University of Arkansas, Fayetteville
ScholarWorks@UARK

Graduate Theses and Dissertations

5-2023

Stream Processor Development using Multi-Threshold NULL Convention Logic Asynchronous Design Methodology

Wassim Khalil
University of Arkansas, Fayetteville

Follow this and additional works at: https://scholarworks.uark.edu/etd

 Part of the Electrical and Electronics Commons, Power and Energy Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

Citation

Khalil, W. (2023). Stream Processor Development using Multi-Threshold NULL Convention Logic Asynchronous Design Methodology. *Graduate Theses and Dissertations* Retrieved from https://scholarworks.uark.edu/etd/5039

This Dissertation is brought to you for free and open access by ScholarWorks@UARK. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu, uarepos@uark.edu.

Stream Processor Development using Multi-Threshold NULL Convention Logic Asynchronous Design Methodology

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Engineering

by

Wassim Khalil
University of Arkansas
Bachelor of Science in Computer Engineering, 2017

May 2023
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

_____
Jia Di, Ph.D.
Dissertation Director

_____
James P. Parkerson, Ph.D.
Committee Member

_____
Dale Thompson, Ph.D.
Committee Member

_____
Zhong Chen, Ph.D.
Committee Member

ABSTRACT

Decreasing transistor feature size has led to an increase in the number of transistors in integrated circuits (IC), allowing for the implementation of more complex logic. However, such logic also requires more complex clock tree synthesis (CTS) to avoid timing violations as the clock must reach many more gates over larger areas. Thus, timing analysis requires significantly more computing power and designer involvement than in the past. For these reasons, IC designers have been pushed to nix conventional synchronous (SYNC) architecture and explore novel methodologies such as asynchronous, self-timed architecture.

This dissertation evaluates the nominal active energy, voltage-scaled active energy, and leakage power dissipation across two cores of a stream processor: Smoothing Filter (SF) and Histogram Equalization (HEQ). Both cores were implemented in Multi-Threshold NULL Convention Logic (MTNCL) and clock-gated synchronous methodologies using a gate-level netlist to avoid any architectural discrepancies while guaranteeing impartial comparisons.

MTNCL designs consumed more active energy than their synchronous counterparts due to the dual-rail encoding system; however, high-threshold-voltage (High-V_t) transistors used in MTNCL threshold gates reduced leakage power dissipation by up to 227%. During voltage-scaling simulations, MTNCL circuits showed a high level of robustness as the output results were logically valid across all voltage sweeps without any additional circuitry. SYNC circuits, however, needed extra logic, such as a DVS controller, to adjust the circuit's speed when V_{DD} changed. Although SYNC circuits still consumed less average energy, MTNCL circuit power gains accelerated when switching to lower voltage domains.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Di, for giving me the opportunity to join TruLogic Lab. Many memories were born in Cato Spring Research Center (CSRC). One of the moments I will miss the most is when my cohorts and I call each other to our cubicles to work together on an issue or give an explanation about why simulations have turned out a certain way. Hearing different analysis and solutions helped me develop my own way of tackling problems. Those discussions were great opportunities for me to connect with my cohorts on a personal level while integrating into the American culture. As an immigrant, this was very important to me. Along our journey in TruLogic, everyone discovered their true potential, which I personally did not have the chance to know before joining. This dissertation is the culmination of knowledge and experience acquired from working on various research projects with my cohorts and friends: Cole, Richard, Kelby, Mark, Spencer, Bill, and Pao.

I also would like to thank Samsung for giving me the chance to join the physical implementation (PI) team at Samsung Austin Research and Development Center (SARC) as an intern in 2021 and a full-time engineer in 2022. Thanks Jose and Hongda for your trust and letting me finish my dissertation while working full-time. Thanks to all my mentors for being generous in sharing your experience, especially Dave, Bob, Chi, Khoa, and Dung.

On a personal level, I would like to thank my family for all the support. I would probably have a different career path—a totally different life—if Baba did not buy me my first PC back in the 90s. Thank you Mama for teaching me how to pay attention to little details. Thank you Koukou for having my back and for all our late-night talks. Finally, I would like to thank all my spiritual fathers and my church community at St Nicholas in Springdale, AR; St Mary in Sacramento, CA; and Holy Cross in Austin, TX.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. Introduction

1. Problem Statement

Historically, digital integrated circuit (IC) design has solely focused on the development of synchronous (SYNC) circuits [1]. In the past few years, the reduction in transistor feature size has allowed designers to build complex circuits with smaller area. However, the rampant number of devices requires more sophisticated timing analysis to prevent potential clock skew issues. Analysis should cover various conditions, such as fluctuating supply voltages, to meet timing constraints and deliver correct results. Consequently, additional logic must be developed to handle timing as well as verify the design's output to ensure correct functionality which, in turn, limits the scalability of SYNC designs. As an alternative solution, asynchronous design methodologies, such as NULL Convention Logic (NCL) [2] and Multi-Threshold NULL Convention Logic (MTNCL) [3], are clockless. This allows for more flexible timing requirements, better adaptability to a wider range of application environments, and a smoother implementation of large modular designs with less scalability overhead. Architecture flexibility and robustness make them great candidates to implement stream processors like Graphical Processing Units (GPUs), especially if deployed in a power-limited system. Stream processor architecture focuses on bridging the gap between arithmetic performance and bandwidth by raising the number of arithmetic units [4] and partitioning the storage structures to reduce the bandwidth demands [5]. In contrast, standard general-purpose processors devote a small fraction of their die area to arithmetic units. Additionally, these processors consolidate all storage into main memories that become bottlenecks, limiting the efficacy of parallelism. This architectural difference expands the execution capabilities of a streaming processor to reach a range of hundreds of Giga Operations per Second (GOPS),

suitable for computationally expensive applications like cybersecurity [6] and image processing [7] [8].

2. Dissertation Statement

This research develops the first MTNCL stream processor to demonstrate the flexibility and minimal overhead that asynchronous methodologies offer over their synchronous counterparts when implementing large designs. In addition, maintaining the clock signal in synchronous circuits across multiple cores is a significant challenge, especially under harsh conditions. It is important to note that NCL and MTNCL asynchronous methodologies use a dual-rail encoding scheme to implement the handshaking mechanism, and this scheme does introduce certain drawbacks. Also, threshold gates are used as building blocks which have higher transistor counts than traditional Boolean gates. For this reason, among others, MTNCL methodology was chosen over NCL because MTNCL threshold gates have fewer transistors. This research is a demonstration that MTNCL design methodology can achieve a higher level of architectural flexibility while maintaining its robustness under various conditions due to the omission of the clock-related logic. Along with flexibility and robustness, MTNCL can offer additional benefits such as reducing leakage power. These qualities will drive more designers to consider asynchronous design methodologies, especially when implementing large designs, thereby revolutionizing IC design conventions.

3. Dissertation Organization

This dissertation encompasses eight chapters. **Section II** provides background knowledge of the asynchronous design methodologies (NCL and MTNCL), stream processors, image processing operations, and circuit robustness. **Section III** presents the design methodology, the stream processor's application, and a proposed high-level architecture. **Section IV** discusses the

design implementation then examines the logic verification of the Smoothing Filter (SF) and the Histogram Equalization (HEQ) cores along with a brief comparison between the MTNCL and SYNC versions. **Section V** begins with an overview of setting up simulations for the designs which is followed by a general power analysis. It finishes with an analysis of the effect of voltage-scaling on power delay and power data. **Section VI** concludes the dissertation. A few scenarios, where the use of MTNCL stream processors would be optimal, are featured in **Section VII**. Finally, **Section VIII** maps a detailed plan for future research.

II. Background

1. NULL Convention Logic (NCL)

NCL is a quasi-delay insensitive (QDI) asynchronous design methodology that uses a multi-rail encoding scheme [2]. In an NCL dual-rail scheme, a signal is represented by two wires (Wire 1, Wire 0). As shown in **Table II.1**, there are only three valid states: DATA0, DATA1, and NULL. DATA0 corresponds to Boolean Logic0, while DATA1 corresponds to Boolean Logic1. The NULL state exists when neither of the wires is asserted. In a typical NCL operation, DATA waves, where signals are either DATA0 or DATA1, are separated by a NULL wave, where all signals exhibit NULL. The INVALID state, where both wires of a signal are asserted, should not occur in normal NCL operation.

State	Wire 0	Wire 1
DATA0	1	0
DATA1	0	1
NULL	0	0
INVALID	1	1

Table II.1. Dual-Rail Signal States [2]

NCL is composed of 27 state-holding gates which constitute the set of all Boolean functions with 4 inputs or fewer [9]. These gates are called threshold (TH) gates which follow *THmn* naming convention. The *n* stands for the number of inputs, while *m* denotes the threshold, i.e., the number of inputs that must be asserted for the output to be asserted. Several NCL gates have a name that contains a *w*, such as TH54w32, which signifies that one or more inputs have a higher weight than the rest. In this case, the first input has a weight of 3, while the second has a weight of 2.

As depicted in **Figure 1**, four different transistor networks implement an NCL gate [1]. The *Set* network is responsible for the output assertion or de-assertion depending on the Boolean equation implemented in NMOS transistors. The *Reset* network is needed to de-assert the output when all inputs are low. Finally, the *Hold0* network keeps the output de-asserted if the threshold is not met, whereas the *Hold1* network keeps the output signal asserted if not all inputs are Logic0. This characteristic is called *Hysteresis*. In addition, NCL has the *input-completeness* quality that maintains the quasi-delay-insensitivity at the circuit level. It requires the output of an NCL combinational circuit to alternate from a NULL wave to a DATA wave only once all inputs present either DATA0 or DATA1. Similar to hysteresis, this property requires the output of an NCL combinational circuit to change from DATA to NULL only after all inputs return to NULL. Thus, the hysteresis characteristic implemented in NCL threshold gates helps apply input-completeness at the circuit level.

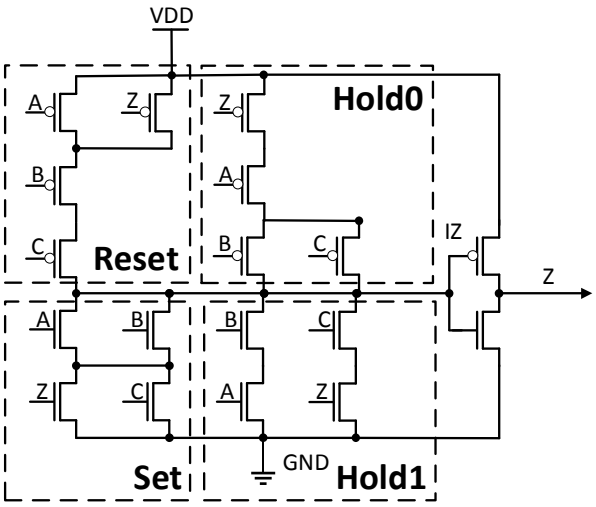

Figure 1. NCL TH23 Schematic [1]

2. Multi-Threshold NULL Convention Logic (MTNCL)

MTNCL is an NCL extension that aims to achieve low-power dual-rail asynchronous circuits. Similarly, MTNCL uses the same states: NULL, DATA0, and DATA1 to represent the dual-rail signals. However, MTNCL differs from NCL in the way NULL cycles are propagated throughout the circuit. MTNCL circuits generate NULL waves instead of providing NULL wave input like NCL circuits. The sleep mechanism implemented at the gate level allows some or all

sections of the circuit to turn idle at given and controlled times. Thus, MTNCL gates incorporate a *sleep* signal input that controls when the gate switches to the idle state. When the *sleep* input asserts, the output of the gate is de-asserted regardless of the input values. In an MTNCL pipeline design, the handshaking signals (*ki* and *ko*) act as the *sleep* control which puts the appropriate blocks into sleep mode at desired times.

