Optimization of On-chip Memory on FPGA Device

LiuyuLin

Delft University of Technology

TU Delft

Optimization of On-chip Memory on FPGA Device

by

LiuyuLin

Student number:	5689155	
Project Duration:	10, 2023 - 8, 2024	
Thesis committee:	Prof. G. Gaydadjiev,	TU Delft, supervisor
	Prof. P. French,	TU Delft
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science, Delft	

Cover:	Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

As I reflect on the journey that has led to the completion of this thesis, I realize that it would not have been possible without the support and guidance of many individuals. It is with immense gratitude that I acknowledge those who have contributed to my academic and personal growth throughout my time at Delft University of Technology.

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Georgi Gaydadjiev. Over the past ten months, our weekly meetings have been invaluable to my progress. His guidance, encouragement, and insightful feedback have greatly enriched my understanding and work throughout this project.

I would also like to extend my sincere thanks to Dr. Abdullah Aljuffri and my colleague Ismail Bourhaeil for their support and expertise. Their advice has been instrumental in refining my research and helping me stay on the right track.

I am especially grateful to the technicians, Yong Liu and Mark van Beusekom, for their technical support, which ensured the smooth execution of my project. Their assistance was crucial in solving various technical challenges that arose along the way.

Finally, I want to express my heartfelt appreciation to my friends and parents. Their unwavering support, both emotionally and practically, has been a constant source of strength throughout my studies and daily life.

Thank you all for your encouragement and contributions, which have made this journey possible.

LiuyuLin
Delft, August 2024

Abstract

In modern High-Performance Computing (HPC) systems based on Field Programmable Gate Arrays (FPGA) accelerators, First-In, First-Out queues (FIFOs) are crucial for data buffering and balancing, particularly in demanding applications requiring high-throughput data streaming. Some prominent examples are: large scale signal processing, machine learning, and online network processing. Typically, these FIFOs are implemented using the available on-chip blocks RAM (BRAM) blocks organized as a circular structure, leveraging the reconfigurable nature of FPGAs to manage data efficiently. However, as applications have to process larger and larger data capacities, the associated area and power consumption of these BRAM based implementations consume significantly large portion of the on-chip resources, posing challenges for both performance and efficiency.

This thesis explores an alternative approach for large and efficient reconfigurable on-chip FIFOs by designing a dedicated ASIC block containing linear shift register structure. Unlike circular FIFOs, which rely on complex control units such as address pointers and decoders, the linear FIFOs simplify the implementation by directly shifting the data through the registers. This approach eliminates the need for the additional intricate control logic typically associated with FPGA circular buffers, making it a potentially more efficient solution for a large class of streaming applications.

The first phase of this work focuses on implementing various FIFO designs on a Xilinx Virtex-7 series FPGA, utilizing different on-chip memory resources: registers, lookup tables (LUTs), and BRAM blocks. These implementations are investigated across different bit widths, in the range of hundreds of bits, and FIFO depths of several thousand stages. The second phase compares these FPGA implementations with an ASIC-based linear FIFO of similar size, synthesized using the TSMC 40 nm standard cell library. The results demonstrate that while the ASIC-based linear FIFO offers over twice the performance of the FPGA-based designs, it requires six to seven times more area. This is attributed to the fact that register-based storage cells are approximately ten times larger than SRAM cells, highlighting the inherent trade-off between performance and area in such designs.

To address the above area overhead, this project further investigates the use of two ring-counters to replace the address decoders typically found in SRAM-based FIFO designs. As data storage requirements increase, the complexity of address decoders also grows. By simplifying the control logic with ring-counters, the design achieved area reductions of 50% to 76% for FIFO depths of 1 Kbits and 2 Kbits across various bit widths. These findings underscore the significant potential of this approach to optimize both area and performance in memory-intensive FPGA applications.

Contents

List of Figures

List of Tables

1

Introduction

This Chapter begins by presenting the motivation for optimizing on-chip memory in FPGAs with a focus on FIFO implementation. It discusses the promising future of FPGA technology and the critical role FIFOs play in various FPGA applications. Next, the chapter outlines the thesis contributions, summarizing the scope of work and key contributions made in the project. A brief overview of each subsequent chapter is then provided.

1.1. Motivation

Field-Programmable Gate Arrays (FPGAs) are specialized integrated circuits that can be reprogrammed after fabrication, therefore offer a adaptable hardware platform, allowing the implementation of complex digital logic and algorithms, such as processors, memory interfaces, and digital signal processing (DSP) units[17]. This adaptability has contributed significantly to the rapid expansion of the FPGA market.

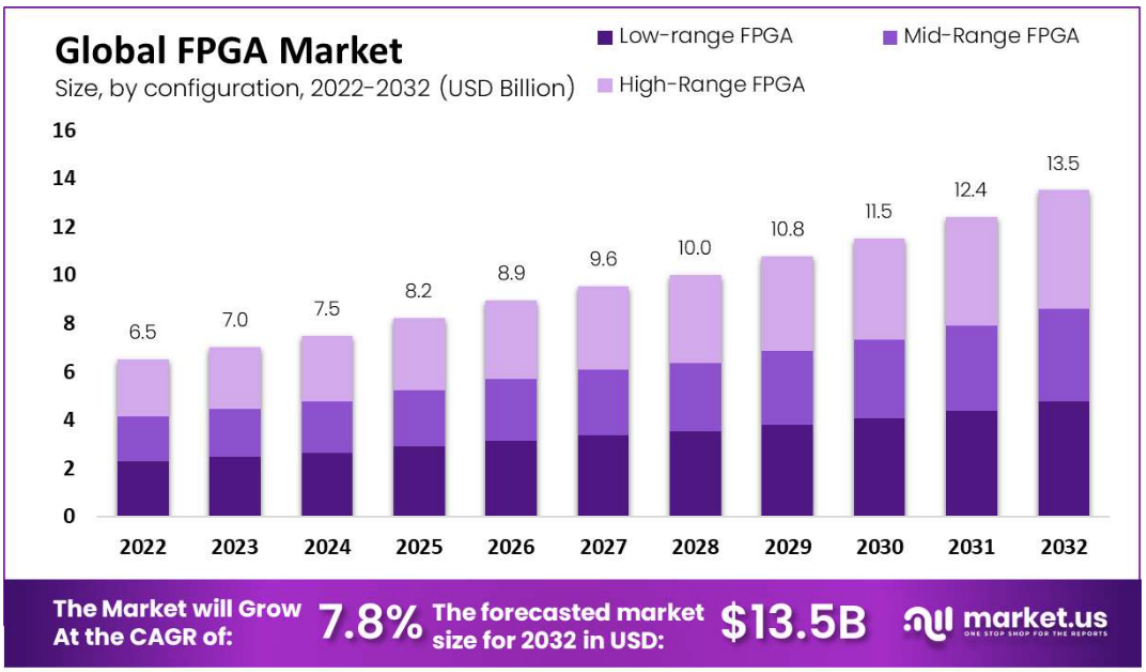


Figure 1.1: Global FPGA market [17].

According to Figure 1.1, the global FPGA market was valued at USD 7.0 billion in 2023 and is forecasted to reach USD 13.5 billion by 2032, growing at a compound annual growth rate (CAGR) of 7.8%. Figure 1.2 depicts the distribution of the FPGA market across various application sectors: telecommunications,

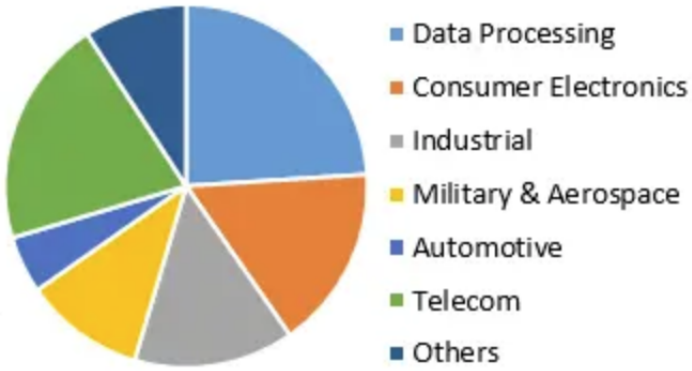


Figure 1.2: Embedded FPGA market by application in 2023 (%) [16].

consumer electronics, data processing, military and aerospace, industrial, automotive, and others. In 2023, telecommunications and data processing emerged as the dominant sectors, holding the largest share of the market.

FPGAs are widely applied in wireless and telecommunications systems, supporting functions such as optical transport networks, packet processing, and packet switching. They also facilitate the bandwidth needs of telecom providers, enabling network upgrades from 3G to LTE(Long Term Evolution Standard) and beyond. The ongoing deployment of 5G networks is anticipated to further boost FPGA demand through 2032 [17]. In data processing, FPGAs are increasingly adopted into data centers for data transport and computation. Embedded FPGAs, in particular, are used to link merchant silicon devices to provide specialized functionality [16], benefiting from FPGAs' advantages in hardware acceleration, flexible configuration, fast switching, and low-latency performance at a competitive cost.

In many of these applications, large volumes of data need to be transmitted, leading to the widespread use of FIFOs (First In, First Out queues) for data transmission and buffering. Within FPGA designs, on-chip Block RAM (BRAM) is frequently utilized for implementing FIFOs, serving as buffers to offload work from CPUs or facilitating data transfer between devices to ensure smooth and orderly communication between systems. However, with the increasing volume of data to be processed, BRAM-based FIFOs face significant challenges related to area, power consumption, and performance. Therefore, this project focuses on optimizing the performance of BRAM-based FIFOs, exploring strategies such as designing and implementing ASIC-based FIFOs to replace existing FPGA FIFO blocks, or optimizing control elements within FPGA FIFOs.

1.2. Thesis Contributions

This thesis aims to optimize FPGA on-chip memory through various design approaches. However, the topic of FPGA on-chip memory is broad. Given the extensive applications and promising potential of FPGAs in telecommunications and data processing—where FIFO buffers play a key role in caching and transmitting data—this project focuses specifically on optimizing FPGA on-chip memory for FIFO buffers. The first objective is develop and implement ASIC-based linear FIFOs to investigate its potential to be a better alternative for the conventional circular FIFOs on FPGA. The second objective is to optimize the address decoder in conventional BRAM-based FIFOs. Because BRAM is the primary resource used for implementing FIFOs on FPGAs and has the optimal performance compared with other type of on chip memory resource on FPGA. An ASIC ring-counter structure is designed and implemented to replace the address decoder. The key contributions of this thesis are as follows:

1. **In-Depth analysis of circuilar FIFO implementations on FPGAs:** We studied circular FIFOs with varying bit widths and depths on Xilinx Virtex-7 series FPGAs, utilizing different memory resources, e.g., registers, LUTs, and BRAMs. The performance metrics—area, power consumption, and performance-were compared across these memory types. Our results demonstrate that BRAM is the most optimal choice for FIFO applications in terms of performance, area, and power efficiency;

2. **Linear ASIC FIFOs design as a potential alternative for BRAM-based Circular FIFOs on FP-**

GAs: We successfully developed and implemented linear ASIC FIFOs using TSMC's 40nm standard library, matching the dimensions of the previously implemented FPGA FIFOs. By applying the comparison model, we observed that while the linear ASIC FIFOs offered clear performance improvements, they came with significant area overhead compared to the BRAM FIFOs, rendering them suboptimal as on-chip memory alternative;

3. Novel, linear-access custom address decoder for BRAM modules based on ring counters: We designed and implemented an ASIC-based address pointer using ring counters, again leveraging TSMC's 40nm standard library. This design was tested with FIFO depths of 1Kbits and 2Kbits. The results, analyzed using the comparison model and reasonable estimations, indicate that the ASIC ring-counter design has significant potential for area optimization by replacing conventional address decoder in BRAM;

Through these contributions, this thesis provides valuable insights and practical solutions for improving the performance, area efficiency, and power consumption of on chip memory on FPGAs, specifically on FIFO implementation. It offers potential pathways for future research and industrial application.

1.3. Thesis Organization

This rest of the thesis is organized into five chapters.

Chapter 2 first introduces the three on-chip memory resources available in FPGAs: registers, lookup tables (LUTs), and block RAM (BRAM), and explores the composition and evolution of FPGA on-chip memory systems. It then delves into a detailed discussion of the two types of FIFOs employed in the proposed designs: linear FIFO and circular FIFO;

Chapter 3 describes the design and the implementation flows of both ASIC-based linear FIFO and FPGA-based circular FIFO. The later one utilizing the three different on-chip memory resources. It also includes the process of correcting hold time violations on ASIC design through multi-tap H-tree clock tree synthesis;

Chapter 4 focuses on the process of the transformation of the obtained results to ensure comparability between designs across different technology nodes and implementation platforms, followed by a detailed analysis and comparison of the outcomes;

Chapter 5 presents the optimization of the BRAM address decoder by proposing and implementing a ring-counter ASIC structure. It leads to substantial improvements in area efficiency by replacing the conventional address decoder in the BRAM;

Chapter 6 provides a summary and future work of this thesis.

2

Background

The Chapter begins by providing an overview of the three types of on-chip memory in FPGAs: registers, LUTs, and BRAMs. It discusses their evolution, structure, and implementation methods, as well as the trade-offs involved in selecting key architectural parameters. This analysis lays the groundwork for understanding FPGA on-chip memory, which is essential for optimizing memory configurations within the project. Following this, the Chapter delves into FPGA on-chip memory architecture. Finally, it covers the fundamental principles of FIFOs, their application scenarios, and examines two distinct FIFO structures: linear FIFO and circular FIFO.

2.1. On-Chip Memory Resource On FPGA

This section explores the development and characteristics of FPGA on-chip memory resources. We begin with the historical PAL (Programmable Array Logic) structures and progress to the more advanced LUT-based logic blocks, including the modern Fracturable LUTs. It covers key parameters such as LUT input count (K) and the number of LUT (N) in a single logic block, as well as LUT-RAM implementations. Next, Section 2.1.2 discusses the role and evolution of registers as fast-access cache memory, highlighting their increasing number and significance in FPGA designs. The focus then shifts to BRAM in Section 2.1.3, detailing its principles, functionality, and reconfigurability within FPGAs. We address considerations for BRAM design, including memory capacity, bit width, and read/write ports. Finally, Section 2.2reviews the distribution and performance of on-chip memory in FPGA systems, comparing how different memory types impact application performance.

2.1.1. Distributed RAM: Look-Up-Table (LUT)

As semiconductor technology advanced, leading to the early development of smaller chips capable of implementing specific logic functions, the design and manufacture of fixed-logic-gate chips became increasingly challenging. The growing complexity of logic functions and the rising demand for computational power further exacerbated these difficulties. In response, engineers began to investigate hardware that would allow for the flexible programming of logic functions, resulting in the creation of the Programmable Logic Device (PLD) in the 1960s. As this technology matured, it was commercialized in the late 1970s.

A PLD is an integrated circuit that can be programmed via a configuration file, enabling the rapid implementation and modification of various digital logic functions. Typically, PLDs consist of programmable logic gate arrays (PLAs), flip-flops, programmable internal connections, and programming units. The PLAs generally comprise arrays of AND and OR gates. Configuration files are processed through the programming unit to establish the necessary programmable connections within the arrays, thereby realizing a two-stage structure capable of executing the target logic function. Programmable flip-flops are commonly employed to control different output signals [7].

However, as digital circuits continued to evolve, it became increasingly difficult to implement more complex functions using this relatively rigid structure. Consequently, a LUT-based FPGA architecture

was developed. A LUT utilizes a truth table to compute the output of a logical function for every possible combination of input variables. By pre-programming all potential outputs into the FPGA hardware, designers can achieve the desired output by controlling a smaller, more manageable set of inputs.

Introduction to LUTs: Background and Basic Structure

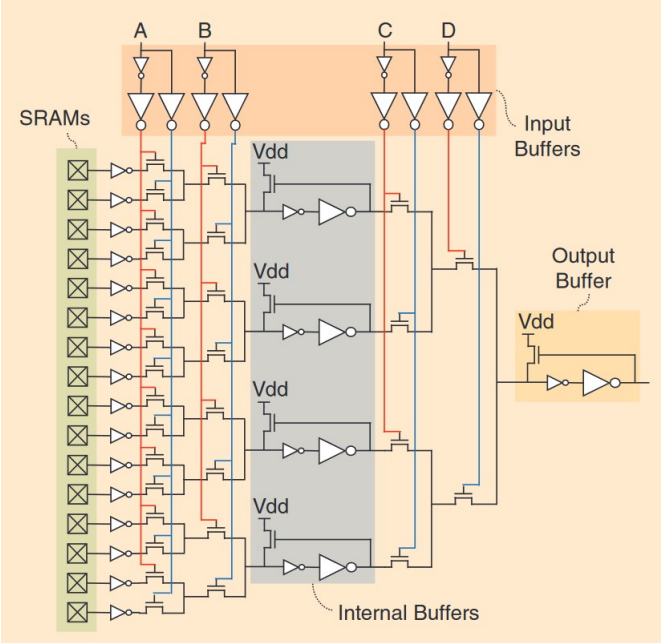


Figure 2.1: A 4-LUT in transistor-level [7].

The significant breakthrough in FPGA technology occurred in 1984 when Xilinx introduced the world's first look-up table (LUT)-based Field-Programmable Gate Arrays (FPGAs), offering a more flexible and efficient solution. Figure 2.1 illustrates the internal transistor-level structure of an SRAM-based LUT, which is employed to implement the functionality of a specific truth table. In the figure, A, B, C, and D represent the four input signals. The structure features four columns of transistors, forming a horizontal tree from the leftmost 16 SRAM cells to the rightmost output port. The four input signals and their complements are connected to the gates of the transistors in each column, with alternating halves of inputs and their complements in a single column. By loading preset data into the SRAM cells, different input values can retrieve the data from the corresponding SRAM cell, thereby implementing the desired logic function.

The smallest replicable unit within this FPGA architecture is the basic logic element (BLE), depicted in Figure 2.2. The BLE comprises a K-LUT, a register, and two multiplexers. The configuration and number of these elements vary as the FPGA evolves. The k-inputs function as selection signals, enabling the retrieval of target data from 2^k SRAM cells. The register is utilized to implement sequential logic, while the two multiplexers handle output and feedback signals, respectively. This structure enables the BLE to perform a complete combinational or sequential logic function [4].

Multiple BLEs are further combined to form a logic block (LB), as shown in Figure 2.3. Input signals are fed into the LB via connection block multiplexers located at the bottom routing nets. The multiplexers within the local crossbar then process and select external input signals and internal feedback signals before routing them to the appropriate BLEs. The output signals from the LBs are routed to other LBs through switch block multiplexers on the right.

Compared to the rigid structure of Programmable Array Logic (PAL), this LUT-based architecture enables the realization of complex logic functions more efficiently and within a smaller chip area. Consequently, this approach has been gradually adopted by mainstream FPGA manufacturers, including Altera and AT&T/Lucent, and integrated into commercial products [38].

The number of LUT inputs (denoted as K) and the number of BLEs (denoted as N) are critical param-

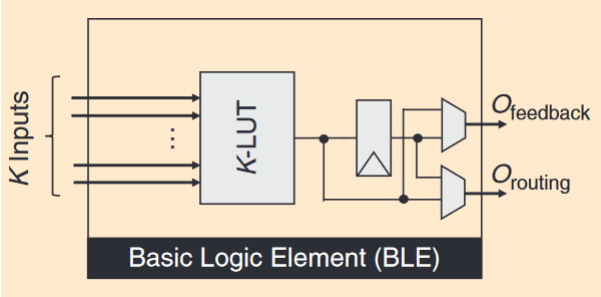


Figure 2.2: Basic structure of basic logic element (BLE)[7].

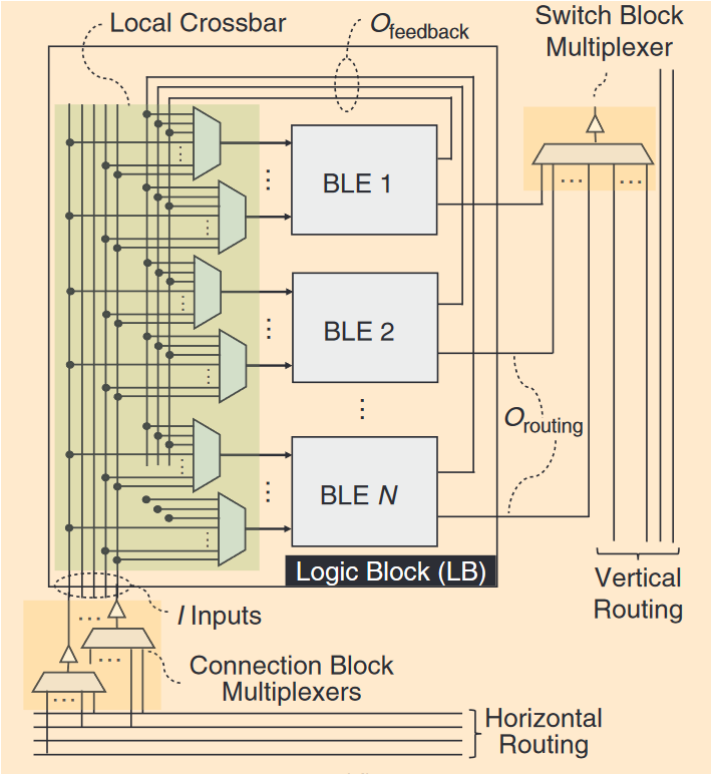


Figure 2.3: Basic structure of logic block (LB) [7].

eters in the design of LBs in FPGAs. These parameters fundamentally determine the distribution of resources within and between LBs. As K increases, a LUT can implement more complex functions, reducing the number of intermediate steps needed to generate partial results before the signal is processed by the LUT. This reduction in logic levels enhances the overall system performance. However, increasing K also escalates the resource requirements within a LUT, and the introduction of additional internal critical paths can diminish the performance of an individual LUT.

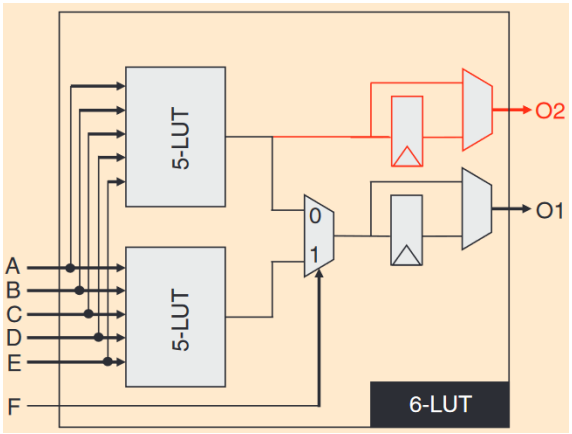
Similarly, in contrast to the purely metallic interconnections found in ASICs, FPGA inter-LB connections necessitate a substantial number of switch blocks to manage routing, which incurs significant costs. An increase in N allows a single LB to handle more complex functions, thereby reducing the necessity for inter-LB communication. However, this also leads to a substantial rise in local interconnections, causing the internal operation speed of the LB to decrease linearly with increasing K.

The design of parameters K and N is thus a delicate balance in determining the granularity of the FPGA fabric—whether to opt for fewer large LBs or a greater number of smaller, lightweight LBs to implement logic functions. A study conducted in 2004 examined this trade-off and concluded that optimal area-delay products were achieved with K values between 4 and 6 and N values between 3 and [1]. In industry, early LUT-based FPGAs, such as those from 1984, featured only two 3-LUTs per LB. By

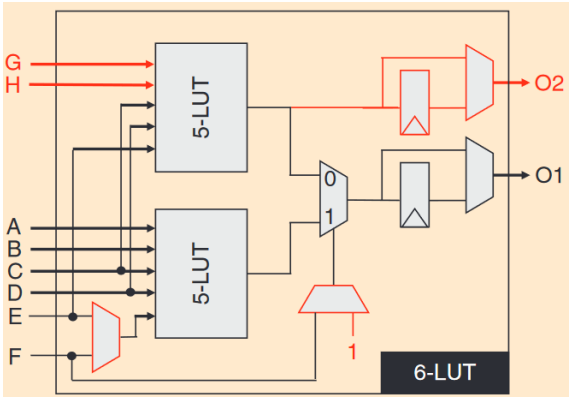
1999, the Xilinx Virtex series and Altera's Apex 20K series had evolved to configurations with K=4, N=4,and K=4, N=10, respectively [7].

Fracturable LUTs: Next-Generation Structures

In a study investigating the optimal values of K and N, researchers found that using ten 6-LUTs improved performance by 14% compared to structures with ten 4-LUTs, albeit with a 17% increase in area [1]. Notably, 64% of the 6-LUTs tested by various applications did not fully utilize all six input ports [29]. This observation led to the development of the fracturable-LUT, a structure designed to combine the benefits of different LUT sizes from Stratix II Altera in 2003 [12]. A fracturable K,L-LUT can function as either a single K-LUT or as two smaller (K-1)-LUTs, with a maximum of K+L external inputs.



