump = repelem("False",mem_size);
Prog_jump(prog_cnt) = "True";
program_length(num_mem) = prog_cnt;

%For debugging, add all instruction sections in one matrix.
program_matrix = [Xbase_inc' Xwr_en' Prog_jump' Mem_pnt' Outp' Acc_en'];

%% Write program, input vector, initial X data and coefficients to file
if enableFileout
fileID = fopen(sprintf('programs_DSP13/DSP13_program1_num_mem_%d.hs',num_mem),'w'); % / for
    linux, programs_DSP13\\DSP13_program1_num_mem_%d.hs for windows
fprintf(fileID,'--FIR program for DSP13 with %d X and Y memories\n',num_mem);
fprintf(fileID,'module DSP13_program1_num_mem_%d where\n',num_mem);
fprintf(fileID,'import Clash.Prelude\n');
fprintf(fileID,'import Data_types_DSP13_num_mem_%d\n',num_mem);
fprintf(fileID,'import DSP_barrel_shifters\n');

fprintf(fileID,'program1 = (((%s :: Xbase_inc),(%s :: Xwr_en),(%s :: Prog_jump),(%d :: Mem_pnt
    ),(%s :: Outp_instr),(%s :: Acc_en)) :>',Xbase_inc(1),Xwr_en(1),Prog_jump(1),Mem_pnt(1),
    Outp(1),Acc_en(1));
for i = 2 : length(Xbase_inc)
fprintf(fileID,'(%s,%s,%s,%d,%s,%s) :>',Xbase_inc(i),Xwr_en(i),Prog_jump(i),Mem_pnt(i),Outp(i)
    ,Acc_en(i));
end
fprintf(fileID,' Nil)\n');

fprintf(fileID,'in_vec = ');
fprintf(fileID,'(replicate d%d (0 :: Sin)) ++ ',mem_size);
fprintf(fileID,'(replicate d256 (%.31f :: Sin)) ++ ',sin(1));
fprintf(fileID,'(replicate d256 (%.31f)) ++ ', sin(2:end));
fprintf(fileID,'(Nil)\n');

fprintf(fileID,'coeff_vec = (');
for i = 1 : size(coeff_split,2)
    fprintf(fileID,'(');
    for j = 1 : num_mem
    fprintf(fileID,'Just ((%d :: YMem_addr),(%.31f::Coeff)):>',(i-1),coeff_split(j,i));
    end
fprintf(fileID,'Nil):>');
end
fprintf(fileID,'Nil)\n');

fprintf(fileID,'ext_wr_vec = (');
for i = 1 : size(sin_xmem_init_split,2)
    fprintf(fileID,'(');
    for j = 1 : num_mem
    fprintf(fileID,'Just ((%d :: XMem_addr),(%.31f::Sin)):>',(i-1),sin_xmem_init_split(j,i));
    end
fprintf(fileID,'Nil):>');
end
fprintf(fileID,'Nil)\n');


fprintf(fileID,'frame_trig_vec = ');
fprintf(fileID,'(replicate d256 False) ++ ');
fprintf(fileID,'concat (replicate d%d ((True :> Nil) ++ (replicate d255 False)))\n',
    num_inputs);

fprintf(fileID,'out_vec = ');
fprintf(fileID,'(replicate d256 (0 :: Sout)) ++ ');
fprintf(fileID,'(replicate d256 (%.31f :: Sout)) ++ ',sout_des(1));
fprintf(fileID,'(replicate d256 (%.31f)) ++ ', sout_des(2:end));
fprintf(fileID,'(Nil)\n');

fprintf(fileID,'regs_init = ');
if fill_xmem_rand
    fprintf(fileID,'(');
    reversed_regs = flip(sin_xmem_init);
    fprintf(fileID,'(%.31f :: Sin):>',reversed_regs(1:length(coeff)));
    fprintf(fileID,'Nil)\n');
else
    fprintf(fileID,'(replicate d%d (0 :: Sin))\n',length(coeff));
end
fprintf(fileID,'in_vec_hs = ');
fprintf(fileID,'(%.31f :: Sin):>',sin(1));
fprintf(fileID,'(%.31f):>',sin(2:end));
fprintf(fileID,'(Nil)\n');

fprintf(fileID,'output_offset = d%d\n',(mem_size+prog_cnt));

fprintf(fileID,'coeff_vec_hs = (');
fprintf(fileID,'(%.31f::Coeff):>',coeff);
fprintf(fileID,'Nil)\n');

