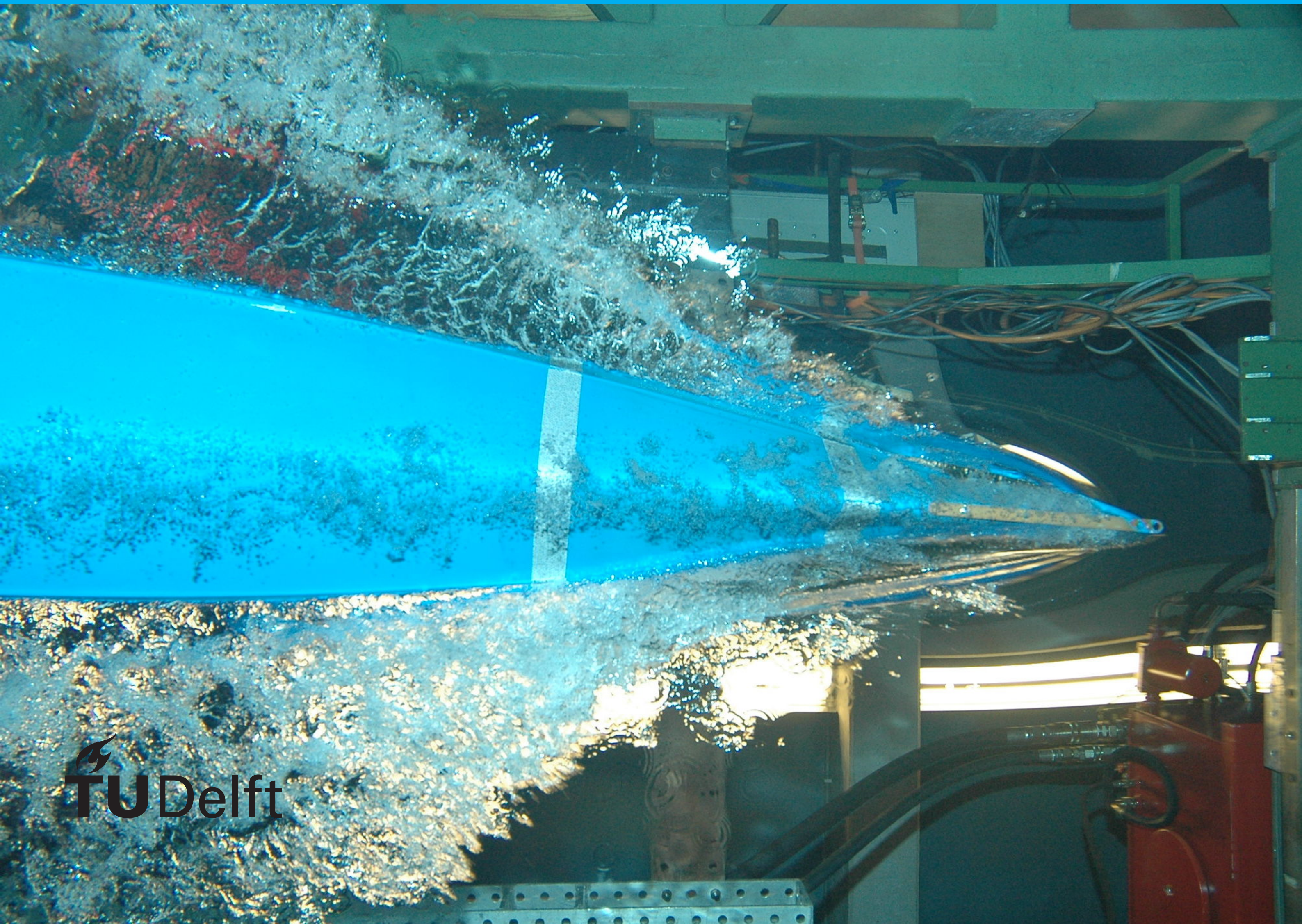
Address Calculation Acceleration

for energy efficient computing

I. Bourhaeil

TU Delft – EEMCS



TU Delft

Address Calculation Acceleration

for energy efficient computing

by

I. Bourhaeil

to obtain the degree of Master of Science in Computer Engineering
at the Delft University of Technology,
to be defended publicly on Tuesday July 9, 2024 at 15:30.

Student number:	4875400	
Project duration:	April 1, 2023 – July 1, 2024	
Thesis committee:	dr. S. Hamdioui,	TU Delft, supervisor
	Dr. G. Gaydadjiev,	TU Delft
	Dr. S. Vollebregt,	TU Delft
	Dr. F. Catthoor,	IMEC, supervisor

This thesis is confidential and cannot be made public until July 8th, 2025.

TU Delft

Preface

This thesis is the product of my work for the past 13 months. I would like to start by thanking my supervisor Said Hamdioui for his support and guidance during it. I would like to also thank Francky Catthoor for offering me this project and for all of his support while I had to climb the learning curve to be able to reach the results that I needed. I would like also to extend my appreciation to Dwaipayan Biswas for guiding me through the project and making sure I had everything needed, to Dawit Abdi for his time helping me set-up all the tools and software, and to Samantha Liu for her help explaining to me many things I needed to understand.

I would like to also thank my parents for all their support during my educational journey.

I. Bourhaeil
Delft, July 2024

Contents

Abstract

The energy consumption of computing systems is predominantly driven by memory accesses, a challenge increasingly exacerbated by the advances of data-intensive applications. To address this critical bottleneck, Computation in Memory (CIM) has emerged as a new paradigm aimed at mitigating the memory wall by significantly reducing data transfers between the memory system and the processor, thereby lowering overall energy consumption. Extensive research on CIM has concentrated mainly on reducing the data portion of communication between processor and memory. Reducing the communication of addresses and control bits between the processor and memory is another option that could be explored for further reduction of energy consumption. The Address Calculation Acceleration (ACA) within the memory introduces a novel approach that targets the reduction of address and control bits within this communication channel. ACA achieves this by offloading address generation to the sub-array level of a memory system, which substantially reduces memory bus usage and overall energy consumption. Having the addresses generated locally on the sub-array level removes the need to transfer them over the memory bus for each memory operation. The potential of ACA is considerable, given that address and control bits can sometimes comprise more than half of the communication over the memory bus (especially for large loops of code), and that the energy consumed by driving the memory bus dominates the energy consumed by memory operations. Despite its potential, an implementation of ACA has been lacking, necessitating a thorough evaluation of its impact.

This thesis addresses the exploration, design, implementation and evaluation of ACA for SRAM intra-System on Chip (SoC) memory. It proposes a circuit design of ACA that minimizes energy consumption within the memory sub-array. This proposed ACA design is compared against both the basic ACA design and a sub-array without ACA integration. After implementation using IMEC gate-all-around (GAA) nanosheet CMOS research process design kit (PDK) and simulating kernels on the sub-array level (Guided Filter, Forward Fan-Beam projection, March algorithm), the comparative analysis reveals that relative to a sub-array without ACA, the proposed ACA achieves a reduction in energy of 5-10% at the sub-array level. This reduction of energy does not account for the energy savings expected from reducing memory bus usage, which is expected to lead to a much more significant reduction of energy consumption on the system level.

This work highlights the potential of ACA to substantially improve the energy efficiency of computing systems by reducing the memory bus usage, and also reducing energy consumption within memory sub-arrays. The findings highlight the significance of continuing development of ACA, aiming to further integrate this solution at the system level (e.g. multi-core) and to assess the overall energy savings.

1 Introduction

1.1. Motivation

Computing performance is increasingly constrained by the memory access bottleneck that exists between the processor and the memory. This bottleneck affects the overall performance and efficiency of computing systems [1]. As the processor performs computation, it requires data from memory, leading to constant data transfer across this channel. However, the channel's limited bandwidth and inherent latency mean that the processor often experiences delays waiting for the necessary data to arrive [2]. This results in substantial inefficiencies, as the processor spends considerable time idle, unable to proceed with computations.

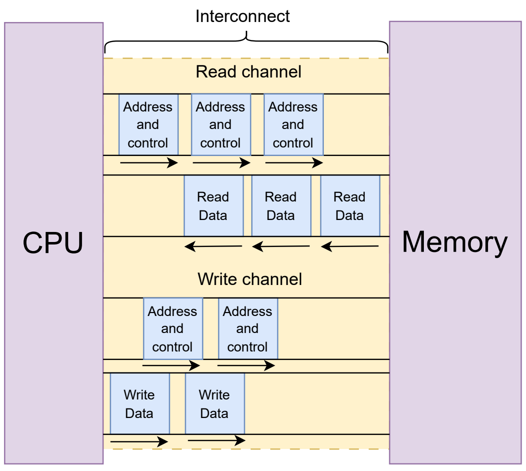


Figure 1.1: Read and Write transactions between CPU and Memory without bursts. Address and control bits are transmitted for every read data or write data block

In addition to performance issues, the memory access bottleneck in computing systems significantly impacts energy consumption [3]. In fact, for any computer architecture where memory requests are generated at a distance from the memory, a considerable portion of this energy expenditure arises from the overheads associated with data communication between the processor and memory. The communication between the processor and memory consists of three main components: the control bits (read or write for example), the target word address, and the data portion (read data sent from memory to CPU or write data sent from CPU to memory) [4]. The control bits and the address are essential for directing the data transfer but do not contribute directly to the actual computational tasks being performed. These components are therefore considered overheads of communication.

The transfer of these communication overheads between the processor and memory can account for more than half of the total transferred bits during data access operations [5]. This is because as shown in figure 1.1, address and control bits are transmitted for every read data or write data block. Consequently, a substantial amount of energy is consumed merely in transmitting these overheads, highlighting the inefficiencies caused by generating the addresses at a distance from memory. This excessive energy consumption highlights the need for more efficient data transfer methods that can minimize these overheads and improve the overall energy efficiency of computing systems.

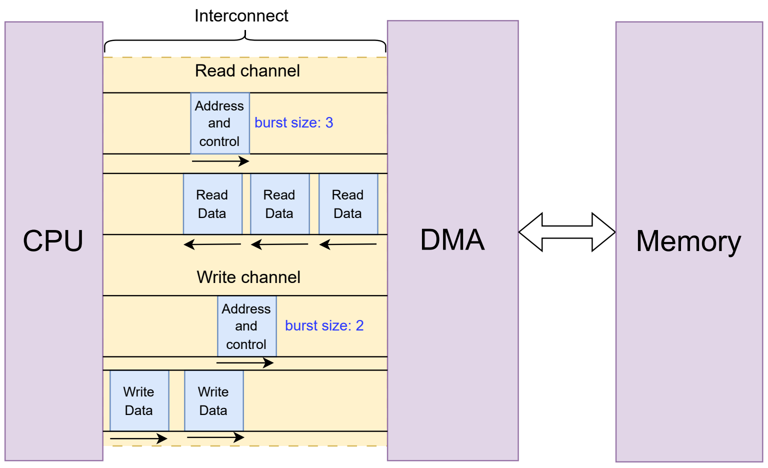


Figure 1.2: Read and Write transactions between CPU and Memory with bursts. The address and control bits are only sent once, instead of repeatedly with each data block

1.2. State of the art

To address the inefficiencies associated with data transfer overheads, one can optimize the communication. Given the loop-based nature of many computer programs, a significant proportion of memory accesses involve sequentially incrementing/decrementing addresses [6] [7] [8]. Instead of transmitting each individual address from the processor to the memory, which is both time-consuming and bandwidth-intensive, an alternative method involves sending only the initial address along with the count of subsequent sequential accesses. This technique, known as burst access [9], allows for a more efficient use of the memory channel. A dedicated address generation engine can then compute the absolute addresses locally and manage the memory interface. By reducing the volume of address transmissions, burst access minimizes the latency and bandwidth consumption associated with memory operations, thereby enhancing overall system performance. This optimization is particularly beneficial in scenarios involving repetitive data access patterns, as it streamlines the data transfer process and alleviates the processor's workload.

Modern implementations of this concept include Direct Memory Access (DMA) engines [10]. These DMA engines, strategically positioned near the processor and at an intermediate distance from the memory modules, are designed to handle basic address calculations, such as generating sequences of consecutive addresses for burst access. As illustrated in Figure 1.2, by sending a burst size along with the initial address, the subsequent addresses and control bits do not need to be calculated by the CPU and transmitted over the interconnect. This method significantly reduces the volume of data exchanged on the interconnect, depending on its topology, thereby potentially energy consumption. By offloading address calculation tasks to these specialized units, systems can achieve more efficient data transfer, allowing the main processor to focus on more complex computational tasks rather than mundane data handling operations. This not only improves overall system performance but also enhances energy efficiency, as fewer resources are expended on data transfer processes. Additionally, the reduced traffic on the interconnect can lead to lower latency and improved throughput, further optimizing system performance.

Despite these advantages, the capabilities of current DMA engines remain relatively limited, as they primarily manage straightforward address sequences. Moreover, the positioning of the DMA engines at an intermediate distance from the memory system means that the addresses generated still need to be transmitted over relatively long wires, which doesn't address the issue of overhead transmission leading to significant energy consumption and also latency. These limitations present an opportunity for further innovation and enhancement in address generation strategies, and in the position where the addresses are generated.

Due to the flexibility limitations of basic DMA engines, application-specific Address Generation Units (AGUs) have been developed to enhance flexibility and performance in address generation [11]. Their improvement is particularly evident in image processing applications, where up to 70% of all compute operations involve address computations, even after employing methods to improve data locality, such as loop transformations and memory hierarchy optimization [12]. Many AGU designs have been proposed, especially in multi-media applications (e.g. [11] and [13]).

As detailed in [14], AGUs can vary significantly in their design, flexibility, complexity, and performance gains. They are generally classified into table-based AGUs and data path-based AGUs. Table-based AGUs do not directly calculate the addresses to be generated; instead, they simply look them up in a lookup table (LUT). On the other hand, data path-based AGUs contain arithmetic logic and directly compute the addresses to be generated. Data path-based AGUs offer varying degrees of flexibility and can be further categorized into custom AGUs and programmable AGUs. Custom AGUs are tailored for specific applications, providing high efficiency for targeted tasks, whereas programmable AGUs offer greater adaptability, allowing them to handle a broader range of address generation scenarios.

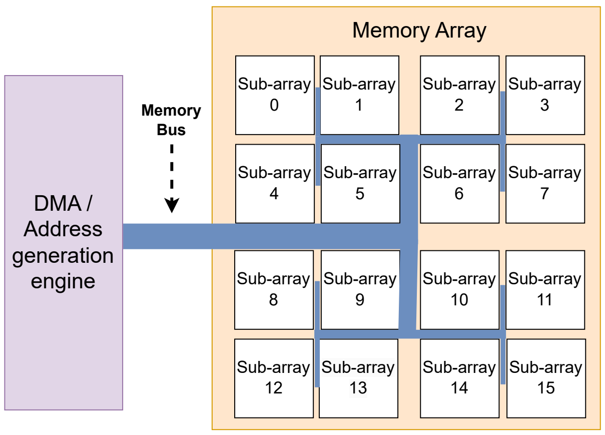


Figure 1.3: Connection between DMA and Memory array. The DMAs and AGUs need to transfer every address they calculate over the memory bus

The development and integration of these advanced AGUs address the computational inefficiencies of basic DMA engines. They are able to generate more complex data patterns and therefore offload more address calculation tasks from the CPU, thereby increasing performance particularly in data-intensive applications. However, all of these AGUs are limited by their effectiveness because these units are positioned far from where the addresses are used to access data. As a result, while they can reduce the computational burden on the processor, they do not alleviate the overhead of address transactions over the memory bus. As illustrated in Figure 1.3, the communication overhead (the addresses and control bits) still needs to be transmitted from the DMA/AGU to the corresponding sub-array for every read or write operation. This data transfer over the memory bus is responsible for the majority of energy consumption and latency in SRAM systems [15]. Therefore, there are substantial gains to be made in energy efficiency and latency reduction by minimizing the overhead associated with data transfers over the memory bus.

1.3. Research topics

To truly optimize memory access and reduce energy consumption, it is essential to explore methods that can localize address generation within the memory sub-array itself. By doing so, the volume of data transmitted over the memory bus can be minimized, thus reducing both latency and energy expenditure.

A promising approach to reducing data transmission over the memory bus is proposed in [5], which involves moving part of the DMA functionality into the memory. This is achieved by integrating **Address Calculation Acceleration (ACA)** modules into each memory sub-array, as shown in figure 1.4. The incorporation of ACA modules eliminates the need to transfer communication overheads (addresses and control bits) over the memory bus for each operation. Instead, only a high-level instruction needs to be transmitted to the ACA module. The ACA module then decodes this instruction and generates the necessary addresses locally.

This method offers several advantages. By generating addresses within the memory sub-array, the data transfer volume over the memory bus is drastically reduced, leading to lower latency and energy consumption. This localized address generation also allows for more efficient memory access patterns, further optimizing the overall performance of the system. Additionally, this approach can alleviate the computational burden on the processor, allowing it to focus on more complex tasks rather than managing data transfers. However, an implementation and analysis of the impact of the ACA modules is still to be made, and the following questions still need to be answered:

1. What is the optimal implementation of ACA to reduce energy consumption on the sub-array level?
2. At which cost does implementing ACA come at the sub-array level?
3. At which cost does implementing ACA come at the system level?
4. What are the energy savings of ACA at the system level?
5. What special features can ACA enable in memory accesses that weren't realizable before?

In this thesis, we focus on questions 1 and 2, and the rest is left to future work. While Yousefzadeh et al. [5] demonstrate the potential of Address Calculation Acceleration (ACA) in reducing energy consumption over the memory bus, a comprehensive implementation and evaluation are still required. Such an implementation is essential to accurately measure the impact of integrating ACA into memory in terms of area, energy, and performance.

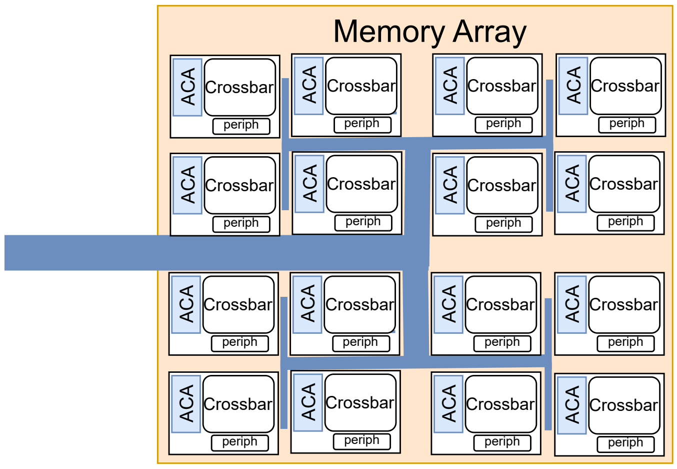


Figure 1.4: Memory Array with added ACA modules in each sub-array. Placing the ACA modules on the sub-array level removes the need to transfer the address and control bits over the memory bus for every memory operation.