Contrary to NCL, MTNCL threshold gates do not need hysteresis due to the sleep mechanism. Therefore, they are significantly smaller than their NCL counterparts due to the omission of the *Reset* and *Hold1* transistor networks as depicted in **Figure 2**. In addition, low-threshold-voltage (low-V_T) and high-threshold-voltage (high-V_T) transistors are used to implement MTNCL threshold gates. The former allows faster switching, while the latter helps reduce the leakage current. Consequently, MTNCL gates are faster, smaller, and consume less power compared to their NCL equivalent gates.

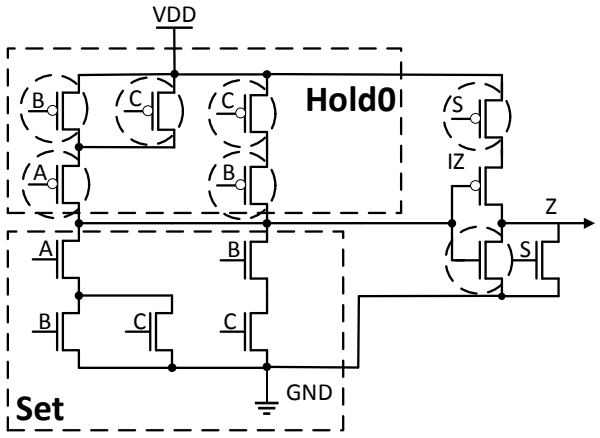

Figure 2. MTNCL TH23 Schematic

As illustrated in **Figure 3**, the *ko* signal allows an MTNCL circuit block to communicate with the previous stage in the pipeline, indicating whether it is ready for inputs or not. Likewise, the *ki* signal allows the proceeding MTNCL stage to indicate its readiness to accept new output. Flowing in the opposite direction, the *sleep* signals, originating from the *ko* output of the previous block, sleep the registers and combinational logic in the next stage in the pipeline. As mentioned previously, this is to emulate the NCL NULL wave and minimize power consumption. To illustrate, if all inputs to a stage are NULL, then *ko* will assert. In this case, there is nothing for the combinational logic to process; therefore, it should be idle. This acknowledgment exchange scheme permits a higher level of flexibility that is difficult to achieve with synchronous methodologies. For example, adding more MTNCL blocks in a sequence should work as long as the MTNCL blocks respect the handshaking and switch correctly between the different states. The omission of a clock also lets the MTNCL circuits be more robust toward operational variability such as a fluctuating supply voltage. On the other hand, synchronous circuit designers need to add

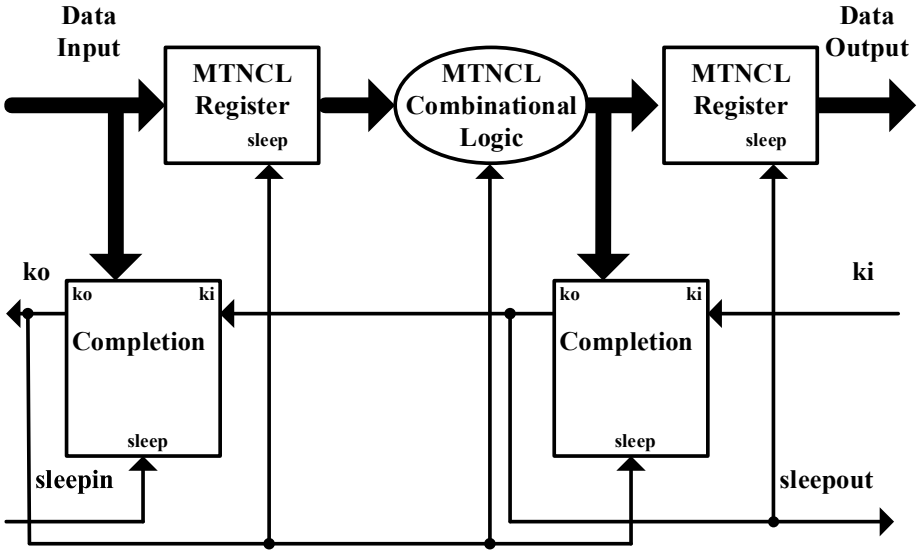

Figure 3. MTNCL Pipelined Architecture

extra logic to support the drive strength and timing of the clock signal when adding new blocks to ensure all blocks are synchronized to latch the correct data in normal conditions. The clock logic development becomes even more challenging to achieve MTNCL's level of robustness if the designer is required to add circuitry to control the frequency of the clock whenever variations in supply voltage or temperature are detected. Moreover, a logic verifier must be attached to the main circuit to ensure the output matches the expected result. Since the supplemental logic controls the clock signal and analyzes the output bits, its size depends on how large the main circuit is. The scalability penalty is a tremendous disadvantage, especially when implementing stream processors. This is due to the high number of cores which range from a few hundred to a few thousand. Therefore, the logic complexity to meet timing requirements in synchronous circuits and the benefit from the architecture flexibility of MTNCL qualify it to implement stream processors with minimal overhead.

3. Stream Processors

Stream processors are composed of two or more processing blocks called *nodes*. As shown in **Figure 4**, the baseline node has a multicore stream processor unit coupled with a s*tream* unit [7]. It is the only persistent storage structure on the node. This includes a 32KB stream instruction storage for application instructions, an 8KB stream commands storage for dispatch and output logic programming, and a 64KB memory subsystem

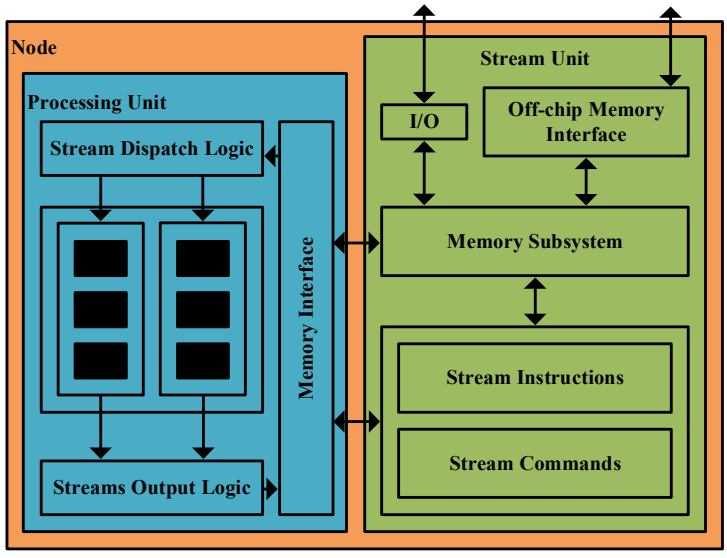

Figure 4. Node Architecture of a Stream Processor

for I/O and off-chip stream buffering. The memory structure in the stream unit should not exceed the sizes denoted above to avoid higher latencies [5]. The main design goal of stream processors is to decentralize as many components as possible to eliminate any bottlenecks that can potentially stall the system.

As illustrated in **Figure 5**, the architecture of a stream processor allows for data, task, and pipeline parallelism. Data parallelism is represented by the bit width of the data processed by the processor (e.g., 8 bits or 16 bits). For task parallelism, stream processors should execute two or more tasks at the same time. The number of parallel tasks depends on the nature of the instruction. Some instructions allow for task parallelism, illustrated as green boxes in **Figure 5**. The input of these instructions can typically be segmented then handed to the cores which can execute the respective function [8]. Therefore, the instructions and the number of cores are the primary factors in determining the level of task parallelism. Lastly, stream processors manage the data exchange

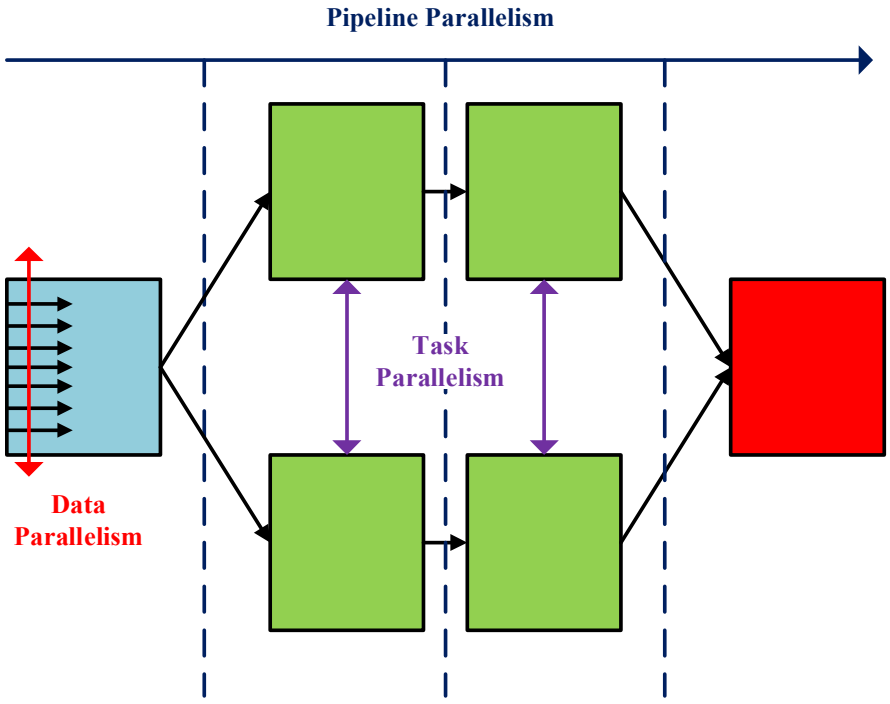

Figure 5. Data, Task, and Pipeline Parallelism in Stream Processing

between the different pipeline stages to enable pipeline parallelism. For example, it must handle the *sync* core, depicted as the red box in **Figure 5**, situation where two or more cores provide the input data. The processor must merge the output of the feeding cores for the sync core to start execution. In a synchronous stream processor, the clock period is set to accommodate the delay of the slowest stage to ensure the correct data is always latched. However, that does not apply to asynchronous circuits. The asynchronous blocks will receive/output the data when they are ready or when the data is available. Regardless of the used architecture, it is necessary to reduce the complexity of every core, especially the larger ones, to avoid slowing down the entire processor.

4. Smoothing Linear Filters

Smoothing filters are used for blurring and noise reduction [10]. Blurring assists in preprocessing tasks, such as the removal of small details from an image before object extraction. Also, it can help bridge small gaps in lines or curves. The output of a smoothing-linear filter is simply the average of the pixels contained in the neighborhood of the filter mask. These filters are called *averaging filters*.

The idea behind smoothing is to replace the value of every pixel in an image with the average of the intensity levels in the neighborhood defined by the filter mask. This process reduces the "sharp" transitions in intensities across all pixels, called *noise*. The primary challenge with noise is its randomness. In addition, there is no one-size-fits-all filter to recover a picture from noise degradation. It is essential to choose suitable coefficients and filter sizes when applying a filter mask to an image. A good smoothing filter should reduce the noise while preserving as many details as possible.

a. Choosing Suitable Coefficients

Figure 6 shows two 3×3 smoothing filters. The first filter is used to compute the standard average of the pixels under the mask. A spatial averaging filter in which all coefficients are equal is called a *box filter* [10]. The second filter denotes a *weighted average* which indicates that pixels are multiplied by different coefficients, thus granting a higher rank (weight) to some pixels at the expense of others. As illustrated in **Figure 6**, the pixel at the center of the mask is multiplied by a higher value "4" than the surrounding pixels. Additionally, these pixels are inversely weighted depending on the distance from the center pixel. This strategy is an attempt to reduce blurring in the smoothing filter. It is important to note that other weights can accomplish the same general effect; however, the power-of-two integers are desirable because they can reduce the overall complexity of the design. In this instance, the sum of all coefficients in the mask of **Figure 6** is equal to 16. In practice, it is difficult to see differences between images smoothed by either mask in **Figure 6** (or similar arrangements) because the area spanned by these masks at any pixel in an image is small.