%% Write needed barrel shifter instantiation
fileID_2 = fopen(sprintf('programs_DSP13/DSP13_barrel_shift_num_mem_%d.hs',num_mem),'w'); % \\
     for windows, / for linux
fprintf(fileID_2,'\n--Barrel shifter instantiation and help function\n');
fprintf(fileID_2,'to_bool_vec :: Shft_cnt -> Vec (CLog 2 Num_mem) Bool\n');
fprintf(fileID_2,'to_bool_vec cnt = unpack (pack cnt) :: Vec (CLog 2 Num_mem) Bool\n\n');

fprintf(fileID_2,'barrel_shift :: Vec Num_mem Xreg -> Shft_cnt -> Vec Num_mem Xreg\n');
fprintf(fileID_2,'barrel_shift vec_in cnt = fst $ (');
for i = ceil(log2(num_mem)) : (-1) : 2
    fprintf(fileID_2,'(vec_shft_%d (shft_en !! %d)) . ',i,i-1);
end
fprintf(fileID_2,'(vec_shft_1 (shft_en !! 0))) (vec_in,d0)\n');
fprintf(fileID_2,'    where\n');
fprintf(fileID_2,'        shft_en = reverse (to_bool_vec cnt)\n');

fprintf(fileID_2,'{-# NOINLINE barrel_shift #-}\n');

end
end
%% Plot number of instructions in program for different designs.
if enablePlots
   plot_prog_size_f = figure(figIdx);
   figIdx + 1;
   plot(2:33,program_length(2:33),'-o')
   title('Number of instructions in 100 tap FIR program');
   xlabel('Number of prallel multipliers and memories');
   ylabel('Number of instructions in program');
   set(plot_prog_size_f,'Units','Inches');
   pos = get(plot_prog_size_f,'Position');
   set(plot_prog_size_f,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3),
    pos(4)])
   print(plot_prog_size_f,'plot_prog_size','-dpdf','-r0')

end
```

C.2 Shell Scripts for Generating all Haskell, Verilog and Synthesis files

C.2.1 Top Level Generate Script

```bash
#!/bin/bash

#Set range of designs to generate
MIN_NUM_MEM=2
MAX_NUM_MEM=33

#copy barrel shifter file and RAM primitive to haskell folder
cp DSP_barrel_shifters.hs ./generated_haskell/
cp Clash_Explicit_RAM.primitives ./generated_haskell/

cd ./synthesize_folder

#Create power_reports directory if it does not yet exist
if (test -d power_reports)
then
 :
else
 mkdir power_reports
fi

cd ..

#Main loop, for every index create all needed Haskell files with unique module and topentity names.
#Additionally, create Synopsys design compiler configuration scripts and Synopsys PrimeTime configuration script
#Lastly, create a shell script which when run will call upon every Synthesis and Power analysis script and remove large files afterwards
for (( NUM_MEM=${MIN_NUM_MEM}; NUM_MEM<=${MAX_NUM_MEM}; NUM_MEM++ ))
do
    echo "module Data_types_DSP13_num_mem_${NUM_MEM} where" > "./generated_haskell/Data_types_DSP13_num_mem_${NUM_MEM}.hs"
    echo "import Clash.Prelude" >> "./generated_haskell/Data_types_DSP13_num_mem_${NUM_MEM}.hs"
    echo "import Clash.Explicit.Testbench" >> "./generated_haskell/Data_types_DSP13_num_mem_${NUM_MEM}.hs"
    echo "import Clash.Promoted.Nat" >> "./generated_haskell/Data_types_DSP13_num_mem_${NUM_MEM}.hs"
    echo "type Num_mem=${NUM_MEM}" >> "./generated_haskell/Data_types_DSP13_num_mem_${NUM_MEM}.hs"
    cat Data_types_DSP13_num_mem_.hs >> "./generated_haskell/Data_types_DSP13_num_mem_${NUM_MEM}.hs"