1.4. Contributions

In this work, ACA is analyzed at the intra-SoC memory level. The circuit design of ACA design is improved to reduce energy consumption through various schemes, which are then compared. ACA is implemented at the SRAM sub-array level using the imec gate-all-around (GAA) nanosheet CMOS research process design kit (PDK). Finally, the proposed ACA design is simulated in conjunction with the rest of the sub-array to evaluate its energy efficiency.

The contributions of this work are:

- Analysis of the basic design of ACA and identification of its weaknesses.
- Development of a new proposed circuit design of ACA in order to significantly reduce energy consumption of ACA by using efficient clock gating techniques.
- Design space exploration of the proposed circuit design to find its best configuration in terms of energy and area.
- Implementation of the proposed ACA design on the sub-array level using IMEC GAA nanosheet CMOS research PDK. Its impact on area is measured and compared with the sub-array without ACA and with the basic ACA implementation.
- Evaluation of ACA's impact on energy on the sub-array level using circuit simulation, and comparison with the basic ACA and sub-array without ACA.

1.5. Thesis outline

This thesis is composed of 6 chapters:

- Chapter 2 presents background knowledge on in-memory computing.
- Chapter 3 presents the basic design of ACA inside the SRAM sub-array.
- Chapter 4 presents a proposed design of ACA that deals with the shortcomings of its basic design.
- Chapter 5 presents the results obtained from integrating ACA inside the sub-array.
- Chapter 6 presents the conclusion, and certain questions left for future work.

2

Computation in memory architectures

This chapter presents some background knowledge on Computation in Memory. The classification of Computation in Memory is presented in section 2.1, with the properties of its three different classes discussed. Section 2.2 presents a summary of the comparison between the different CIM classes based on available bandwidth, data movement outside the memory core and the design effort. Note that the classification presented in this chapter is also used in section 3.1 to classify ACA.

2.1. Classification

Computation in memory (CIM) emerges as a critical paradigm shift necessary to address the inherent limitations of von Neumann architectures, notably the persistent memory bottleneck [1]. This bottleneck, characterized by the disparity in speed between the processor and memory, impedes computational efficiency and hampers overall system performance. CIM offers a transformative approach by redefining where computations take place within the computing system. Its applications include big data analytics, signal processing, machine learning [16] [17] etc...

Depending on the location where the result is generated, CIM architectures can be broadly classified into three distinct types: CIM-A, CIM-P and COM.

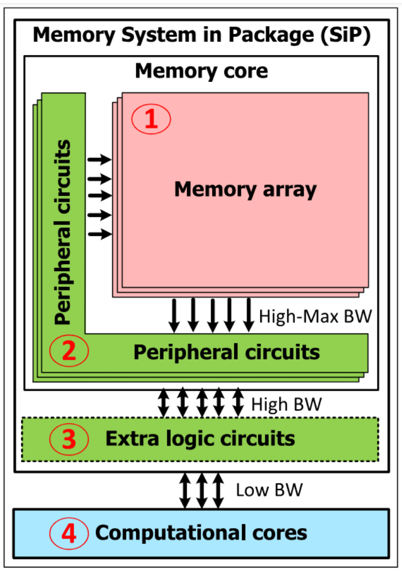


Figure 2.1: Architecture of a computing system [18]

2.1.1. CIM-A

CIM-A (Computing in Memory - Analog) refers to an architectural approach where computations are performed directly within the memory array, exploiting the analog properties of memory cells. This location is marked as (1) in Figure 2.1. In CIM-A architectures, data is stored in memory cells that possess physical characteristics allowing them to perform analog computations. These computations typically involve operations such as addition, multiplication, and more complex functions, executed at the data storage location. By harnessing the inherent analog capabilities of memory technologies, CIM-A architectures can achieve significant reductions in power consumption and latency. However, because memory cells are typically optimized for specific control voltages and currents, they need to be redesigned for CIM-A implementation.

CIM-A can be further categorized into basic CIM-A and hybrid CIM-A [18]:

- **Basic CIM-A**: in which only changes inside the memory array are required. For example, Lehtonen et al. [19] use memristors to store logic values and perform logical operations on these values. In their work, only one memory row is activated at a time, eliminating the need to modify the peripheries. This approach simplifies implementation and maintains the integrity of the existing memory architecture while still leveraging the analog computation capabilities.
- **Hybrid CIM-A**: in which modifications to both the memory array and the peripheries are needed. For instance, Kvatinsky et al. [20] propose a design where multiple rows are accessed simultaneously to perform computations, necessitating modifications to the memory array's peripheries to accommodate this functionality. This method can potentially offer greater computational throughput but at the cost of increased complexity in design and implementation. Moreover, Singh et al [21] propose a memory-periphery co-design approach to improve the accuracy of matrix-vector-multiplication (MVM) using CIM-A.

2.1.2. CIM-P

CIM-P (Computing in Memory - Periphery) involves integrating processing units within the peripheral circuits of memory chips, enabling basic computations to be performed close to the data. These peripheries typically perform operations such as bit-wise logic, addition, and multiplication, with results generated at the location marked as (2) in Figure 2.1. CIM-P is also divided into basic and hybrid categories [18]:

- **Basic CIM-P:** in which changes are required only to the peripheries, with no modifications to the memory cells. For example, Pinatubo [22] uses a customized sense amplifier to perform calculations, activating a low number of rows simultaneously, thus avoiding modifications to the memory array. This approach is advantageous for maintaining the existing memory architecture while enhancing computational capabilities.
- **Hybrid CIM-P:** in which both the memory array and peripheries require modifications. ISAAC [23], for instance, activates all rows of a memory array simultaneously to perform vector-matrix multiplication. The high current accumulated on the bit-lines necessitates modifications to the memory cells to handle this current. This method enhances computational efficiency and speed, particularly for data-intensive tasks, but requires a more complex redesign of the memory architecture.

2.1.3. COM

COM (Computation Outside Memory) refers to computations performed either in extra logic circuits within the memory or in computational cores such as CPUs and GPUs. This is marked as (3) and (4) in Figure 2.1. When computations occur in extra logic circuits near the memory, it is called Computation Outside Memory Near Memory (COM-N). When performed in computational cores, it is known as Computation Outside Memory Far (COM-F) [18]

- **COM-N (Computation Outside Memory Near Memory) or CnM (Computing near Memory)**: This approach involves integrating additional logic circuits within or close to the memory module to perform computations. This setup minimizes the distance data needs to travel, thus reducing latency and energy consumption compared to traditional von Neumann architectures. COM-N is particularly effective for applications requiring frequent memory access and intermediate computations. As an example of computation near memory, [24] proposes an In-Memory PoInter Chasing Accelerator (IMPICA). IMPICA is implemented in the logic layer of 3D DRAM technology and allows to the generation of the addresses of memory accesses to perform pointer chasing. Pointer chasing is an operation traditionally performed by the CPU in applications using linked data strictures such as trees, hash tables and linked lists. IMPICA has lead to an overall reduction of system energy consumption of 10%-41% [24].

- **COM-F (Computation Outside Memory Far):** This traditional approach involves using external computational units like CPUs, GPUs, or other processing cores to handle computations. While this method leverages the powerful processing capabilities of dedicated computational cores, it suffers from higher latency and energy costs due to the constant data transfer between memory and the processing units.

2.2. Comparison

These discussed classes of computation in memory designs can be compared in terms of various metrics. Table 2.1 shows a comparison between these CIM classes in terms of data movement outside of memory core, available bandwidth, memory design effort and scalability [18].

	Data Movement outside Memory Core	Available Bandwidth	Memory Design Effort	
			Cells & Array	Periphery
CIM-A	No	Max	High	Low/medium
CIM-P	No	High-Max	Low/medium	High
COM-N (CnM)	Yes	High	Low	Low
COM-F	Yes	Low	Low	Low

Table 2.1: Comparison of the CIM classes in terms of data movement, available bandwidth, and memory design effort

Both CIM-A and CIM-P display no data movement outside the memory core, as the data is calculated in the memory array or in the periphery. CnM however requires data to be transferred to the extra logic circuit where the computation is performed, and COM-F involved the transfer of data back to the logic core to perform the computation. When it comes to the available bandwidth of each CIM class, the classes closest to the array have the most available bandwidth. This way, CIM-A and CIM-P benefit from the highest bandwidth. They are then followed by CnM and finally by COM-F.

The memory design effort varies significantly per CIM class and per component of the memory system. When it comes to the memory cells and the array, CIM-A requires the re-design of the cells in order to allow the computation to be performed inside the array. CIM-P could require some modifications to the arrays in case of hybrid CIM-P, and the rest of the classes require no modifications. In terms of the periphery, CIM-P requires the most modifications as that's where the computation happens, which is followed by hybrid CIM-A. The rest of the classes require no modifications to the periphery.

3 ACA architecture

In this chapter, an overview of ACA on the architecture level is presented in section 3.1. Subsequently, the architecture of the baseline sub-array used is presented in section 3.2. After that, the design of the baseline ACA module from [5] inside the sub-array is presented in section 3.3.

3.1. Overview of ACA architecture

ACA is a module that is added to the sub-array in order to perform address calculation inside the memory sub-array. The aim of ACA is to perform address generation within the sub-array, reducing the communication over the memory bus in the process. This reduction of communication would lead to a reduction of energy consumption and latency of memory operations. However, its potential cost is that adding these ACA modules could lead to more area utilization, both on the sub-array level and on the system level (more wiring necessary to transmit the large high-level instructions), and could make the control on the system level more complicated.

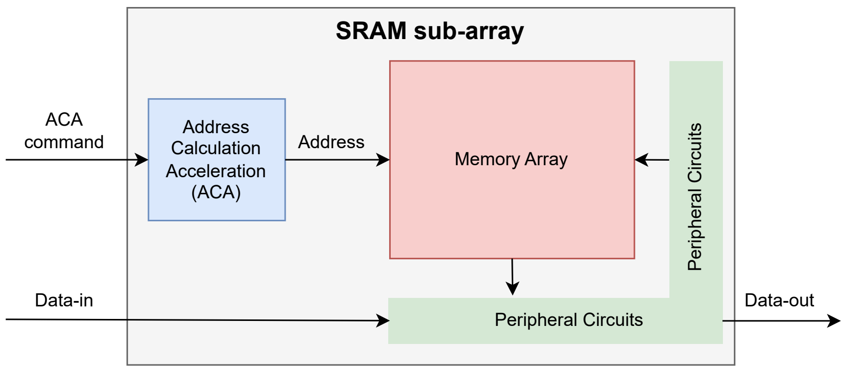


Figure 3.1: Address Calculation Acceleration inside the SRAM sub-array

Figure 3.1 shows how ACA fits inside the sub-array on an architectural level. The ACA module receives a high-level ACA command, and based on it starts address generation. While addresses are generated locally, data can then keep flowing in and out of the sub-array through the memory bus.

Figure 3.2 shows the classification of IMPICA [24], DMA and ACA. As discussed in section 2.1.3, IMPICA is implemented in the logic layer of 3D stacked DRAM. This makes IMPICA a COM-N (or CnM) type. DMAs compute the addresses outside of the memory module, making them a COM-F type. Meanwhile, ACA is implemented on the periphery circuit level of the memory array to generate the addresses. **This makes ACA a CIM-P type**. Moreover, it can be also further classified as basic CIM-P, as it only activates one row at a time normally and therefore does not require modifying the memory cells. Most of the work on CIM-P has been focused on computations involving data, and ACA has been the first work where CIM-P is used to compute addresses, rather than data. Some other

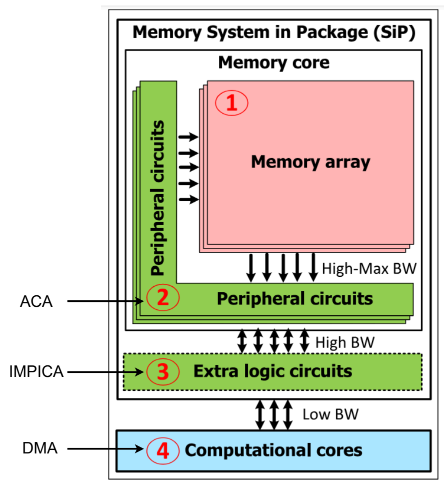


Figure 3.2: CIM Classificatoin of ACA [5], IMPICA [24] and DMA engines

works ([25] and [26]) tried to improve particular memory operations in DRAM by modifying the address decoder, but none of them optimized the address communication over the memory bus like ACA.

Unlike other CIM-P class designs which are meant more for servers and HPC (for example [27], [28] and [29]), ACA is meant for embedded applications. In embedded systems, energy consumption is a very restrictive metric in the design. This is why energy consumption on the sub-array level is given significant importance in this work.

Before exploring the design of ACA, section 3.2 presents the design of the standard SRAM sub-array without ACA.

3.2. SRAM sub-array electrical design

ACA is implemented on the SRAM sub-array level. The organization of the sub-array used as a baseline is shown in figure 3.3. The row decoder decodes the row address and drives the corresponding word-line. The column multiplexer selects a number of bit-lines corresponding to the word size of the configuration and connects them to the write driver and sense amplifier. The write driver receives the data to be written and drives the bit-lines selected by the column multiplexer. The sense amplifier reads the bit-lines selected and provides the read data.

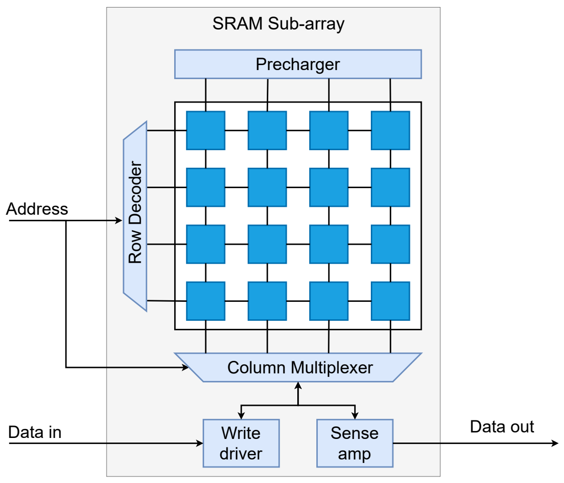


Figure 3.3: Baseline sub-array

The chosen sub-array size used has 256 rows, 256 column and a 32 bit word. This size is used since 256 by 256 is a size that can be used reliably without requiring the usage of write assist techniques (WAT), for which the circuitry needs to be designed. Going for sizes larger than 256 by 256 may lead to WATs being required. ACA is expected to perform better for larger sub-arrays as it can generate more addresses, and therefore this configuration is picked an not a smaller one.

3.2.1. Memory array

The bit-cell used is a standard 6 transistor SRAM bit-cell, with two access transistors and two inverters, shown in figure 3.4. The two inverters form a bi-stable latching circuitry that allows the storage of information. The bit-cell has two stable states, which are '0' and '1'. The SRAM bit-cell has 3 modes: hold, read and write. During the hold mode, the word-line (WL) signal is low. During the read and write modes, the WL signal is high, and therefore the two access transistors (M5 and M6 in figure 3.4) are activated, providing a connection between the bit-line (BL) and complementary bit-line ($\overline{BL}$ or BLB). During the read operation, depending on the value stored in the bit-cell, either the bit-line or the complementary bit-line gets discharged through its corresponding access transistor and pull down transistor. During a write operation, the write driver pulls either the bit-line or complementary bit-line down, and the value stored in the bit-cell is set through the access transistors.

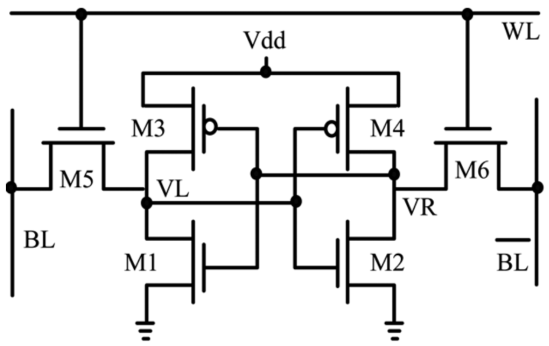


Figure 3.4: Typical 6 transistors SRAM bit-cell [30]

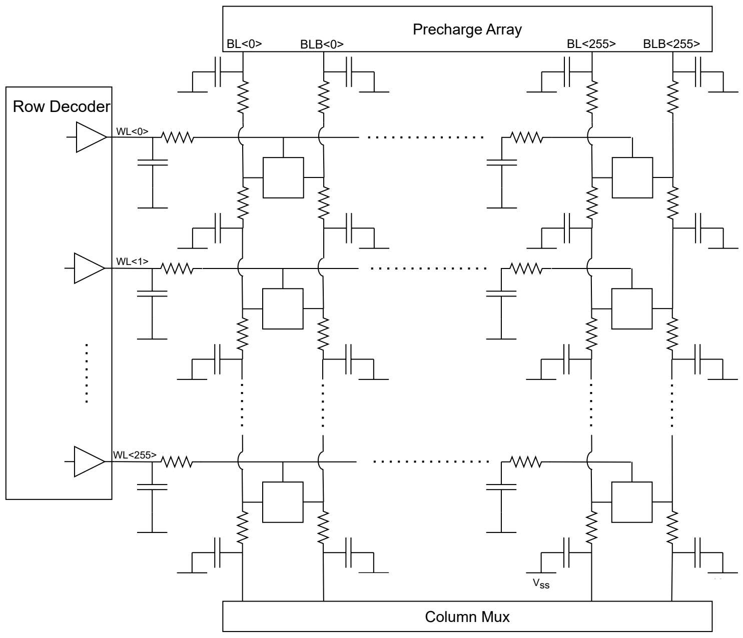


Figure 3.5: Bit-cells connected with parasitics to model crossbar

These bit-cells are connected in a crossbar format in order to form the SRAM crossbar. The crossbar size chosen is **256 rows by 256 columns**. In order to take the parasitics of the world-lines and bit-line into account, the parasitic extraction is performed on a layout of the crossbar, and the resistances and capacitances of the sections of word-lines and bit-lines connecting two neighboring bit-cells are determined. Using these parasitics, figured 3.5 shows how the bit-cells are connected with the parasitics to model the crossbar.

3.2.2. Precharger

The precharger circuit is used to precharge the bit-lines before a read or write operation is performed. The circuit is shown in 3.6 and consists of two PMOS transistors that each charge the bit-line and complementary bit-line. A third PMOS transistor is used an equalizer between BL and BLB and ensures that both bit-lines are charged equally.