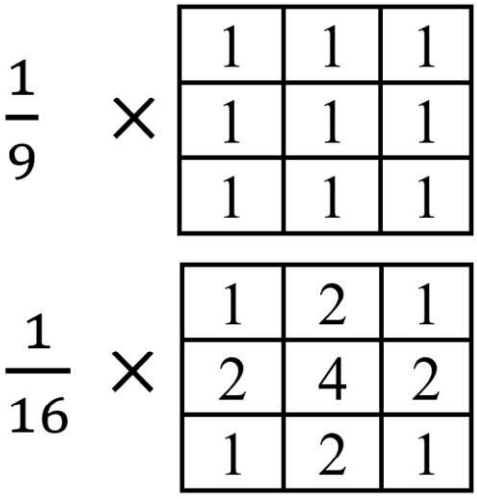

Figure 6. Two 3×3 Smoothing-Filter Masks: box filter (a), and weighted average (b)

b. Choosing a Suitable Filter Size

Figure 7 depicts the effects of smoothing as a function of filter size ($m \times m$) [10]. For the 3×3 filter, a slight blurring is noticeable in the entire image, but details of the same size as the

mask are considerably affected. For example, the black squares, the small letter *a*, and the little grain noise show significant blurring in the 3×3 and 5×5 smoothed pictures compared to the rest of the objects. In the 9×9, there is significantly more blurring. Also, the black circles are no longer distinct from the background like the previous three images. Finally, the results for 15×15 and 35×35 filters are extreme compared to the sizes of the objects in the image. This type of aggressive blurring helps blend small details into the background. That includes the three small squares, two circles, and almost all the noisy rectangles in **Figure 7(f)**. Additionally, the pronounced black border is a consequence of padding the original image with zero (black) pixels and then trimming off the padded area after filtering. The black outline blended into all filtered images; however, it became noticeable for the images smoothed with the larger filters.

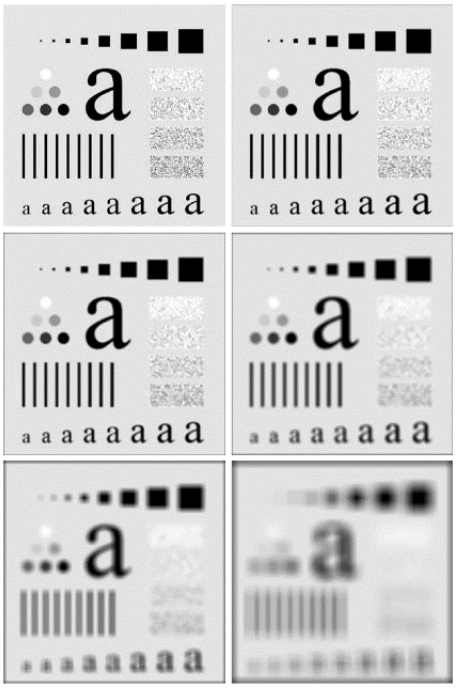

Figure 7. Original image, of size 500×500. (b)-(f) Results of smoothing with square averaging filter masks of size m = 3, 5, 9, 15, and 35, respectively *[10]*

5. Histogram Equalization

A histogram is a graphical representation of the tonal distribution in a digital image which is the basis for numerous spatial domain processing techniques achieved through histogram manipulation. Thus, the image histogram is a powerful tool for image enhancement and provides useful image statistics used in other image processing applications, such as image compression and segmentation [10].

A histogram of a digital image with intensity levels [0, L-1] is a discrete function $h(r_k) = n_k$, where r_k is the k^{th} intensity value and n_k is the number of pixels in the image with intensity r_k. As illustrated in **Figure 8** [10], the horizontal axis of each histogram plot corresponds to intensity r_k. The vertical axis corresponds to values of $h(r_k) = n_k$ or $p(r_k) = \frac{n_k}{Total\ number\ of\ pixels}$. In the dark image, **Figure 8(a)**, the pixel distribution settles in the low (dark) side of the intensity scale. On the other hand, the components of the light image, **Figure 8(b)**, are shifted towards the high (light) side of the intensity scale. The picture with low contrast, **Figure 8(c)**, has a washed-out gray look due to the narrow histogram distribution towards the middle of the scale. Finally, a high-contrast image, **Figure 8(d)**, covers a broader range of shades. Thus, a uniform pixel distribution indicates more detail and has a high dynamic range. A transformation (mapping) function can achieve this goal using only the histogram of the input image.

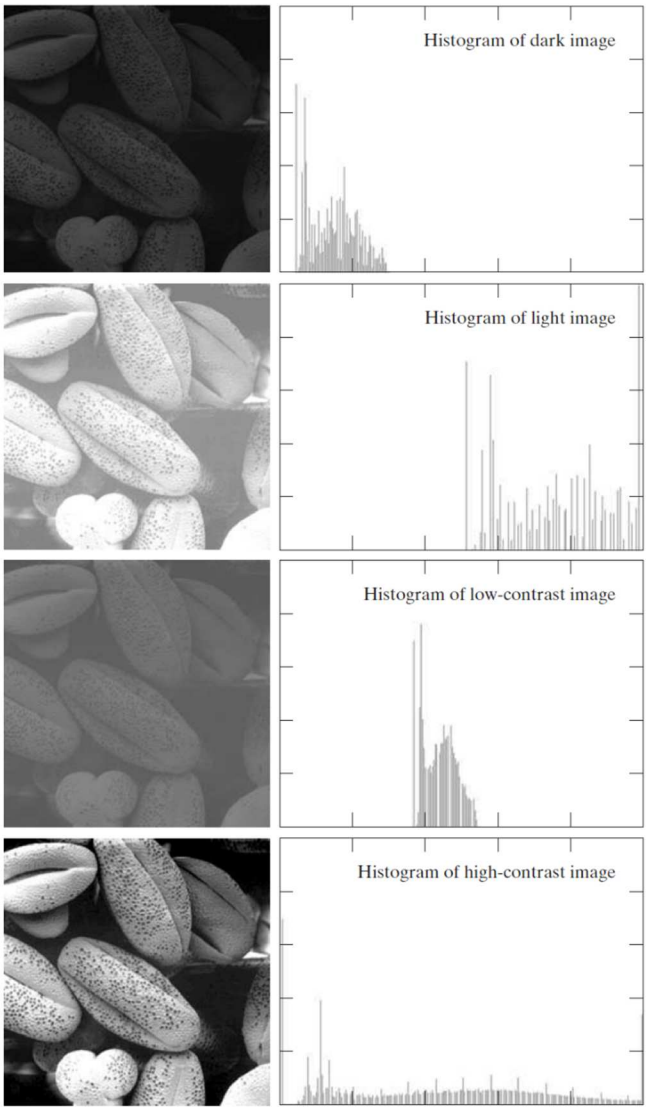

Figure 8. Four basic image types: dark (a), light (b), low contrast (c), high contrast (d), and their corresponding histograms [10]

As mentioned earlier, a histogram represents the probability of occurrence of intensity level r_k in a digital image approximated by $p(r_k) = \frac{n_k}{Total\ number\ of\ pixels}$ where k = 0, 1, 2, …, L-1. L

denotes the number of possible intensity levels in an image with 0 representing black and L-1 representing white (e.g., 256 for an 8-bit image). Thus, an equalized image is obtained by mapping each pixel in the input image with intensity r_k into a corresponding pixel with level s_k in the output image using the following equation:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} \mathrm{p_r}(r_j) = \frac{L-1}{Total\ number\ of\ pixels}\sum_{j=0}^{k} \mathrm{n_j}$$

Equation 1. Transformation Equation of Histogram Equalization

The transformation $T(r_k)$ in this equation is called a *histogram equalization* or *histogram linearization* transformation.

6. Process, Voltage, Temperature (PVT) Corners

In this dissertation, circuits' robustness is achieved when the circuit operates correctly under unideal and highly dynamic circumstances. This leads to a few questions: What are the variables that determine the operational conditions? What are the benefits and trade-offs from changing one of those variables? How do circuit designers exploit those variables to achieve their power and delay goals?

When considering various aspects of examining operational conditions, there are three different sources of variation: Process Variation (P), Supply Voltage (V), and Operating Temperature (T) – or PVT. Ideally, circuit designers aim to implement circuits that operate in all extreme cases of these variables. Realistically, they tweak one of the variables to achieve their design goals. By opting for a smaller process node, the area of a circuit decreases as well as the dynamic power through operating at a lower supply voltage. Nevertheless, transistors' behavior varies in lower nodes due to photolithography limitations, deviations in the optics, and doping

density inconsistencies. These cause a variation in channel length and threshold voltage, meaning that the devices' speed and power consumption are different from one transistor to another even on the same die. Similarly, temperature increase causes the drain current to decrease which slows down the circuit's speed. Since the industry is trending towards smaller process nodes with higher transistor density and switching activity, higher average temperatures will result and become a larger consideration.

Lastly, the supply voltage variation is more relevant to this research than the other two corners because it is a very powerful tool used in a wide range of applications. All devices are typically designed to operate in a range of ±10% around the nominal supply voltage. This slack lets the transistors function correctly despite voltage variation that can be caused by certain factors, such as IR drop. According to [11], speed is roughly proportional to V_{DD}. This means that the circuit is 10% faster when the supply voltage is 10% higher than nominal voltage and 10% slower when the supply voltage drops 10% under the nominal voltage. Circuit designers may intentionally lower the supply voltage to the minimum to save power. This technique is called Dynamic Voltage Scaling (DVS). Furthermore, designers may reduce the clock frequency to the minimum necessary to complete a task in schedule then cut down the voltage to the minimum necessary to operate at the new frequency. This method is called Dynamic Voltage/Frequency Scaling (DVFS). Both DVS and DVFS are used in circuits where power is a top constraint, such as consumer electronics, since they achieve high power gains according to the following equation:

$$P_{total} = P_{dynamic} + P_{static} = (P_{switching} + P_{short\ circuit}) + P_{static}$$
$$= (\alpha C_L V_{DD}^2 f + \alpha t_{sc} V_{DD} I_{peak} f) + V_{DD} I_{leakage}\ [11]$$

Equation 2. Total Power Dissipation

It is important to note that the effect of lowering supply voltage and frequency is cubed in the $P_{Switching}$ equation. This means that operating at 50% of the speed and 50% of the nominal supply voltage costs only $\frac{1}{8}$ of the switching power. To implement either method, a DVS controller is required to set the operating frequency, and then it chooses the lowest supply voltage necessary for this speed. Furthermore, characterizing a circuit across a continuous range of voltages and frequencies is very challenging, therefore timing and power engineers limit the voltage-frequency setting to *n*-discrete levels. On the other hand, MTNCL circuits do not need a controller since all asynchronous blocks adapt to the new supply voltage without any external intervention. In fact, MTNCL circuits are theoretically adaptive to all three PVT variations; however, voltage scaling is used in this research to prove the robustness of the MTNCL circuits in the case of voltage fluctuation.