    echo "module DSP13_num_mem_${NUM_MEM} where" > "./generated_haskell/DSP13_num_mem_${NUM_MEM}.hs"
    echo "import Clash.Prelude" > "./generated_haskell/DSP13_num_mem_${NUM_MEM}.hs"
    echo "import DSP13_program1_num_mem_${NUM_MEM}" >> "./generated_haskell/DSP13_num_mem_${NUM_MEM}.hs"
    echo "import Data_types_DSP13_num_mem_${NUM_MEM}" >> "./generated_haskell/DSP13_num_mem_${NUM_MEM}.hs"
    cat DSP13_num_mem_.hs >> "./generated_haskell/DSP13_num_mem_${NUM_MEM}.hs"
    cat ./generated_haskell/DSP13_barrel_shift_num_mem_${NUM_MEM}.hs >> "./generated_haskell/DSP13_num_mem_${NUM_MEM}.hs"
    cat << eof >> "./generated_haskell/DSP13_num_mem_${NUM_MEM}.hs"
{-# ANN topEntity
    (Synthesize
    { t_name   = "DSP13_num_mem_${NUM_MEM}"
    , t_inputs = [ PortName "clk"
                 , PortName "rst"
                 , PortName "en"
                 , PortName "en_mac"
                 , PortName "inp"
                 , PortName "frame_trig"
                 , PortName "p_in"
                 , PortName "y_in"
                 , PortName "ext_wr"
                  ]
    , t_output = PortName "outp"
    })
  #-}
eof


```

```
cd ./generated_haskell
#Call upon Clash to generate verilog files of the design
stack exec --resolver=lts-18.21 --package clash-ghc -- clash DSP13_num_mem_${NUM_MEM}.hs --
    verilog -fclash-hdldir verilog_num_mem_${NUM_MEM}
cd ..
cd ./synthesize_folder
#Check if Synthesis folder for specific num_mem exists and create it if it does not.
if (test -d Synthesis_DSP13_num_mem_${NUM_MEM})
then
 :
else
 mkdir Synthesis_DSP13_num_mem_${NUM_MEM}
fi
cd ./Synthesis_DSP13_num_mem_${NUM_MEM}
#Create include folder if does not yet exist
if (test -d include)
then
 :
else
 mkdir include
fi
cd ..
cd ..
#Copy generated verilog files to correct folders
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}.v ./
    synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}
    _acc_handle.v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}_pmem
    .v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}_xmem
    .v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}_ymem
    .v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}
    _barrel_shift.v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}_xreg
    .v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}_yreg
    .v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}
    _x_rd_addrs_handle.v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.topEntity/DSP13_num_mem_${NUM_MEM}
    _x_wr_inps_handle.v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp ./generated_haskell/verilog_num_mem_${NUM_MEM}/Main.testBench/testBench.v ./
    synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include
cp uk65lscllmvbbr_sdf30.v ./synthesize_folder/Synthesis_DSP13_num_mem_${NUM_MEM}/include

#Create synthesis and power analysis shell script with correct num_mem index
echo "DESIGN_NAME=\"DSP13_num_mem_${NUM_MEM}\"" > ./synthesize_folder/
    Synthesis_DSP13_num_mem_${NUM_MEM}/generate_primepower_design_DSP13_num_mem_${NUM_MEM}
cat generate_primepower_design_DSP13_num_mem_ >> ./synthesize_folder/Synthesis_DSP13_num_mem_$
    {NUM_MEM}/generate_primepower_design_DSP13_num_mem_${NUM_MEM}

done
#After loop is finished create top level script which calls upon all the generated shell
    scripts which will synthesize the designs and do power analysis, after which the synout
    and modelsim directory are removed to conserve disk space.
cat << eof > "./synthesize_folder/run_all_power_reports"
for (( NUM_MEM=${MIN_NUM_MEM}; NUM_MEM<=${MAX_NUM_MEM}; NUM_MEM++ ))
do
cd Synthesis_DSP13_num_mem_\${NUM_MEM}
. generate_primepower_design_DSP13_num_mem_\${NUM_MEM}
rm -r ./SynOut
rm -r ./modelsim
cd ..
done

eof
```

C.2.2 Script Controlling Synopsys Design Compiler, Synopsys Primetime and Modelsim