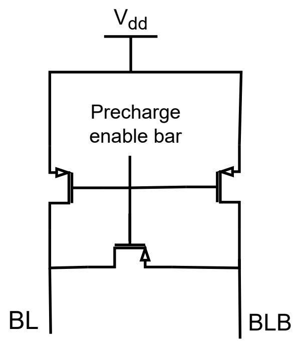


Figure 3.6: Precharge circuit

3.2.3. Sense amplifier

The sense amplifier is part of the read circuitry and is used to retrieve data stored in the SRAM sub-array. The schematic of the used sense amplifier is show in figure 3.7. The sense amplifier reads the bit-lines voltage and provides the output based on that. The shown sense amplifier is a latch-based sense amplifier. Latch based sense amplifiers are the most popular sense amplifiers due to their high sensing speed and low power consumption [31]. More specifically, the sense amplifier is a current-latched sense amplifier with an nMOS footswitch (FS-CLSA). It senses the difference in voltage between the bit-lines and amplifies it to rail-to-rail output voltages [31]. One sense amplifier is used per bit of the word size. Since the implemented sub-array has a word size of 32, then 32 sense amplifiers are implemented.

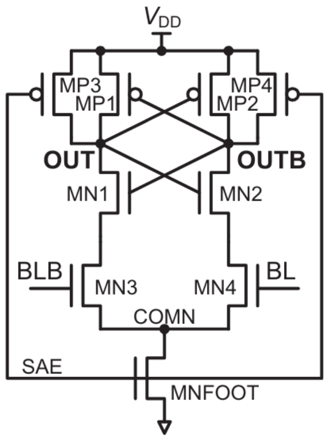


Figure 3.7: Sense amplifier schematic [31]

3.2.4. Write driver

The write driver circuit is used to discharge one of the bit-lines (BL or BLB) from its precharge level based on the value of the bit to be written. The schematic of the write driver is shown in figure 3.8. The circuit is enabled using the WE signal. One write driver circuit is used per bit of the word. Since in this implementation the word size is 32 bits, 32 write-driver circuits are implemented.

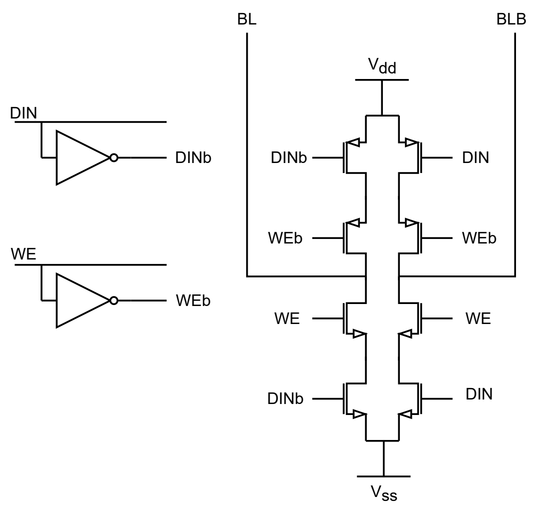


Figure 3.8: Schematic of the write driver

3.2.5. Decoder

Row Decoder Figure 3.9 shows the schematic of the row decoder. The row address is first decoded by a tree of NAND-gates, of which the output is given to an array of AND-gates connected to the clock. This ensures that the word-lines are not driven when the clock signal is low, meaning that the precharge cycle is taking place. The output of the array of AND-gates is given to an array of buffers that drive the word-lines.

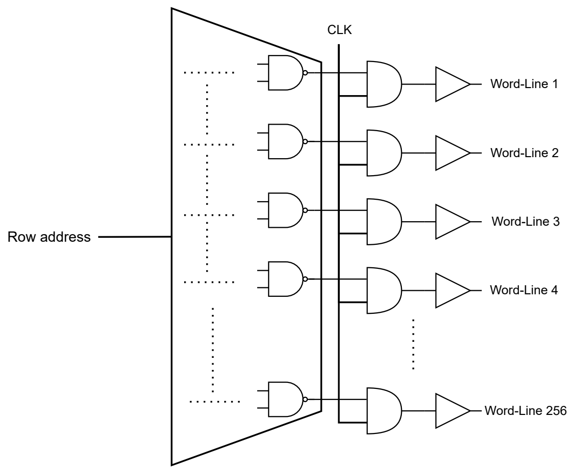


Figure 3.9: Row decoder schematic

Figure 3.10 describes the flow used to model and simulate the digital blocks (whether it is the row decoder or the ACA module) together with the other modules of the sub-array, which are modelled as analog schematics. The digital block is first modelled in RTL. After that, it is place and route-d. The area of the layout can then be obtained, Quantus can also then be ran on the obtained layout to extract the Detailed Standard Parasitic Format (DSPF) representation of the layout. a DSPF file contains all the standard cells in a layout, together with how these cells are connected, and the value of the parasitics on these connections. The DSPF representation of the digital block can then be simulated either alone or with the rest of the sub-array to check its functionality and measure the energy consumed. So the decoder is written in Verilog, then place and route-d, and simulated as a DSPF.

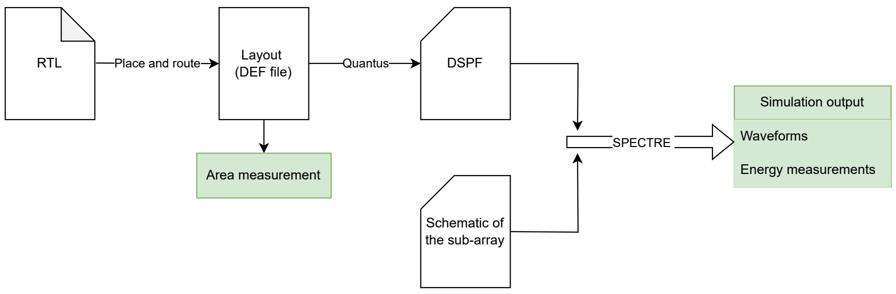


Figure 3.10: Flow to simulate place and route and perform post layout simulation of digital blocks

Column Transmission gates The column tranmission gates selects a number of bits corresponding to the word size and connects them to the write driver and sense amplifier. This implementation has a crossbar with 256 columns and a word size of 32 bits. Therefore, it selects 32 bit-lines from the 256 bit-lines coming from the crossbar based on the 3 bits of the column address.

Figure 3.11 shows the schematic of the column decoder. It is designed as a binary tree of 3 levels of transmission gates, that are controlled by the three bits of the column address. With every different combination of the three bits of the column address, a different 32 bit segment of the bit-lines is connected to the sense amplifier and write driver.

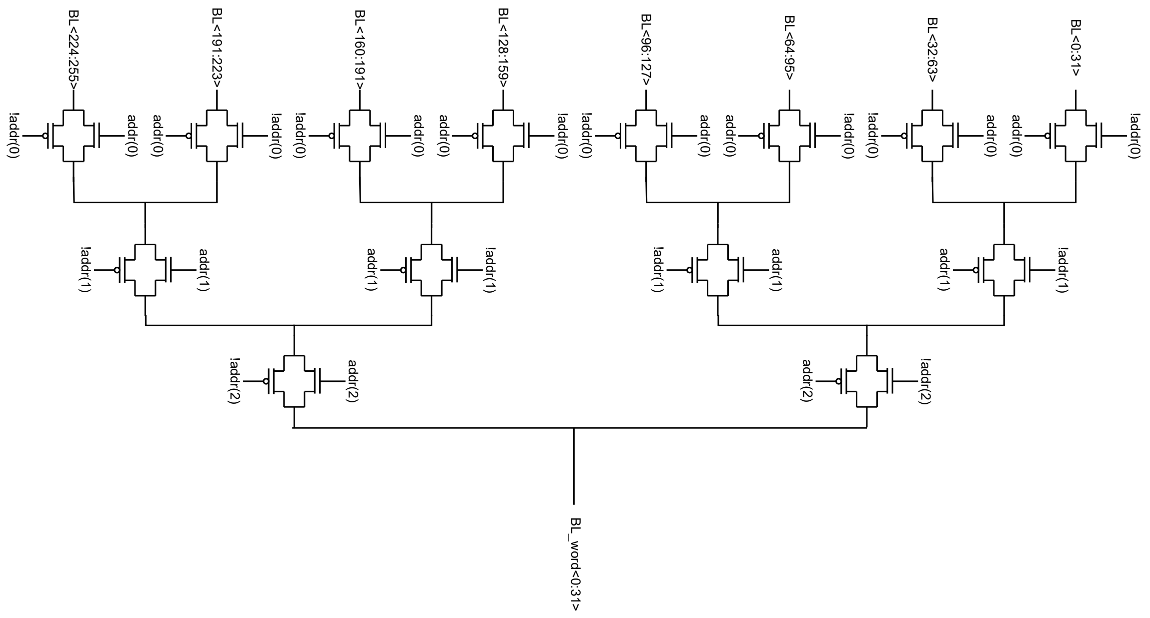


Figure 3.11: Schematic of the column decoder

3.3. Baseline ACA design

ACA works by offloading the address generation to the peripheries of the memory sub-array. By doing this, it significantly reduces the volume of data that needs to be transmitted through the memory bus, improving latency and energy consumption because of that. ACA is implemented by replacing the address decoders by two shift registers, one for the rows and one for the columns, as shown in figure 3.12.

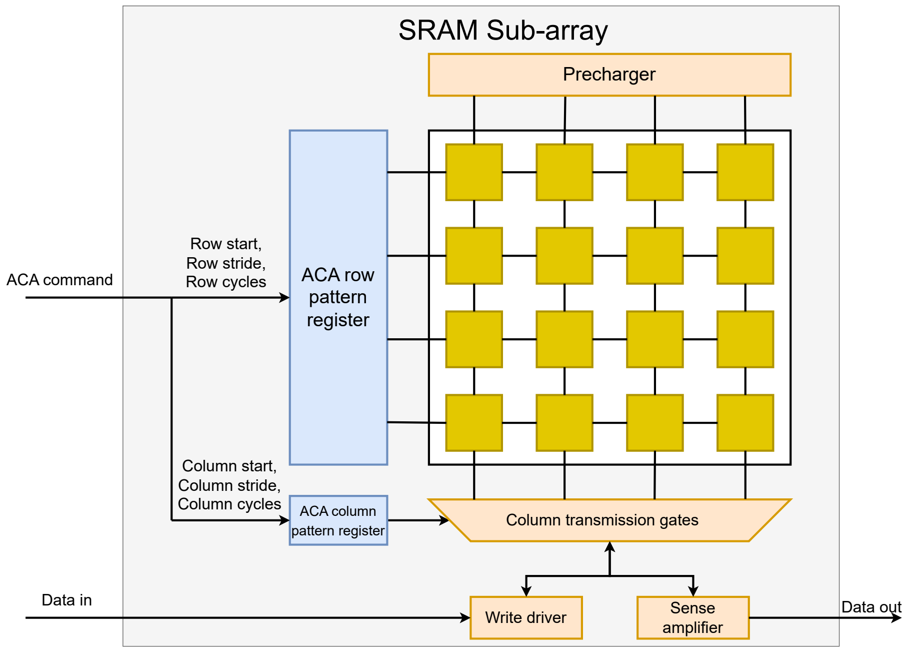

SRAM Sub-array
Precharger
ACA command
Row start, Row stride, Row cycles
ACA row pattern register
Column start, Column stride, Column cycles
ACA column pattern register
Column transmission gates
Data in
Write driver
Sense amplifier
Data out

Figure 3.12: Sub-array with ACA. The row decoder is replaced by a row shift register, and the column transmission gates are controlled by a column shift register

Figure 3.13 illustrates an example of how ACA works. The words highlighted in red in Figure 3.13 exhibit a distinct loop-like pattern. This pattern can be described using specific loop parameters: the columns start at 0, with a stride of 2, iterating 3 times; while the rows start at 1, with a stride of 4, iterating 2 times. To access the words highlighted in red, the column shift register is initialized with 100000 and the row shift register with 010000. The column shift register then begins shifting with a stride of 2. After completing 3 iterations, the row shift register is enabled, shifting with a stride of 4. Subsequently, the column shift register is re-initialized and begins shifting again. ACA allows for more flexibility in memory accesses than a regular burst access. It allows for a distinct stride and number of cycles for the rows and columns, which is not possible using simply burst accesses. This would lead to significant performance increases in image processing applications that display a 2D pattern, with efficient mapping of the data into the memory space.

In order to perform address generation, ACA first receives a compressed higher-level instructions sent to the sub-array. It can then decode this higher-level instruction and generate addresses based on it. In order to perform the address generation, ACA needs to receive 6 parameters: row start, row stride, row cycles, column start, column stride and column cycles. In the case of the example in figure 3.13, table 3.1 shows the value of these parameters.

Parameters	Row Start	Row Strides	Row Cycles	Column Start	Column Stride	Column Cycles
Value	1	4	2	1	2	3

Table 3.1: ACA parameters for the given example

Therefore, ACA receives an ACA command consisting of the values shown in table 3.1. From the ACA command, the row start and column start value are decoded and used to initialize the shift registers. The stride and number of cycles value are stored in latches and used to control the flow of

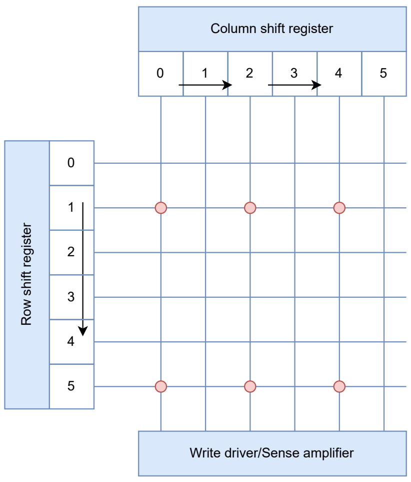


Figure 3.13: Example of ACA functionality

the address generation. Both shift registers have a corresponding counter that counts the number of elapsed clock cycles during the shifting operation, and compares this to the number of cycles received from the ACA command and stored in latches. Based on this comparison, the shift registers can resume shifting, or stop the shifting if address generation is complete.

Figure 3.14 provides a detailed view of the row shift register. For clarity, the figure omits the gate connections between the different registers that enable shifting with varying strides. The outputs of the shift register are connected to an AND gate together with the clock signal, ensuring that a word line is driven only when the clock is high, thus preventing activation during the precharge stage.

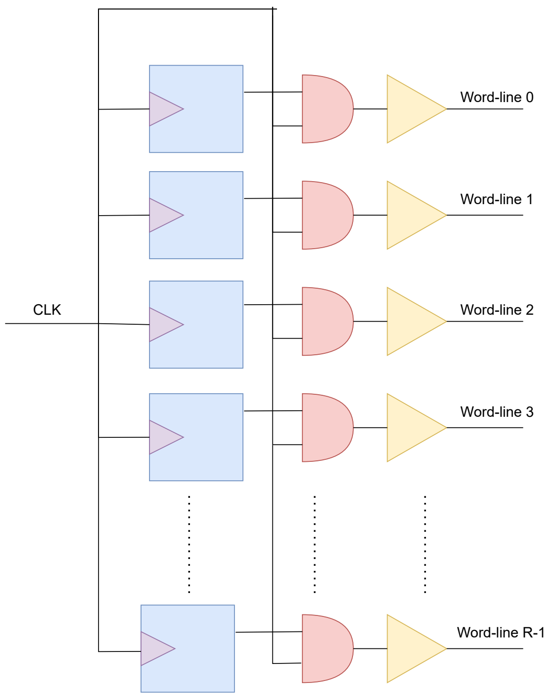


Figure 3.14: Baseline row shift register

3.3.1. Implementation aspects

The shift register in ACA can support strides longer than 1, to increase the flexibility of the address generation, making ACA more useful for applications that require accesses with strides longer than 1. There are two options to perform a shift operation with a stride longer than 1:

1. Break it down into shift operations with strides of 1, and provide the shift register with a clock having a higher frequency than the frequency of the rest of the sub-array. This would allow the shift register to perform multiple shifts of stride 1 in one clock cycle of the sub-array.

2. Implement logic between the flip-flops of the shift register to allow the shift operations with strides longer than 1 in one clock cycle

Option 1 would lead to more energy consumption, as it introduces unnecessary toggling of the intermediary flip-flops in a shift operation with a long stride. Also, having the shift operation take multiple clock cycles would significantly increase the dynamic power consumption in general, as the clock toggles multiple times to perform a single shift operation. And since energy consumption is the primary metric of comparison in this work, option 2 is picked instead.

The ACA shift registers are implemented together with logic to perform shift operations with a stride longer then 1. A design parameter is the set of possible strides that are implemented in both the horizontal and vertical directions. Two image processing benchmarks have been profiled and used to decide which stride length options to use:

Guided Filter The guided filter is an edge preserving smoothing image filter used in image processing [32]. For the guided filter benchmark, having a stride of 1 in both the horizontal and vertical directions covered 98% of the possibility of compression for memory accesses into ACA commands. This is because each thread accesses the address of a pixel and its neighbors.

Forward Fan-Beam Projection Forward Fan-Beam projection has more irregular data access strides. As shown in figure 3.15, during each step, the stride by which each thread accesses the pixels of an image changes. Strides 1, 2 and 4 covered 91% of the possible compression of address accesses into ACA commands. After that, adding a new stride only increases compression capability marginally. For example, adding a possible stride of 3 only increases the coverage from 91% to 92.5%.

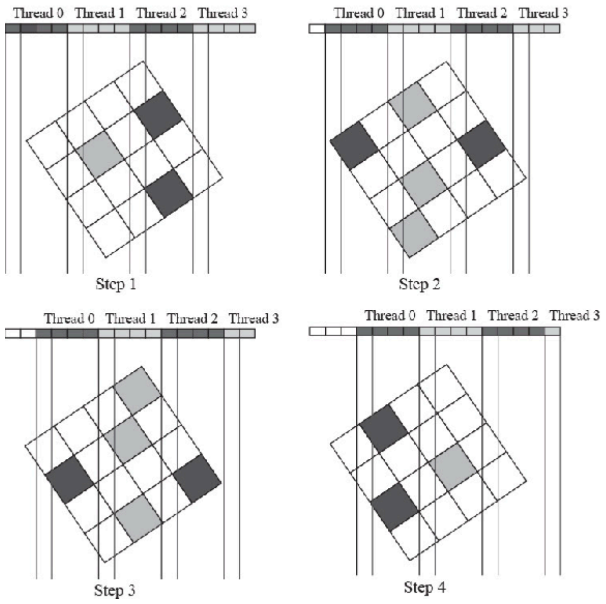


Figure 3.15: Irregularity of strides in forward fan-beam projection [33]

Chosen strides: In the exploration in this work, strides of: 1, 2 and 4 are chosen. This is because these strides offer broad coverage for the chosen benchmarks. So logic is implemented in both the row and column shift registers to allow performing shifting operations with these 3 options.