(a) With 5 shared imputs



(b) With 2 additonal inputs ports and steering multiplexers, so that only 2 shared inputs

Figure 2.4: Two different configurations for fracturable 6-LUT with two 5-LUTs [7].

As illustrated in Figure 2.4a, two 5-LUTs share five identical inputs, while a 6-LUT is realized by feeding the sixth input, F, into a multiplexer. The structure also allows both 5-LUTs to operate simultaneously by enabling the red portion of the multiplexer. However, because the two LUTs share the same inputs, it is challenging to use them concurrently to implement different logic functions. Figure 2.4b presents an alternative fracturable-LUT design with eight external inputs, only two of which are applied directly to both LUTs. Although this alternative design occupies a larger area, it offers greater flexibility for implementing diverse, smaller logic functions.

In the industry, Xilinx adopted the fully shared input architecture in their Virtex-5 series to minimize area [2], while Intel employed the partially shared input architecture in their Stratix II series. This architecture provides the flexibility to implement logic functions of varying complexity by allowing the LUTs to be split into several 6-LUTs, 5-LUTs, 4-LUTs, or 3-LUTs. Compared to the traditional 4-LUT LB architecture, fracturable-LUT LBs offer a 15% performance increase and a slight reduction in area, presenting a significant advantage [29] [7].

This overview highlights the progression from early FPGA LUT designs to advanced fracturable LUTs, showcasing how innovations in LUT architecture continue to enhance FPGA performance.

Implementation of LUT-RAM/distributed-RAM

In FPGA designs, transforming LUTs into RAM blocks represents a significant advancement in enhancing both flexibility and functionality. This section delves into the methodologies, limitations, and design considerations pertinent to this conversion. K-LUTs, with their 2^K SRAM cells, can be repurposed as memory cells. Each K-LUT functions as a $2^K \times 1$-bit ROM cell and, with the addition of a write module, can serve as an RAM cell. Given their even distribution across each BLE, LUTs are typically employed for the temporary storage of small cache data within logic modules to enhance access efficiency. For instance, a fracturable 6, 2-LUT can be configured as either a 64×1-bit or 32×2-bit memory; however, the 8 available LUT inputs are insufficient for both read and write operations once 5 or 6 inputs are dedicated to the read address. Consequently, additional ports are necessary to handle the write address and enable signals.

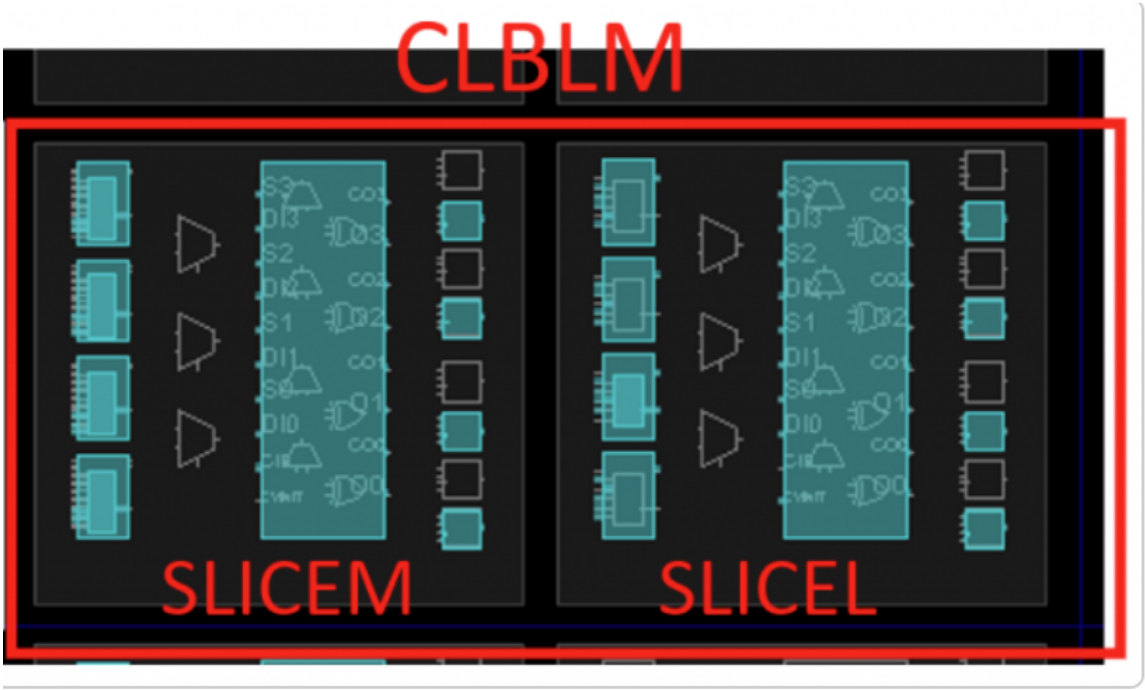


Figure 2.5: Basic structure of the programmable logic block (CLB) of a Xilinx device.

To address the port limitation, manufacturers have designed the 10 BLEs within each LB to function collectively as a single 64 x 10-bit or 32 x 20-bit LUT-RAM module, thereby sharing the required signaling ports. Unused ports in each LB are allocated for the connection of write addresses and enable signals, which are then propagated to all BLEs [28]. Considering the trade-off between the increased area required for LUT-RAM and the frequency of its usage, only half of the LUTs in commercial FPGAs from both Altera and Xilinx are convertible to distributed-RAM [7].

Figure 2.5 illustrates the programmable logic block (CLB) of a Xilinx device, which comprises two SLICEs: SLICEM (memory) and SLICEL (logic). Each SLICE consists of four 6-LUTs, three multiplexers, carry units, and eight registers. Although SLICEM and SLICEL are nearly identical in structure, SLICEM includes additional functionality to support LUT-RAM configurations, which requires extra logic. Figure 2.6 illustrates the LUTs in SLICEM and SLICEL. Compared to SLICEL, SLICEM's four LUTs each have an additional Dedicated Input (DI) signal for memory addressing: AI, BI, CI, and DI. Additionally, SLICEM features an extra input port, WE, for the write enable signal. These additional inputs allows SLICEM to operate as LUT-RAM, enhancing its capability to perform memory-related functions.

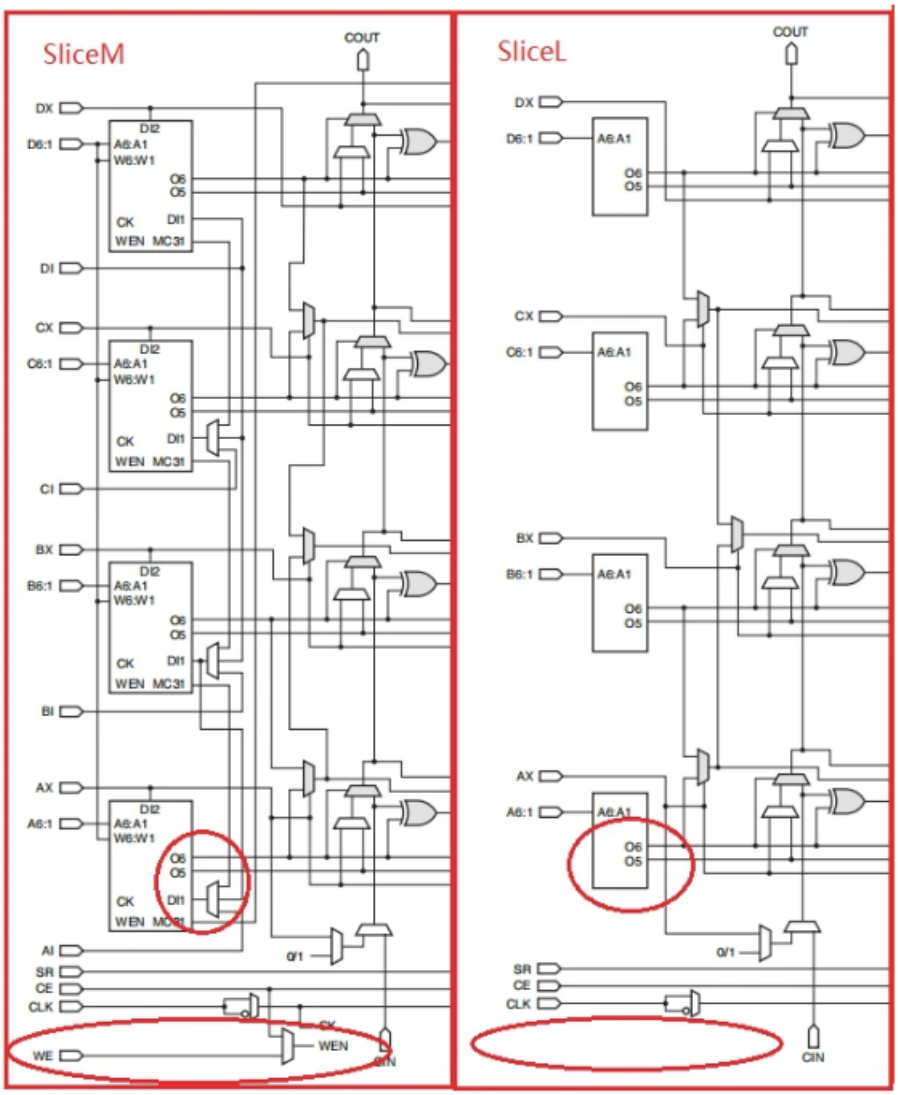


Figure 2.6: Different part between SLICEM and SLICEL

2.1.2. Register-based Memory: Flip-Flop (FF)

The first form of on-chip memory resource in FPGAs is the register. Digital circuits utilize two primary types of storage units: flip-flops and latches. The key distinction between these two lies in their operational behavior. Flip-flops update the stored output state only when the clock edge arrives, whereas latches continuously change the stored output state when the enable signal level changes to high or low. A flip-flop forms a one-bit register, and multiple flip-flops can be combined to create a multi-bit register. Memory, which consists of numerous registers, stores binary codes where each register, known as a memory cell, holds an independent value.

In early FPGA architectures, each BLE had only one register primarily used for controlling output signals, as depicted in Figure 2.2. With the introduction of Fracturable LUTs, manufacturers added a second register to enable simultaneous use of two LUTs, as shown in Figure 2.4. The number of flip-flops was further increased to four in the Xilinx Stratix V architecture to accommodate deeper pipelining requirements [27]. A flip-flop, consisting of two latches, occupies a larger area. To address the area constraints and improve operational speed, some clock-edge flip-flops in the Stratix V family were replaced with pulse latches. However, pulse latches are less reliable for timing control, as they can continuously change the output during short bursts and are sensitive to glitches. Additionally, pulse latches cannot be asynchronously reset, leading to indeterminate states after power-up [7].

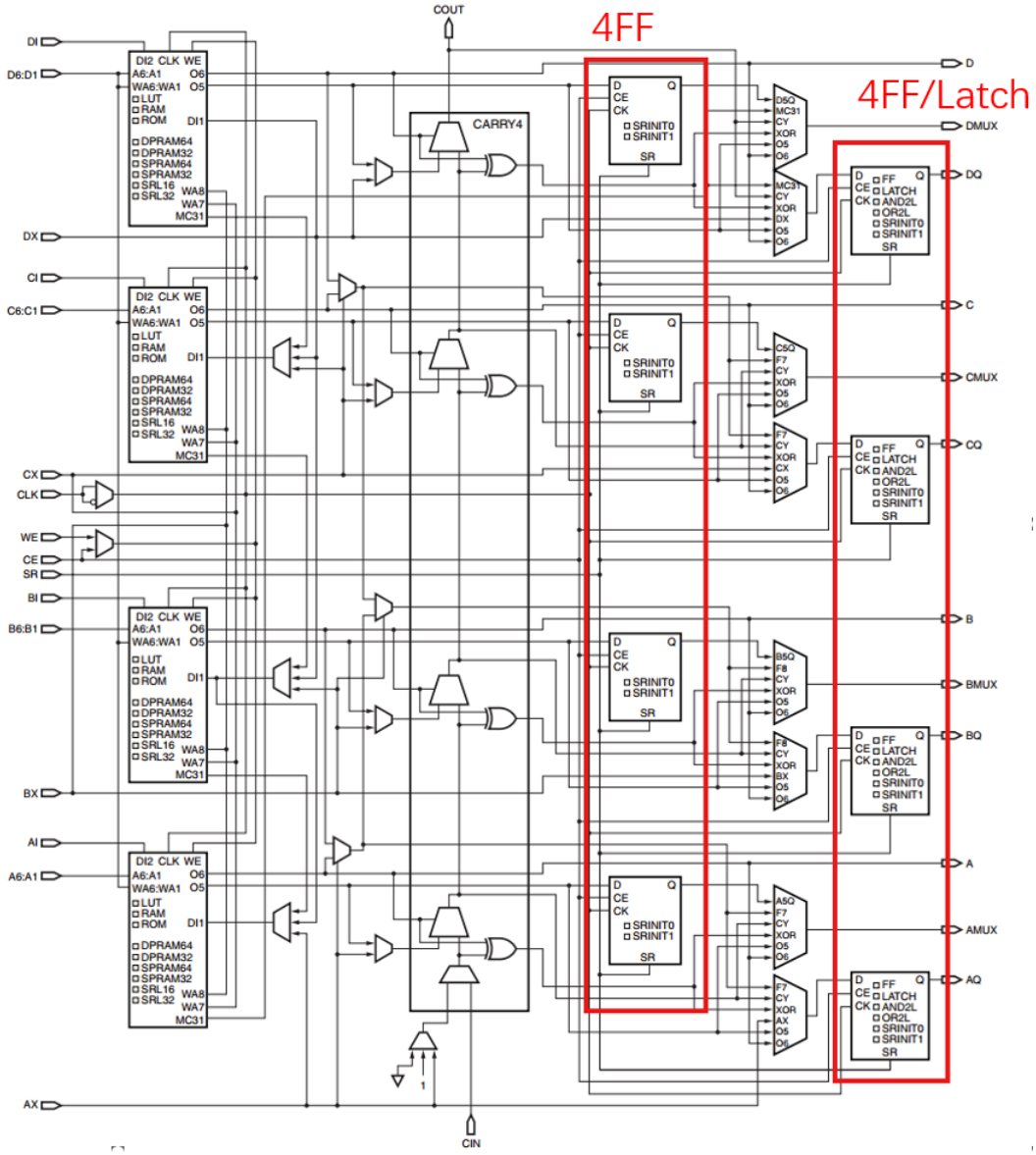


Figure 2.7: Detail structure of SLICE with highlighted flip-flops and latches.

At the foundation of the Series 7 FPGA architecture, each CLB contains two SLICEs, as illustrated in Figure 2.5, with each SLICE housing eight storage cells. Although these are referred to as memory cells, they actually consist of 4 flip-flops and 4 configurable flip-flops or latches, as shown in Figure 2.7. All flip-flops within the FPGA are typically configured as D-type flip-flops, which can vary based on their reset method, reset level, and power-up state. The four storage cells on the left side of the SLICE can only function as flip-flops, while the four on the right can be configured as either flip-flops or latches. It is important to note that if the four flip-flops in the SLICE are configured as latches, the remaining four flip-flops cannot be utilized, resulting in a potential waste of resources.

In FPGA design, flip-flops are generally used to clock or cache logic blocks. Given that there are only a few flip-flops per LB and each flip-flop occupies significantly more area than other type of memory cells, they are not typically used to construct larger-scale memory arrays.

2.1.3. Block RAM

Overview and Fundamental Principle

With the continuous evolution of contemporary FPGAs, the demand for systems-on-chip (SoCs) to address increasingly complex tasks has intensified. This progression has pushed the boundaries of traditional storage methodologies, such as register storage and LUT-RAM , which are less integrated and optimized than their counterparts in ASICs. Consequently, it becomes challenging to store large amounts of data directly on-chip, necessitating the use of external memory in FPGA-based applications that require substantial data storage. However, in many cases, the bandwidth between the FPGA and the external memory can become a bottleneck, restricting the overall system performance. To address these limitations, the need for specialized on-chip memory modules that can flexibly adjust storage capacity, bit width, and the number of ports has emerged as a critical requirement in modern FPGA design. The introduction of on-chip hardware modules dedicated to storage, known as block RAMs (BRAMs), marked a significant milestone in the development of FPGA technology. These BRAMs were first introduced in Altera's Flex 10K in 1995 [11] and have since become an essential feature of contemporary FPGA architectures, providing high-speed, low-latency data storage capabilities.

Distributed RAM, synthesized by a synthesis tool, is typically realized by cascading resources through multi-level LUTs. This process, however, can be inefficient as the LUTs may be distributed across different regions of the FPGA fabric, resulting in suboptimal performance due to the physical distance and routing delays. Despite its inefficiencies, distributed RAM offers considerable flexibility. It leverages the abundance of LUT resources available in FPGAs and can be configured dynamically based on specific application needs, making it suitable for scenarios where RAM latency is not a critical concern.

In contrast, BRAM is a specialized memory resource designed explicitly for high-speed data storage and retrieval, addressing the limitations of distributed RAM. Unlike distributed RAM, which relies on the flexible but less efficient use of LUTs, BRAMs are strategically placed and wired within the FPGA architecture to ensure high-speed access and predictable, low-latency cycles. The deliberate layout of BRAMs, coupled with their integration with logic resources, makes them indispensable in applications such as network communication and digital signal processing, where high-speed data caching is essential. Modern FPGAs have evolved to include maximum nearly 200MB of BRAM resources, underscoring their significance. Consequently, BRAMs now occupy a substantial portion—typically around a quarter—of the on-chip area in contemporary FPGA devices [35].

The architecture of BRAM is depicted in Figure 2.8. The blue components represent elements common to SRAM, while the green components are specific to FPGA BRAM. At the core of the BRAM is a crossbar, which serves as the SRAM cell array responsible for information storage. Two row decoders flank this array, tasked with compiling the row address and activating the wordlines of the target rows. During a read operation, all bitlines are precharged. The row decoder then asserts the wordline, connecting all cells in the selected row to their respective bitlines. The stored data induces slight voltage changes on the bitlines, which are detected by sense amplifiers. The required data is then outputted through a column multiplexer, controlled by the column address. For write operations, the row decoder again activates the selected wordline, and the write circuit inputs the new data via the write driver, overwriting the existing data in the memory cells [35] [7].

This architectural sophistication enables BRAMs to meet the high demands of modern FPGA applications, balancing the need for speed, flexibility, and efficiency in complex, data-intensive tasks. As FPGAs continue to evolve, BRAMs will likely play an even more central role, pushing the boundaries of what can be achieved with reconfigurable computing systems.

Feature Trade-offs and Configurability

When designing and characterizing BRAM in FPGAs, several critical metrics must be considered: memory capacity, bit width, and the number and type of read/write ports. These parameters significantly impact the performance, area efficiency, and applicability of the BRAM in various scenarios.

The memory capacity of an FPGA BRAM directly influences the area of the memory array. As the number of bits stored in the BRAM increases, the area required for the memory array scales linearly. This relationship is crucial because it affects the overall area utilization of the FPGA. Larger memory capacities are advantageous in applications that fully exploit the storage potential, as they lead to more efficient use of the available area. Specifically, the control circuitry's overhead per bit decreases as the

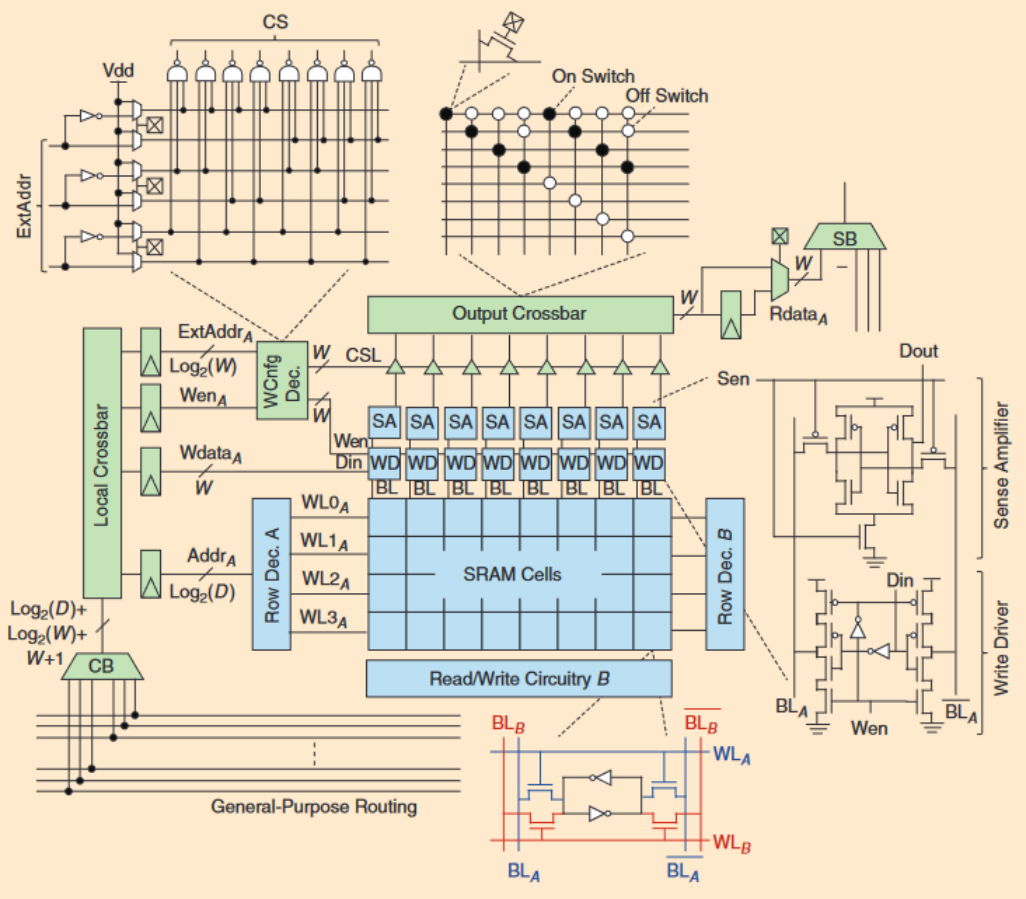


Figure 2.8: Architecture of a dual-port SRAM-based FPGA BRAM with maximum 8 bits data width. The blue parts are common in any SRAM module, while the green parts are only for FPGA [7].

memory capacity increases, enhancing the overall space utilization. However, in small applications, the allocation of a large BRAM results in underutilization and waste of resources. Thus, designers must carefully balance the memory capacity against the specific needs of the application, ensuring that the BRAM is neither oversized nor underutilized. This balance is essential in optimizing the FPGA's performance and resource efficiency.

Another critical design consideration is the bit width of the BRAM. The bit width determines how many bits can be read or written simultaneously. Wider bit widths enable faster data access and are beneficial in applications requiring high throughput. They also increase the versatility of the BRAM, allowing it to be used in a broader range of scenarios. However, increasing the bit width imposes additional demands on the read/write circuitry, which in turn affects the area and power consumption of the BRAM. Modern FPGA BRAMs typically include programmable bit widths to accommodate varying application needs. For example, as illustrated in the green section of Figure 2.8, a three-bit bit-width select address is used to configure the BRAM, which, after passing through a multiplexer, generate column select(CS) signals and enable signal(Wen) to manage the sense amplifier and write driver for selective bitline activation. This flexibility allows for optimized bit-width selection based on the application, and requires less costs than conventional SRAM with the same functionality.

The design of read/write ports in BRAM is another aspect where trade-offs are necessary. The choice of the number and type of ports affects both the functionality and the area required for connectivity resources. For instance, a dual-port memory cell, as shown at the bottom of Figure 2.8, with red and blue lines, allows simultaneous read and write operations within a single clock cycle. The use of dual ports increases the flexibility of the BRAM but also demands more routing resources.

In FPGA architectures, routing resources are especially significant for smaller BRAMs. For example, in

an 8Kb BRAM, approximately 35% of the total area is dedicated to connection ports [44]. These ports are necessary to interface with the connection block multiplexer(CB) and switch block multiplexer(SB), as depicted in Figure 2.8, which filter input signals and direct output signals to the appropriate modules. Consequently, designers must balance the desire for increased read/write flexibility against the available area budget.

BRAM Ports	BRAM Mode	# Routing Ports
1r	Single-port ROM	$log_2(D) + W$
1r/W	Single-port RAM	$log_2(D) + 2W$
1r+1W	Simple dual-port RAM	$2log_2(D) + 2W$
2r/W	True dual-port RAM	$2log_2(D) + 4W$
2r+2W	Quad-port RAM	$4log_2(D) + 4W$

Table 2.1: Routing port resources for different types and numbers of BRAM read/write ports (W: data width and D: depth [7].)