III. Stream Processor Design

1. Design Methodology

The goal of this research is to implement a QDI asynchronous stream processor using the MTNCL methodology. Furthermore, comparison and analysis are conducted against a synchronous version to juxtapose the performance, robustness, and cost. As mentioned in **Section II**, designing a clock tree in a stream processor is challenging given the large number of cores that can range from a few hundred to a few thousand. When also taking into consideration Process, Voltage, and Temperature (PVT [11]) corners, ensuring all cores are latching the correct data can be difficult. On the other hand, blocks are self-timed in a QDI asynchronous design. That guarantees the design will work properly as long as the respective cores and blocks in sequence exchange the handshaking signals correctly. It is important to note that one of the main benefits from the clock omission is increasing the circuit's robustness in various conditions while reducing the logic complexity that results from additional clock circuitry. However, this will not necessarily translate into area reduction because MTNCL uses a dual-rail encoding scheme. After a thorough analysis, MTNCL remains the best candidate.

2. Define the Stream Processor's Application

As mentioned in **Section II.3**, stream processors exhibit three kinds of parallelism: Data, Task, and Pipeline. This permits them to accomplish a variety of tasks, especially the ones comprised of multiple simple tasks. For this reason, stream processors are great for stream cipher and image processing. A stream cryptographic processor is presented in [6] which can implement various stream cipher algorithms such as Grain-80 and A5-1 with a throughput of 100 Mbps and 66.67 Mbps, respectively. Nevertheless, the usage of stream encryption is limited to specific applications, according to [12], due to its low diffusion and susceptibility to insertions and

modifications. Consequently, this research focuses on image processing over stream cipher. Stream architectures reduce the bandwidth demands by streaming the image pixels into an array of processing elements and transferring intermediate data results to the proceeding processing element instead of storing them back to the memory [8]. Since memory-related operations contribute significantly to total latency in the conventional general-purpose processors, stream processors have the potential to be much faster. Combined with efficient exploitation of parallelism, stream processors are able to reach a range of hundreds of GOPS [7].

3. Stream Processor's Building Blocks

As noted in **Section II.3**, every stream processor consists of multiple nodes that have multiple cores. Since this research is one of the earliest attempts to build an MTNCL stream processor, the preliminary version should not be complicated when applying the three types of parallelism. Accordingly, this architecture should have at least three cores. As depicted in **Figure 5**, task parallelism requires a minimum of two cores to execute a task in parallel. These nodes need to feed their results to another core or vice versa to implement the pipelining aspect, or a *produce-consumer* relationship [7]. Next, a large variety of filters were discussed in [10]. Smoothing filters and histogram equalization were the most attractive since both can be simple to physically implement.

4. Stream Processor High-Level Architecture

Figure 9 depicts the high-level architecture of the two-node stream processor. Both nodes share the same architecture with a total of three cores. Two of the cores implement the smoothing filter, whereas the third implements the histogram equalization. Since the cores are processing pixels which represent the amount of gray intensity from 0 to 255, the data buses should each be 8-bit wide. This implements the data parallelism principle, while both smoothing cores function in

parallel to achieve the task parallelism principle. In most circumstances, the output of the Histogram Equalization (HEQ) core will be fed into the Smoothing Filter (SF) cores to establish the pipelining concept. Task parallelism is implemented across nodes by smoothing an input image using all four SF cores. Also, each node has a local dual-port SRAM of 32Kb for two reasons. First, a dual-port SRAM allows a core to write its output pixel while another core fetches its input pixel. Second, the 32Kb storage is enough to let the processor run on only one node instead of both nodes together which adds a reconfigurability feature to the processor. The 32Kb SRAM is split into two equal partitions to exploit the task parallelism feature by storing the output stream of pixels from both SF cores at the same time. Otherwise, the second

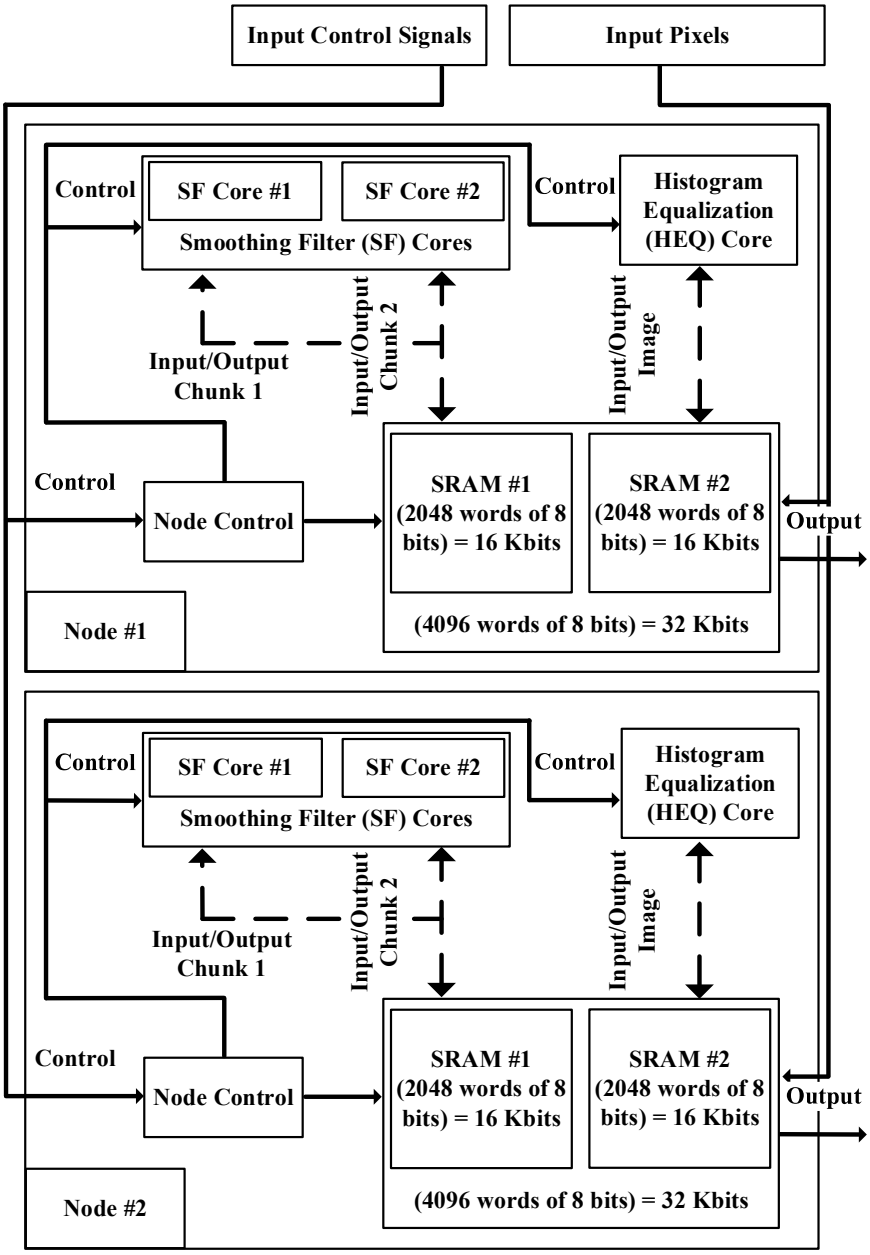

Figure 9. High-Level Architecture of the Proposed Stream Processor

core would wait for the first core to finish smoothing before it began feeding new pixels to the SRAM. This partitioning should reduce the delay significantly. Unfortunately, the HEQ core does not benefit from the partitioned storage since there is only one core providing the data. It is also important to note that equalization cannot be parallelized because the calculation of the new shades depends on the occurrence of the shades in the original picture. If the core receives half of the picture, then the calculation implements a different version of equalization, called Regional Adaptive [10], which this dissertation work does not cover. According to [10], the smoothing step cannot be followed by equalization because each pixel is averaged with its neighboring pixels, thereby diverting the original histogram of the image. The inaccuracy of the smoothed histogram results in the equalized image having either low or high contrast. On the other hand, equalized images can be smoothed since the new shades are derived from their original values.

5. Stream Processor Modes of Operation

After featuring the high-level architecture of the stream processor in previous sections, the protocols that the blocks use to communicate with each other must now be addressed. For simplicity, all instructions were classified into two main groups: intra-node and inter-node modes of operation.

In the intra-node mode, the user is smoothing, equalizing, or equalizing-smoothing in only one node – node #1 or #2. The control logic directs the input pixels to the correct core, either SF or HEQ. In the case of smoothing, the SF Core #1 smooths the top half of the picture while the SF Core #2 smooths the bottom half. The control logic also handles the case of the equalizing-smoothing operation by connecting the output of the HEQ core to the input of the SF. Whenever the core is ready to output the pixels of the output image, the SRAM I/O logic ensures that the pixels are stored in the right SRAM core depending on the operation. For the smoothing and

equalizing-smoothing instructions, the SRAM stores the pixels from the SF Core #1 in SRAM #1 while the pixels from the SF Core #2 are stored in SRAM #2. In the case of equalizing, the memory input logic stores the first 2048 pixels (top half of the equalized image) in SRAM #1; however, the remaining pixels (bottom half of the equalized image) are stored in the second partition. When the operating node is outputting the results, the SRAM output logic starts at the first pixel in the first partition. Whenever the counter hits ($\frac{\text{total number of pixels}}{2}$), the logic starts processing the first pixel in the second partition.

In the inter-node mode, the user is either smoothing or equalizing-smoothing across two nodes; or the user is smoothing, equalizing, or equalizing-smoothing two pictures at the same time. In the first case, smoothing is performed across four cores – two cores for each node – where each individual core smooths only a quarter of the original picture. To avoid parallelizing the equalization step in the equalizing-smoothing case, equalization is performed twice in each node, then the equalized pixels are streamed out to their node's SF core. Whenever the SF cores are ready to output pixels, the SRAM input logic stores each quarter of the output image in the SRAMs. Consequently, SRAM #1 of the first node holds the top quarter of the image, SRAM #2 of the first node and SRAM #1 of the second node hold the two middle quarters, and SRAM #2 of the second node holds the bottom quarter. In contrast to the intra-node mode, the SRAM output logic is aware of what part of the image is being processed to ensure a correct sequence of pixels. Thus, an extra bit, called *ID*, was assigned to the control bus that identifies the portion each node is processing. *ID == 0* means that the top half of the input image is assigned to this node, whereas *ID == 1* refers to the bottom half. In the former case, the SRAM output logic will load the pixels from SRAM #1 then switch to SRAM #2 whenever the counter hits ($\frac{\text{total number of pixels}}{4}$). In the latter case, the SRAM output logic in the second node starts producing the third quarter's pixels

whenever the main counter in the node hits ($\frac{\text{total number of pixels}}{2}$). Finally, the SRAM output logic of the second node switches to the last quarter when its counter reaches ($\frac{\text{total number of pixels}}{4}$).

For the second sub-mode in the intra-node mode, both nodes execute their own instruction separately from each other. To step up the reconfigurability capabilities, the user can assign two different pictures to each node. In other words, this sub-mode can be portrayed as two intra-node instructions concurrently executed by each node.

6. Design's Scope

One of the main challenges with large designs is simulation time. In this project, the input picture size is a significant factor that impacts the length of simulations. Therefore, it was essential to find the right balance between smaller dimensions, where the processor's modifications are noticeable, and reducing the simulation time as much as possible. As illustrated in **Figure 10**, MATLAB simulations show that a 64×64 image is the optimal size. Another factor in reducing the simulation time is the circuit size. For the HEQ core, the image size influences the design size due to the division in the transformation equation in **Section II.5**. Since the total number of pixels is a power of two (64×64 = 4096 pixels), there is no need to design a circuit for division. Likewise, the SF core architecture depends on the size of the mask and the size of the image. For example, the implementation of a 5×5 mask requires 24 adders; however, a 3×3 mask requires only eight adders. Since the dimensions of the input image shrank to 64×64, a large smoothing mask will destroy most of the features in the input image. In addition, the coefficients of the smoothing-filter mask can considerably impact the size of the SF core. The implementation of the box filter, **Figure 6(a)**, will add a significant area and complexity cost to the SF core due to the non-power-of-two division. Thus, changing the coefficients will drastically diminish the area because n-shift-right is equivalent

to a 2^n division. It is important to note that both masks will output similar results due to the different weights, but the delta is almost negligible due to the small mask size. For the sake of architecture flexibility, the SF core implemented a weighted filter instead of the standard box filter.