```
# check for output directory and create it if necessary
if (test -d SynOut)
then
  :
else
  mkdir SynOut
fi

if (test -d Reports)
then
  :
else
  mkdir Reports
fi

if (test -d modelsim)
then
  :
else
  mkdir modelsim
fi

#create synopsys .dc file
cat << eof > .synopsys_dc.setup
set_app_var target_library "uk65lscllmvbbr_120c25_tc_ccs.db"

set_app_var synthetic_library "dw_foundation.sldb"
set_app_var link_library "* \$target_library \$synthetic_library"
set_app_var search_path "/local/opt/Technology/UMC/UMC_65nm/_G-01-LOGIC_MIXED_MODE65N-
    LL_LOW_K_UMC-IP/G-9LT-LOGIC_MIXED_MODE65N-LL_LOW_K_UMK65LSCLLMVBBR-LIBRARY_TAPE_OUT_KIT-
    Ver.B03_P.B/UMK65LSCLLMVBBR_B03_TAPEOUTKIT/synopsys/ccs/"

#paths to the source files (.vhdl/.v)
lappend search_path /home/s1568426/synthesize_folder/Synthesis_${DESIGN_NAME}/include
lappend search_path /home/s1568426/synthesize_folder/Synthesis_${DESIGN_NAME}/SynOut
lappend search_path /home/s1568426/synthesize_folder/Synthesis_${DESIGN_NAME}
set_app_var designer "Raben"
eof

source /opt/Synopsys/local/setenv_syn_O-2018.06.sh
### Synthesize the design using Synopsys Design Compiler
dc_shell-t << eof > SynOut/log_syn_$DESIGN_NAME
    remove_design -all
    set DESIGN_NAME $DESIGN_NAME
    set CLK "clk"
    # set the clock in ns (as per the library sepcifications of UMC65nm)
    set clk_pr 20

    analyze -format verilog include/${DESIGN_NAME}.v
    analyze -format verilog include/${DESIGN_NAME}_acc_handle.v
    analyze -format verilog include/${DESIGN_NAME}_pmem.v
    analyze -format verilog include/${DESIGN_NAME}_xmem.v
    analyze -format verilog include/${DESIGN_NAME}_ymem.v
    analyze -format verilog include/${DESIGN_NAME}_barrel_shift.v
    analyze -format verilog include/${DESIGN_NAME}_xreg.v
    analyze -format verilog include/${DESIGN_NAME}_yreg.v
    analyze -format verilog include/${DESIGN_NAME}_x_rd_addrs_handle.v
    analyze -format verilog include/${DESIGN_NAME}_x_wr_inps_handle.v


    elaborate ${DESIGN_NAME}

    if { ! [link] } {
    puts "Error: Failed to link 'Design'."
    exit 1
    }

    check_design
    create_clock \${CLK} -period \${clk_pr}
    # Working out a proper clock tree was not possible in the available time-frame, instead
    the clock is set to ideal (no propagation delay).
    set_ideal_network -no_propagate {\${CLK}}

    uniquify -force

    compile_ultra -no_autoungroup

    change_names -rules verilog -hierarchy

    # Write output files
    write -format ddc -hierarchy -output SynOut/${DESIGN_NAME}.mapped.ddc
    write -f verilog -hierarchy -output SynOut/${DESIGN_NAME}.mapped.v
    write_sdf -version 3.0 SynOut/${DESIGN_NAME}.mapped.sdf
    write_sdc -nosplit SynOut/${DESIGN_NAME}.mapped.sdc
    write_parasitics -output SynOut/${DESIGN_NAME}.mapped.spef

    # Report Area and Power
    report_area -hierarchy > ../power_reports/${DESIGN_NAME}_area.rpt
    report_power > Reports/${DESIGN_NAME}_power0.rpt
    report_power -hierarchy > Reports/${DESIGN_NAME}_power1.rpt