Table 2.1 provides a detailed comparison of the number of connected ports required for different types and configurations of BRAMs. For single-port read-only memories (ROM), the number of required routing ports is determined by the formula $\log_2(D) + W$, where D is the depth of the BRAM and W is the bit width. For single-port random-access memory (RAM), additional W ports is needed to facilitate data read operations. Simple dual-port RAMs, which allow for simultaneous reading and writing, require twice as many ports as single-port ROMs. True dual-port RAMs, which offer two fully independent read/write ports, require significantly more ports, with a total of $2\log_2(D) + 4W$.

Despite the internal architecture of BRAMs supporting true dual-port functionality, the connection interfaces typically adhere to the simple dual-port standard due to the common application of BRAMs in FIFO configurations, where only one read and one write operation occur per cycle. As a result, when true dual-port functionality is necessary, it can only operate at a half bit width, a design choice driven by the need to balance application demands and routing resources [7].

An alternative approach to managing connectivity resources is to increase the operating frequency of the BRAM relative to other logic modules. By doing so, a BRAM with fewer connectivity resources can achieve similar performance to a BRAM with additional ports [18]. However, this method introduces significant timing challenges, which must be carefully managed to avoid compromising system stability and performance.

2.2. Memory Architecture Evaluation

This section provides an overview of the on-chip memory system. It begins with a discussion of memory mapping for various memory types and sizes. It then examines the evolution of the distribution of different memory resources within the device. To gain a deeper understanding, different memory structures are implemented for a specific memory size requirement. The results are compared to better elucidate the role of each memory type within the system.

2.2.1. Memory Mapping Strategies

In FPGA design, an essential phase involves the mapping of RAM resources, where the memory elements described by the designer in the hardware description language (HDL) are translated into physical memory modules on the FPGA chip. This process, managed by Electronic Design Automation (EDA) tools, ensures that the memory structures align with the physical architecture of the FPGA.

One of the critical features of EDA tools is the ability to allow designers to customize memory resource allocation. By incorporating a ”RAM_TYPE” directive within the HDL or constraints file, designers can specify the type of on-chip memory to which the logical memory should be mapped. This customization enables optimized usage of the FPGA’s memory hierarchy, ensuring that the most suitable type of memory (such as Block RAM, distributed RAM, or UltraRAM in certain FPGA families) is used for the intended application.

To accommodate memory structures of various depths and bit-widths, EDA tools can concatenate multiple BRAM blocks. This process is essential when the required memory configuration exceeds the

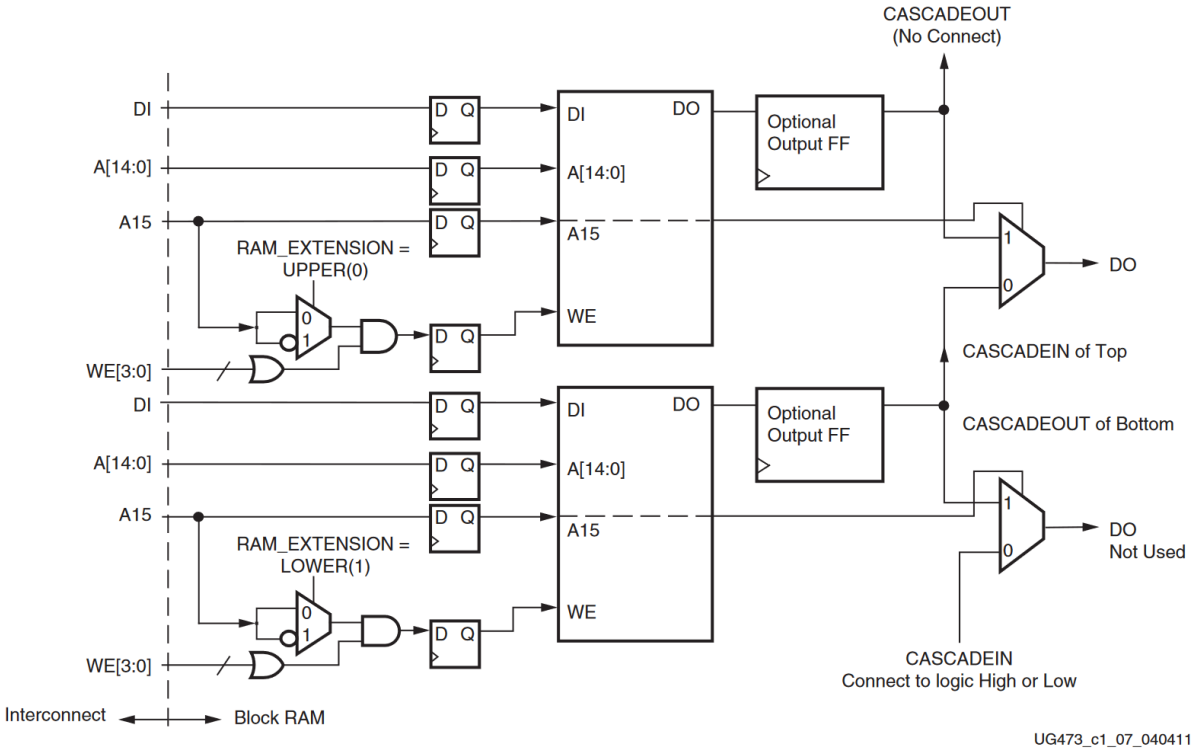


Figure 2.9: Architecture of cascadable Block RAM [41].

capacity of a single BRAM block. For example, to implement a memory space of 2024×32bits, four 1024×8-bit BRAM blocks can be arranged in parallel first to form a 1024×32bits RAM. Then two set of the block further form a 2024×32bits RAM. The concatenation process involves utilizing the most significant bits (MSBs) of the write address to generate the write enable signals for the individual BRAMs, ensuring that data is correctly distributed across the parallel blocks. Similarly, the read address is used to select the appropriate output from the BRAMs via a multiplexer, which combines the outputs into a single 32-bit data word. This method of BRAM stitching allows for flexible memory configurations while maintaining efficient use of the FPGA's BRAM resources [7].

Figure 2.9 demonstrates this concept by illustrating the cascading of two 32K bit RAM blocks into a single 64K bit RAM in a Xilinx 7-series FPGA. The cascading process in this case is straightforward, requiring only a single enable signal to manage the combined memory block. No additional logic circuitry is necessary, which simplifies the design and reduces the potential for timing issues. However, it is important to note that the cascade mode in this FPGA family is limited to 64K bits, which imposes a constraint on the maximum memory size that can be achieved through this method.

In conclusion, RAM mapping and the concatenation of BRAMs are crucial in FPGA design, facilitating the development of flexible and efficient memory architectures that are tailored to meet the specific requirements of various applications.

2.2.2. Evolution of FPGA Memory Capacity and Diverse Architecture

Over the decades, the architecture of on-chip storage in FPGAs has evolved significantly, paralleling the growth in the complexity and capabilities of logic modules. This evolution is well illustrated in Figure 2.10, which traces the changes in the memory architecture of Altera/Intel FPGAs over time. Each dot in the figure represents a different FPGA model, with the x-axis showing the number of LEs in the device and the y-axis indicating the number of memory bits allocated to each LE. The labels denote the capacity of BRAM available on these devices.

In the early stages of FPGA development, as represented by the 350 nm Flex 10K device introduced

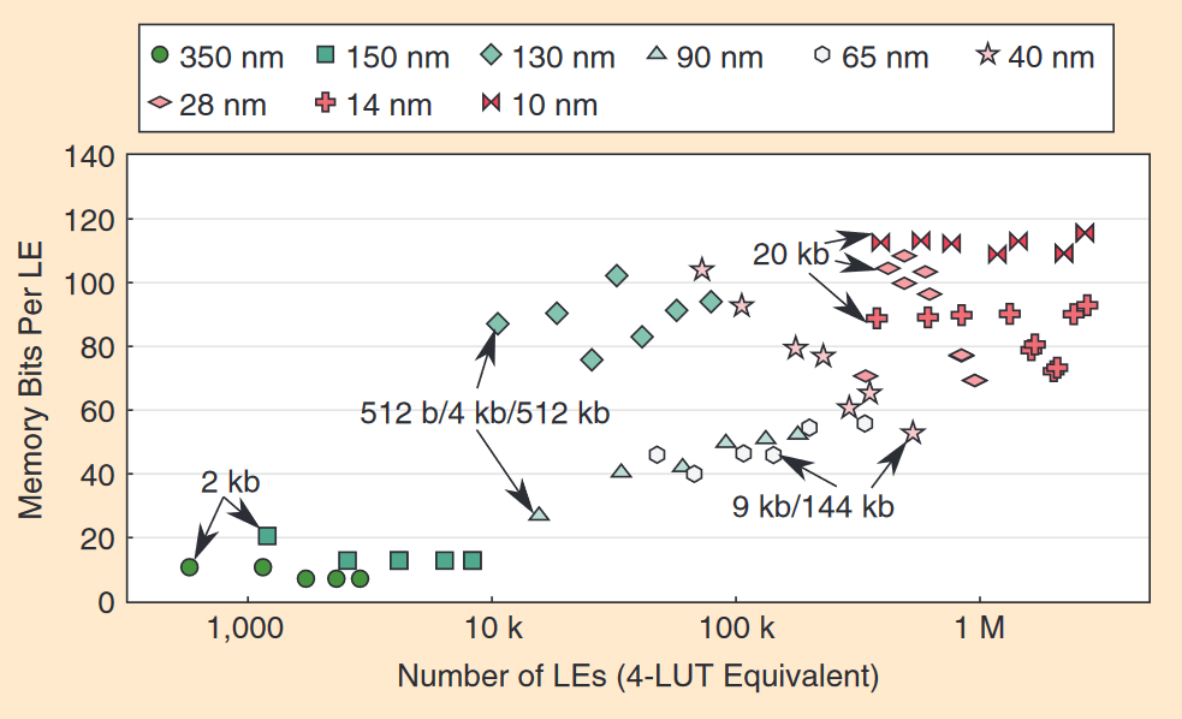


Figure 2.10: Memory bits per LE and number of LEs for Altera/Intel FPGAs from 350 nm to 10 nm technology node. The labels represent the sizes of BRAMs in each category [7].

in 1995, the on-chip memory resources were relatively modest. This device featured around 1,000 logic modules, each with approximately 20 bits of storage, and a total BRAM capacity of 2Kb. The limited storage in each LE and the small BRAM size meant that these devices were suitable only for simpler applications with minimal memory requirements. As semiconductor technology advanced, FPGA manufacturers began to increase both the number of LEs and the storage capacity within these LEs. By the time technology reached the 130 nm and 90 nm nodes, the demand for greater storage capacity led to the introduction of FPGAs with a wide range of BRAM sizes. This era saw a thousand-fold between the smallest and the largest BRAM capacity, providing designers with more options to tailor memory resources to specific application needs. At the 65 nm technology node, a significant shift occurred with the introduction of the Stratix III family, which began to incorporate LUT-based distributed RAM. This innovation allowed for smaller applications to be efficiently implemented using distributed RAM, which could be flexibly configured across the FPGA. As a result, smaller BRAM modules of 512b and 4Kb were phased out in favor of larger BRAM blocks of 9Kb and 144Kb. This transition marked a strategic move towards simplifying the FPGA architecture, reducing the complexity of memory management while providing larger, more versatile memory blocks to meet the increasing demands of applications. As technology continued to scale down to 28nm and beyond, the number of LEs in FPGAs surged into the millions, with each logic block now incorporating around 100 bits of memory. This substantial increase in logic density necessitated a streamlined memory architecture. At this stage, BRAM evolved into a single 20Kb size, which allowed for a more efficient and cleaner FPGA layout and simplified design processes [7]. While section 2.2.1 discussed the possibility of cascading smaller BRAM modules to create larger memory cells, it is important to note that this approach has its limitations. For instance, in Xilinx devices, the maximum size of a cascaded BRAM module is capped at 64KB.

In response to the escalating demands of high-speed and high-bandwidth applications, FPGA manufacturers have introduced advanced on-chip memory storage modules designed to mitigate the bottleneck often associated with external memory access. External memory systems, while capable of providing substantial storage capacity, often suffer from bandwidth limitations due to the physical constraints of data transfer channels. The bit width of these channels, which dictates the number of bits that can be transferred in parallel during a single clock cycle, becomes a critical factor in determining overall system performance. As application requirements for data throughput have increased, these bit width

limitations have increasingly become a bottleneck, hindering the ability of FPGAs to fully exploit the potential of high-performance external memory.

To address this issue, FPGA manufacturers such as Intel and Xilinx have developed new, large-scale on-chip memory modules. Intel's eSRAM system, introduced in their latest FPGA models, provides a total storage capacity of up to 45 Mbits, with a maximum operating frequency of 750 MHz [14]. Similarly, Xilinx has developed UltraRAM, which delivers up to 500 Mb of on-chip storage [42]. These high-speed memory modules are designed to meet the rigorous performance requirements of cutting-edge applications, offering both high capacity and fast access times.

Comparative Analysis of RAM Implementations

To better understand the performance trade-offs between various on-chip storage resources in FPGAs, a practical implementation of a 2048×72-bit logic RAM was conducted on a Stratix IV device using four different types of memory resources: 144Kb BRAMs, 9Kb BRAMs, LUT-RAMs, and registers. The results of this implementation are summarized in Table 2.2, highlighting the differences in area usage and operating frequency among these storage types.

Implementation	half-ALM	BRAM/9kb	BRAM/144kb	Area (mm^2)	Freq. (Mhz)
144kb BRAMs	0	0	1	0.22 (1.0×)	336 (1.0×)
9kb BRAMs	0	18	0	0.41 (1.9×)	497 (1.5×)
LUT-RAM	6597	0	0	2.81 (12.8×)	134 (0.4×)
Registers	165155	0	0	68.8 (313×)	129 (0.4×)

Table 2.2: Implementation results for a 2048×72 bits 1r+1w RAM using BRAMs, LUT-RAMs and registers on Stratix IV [7].

The 144Kb BRAMs provide an optimal solution for implementing the 2048×72-bit memory, as this memory size closely matches the capacity of a single 144Kb BRAM. When implementing the same 2048×72-bit RAM using 9Kb BRAMs, the area requirement is approximately double that of the 144Kb BRAM implementation. This increased area is due to the necessity of using multiple 9Kb BRAMs to achieve the same memory size. As discussed in Section 2.1.3, smaller BRAMs incur a relatively higher area cost for peripheral circuitry compared to larger BRAMs. However, despite the larger area footprint, the 9Kb BRAMs operate at a significantly higher frequency—approximately 1.9 times greater—compared to the 144Kb BRAMs. LUT-RAMs, which are implemented using the adaptive logic modules (ALMs) within the FPGA, provide a flexible and efficient storage option, particularly for smaller memory configurations. In the case of the 2048×72-bit RAM, the LUT-RAMs deliver a moderate performance in terms of both area and frequency. Registers, also derived from the ALMs, represent the least area-efficient option among the four storage types evaluated. Because the storage capacity of registers within a single ALM is significantly smaller than that of LUT-RAMs, implementing the 2048×72-bit RAM using registers results in a much larger area footprint.

The comparative analysis of these four memory modules underscores the importance of selecting the appropriate storage resource based on the specific requirements of the application. The 144Kb BRAMs are the most area-efficient and provide solid operating frequency performance, making them ideal for large, contiguous memory blocks. The 9Kb BRAMs, while requiring more area, excel in high-frequency applications. LUT-RAMs offer a balanced approach, suitable for mid-sized memory needs, while registers, despite their large area consumption, serve well for small, distributed storage tasks.

2.3. First In First Out Memory Structures

FPGAs have fundamentally transformed the landscape of digital design by offering unparalleled flexibility and configurability in hardware implementations. Central to the operation of FPGAs is their on-chip memory, a critical resource that supports efficient data processing and storage within the programmable fabric. Among the various uses of FPGA on-chip memory, one of the most prevalent and crucial applications is the implementation of FIFO buffers.

In modern digital systems, the exchange of data between different systems and modules is a fundamental operation. This exchange becomes particularly challenging when data arrives at high rates or

in irregular batches, especially when there is a mismatch in processing speeds between the sending and receiving modules. In these scenarios, an intermediate storage mechanism, commonly referred to as a cache, becomes essential to facilitate smooth data transfer and prevent data loss.

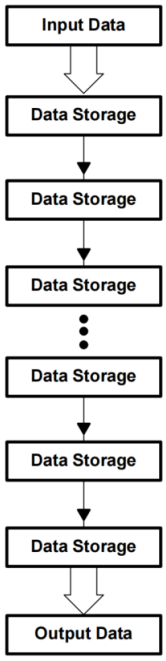


Figure 2.11: First-In First-Out Data Flow[20].

One widely used form of buffer is the FIFO buffer, as illustrated in Figure 2.11. The FIFO mechanism operates on the principle that the first data to enter the buffer is also the first to be output after a certain period, adhering to a traditional sequential execution method. Essentially, a FIFO is a two-port data buffer that operates without an external address; instead, an internal pointer is incremented to act as the data address. This design allows FIFO to perform sequential read and write operations, but it does not permit random access to specific addresses. Beyond FIFO, other types of caches, such as Last-In-First-Out (LIFO) buffers (also known as stack memory), are employed depending on the specific application requirements.

System designs often incorporate numerous components, such as processors and peripherals, each operating under different clock frequencies. These components may have their own clock oscillators, leading to variations in processing speeds across different modules. For instance, one common use case involves data acquisition from an Analog-to-Digital (AD) converter on one end of the FIFO and data transfer via a PCI bus on the other end. Despite these differences, each module's respective clock runs continuously, and when data interaction occurs between modules with mismatched clocks, computational disorders can arise. This issue is particularly pronounced when large volumes of data are involved, or when high data transfer rates are required. In such cases, Asynchronous FIFO buffers are the most effective solution. Asynchronous FIFO is a type of FIFO that allows read and write operations to occur under different clock domains. This capability is crucial for managing data transfers across different clock domains, ensuring synchronized communication.

In addition to clock domain crossing, asynchronous FIFOs are also useful for matching data interface bit widths. For example, a microcontroller might output 8-bit data, while a Digital Signal Processor (DSP) might require 16-bit data input. An asynchronous FIFO buffer can be employed to bridge this bit-width mismatch, facilitating smooth data transfer between the microcontroller and the DSP.

FIFOs also serve a critical role in caching continuous data streams, preventing data loss during feed and store operations. A typical application is in Universal Asynchronous Receiver/Transmitter (UART) communication, where data needs to be converted from parallel to serial for transmission, and then back to parallel on the receiving end. UART devices must adhere to the same transmission rate (baud rate),

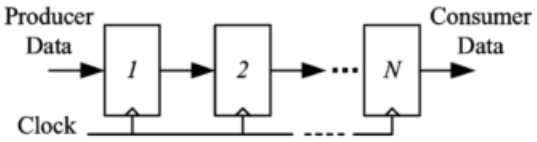


Figure 2.12: Basic structure of linear register FIFO[3].

and protocols such as start and stop bits. A FIFO buffer is simply a temporary storage area. If the CPU is capable of handling UART data in real-time, the use of a FIFO buffer might be unnecessary. However, in complex systems where the CPU is tasked with multiple operations, data handling interruptions can significantly reduce efficiency. In such cases, Synchronous FIFOs indispensable. This type of FIFOs operate under a single clock domain, meaning both read and write operations are governed by the same clock signal. It is typically used as a buffer to manage bursts of data, where data might be written to the FIFO at a high rate, followed by periods of inactivity. By setting an appropriate depth for the FIFO, it temporarily stores data during these bursts, preventing data loss and and allowing the CPU to process them in batches, thereby improving overall system efficiency.

Moreover, FIFO buffers provide a critical additional feature: the introduction of controlled latency. By intentionally delaying faster incoming signals, FIFOs can align data streams with varying speeds so that they reach their destination simultaneously. This synchronization allows the system to process multiple signals together, optimizing performance and maintaining data coherence across different modules.

FIFOs can be implemented in both software and hardware. Software FIFOs offer the flexibility of programmability, allowing their functionality to be easily modified. In contrast, hardware FIFOs require a specific physical design and layout but offer superior speed performance. The choice between software and hardware FIFO implementations depends on the specific requirements of the system, such as the need for speed versus flexibility.

2.3.1. Linear FIFO

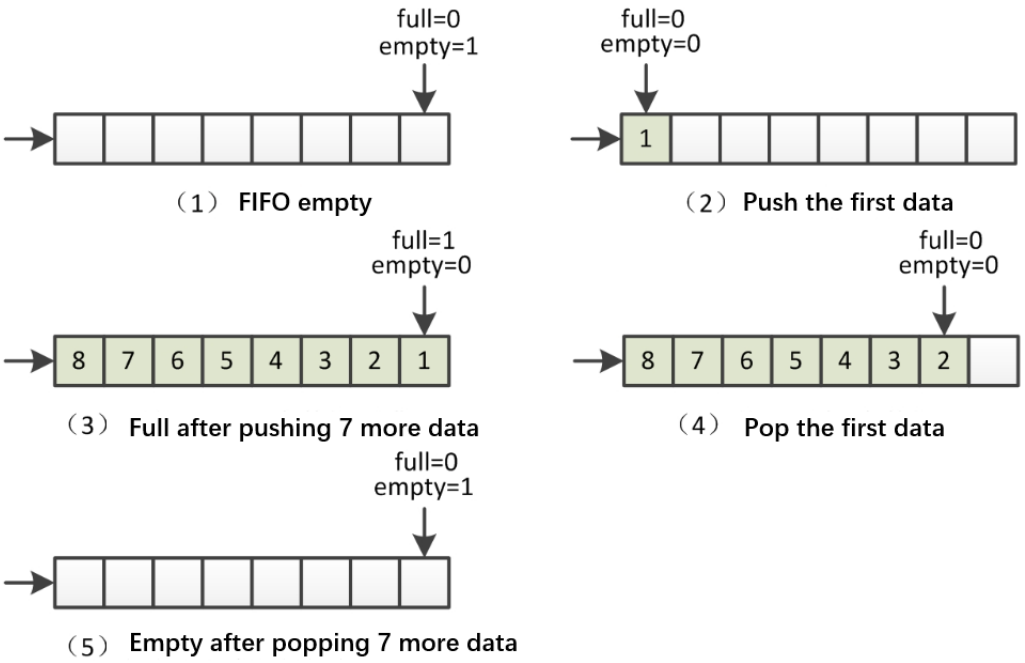


Figure 2.13: Basic flow of linear register FIFO.

Linear FIFOs represent one of the simplest and most fundamental FIFO structures, primarily consisting of a series of linearly arranged latches or registers. These FIFOs operate on a straightforward principle

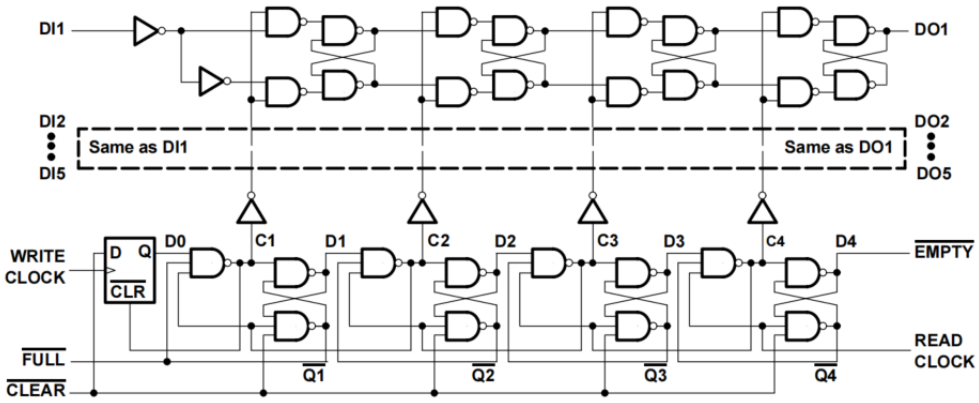


Figure 2.14: Structure of linear latch FIFO[20].

where data is pushed into the first register or latch and sequentially moves through the subsequent stages until it reaches the final stage, from which it is eventually output.

A basic example of a linear FIFO structure is the register-based FIFO, as illustrated in Figure 2.12. The data feed into the most left register and shift to the next register at every clock edge. Then, it is popped to the output pin after N clock cycles. Figure 2.13 provides a dynamic view of a shift register FIFO. Initially, the first data item is pushed into the first register. As additional data items are introduced, they are sequentially shifted through the next seven registers, resulting in a fully occupied FIFO. When the FIFO is full, the first data item is ”popped” out from the shift register, making space for subsequent data to move through the sequence. This continues until all the data has been processed and the FIFO is empty. This simple and efficient structure is widely used in applications where timing requirements are less stringent and the overhead of additional circuitry is minimal.