IV. Implementation and Design Verification

1. MATLAB Simulations

The goal of MATLAB simulations is to generate a visual representation of the result from the image processor. This is essential to visualize the outcome of each of the mask candidates to assess the complexity of the circuit and the smoothing quality. Accordingly, a MATLAB script was developed to generate various dimensions of the input image, apply histogram equalization, apply the smoothing filter on each size separately, and export the raw and processed image of each size. **Figure 10** features the exported images.

Figure 10. Raw and Processed Images of Sizes: 128×128 (a), 64×64 (b), and 32×32 (c) [10]

Next, it was important to export the image in a computer-friendly format to be used in simulations later. Therefore, a feature was incorporated in the previous MATLAB script that exports the picture as a two-dimensional matrix of integers stored in a text file. In addition, a Python script was developed to place each pixel value on a single line because hardware simulation tools, such as Questa, read a file in a line-by-line manner. The combination of both scripts generated raw, smoothed, equalized, smoothed-equalized, and equalized-smoothed images as separate formatted-text files. VHDL and Verilog-A testbenches then used the raw image file to

feed the pixels to the design. Finally, the testbench compares the design's output pixels against the MATLAB-generated picture for logical verification.

2. Smoothing Filter Core (SF Core)

a. SF Logic

After deciding on the mask size and the value of the coefficients, it was essential to have a preliminary architecture of the SF Core to design the building blocks in VHDL. The input pixels of the SF Core will be stored in an un-spooling vector unit which was initially developed in [13] [14] and then modified to fit this design. The un-spooling unit receives nine 8-bit values then outputs all 72 bits at once. Regarding the weights, the bits belonging to the pixels with double weight are shifted left once, while the middle pixel's bits are shifted left twice. As mentioned in **Section IV.1**, a 3×3 mask requires eight ripple-carry adders (RCAs). The tree structure, illustrated in **Figure 11**, was preferred over the cascading structure because it reduces the complexity from $n-1$ adder levels to $\lceil \log_2 n \rceil$. This reduces the complexity from $T_{cascading-ripple-multi-add} = O(k+n)$ to $T_{tree-ripple-multi-add} = O(k + \log(n))$ according to [14]. Lastly, the four least significant bits

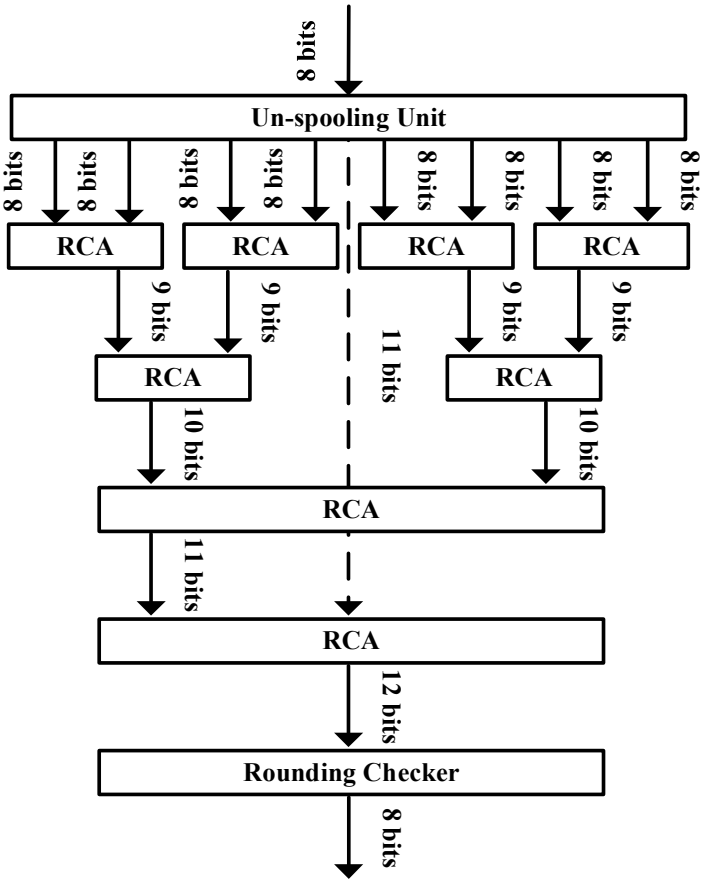

Figure 11. High-Level Architecture of the SF Logic

(LSBs) are truncated to implement the division by 16. For rounding, a rounding checker unit looks at the truncated result and the 4th LSB. If the 4th LSB is DATA1 in MTNCL or LOGIC1 in SYNC (decimal value 0.5), an adder will add one to the new pixel value; otherwise, the output stays the same.

b. SF Input/Output (I/O) Logic

1. Design Challenges and Solutions

As mentioned in the previous section, the SF Logic processes nine pixels to calculate a new pixel. **Figure 12** illustrates a 5×5 image where the numbers represent the pixels' ID, and the highlighted cells represent the input pixels required to smooth pixel #7. As shown in the example, the input pixels correspond to pixels: 1, 2, 3, 6, 7, 8, 11, 12, and 13. One major issue, however, is that the input pixels are not in sequence. Consequently, the SF I/O logic should feed the SF logic the three pixels above the target pixel then move down one row and fetch the three pixels where the center pixel is the target. It should then move down one final row and grab the three pixels below the target pixel. Additionally, the logic needs to cover edge cases such as smoothing border pixels. There are multiple ways to handle this scenario; however, I opted to add a ring of zero (black) pixels around the image, as depicted in **Figure 13**, where the numbers represent the shade of a pixel. The I/O Logic needs to provide an additional stream of pixels for the second SF Logic to implement the node-level parallelism, as described in **Figure 9**. Also, the logic is aware if the user opted to split the smoothing step over four (two nodes) or two (one node)

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Figure 12. Box Filter to Smooth Pixel #7

90	90	90	90	90
90	90	90	90	90
90	90	90	90	90
90	90	90	90	90
90	90	90	90	90

0	0	0	0	0	0	0
0	90	90	90	90	90	0
0	90	90	90	90	90	0
0	90	90	90	90	90	0
0	90	90	90	90	90	0
0	90	90	90	90	90	0
0	0	0	0	0	0	0

Figure 13. 5×5 image (a), Padded 5×5 image (b)

SF cores. As a result, the I/O Logic was programmed to function under three modes. The first configuration is Smoothing Filter, Parallelism Off (SF) where the image is smoothed in only one node by splitting the image into two chunks processed by each of the SF logic, respectively. The second configuration is Smoothing Filter, Parallelism On, ID 0 (SFP0) where the SF core smooths the upper half of the picture. The third configuration is Smoothing Filter, Parallelism On, ID 1 (SFP1) where the SF core smooths the second half of the picture. **Figure 14** shows an example of an 8×8 input image split across two nodes (four SF cores).

One important feature is that the smoothing logic does not require the whole image to be loaded to start smoothing. It only needs the target pixel along with the eight surrounding pixels to start execution. As an example, the SF core receives the 8×8 image depicted in **Figure 14**. In the SF configuration, the SF logic starts smoothing whenever it receives the $n_{SF} = 42^{nd}$ pixel for target pixels 1 and 33 since pixel 42 is the last received pixel that surrounds target pixel 33. In the SFP0 configuration, the SF Logic starts smoothing whenever it receives the $n_{SFP0} = 26^{th}$ pixel because target pixels 1 and 17 have all their surrounding pixels received once the 26^{th} pixel arrives. In the SFP1 configuration, the SF Logic starts smoothing once the $n_{SFP0} = 58^{th}$ pixel is ready because both pixels 33 and 49 then have all their surrounding pixels ready. This means that each configuration will result in different delays. Consequently, the SFP0 is the fastest configuration

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Figure 14. SF Core Configurations: SF (a), and SFP0 and SFP1 configurations (b)

since the design waits the least number of pixels to start smoothing, while SFP1 is the slowest given that the design waits for more than 75% of the image to start execution. For the MTNCL implementation, the SF logic waits on the pixel's value to switch from NULL to either DATA1 or DATA0 to smooth the pixel. In contrast, the SYNC SF logic needs an additional FSM to raise a flag when the needed pixels are ready. This added logic for the SYNC design reduces the gap between the MTNCL and SYNC circuit sizes and introduces more design complexity. Since the example in **Figure 14** is based on an 8×8 picture, each configuration should be expressed in an equation to scale it to the 64×64 size.

$$n_{\mathrm{SF}} = \frac{\text{Total number of pixels}}{2} + (Number\ of\ Pixels\ in\ a\ Row + 2)$$

Equation 3. SF Logic Kick-off Pixel @ SF Configuration

$$n_{\mathrm{SFP0}} = \frac{\text{Total number of pixels}}{4} + (Number\ of\ Pixels\ in\ a\ Row + 2)$$

Equation 4. SF Logic Kick-off Pixel @ SFP0 Configuration

$$n_{\mathrm{SFP1}} = \frac{\text{Total number of pixels} * 3}{4} + (Number\ of\ Pixels\ in\ a\ Row + 2)$$

Equation 5. SF Logic Kick-off Pixel @ SFP1 Configuration

2. High-level Operation

The pixels first go through an un-spooling unit where the first pixel is stored in the least significant eight bits and the 4096th pixel is stored in the most significant eight bits. Depending on the configuration, the input pixels might change. In the SFP0 configuration, one MUX will receive the stream of pixels representing the first quarter of the input image while the second MUX receives the second quarter. In the case of SF, one MUX receives the top half of the image while the other receives the bottom half. Lastly, one of the MUXes in the SFP1 mode receives the third quarter of the image, and the other receives the fourth quarter.