    # Report constraint violations
    report_constraints -all_viol > Reports/all_violations.rpt

    puts "Synthesis Done and Successfull for $DESIGN_NAME !"
    exit
eof

cd modelsim
vmap -c
cd ..
### Use command line modelsim to stimulate the design and record switching behaviour
eval vsim -c << eof > modelsim_log
project new ./modelsim ${DESIGN_NAME}
project addfile ../SynOut/${DESIGN_NAME}.mapped.v
project addfile ../include/testBench.v
project addfile ../include/uk65lscllmvbbr_sdf30.v
project compileall

vsim +notimingchecks -noglitch work.testBench -novopt -sdftyp {/testBench/${DESIGN_NAME}_c\
    $MaintestBench_app_arg=/home/s1568426/synthesize_folder/Synthesis_${DESIGN_NAME}/SynOut/${
    DESIGN_NAME}.mapped.sdf}

run 5120 ns
vcd file ${DESIGN_NAME}.vcd
vcd add /testBench/${DESIGN_NAME}_c\\\$MaintestBench_app_arg/*
run -all
eof

module load synopsys/prime/S-2021.06-SP2
cat << eof > primepower_script_${DESIGN_NAME}.tcl

    set power_enable_analysis TRUE
    set DESIGN_NAME $DESIGN_NAME
    if {\$power_enable_timing_analysis == false} {set_app_var power_enable_timing_analysis
    true}
    set power_analysis_mode averaged

    #####################################################################
    #       link design
    #####################################################################
    set target_library "uk65lscllmvbbr_120c25_tc_ccs.db"
    set synthetic_library "dw_foundation.sldb"
    set search_path         "/local/opt/Technology/UMC/UMC_65nm/_G-01-LOGIC_MIXED_MODE65N-
    LL_LOW_K_UMC-IP/G-9LT-LOGIC_MIXED_MODE65N-LL_LOW_K_UMK65LSCLLMVBBR-LIBRARY_TAPE_OUT_KIT-
    Ver.B03_P.B/UMK65LSCLLMVBBR_B03_TAPEOUTKIT/synopsys/ccs/"
    set link_library  "* \$target_library \$synthetic_library"

    read_verilog    ./SynOut/${DESIGN_NAME}.mapped.v
    current_design    $DESIGN_NAME
    link


    #####################################################################
    #       set transition time / annotate parasitics
    #####################################################################
    read_sdc ./SynOut/${DESIGN_NAME}.mapped.sdc
    #set_disable_timing [get_lib_pins ssc_core_typ/*/G]
    read_parasitics     ./SynOut/${DESIGN_NAME}.mapped.spef

    ####################################################################
    #       check/update/report timing
    ####################################################################
    check_timing
    update_timing
    report_timing

    ####################################################################
    #       read switching activity file
    ####################################################################
    read_vcd -strip_path testBench/${DESIGN_NAME}_c\\\$MaintestBench_app_arg "/home/s1568426/
    synthesize_folder/Synthesis_${DESIGN_NAME}/modelsim/${DESIGN_NAME}.vcd"

    ####################################################################
    #       check/update/report power
    ####################################################################
    check_power -verbose
    update_power
    report_power -verbose

    report_power -verbose > ../power_reports/${DESIGN_NAME}_avg_primepower0.rpt
    report_power -hierarchy -levels 1 -verbose > ../power_reports/${DESIGN_NAME}
    _avg_primepower1.rpt