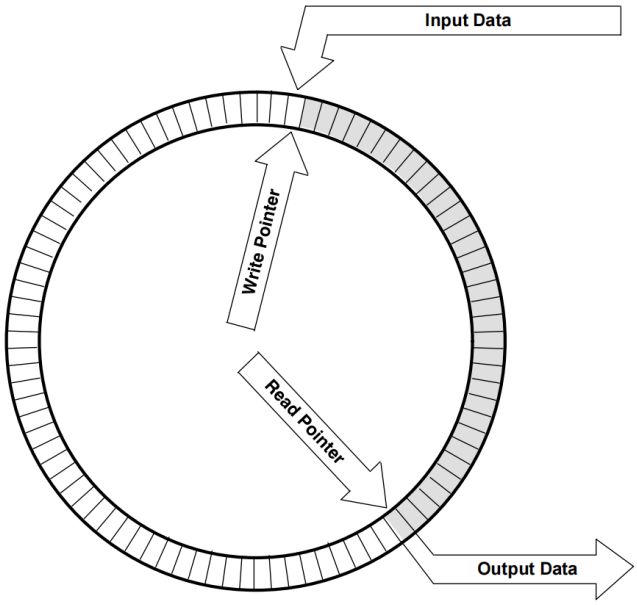


Figure 2.15: Two pointers for the circular FIFO[20].

An alternative to the register-based approach is the latch-based linear FIFO, which offers a more area-efficient design. As shown in Figure 2.14, this type of FIFO consists of an upper row of latches for data storage and a lower section that includes a clock generator to manage the input, shifting, and output of data. As shown, the upper row is composed of a series of latches, while the lower row consists of logic controlled by full and empty flags, as well as clear signals and write/read clocks. These logic elements

control each latch through an intermediate buffer. The primary advantage of latch-based FIFOs lies in their reduced area budget. Since two latches are equivalent to one register in terms of functionality, the latch-based FIFO requires fewer resources, making it a more compact solution. However, this compactness comes with a trade-off. Unlike registers, which are controlled by the rising edge of the clock signal, latches are level-sensitive, meaning they are controlled by the level of the clock signal rather than its edges. This characteristic imposes more stringent timing requirements on the system, as the timing margins for correct operation are narrower.

2.3.2. Circular FIFO

The next type of FIFO is circular buffer, also known as a ring buffer or ring queue, is a specialized data structure designed to handle continuous data streams. Unlike a traditional FIFO, which has a linear structure, the ring buffer uses a fixed-size, head-to-tail buffer that wraps around upon reaching its end, creating a circular structure.

Circular buffers are particularly well-suited for applications that involve continuous data flow. A circular buffer is a FIFO structure at its core, but with a logical arrangement that makes its linear data flow appear circular. This is typically implemented using a fixed-size array, with two pointers— a read pointer and a write pointer—controlling the operations, as shown in Figure 2.15. Read pointer points to the next data position that can be read from the buffer while write pointer points to the next position where new data can be written into the buffer. Initially, both pointers start at the zero position of the buffer. As data is written, the write pointer advances, and as data is read, the read pointer moves forward. Once either pointer reaches the end of the array, it wraps around to the beginning, thus maintaining the circular structure of the buffer.

An advanced implementation of FIFO is the dual-port SRAM-based FIFO, which can perform simultaneous read and write operations within the same clock cycle. This type of FIFO is asynchronous, allowing for independent read and write clocks. In a dual-port SRAM FIFO, the write pointer advances when new data is written, while the read pointer outputs data when the FIFO indicates it is full [3] [20].

3

Proposed Designs and Implementation

The Chapter presents the design and implementation of FIFO structures on both ASIC and FPGA platforms, focusing on two distinct FIFO configurations: the ASIC-based linear FIFO and the FPGA-based circular FIFO. It provides detailed descriptions of the designs and their comparison metrics. The Chapter also outlines the design flows for both FPGA and ASIC implementations. Additionally, in Section 3.2.2, it addresses hold time violations in ASIC designs by employing multi-tap H-tree clock tree synthesis.

3.1. Proposed FIFO Designs

This project seeks to investigate the area, performance, and power consumption of ASIC-based linear FIFOs compared to FPGA-based circular FIFOs.

FPGA Implementation

On the FPGA side, a dual-port FIFO is designed and implemented on a Xilinx 28nm Virtex 7 series FPGA device. The FIFO supports simultaneous read and write operations in the same cycle, controlled by separate address pointers for read and write functions. The design leverages three types of on-chip memory resources available in FPGAs:

1. **Register-based memory**: Utilizes flip-flops in logic blocks to store data;
2. **LUT-based memory**: Utilizes LUTs in logic blocks as memory elements;
3. **SRAM-based memory**: Employs FPGA Block RAM.

The circular FIFO design is implemented in Verilog with varying bit widths (200, 300, and 400 bits) and depths (1Kbits, 1.5Kbits, 2Kbits, 2.5Kbits, and 3Kbits). Each FIFO utilizes two address pointers—one for read control and one for write control. The ”RAM_TYPE” directive in the HDL code is employed to infer memory across the three different resources. Simulation, synthesis, and implementation are performed using Vivado version 2022.2. The target FPGA device is the Xilinx Series-7 xc7vx1140tflg1926_2, which features a relatively large amount of BRAM (1880 blocks). The designs are analyzed for their impact on area, power, and performance across different configurations.

ASIC Implementation

For the ASIC implementation, a linear FIFO is designed using a shift register structure, implemented in a TSMC 40nm process technology. In this design, data is written sequentially to the FIFO at the input side every cycle, while the oldest data is shifted out from the output side. The linear FIFO’s performance is directly linked to the depth and bit width, with the simplicity of the shift register architecture offering insights into the minimalistic design trade-offs.

Similar to the FPGA implementation, the ASIC FIFO design supports bit widths of 200, 300, and 400 bits, and depths of 1Kbits, 1.5Kbits, 2Kbits, 2.5Kbits, and 3Kbits. The primary focus is on evaluating the performance, area, and power consumption of the shift register-based FIFO design.

Comparative Analysis

Both the FPGA and ASIC implementations are systematically analyzed across multiple configurations: with varying bit widths and FIFO depths, and with circular structure for FPGA and linear structure for ASIC. The study focuses on three key performance metrics:

1. Performance: The throughput of the FIFO in terms of read and write operations per cycle, as well as the overall speed of the design;

2. Area: The physical area consumed by the FIFO in each implementation, evaluated across different bit widths and depths. For FPGA, this includes the utilization of flip-flops, LUTs, and BRAM, while for ASIC, the area is based on the transistor count and layout area in the 40 nm process;

3. Power Consumption: The power required by the FIFO in each configuration, with a focus on how different architectures and bit widths affect power efficiency.

The project aims to provide a detailed comparison between the FPGA and ASIC implementations, identifying the scenarios where each approach excels. The FPGA-based circular FIFO, with its dynamic address pointers, is expected to offer greater flexibility but may come at the cost of increased area overhead and power consumption for the additional control logic. On the other hand, the ASIC-based linear FIFO, due to its simpler architecture, may provide advantages in terms of area and power, particularly in applications with fixed data flow patterns.

Ultimately, the goal is to assess the trade-offs between these two architectures and determine the most suitable design approach for specific applications, such as high-speed data buffering or low-power embedded systems. By exploring these various configurations and architectures, the project will highlight the strengths and limitations of both FPGA and ASIC-based FIFO designs.

3.2. FPGA and ASIC implementation Flows

This section describes the design flows for both FPGA and ASIC FIFOs, from Verilog coding through to synthesis and implementation. The ASIC design flow additionally incorporates the H-tree multi-tap technique to address hold time violations.

3.2.1. FPGA Design Flow

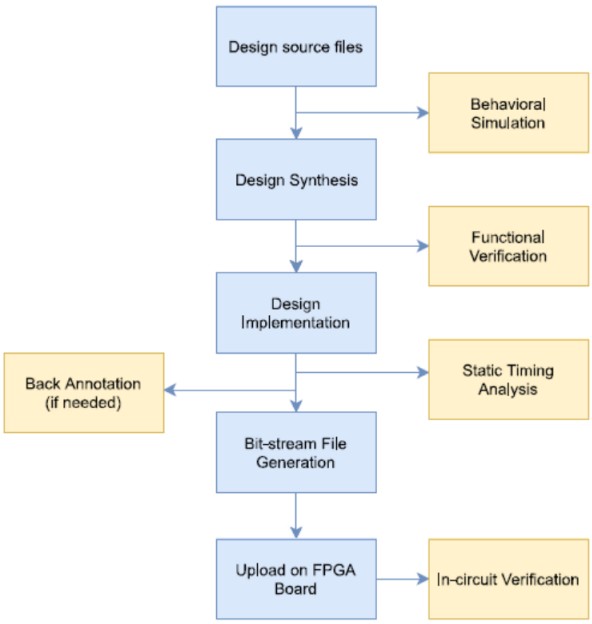


Figure 3.1: Design flow for FPGA implementation.

The proposed design is implemented on a Xilinx Series-7 FPGA (xc7vx1140tflg1926-2), which features 712,000 LUT elements, 1,424,000 flip-flops, and 1,880 BRAMs. This model, with its abundant resources, particularly the large number of BRAMs, is well-suited for BRAM-based FIFO implementa-

tions, allowing for better performance. The FPGA design process is relatively straightforward and can be completed using automated tools, as illustrated in Figure 3.1. The process begins with writing the RTL (Register Transfer Level) description of the design in Verilog. Since the design involves implementing the FIFOs on three different on-chip resources, the type of storage resource is inferred by using the *RAM_TYPE* directive in the Verilog code. The next step is functional testing, where the Verilog code is verified using Vivado's *behavioral simulation* to ensure it meets the expected functionality. Once the design passes this simulation, timing constraints are written to guide the synthesis process. These constraints define parameters such as clock speed, input and output delays, and setup/hold times, the detail is shown in Section 3.2.2.

After defining the timing constraints, the RTL design is synthesized into a gate-level netlist by simply clicking the *Synthesis* button in Vivado. The synthesis strategy used is Vivado Synthesis Default, which leads to moderate results. The synthesis process converts the high-level RTL design into a lower-level gate representation, which is closer to the actual hardware implementation. Once synthesized, the design undergoes another round of *functional verification* with the same testbench to ensure that the synthesized gate-level netlist still meets the functional requirements.

If the design passes this verification stage, it proceeds to the *implementation* phase, where the synthesized netlist is mapped to the FPGA's physical resources to produce the final layout. The strategy used here is Vivado Implementation Defaults. During implementation, the tool optimizes the placement of logic elements, routing, and clock distribution to meet the specified performance goals.

Although the design will not be deployed on a physical FPGA device in this project, *back annotation* is used to obtain more accurate estimates of dynamic power consumption by adding the switch activity in real situation during synthesis. This step, discussed in more detail in Section 4.2.3, applies real timing data from the FPGA's physical implementation to the power analysis, providing more realistic power consumption figures. The detailed steps of this flow will be explained further in Section 3.2.2, covering the synthesis, verification, and implementation processes in depth.

3.2.2. ASIC Design Flow

An ASIC is a custom-designed IC tailored for a specific application or scenario, as opposed to general-purpose ICs such as RAM, ROM, SRAM, or microprocessors, which fall under the category of Application-Specific Standard Products (ASSPs). The structure of an ASIC can vary significantly. For instance, it may consist of a block generated from a general-purpose design combined with custom logic or serve as an interface between two general-purpose ICs. Modern ASICs typically contain millions of transistors, making it infeasible to manually enter schematic views and layouts. Consequently, an Electronic Design Automation (EDA)-dependent ASIC design flow is essential for efficient development.

The ASIC design process is categorized into two main approaches: full-custom and semi-custom. In the full-custom approach, developers must design entirely new libraries if no suitable standard libraries exist or if the available libraries cannot meet specific requirements for speed, power efficiency, or area constraints. Although this method can significantly improve performance and design efficiency, it requires more time and effort to solve complex design challenges. In contrast, the semi-custom approach leverages pre-designed cells and macros from existing standard libraries, significantly reducing design time and cost [32]. In the project, the semi-custom design flow is chosen by using the TSMC 40 nm standard library.

Design Flow for the Proposed ASIC Design

The proposed designs are relatively simple and small single-block ASIC FIFO. Figure 3.2 illustrates the streamlined flow used in this project. The design begins with the functionality of the FIFO described in Verilog. RTL typically describes synchronous logic, which consists of registers that store the current state, combinational logic that determines the next state, and clocks that control state updates. To ensure that the design meets the target functionality, a behavioral simulation is performed in Vivado. After verifying functionality, the design moves into synthesis.

After successful verification, the next phase is synthesis, where the verified RTL design is translated into a gate-level netlist. At the RTL level, only the registers and the combinational logic are described, but during synthesis, the RTL description is mapped onto the cells and macros from the standard library. The EDA tools generate a gate-level Boolean equation that contains all the logic gates and

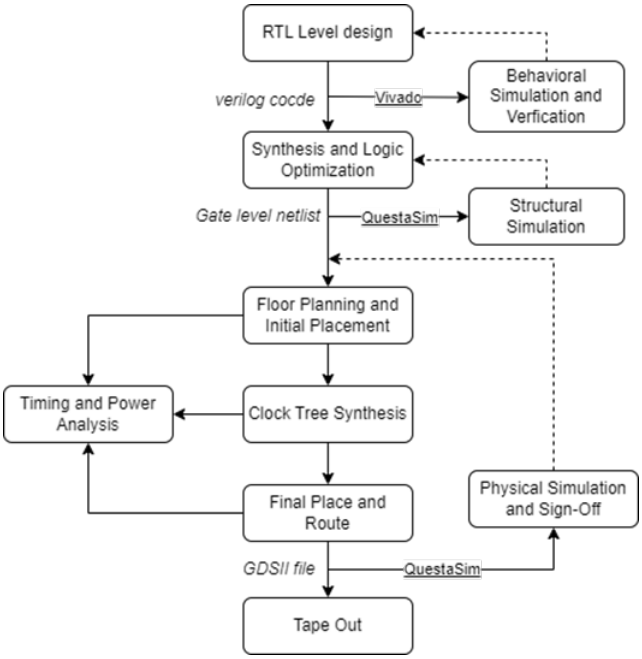
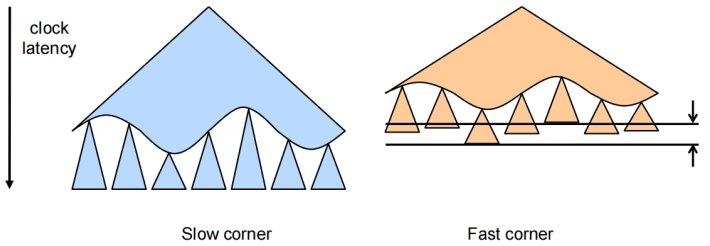


Figure 3.2: Design Flow for the Proposed ASIC Design.

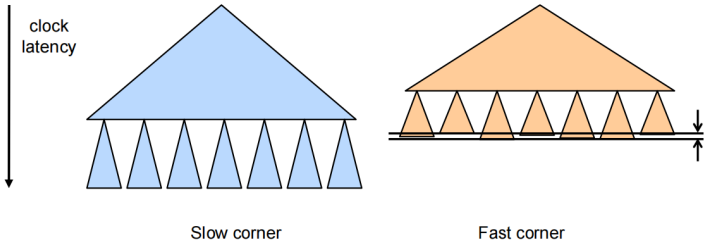
their interconnections. This step is critical for balancing the trade-offs between performance, power consumption, timing, and area, as these factors impact all subsequent back-end design stages. The design is synthesized using Cadence Genus Solution with .lib file(Liberty File) and .sdc file(Synopsys Design Constraints File). The .lib file contains information about the functionality, timing, power, and other physical characteristics of standard cells. The .lib file tcbn40lpbwpwc is used. It infer to a TSMC 40nm low power standard. A timing constraints file in .sdc format is created to define critical timing parameters such as clock characteristics (waveform, operating frequency, transition times, and uncertainty) and port constraints (input delay, drive strength, output delay, load, etc.). During synthesis, RTL code is mapped to gate-level components, and the switch activity annotation is applied to obtain a more accurate estimate of dynamic power consumption, detail in Section 4.2.3. The synthesis process outputs a gate-level netlist, a Standard Delay Format (SDF) file (which captures timing data as per IEEE standards), and an updated .sdc file. The next stage involves running a structural simulation in QuestaSim, where the synthesized gate-level netlist is tested using a testbench same to the one used for behavioral simulation. This step verifies that the synthesized design still functions as expected.

Once the netlist is verified, physical implementation begins using Cadence Innovus Solution. The stage takes the synthesized netlist, along with a .lef (Library Exchange Format file), which contains physical information such as layout and design rules, an MMMC (multi-mode multi-corner) file for static timing analysis under various corner conditions, and the synthesized .sdc file. The physical implementation of the design starts with floorplanning, where the size of the chip and the distribution of the macros are determined, followed by power planning to ensure adequate power distribution across the chip. The next step is placement, where standard cells are positioned within the floorplan. Clock Tree Synthesis (CTS) is then performed to distribute the clock signal from the global clock port to all the clock pins for cells and macros, ensuring minimal clock skew across the design. The final stage of the implementation is routing, where metal connections are made between the pins of all cells in accordance with the gate-level netlist. This step ensures that timing constraints, Design Rule Checks (DRC), and Layout Versus Schematic (LVS) rules are met. Successful routing completes the physical design, bringing the chip closer to fabrication.

Another critical stage in the ASIC design flow is Static Timing Analysis (STA). STA focuses on verifying the timing characteristics to ensure that the design's timing constraints are met under various operating conditions. STA involves analyzing the timing paths in the circuit to ensure that signals propagate within acceptable time windows. It assesses the design under different process, voltage, and temper-



(a) Clock latency with regular CTS.



(b) Clock latency with H-tree CTS.

Figure 3.3: Clock latency cross slow and fast corner[9].

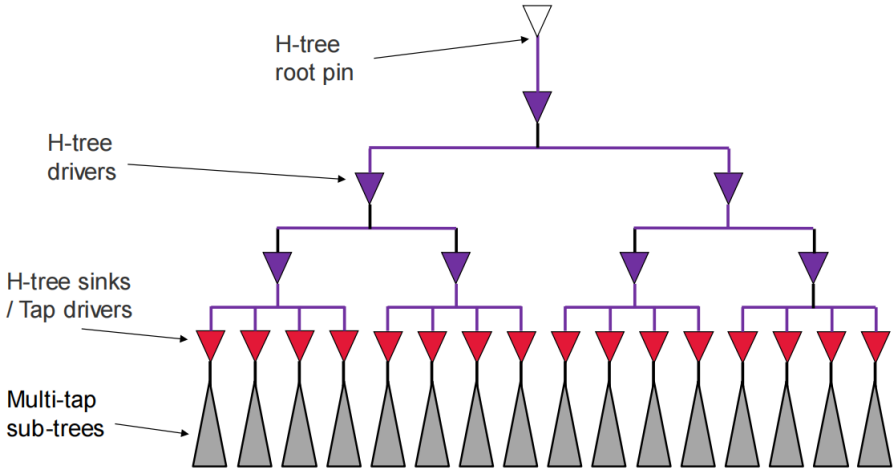


Figure 3.4: Architecture of multi-tap H-tree CTS[9].

ature (PVT) corners, capturing both the best-case and worst-case scenarios for chip operation, these conditions are defined in the MMMC file. For instance, the best-case scenario typically involves a fast process, low temperature, and high supply voltage, conditions that are conducive to faster signal propagation. This scenario is critical for identifying hold time violations, which occur when data arrives at a flip-flop too early. Conversely, the worst-case scenario accounts for slow process variations, high temperature, and low supply voltage—factors that can lead to slower signal propagation. This is used to detect setup time violations, where data may not arrive at the flip-flop within the required time window, potentially causing functional errors. STA is not a one-time process but is conducted at various stages of the physical design flow. Initially, it is run after the gate-level netlist is generated. As the design progresses through placement, CTS, and routing stages, STA is performed again and again to ensure that timing constraints are still satisfied after these modifications. These incremental checks help identify and resolve timing violations before they cascade into larger issues that could compromise chip functionality or performance [32].

The final product of this implementation process is a layout with GDSII File Format, along with the updated .sdf and timing files. The design is then run through QuestaSim again for physical simulation, which ensures that the functionality of the final physical design is intact. This process confirms that

the design operates correctly under real-world conditions, closing the loop from the initial behavioral simulation to the final physical realization.

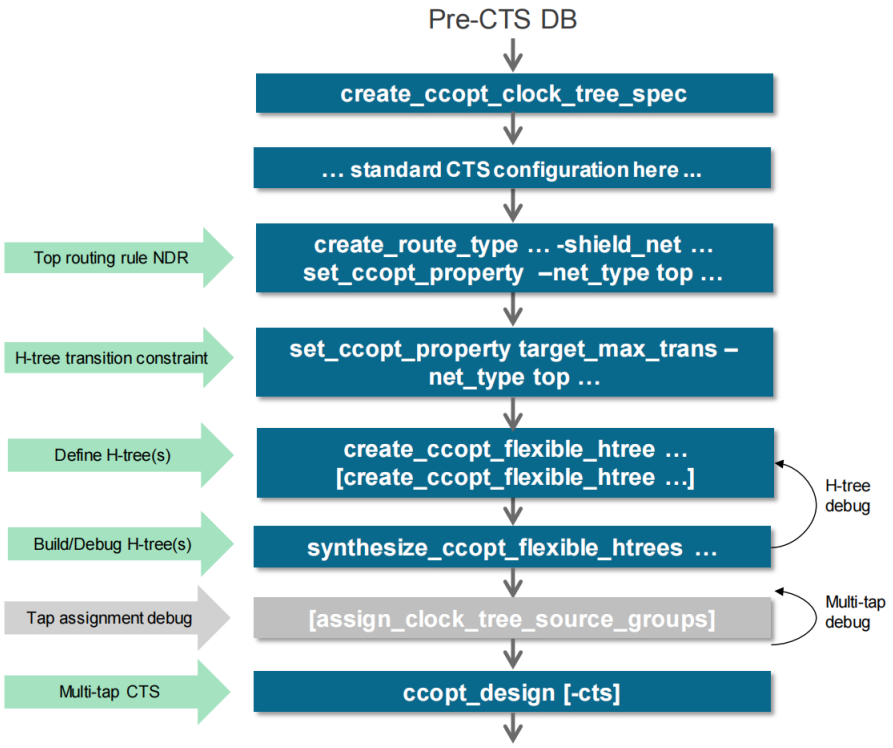


Figure 3.5: Design Flow for H-tree CTS with multi-tap[9].

Flexible H-Tree Multi-tap CTS Implementation

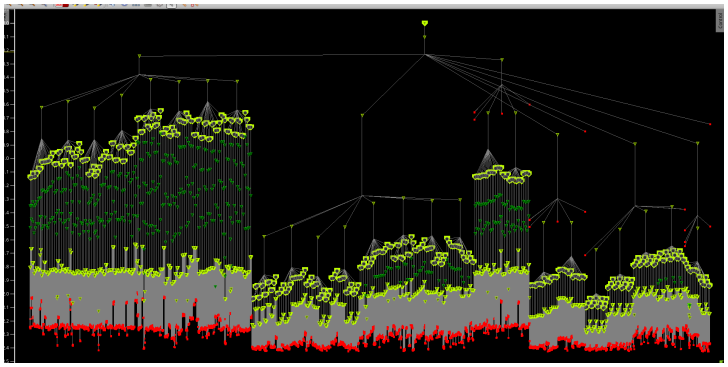
After designing and implementing the ASIC FIFO, timing reports revealed that a hold timing violation occurred on the register-to-register path when the memory cell size of the FIFO exceeded 600 Kbits. This issue arises because the shift register structure does not include any combinatorial logic between each register, leading to rapid data propagation through the registers. When the number of registers reaches a certain threshold, the likelihood of a hold time violation increases. To address this, a Flexible H-tree Multi-Tap CTS approach was adopted instead of the standard CTS method.

Unlike regular CTS, which distributes the clock signal by branching out from one root to the sinks like a real tree, the Flexible H-tree employs a symmetric H-tree structure. It begins at a central point, branching out in four directions to form an H-shape, and further expands by creating smaller H-lines from each of the four points. This approach ensures more balanced clock distribution. Figure 3.3a shows the timing simulation results for both slow and fast corners under regular CTS. The primary goal of this method is to evenly distribute the clock signal from the root to the sinks in the slow corner scenario. By inserting different cells, adjusting cell sizes, and fine-tuning the wire lengths, delays in the upper and lower paths can be equalized. However, in fast corner conditions, these adjustments have varying scale factors, leading to unequal delay reductions in different paths and resulting in clock skew between the sinks.