An SRAM crossbar of 256 columns by 256 rows is used, and a word size of 32 is used. This means that there are 8 words per row. Therefore, the row shift register includes 256 registers, while the column shift register includes 8 registers.

3.3.2. Limitations

The issue with this basic ACA design is the significant dynamic power consumption caused by connecting all the flip-flops to the clock. This power waste occurs mainly due to internal switching within the flip-flops, even when their states do not change.

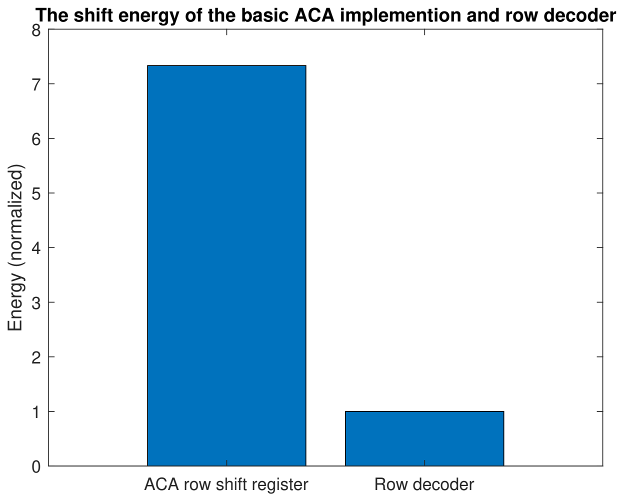


Figure 3.16: Shift energy of the basic row shift register of ACA implementation and of the row decoder

Figure 3.16 shows a comparison between the energy consumed by one shift operation in the basic ACA implementation of the row decoder, with the average energy consumed by the row decoder. One shifting operation consumes about 7.3 times as much more than the row decoder. The vast majority of this energy consumption comes from dynamic power consumption of the flip-flops. Therefore, in chapter 4, methods to reduce this dynamic power consumption for ACA are proposed.

4 Proposed ACA

In this chapter, improvements on the circuit design of ACA are proposed, built-upon, and compared. Section 4.1 presents an overview of the features of the proposed ACA design. Sections 4.2, 4.4, 4.5 and 4.6 present successive improvements to the circuit design of the basic row shift register. Section 4.7 presents the measured results obtained using various circuit designs. Based on certain metrics, one design and configuration are picked for the row shift register. Section 4.8 presents how the column shift register is improved, and section 4.9 presents how the control module of ACA is improved.

4.1. Overview of the proposed ACA features

As shown in section 3.3.2, one shift operation of the ACA row shift register consumed about 7.3x the average energy of the decoder. As the row shift register replaces the row ACA, implementing this design of ACA would lead to a significant increase of energy consumption on the sub-array level. Although ACA's primary goal is the reduction of communication over the memory bus in order to save latency and energy, this substantial increase of energy consumption on the sub-array level is not desirable, as it would reduce ACA's potential energy savings on the macro-level. Therefore, reducing the energy consumption of the ACA modules on the sub-array level is imperative.

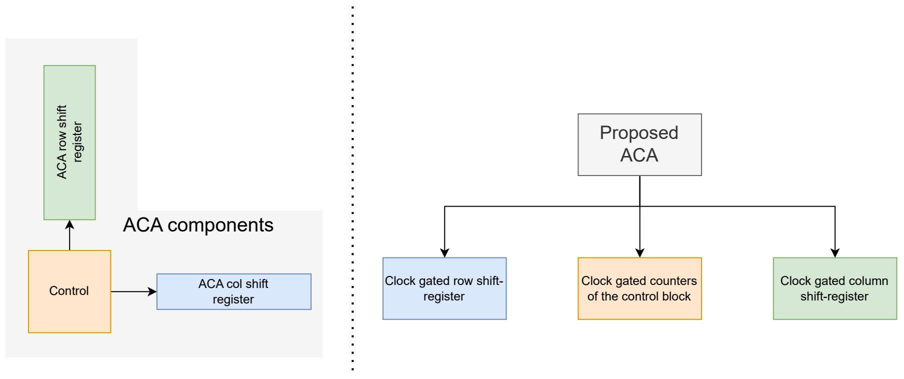

ACA row shift register
ACA components
Control
ACA col shift register
Proposed ACA
Clock gated row shift-register
Clock gated counters of the control block
Clock gated column shift-register

Figure 4.1: Proposed ACA design introduces clock gating for the row shift register, column shift register and control blocks

This work proposes a circuit design of the ACA modules in order to reduce the energy consumed during address generation. As shown in figure 4.1, the proposed ACA design introduces clock gating techniques to the row shift register, column shift register and control block.

As the row shift register is the largest block out of ACA with a large margin, this work starts by proposing different circuit designs of the ACA row shift register in order to minimize energy consumption. These designs of the shift register are built based on Partial Specific Clock Gating (PBSC) [34], and can be considered different flavours of PBSC that are application specific to ACA. These different flavours of PBSC are:

- **PBSC with XOR-based activity detection (PBSC-XOR-AD)**: This is presented in section 4.3, and is based on the work in [34].
- **Augmented PBSC with XOR-based activity detection (PBSC-XOR-AD+)**: This is presented in section 4.4: It builds on PBSC-XOR-AD by introducing an optimization to drastically reduce the energy consumption of the latches.
- **Augmented PBSC with OR-based activity detection (PBSC-OR-AD+)**: This is presented in section 4.5. It builds on PBSC-XOR-AD+ by replacing the XOR-based activity detection circuit, with OR-based activity detection circuits.
- **Augmented PBSC with shift detection (PBSC-SD+)**: This is presented in section 4.6. This circuits build on PBSC-XOR-AD+, but uses shift detection between different clock domains, rather than activity detection within one clock domain.

These different flavours of PBSC are presented in sections 4.3-4.6. The clock gating of the column shift register is presented in section 4.8. As for the counters in the control module, a different clock gating scheme is presented in section 4.9.

But before introducing these different designs, some background knowledge on clock gating techniques is presented in section 4.2.

4.2. Basics of clock gating

Power consumption in an integrated circuit (IC) can be categorized into two categories: Dynamic power consumption and static power consumption. Dynamic power is the power consumed from the transient switching phenomena of the IC. Static power consumption happens even when there is so switching present. In the case of the basic implementation of the ACA shift register, the vast majority of the power consumption is dynamic power consumption, as it is caused by the switching of the clock which is connected to 256 flip-flips, that causes unnecessary toggling inside them.

Clock gating is a technique meant to reduce the dynamic power consumption by reducing the toggles of the clock in the branches of the clock tree where it is not necessary to toggle, which is done by gating off the clock at these branches [35]. This is very relevant to the row shift register, as for extended periods of time, most flip-flops stay idle without changing their value. And so blocking the clock from these flip-flops would lead to a significant reduction of power consumption. Clock gating implementations can vary significantly. Figure 4.2 shows two examples of clock gating using an Enable signal, either connected to an AND gate or an OR gate.

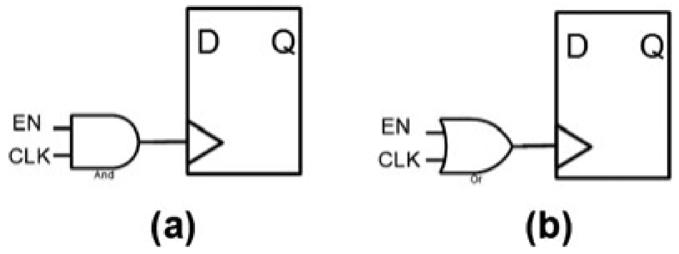


Figure 4.2: Latch-free clock gating using an (a) AND gate and an (b) OR gate [35]

Simply connecting the clock signal to a gate could lead to glitches and timing issues depending on the implementation. This is where latch-based clock gating can be used, which is illustrated in figure 4.3. As shown in the wave-forms, adding the latch mitigates timing issues even if the enable signal comes slightly before the clock edge or slightly after it.

4.3. PBSC-XOR-AD based row shift-register

Partial Bus-Specific Clock Gating, as introduced in [34], can be used to reduce dynamic power consumption. In this approach, flip-flops are divided into several clock domains, each with its own clock

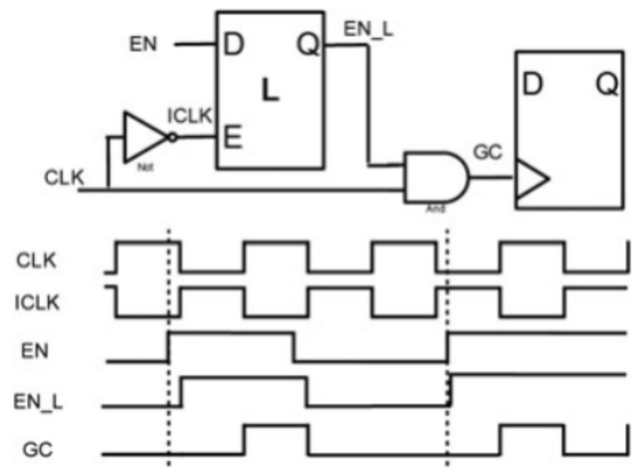


Figure 4.3: Latch-based clock gating[35]

gating circuitry, as show in figure 4.4. This setup blocks redundant clock pulses by enabling each clock domain's clock only when there is a difference between the input and output of any contained flip-flop, which detected using XOR gates connected to each flip-flop. Additionally, a latch in each clock region prevents glitches and timing issues by breaking the loop between a flip-flop's output and its clock.

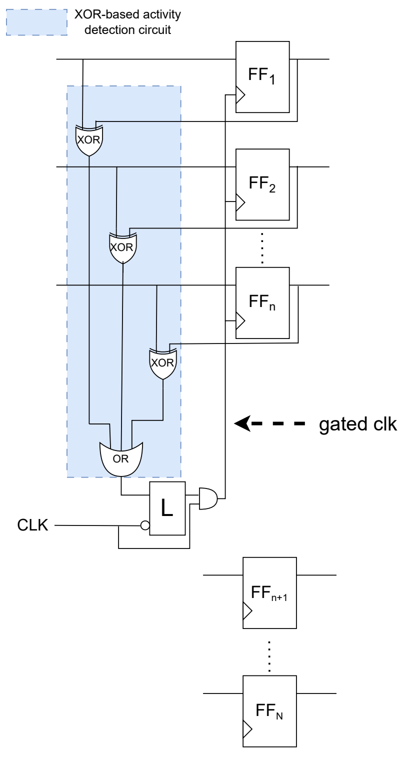


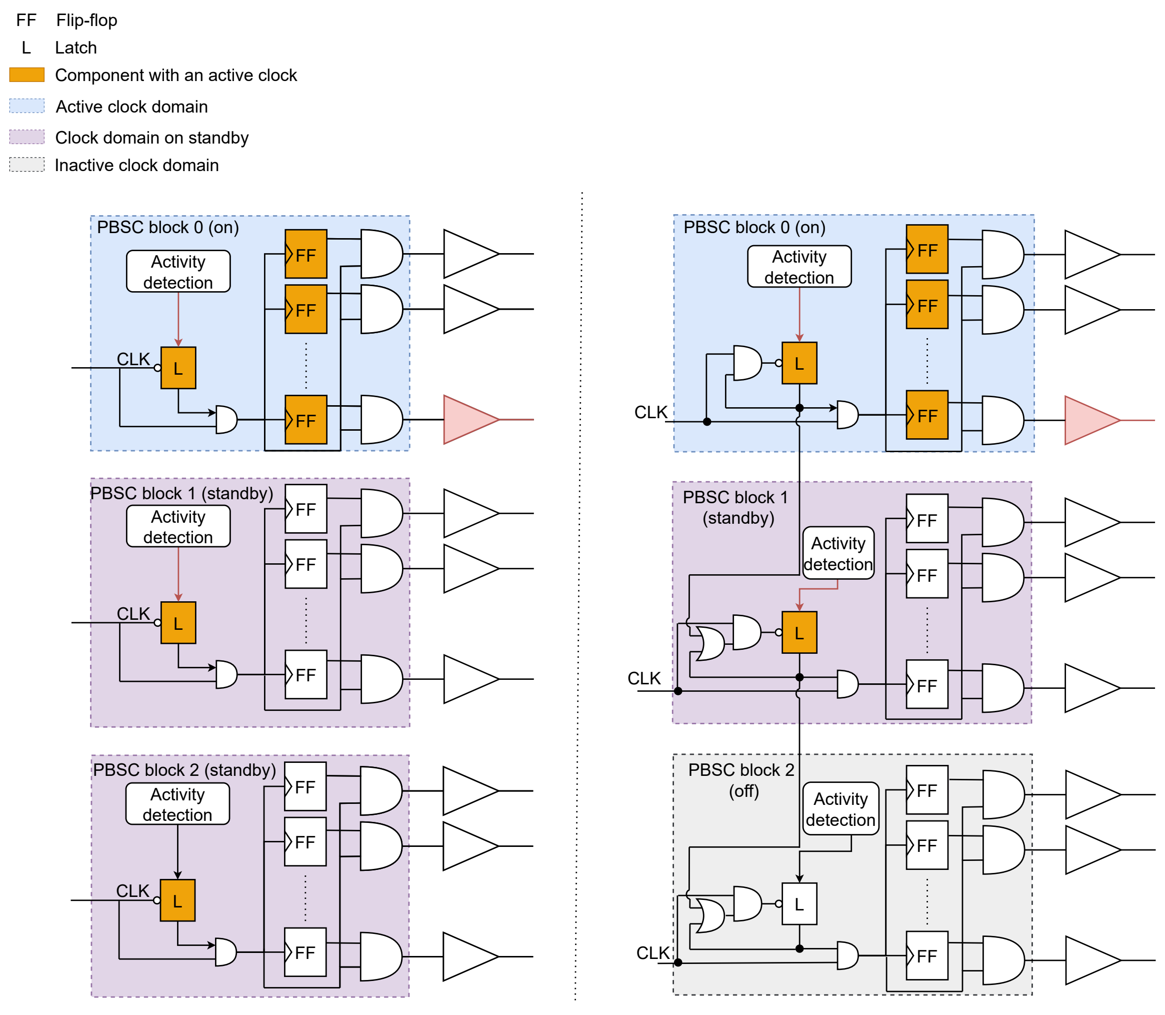
Figure 4.4: Partial Bus-Specific Clock Gating scheme [34]

The flip-flops need to be grouped into different clock domains based on the correlation of their activity [34]. This means that flip-flops that are likely to toggle together are placed in the same clock domain. By doing this, the number of clock domains that can be turned off at any given instance is maximized, leading to improved energy efficiency. In the case of a shift register, each flip-flop's activity is most correlated with its immediate neighbors. Consequently, neighboring flip-flops can be grouped into the same clock domain.

This grouping makes the number of clock domains in the implementation, and consequently the number of flip-flops in each clock domain, a design variable. An implementation with more clock domains is more fine-grained as the number of active flip-flops can be minimized by having the clock domains be smaller. It would however lead to an increase in area, as each clock domain comes with its own circuitry and latch.

4.4. PBSC-XOR-AD+ based row shift-register

The PBSC setup shown in figure 4.4 is general and can be used for any digital design with flip-flops. It suffers from the drawback that the latches of all the blocks are always driven by the clock, which causes dynamic power consumption. For example, in a configuration of 16 PBSC blocks, each containing 16 flip-flops: the clock is distributed to 16 latches, and to the 16 flip-flops of the active block (except when a shift is happened across blocks, then 32 flip-flops are active). As shown in figure 4.5(A), all the latches are active. The activity detection logic in figure 4.5 is the xor and or tree from figure 4.4.



(B) PBSC-XOR-AD (B) PBSC-XOR-AD+

Figure 4.5: (A): Standard XOR-based PBSC. (B): Augmented XOR-based PBSC for shift register

Having all the latches active all the time means that all the clock domains that have no activity are on standby waiting for an incoming cross clock domain shift. However, if the size of the clock domain (number of flip-flops in it) is larger than the maximum stride in the design, a cross clock domain shift can only come if the preceding clock domain is active. This means that if a clock domain is active, only the subsequent clock domain needs to be on standby, and all the other clock domains can be turned off. PBSC-XOR-AD+ makes use of this and is shown in figure 4.5(B). The added gates to the clock of the latches make the latches only active if its corresponding clock domain or the one preceding it (or both) are active. With this design, only 2 latches would be active at once in a configuration of 16 clock domains, each containing 16 flip-flop; in contrast with 16 using regular PBSC.

Even if the latches are negative clock level sensitive, their clock is gated so that it is low when the clock region is off, making the latches transparent by default. This removes the need to have separate logic to initialize the latches when the shift register is initialized, as the latch can be set after the flip-flops are initialized and activity is detected.

Compared to PBSC-XOR-AD, PBSC-XOR-AD+ is expected to lead to significantly less energy consumption for configurations using a large number of clock domains, as the inactive latches can be gated. However, this would come at the cost of adding more logic to gate the latches, are thefore more area overhead.

4.5. PBSC-OR-AD+ based row shift-register

The XOR gates in the activity detection circuit shown in Figure 4.4 are bulky and introduce significant area overhead. This circuit is designed to be general and applicable to any digital design with flip-flops. However, in this specific implementation for reading and writing from memory (not performing CIM or using a multi-row implementation), only one '1' value will propagate through the shift register. Therefore, activity does not need to be checked by comparing the input and output of the flip-flops. Instead, it can be simply checked by determining if any flip-flop holds a '1' value. This simplifies the activity detection circuit to an OR-reduction tree of the outputs of the flip-flops. The circuit design using this is called PBSC-OR-AD+ and is show in figure 4.6.

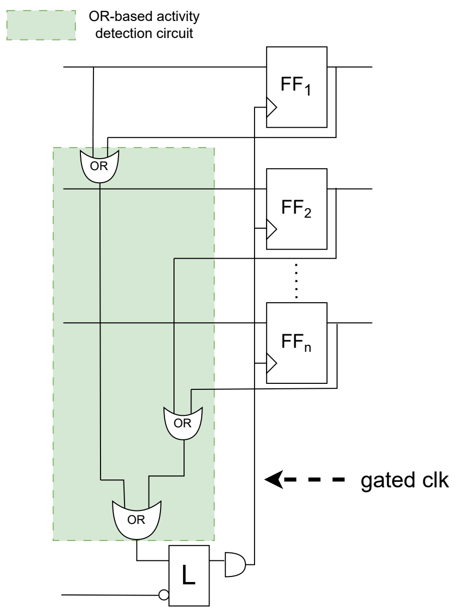


Figure 4.6: Clock gating using or-based activity detection

The total number of gates used in the or-based activity detection shown in figure 4.6 is less than the gates used in the xor gates activity detection shown in figure 4.4. In addition to this, XOR-gates are larger in size than OR-gates, which can be more efficiently implemented. This shows that the PBSC-OR-AD+ would lead to significantly less area usage than PBSC-XOR-AD+ based activity detection.