    quit
eof

pt_shell -f primepower_script_${DESIGN_NAME}.tcl
```

D TABLES CONTAINING AVERAGE POWER AND TOTAL CELL AREA RESULTS

D.1 Designs without Clock Gating

# multipliers	1	2	3	4	5	6	7	8	9	10	11	12
ymem	0.0029500	0.0029200	0.0029400	0.0029120	0.0029600	0.0029390	0.0029470	0.0029120	0.0029700	0.0029600	0.0030030	0.0030000
xmem	0.0029500	0.0029400	0.0029550	0.0029230	0.0029650	0.0029460	0.0029580	0.0029280	0.0029790	0.0029700	0.0030140	0.0030120
pmem	0.0018300	0.0017300	0.0017300	0.0016400	0.0016400	0.0016400	0.0016400	0.0015500	0.0015500	0.0015500	0.0015500	0.0015500
MAC	0.0002950	0.0004070	0.0004900	0.0005210	0.0005840	0.0006340	0.0006920	0.0006970	0.0007000	0.0006810	0.0007500	0.0007810
barrel shift	0.0000000	0.0000042	0.0000034	0.0000031	0.0000038	0.0000036	0.0000038	0.0000050	0.0000066	0.0000071	0.0000076	0.0000092
yregs	0.0000000	0.0000309	0.0000441	0.0000555	0.0000669	0.0000786	0.0000902	0.0001013	0.0001155	0.0001275	0.0001381	0.0001490
xregs	0.0000000	0.0000336	0.0000449	0.0000553	0.0000668	0.0000783	0.0000862	0.0001019	0.0001104	0.0001216	0.0001342	0.0001452
remainder	0.0000780	0.0000333	0.0000416	0.0000381	0.0000395	0.0000365	0.0000368	0.0000348	0.0000365	0.0000298	0.0000361	0.0000397

# multipliers	13	14	15	16	17	18	19	20	21	22	23	24
ymem	0.0029640	0.0030240	0.0030750	0.0029120	0.0030940	0.0030780	0.0030210	0.0029600	0.0031080	0.0029920	0.0031280	0.0030000
xmem	0.0029640	0.0030380	0.0030750	0.0029120	0.0030940	0.0030780	0.0030330	0.0029600	0.0031080	0.0030140	0.0031510	0.0030000
pmem	0.0015500	0.0015500	0.0015500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500
MAC	0.0007750	0.0008130	0.0007880	0.0008420	0.0008270	0.0008660	0.0009260	0.0008470	0.0008540	0.0008970	0.0009130	0.0009380
barrel shift	0.0000083	0.0000091	0.0000062	0.0000081	0.0000113	0.0000099	0.0000133	0.0000127	0.0000116	0.0000121	0.0000102	0.0000137
yregs	0.0001615	0.0001735	0.0001835	0.0001964	0.0002065	0.0002180	0.0002301	0.0002392	0.0002513	0.0002630	0.0002753	0.0002871
xregs	0.0001554	0.0001684	0.0001768	0.0001891	0.0002012	0.0002131	0.0002255	0.0002373	0.0002475	0.0002589	0.0002696	0.0002823
remainder	0.0000328	0.0000370	0.0000305	0.0000524	0.0000480	0.0000370	0.0000491	0.0000438	0.0000436	0.0000490	0.0000479	0.0000599

# multipliers	25	26	27	28	29	30	31	32	33	Fully Parallel
ymem	0.0031250	0.0029640	0.0030780	0.0031920	0.0029580	0.0030600	0.0031620	0.0029120	0.0030030	0.0030900
xmem	0.0031250	0.0029640	0.0030780	0.0031920	0.0029860	0.0030890	0.0031900	0.0029183	0.0030098	0.0030900
pmem	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0013600	0.0013600	0.0000000
MAC	0.0009080	0.0009390	0.0009490	0.0009670	0.0010400	0.0010300	0.0010900	0.0011300	0.0014100	0.0014700
barrel shift	0.0000085	0.0000115	0.0000155	0.0000125	0.0000148	0.0000139	0.0000123	0.0000124	0.0000129	0.0000000
yregs	0.0002968	0.0003083	0.0003194	0.0003323	0.0003435	0.0003551	0.0003683	0.0003788	0.0003830	0.0000000
xregs	0.0002925	0.0003047	0.0003164	0.0003277	0.0003395	0.0003510	0.0003627	0.0003744	0.0004154	0.0000000
remainder	0.0000562	0.0000385	0.0000377	0.0000385	0.0000482	0.0000510	0.0000467	0.0000471	0.0000509	0.0000370

Table D.1: Average power consumption (W) values corresponding to the designs without clock gating presented in Figure 6.15.

# multipliers	1	2	3	4	5	6	7	8	9	10
ymem	95574.24220	95063.76220	95982.48210	95258.88200	96661.80200	96071.40240	96687.36210	95382.36240	97433.28180	97041.60200