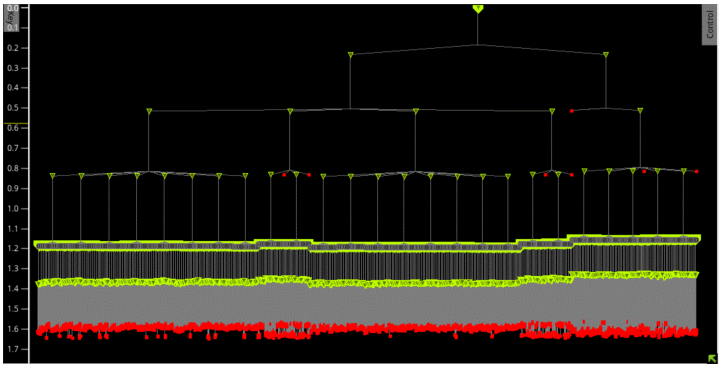
In contrast, Figure 3.3b illustrates the timing results under the Flexible H-tree approach. Due to its electrically symmetric nature, the clock delays across different paths scale uniformly in both slow and fast corners, minimizing clock skew even under fast corner conditions. The ”flexibility” of the Flexible H-tree refers to its ability to distribute the clock signal using electrically symmetric buffering and balanced wire lengths, without requiring strict geometric symmetry. This flexibility allows it to adapt effectively even with floorplan and placement constraints.

The H-tree method alone, however, cannot achieve all necessary functions. As depicted in Figure 3.4, it must be combined with multi-tap CTS to accomplish the final distribution of clock signals. In this setup,

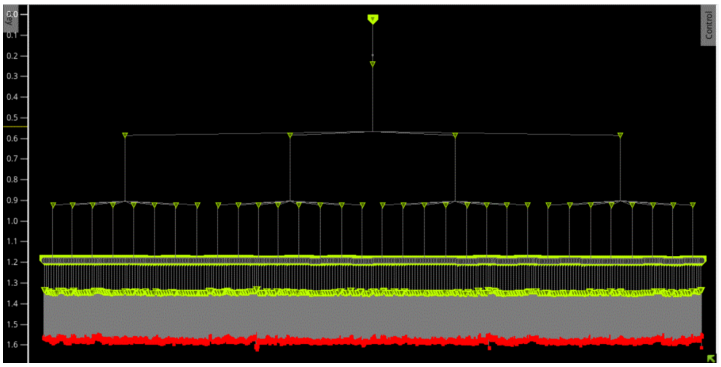
the root of the H-tree could be a clock generation cell or an external port. The H-tree distributes the clock signal to intermediate sinks(marked as read), which then act as roots for multi-tap CTS sub-trees, eventually distributing the signal to the clock pins of the registers, which shown in gray [9]. Figure 3.5 outlines the multi-tap H-tree flow, illustrating how the routing rules and the structure of the H-tree are defined as key parameters. The process begins with the generation of the existing clock tree specifications, accompanied by the configuration of standard CTS parameters. Following this, routing rules are established in the third step to precisely define the implementation of H-tree routing. Next, transition constraints are introduced to enhance the realism of the H-tree CTS. In the subsequent step, the structure of the H-tree is defined, such as the configuration and number of sinks. Finally, the synthesis and realization of the multi-tap H-tree are completed.



(a) Single-width routing metal and 20×20 H-tree sink grid.



(b) Double-width routing metal and 20×20 H-tree sink grid.



(c) Double-width routing metal and 16×16 H-tree sink grid.

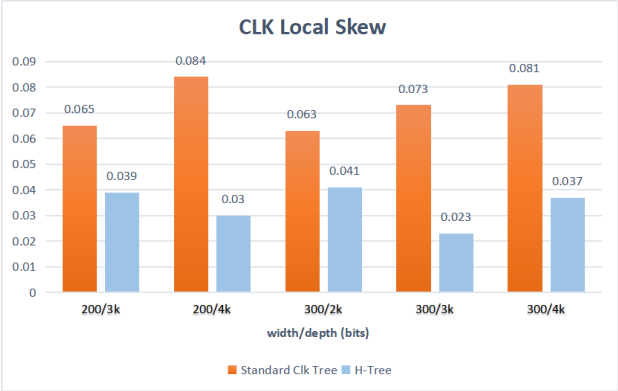
Figure 3.6: Clock latency distribution across different H-tree settings.

Figures 3.6a and 3.6b display the results of clock tree distribution and clock latency for a FIFO with a bit width of 400 bits and a depth of 1Kbits. In Figure 3.6a, the routing concatenation width is set to the standard width, while in Figure 3.6b, it is doubled. Both H-tree structures are configured as a 20x20 grid. The figures illustrate the tree structure originating from the clock resource root. After three levels of H-tree distribution (highlighted in yellow), the clock signal is buffered at each sink (shown in green)

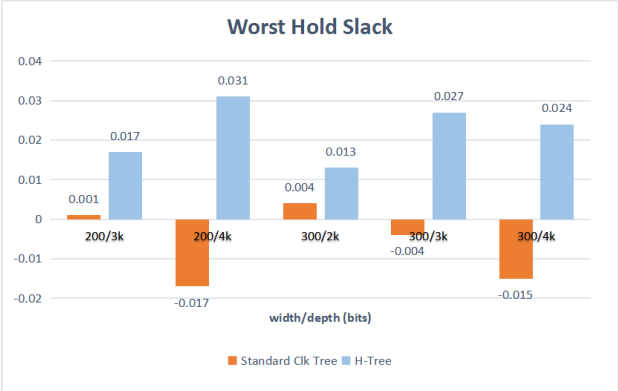
before finally connecting to the clock pins of memory cells (marked in red). The vertical length of the connection lines represents the clock latency between two nodes.

In Figure 3.6a, the H-tree distribution is highly uneven in terms of timing, as evidenced by the significant vertical displacement among the H-tree sinks. This results in large disparities in the positions of the memory cell clock pins (red dots). Since nearby memory cells are typically clustered together, greater vertical displacement between adjacent red dots indicates a larger local clock skew, increasing the likelihood of hold time violations. In contrast, Figure 3.6b, with twice the standard routing width, shows a much more uniform H-tree distribution. The H-tree sinks are nearly aligned horizontally, resulting in minimal differences in the positions of the memory cell clock pins. These results suggest that doubling the routing width significantly improves CTS performance. This improvement occurs because, as the size of the FIFO increases, a single clock root must distribute the signal to a growing number of registers. A wider routing width enhances the efficiency of clock signal propagation, reducing skew and improving overall timing performance.

Figure 3.6c further investigates the impact of H-tree structure on CTS performance. In this scenario, the routing rule's metal width is doubled, and the H-tree sink grid is reduced to 16x16. Compared to Figure 3.6b, the H-tree structure in Figure 3.6c is more uniform, with nodes at each stage almost perfectly aligned horizontally. Notably, in Figure 3.6b, there are redundant nodes (highlighted in red among the yellow ones) from the second and third expansions (third and fourth rows) of the root that do not extend further. This is due to the 20x20 sink grid, which cannot fully unfold symmetrically in the layout. However, in Figure 3.6c, the clock tree achieves perfect symmetry in the layout, resulting in each node handling approximately the same load. Consequently, the clock latency distribution is more uniform than in Figure 3.6b, despite the lower number of sinks. This highlights that ensuring symmetry in the placement of sinks within the H-tree is crucial for maximizing the advantages of H-tree CTS.



(a) Clock local skew with unit in ns.



(b) Worst hold slack with unit in ns.

Figure 3.7: Optimization of hold time with multi-tap H-tree CTS.

Finally, Figures 3.7a and 3.7b present the timing outcomes for the design optimized using multi-tap

H-tree CTA. The FIFOs used have bit widths of 200 bits and 300 bits, with depths ranging from 2Kbits to 3Kbits. These configurations were chosen due to the significant hold time violations observed when multi-tap H-tree CTS was not applied. Figure 3.7a illustrates the local clock skew, which represents the largest skew between two adjacent cells. As the local clock skew increases, it indicates that the clock signal takes longer to propagate from one memory cell to the next, leading to a higher likelihood of hold time violations as data moves quickly between the cells. Without optimization, the local clock skew for a standard clock tree ranged from 0.063 ns to 0.084 ns, resulting in a worst-case hold slack of -0.017 ns, as shown in Figure 3.7b. This negative hold slack can lead to functional errors. The blue bars in the figures show the results after applying multi-tap H-tree CTS. The local clock skew dropped below 0.041 ns across all configurations, indicating that the clock signal was distributed more evenly among the memory cells. Adjacent cells were connected in a radial pattern to the nearest H-tree sink, reducing clock signal propagation delays between them. Consequently, the worst-case hold slack improved, turning all negative values into positive values above 0.013 ns, as seen in Figure 3.7b. This optimization successfully transformed the worst hold slack from negative to positive, effectively resolving the hold time violation issue.

4

Data Analysis Models and Results

This Chapter presents a comparative analysis of the performance, power consumption, and area metrics of two designs, evaluated across different technology nodes—40 nm and 28 nm—and implemented on different devices, namely FPGAs and ASICs. Given that feature size significantly impacts the latency of digital circuits, it is essential to consider that improvements in process technology generally result in reduced latency. Therefore, conversions are necessary to make the results across these different nodes comparable, see Section 4.1. Section 4.2 introduces the models used for transforming results across different device types. Section 4.3 presents a final comparative analysis of the result metrics of two designs.

4.1. Comparing Across Technology Nodes: Fan Out of 4

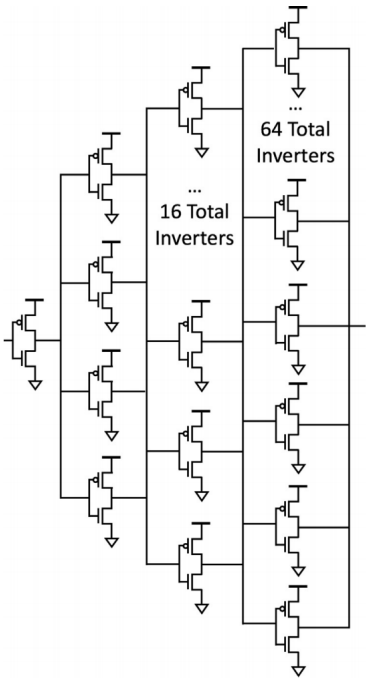


Figure 4.1: The architecture of the fan out of 4 model with 4-times-minimum-size CMOS inverters [34].

The earliest scaling equations that describe the relationships between area, delay, power, and energy are based on the ratios of transistor dimensions and supply voltages used in the designs. These coefficients are derived from fundamental circuit equations, with feature size and voltage being scaled simultaneously [31] [39]. However, as technology has advanced, particularly below the 45 nm node

using high-k dielectrics and below 20 nm with multi-gate configurations, several factors—such as short-channel effects, process variations, and leakage currents—have become increasingly influential. Consequently, these processes cannot be accurately characterized by simply scaling geometrical parameters (e.g., gate width, length, and oxide thickness) and voltage parameters (e.g., supply voltage and threshold voltage) [37] [6] [24] [25]. This section introduces a model capable of accurately predicting scaling factors for different process characteristics without requiring detailed netlists.

For area scaling, direct comparisons between specific parameters across different technology nodes are employed to derive accurate scaling factors. Regarding timing and energy consumption, a model based on simulation results from the SPICE tool at varying process nodes and supply voltages is utilized to obtain reliable scaling factors. A suitable hardware model for simulation is essential to ensure a fair comparison of different hardware implementations. This can be achieved using an inverter chain, where each inverter is followed by four additional inverters—a configuration known as Fan Out 4 (FO4). In other words, if a circuit has a delay or energy consumption equivalent to X FO4 inverters in a particular technology process, a design in a different feature size would exhibit roughly the same performance and energy cost as X FO4 inverters under this condition [36].

Minimum FeatureSize (nm)	Metal I HalfPitch (nm)	(4 T) Logic GateSize (µm2)
180	230.0	57.000
130	150.0	10.400
90	90.0	5.200
65	68.0	2.600
45	59.0	2.100
32	45.0	0.710
20	32.0	0.350
16/14	40.0	0.248
10	31.8	0.157
7	25.3	0.099

Table 4.1: Geometric values of the three parameters across different technology nodes[34].

Figure 4.1 illustrates the inverter chain, composed of 4-times-minimum-size CMOS inverters under the test technology node. The chain begins with a single inverter, followed by four inverters, and continues with outputs connected to 16 inverters, then 64 inverters. The 16 inverters in the third stage are used for sampling, while the first and second stages generate the input waveform, and the final stage inverters serve as the load [19] [34].

4.1.1. Area scaling factor

		Starting Node										
		180 nm	**130 nm**	**90 nm**	**65 nm**	**45 nm**	**32 nm**	**20 nm**	**16 nm**	**14 nm**	**10 nm**	**7 nm**
Desired Node	**180 nm**	1	0.34	0.15	0.08	0.053	0.025	0.011	0.01	0.0093	0.0055	0.0032
	130 nm	2.9	1	0.44	0.23	0.16	0.072	0.033	0.03	0.027	0.016	0.0092
	90 nm	6.6	2.3	1	0.53	0.35	0.16	0.075	0.067	0.061	0.036	0.021
	65 nm	12	4.3	1.9	1	0.66	0.31	0.14	0.13	0.12	0.068	0.039
	45 nm	19	6.4	2.8	1.5	1	0.46	0.21	0.19	0.17	0.1	0.059
	32 nm	40	14	6.1	3.3	2.2	1	0.46	0.41	0.38	0.22	0.13
	20 nm	88	30	13	7.1	4.7	2.2	1	0.89	0.82	0.48	0.28
	16 nm	99	34	15	7.9	5.3	2.4	1.1	1	0.91	0.54	0.31
	14 nm	110	37	16	8.7	5.8	2.7	1.2	1.1	1	0.59	0.34
	10 nm	180	63	28	15	9.8	4.5	2.1	1.9	1.7	1	0.58
	7 nm	320	110	48	25	17	7.8	3.6	3.2	2.9	1.7	1

Table 4.2: Area scaling factors using geometric mean of area values given by the three parameters from 4.2 [34].

Three key parameters are selected to model area scaling across feature sizes: minimum feature size, the half-pitch of Metal 1, and the size of a 4T logic gate. The minimum feature sizes and Metal 1 pitch values are squared to derive area values from single length dimensions. The measured values and their geometric averages for these three sizes, presented in Table 4.1, are depicted in Figure 4.2 with equal weighting. Each value is normalized to its 7 nm node size, allowing all data to be plotted on a single graph. The resulting scaling factors for area are shown in Table 4.2, derived from the geometric means

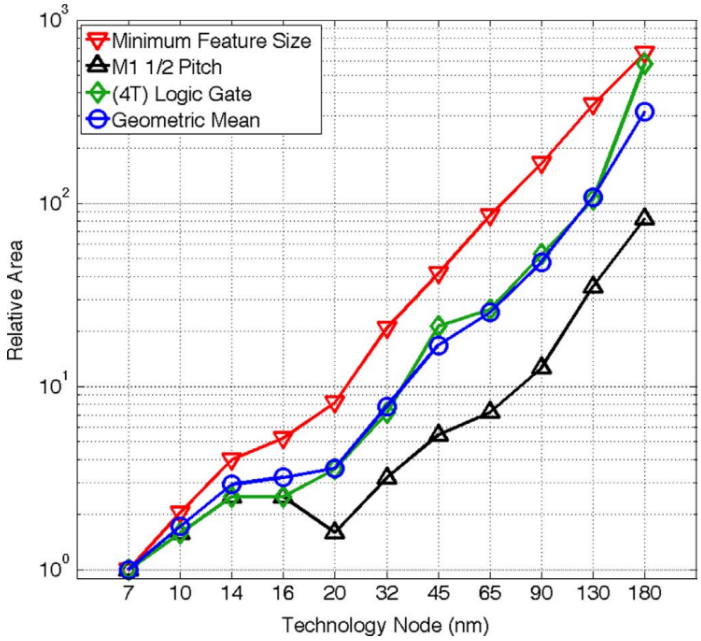


Figure 4.2: The three area parameters across different technology process nodes from Table 4.1 [34].

in Figure 4.2. The parameters in the table are derived from simulations and summaries of these three parameters across different technology nodes, revealing their approximate conversion relationships. To scale the area, identify the starting and target nodes in the table, and apply the corresponding factor to the area of the starting node to estimate the area under the desired technology node [34].

In this work, as the designs are based on 40 nm and 28 nm nodes, an approximation is made using values from the 45 nm, 32 nm, and 20 nm nodes, with the ratio of their feature sizes applied to generate the necessary area factor. Thus, for the scenario where the starting node is 40 nm and the target node is 28 nm, the appropriate area factor is approximately 0.47, indicating that the design area at 28 nm is roughly 47% of the area at 40 nm. The Figure 4.3 illustrates area for the 28nm ASIC design that scaled from the 40nm design. The three colors represent three different widths: blue for 100 bits, orange for 300 bits, and gray for 400 bits. It can be observed that as the FIFO bit width and depth increase, the area of the ASIC design also grows accordingly. By examining the data for the same depth across different widths, or for the same width across different depths, it is evident that the area increases linearly with both depth and bit width.

4.1.2. Delay, energy and power scaling factors

Using the FO4 model depicted in Figure 4.1, scaling factors for delay, energy, and power can also be determined. The simulation for the 45 nm and 32 nm nodes is based on high-k dielectric transistors, while the 20 nm and 7 nm nodes utilize multi-gate transistors. The predictive technology model also accounts for high-performance (HP) and low-power (LP) configurations. HP devices offer faster switching times due to lower threshold voltages but at the cost of increased leakage power, making them more susceptible to noise. Conversely, LP transistors, designed with higher threshold voltages to enhance noise immunity, operate more slowly and reduce leakage current.

The average signal propagation time is first determined by simulating a randomly selected inverter from the third column of the FO4 inverter chain in Figure 4.1. This result, illustrated in Figure 4.4, is obtained by measuring the delay between the time a varying signal passes through the midpoint at the input and the midpoint at the output. As shown, the delay decreases as the supply voltage increases, and with advancements in technology processes, the delay further reduces. Notably, this decreasing trend is more pronounced in the lower supply voltage range. Furthermore, under the same process, high-performance devices exhibit smaller delays compared to low-power devices. Next, the energy required to toggle the signal in the inverter is measured, as depicted in Figure 4.5. As supply

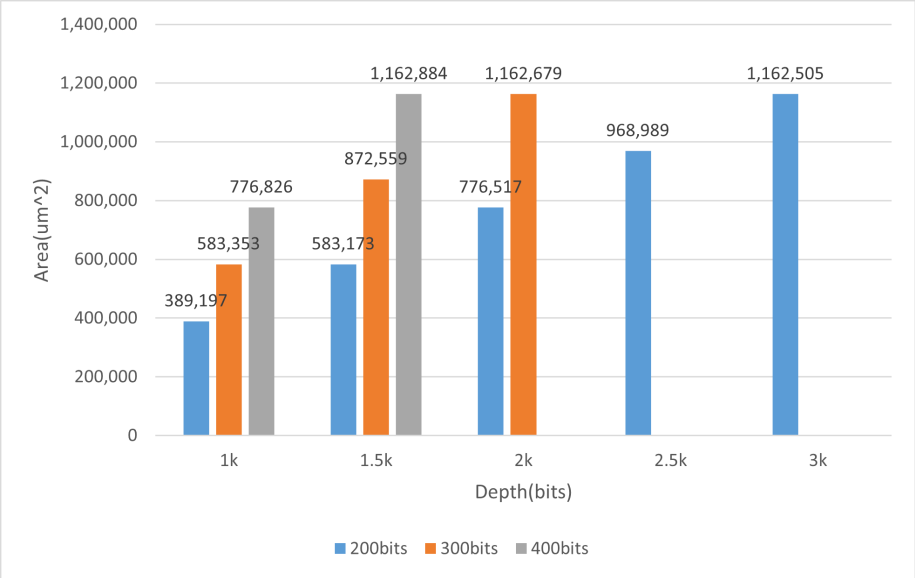


Figure 4.3: Area for the 28nm ASIC design by using the area scaling factor.

voltage or technology process increases, energy consumption also increases, following a nearly linear trend. Low standby power multi-gate (LSTP MG) devices exhibit slightly lower energy consumption compared to high-performance multi-gate (HP MG) devices under the same process; however, this difference is not observed when using high-k materials. Finally, the average power consumption of the inverter over a 1000 ps period, accounting for leakage current effects, is measured and presented in Figure 4.6. It can be observed that with increases in supply voltage and technology process increase, average power consumption also gradually rises. HP MG devices consistently demonstrate higher power consumption compared to LSTP MG devices under the same process. However, this trend is less pronounced between high-k low power and high-performance devices. These graphs reveal a roughly linear relationship for the same device type across different processes and voltages, as indicated by consistent color coding. However, no strong linear correlation is observed between different device types, emphasizing the significant influence of technology type on the observed trends.

The next step involves processing the simulation data to develop a reliable approximation model. This model is developed using polynomial approximation, which estimates the original function as a polynomial derived from a Taylor expansion. While increasing the polynomial's degree generally enhances approximation accuracy, it also introduces greater complexity and reduces computational speed[10]. Therefore, only a limited number of terms are retained to balance efficiency and accuracy. The objective is to ensure that the polynomial approximation closely mirrors the original function while preserving overall precision. The coefficient of determination is employed to measure the similarity between the approximated and original functions, with a value of 1 indicating a perfect match. The model strives for a coefficient of determination greater than 0.95 by selecting an appropriate order for the approximating polynomials. The final result is a third-order polynomial that fits the delay data and two second-order polynomials for the energy and power data, all achieving determination coefficients exceeding 0.95.

This approximation method yields excellent results, and although these data are not explicitly depicted in the figures, they closely align with the actual measured data from simulations, as demonstrated in Figures 4.4, 4.5, and 4.6, discussed earlier.

The equations below represent the third-order polynomial for the delay factor and the second-order polynomials for the energy and power factors, respectively. Here, V denotes the supply voltage, with corresponding parameters listed in Table 4.3 based on the selected technology class and process.

$$DelayFactor = a_{d3}V^3 + a_{d2}V^2 + a_{d1}V + a_{d0} \tag{4.1}$$

$$EnergyFactor = a_{e2}V^2 + a_{e1}V + a_{e0} \tag{4.2}$$

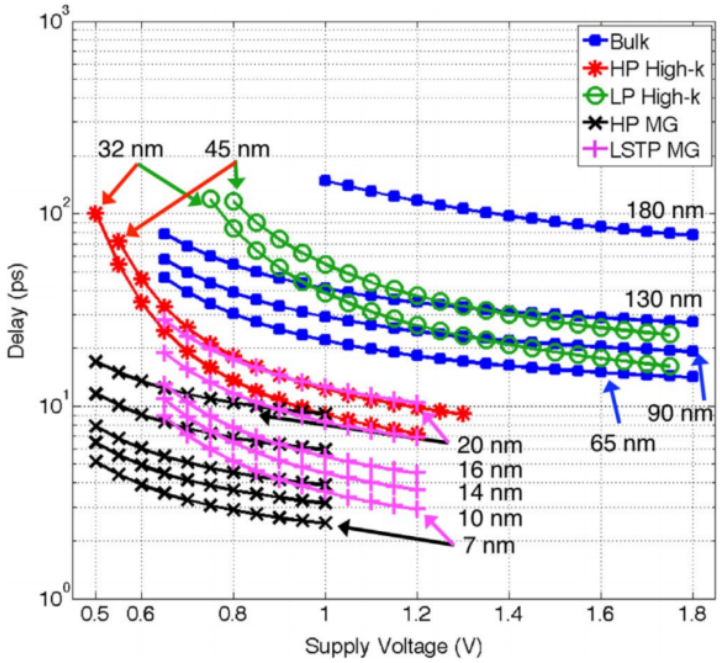


Figure 4.4: The average signal propagation time through one inverter in the middle of the FO4 inverter chain for different technologies [34].

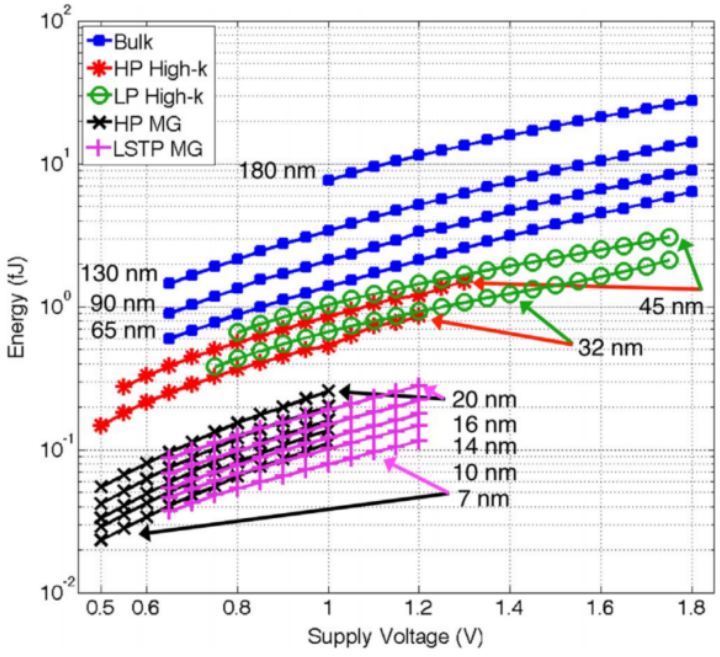


Figure 4.5: The energy required to toggle the signal in the inverter in the middle of the FO4 inverter chain for different technologies [34].