4.6. PBSC-SD+ based row shift-register

A drawback of using activity detection-based clock gating is that the activity detection circuitry toggles internally at every shift operation within a clock domain. In addition to that, the activity detection circuit can be large for a configuration with large clock domains, as the reduction tree would have a significant number of inputs. While this approach can be applied to any circuit with flip-flops, the activity of the row shift register is predictable, eliminating the need for a general approach. Assuming the size of the clock domain is larger than the maximum stride in the design, a clock domain only needs to be enabled if it is initialized with a '1' in one of its registers or if it receives a cross-clock domain shift from its preceding clock domain.

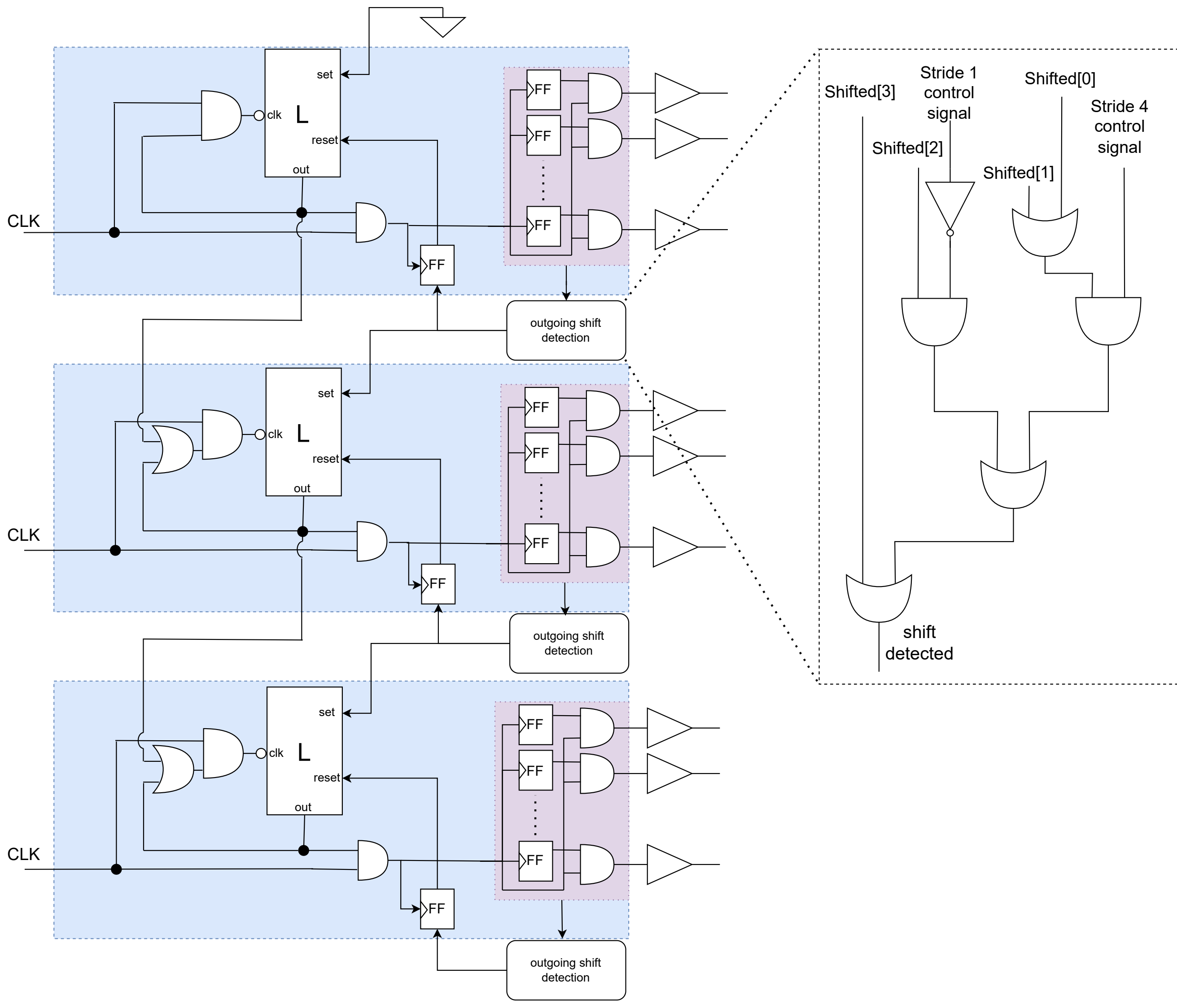


Figure 4.7: Row shift register with shift detection based clock gating

Figure 4.7 shows the design of PBSC-SD+, which is the design that utilizes shift detection instead of activity detection. The shift detection module identifies when a cross-clock domain shift happens. To reduce dynamic power consumption, the clocks of the latches are gated and are only enabled if the corresponding latch or the one before it holds a '1' value. In each clock domain, the latch is enabled, and the clock domain is activated when an incoming cross clock domain shift is detected. The latch is reset one clock cycle after an outgoing shifted '1' is detected, with an added flip-flop in each clock domain providing the necessary one clock cycle delay for the latch reset signal.

This approach eliminates the need for the activity detection module. The shift detection module only toggles when an cross-clock domain shift is detected, unlike the activity detection module, which toggles with every shift. This could result in energy savings. However, the added flip-flop to delay the reset signal of the latch causes additional dynamic energy consumption, which could potentially be higher than the savings from removing the toggling within the activity detection module. Additionally, this design requires the latch to be reset when the shift register is initialized. This is because the latches can not be set automatically like in the case of activity detection, as the shift detection logic only detects cross clock domain shifts. Therefore, logic must be added to each latch to enable resetting, and decoding circuitry is needed to find and set the appropriate latch during initialization. As shown in Figure 4.8, OR-reduction blocks are added to initialize the latches. This would lead to a larger area, and more energy consumption in the parallel load operation to initialize the shift register.

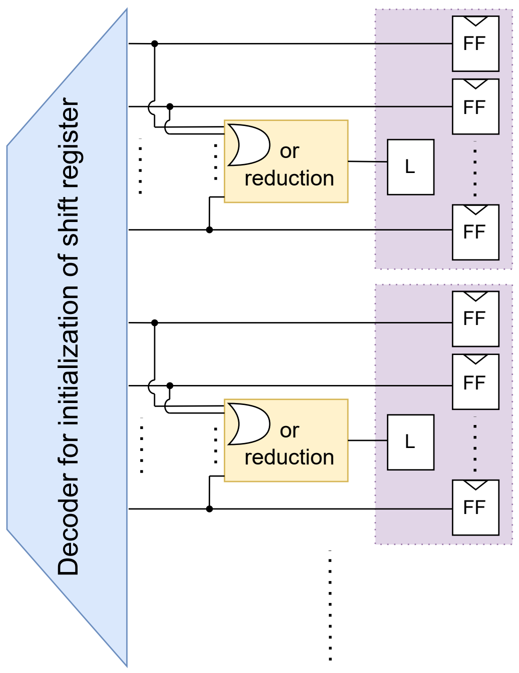


Figure 4.8: Added logic to initialize the latches in the shift detection scheme

4.7. Implementation and Simulation of row shift-register

In order to pick the best clock gating method and configuration (number of clock domains), the row shift register is implemented using the different clock gating techniques and configurations. Sections 4.7.3-4.7.3 present the results using the different flavours of PBSC. And section 4.7.4 presents a summary of the comparison between the different PBSC flavours.

4.7.1. Metrics

The different designs and all their different configurations (numbers of clock domains) are compared based on their area utilization, and on their energy consumption. When it comes to energy, two metrics are used: internal shift energy consumption, and average shift energy consumption. The former is measured, and the latter is calculated based on the former. The following is an explanation of each of these metrics:

Area: The different designs are also compared based on their introduced area overhead depending on the number of clock domains.

Internal shift energy consumption: Figure 4.9 illustrates the behavior of the clock domains during three operations: parallel load, internal shift, and cross clock-domain shift. When the shift register is initialized with a parallel load, all clock domains must be enabled to update the value of the shift registers. During an internal shift within a clock domain, only the corresponding clock domain is enabled, and the other flip-flops remain inactive. In contrast, during a cross clock-domain shift, both the originating and the receiving clock domains must be enabled.

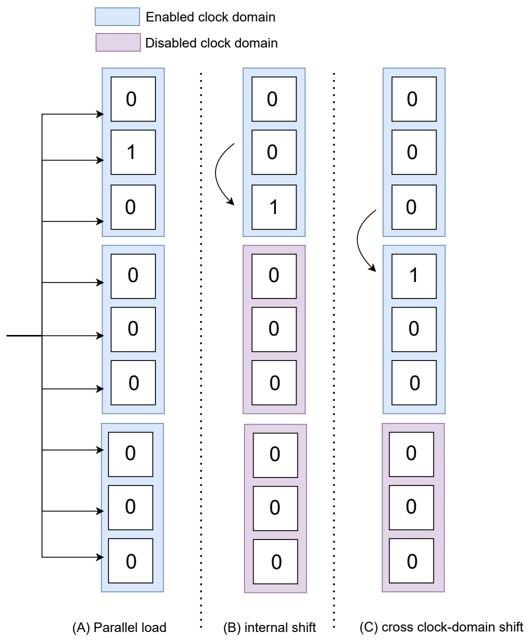


Figure 4.9: Behavior of clock domains in each operation (A): Parallel load. (B): shift internal to clock domain. (C): cross clock-domain shift

It becomes clear from this that the internal shift consumes the least energy of the three operations, as it involves the fewest active clock domains. This is followed by the cross clock-domain shift, which consumes approximately twice as much energy since it involves two active clock domains. Finally, the parallel load consumes the most energy, as it requires all clock domains to be active simultaneously. Initially, the internal shift energy is used as the metric to be measured from the simulation to compare the energy consumption of different configurations.

Average shift energy consumption Cross-clock domain shifts happen more frequently in configurations with small clock domains in comparison with configurations with larger clock domains. Figure 4.10 shows a comparison of the frequency of cross-clock domain shift operations between a configuration with a clock domain size of 4, and a clock domain size of 2, using a stride of 1. The frequency of cross-clock-domain shift operations in the configuration with a clock domain size of 4 is 1 in 4. For the configuration with a size of 2, it is 1 in 2.

As cross-clock-domain shift operations cost approximately twice as much energy as an internal shift operation. This higher likelihood for cross-clock-domain shift operations for configurations with smaller clock domains therefore needs to be analyzed on how it affects the comparison between the different configurations.

Given a stride length of $Stride$ and a configuration with a clock domain size of $Size$, the probability of a cross clock domain shift is:

$$P_{cross}(Stride, Size) = \frac{Stride}{Size} \tag{4.1}$$

and the probability of an internal shift (which is the complementary probability of the one in equation 4.1) is:

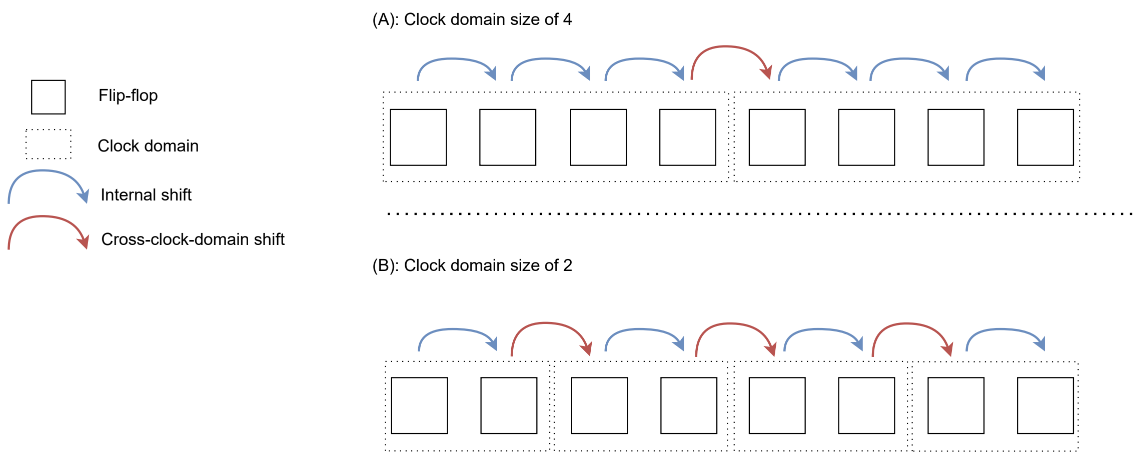


Figure 4.10: Cross-clock domain shift operation frequency comparison

$$P_{internal}(Stride, Size) = \frac{Size - Stride}{Size} \tag{4.2}$$

From these equations, it can be seen that the probability of an internal shift increases with the clock domain size increasing and/or the stride length decreasing, and the probability of a cross clock domain shift increases as the clock domain get smaller and/or stride length gets bigger.

Given a fixed stride, the average energy can then be expressed as a weighted average as:

$$E(stride) = P_{cross}(Stride, Size) \times E_{cross} + P_{internal}(Stride, Size) \times E_{internal} \tag{4.3}$$

By approximating that $E_{cross} = 2 \times E_{internal}$ and substituting P_{cross} and $P_{internal}$ with their formulas in equations 4.1 and 4.2, the following equation is obtained:

$$E(stride) = \frac{Stride}{Size} \times 2 \times E_{internal} + \frac{Size - Stride}{Size} \times E_{internal} \tag{4.4}$$

which can be simplified to

$$E(stride) = \frac{Stride + Size}{Size} \times E(internal) \tag{4.5}$$

Now that a formula for the average shifting energy per stride is obtained, The average shifting energy in general can be formulated according to the law of total probability, as follows:

$$E_{average} = P(stride = 1)E(1) + P(stride = 2)E(2) + P(stride = 4)E(4) \tag{4.6}$$

Where $P(stride = 1)$, $P(stride = 2)$ and $P(stride = 4)$ are the probabilities that the stride is equal to 1, 2 and 4 respectively.

These probabilities will differ per application and manner in which the data is mapped into the address space. However, for simplicity, we assume that these probabilities are equal to $1/3$ each, meaning that they are all as likely to happen.

By substituting the probability terms with their values:

$$E_{average} = \frac{1}{3}E(1) + \frac{1}{3}E(2) + \frac{1}{3}E(4) \tag{4.7}$$

Where the formulas for $E(1)$, $E(2)$ and $E(4)$ are given in equation 4.5.

This was, equation 4.7 can be applied on the measured internal shift energy consumption to calculate the average shifting energy.

4.7.2. Setup for implementation and simulation

In order to rapidly implement and simulate the different designs presented in this chapter, figure 4.13 shows the simulation and implementation flow. A python script is written to parse the netlist of the shift register depending on the design and number of clock domain. This netlist is then place-and-routed'd using Cadence Innovus and the layout can be obtained. The area utilization number can be then obtained from the layout. Quantus can then be used to perform parasitics extraction on the layout and parse the DSPF representation of the layout, which can be simulated using Cadence Spectre to obtain energy consumption numbers. The scenarios simulated are scenarios involving internal shift operations, in order to obtain internal shift operation energy.

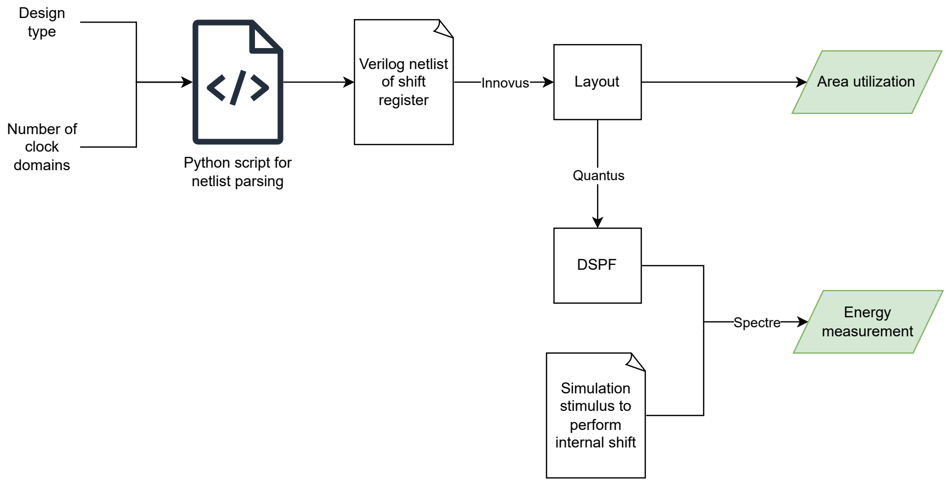

Design type
Number of clock domains
Python script for netlist parsing
Verilog netlist of shift register
Innovus
Layout
Area utilization
Quantus
DSPF
Spectre
Energy measurement
Simulation stimulus to perform internal shift

Figure 4.11: Flow to rapidly implement and simulate the different shift register designs

4.7.3. Results

PBSC-XOR-AD results

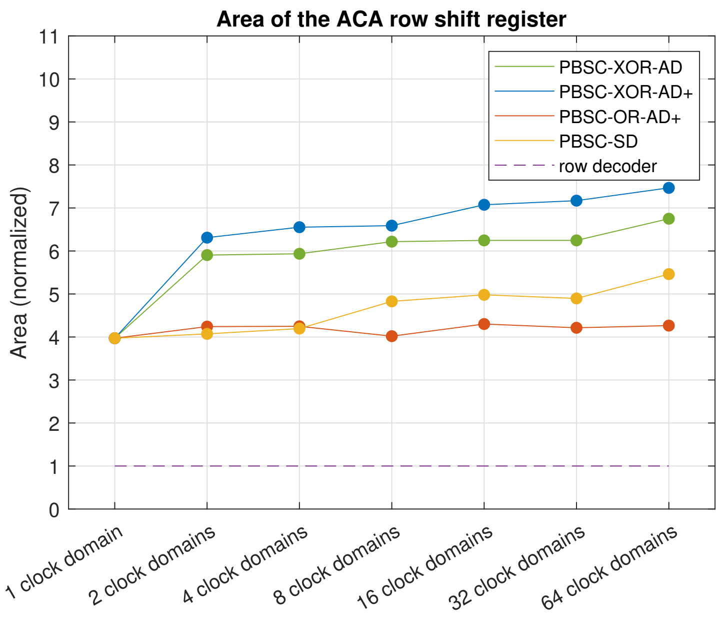


Figure 4.12: Area of the ACA row shift register

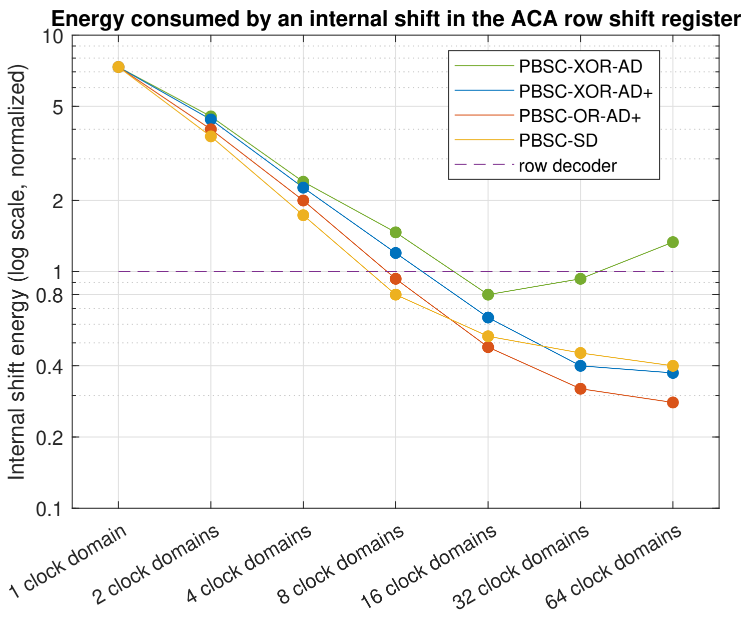


Figure 4.13: Internal shifting energy of the ACA row shift register

Figure 4.12 shows the area results, including of the row shift register with PBSC-XOR-AD. The configuration with a single clock region refers to the ACA row shift register without any clock gating. As illustrated in the figure, increasing the number of clock regions increases the area due to the addition of more latches and activity detection circuits. And implementation with 2 clock domains leads to an area overhead of about 48%, and an implementation with 64 clock domains leads to an overhead of 70%. This area overhead is significant and is caused by how big the XOR-based activity detection circuit is, which has been the motivation to explore PBSC-OR-AD+ in this chapter.