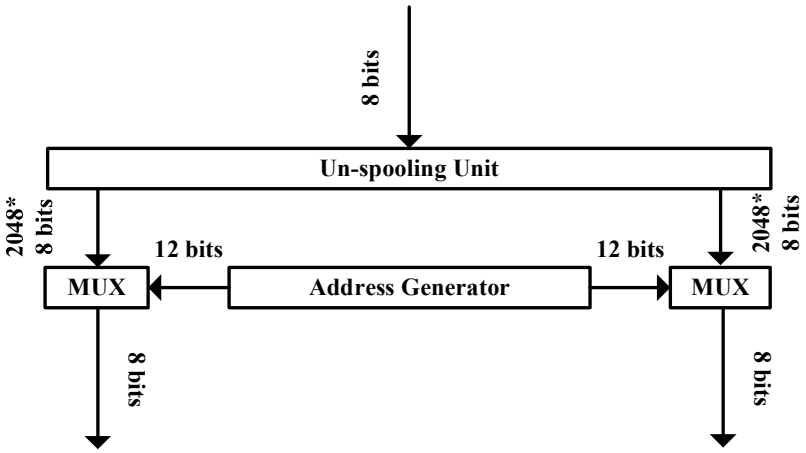

Figure 15. High-level Architecture of the SF I/O Logic

Next, an address generator controls both MUXes, as shown in **Figure 15**, to output the right sequence of pixels. As an example, the box depicted in **Figure 12** may be considered. The top counter and all the RCAs in **Figure 16** generate the addresses of the pixels in the box while the bottom counter determines the sequence of the addresses. Consequently, it counts from 0 to 8

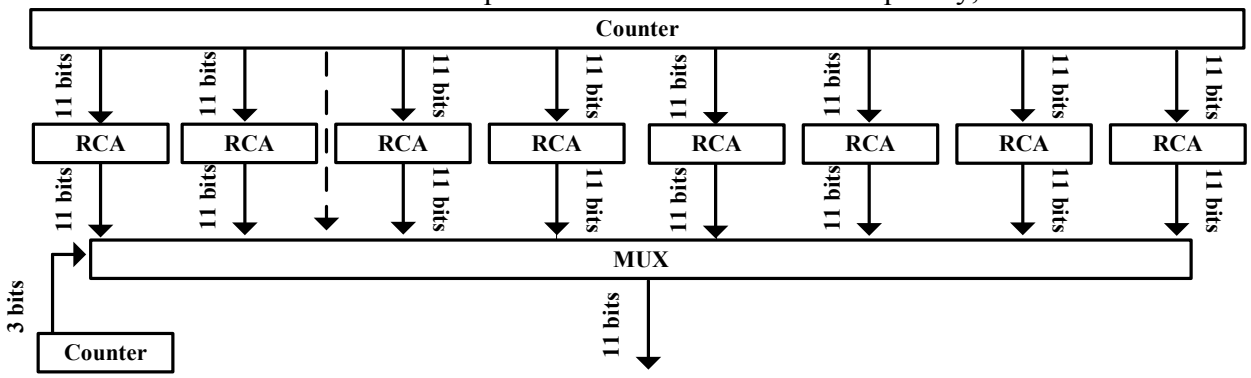

Figure 16. High-Level Architecture of the Address Generator

where the least significant bit refers to the top-left pixel in the box, and the most significant bit refers to the bottom-right pixel. Whenever the second counter reaches eight, it resets and increments the initial counter, marking that all nine pixels were fed into the SF Logic. It is important to maintain the pixel sequence to ensure that the right pixel receives the correct weight. The zero-pixel ring, added around the picture, is a challenge to this method because the extra pixels are required to smooth the original border pixels. Nevertheless, the padded pixels do not need to be smoothed themselves. Therefore, an option to skip smoothing the extra pixels was implemented as an additional feature. **Figure 17** shows the high-level architecture of the SF core described in this section.

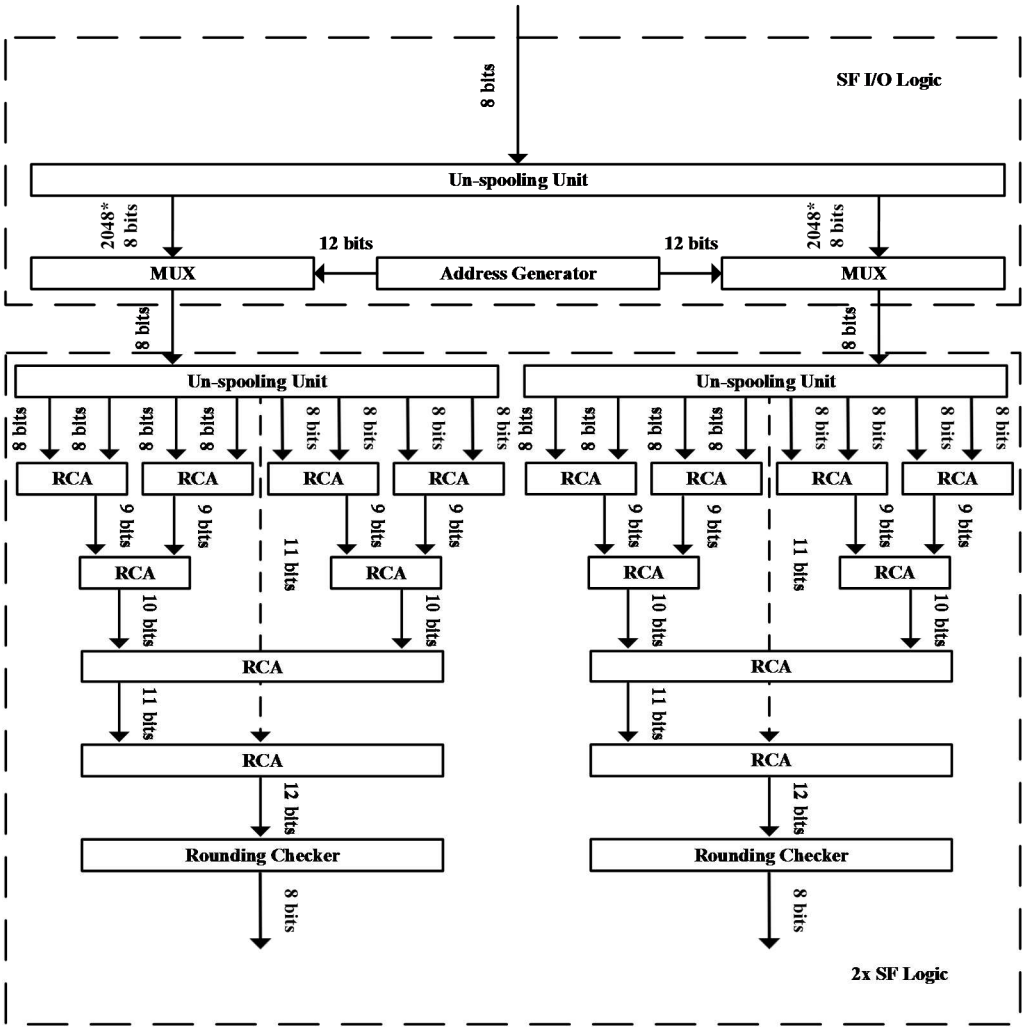

Figure 17. High-Level Architecture of the SF Core

3. Histogram Equalization Core (HEQ Core)

As depicted in **Figure 18**, there are three central units in this core: shade counter, share calculator, and image reconstructor. The shade counter is responsible for counting the occurrences of each shade in the picture. It contains 257 counters representing all the greyscale shades from 0 to 255 and a general counter for the unit to track its iteration through all the pixels. Then, the general counter turns on the shade calculator unit. This fetches the count values of each shade stored in the spooling unit for a cumulative addition and places every result in the un-spooling unit. The image reconstructor unit then goes through every pixel of the original image, stored in the SRAM, and maps each pixel's value with the new calculated intensity loaded in the un-spooling unit. Finally, it outputs the pixels of the processed image in order.

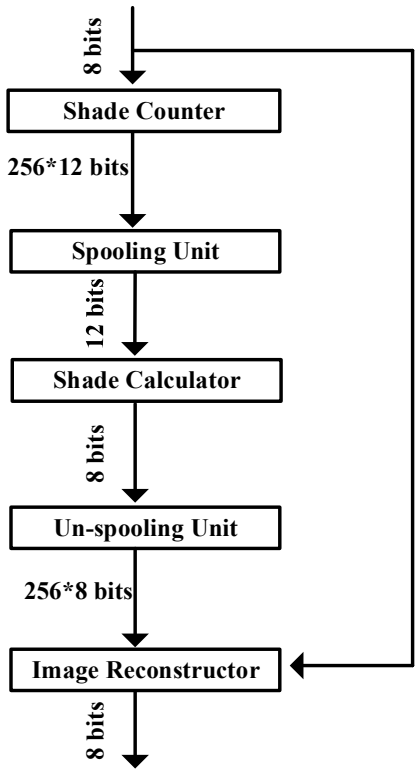

Figure 18. High-Level Architecture of the Histogram Equalization Core

4. From MTNCL to SYNC

After planning the high-level architecture of both the SF and HEQ cores, the MTNCL versions of both cores were implemented then the SYNC versions were designed based on the MTNCL architecture. Both MTNCL and SYNC VHDL netlists were developed using Gate-Level Logic instead of Register-Transfer Logic (RTL) for several reasons. First, there is no commercial

synthesis tool for the MTNCL methodology. Second, using Gate-Level Logic allowed me to ensure that the SYNC version has an almost identical architecture to the MTNCL architecture for fair comparisons. This is hard to achieve with a synthesis tool. Third, since MTNCL is a low-power paradigm by nature, the SYNC version had to be designed to achieve low-power usage as well by incorporating a clock gating technique to reduce dynamic power. After implementation, the impact of the dual-rail encoding was clear on the size of the MTNCL cores. For example, the MTNCL HEQ Core has 1,084,964 transistors, while the SYNC HEQ Core has 326,221 transistors. The MTNCL SF Core has 4,420,668 transistors, while the SYNC SF Core has 2,982,933 transistors.

5. Brief Comparison Between SF and HEQ Cores

Unlike the SF Core, the HEQ core does not need I/O Logic due to several factors. First, the sequence of the pixels is not important because the pixels are only used to decide which counter to increment in the shade counter. Second, the image reconstructor requests the image one more time from the SRAM to output the new pixels. Consequently, the output pixels follow the correct sequence since the input stream of pixels coming from the SRAM is in sequence. Third, the HEQ operation cannot be parallelized, as mentioned in **Section II.5**, because calculating the new shades is based on the occurrence of the shades in an image. Thus, all pixels need to be loaded into the same HEQ logic. The absence of the I/O logic is a great advantage for the size of the HEQ core. The SF core, in contrast, needs an I/O Logic that uses a costly 98% of the total number of transistors of the SF logic due to the reasons described in **Section IV.2**.

V. Results and Analysis

1. Simulation Setup

After developing the VHDL netlists of both the MTNCL and SYNC versions of the cores described in **Section IV**, logic simulations were performed using Questa Advanced Simulator to prove the validity of the logic. Next, the VHDL netlists were flattened using Synopsys Design Compiler and imported into Cadence Virtuoso. All designs were implemented in TSMC 65nm bulk CMOS process with a nominal supply voltage of 1.0 V. Verilog-A modules were developed to control the imported designs in the transistor-level simulations. Due to simulation time constraints, transistor-level simulations were limited to the individual cores. Nonetheless, the MTNCL SFP1's results are based on mix-and-match simulations, meaning overall power data was derived from simulations run on individual blocks. Likewise, total delay is based on the delay values gathered from the individual block simulations then multiplied by the number of operations, derived from **Equation 5.**

2. Average Active Energy Comparison

a. Method

Average energy data collection was split into two steps. The first step is to collect four timestamps:

1. When one of the inputs' bits reaches 5% of V_{DD} during the assertion of the first input data wave;

2. When one of the outputs' bits reaches 95% of V_{DD} during the assertion of the first output data wave;

3. When one of the inputs' bits falls to 95% of V_{DD} during the deassertion of the first input data wave; and

4. When one of the outputs' bits falls to 5% of V_{DD} during the deassertion of the first output data wave.

The second step is the integration between the first two timestamps to calculate the rising active energy, while the integration between the last two timestamps calculates the falling active energy. The multiplication of the average of both values and V_{DD} equals the average active energy.

b. MTNCL vs. SYNC

As shown in **Figure 19**, the MTNCL versions consume more active energy compared to their SYNC counterparts due to the dual-rail encoding. This encoding scheme also makes the MTNCL circuits larger than their SYNC circuit counterparts as mentioned in **Section IV.4**. Fortunately, however, dual-rail encoding does not translate to double the active energy due to different switching (α) patterns across different logics following this equation $P_{dynamic} = P_{switching} + P_{short\ circuit} = \alpha C_L V_{DD}^2 f + \alpha t_{sc} V_{DD} I_{peak} f$ [11]. MTNCL HEQ's average active energy is 45.18nJ, whereas the SYNC HEQ's average energy is