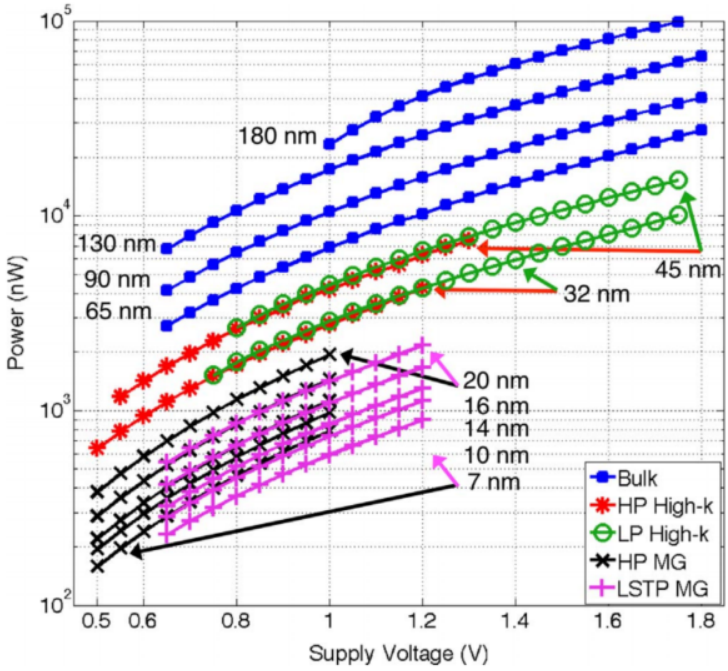


Figure 4.6: Average power for a entire clock cycle through one inverter in the middle of the FO4 inverter chain for different technologies [34].

$$PowerFactor = a_{p2}V^2 + a_{p1}V + a_{p0} \tag{4.3}$$

Type		Node	Delay Coefficients (Eq. 4.1)				Energy Coefficients (Eq. 4.2)			Power Coefficients (Eq. 4.3)		
			a_{d3}	a_{d2}	a_{d1}	a_{d0}	a_{e2}	a_{e1}	a_{e0}	a_{p2}	a_{p1}	a_{p0}
Bulk		**180 nm**	–	97.09	−356.7	406.5	–	24.64	−17.98	–	101000	−79720
		130 nm	−76.65	334.9	−493.4	275.8	7.171	−6.709	2.904	27020	−15450	5630
		90 nm	−60.34	262.5	−384.2	210.9	4.762	−4.781	2.092	17320	−11230	4328
		65 nm	−53.3	230.4	−333.9	178.6	3.755	−4.398	1.975	12890	−10510	4362
High-k	**HP**	**45 nm**	−501.6	1567	−1619	566.1	1.018	−0.3107	0.1539	5462	−1760	522.4
		32 nm	−1047	2982	−2797	873.5	0.8367	−0.4341	0.1701	4001	−1733	533.6
	LP	**45 nm**	−285.7	1239	−1795	898.8	1.103	−0.362	0.2767	6297	−3009	1124
		32 nm	−325.9	1374	−1922	913.2	0.9559	−0.7823	0.471	4557	−3037	1323
Multi-Gate	**HP**	**20 nm**	–	34.63	−66.37	41.15	0.373	−0.1582	0.04104	2922	−1286	299.9
		16 nm	–	24.8	−47.52	28.87	0.2958	−0.1241	0.03024	2133	−882.6	197.7
		14 nm	−40.66	109.2	−100.6	35.92	0.2363	−0.09675	0.02239	1675	−711	159
		10 nm	−34.95	93.65	−85.99	30.4	0.2068	−0.09311	0.02375	1456	−621.6	143.8
		7 nm	−28.58	76.6	−70.26	24.69	0.1776	−0.09097	0.02447	1179	−515.7	123.4
	LSTP	**20 nm**	−160.5	514.1	−558.6	217.5	0.2632	−0.14	0.06841	2096	−962.4	287.1
		16 nm	−114.6	366.7	−397.4	153.6	0.2139	−0.1187	0.05639	1609	−715.5	205.7
		14 nm	−85.37	271.6	−292.2	111.4	0.1556	−0.06472	0.03066	1259	−554.1	152.3
		10 nm	−71.76	228.6	−246.3	93.91	0.1261	−0.0518	0.02769	1046	−422.7	118.9
		7 nm	−61.79	196.1	−210.3	79.55	0.09365	−0.03409	0.02043	815.2	−307.3	87.54

Table 4.3: The polynomial approximation coefficient values used for the scaling factors across technology nodes and voltages [34].

After determining the scaling factors for the two scenarios under evaluation, their delay, energy, and power can be adjusted using the following equations, where x represents the post-conversion case and y refers to the available data in the current design.

$$D_x = \frac{DelayFactor_x}{DelayFactor_y} D_y \tag{4.4}$$

$$E_x = \frac{EnergyFactor_x}{EnergyFactor_y} E_y \tag{4.5}$$

$$P_x = \frac{PowerFactor_x}{PowerFactor_y} P_y \tag{4.6}$$

Since the power data represents the average power consumption of the inverter over an entire cycle, this result predominantly reflects standby power. The following equation also accounts for dynamic power, which is related to operating frequency and delay [34].

$$P_x = \frac{EnergyFactor_x \cdot DelayFactor_y}{EnergyFactor_y \cdot DelayFactor_x} P_y \tag{4.7}$$

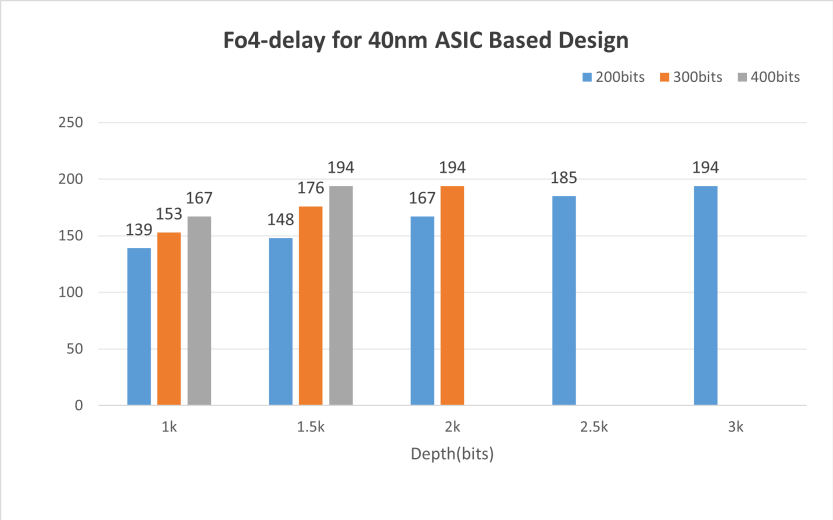


Figure 4.7: The FO4 delays for the 40 nm ASIC design varying by depth and width for fifo application.

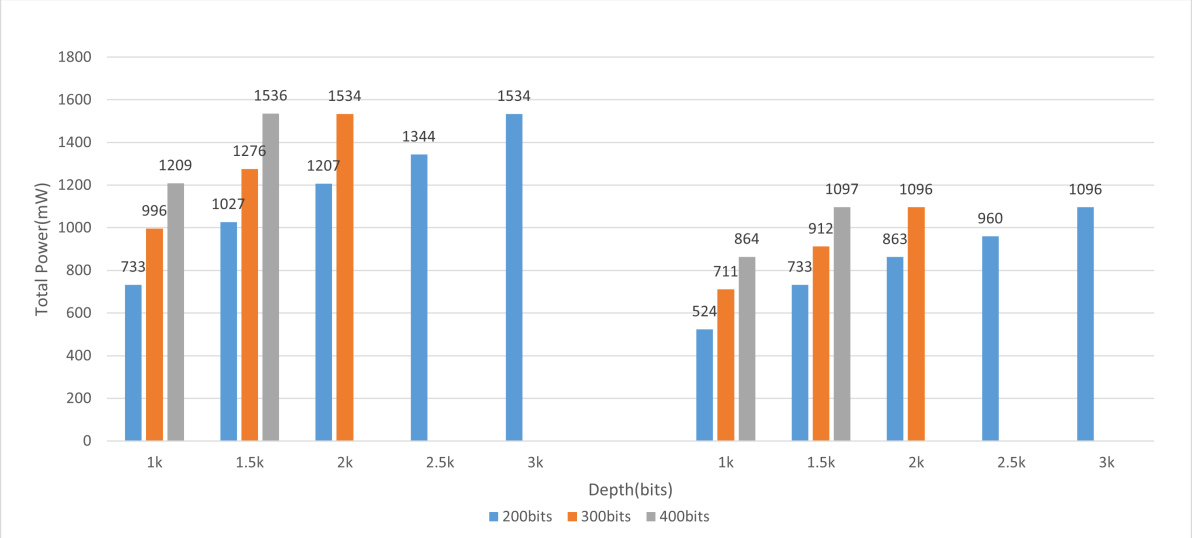


Figure 4.8: The power for the 40 nm (left) and the approximated 28nm (right) ASIC design varying by depth and width(three different colours) for fifo application.

In the context of this project, the delay and power results from the 40 nm ASIC design need to be scaled to 28 nm technology to enable a fair comparison with the 28 nm FPGA design. If the aforementioned inverter chain modeling results are applied, two estimates are required to obtain the approximate ratio of FO4 delays between the 40 nm and 28 nm nodes. However, a more accurate approach would be to directly estimate the FO4 delays for the 28 nm technology using the data from the exist technology. A 32nm process introduced in 2010 exhibited a 9.8 ps delay for a single inverter in an FO4 chain, while a 20nm process in 2012 achieved a 9.66 ps delay [21]. Extrapolating from these values, the estimated delay for a 28nm process is approximately 9.7 ps. As for 40nm, using the EDA tool Cadence and the corresponding 40 nm TSMC technology library, the FO4 delay for the technology was simulated to be

approximately 10.8 ns. The FO4 delays for the 40 nm ASIC design varying by depth and width, are shown in Figures 4.7. It can be seen that the three different colors represent three distinct bit widths. As the FIFO depth or bit width increases, the FO4 delay also increases. Observing data for different bit widths at the same depth, or for different depths at the same bit width, reveals that the FO4 delay increases linearly with both depth and width.

Regarding power, the results from Cadence include static power, which reflects leakage power, and dynamic power, which comprises switching power and internal power. Equation 4.6 is used to evaluate static power scaling from 40 nm to 28 nm, while Equation 4.7 is applied to approximate dynamic power scaling to the desired node, as it is frequency-dependent. Figure 4.8 compares the total power for the 40 nm and 28 nm ASIC designs, with the former derived from actual simulation results based on post-place-and-route implementation and the latter from approximations using the power factors method described earlier. It can be observed that the power consumption at 28 nm is approximately two-thirds that of the 40 nm process despite its depth or width.

4.2. A Comparative Analysis Approach for FPGA and ASIC Design

After addressing the comparability among different fabrication nodes, the next challenge is to establish a fair method for comparing the performance of FPGA and ASIC designs. This section discusses approaches to achieve a balanced comparison. Investigating the differences between FPGAs and ASICs is crucial yet often overlooked. A robust quantitative analysis model can assist designers in determining whether a design is better suited for FPGA implementation. Additionally, ASIC and FPGA technologies are not entirely isolated; for instance, some ASIC designs incorporate programmable elements. Furthermore, with the increasing demand for efficiency and computational power, modern FPGAs are integrating more specialized modules, such as block memory for storage and multipliers for computation [13] [15] [40]. While these additions enhance system performance, they also compromise the unique programmability of FPGAs. Therefore, a fair comparison algorithm is necessary to evaluate and optimize designs across these platforms

In a recent study, Zuchowski et al. [45] compared the delay and dynamic power consumption of a lookup table (LUT) in an FPGA to that of a gate in an ASIC standard library. They also analyzed the gate density of both in terms of area, measured in square microns. However, this approach is constrained by the limited number of gates that can be implemented in a LUT. Ian Kuon and colleagues [26] addressed this limitation by comparing FPGAs and ASICs through the performance of both implementations using the same set of benchmarks. This methodology aligns with the project's objective to compare FPGA-based and standard-cell-based designs under identical application scenarios. Consequently, this project adopts these quantitative models to evaluate and compare the area, timing, and power consumption of the two designs.

4.2.1. Area Comparison

To ensure a fair comparison between FPGA-based and ASIC-based designs, it is essential to normalize the area measurements. For designs using standard library cells, the final silicon area, post place-and-route, can be directly obtained from reports generated by the Cadence tool. In contrast, FPGA area measurements require translation, as the designer cannot alter the chip area once the device type is selected. During FPGA implementation, the Vivado tool generates an area report based on the number of various modules utilized. The final physical area is then calculated by multiplying the module's actual area by the number of instances deployed.

Notably, this estimation accounts for the entire footprint of modules, even if they are not fully utilized. For example, if a block RAM module uses only a fraction of its bits, the entire module is still included in the area calculation. This approach assumes a relatively optimistic view of FPGA heterogeneity, as designers must select devices with discontinuous areas and tolerate the area wasted by the discrete distribution of modules and the constant percentage of unused functionalities. Nevertheless, since the aim of this comparison is to explore the inherent costs of programmable manufacturing, a degree of optimism is acceptable [26].

For this thesis, the TSMC Virtex-7 family, specifically the Xc7vx1140tflg1926-2 model, was selected. The Vivado *implementation* window reveals that this device consists of four Super Logic Regions

(SLRs), which represent a single device die slice in Stacked Silicon Interconnect (SSI) Technology. Each SLR has its own dedicated logic resources, memory, and I/O resources. The connection between SLRs is achieved through specialized interconnect technology, ensuring efficient data exchange [43]. Within a SLR, it is possible to determine the percentage of the total die area occupied by distinct functional modules. By referencing the actual die area, the approximate area of each module can be estimated. For instance, the area of a slice containing a series of LUTs, registers, and multiplexers is estimated at 30 µm by 33 µm, while the area of a RAM36 block is 151 µm by 40 µm. Using this data, the physical area of register-based, LUT-based, and BRAM-based FPGA circuits for various FIFO sizes can be calculated, as shown in Table 4.4. The area of FIFOs utilizing registers is observed to fall within the range of 10^7 to 10^8 square micrometers. In contrast, designs based on LUTs exhibit an area on the order of 10^6 square micrometers. Designs employing BRAM as storage elements demonstrate the smallest footprint, with an area on the order of 10^4 square micrometers.

	Type	**BRAM-based**			**LUT-based**			**Register-based**		
	Width(bit)	**200bits**	**300bits**	**400bits**	**200bits**	**300bits**	**400bits**	**200bits**	**300bits**	**400bits**
Depth(bits)	**1k**	6.28×10^4	9.20×10^4	1.16×10^5	1.45×10^6	2.59×10^6	2.79×10^6	5.86×10^7	1.03×10^8	1.51×10^8
	1.5k	9.63×10^4	1.39×10^5	1.81×10^5	2.34×10^6	3.39×10^6	4.41×10^6	9.16×10^7	1.38×10^8	1.92×10^8
	2k	9.63×10^4	1.40×10^5		2.88×10^6	5.22×10^6		1.15×10^8	1.90×10^8	
	2.5k	1.72×10^5			3.49×10^6					
	3k	1.70×10^5			4.06×10^6					

Table 4.4: Approximated silicon area in μm^2 unit for various type of FPGA designs, with different depth and width for the fifo implementaions.

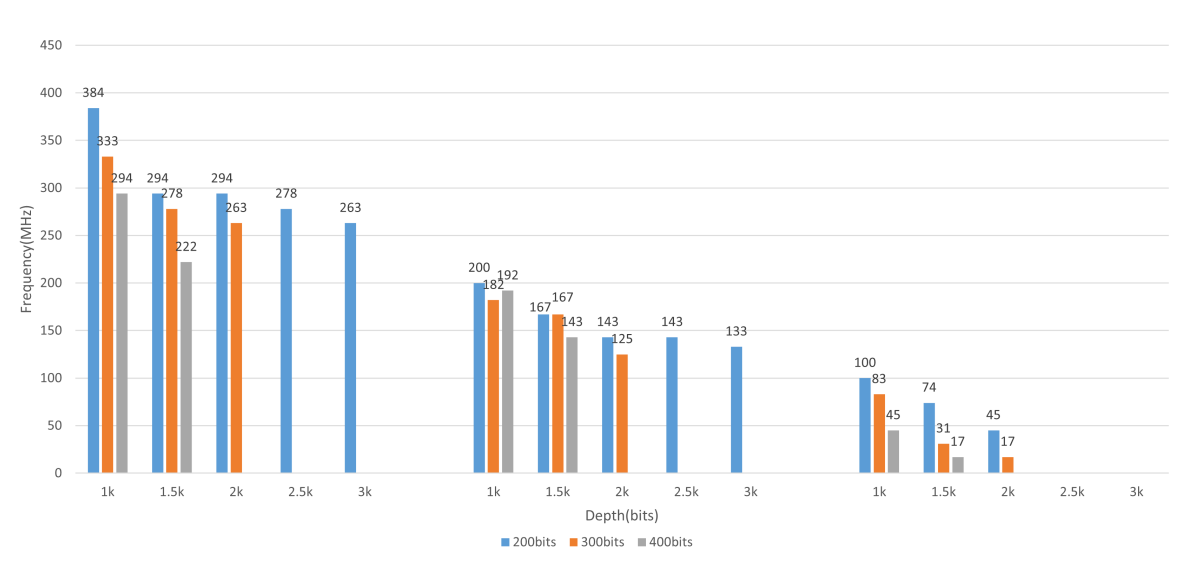


Figure 4.9: The frequency for the 28 nm FPGA design with BRAM-based (left), LUT-based (middle) and Register-based (left) with various depth and width(three different colours) for fifo application.

4.2.2. Speed Comparison

The next comparison is based on timing outputs. Static timing analysis is conducted to determine the maximum operating frequency on the critical path. ASIC and FPGA design can directly compare the maximum operational frequency [26]. The frequency for the 28 nm BRAM-based, LUT-based and Register-based FPGA fifo designs with different depth and width, are shown in Figures 4.9. It can be observed that designs based on BRAM achieve the highest performance, with operating frequencies ranging from 200 to 400 MHz. Next in performance are designs that utilize LUTs as storage elements, which can reach frequencies of over 100 MHz. The slowest performance is observed in designs that use registers for data storage, with maximum operating frequencies limited to the tens of MHz range. Additionally, in the sections with depths of 2.5Kbits and 3Kbits, it can be observed that the register-based designs are missing this data. This is because the FPGA on-chip register resources are insufficient to support such large FIFO sizes.

4.2.3. Power Comparison

Finally, power consumption is compared. Generally, energy consumption is divided into static and dynamic power consumption. Static power, or standby power, includes power consumption due to leakage currents in transistors. Dynamic power includes switching power (related to charging and discharging the load capacitance during switching) and short-circuit power (due to non-ideal voltage transitions). Switching power consumption is determined by the circuit's activity during charge and discharge cycles. In both ASIC and FPGA designs, hardware description languages are typically used, but they may lack accurate descriptions of the toggle rates and usage frequencies of various parts in real-world applications. This can lead to inaccuracies in switching power estimates. To obtain more accurate results, a testbench is designed to simulate the circuit and capture dynamic power consumption. For FPGA designs, a SAIF (Switching Activity Interchange Format) file is generated during testbench simulation to capture signal switching activity, which is then used to estimate power consumption. For ASICs, similar data is obtained by generating a VCD (Value Change Dump) file with switch annotations during synthesis. However, due to the difficulty in obtaining appropriate testbenches for most designs, this thesis assumes that all signals toggle at the same frequency and that all nets have the same static probability. Short-circuit power consumption arises because the input voltage waveform is not an ideal step signal and has a finite rise and fall time. During the rise and fall of the input waveform, both NMOS and PMOS transistors may be simultaneously on, resulting in a direct current path from the power supply to ground, contributing to short-circuit power consumption.

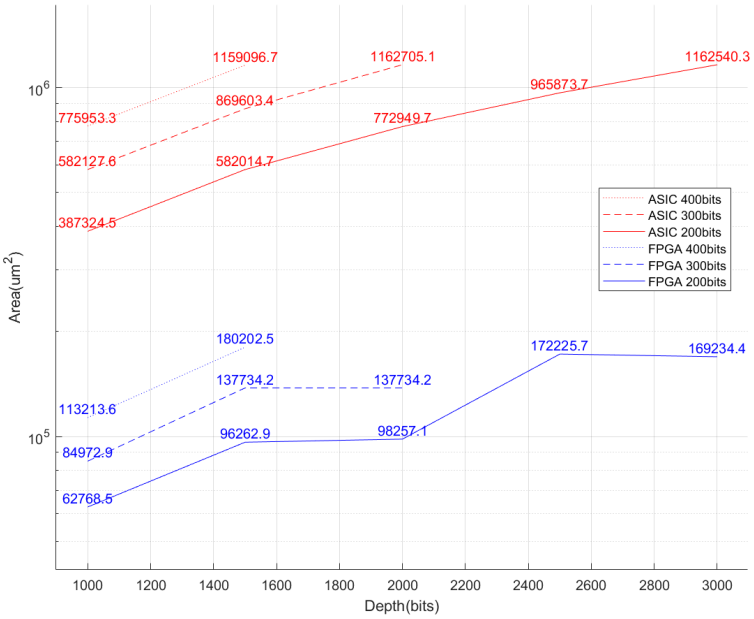


Figure 4.10: Physical Area for 28nm ASIC Design (Top) and 28nm FPGA-BRAM Design (Bottom) with 200MHz.

Power consumption measurements are conducted under the same temperature and frequency conditions. Dynamic power consumption can be directly compared between FPGA and ASIC designs. Currently, FPGA devices must enter standby mode before operation, although future high-end, low-power devices are expected to support partial power-down functionality. Therefore, FPGA static power may need to be adjusted, as designs generally do not utilize all on-chip resources. Static power is roughly proportional to transistor width and, by extension, on-chip are [22]. Thus, FPGA static power should be multiplied by the ratio of the silicon area used by the design to the total FPGA on-chip area. This ratio can be estimated using the aforementioned area estimation method. While this method may overlook some of the excess power consumption due to FPGA heterogeneity and device selection limitations, it is acceptable for exploring FPGA manufacturing characteristics. Since power and energy reflect design characteristic under different conditions, power is generally measured at high frequencies to depict the power consumption required for optimal performance. In contrast, energy represents the total energy

needed to accomplish the same task at lower frequencies. Therefore, this thesis primarily concentrates on comparing power consumption results [26].

4.3. ASIC Shift-register FIFO vs circular FPGA FIFO results

After systematically presenting the data comparison models for designs based on different processes and device types, the following section provides a comparative analysis between a 40 nm ASIC design and a 28 nm FPGA design across various aspects. The analysis focuses on BRAM-based designs, as they demonstrate superior performance, power efficiency, and area utilization compared to LUT-based and register-based alternatives. Therefore, the subsequent comparisons are centered on BRAM-based FPGA designs and ASIC designs.

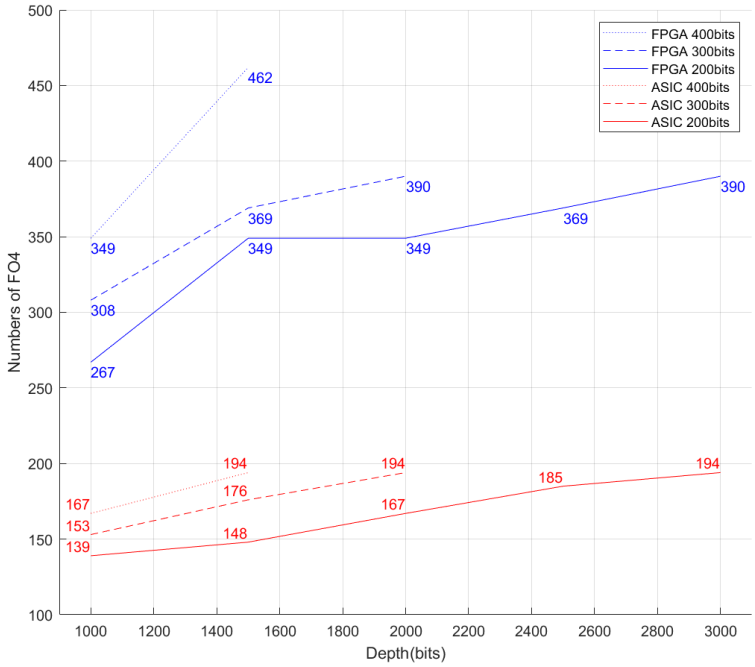


Figure 4.11: Number of the FO4-Delay for ASIC Design and FPGA BRAM-Based Design.