Figure 4.13 shows the energy consumed by all the designs, including by PBSC-XOR-AD. of an internal shift using different configurations of the ACA row shift register. The y-axis is presented on a logarithmic scale to better illustrate the differences. As the number of clock regions increases from 1 to 16, the energy consumed by an internal shift using PBSC-XOR-AD decreases significantly. Specifically, the energy consumption is reduced by approximately a factor of 2 each time the number of clock regions is doubled. This reduction occurs because increasing the number of clock regions decreases the number of registers in each clock region, thereby reducing the active components during an internal shift.

However, beyond 16 clock regions, the energy consumed by an internal shift using PBSC-XOR-AD begins to increase rather than decrease. This rise in energy consumption is due to the additional latches introduced which increase with the number of clock domains. These additional latches contribute to dynamic power consumption, offsetting the benefits of having more clock regions. This was the motivation to explore PBSC-XOR-AD+.

Figure 4.14 shoes the average energy consumption of shift operation. The data for the PBSC-XOR-AD is not shown in figure 4.14. This is because the shifting energy in it for configuration with small clock domains is dominated by the dynamic power consumption of all the latches, and therefore the approximation that $E_{cross} = 2 \times E_{internal}$ does not hold, making equation 4.7 invalid. Average shift energy consumption is computed however for the other flavours of PBSC.

PBSC-XOR-AD+ results

As shown in figure 4.12, PBSC-XOR-AD+ comes at an non-negligible area overhead of 5%-20% with reference to PBSC-XOR-AD. In general, this area overhead increases with more clock domains as more latches are implemented, each with its own gating logic.

The energy consumption of an internal shift operation using PBSC-XOR-AD+ is shown in figure 4.13 using multiple configurations with different numbers of clock domains. As the number of clock domains increases, the energy savings by PBSC-XOR-AD+ compared to PBSC-XOR-AD become clear. That's because the number of latches increases, and gating their clock saves a significant amount of energy.

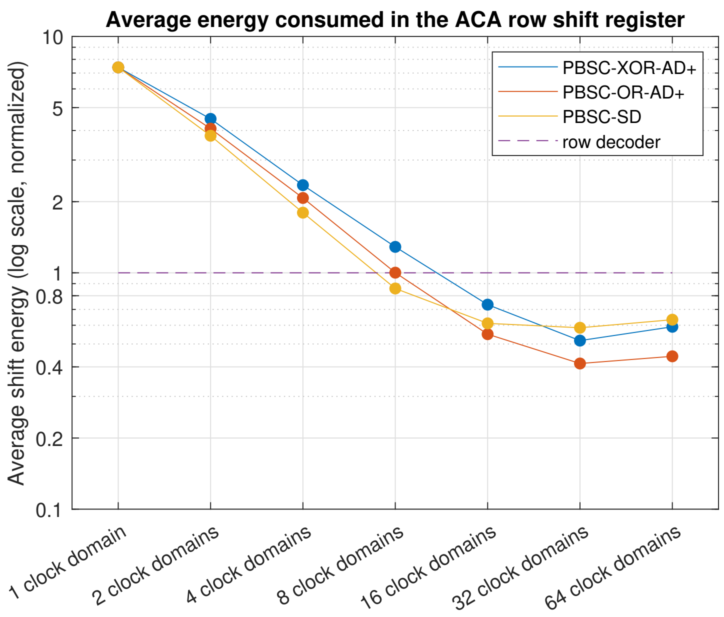


Figure 4.14: Average shifting energy

Now by applying formula 4.7 on the data in figure 4.13, the data shown in figure 4.14 is obtained. Looking at figure 4.14 and 4.13 shows a negligible difference between the average shifting energy, and the internal shifting energy shown for configurations with less clock domains (and therefore larger clock domains. However, this difference is not negligible for configurations with 32 clock domains or 64 clock domains. As shown in figure 4.14, the PBSC-XOR-AD+ configuration with 64 clock domains consumes more energy on average in shifting operations than the one with 32 clock domains.

PBSC-OR-AD+ results

Figure 4.12 shows the area overhead using the PBSC-OR-AD+ scheme. And indeed as intended, the or-based activity detection clock gating scheme leads to far less area utilization compared to its XOR-based counterparts. In fact, compared the baseline shift register with no clock gating implemented, the or-based activity detection clock gating scheme leads to a maximum area overhead of approximately 10%.

Figure 4.13 shows the energy consumed by an internal shift. PBSC-OR-AD+ leads to noticeably less energy consumption compared to PBSC-XOR-AD+ and PBSC-XOR-AD. In fact, it leads to a reduction of up-to 96% in energy consumption of the internal shift, compared to the baseline shift-register with no clock-gating.

Figure 4.14 shows the average shift operation energy consumption of PBSC-OR-AD+. It follows a similar trend to the internal shift energy consumption, except that the configuration with 64 clock domains consumes more than the one with 32 clock domains.

PBSC-SD+ results

Figure 4.12 show the area numbers of the implementation using PBSC-SD+. Shift detection leads to some gains in area compared to PBSC-OR-AD+ for configurations with a less clock domains (meaning larger clock domains). This is because for a large clock domain, the activity detection circuit which consists of an OR-reduction tree is bulky. However, for configurations with more clock domains, PBSC-SD+ consumes more area than PBSC-OR-AD+. This is because each clock domain has an added flip-flop to delay the reset signal of the latch, and those flip flops take-up more area in implementations with many clock domains.

Figure 4.13 shows the energy consumed by an internal shift using PBSC-SD+ in comparison with the other PBSC flavours. PBSC-SD+ is the most energy efficient for implementations with 8 clock domains or less. However, for configurations with more clock domains, the rate of reduction of energy consumption of PBSC-SD+ scheme degrades rapidly, due to the dynamic power consumption of the added flip-flop to delay the reset signal.

Figure 4.14 shows the average shift operation energy consumption of PBSC-SD+ compared to all the other flavours of PBSC. The average shift energy reaches its optimum at 16 clock domains, and increase with the addition of more clock domains.

4.7.4. Comparison and analysis

Comparison

Table 4.1: Summary of the comparison between the different flavors of PBSC

	Average shift energy consumption		Area overhead	
	Less clock domains (coarse-grained)	More clock domains (fine-grained)	Less clock domains (coarse-grained)	More clock domains (fine-grained)
PBSC-XOR-AD	=3rd best	worst	3rd best	3rd best
PBSC-XOR-AD+	=3rd best	2nd best	worst	worst
PBSC-OR-AD+	2nd best	best	2nd best	best
PBSC-SD+	best	3rd best	best	2nd best

This chapter proposes various improvements to the circuit-level design of the ACA row-shift register, in order to reduce its dynamic energy consumption. These improvements are implemented using various numbers of clock domains, and the impact on area and energy consumed by a shift operation internal to a clock domain is measured. Table 4.1 provides a summary of the comparison between the different flavors of PBSC. For configurations with less clock domains, PBSC-SD+ is the best in both metrics, while PBSC-OR-AD+ is the best for configurations with more clock domains.

From figure 4.14, it can also be seen that the design with **PBSC-OR-AD+ and 32 clock domains is the optimum in terms of average shift energy consumption**. And since the OR-based activity detection scheme adds the smallest area overhead as shown in figure 4.12, this configuration is the one picked for the row shift register.

Analysis of picked configuration

While clock gating techniques help reduce the dynamic power consumption during the shifting operations, they can not reduce the energy consumed during the parallel load operations. This is because all the flip-flops need to have an active clock during them in order to be initialized to the proper value. For the chosen OR-based activity detection configuration with 32 clock domains, the parallel load operation consumes about 8.15 times as much as the average energy consumed by the row decoder.

Figure 4.15 shows a comparison between the energy consumed by the ACA row shift register and the row decoder depending on the number of addresses decoded/generated. After the first address, ACA consumes much more than the row decoder due to how energy consuming the parallel load operation is. After that, the energy consumed by the proposed ACA using clock gating increases slower than the row decoder. This is because the configuration chosen consumed about 40% the energy of the decoder as show in figure 4.14.

Starting from the 14th address, the ACA row shift register starts to consume less than the row decoder. On the other hand, the energy consumed by the baseline ACA row shift register keeps going up always in a faster rate than the row decoder.

It should be noted however that this analysis ignores the energy required to drive the row decoder using the memory bus, which falls outside the scope of this work. Due to how energy expensive driving the memory bus is [15], this analysis would look significantly different with it taken into account, and the ACA row shift register would cost less energy than the row decoder far earlier than 12 addresses.

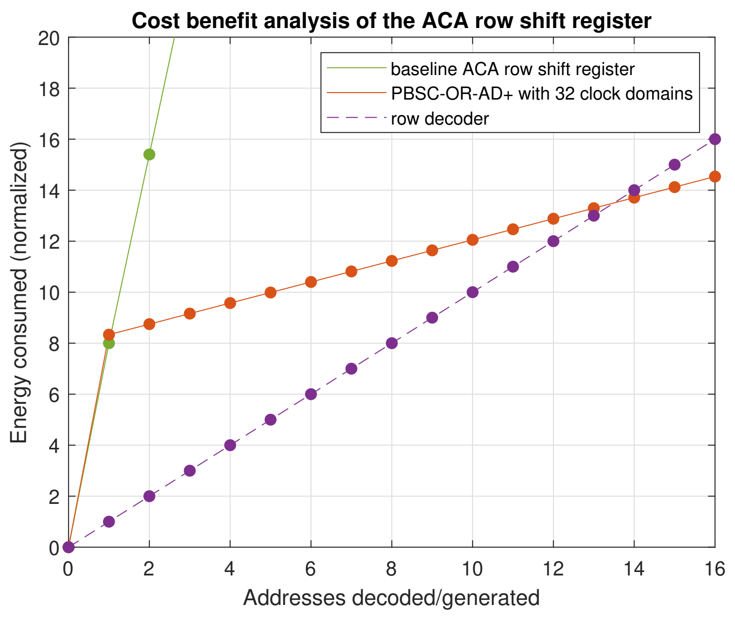


Figure 4.15: Cost benefit analysis in terms of energy consumption of the ACA row shift register

4.8. Column shift-register

The column shift register has 8 registers. Due to the limited design space with 8 registers, the optimal configuration in terms of energy consumption was found based on experimentation and is a configuration with 4 clock domains, each with 2 registers, and using OR-based activity detection.

When the column shift register is added to the sub-array, the two last stages of the column decoder shown in figure 3.11 are removed, and the shift register provides the 8 control signals to the 8 blocks of transmissions gates in the first stage. This is illustrated in figure 4.16. This removal of 2 stages of the column multiplexer is expected to decrease its energy consumption.

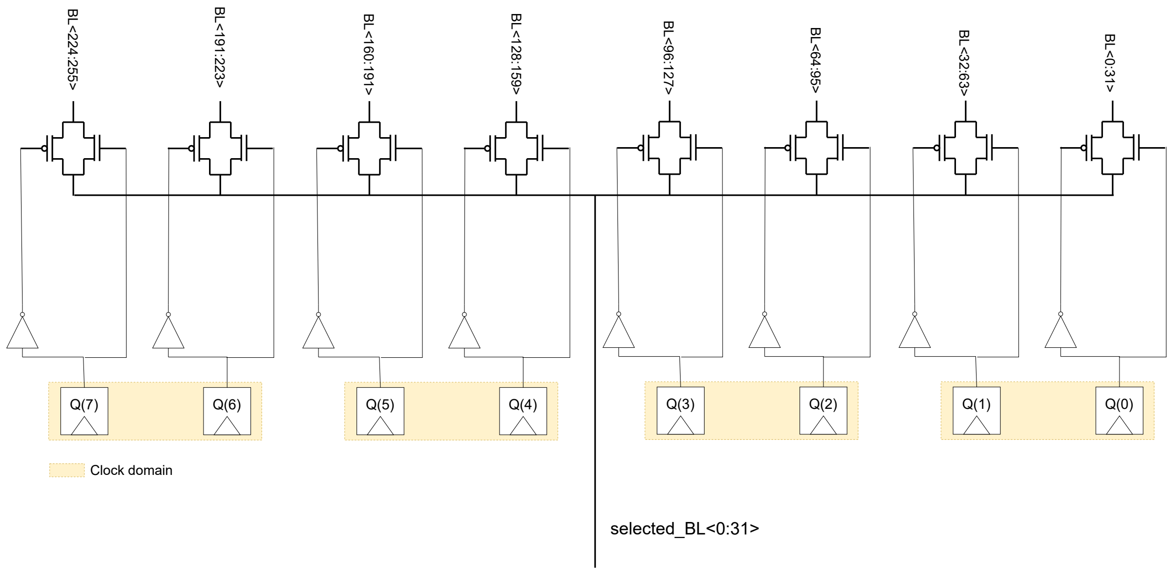


Figure 4.16: Column shift register controlling the transmission gates

4.9. Control module

The control block of the row shift register is shown in figure 4.17. The encoded stride, and the number of cycles are stored in latches. The encoded stride is decoded to generate the control signals specifying which stride distance needs to be used.

A counter is also implemented in order to count the number of shifts performed and compare it to the stored number of cycles aimed for. For the row decoder, the counter has 8 bits. It is separated into 3 clock regions. The two LSBs are not gated, as they are always toggling, the next 3 bits are gated based on the content of the 2 LSBs, and the 3 MSBs are gated based on the content of the 5 LSBs. This clock gating scheme is presented in [36].

The counter of the column shift register, has 3 bits. It is separated into 3 clock domains: the 2 LSBs are not gated, and the 1 MSB is gated.

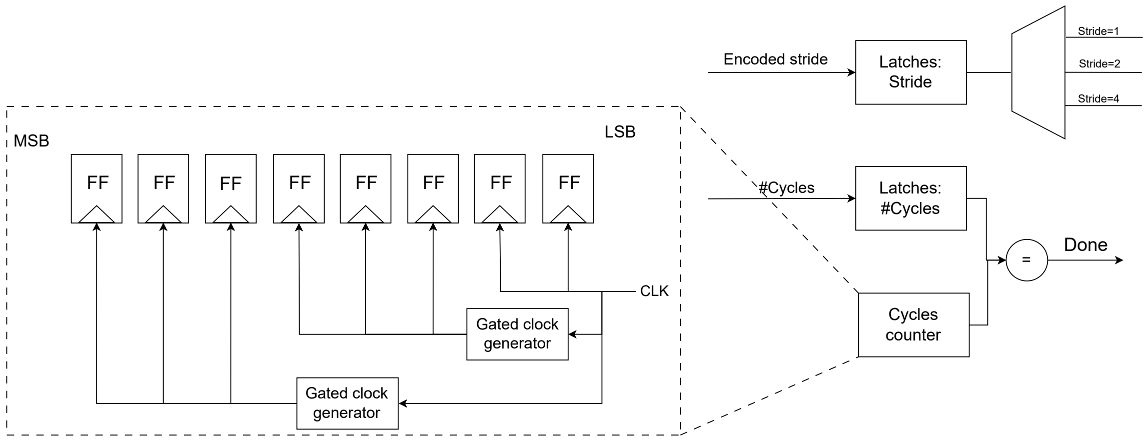


Figure 4.17: Control block of the row shift register

4.10. Conclusion

The ACA module has been implemented for a sub-array with 256 rows, 256 column, with a word size of 32 bits. An improved circuit design of ACA has been design and implemented using the following configuration:

- As the ACA row shift register is large (256 flip-flops), various clock gating schemes have been explored, and it has been shown that a row shift register using PBSC-OR-AD+ with 32 clock domains each containing 8 flip-flops is optimal in terms of energy (its average shifting operation energy is 40% of the average energy consumption of the row decoder), and has close to minimal area overhead (about 5% increase in area compared to the row shift register with no clock gating).
- The ACA column shift register has 8 flip-flops, and an implementation using PBSC-OR-AD+ with 4 clock domains each with 2 flip-flops has been chosen for similar reasons.
- The counters have also been gated using latch-based grouping bits clock gating. The counter of the row shift register has been gated into 3 groups (clock domains), and the counter of the column shift register has been gated into 2 groups.

5 Integration of ACA in sub-array

In this chapter, the sub-arrays with and without ACA are implemented and simulated in various scenarios and compared in terms of energy consumption. Baseline ACA refers to the basic implementation of ACA, and proposed ACA refers to the implementation of ACA in the chosen improved configuration with clock gating.

5.1. Simulation setup

Now that the ACA components have been designed and tested separately from the rest of the sub-array, they are simulated together with the rest of the sub-array in order to obtain the energy consumption numbers, and obtain a breakdown of the energy consumed by each element of the sub-array.