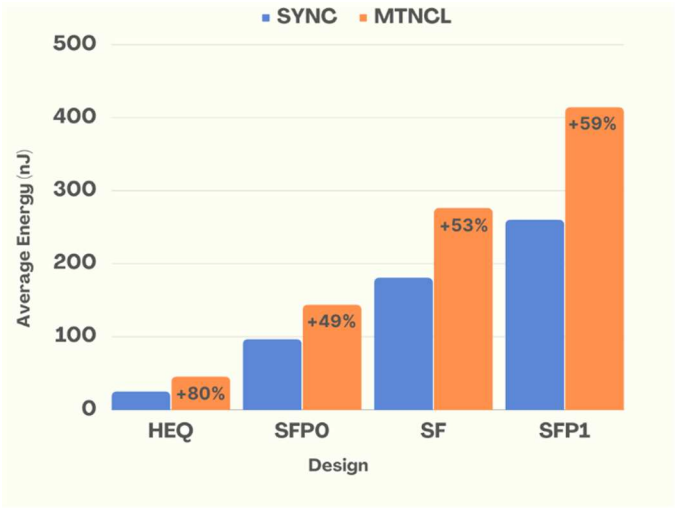

Figure 19. Average Active Energy Results

25.08nJ. Therefore, the average energy overhead is 80%; whereas, the average active energy overhead for MTNCL SFP0, SF, and SFP1 is less than 60%. The smaller overhead is caused by the logic added to the SYNC design to implement certain features that are implicitly built into the MTNCL design. As described in **Section IV.2.b.1**, the SYNC SF I/O logic needs an extra 12-bit counter to implement the FSM that starts the SF logic when enough n pixels are ready. The extra switching resulting from the additional logic reduced the SYNC's power efficiency. The following kick-off pixels were calculated based on equations **3**, **4**, and **5**. For SFP0, the SF logic starts consuming pixels whenever the 1090th ($n_{SFP0} = \frac{4096}{4} + 64 + 2$) pixel is received. While in the SF configuration, the SF logic starts after the 2114th ($n_{SF} = \frac{4096}{2} + 64 + 2$) pixel is received. For the SFP1, the SF logic waits until the 3138th ($n_{SFP1} = \frac{4096*3}{4} + 64 + 2$) pixel is fetched. As expected, the SFP1 accumulated the most switching leading to an average energy of 413.98nJ for MTNCL and 260.13nJ for SYNC. MTNCL SF consumed 275.99nJ, while SYNC SFP0 consumed only 180.70nJ. Out of all configurations, SFP0 used the least energy: 143.59nJ for MTNCL and 96.56nJ for SYNC.

c. SF vs. HEQ

In general, the HEQ designs use less energy than the SF designs due to two reasons: the size and the switching activity. Regarding the size, the MTNCL SF core is four times larger than the MTNCL HEQ core (4420668 vs. 1084964 transistors). Similarly, the SYNC SF core is nine times larger than SYNC HEQ (2982933 vs. 326221 transistors). In addition, the SF I/O Logic, which uses 98% of the total number of transistors, receives the pixels and feeds them to the MUXes, as mentioned in **Section IV.2.b.2**. Therefore, the circuit cannot be idle for any period of time which limits the low-power capabilities of the sleep mechanism in MTNCL or clock-gating

in SYNC. The Shade Counter has the highest number of transistors in the HEQ design; but only this counter, which represents the value of the shade of the input pixel, is turned on while the other 255 counters are idle. Therefore, the HEQ's average energy consumption is significantly less than the SF's.

3. Leakage Power Comparison

a. Method

Leakage power data collection was split into two steps: ground all inputs except the V_{DD} pin then integrate the current dissipation between fixed time bounds (0 to 0.5 ns) across all designs.

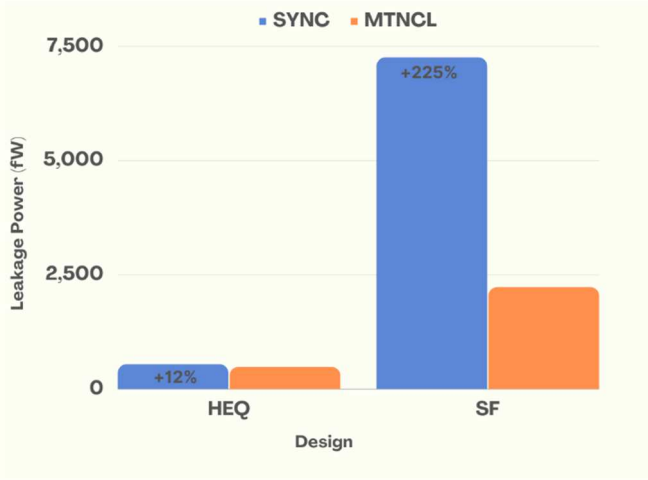

Figure 20. Leakage Power Results

b. Analysis

Unlike average energy, leakage power does not differ across the various SF configurations since they all use the same circuit. As shown in **Figure 20**, the leakage in the SF designs is much higher than the HEQ designs due to the larger number of transistors, which translates to more paths between power and ground. MTNCL SF's leakage is 2230.44fW, while MTNCL HEQ's leakage is 478.79fW. Regarding SYNC designs, HEQ consumes 535.84fW and SF consumes 7256.12fW. These results lead to a discrepancy where SYNC's leakage power is higher than MTNCL's despite much lower transistor counts. Several factors cause this discrepancy. First, the clock signal needs large buffers to drive almost every cell in the SYNC design. Wider transistor channels cause more current to leak. Second, MTNCL uses strategic high-V_T transistors to achieve better leakage performance, as mentioned in **Section II.2**. Finally, the

logic plays an important role in determining the leakage power footprint. For example, SYNC HEQ consumes 12% more leakage power than MTNCL HEQ, but SYNC SF's leakage is 225% higher compared to its MTNCL counterpart.

4. Voltage Scaling Comparison

For voltage scaling, transistor-level simulations were run at 10% decrements of the nominal voltage: 900mV, 800mV, and 700mV. Different circuit blocks were simulated under these voltage sweeps before running core-level simulations to ensure the blocks operated as expected. However, output errors were detected in the spooling units below 700mV, thus the cut-off supply voltage was set at 700mV. Power data and delay were collected since the output results of all designs were logically valid across all voltage sweeps.

a. MTNCL vs. SYNC

MTNCL designs adapt to the lower voltage without any external intervention. On the other hand, a DVS controller must be attached to the SYNC designs to control the frequency in order to allocate enough time for the gates to assert or de-assert under a different supply voltage. In addition, SYNC circuits might need a logic verifier to ensure that the output results are correct. In case of an error, the verification logic must either adjust the frequency to ensure the registers latch the correct data or correct the output results, assuming the error persists across all outputs. Consequently, SYNC designs increase in size to accommodate the DVS controller, logic verification, and output correction circuitries to guarantee the circuit's behavior maintains validity during supply voltage fluctuation. In this dissertation work, DVS controllers were not implemented; however, the MTNCL simulations were run first to determine the delay for a specific

circuit then ran the SYNC simulations with a frequency that produces the same delay to avoid timing violations. Otherwise, frequency sweeps would need to be conducted on each voltage step to determine the right frequency to achieve highest speed and lowest error. This is impractical given the large designs and long simulation times. Other simulation methods, such as mix-and-match, can be used as a workaround; but this can impact the accuracy of the data. Since SYNC designs were timed after MTNCL, both architectures scored the same delay. **Figure 21** shows the performance degradation across the SF and HEQ cores resulting from lowering the supply voltage. As mentioned in **Section II.6**, speed is proportional to V_{DD}. At 900mV, all designs slowed down by 16% on average relative to the speed at nominal voltage (1V). At 800 mV, the performance degradation reached 35% followed by a 65% degradation at 700mV.

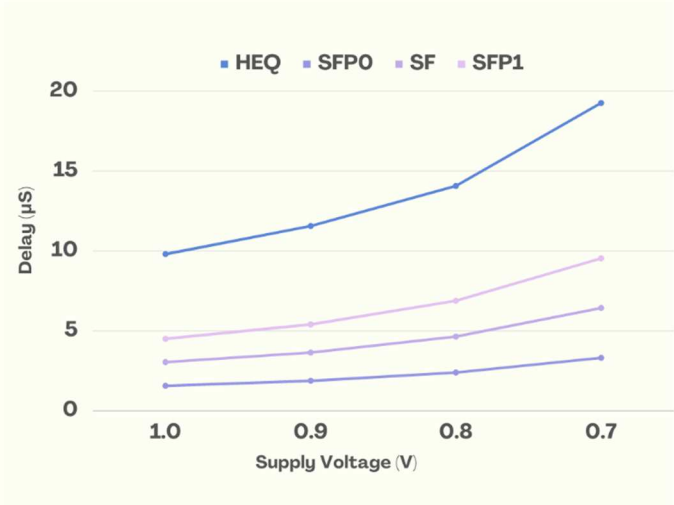

Figure 21. Impact of Lower Supply Voltage on MTNCL and SYNC's Delay

b. Operating Voltage vs. Active Energy

Table V.1 highlights the average energy results from the voltage sweeping simulations using the same method in **Section V.2.a**. MTNCL designs consume higher energy compared to their SYNC counterparts due to additional switching activity produced by dual-rail encoding. Therefore, the "MTNCL Average Energy Overhead" column was added to present the average energy increase to help quantify the trade-off in opting for MTNCL over SYNC. Interestingly, the extra switching causes the MTNCL designs to have higher power savings under lower supply voltages according to **Equation 2**. Consequently, the MTNCL HEQ benefits more than MTNCL

SF from voltage scaling due to the HEQ's higher switching activity. Thus, the average energy overhead drops from 80% at 1V to 28% at 700mV in HEQ. On the other hand, all switching occurs in the SF logic which is an insignificant 2% of the total SF core size. Also, the data is captured after smoothing only one output since average energy is calculated between the first input and first output. Consequently, the average energy overhead only drops from 54% at 1V to 60% at 700mV across all SF configurations.

Supply Voltage (V)	Design	MTNCL Average Energy (nJ)	SYNC Average Energy (nJ)	MTNCL Average Energy Overhead (%)
	1.0	45.18	25.08	80%
	0.9	32.19	22.40	44%
	0.8	25.16	18.60	35%
HEQ	0.7	19.83	15.46	28%
	1.0	143.59	96.56	49%
	0.9	113.87	71.86	58%
	0.8	88.60	54.43	63%
SFP0	0.7	67.00	42.91	56%
	1.0	275.99	180.70	53%
	0.9	218.69	135.59	61%
	0.8	170.29	102.79	66%
SF	0.7	128.78	81.41	58%
	1.0	413.98	260.13	59%
	0.9	328.03	194.75	68%
	0.8	255.43	148.24	72%
SFP1	0.7	193.17	117.43	65%

Table V.1. Average Energy Results in Voltage Sweeping

c. Operating Voltage vs. Leakage Power

Table V.2 features the leakage power results from the voltage sweeping simulations using the same method as **Section V.3.a**. Unlike average energy, leakage power in the SYNC designs is

higher across all simulations. Consequently, the "SYNC Leakage Power Overhead" column was added to show how much higher SYNC design leakage is compared to MTNCL. In general, leakage power improves by reducing the supply voltage according to **Equation 2**. Leakage power drops by 25% on average in each 10% decrement in supply voltage. MTNCL's leakage declines at a faster rate than SYNC in each decrement due to the high-V_T transistors used in MTNCL gates. The logic again determines leakage power outcomes as the HEQ core does not improve at the same rate while lowering the supply voltage. For example, the SYNC HEQ leakage overhead is 55% at 700mV vs. 12% at 1V. However, the SYNC SF leakage overhead is 267% at 700mV vs. 225% at 1V. There is an improvement, but the savings for the MTNCL HEQ are much more substantial. This discrepancy is due to the higher amount of switching in the HEQ core, whereas the transistors in SF core do not switch states as often.