First, after scaling the area of the 40 nm ASIC design to account for process technology and estimating the actual area usage of the 28 nm FPGA design, Figure 4.10 illustrates the overhead required to stabilize FIFOs of different widths and depths in both ASIC and BRAM-based FPGA designs at 200 MHz in a 28 nm process. As depicted, the area of the ASIC design is 6 to 7 times larger than that of the BRAM-based FPGA. Furthermore, the area of ASIC designs increases linearly with depth, whereas the area of FPGA designs shows discrete jumps. This discrepancy arises because FPGA designs can only use integer numbers of BRAMs to implement FIFO buffers. Consequently, when the FIFO size increases to a point where it requires the use of the N-th BRAM, and as the FIFO size continues to grow until the N-th BRAM is nearly fully utilized, the area remains relatively constant during this range.

Analysis of Cadence reports reveals that the ASIC design employs the smallest D flip-flop buffer from the standard library without reset functionality: the DFQD1BWP, with a macro area of 3.528 μm^2. In contrast, a 40 nm TSMC 6T SRAM cell occupies between 0.376 and 0.242 μm^2, suggesting that each transistor occupies approximately 0.063 μm^2. Consequently, a storage register roughly requires 56 transistors. The general schematic of a D flip-flop shows that it requires 36 to 40 transistors. Since SRAM memory cells are optimized for area and each transistor occupies less space in these cells compared to standard library cells, the estimate of 56 transistors per register is considered reasonable.

Therefore, in a shift-register design, the area of a memory cell (56 transistors) is approximately ten times that of an SRAM cell (6 transistors), leading to an overall area that is 6 to 7 times larger. This

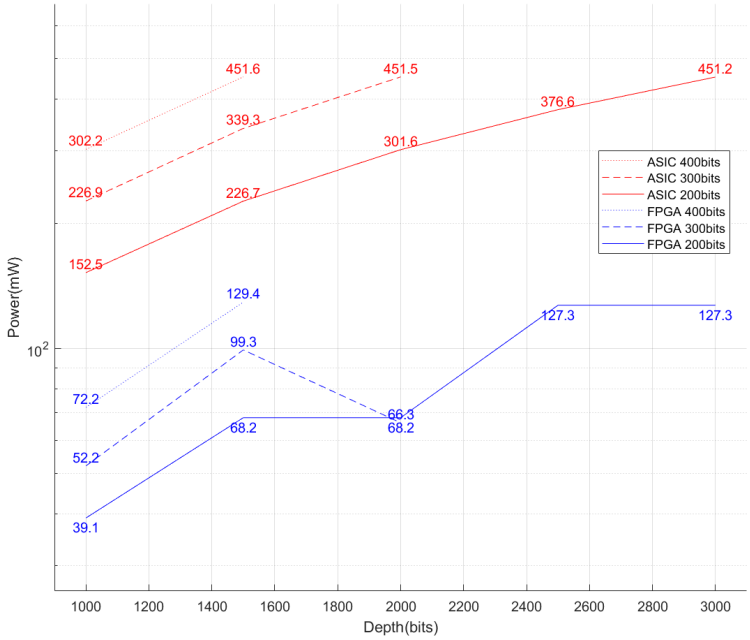


Figure 4.12: Total Power for 28nm ASIC Design (Top) and 28nm FPGA-BRAM Design (Bottom) with 200MHz.

finding highlights that, even though a linear FIFO can be constructed using pure memory cells—thus eliminating the need for address pointers, address decoders, and column multiplexers required in a circular FIFO—registers still impose a significant area overhead compared to SRAM cells. Consequently, the area cost remains considerably higher in larger FIFO designs.

The following presents a comparison of operating frequencies. Due to process differences between the two platforms, the FPGA system's operating frequency must be normalized using the FO4 (Fan-Out of 4) model to yield comparable results. Figure 4.11 displays the FO4 delay values for the 40 nm ASIC design and the 28 nm BRAM-based FPGA design, across varying FIFO sizes. The ASIC results show a gradual increase from 100 to 200, while the FPGA results range from a minimum of 267 to a maximum of 462, with the FPGA exhibiting a steeper rate of increase than the ASIC. This indicates that the ASIC system outperforms the FPGA by a factor of two. It can be concluded that, since control signals such as read and write addresses do not need to be processed, shift-register FIFOs are significantly faster than circular FIFOs.

The final comparison focuses on power consumption. Figure 4.12 presents the normalized power data at the 28 nm standard, obtained at a frequency of 200 MHz. It is evident that the power consumption of the ASIC design ranges from 152.5 mW to 451.5 mW, which is 3 to 4 times higher than that of the FPGA design. This finding is consistent with the results shown in Figure 4.10, where the ASIC design occupies a larger area compared to the FPGA.

In summary, the ASIC FIFO based on shift registers operates 2 to 3 times faster than the SRAM-based FPGA circular FIFO but incurs a 6 to 7 times larger area penalty, making it a suboptimal solution. In the next chapter, an alternative approach employing ring counters to replace the address control logic in SRAM will be discussed.

5

BRAM Address Decoder Optimization

The Chapter focuses on further optimization of BRAM address decoders. Section 5.1 first discuss the exsiting address decoders from single block type to predecoder type. Section 5.2 and Section 5.3 then show the structure and the implementation of the proposed ASIC ring-counter design with its final result analysis compared with the conventional address decoder in BRAM.

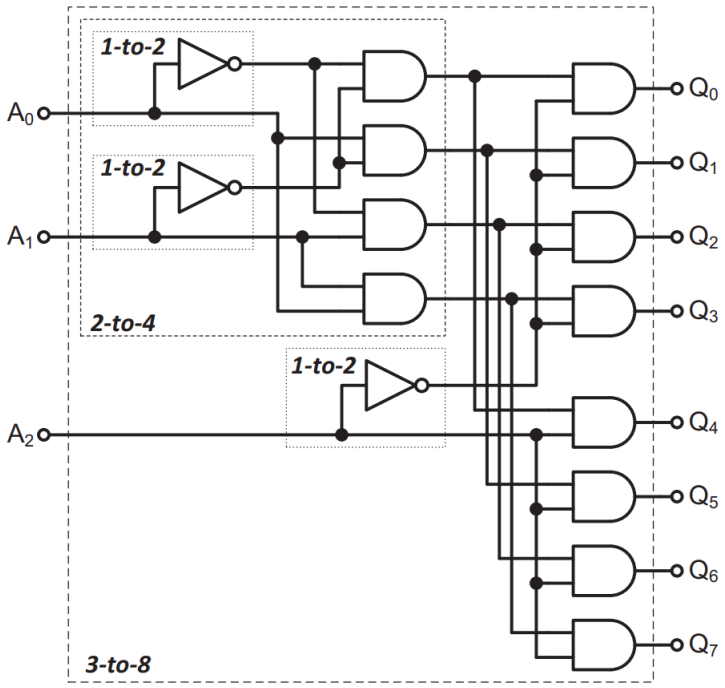


Figure 5.1: Schematic view of 3-8 conventional decoder [8].

Given the considerable drawbacks of the linear FIFO, particularly in terms of area cost, it becomes clear that further optimizations are necessary. The direction of this project will thus shift towards enhancing the area efficiency of decoders within BRAM. By focusing on reducing the area footprint the BRAM address decoders, the aim is to achieve a more optimal balance between resource utilization and performance in future designs. This shift in focus is essential for developing memory architectures that meet the stringent demands of modern high-performance computing systems, where both space and speed are at a premium.

5.1. Overview of Exsiting Address Decoder

In SRAM design, the efficiency and performance of memory operations are heavily dependent on the row and column decoders. These decoders are crucial components that enable access to specific memory cells by interpreting the input address signals. In a typical SRAM architecture, the row and column decoders process an n-bit address to select one of the 2^n possible outputs, thereby determining the specific row or column to be accessed.

Figure 5.1 illustrates a traditional 3-to-8 decoder, which is constructed using AND gates. This decoder operates by selecting three input signals and their complements, effectively covering all possible input combinations. Each unique combination corresponds to a specific output being selected. However, as the number of input signals increases, the complexity and latency of the decoder also increase, primarily due to the cascading nature of the AND gates used. It is also important to note that while the conventional design utilizes AND gates, these are often implemented as a combination of NAND gates followed by NOT gates. This series configuration results in a larger circuit footprint and increased delay, especially when additional inputs are required [8] [30].

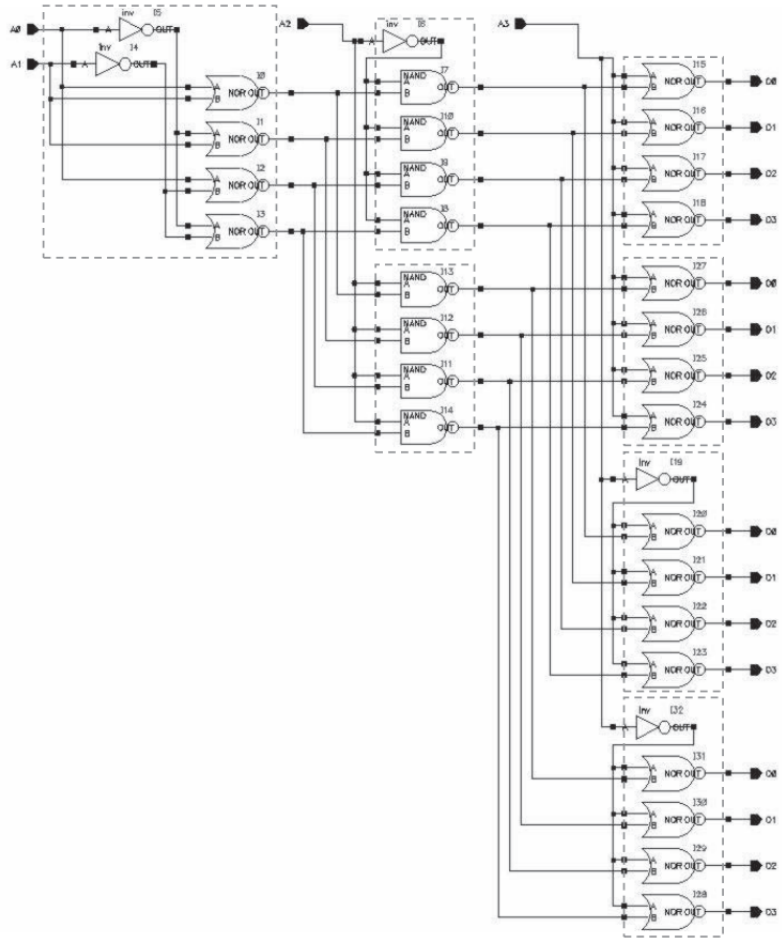


Figure 5.2: Schematic view of 4-16 based on NAND gate and NOR gate. [8]

To address the inefficiencies of AND gate-based decoders, designers have explored alternative designs using simpler gates such as NOT, NAND, and NOR gates. A notable approach is to directly employ NAND and NOR gates, which offer more compact and efficient circuit designs. For example, a NAND gate outputs a low signal only when both inputs are high, while a NOR gate outputs a high signal only when both inputs are low. By strategically combining these gates, it is possible to construct more efficient decoders.

As shown in Figure 5.2, a three-stage 4-to-16 decoder can be built using alternating blocks of NOR

and NAND gates. The first stage, composed of NOR gates, implements a 2-to-4 decoding function. The second stage adds an additional input signal, A_2, and its complement to expand the decoding to a 3-to-8 configuration. The final stage completes the 4-to-16 decoding. While this design is more efficient in terms of area, it introduces a new challenge: unbalanced input signal paths. The least significant bit (LSB) must traverse all stages, whereas the most significant bit (MSB) only needs to pass through the final stage. This discrepancy in signal paths can lead to varying critical path delays, potentially causing errors where multiple outputs are erroneously activated simultaneously [8] [30].

To mitigate the issues associated with traditional decoding methods, modern SRAM architectures often employ a predecoder-based design, which is widely adopted in commercial FPGA devices. The predecoder approach addresses the problem of increasing delay and area associated with large fan-outs in conventional decoders.

The predecoder-based architecture divides the decoding process into two stages: the predecoder and the postdecoder. This division reduces the fan-out in each stage, allowing for the use of smaller gates and, consequently, reducing overall area and delay.

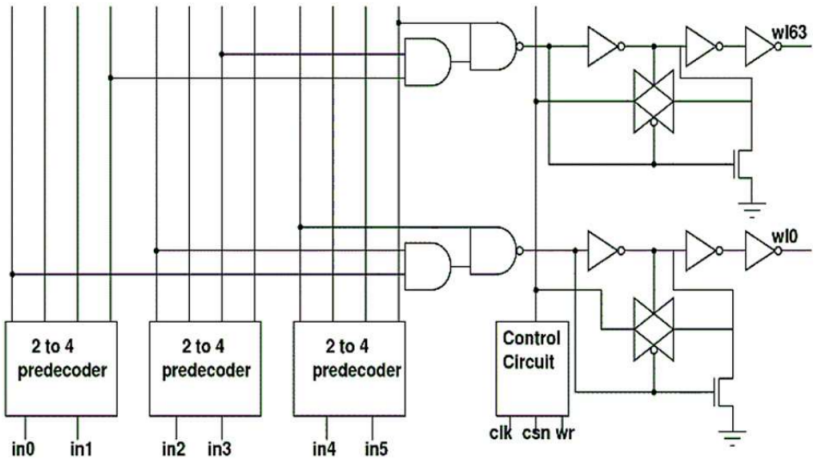


Figure 5.3: Overview of 6-64 decoder with three 2-4 predecoders [23].

As illustrated in Figure 5.3, a 6-to-64 decoder is implemented in two stages. The first stage consists of three 2-to-4 predecoders. Each predecoder handles a portion of the input signals, effectively reducing the complexity of the subsequent decoding stage. The outputs of these predecoders serve as inputs to the postdecoder, which then generates the final 64-wordline outputs. This hierarchical structure not only balances the critical paths but also enhances the overall speed and efficiency of the decoder [23] [5].

5.2. Principle of the Proposed Ring-Counter Structure

The progression from traditional AND gate-based designs to more advanced predecoder-based approaches in SRAM decoding architectures highlights the continuous pursuit of higher efficiency and reduced latency in memory systems. However, one persistent challenge remains: the need to decode read and write addresses efficiently. As the data capacity increases, the number of address bits also rises, leading to a significant increase in both area and delay budgets required for address decoding.

This issue is particularly relevant in the context of this project, where the target application is a FIFO memory system. Unlike general-purpose memory, a FIFO does not require access to specific storage locations; instead, data is written sequentially into the memory and read out once the defined depth is reached. Therefore, instead of relying solely on address decoding to control and assert the corresponding wordlines, an alternative approach can be employed.

One such approach involves using two ring counters to directly manage the read and write positions within the two-dimensional memory array, effectively shifting the data through the array. This method leverages the sequential nature of FIFO operations, simplifying the control logic and potentially reducing both the area and delay associated with traditional address decoding.

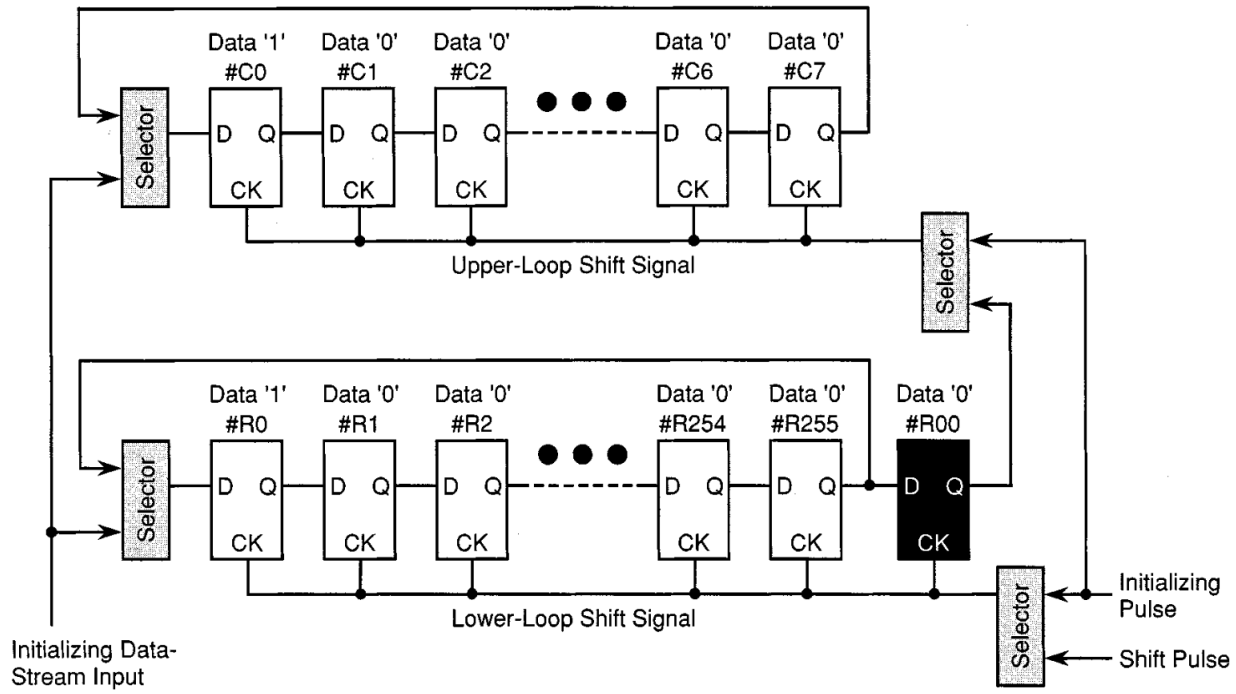


Figure 5.4: Architecture of the proposed ring-counters address pointer [33].

Figure 5.4 illustrates the architecture of an address pointer for an SRAM memory array with dimensions of 256 rows by 8 columns. The design includes a row shift register at the bottom that spans 255 rows and a column shift register at the top for 8 columns. Each row and column in this configuration requires a dedicated register for control, leading to a substantial number of registers overall. According to data from the TSMC 40nm LEF library, the area occupied by a register with a reset function is 4.057 m^2, while a simpler register without the reset function occupies only 3.528 µm². By opting for these simpler registers without the clear function, the area budget can be reduced by approximately 13%.

However, without a reset function, the initialization process for the ring counters becomes necessary. For instance, if the starting point for writing data is the first word of the first row and column, the C0 and R0 registers must be set to ”1,” with the remaining registers must set to ”0.” If the output of R00 were to directly control the upper-loop shift signal, the signal need to be propagated through the entire row shift register while the column shift register only shift one bit to the right. This would result in an initial setup time of 256 x 8 = 2048 cycles.

To reduce this initial delay, a set of initialization data can be directly loaded into both the upper and lower shift registers through the leftmost two multiplexers. The clock signal for initialization is then managed via the rightmost two multiplexers, controlling the two shift registers such that the initialization process completes in only 257 cycles.

It is important to note the presence of an additional R00 register in the row shift register. This extra register is necessary because if the signal were to connect directly from the output of R0, the signal would cause an unintended pulse in the upper-loop shift signal due to R0 being set to ”1” at the end of the initialization. This unwanted pulse, as depicted in Figure 5.5, would interfere with the proper operation of the FIFO.

In the subsequent FIFO operation phase, the left two multiplexers select the corresponding feedback signal, and the right multiplexer inputs the shift pulse to the lower-loop shift signal, while connect the output of R00 to the upper-loop shift signal. The ”1” signal propagates from R0 through to R255, indicating the movement of the pointer from the first column in the first row to the last row, and then loops back to R0 as C0’s ”1” transitions to C1. This transition indicates the pointer’s shift to the second column of the first word, continuing in this manner to form a closed loop [33].

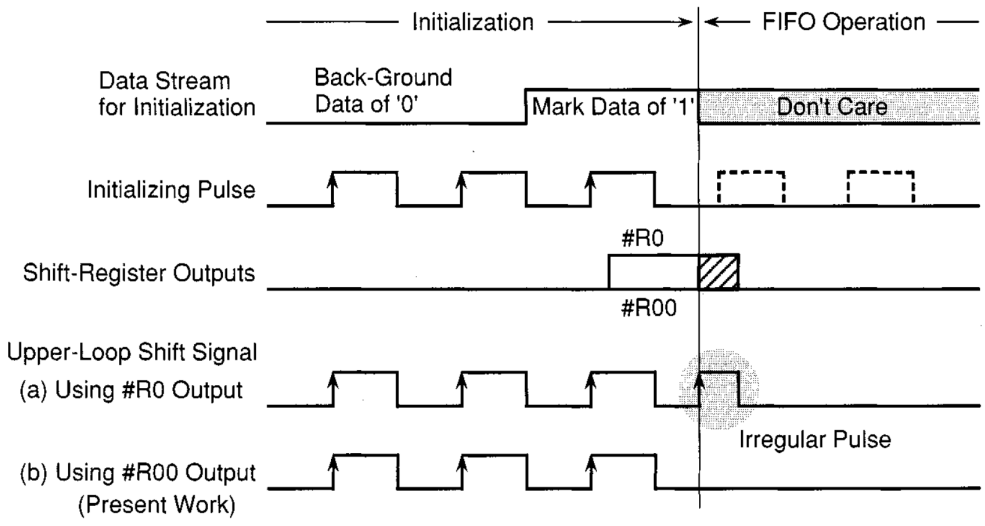


Figure 5.5: Waveform for the initialization of the ring-counters [33].

This dual ring counter design for the address pointer allows for efficient management of both read and write addresses. By directly controlling and selecting the wordline and bitline through physical means, this method eliminates the need for traditional address decoding.

5.3. Implementation and Validation of the Ring-Counter Design

Figures 5.6 and 5.7 provide a detailed illustration and simulation of an 8-row by 4-column ring counter architecture. In the simulation waveform, the row pointer is observed to sequentially move from 0 to 7, while the column pointer advances every 8 cycles. This pattern demonstrates the orderly progression of the pointers within the ring counter system.

Given the disparity in the number of rows and columns, it is necessary to employ a counter to oversee the initialization of both the row and column shift registers. For example, when the number of columns is smaller than the number of rows, the initialization process for the column shift register will finish before that of the row shift register. Upon completion of the column initialization, the counter exerts control over the column shift register, halting its shifting process. Meanwhile, the row shift register continues its shifting until it is fully initialized. Once both the row and column shift registers have been properly initialized, if the FIFO operation has not yet commenced, both shift registers will maintain their initialized state, ready to begin the data processing tasks as soon as required.

This implementation underscores the importance of managing the timing and control within a ring counter-based memory array. By monitoring and regulating the initialization process through the use of counters, the system ensures that both the row and column shift registers are synchronized and correctly prepared for subsequent operations.

Depth	Width	200	300	400	Depth	Width	200	300
1K	**Total**	36539.4	51764.15	70033.85	2K	**Total**	70033.85	103528.3
	Memory Array	22740	34110	45480		**Memory Array**	45480	68220
	Remain	13799.4	17654.15	24553.85		**Remain**	24553.85	35308.3
	SA/WR per 1Kbit	5377.225				**SA/WR per 1Kbit**	5377.225	
	Decoder	3044	1527.465	3044.95		**Decoder**	3044.95	3044.95
	Decoder Average	2537.13				**Decoder Average**	3044.95	

Table 5.1: The area composition results of 28 nm BRAM in various depths and widths. The unit for depth and width is bits, for area is μm^2.