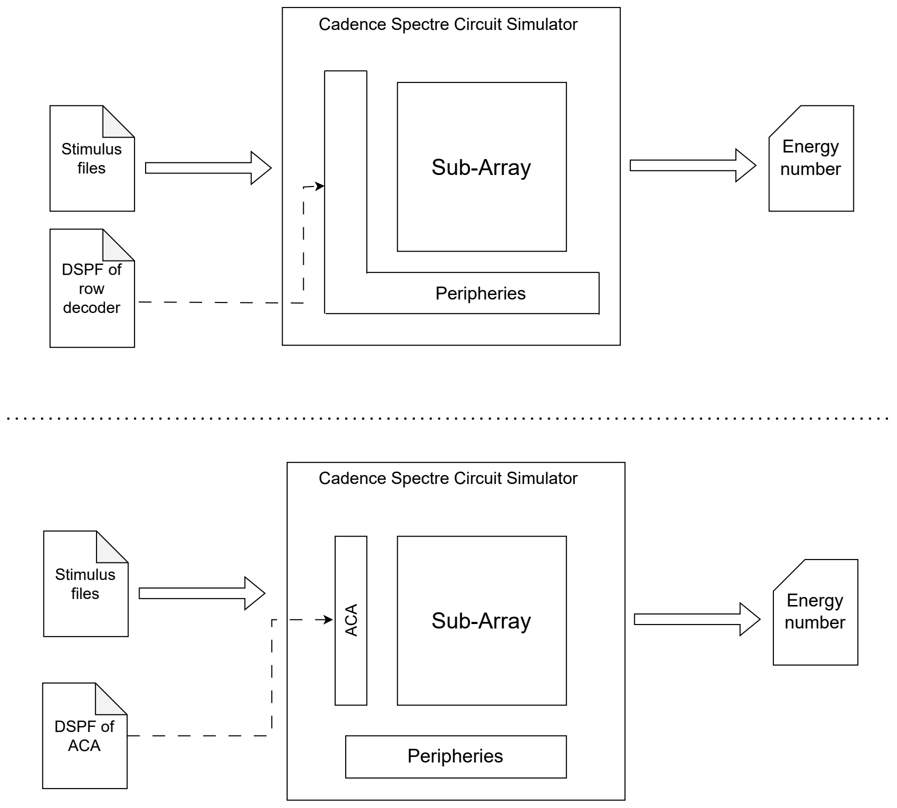


Figure 5.1: Spectre simulation setup

Figure 5.1 shows the flow used in order to simulate both the standard sub-array, as well as the sub-array together with the ACA modules. Both the baseline ACA and proposed ACA are implemented and simulated. The netlist of the macro to be simulated is described and simulated in the Cadence Spectre Circuit Simulator. Stimulus files are also provided to the simulator which describe how the inputs of the macro are driven, which allows performing any functionality test or simulating any desired scenario.

These stimulus files can either be written manually or generated by a Python script, depending on the complexity of the chosen scenario to simulate.

After the simulation completes, functionality can be monitored and energy numbers can be extracted and broken down by component.

5.2. Experiments performed

5.2.1. March algorithm

The first test performed is a march algorithm [37] through the whole memory sub-array. During the first iteration, a random 32 bit word is written to each word in the sub-array in order. During the second iteration, all the words in the sub-array are read in order. The stimulus files for this experiment are generated from a python script.

5.2.2. Guided filter GPU thread

The guided filter algorithm is briefly introduced in section 3.3.1. Figure 5.2 shows the flow used to generate the stimulus flow to simulate a scenario emulating the memory accessed of GPU thread running the Guided filter CUDA code.

First, the code of the algorithm is executed on an open-source GPU simulator called GPGPU-sim [38]. The code of GPGPU-sim is modified in order to make it print out a log of data and addressees in the accesses to memory by all the threads. After that, one thread is picked, and its memory accesses are processed in order to be converted into ACA instructions by an ACA ”compiler” written in Python. The file containing all the ACA command can then simply be processed by a written Python script to convert it into a Cadence Spectre stimulus file that determines the values of the input of the simulated system with ACA. For the sub-array without ACA, the memory traces of the chosen thread are what is converted into a Cadence Spectre stimulus file.

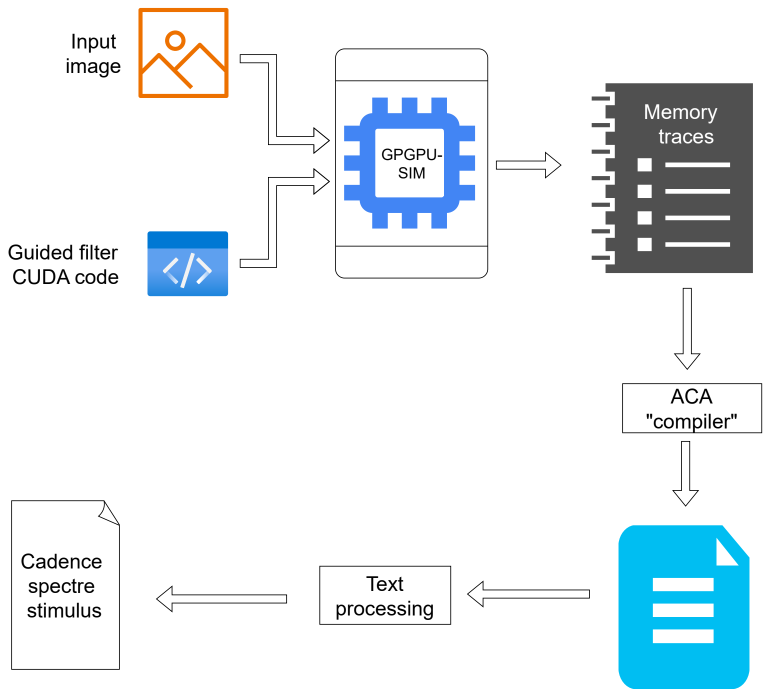


Figure 5.2: Guided filter GPU thread simulation methodology

5.2.3. Forward fan-beam projection GPU thread

The forward Fan-beam projection scenario was briefly introduced in section 3.3.1, and is also simulated similarly to the Guided filter GPU thread.

5.3. Results

5.3.1. March Algorithm

Figure 5.3 shows the results of running the March algorithm on the sub-array without ACA. The energy is broken down by component of the sub-array, and separated between the read and write. In the march algorithm, there are as many read operations as write operations, so they both contribute with the same ratio to the total. The write operations consume about twice as much energy as the read operations due to the larger currents involved in driving the bit-lines to write in the bit-cells.

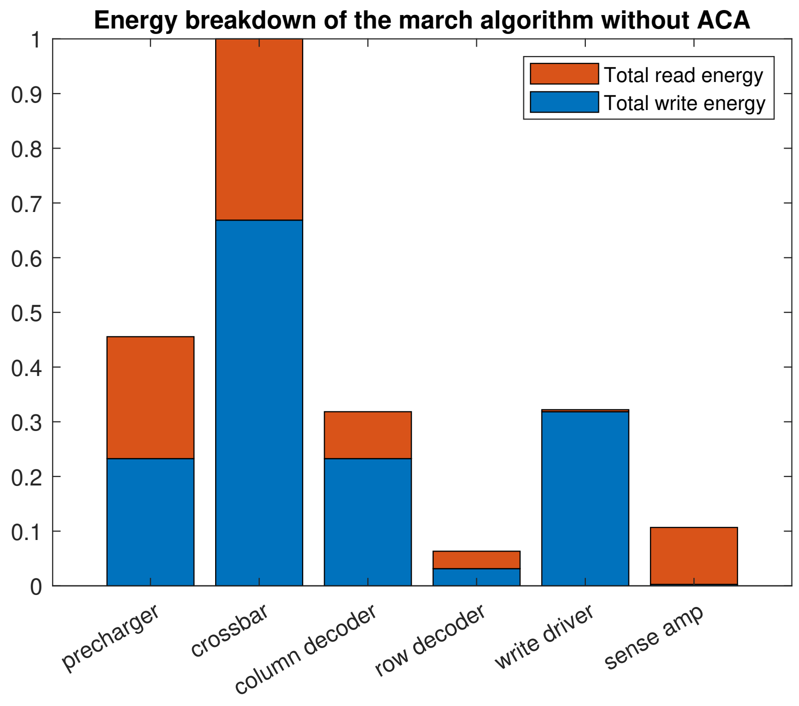


Figure 5.3: Results of the March algorithm without ACA

I can be also noted that the row decoder does not consume a significant amount of energy in this scenario. This is because of the way the March algorithm is run. For every row address, 8 column addresses are provided before the algorithm moves to the next row address. This means that the row decoder only decodes a new address every 8th address, which explains its small energy consumption. The column decoder also consumes a considerable amount of energy, especially during write operations.

Figure 5.4 shows the energy breakdown from executing the March algorithm scenario on the sub-array with baseline ACA. ACA consumes about 7 times the energy of the removed row decoder, and the column decoder energy decreases by about 30%. This is because of the removal of the 2 later stages of the column decoder tree shown in figure 3.11.

Figure 5.5 shows the energy breakdown after running the March Algorithm on the sub-array with the proposed ACA implemented. When using the proposed ACA, the column decoder consumes about 30% less energy. ACA consumes about 40% the total energy of the removed row decoder.

Figure 5.6 shows a comparison between the total energy consumed by the sub-array with ACA and without it. Using proposed ACA saves about 5% in this scenario, while using baseline ACA leads to an increase of 13%. These small savings are because of the nature of the scenario simulated, in which the row address does not change frequently.

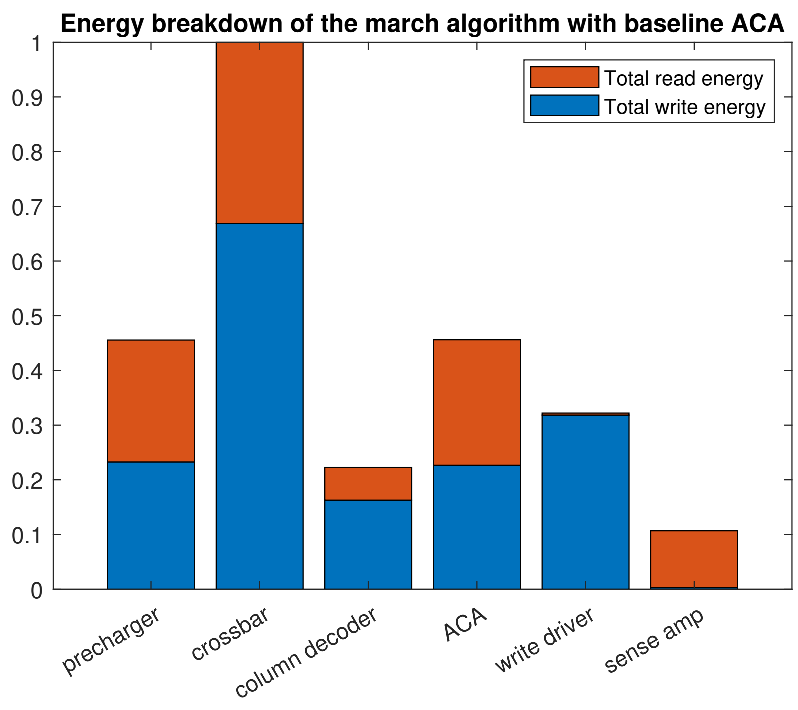


Figure 5.4: Results of the March algorithm with the baseline ACA

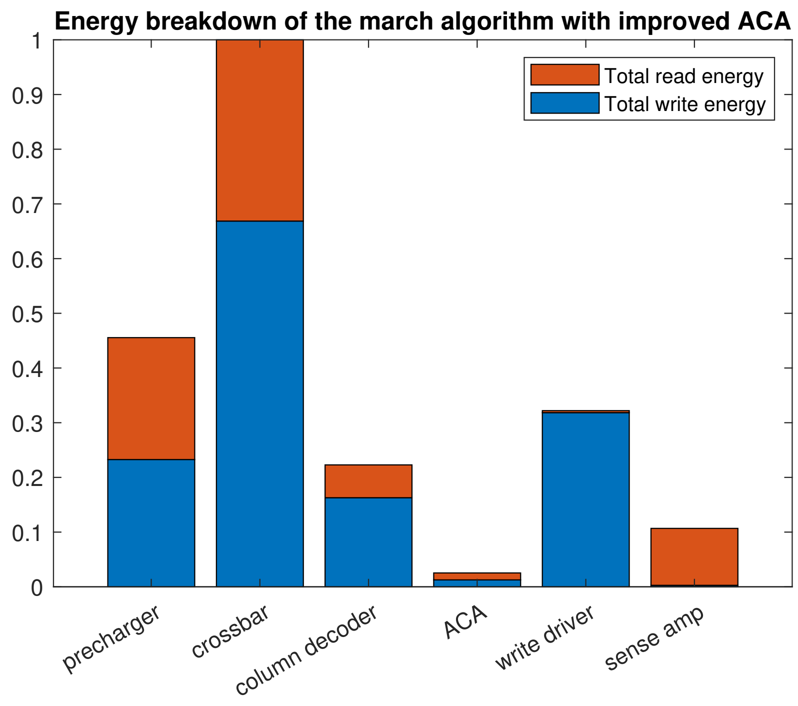


Figure 5.5: Results of the March algorithm with the proposed ACA

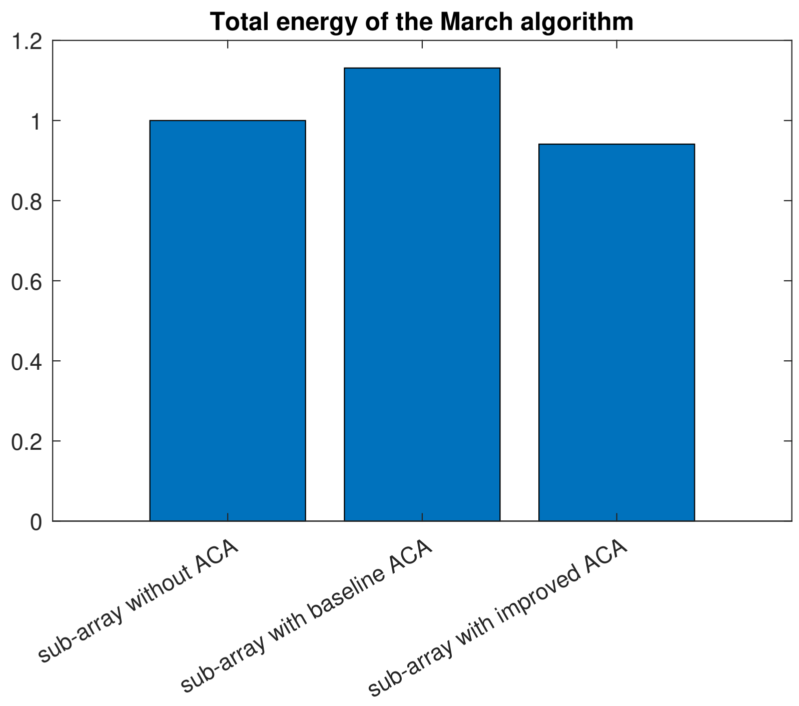


Figure 5.6: Total energy of the March algorithm

5.3.2. Guided Filter

Figure 5.7 shows the breakdown of the energy consumed by the sub-array component without ACA in the Guided Filter Scenario. In this scenario, there have been only read operations performed by the thread, which is why the energy consumed by the write operations is zero. The row decoder consumed now about 7 times as much energy as in the March algorithm scenario. This is because the row addresses change more frequently in the guided filter scenario.

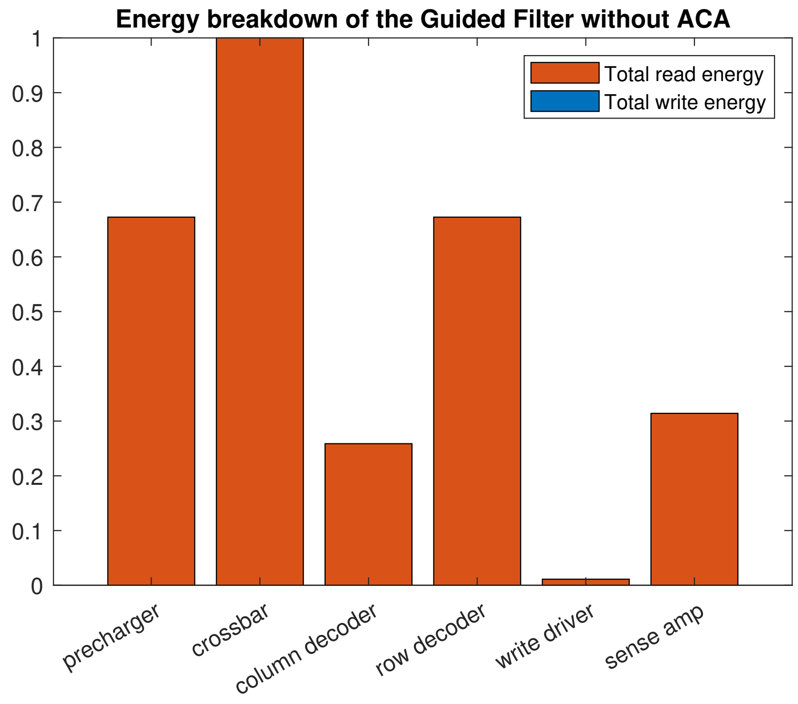


Figure 5.7: Energy results of the Guided Filter without ACA

Figure 5.8 shows the energy breakdown using the sub-array with the baseline ACA. Baseline ACA consumed about 7 times the energy of the removed row decoder, and the column decoder consumed approximately 73% less energy (Figure 5.8 has a larger range in the y-axis than figure 5.7).

Figure 5.9 shows the energy consumed by the components of the sub-array using proposed ACA. proposed ACA consumed about 65% the energy of the row decoder. However, in the march algorithm example, it consumed about 40% the energy of the row decoder. This difference comes from the analysis of the relationship of the number of cycles in the ACA operation with the energy savings, discussion in section 4.7.4. While the March algorithm scenario had 256 cycles for the ACA row shift register instruction, the average number of cycles for the row shift register in the Guided Filter scenario has merely been 27.

The column decoder implemented with only one stage of transmission gates in the implementation using ACA has also consumed about 70% of its energy consumption in the implementation without ACA.

Figure 5.10 shows a comparison of the total energy savings by using ACA in the Guided Filter scenario. Proposed ACA has shown a reduction of about 9% in terms of energy consumed, while baseline ACA has shown an increase of 38% in terms of energy.