xmem	95641.56220	95063.76220	95981.76210	95258.88200	96661.80200	96070.32240	96687.36210	95376.96240	97433.28180	97041.60200
pmem	60273.36140	56640.24130	56589.12130	53661.24120	53661.24120	53665.56120	53625.60120	50748.48110	50753.52110	50753.16110
MAC	6795.00010	13527.72010	20297.88020	26973.00030	33787.80030	40481.64040	47363.76050	54006.48050	63437.76010	69960.24010
barrel shift	0.00000	221.40000	424.80000	846.00000	1304.28000	1896.84000	2629.08000	3286.44000	3794.40000	4210.20000
yregs	0.00000	693.36000	1040.04000	1386.72000	1733.40000	2080.08000	2507.76000	2773.44000	3223.80000	3469.32000
xregs	0.00000	693.36000	1040.04000	1386.72000	1734.12000	2080.08000	2508.84000	2775.60000	3223.80000	3471.12000
remainder	1952.63997	1620.36003	1874.16011	1875.24024	2087.64049	2234.87960	2363.40010	2351.87964	2552.04065	2694.60070

# multipliers	11	12	13	14	15	16	17	18	19	20
ymem	98552.52220	98651.52240	97119.36260	99600.48280	101190.60150	96048.00160	102258.00170	101250.00180	99703.44190	97777.08200
xmem	98552.52220	98651.52240	97119.36260	99600.48280	101190.60150	96048.00160	102234.60170	101228.04180	99642.96190	97733.88200
pmem	50755.32110	50750.64110	50755.68110	50708.16110	50761.44110	48338.28110	48366.00110	48366.72110	48326.40110	48329.28110
MAC	76791.60020	81847.80050	90524.88030	97194.60030	103988.88040	110904.12040	125807.03990	132395.39990	139032.72000	145805.76010
barrel shift	4628.16000	4177.80000	5470.20000	5891.40000	6307.56000	6733.08000	8917.92010	9446.40010	9949.68010	10578.24010
yregs	3816.00000	4161.24000	4658.76000	4868.64000	5374.44000	5564.16000	5893.56000	6240.24000	6586.92000	6933.60000
xregs	3823.56000	4170.60000	4670.64000	4856.40000	5374.44000	5552.64000	5893.56000	6240.24000	6586.92000	6933.60000
remainder	2790.00016	2933.63982	3035.87917	3205.79925	3303.00132	3246.48161	3443.76179	3608.64153	3714.12107	3848.04085

# multipliers	21	22	23	24	25	26	27	28	29	30
ymem	102660.48210	99290.52220	103799.16230	99177.12240	103336.20250	98205.12260	101982.24270	105759.36280	98480.52290	101876.40300
xmem	102619.44210	99221.76220	103731.84230	99169.92240	103302.00250	98130.60260	101894.76270	105668.64280	98417.88290	101811.60300
pmem	48323.52110	48328.92110	48332.16110	48332.16110	48324.96110	48332.88110	48325.32110	48326.04110	48331.08110	48326.76110
MAC	152454.60010	159177.96010	165938.04010	172211.04020	185129.27980	191855.15980	198518.75990	204593.75990	214898.75970	220795.55970
barrel shift	11006.64010	11517.48010	12059.28010	12550.68010	13097.52010	13607.28010	14149.44010	14673.60010	15176.16010	15690.60010
yregs	7280.28000	7626.96000	7973.64000	8321.40000	8667.00000	9016.56000	9360.36000	9707.04000	10053.72000	10400.40000
xregs	7281.72000	7627.68000	7973.64000	8323.56000	8667.00000	9013.68000	9360.36000	9707.04000	10053.72000	10400.40000
remainder	3979.80087	4066.20067	4188.24071	4337.28033	4462.56038	4577.03991	4701.95986	4810.67986	5010.47951	5114.87949

# multipliers	31	32	33	Fully Parallel
ymem	105262.92310	95649.48320	98734.68330	91481.76170
xmem	105205.32310	95604.48320	98715.24330	88758.36160
pmem	48336.48110	45317.16100	45365.04100	0.00000
MAC	228889.07970	236843.27960	264176.27840	1738373.77500
barrel shift	16183.80010	16687.80010	20843.28010	0.00000
yregs	10747.08000	11093.76000	11440.44000	0.00000