Table 5.1 presents a detailed area composition analysis of BRAM in a 28nm FPGA design, focusing on BRAM-based FIFOs with various depths and word sizes. The analysis specifically examines FIFO configurations with depths of 1 Kbit and 2 Kbits and word sizes of 200 bits, 300 bits, and 400 bits. The

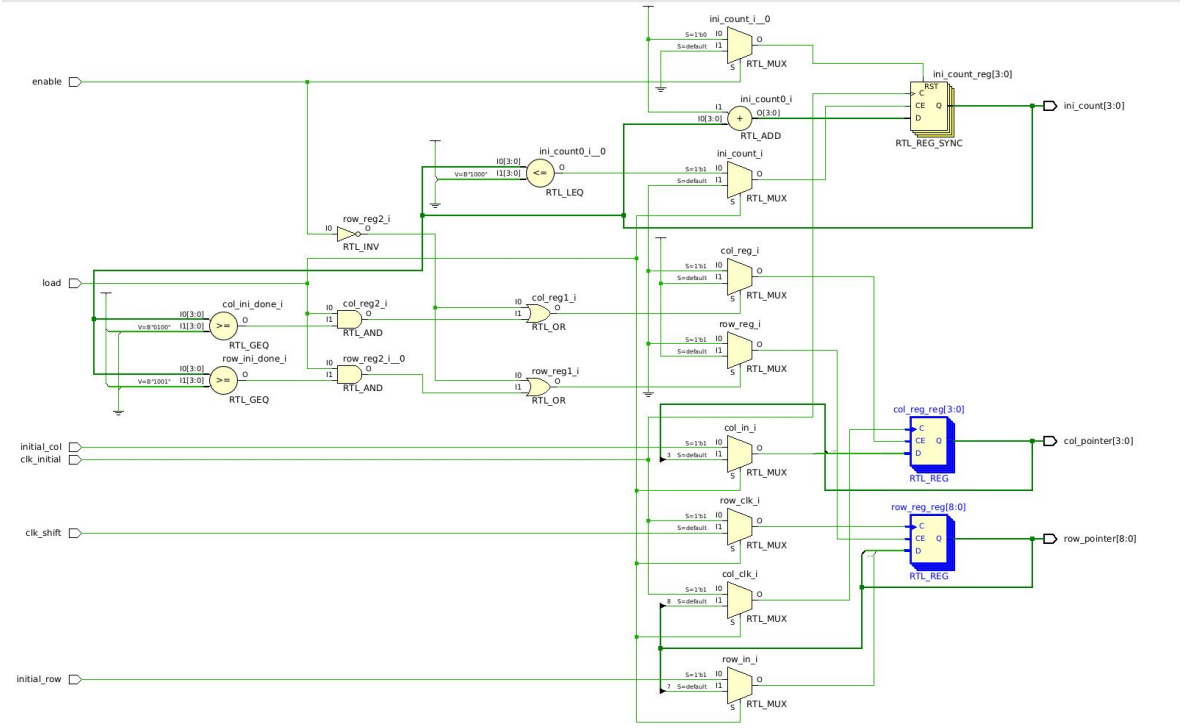


Figure 5.6: Schematic view of 8 row-4 column ring-counter address pointer.

total BRAM area is first determined by multiplying the number of BRAMs reported in Vivado by the estimated size of each BRAM. According to the datasheet, the memory cell area in a 40nm SRAM IP ranges from 0.242 μm^2 to 0.375 μm^2. Given that FPGA BRAMs are typically optimized for area efficiency, the memory bit cell area for the 28nm BRAM is calculated as 0.242 μm^2, adjusted by a process area factor of 0.47, which shown in section 4, resulting in a memory cell area of 0.1137 μm^2. This area is then multiplied by the total number of memory bits in the design to obtain the total memory array area in every BRAM.

First, analyze the data with a depth of 1 Kbits. Upon subtracting the memory array area from the total BRAM area, the remaining area—attributed to components such as the address decoder, precharger, sense amplifier (SA), and write driver (WR)—amounts to 13,799.4 μm^2, 17,654.15 μm^2, and 24,553.85 μm^2 for the different word sizes. Notably, the area dedicated to the address decoder remains roughly proportional to the depth, as it is designed to manage 1,000 read and write locations consistently across different word sizes. However, the areas associated with the precharger, SA, and WR components vary directly with the number of bit cells in the memory array, reflecting their dependence on the word size.

The incremental area for the remain part between 300-bit and 200-bit configurations, as well as between 300-bit and 400-bit configurations, is predominantly due to the increased number of these word-size related components. Subtracting the remaining area associated with the 200-bit configuration from the 300-bit configuration provides an estimate of the area required for the precharger, SA, and WR components that are necessary to support an additional 1 Kbits of memory cells, from 200Kbits to 300Kbits in this case. The same method can be applied to determine the difference between the remaining areas of the 400-bit and 300-bit configurations.

By averaging the differences, the area of 1 Kbit worth of word-size related components is estimated to be 5,377.225 μm^2. Subsequently, the area of the address decoder is approximated by subtracting this estimated area from the total remaining area for each word size configuration shown in the fourth row in Table 5.1, yielding an average address decoder area of 2,537.13 μm^2 for the 1 Kbits depth BRAM. Applying the same analytical method to a BRAM with a 2 Kbit depth, the average area of the address decoder is found to be 3,044.95 μm^2.

Table 5.2 provides a detailed area analysis of ASIC ring-counter address pointers designed to control

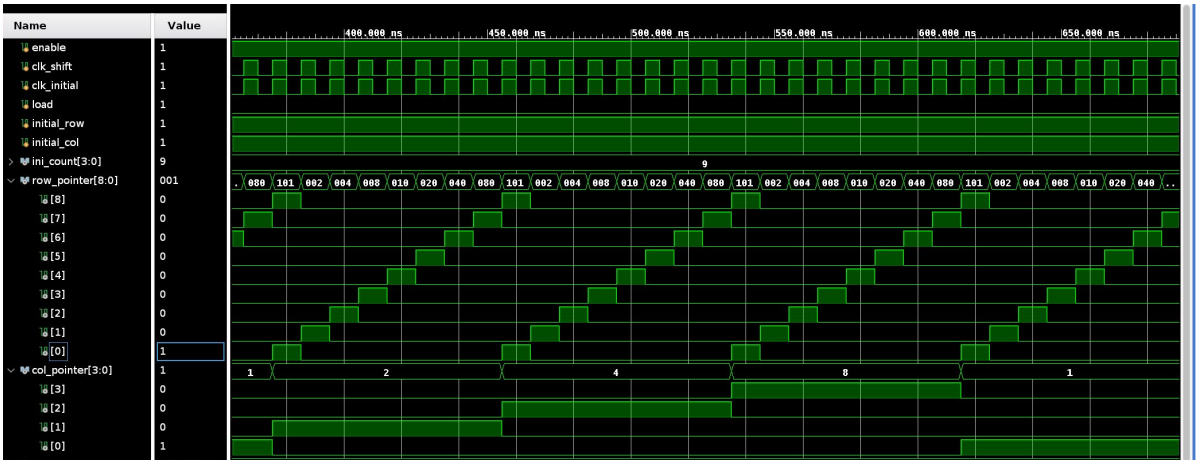


Figure 5.7: Waveform for the 8row-4column ring-counter address pointer.

	Ring-counter Combination		Total Address	Operational Frequency(MHz)
	128 Row - 8 Column	**256 Row - 4 Column**		
Area of one ring-counter in 40 nm	740.880	1353.341	1024	385
Area of two ring-counters in 28 nm	666.792	1272.14		
	128 Row - 16 Column	**256 Row - 8 Column**		
Area of one ring-counter in 40 nm	780.394	1376.449	2048	295
Area of two ring-counters in 28 nm	733.57	1293.86		

Table 5.2: The area of ring-counter with different row/column combinations. Area unit in μm^2.

1024 and 2048 positions, corresponding to FIFO depths of 1 Kbit and 2 Kbits, respectively. The analysis examines different combinations of rows and columns based on common setting of SRAM memory array dimension. For a 1024-position address pointer, two configurations are implemented: 128 rows by 8 columns and 256 rows by 4 columns.

The first configuration, 128 rows by 8 columns, occupies an area of 740.880 μm^2. Since read and write ports are controlled by two separate ring counters, area of the final proposed design need to composed with two ring counters. And the area results for BRAM address decoders are provided for a 28nm process, therefore the area of proposed design need to scale to the same process. Using the process conversion factor of 0.47, derived in Section 4, the scaled area for the two ring counters in a 28nm process is 666.792 μm^2. The second configuration of 256 rows by 4 columns, though designed for the same 1024 positions, requires more registers, thus has approximately 1.8 times the area of the 128 rows by 8 columns configuration.

For the 2048-position address pointer, which corresponds to a FIFO depth of 2 Kbits, two configurations are considered: 128 rows by 16 columns and 256 rows by 8 columns. The area for the 256 rows by 8 columns configuration is similarly about 1.8 times larger than that of the 128 rows by 16 columns configuration. This consistent area ratio across different configurations suggests that ratio of number of row and column plays a critical role in determining the overall area requirement for the ring-counter address pointers.

This analysis highlights the trade-offs among different SRAM memory array configurations, particularly emphasizing how the ratio of numbers of rows to columns significantly impacts the area budget. The findings indicate that as the memory array becomes more elongated—characterized by a higher ratio of rows to columns—the design requires a greater number of registers, leading to a longer critical path. Consequently, both the area efficiency and timing performance deteriorate. These insights underscore that the proposed ring-counter address pointer is more suitable for a square SRAM memory array configuration, where the balance between rows and columns minimizes the area footprint and optimizes timing performance.

Table 5.3 presents a comparative analysis of the area budgets for address pointers in FIFOs with depths of 1Kbits and 2Kbits, implemented both as traditional address decoders in BRAM and using the proposed ring-counter approach. The results show two different row and column configurations

Depth	Area for address decoder in BRAM	Combination of ring-counter	Area for ring-counter design	Percentage Reduction
1Kbits	2,537.13	128 Row - 8 Column	666.792	74%
		256 Row - 4 Column	1,272.14	50%
2Kbits	3,044.95	128 Row - 16 Column	733.57	76%
		256 Row - 8 Column	1,293.86	57.5%

Table 5.3: The area of the address pointer for BRAM and the proposed design, with the reduction in percentage.

for the ring-counter, based on the data from Table 5.2. Remarkably, the proposed ring-counter design demonstrates a significant reduction in the area required for the address decoder. For a 1Kbits FIFO depth, the ring-counter reduces the area by 50% to 74% compared to the conventional BRAM address decoder. Similarly, for a 2Kbits FIFO depth, the area reduction ranges from 57.5% to 76%.

These findings suggest that the proposed ring-counter design offers substantial potential for optimizing the area of address decoders in BRAM-based systems. Although some of the experimental data is based on estimations, the overall trend indicates that the ring-counter approach could lead to more efficient and compact memory designs, making it a promising alternative to traditional address decoders, especially in scenarios where area optimization is critical. This area efficiency, coupled with the potential for reduced timing delays in square or near-square memory array configurations, underscores the value of the proposed method in advanced SRAM-based systems.

6

Conclusions

This thesis explored various strategies for optimizing FPGA on-chip memory in the context of large FIFO-style memory buffers. Given the widespread use and vast potential of FPGAs in data processing, where FIFO buffers were particularly vital for tasks like data transmission, caching, and connecting different clock domains, this project focused on enhancing the efficiency of FPGA on-chip memory resources for FIFO implementations.

First, a conventional circular FIFO was implemented on widely used commercial FPGAs using different types of on-chip memory resources, including registers, LUTs, and BRAM. Among these, the BRAM-based FIFO. This is the common practise for implementing FIFOs on FPGA devices. BRAM on-chip memory blocks demonstrated the best performance. Next, an ASIC linear FIFO was developed and implemented as a potential replacement for the conventional BRAM-based circular FIFO. This design eliminated control components such as the address pointer and address decoder, which usually require additional logic resources. However, compared to the BRAM-based FPGA circular FIFO, the proposed ASIC linear FIFO achieved a 2x improvement in speed but occupied a significantly larger area, approximately 7 to 8 times more. The third phase of the project focused on optimizing the BRAM address decoder, which consumes substantial area resources and increases considerably with BRAM capacity. An ASIC ring-counter structure was designed and implemented to replace the conventional BRAM address decoder, showing significant potential for area reduction, with more than a 50% decrease in area for FIFO depths of 1Kbits and 2Kbits.

Chapter 1 introduced the motivation for optimizing FIFO on FPGA by emphasizing the significant potential and market growth of FPGAs, as well as the widespread use of FIFOs in various FPGA applications. This chapter also outlined the thesis contributions and provided an overview of the main content covered in each subsequent chapter;

Chapter 2 discussed the background knowledge for the thesis project. It first provided a comprehensive overview of FPGA on-chip memory resources, include LUT, register and BRAM. It discussed the evolution and structure of these memory resources. It further offered insights into their impact on FPGA system by discussing the evolution of FPGA on-chip memory architecture. It continued by introducing the fundamental principles and application scenarios of FIFO buffers. The two important structure: linear and circular FIFO were explained;

Chapter 3 shifted the focus to the design and implementation of two FIFO structures: the proposed ASIC-based linear FIFO and the FPGA-based circular FIFO, which required optimization. It highlighted the comparative metrics and design flows for both approaches. The chapter also addressed hold time violations in the ASIC designs by utilizing multi-tap H-tree clock tree synthesis instead of standard clock tree synthesis. This method resolved the violations by ensuring a more even clock distribution;

Chapter 4 provided a comparative analysis of performance, power consumption, and area for the two FIFO designs. It began by introducing models to compare the designs across various technology nodes and device types. The analysis first compared FPGA-based FIFO implementations using different on-

chip memory resources, with the BRAM-based FIFO demonstrating the highest performance. The comparison then shifted to BRAM-based FIFO on FPGA versus ASIC-based FIFO. The results revealed that, while the linear ASIC FIFO simplified the design by eliminating address control components and achieved over twice the performance of the circular FPGA FIFO, it incurred a significantly larger area footprint—approximately 7 to 8 times that of the circular BRAM-based FPGA FIFO. The underlying cause of this discrepancy was also discussed; it was found that ASIC FIFOs use registers for storage, whereas BRAM-based FPGA FIFOs use SRAM, with the area cost per memory bit in registers being roughly ten times larger than that of SRAM;

Chapter 5 moved the focus to optimizing the address decoder in BRAM, which consumes a significant area budget and scales dramatically as the input size increases. The chapter proposed an ASIC ring-counter design to replace the conventional address decoder, with the goal of improving area efficiency. The ring-counter design was implemented for FIFO depths of 1Kbits and 2Kbits, demonstrating significant improvements in resource utilization, achieving over 50% area reduction after thorough analysis and careful estimation.

In summary, this research highlighted the importance of optimizing on-chip memory in FPGAs by integrating ASIC design techniques. The project focused on FIFO implementation due to its critical role in a wide range of FPGA memory-related applications. More specifically, the optimization and comparison of BRAM-based circular FIFOs on FPGAs were prioritized since this architecture is the most common and effective implementation of FIFO on FPGAs. The first approach involved replacing the conventional BRAM-based circular FIFO with a linear ASIC FIFO. This replacement demonstrated that linear ASIC FIFO structures could achieve performance gains of 2 to 3 times compared to conventional BRAM-based circular FIFOs in terms of maximum frequency. However, these performance benefits came at a considerable area cost, roughly 7 to 8 times greater than that of BRAM-based FIFOs. This increased area cost was mainly due to the fact that BRAM uses SRAM as its storage unit, with each bit requiring about 6 transistors, whereas linear ASIC FIFOs use registers, requiring over 56 transistors per bit. Despite the simplified structure of the linear ASIC FIFO, the discrepancy in memory bit structure led to a tenfold increase in area for each memory bit. This resulted in an overall design area that was 7 to 8 times larger than that of the BRAM-based circular FPGA FIFO. In the second optimization experiment focused on the conventional BRAM address decoder. The findings showed that replacing the BRAM address decoder with an ASIC ring-counter design could reduce area requirements by 50% to 76% for FIFO depths of 1Kbits and 2Kbits. These proposed optimizations offer promising potential for reducing area budget while maintaining functionality.

6.1. Future Work

Future research should focus on further refining the ring-counter design to enhance its scalability and integration into larger and more complex systems. Currently, the ring-counter design requires manual entry of initialization streams, which presents an opportunity for automation. Developing methods to automate this initialization process would streamline the design flow and reduce manual errors.

Additionally, exploring further optimizations for BRAM decoders is crucial. Investigating alternative architectures and techniques for improving performance of BRAM decoders could lead to significant advancements in on-chip memory management.

Expanding research to encompass other on-chip memory resources and a wider range of application scenarios would be highly beneficial. Such an expansion could lead to the development of new optimization strategies for on-chip memory. This broader perspective would help in creating more effective and adaptable memory solutions for future hardware designs.

References

[1] E. Ahmed and J. Rose. “The effect of LUT and cluster size on deep-submicron FPGA performance and density”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.3 (2004), pp. 288–298. DOI: 10.1109/TVLSI.2004.824300.

[2] Taneem Ahmed, Paul D. Kundarewich, and Jason H. Anderson. “Packing Techniques for Virtex-5 FPGAs”. In: *ACM Trans. Reconfigurable Technol. Syst.* 2.3 (Sept. 2009). ISSN: 1936-7406. DOI: 10.1145/1575774.1575777. URL: https://doi.org/10.1145/1575774.1575777.

[3] Ryan W. Apperson et al. “A Scalable Dual-Clock FIFO for Data Transfers Between Arbitrary and Haltable Clock Domains”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15.10 (2007), pp. 1125–1134. DOI: 10.1109/TVLSI.2007.903938.

[4] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. “Architecture and CAD for Deep-Submicron FPGAS”. In: ().

[5] Vipul Bhatnagar, Chandani Attri, and Sujata Pandey. “Optimization of row decoder for 128×128 6T SRAMs”. In: *2015 International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI-SATA)*. 2015, pp. 1–4. DOI: 10.1109/VLSI-SATA.2015.7050451.

[6] Marion Bohr. “The evolution of scaling from the homogeneous era to the heterogeneous era”. In: *Technical Digest - International Electron Devices Meeting, IEDM* (Dec. 2011). DOI: 10.1109/IEDM.2011.6131469.

[7] Andrew Boutros and Vaughn Betz. “FPGA Architecture: Principles and Progression”. In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. DOI: 10.1109/MCAS.2021.3071607.

[8] Ireneusz Brzozowski, Łukasz Zachara, and Andrzej Kos. “Universal design method of n-to-2n decoders”. In: *Proceedings of the 20th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES 2013*. 2013, pp. 279–284.

[9] Cadence. *Flexible H-tree and Multi-Tap Clock Flow in Innovus (Legacy)(v20.10)*. 2020.

[10] Michael A. Cohen and Can Ozan Tan. “A polynomial approximation for arbitrary functions”. In: *Applied Mathematics Letters* 25.11 (2012), pp. 1947–1952. ISSN: 0893-9659. DOI: https://doi.org/10.1016/j.aml.2012.03.007. URL: https://www.sciencedirect.com/science/article/pii/S0893965912001607.

[11] Altera Corp. “Implementing RAM functions in FLEX 10K Devices (A-AN-052-01)”. In: (1995).

[12] Altera Corp. “Stratix II Device Handbook, Volume 1 (SII5V1-4.5)”. In: (2007).

[13] Altera Corporation. *Stratix II Device Handbook*. 2005.

[14] Altera Corporation. *Stratix® 10 TX Device Overview*. https://cdrdv2-public.intel.com/670455/s10_tx_overview-683717-670455.pdf. [Accessed 05-07-2024]. 2023.

[15] Lattice Semiconductor Corporation. *LatticeECP/EC Family Data Sheet*. 2005.

[16] *Embedded FPGA market-global industry analysis and forecast (2024-2030)*. July 2024. URL: https://www.maximizemarketresearch.com/market-report/global-embedded-fpga-market/27969/.

[17] *Global Field-Programmable Gate Array (FPGA) Market Report*. Feb. 2024. URL: https://market.us/report/fpga-market/#overview.

[18] Tom R. Halfhill. “Tabula’s Time Machine: Rapidly Reconfigurable Chips Will Challenge Conventional FPGAs”. In: *Microprocessor Rep* 131 (2010).

[19] David Money Harris et al. “The Fanout-of-4 Inverter Delay Metric”. In: 1998. URL: https://api.semanticscholar.org/CorpusID:9167634.

[20] Texas Instruments. *FIFO Architecture, Functions, and Applications*. 1999.

[21] *International technology roadmap for semiconductors, [Online]*. 2015. URL: http://www.itrs.net/.

[22] Wenjie Jiang et al. “Topological analysis for leakage prediction of digital circuits”. In: Feb. 2002, pp. 39–44. ISBN: 0-7695-1441-3. DOI: 10.1109/ASPDAC.2002.994882.

[23] Biby Joseph, Gopireddy Chaithanyakumar Reddy, and R. K. Kavitha. “Energy Efficient Memory Decoder for SRAM Based AI Accelerator”. In: *2023 2nd International Conference on Paradigm Shifts in Communications Embedded Systems, Machine Learning and Signal Processing (PCEMS)*. 2023, pp. 1–4. DOI: 10.1109/PCEMS58491.2023.10136095.

[24] Kelin Kuhn. “CMOS Transistor Scaling Past 32nm and Implications on Variation”. In: Aug. 2010, pp. 241–246. DOI: 10.1109/ASMC.2010.5551461.

[25] Kelin J. Kuhn. “Considerations for Ultimate CMOS Scaling”. In: *IEEE Transactions on Electron Devices* 59 (2012), pp. 1813–1828. URL: https://api.semanticscholar.org/CorpusID:11918304.

[26] Ian Kuon and Jonathan Rose. “Measuring the Gap Between FPGAs and ASICs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 203–215. DOI: 10.1109/TCAD.2006.884574.

[27] David Lewis et al. “Architectural enhancements in Stratix V™”. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’13. Monterey, California, USA: Association for Computing Machinery, 2013, pp. 147–156. ISBN: 9781450318877. DOI: 10.1145/2435264.2435292. URL: https://doi.org/10.1145/2435264.2435292.

[28] David Lewis et al. “Architectural enhancements in Stratix-III™ and Stratix-IV™”. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’09. Monterey, California, USA: Association for Computing Machinery, 2009, pp. 33–42. ISBN: 9781605584102. DOI: 10.1145/1508128.1508135. URL: https://doi.org/10.1145/1508128.1508135.

[29] David Lewis et al. “The Stratix II logic and routing architecture”. In: *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*. FPGA ’05. Monterey, California, USA: Association for Computing Machinery, 2005, pp. 14–20. ISBN: 1595930299. DOI: 10.1145/1046192.1046195. URL: https://doi.org/10.1145/1046192.1046195.

[30] Arvind Kumar Mishra, Debiprasad Priyabrata Acharya, and Pradip Kumar Patra. “Novel design technique of address Decoder for SRAM”. In: *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*. 2014, pp. 1032–1035. DOI: 10.1109/ICACCCT.2014.7019253.

[31] Jan M Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits*. Vol. 2. Prentice hall Englewood Cliffs, 2002.

[32] Ashish Shetty. “ASIC design flow and methodology–an overview”. In: *International Journal of Electrical and Electronics Engineering* 6.1 (2019), pp. 1–5.

[33] N. Shibata, M. Watanabe, and Y. Tanabe. “A current-sensed high-speed and low-power first-in-first-out memory using a wordline/bitline-swapped dual-port SRAM cell”. In: *IEEE Journal of Solid-State Circuits* 37.6 (2002), pp. 735–750. DOI: 10.1109/JSSC.2002.1004578.

[34] Aaron Stillmaker and Bevan Baas. “Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm”. In: *Integration* 58 (2017), pp. 74–81. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2017.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0167926017300755.

[35] Kosuke Tatsumura, Sadegh Yazdanshenas, and Vaughn Betz. “High density, low energy, magnetic tunnel junction based block RAMs for memory-rich FPGAs”. In: Dec. 2016, pp. 4–11. DOI: 10.1109/FPT.2016.7929181.

[36] ITRS Tech. rep. “FO4 writeup: International technology roadmap for semiconductors 2003 edition”. In: (2002).

[37] Scott Thompson et al. “In Search of “Forever,” Continued Transistor Scaling One New Material at a Time”. In: *Semiconductor Manufacturing, IEEE Transactions on* 18 (Mar. 2005), pp. 26–36. DOI: 10.1109/TSM.2004.841816.

[38] Stephen M. Trimberger. “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology”. In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331. DOI: 10.1109/JPROC.2015.2392104.

[39] John P Uyemura. “Introduction to VLSI circuits and systems”. In: (2002).

[40] Xilinx. *Virtex-4 Family Overview, 1.4 edition*. 2005.

[41] Inc. Xilinx. *7 Series FPGAs Memory Resources UG473 (v1.14)*. 2019.

[42] Inc. Xilinx. *UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices*. https://docs.amd.com/v/u/en-US/wp477-ultraram. [Accessed 05-07-2024]. 2016.

[43] Inc. Xilinx. *Vivado Design Suite Properties Reference Guide (UG912)*. 2024.

[44] Sadegh Yazdanshenas, Kosuke Tatsumura, and Vaughn Betz. “Don’t Forget the Memory: Automatic Block RAM Modelling, Optimization, and Architecture Exploration”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 115–124. ISBN: 9781450343541. DOI: 10.1145/3020078.3021731. URL: https://doi.org/10.1145/3020078.3021731.

[45] P.S. Zuchowski et al. “A hybrid ASIC and FPGA architecture”. In: Dec. 2002, pp. 187–194. ISBN: 0-7803-7607-2. DOI: 10.1109/ICCAD.2002.1167533.