Design	Supply Voltage (V)	MTNCL Leakage Power (fW)	SYNC Leakage Power (fW)	SYNC Leakage Power Overhead (%)
	1.0	478.79	535.84	12%
	0.9	360.42	516.96	43%
	0.8	269.81	402.06	49%
HEQ	0.7	200.63	311.86	55%
	1.0	2230.44	7256.12	225%
	0.9	1666.82	5371.54	222%
	0.8	1236.76	4240.86	243%
SF	0.7	910.00	3340.84	267%

Table V.2. Leakage Power Results in Voltage Sweeping

d. Performance vs. Power

This chapter started with a brief explanation of the simulation setup process followed by power analysis at nominal voltage. Next, the impact of supply voltage on average energy and leakage power was presented separately. This section is a synthesis of all the simulation variables

(supply voltage, power, and delay) to grant circuit architects a better perspective when discerning a design methodology. The delay is equivalent across SYNC and MTNCL because SYNC circuits were timed to meet MTNCL circuit delays. Additionally, SYNC power data does not account for a DVS controller or output verifier to guarantee the circuit's functionality; therefore, the power data may increase when the extra logic is added. However, all MTNCL data is complete. As shown in **Table V.3**, the data gives a clearer picture of both MTNCL and SYNC behavior in various operational scenarios. For example, if HEQ is deployed in a scenario where supply voltage swings between 700mV and 1V, then designers should expect a 97% increase in delay in the worst case. The voltage fluctuation can be due to extreme conditions or intentional. If the supply voltage swings are due to extreme conditions, then MTNCL should be the better choice because the circuit operates at the maximum possible speed. For SYNC circuits, DVS controllers do not provide a continuously adjustable clock, meaning that each supply voltage is tied to a set frequency value in a discrete fashion. This means that the circuit might operate at a slower frequency than required if the supply voltage level is between two of the discrete voltage levels specified in the DVS. Also, the DVS controller and logic verifier become more complex when they cover a wider range of supply voltages. Therefore, architects may opt for SYNC if the supply voltage for their application fluctuates only between 85% and 100% of the nominal voltage. Finally, V_{DD} reduction can be intentional to save power or cool off the circuit. In this case, the additional SYNC logic overhead will not be as significant since the supply voltage will drop to predetermined levels. This should guarantee the maximum performance for SYNC circuits at lower voltages.

Supply Voltage (V)	Design	MTNCL Average Energy (nJ)	SYNC Average Energy (nJ)	MTNCL Leakage Power (fW)	SYNC Leakage Power (fW)	Delay (µs)
	1.0	45.18	25.08	478.79	535.84	9.80
	0.9	32.19	22.40	360.42	516.96	11.56
	0.8	25.16	18.60	269.81	402.06	14.07
HEQ	0.7	19.83	15.46	200.63	311.86	19.26
	1.0	143.59	96.56	2230.44	7256.12	1.56
	0.9	113.87	71.86	1666.82	5371.54	1.87
	0.8	88.60	54.43	1236.76	4240.86	2.39
SFP0	0.7	67.00	42.91	910.00	3340.84	3.31
	1.0	275.99	180.70	2230.44	7256.12	3.04
	0.9	218.69	135.59	1666.82	5371.54	3.64
	0.8	170.29	102.79	1236.76	4240.86	4.64
SF	0.7	128.78	81.41	910.00	3340.84	6.43
	1.0	413.98	260.13	2230.44	7256.12	4.50
	0.9	328.03	194.75	1666.82	5371.54	5.40
	0.8	255.43	148.24	1236.76	4240.86	6.88
SFP1	0.7	193.17	117.43	910.00	3340.84	9.54

Table V.3. Performance vs. Power Results in Voltage Sweeping

VI. Conclusion

In this research, two stream processor cores, HEQ and SF, were implemented in both MTNCL and SYNC methodologies. During the early stages of this dissertation work, MTNCL showed a high-level of architectural flexibility when integrating the different design blocks in their respective cores during the implementation stage. On the contrary, SYNC designs had to go through a larger number of simulations to ensure the designs met all timing constraints.

The architectural flexibility comes at a price. Simulations show that MTNCL circuits consume more average energy than their SYNC counterparts due to dual-rail encoding. The average energy overhead varies from 28% to 80% depending on the logic; however, MTNCL can save up to 267% leakage power compared to SYNC. This is due to several reasons. First, MTNCL gates use high-V_T transistors, which are slower than regular transistors, but significantly reduce leakage power. Second, the absence of the clock signal, which routes to nearly every gate, helps reduce leakage power due to the elimination of large clock driving cells. Additionally, maintaining the clock signal to meet timing constraints can become burdensome for larger SYNC circuits due to the extra logic in the clock tree.

Another important feature is circuit robustness. MTNCL circuits adapt efficiently to their operational conditions. In this dissertation, performance and power data were collected from voltage scaling simulations at 700mV, 800mV, 900mV, and 1V. Results indicate that power consumption in MTNCL circuits scales better at lower supply voltages than SYNC, and average energy also drops at a faster rate when the supply voltage is lowered. Therefore, MTNCL methodology is a great design choice in scenarios where supply voltage fluctuates—intentionally or not. On the other hand, SYNC circuits need additional circuitry, such as a DVS controller and logic verifier, to acquire equivalent robustness.

VII. Recommendations

Based on **Section IV** and previous work [14], MTNCL circuits proved their architectural flexibility. This is manifested by skipping the clock tree synthesis step in the physical design flow. Consequently, MTNCL methodology is recommended to implement large modular designs, such as GPUs.

MTNCL circuits are also robust in continuously changing operational conditions. Reducing supply voltage is used as a technique to reduce the speed, if performance is not a priority, or target heat dissipation to cool off circuits. Phones, for example, can get too hot due to high CPU utilization or from the surrounding temperature. Therefore, scaling down the supply voltage is used to save the transistors from permanent damage due to excessive heat dissipation. As described in **Section II.6**, this technique can also be used to reduce dynamic power which can preserve the phone's battery life in scenarios where power is limited, such as applications deployed to space. As mentioned in [10], the execution of image processing tasks is handled in space to relieve the burden on the communication channel between Earth and the spacecraft.

The adaptability to variable supply voltages can be combined with other features offered by MTNCL such as leakage power saving. Also, MTNCL circuits are more resilient to physical variations like age and manufacturing variations. In fact, leakage power and manufacturing variations impose serious challenges to lower process nodes. This makes MTNCL a great alternative for space applications since space-grade electronic devices are expected to function for tens of years due to the inconvenience and cost of replacing or maintaining those devices.

VIII. Future Work

For future work, taping-out this design is recommended to provide a clearer picture of the advantages and disadvantages because additional issues might be discovered at the physical design level. For example, extra buffers might be added to the clock tree to meet timing constraints during later stages of clock tree synthesis. In contrast, the MTNCL designs should have minimal changes since they are QDI.

Next, additional simulations are advised to cover the rest of the PVT [4] corners. This includes simulating during temperature fluctuations to observe how the MTNCL circuits would react. The logic is expected to adapt to the temperature variations by slowing down when the temperature increases or speeding up when the temperature decreases. However, SYNC circuits would still need a frequency controller to adjust the circuit's speed based on the current temperature. Furthermore, MTNCL circuits are fault-tolerant as they either output valid results or NULL, while the SYNC circuits output results in all cases. Therefore, a logic verifier is required to flag or correct the output results. The implementation and analysis of the SYNC circuits with logic verifier help find the breakpoint when adding extra logic to a circuit becomes too costly, making MTNCL a better option.

Another corner in PVT is the process variation. Thus, building the design in lower process nodes exploits the leakage power gains that MTNCL offers. As featured in **Sections V.3.b** and **V.4.c**, MTNCL designs are expected to perform better than the SYNC designs. Consequently, designers might opt for MTNCL in lower process nodes since leakage power is a greater concern. Lastly, implementing additional designs, where MTNCL could have strong advantages, is recommended. Future researchers should opt for smaller cores to exploit the sleep control

mechanism in MTNCL, and they should opt for higher core counts as this will lead to a large clock tree in the SYNC version.

IX. References

[1] K. Haulmark, W. Khalil, W. Bouillon and J. Di, "Comprehensive Comparison of NULL Convention Logic Threshold Gate Implementations," in *2018 New Generation of CAS (NGCAS)*, 2018.

[2] S. C. Smith, Designing asynchronous circuits using NULL convention logic (NCL), San Rafael, Calif. (1537 Fourth Street, San Rafael, CA 94901 USA): Morgan & Claypool Publishers, 2009.

[3] L. Zhou, R. Parameswaran, F. Parsan, S. Smith and J. Di, "Multi-Threshold NULL Convention Logic (MTNCL): An Ultra-Low Power Asynchronous Circuit Design Methodology," *Journal of low power electronics and applications,* vol. 5, pp. 81-100, 2015.

[4] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owens and B. Towles, "Exploring the VLSI scalability of stream processors," in *NINTH INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, PROCEEDINGS*, LOS ALAMITOS, 2003.

[5] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson and J. D. Owens, "Programmable stream processors," *Computer (Long Beach, Calif.),* vol. 36, pp. 54-62, 2003.

[6] L. Nan, X. Yang, X. Zeng, W. Li, Y. Du, Z. Dai and L. Chen, "A VLIW architecture stream cryptographic processor for information security," *China communications,* vol. 16, pp. 185-199, 2019.

[7] B. K. Khailany, T. Williams, J. Lin, E. P. Long, M. Rygh, D. W. Tovey and W. J. Dally, "A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing," *IEEE journal of solid-state circuits,* vol. 43, pp. 202-213, 2008.

[8] S. Smets, T. Goedeme, A. Mittal and M. Verhelst, "2.2 A 978GOPS/W Flexible Streaming Processor for Real-Time Image Processing Applications in 22nm FDSOI," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019.

[9] W. Khalil, K. Haulmark, M. Howard and J. Di, "Enhancing Voltage Scalability of Asynchronous Circuits through Logic Transformation," in *2019 SoutheastCon*, 2019.

[10] R. C. Gonzalez, Digital image processing, Reading, Mass: Addison-Wesley, 1992.

[11] N. Weste and D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective, 4th ed., USA: Addison-Wesley Publishing Company, 2010.

[12] M. A. Bishop, Introduction to computer security, Boston: Addison-Wesley, 2005.

[13] S. Nelson, S. Kim, J. Di, Z. Zhou, Z. Yuan and G. Sun, "Reconfigurable ASIC Implementation of Asynchronous Recurrent Neural Networks," in *27TH IEEE INTERNATIONAL SYMPOSIUM ON ASYNCHRONOUS CIRCUITS AND SYSTEMS (ASYNC 2021)*, LOS ALAMITOS, 2021.

[14] S. Nelson, *Low-Power and Reconfigurable Asynchronous ASIC Design Implementing Recurrent Neural Networks,* ScholarWorks@UARK.

[15] A. Suchanek, Z. Chen and J. Di, "Asynchronous Circuit Stacking for Simplified Power Management," in *SoutheastCon 2018*, 2018.

[16] A. L. Suchanek, *Asynchronous circuit stacking for simplified power management,* Fayetteville, Arkansas: [University of Arkansas, Fayetteville], 2018.

[17] B. Parhami, Computer arithmetic : algorithms and hardware designs, New York: Oxford University Press, 2000.