xregs	10747.08000	11093.76000	11440.44000	0.00000
remainder	5241.95941	5053.31839	5228.27838	45915.84182

Table D.2: Total cell area values corresponding to the designs without clock gating presented in Figure 6.14.

D.2 Designs with Clock Gating

# multipliers	1	2	3	4	5	6	7	8	9	10	11	12
ymem	0.0029500	0.0029200	0.0029400	0.0029120	0.0029600	0.0029390	0.0029470	0.0029120	0.0029700	0.0029600	0.0030030	0.0030000
xmem	0.0029500	0.0029400	0.0029550	0.0029230	0.0029650	0.0029460	0.0029580	0.0029280	0.0029790	0.0029700	0.0030140	0.0030120
pmem	0.0018300	0.0017300	0.0017300	0.0016400	0.0016400	0.0016400	0.0016400	0.0015500	0.0015500	0.0015500	0.0015500	0.0015500
MAC	0.0002950	0.0004070	0.0004900	0.0005210	0.0005840	0.0006340	0.0006920	0.0006970	0.0007000	0.0006810	0.0007500	0.0007810
barrel shift	0.0000000	0.0000042	0.0000034	0.0000031	0.0000038	0.0000036	0.0000038	0.0000050	0.0000066	0.0000071	0.0000076	0.0000092
yregs	0.0000000	0.0000309	0.0000441	0.0000555	0.0000669	0.0000786	0.0000902	0.0001013	0.0001155	0.0001275	0.0001381	0.0001490
xregs	0.0000000	0.0000336	0.0000449	0.0000553	0.0000668	0.0000783	0.0000862	0.0001019	0.0001104	0.0001216	0.0001342	0.0001452
remainder	0.0000780	0.0000333	0.0000416	0.0000381	0.0000395	0.0000365	0.0000368	0.0000348	0.0000365	0.0000298	0.0000361	0.0000397

# multipliers	13	14	15	16	17	18	19	20	21	22	23	24
ymem	0.0029640	0.0030240	0.0030750	0.0029120	0.0030940	0.0030780	0.0030210	0.0029600	0.0031080	0.0029920	0.0031280	0.0030000
xmem	0.0029640	0.0030380	0.0030750	0.0029120	0.0030940	0.0030780	0.0030330	0.0029600	0.0031080	0.0030140	0.0031510	0.0030000
pmem	0.0015500	0.0015500	0.0015500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500
MAC	0.0007750	0.0008130	0.0007880	0.0008420	0.0008270	0.0008660	0.0009260	0.0008470	0.0008540	0.0008970	0.0009130	0.0009380
barrel shift	0.0000083	0.0000091	0.0000062	0.0000081	0.0000113	0.0000099	0.0000133	0.0000127	0.0000116	0.0000121	0.0000102	0.0000137
yregs	0.0001615	0.0001735	0.0001835	0.0001964	0.0002065	0.0002180	0.0002301	0.0002392	0.0002513	0.0002630	0.0002753	0.0002871
xregs	0.0001554	0.0001684	0.0001768	0.0001891	0.0002012	0.0002131	0.0002255	0.0002373	0.0002475	0.0002589	0.0002696	0.0002823
remainder	0.0000328	0.0000370	0.0000305	0.0000524	0.0000480	0.0000370	0.0000491	0.0000438	0.0000436	0.0000490	0.0000479	0.0000599

# multipliers	25	26	27	28	29	30	31	32	33	Fully Parallel
ymem	0.0031250	0.0029640	0.0030780	0.0031920	0.0029580	0.0030600	0.0031620	0.0029120	0.0030030	0.0030900
xmem	0.0031250	0.0029640	0.0030780	0.0031920	0.0029860	0.0030890	0.0031900	0.0029183	0.0030098	0.0030900
pmem	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0014500	0.0013600	0.0013600	0.0000000
MAC	0.0009080	0.0009390	0.0009490	0.0009670	0.0010400	0.0010300	0.0010900	0.0011300	0.0014100	0.0014700
barrel shift	0.0000085	0.0000115	0.0000155	0.0000125	0.0000148	0.0000139	0.0000123	0.0000124	0.0000129	0.0000000
yregs	0.0002968	0.0003083	0.0003194	0.0003323	0.0003435	0.0003551	0.0003683	0.0003788	0.0003830	0.0000000
xregs	0.0002925	0.0003047	0.0003164	0.0003277	0.0003395	0.0003510	0.0003627	0.0003744	0.0004154	0.0000000
remainder	0.0000562	0.0000385	0.0000377	0.0000385	0.0000482	0.0000510	0.0000467	0.0000471	0.0000509	0.0000370

Table D.3: Average power consumption (W) values corresponding to the clock gated designs presented in Figure 6.17.