5.3.3. Forward Fan-Beam projection

Forward Fan-Beam projection has shown relatively similar pattern of results to the Guided Filter, with slightly less energy savings. Due to the limited size of the sub-array used, the average number of cycles in ACA commands was only 22, and the proposed ACA implementation lead to energy savings of 6.5%.

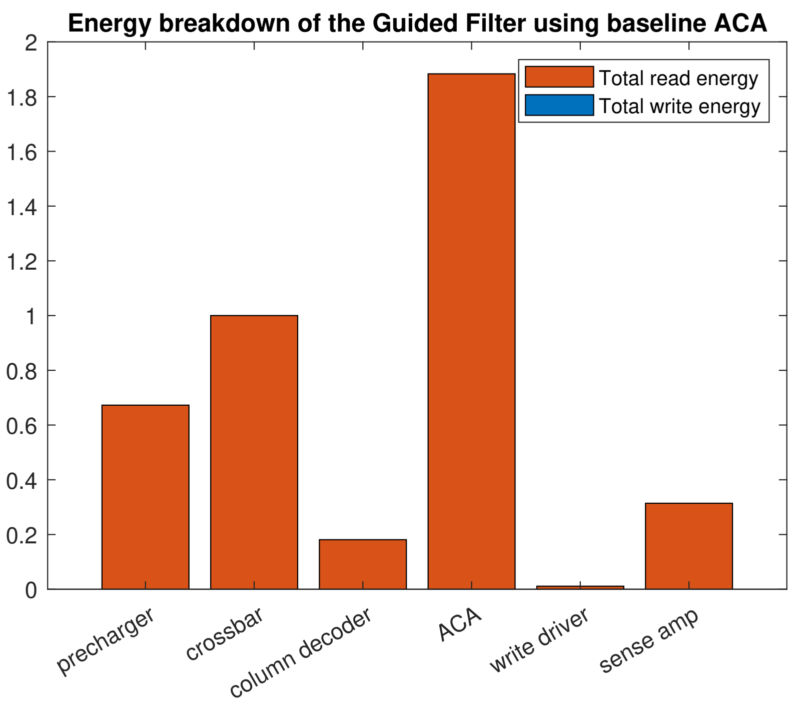


Figure 5.8: Energy results of the Guided Filter with baseline ACA

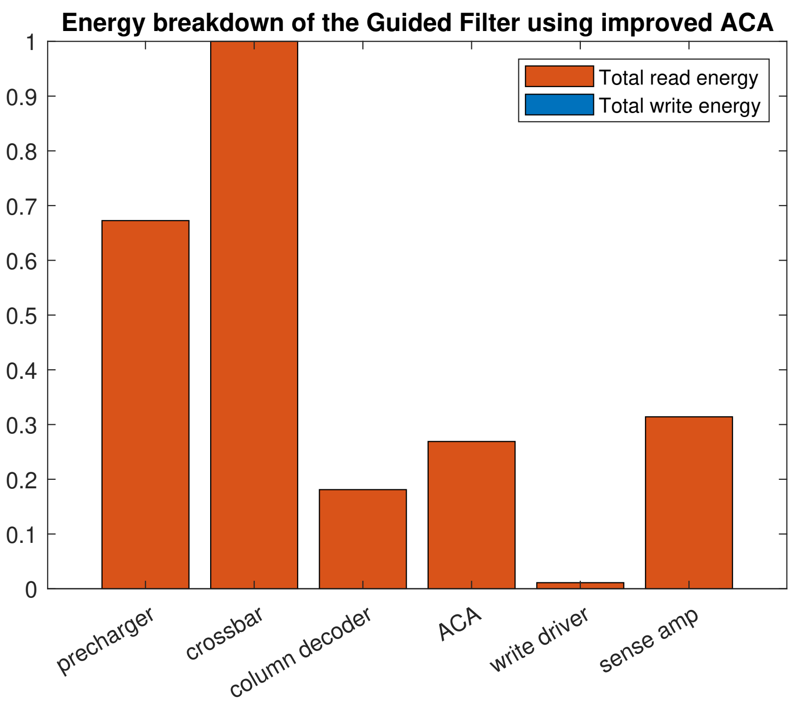


Figure 5.9: Energy results of the Guided Filter with proposed ACA

5.4. Area overhead estimation

Figure 5.11 shows a comparison between the area of the sub-array without ACA implemented, with the area of the sub-array with the chosen implementation of ACA. Implementing the baseline ACA lead to an area increase of the sub-array of 8%, while implementing the chosen configuration of the proposed ACA lead to **an increase of the area of sub-array of about 9%.** This area increase comes because

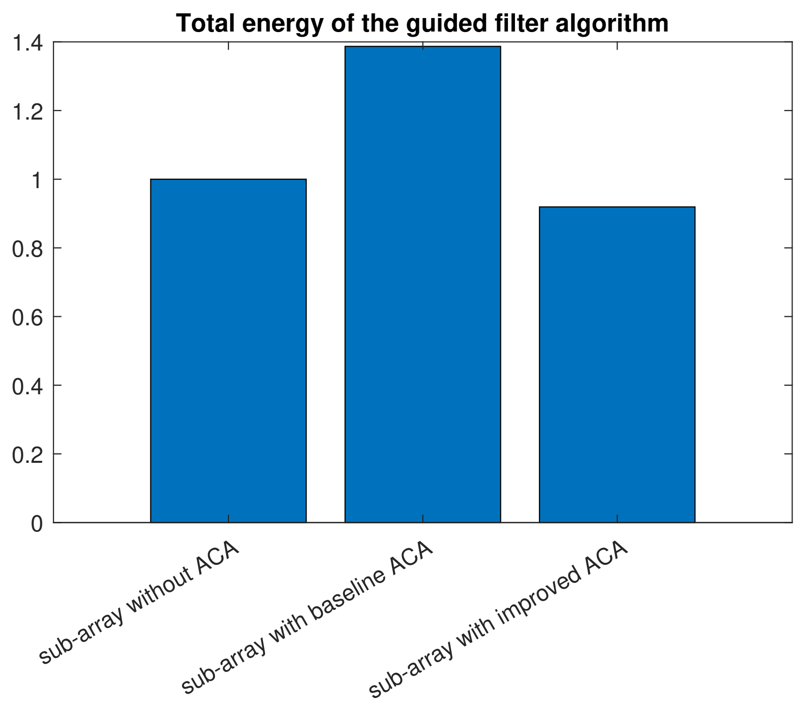


Figure 5.10: Total energy of the Guided Filter

of of how large the ACA modules are in comparison with the modules they replace (e.g row decoder).

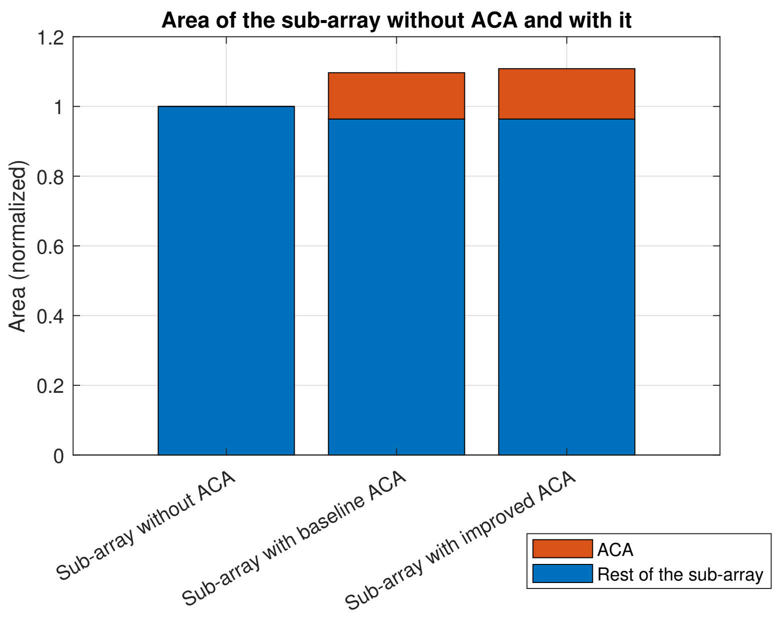


Figure 5.11: Area of the sub-array without ACA compared with the sub-array with ACA implemented

5.5. Discussion

In this chapter, ACA is integrated into the sub-array and its impact on area and energy cosumption is measured.

- Implementing the proposed ACA into the sub-array lead to an **area overhead of 9%** to the sub-array.

- Even without taking into consideration the energy consumed by driving the memory bus to send the addresses to the address decoder, the implementation of the proposed ACA into the sub-array has shown to improve energy consumption, given that the average number of cycles in the commands is larger than 14, which is the break-off point in Figure 4.15. The savings are also more significant for for scenarios in which the row address switches more frequently, as the row decoder does not consume much if consecutive accesses to the same row are frequent, like in the case of the March algorithm. This shows that ACA can be implemented on a sub-array without costing energy on the sub-array level, **but by reducing the energy at the sub-array level too**. However, the biggest gains in terms of energy will be obtained by reducing the usage of the hierarchical interconnect tree (memory bus) to transfer the addresses to the decoder, and also by using larger sub-arrays where ACA can generate more addresses, which would lead to more energy savings. The effect of this will need to be investigated in future work.

6 Conclusion and Future Work

6.1. Conclusion

ACA shows great potential to reduce the usage of the memory bus and therefore reduce energy consumption of memory accesses. In this work, ACA's circuit design is improved and implemented in an SRAM sub-array using the IMEC GAA nanosheet CMOS research PDK. Using the basic implementation, the ACA row shift register was inefficient in terms of energy consumption due to the dynamic power consumption of all the flip-flops. Various clock gating schemes have been explored and built upon, to reach a final improved implementation with

- The row shift register implemented using PBSC-OR-AD+ with 32 clock domains (each with 8 flip-flops).
- The column shift register implemented using PBSC-OR-AD+ with 4 clock domains (each with 2 flip-flops).
- The counter for the row shift register gated into 3 clock domains, and the counter for the column shift register gated into 2 clock domains.

This implementation has lead to an **increase of 9% in terms of area** compared to the sub-array without ACA, while baseline ACA has lead to an increase of 8%.

The sub-arrays with ACA and without it have been simulated on Spectre using different scenarios, and improved ACA lead to a **decrease of energy consumption of about 5-10% within the sub-array**, while baseline ACA lead to an increase of 13-38%. This shows great promise for reduction of energy consumption once improved ACA is implemented within an array and leads to a reduction of memory bus usage.

6.2. Future work

Analysis of the whole memory array

The main advantage of implementing ACA is reducing the address transactions over the memory bus, and therefore reducing energy consumption. Figure 6.1 shows how the memory bus is connected to all the sub-arrays. The energy consumed in memory operations is dominated by the energy consumed by the memory bus tree [15]. Therefore, reducing the usage of the memory bus would reduce the energy consumed significantly. Therefore, exploring the effect of implementing ACA on the entire memory area in terms of area and energy would paint a better picture about its effect and potential savings.

Implementation using larger sub-arrays

As the savings using ACA increase with the number of cycles an ACA command has, larger sub-array sizes need to be worked with in order to increase energy savings.

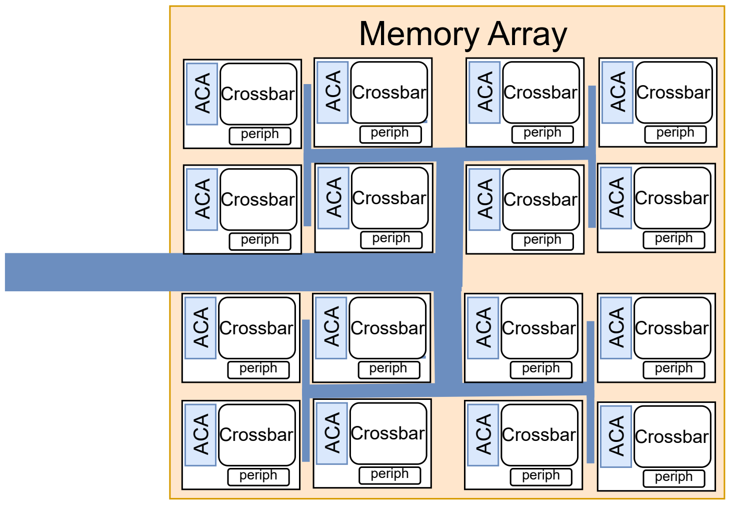


Figure 6.1: Sub-arrays and the memory bus

Multi-row implementation

In multi-row implementations, ACA offers the flexibility of picking different columns from different rows. The effects of this in terms of performance increase still need to be explored.

Using latches

ACA introduces a large overhead of area on the sub-array. This can be reduced by exploring other shift register implementations. For example, Yang [39] proposes an implementation of a shift register that uses pulsed latches and leads to a reduction of 37% in area compared to an implementation using flip-flops.

Bibliography

[1] Said Hamdioui et al. "Memristor for computing: Myth or reality?" In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 722–731.

[2] Philip Jacob et al. "Mitigating memory wall effects in high-clock-rate and multicore CMOS 3-D processor memory stacks". In: *Proceedings of the IEEE* 97.1 (2009), pp. 108–122.

[3] Yuan Xie. "Future memory and interconnect technologies". In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 964–969.

[4] Erfan Azarkhish et al. "High performance AXI-4.0 based interconnect for extensible smart memory cubes". In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 1317–1322.

[5] Amirreza Yousefzadeh et al. "Energy-efficient in-memory address calculation". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 19.4 (2022), pp. 1–16.

[6] Byunghyun Jang et al. "Exploiting memory access patterns to improve memory performance in data-parallel architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2010), pp. 105–118.

[7] Reena Panda et al. "Statistical pattern based modeling of GPU memory access streams". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[8] Peter Braun and Heiner Litz. "Understanding memory access patterns for prefetching". In: *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*. 2019.

[9] Jun Shao and Brian T Davis. "A burst scheduling access reordering mechanism". In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE. 2007, pp. 285–294.

[10] DMACTHDM Works. "DMA Fundamentals on Various PC Platforms". In: ().

[11] Kazukuni Kitagaki et al. "New address-generation-unit architecture for video signal processing". In: *Visual Communications and Image Processing'91: Image Processing*. Vol. 1606. SPIE. 1991, pp. 891–900.

[12] Ittetsu Taniguchi et al. "Systematic architecture exploration based on optimistic cycle estimation for low energy embedded processors". In: *2009 Asia and South Pacific Design Automation Conference*. 2009, pp. 449–454. DOI: 10.1109/ASPDAC.2009.4796521.

[13] Marc Moreno-Berengue et al. "Address generation unit for multimedia applications on application specific instruction set processors". In: *IECON 2010-36th Annual Conference on IEEE Industrial Electronics Society*. IEEE. 2010, pp. 1052–1057.

[14] Guillermo Talavera et al. "Address generation optimization for embedded high-performance processors: A survey". In: *Journal of Signal Processing Systems* 53 (2008), pp. 271–284.

[15] Zhenlin Pei et al. "Technology/Memory Co-Design and Co-Optimization Using E-Tree Interconnect". In: *Proceedings of the Great Lakes Symposium on VLSI 2023*. GLSVLSI '23. , Knoxville, TN, USA, Association for Computing Machinery, 2023, pp. 159–162. ISBN: 9798400701252. DOI: 10.1145/3583781.3590311. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3583781.3590311.

[16] Said Hamdioui et al. "Applications of computation-in-memory architectures based on memristive devices". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 486–491.

[17] Said Hamdioui et al. "Memristor based computation-in-memory architecture for data-intensive applications". In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 1718–1725.

[18] Hoang Anh Du Nguyen et al. “A classification of memory-centric computing”. In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 16.2 (2020), pp. 1–26.

[19] Eero Lehtonen, Jussi H Poikonen, and Mika Laiho. “Memristive stateful logic”. In: *Handbook of Memristor Networks* (2019), pp. 1101–1121.

[20] Shahar Kvatinsky et al. “MAGIC—Memristor-aided logic”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 61.11 (2014), pp. 895–899.

[21] Abhairaj Singh et al. “A 115.1 TOPS/W, 12.1 TOPS/mm 2 Computation-in-Memory using Ring-Oscillator based ADC for Edge AI”. In: *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE. 2023, pp. 1–5.

[22] Shuangchen Li et al. “Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories”. In: *Proceedings of the 53rd Annual Design Automation Conference*. 2016, pp. 1–6.

[23] Ali Shafiee et al. “ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars”. In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 14–26.

[24] Kevin Hsieh et al. “Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation”. In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE. 2016, pp. 25–32.

[25] Vivek Seshadri et al. “RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 2013, pp. 185–197.

[26] Vivek Seshadri et al. “Gather-scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. 2015, pp. 267–280.

[27] Benjamin Y Cho et al. “Near data acceleration with concurrent host access”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 818–831.

[28] Juan Gómez-Luna et al. “Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware”. In: *2021 12th International Green and Sustainable Computing Conference (IGSC)*. IEEE. 2021, pp. 1–7.

[29] Sukhan Lee et al. “Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 43–56.

[30] Vasudha Gupta and Mohab Anis. “Statistical design of the 6T SRAM bit cell”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 57.1 (2009), pp. 93–104.

[31] Taehui Na et al. “Comparative study of various latch-type sense amplifiers”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.2 (2013), pp. 425–429.

[32] Kaiming He, Jian Sun, and Xiaoou Tang. “Guided image filtering. European conference on computer vision”. In: *European conference on computer vision*. 2010.

[33] Meng Wu and Jeffrey A. Fessler. “GPU Acceleration of 3 D Forward and Backward Projection Using Separable Footprints for X-ray CT Image Reconstruction”. In: 2011. URL: `https://api.semanticscholar.org/CorpusID:204840795`.

[34] Li Li, Ken Choi, and Haiqing Nan. “Activity-driven fine-grained clock gating and run time power gating integration”. In: *IEEE transactions on very large scale integration (VLSI) systems* 21.8 (2012), pp. 1540–1544.

[35] Georgios Pouiklis and Georgios Ch Sirakoulis. “Clock gating methodologies and tools: a survey”. In: *International journal of Circuit theory and Applications* 44.4 (2016), pp. 798–816.

[36] Pilar Parra, Antonio Acosta, and Manuel Valencia. “Selective clock-gating for low power/low noise synchronous counters”. In: *International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer. 2002, pp. 448–457.

[37] Said Hamdioui et al. “Linked faults in random access memories: concept, fault models, test algorithms, and industrial results”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.5 (2004), pp. 737–757.

[38] Ali Bakhoda et al. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *2009 IEEE international symposium on performance analysis of systems and software*. IEEE. 2009, pp. 163–174.

[39] Byung-Do Yang. “Low-Power and Area-Efficient Shift Register Using Pulsed Latches”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 62.6 (2015), pp. 1564–1571. DOI: `10.1109/TCSI.2015.2418837`.