A Single-Pass Timing-Analysis and Delay-Placement Framework for Asynchronous Bundled-Data Circuits on FPGA Platforms

by

Tamnuwat Valeeprakhon

A dissertation submitted to the

Engineering Course, Department of Engineering,

Graduate School of Engineering,

Kochi University of Technology,

Kochi, Japan

In partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering

Assessment Committee:

Supervisor: Prof. Makoto IWATA
Co-Supervisor: Prof. Kazutoshi YOKOYAMA
Co-Supervisor: Prof. Kiminori MATSUZAKI
Committee Member: Prof. Masayoshi TACHIBANA
Committee Member: Prof. Yukio MITSUYAMA

February 2026

Department of Engineering, Faculty of Engineering,
Graduate School of Kochi University of Technology
185 Miyanokuchi, Tosayamada, Kami City, Kochi 782-8502, JAPAN
Tel: +81 887 53 1130   Fax: +81 887 57 2000

Abstract

Asynchronous bundled-data (BD) circuits represent an important design paradigm for achieving modular, energy-efficient, and timing-robust computation in digital systems. Unlike conventional synchronous circuits, which rely on a globally distributed clock signal to coordinate operations, asynchronous circuits operate through local request–acknowledge handshaking between connected stages. This self-timed behavior allows each stage to run at its own data-dependent pace, eliminating global clock skew, reducing dynamic power associated with clock distribution networks, and improving resilience to variations in process, voltage, and temperature. The two-phase BD protocol in particular maintains a single-rail data path similar to synchronous architectures while enforcing correctness through the bundling constraint, which requires that the control path must always be slower than the corresponding data path. This balance between efficiency and practical implementability has made the BD protocol especially appealing for asynchronous processors and pipelines. However, despite their conceptual advantages, asynchronous BD circuits have historically remained difficult to implement on commercial FPGA platforms due to the clock-centric assumptions embedded within modern Electronic Design Automation (EDA) tools, especially in static timing analysis (STA) and delay placement.

FPGA architectures impose unique challenges not encountered in ASIC-oriented asynchronous design methodologies. In an FPGA, delays are realized through the discrete timing characteristics of lookup tables (LUTs), flip-flops, carry chains, and routing multiplexers. These components provide only coarse, unpredictable, and placement-dependent delay resolutions, which complicate the precise alignment required for correct BD operation. Conventional FPGA STA engines assume the presence of clock signals as the primary timing reference and lack direct support for expressing handshake causality, minimum-delay behavior, or relative timing constraints between data path and control-path components. As a result, the timing correctness of asynchronous circuits depends heavily on manually constructed constraints, hand-tuned delay elements, and repeated place-and-route iterations. Slight changes in routing decisions can shift critical delays enough to violate handshake sequencing or bundling requirements, further complicating convergence. Consequently, asynchronous circuit designers have traditionally relied on iterative timing-closure methodologies that demand substantial expertise, manual effort, and extensive experimentation. These conventional approaches, including Adaptive Delay Matching (ADM), Propagation Timing Constraint (PTC), and Backward Delay Propagation Constraint (BDPC), attempt to bridge the gap between handshake semantics and synchronous STA tools but each suffers from inherent limitations such as unsupported minimum-delay constraints, incorrect assumptions regarding phase propagation, inability to properly model loop behavior, and dependence on post-layout Engineering Change Order (ECO) operations that jeopardize timing reproducibility.

To address these critical challenges and enable the practical implementation of asynchronous BD circuits on FPGA platforms without requiring modifications to commercial tools, this dissertation proposes a unified design framework that integrates two complementary methodologies: the Generated Clock Propagation (GCP) method for timing analysis and the Morphological Delay-Placement Constraint (MDPC) method for delay insertion. Together, these methods create a semi-automated, deterministic, and FPGA-compatible flow that achieves timing closure in a single synthesis pass while maintaining high precision, scalability, and resource efficiency. The central motivation behind this framework is to reconcile the self-timed behavior of two-phase BD circuits with the synchronous assumptions of FPGA STA engines by constructing a mapping that preserves handshake semantics, enables robust slack computation, and ensures placement-aware delay balancing.

The GCP method in this thesis reframes handshake timing behavior into a form that can be interpreted by existing STA engines without modifying tool internals. Instead of treating handshake signals as arbitrary, tool-agnostic transitions, GCP interprets each request event as a virtual clock domain. By constructing a network of generated clocks, the timing relationship between handshake events can be expressed using standard commands such as create_clock, create_generated_clock, set_clock_groups, and set_false_path. Through this formulation, GCP allows the STA engine to evaluate setup and hold constraints across complex asynchronous networks in exactly the same manner as synchronous multi-clock systems. The propagation of virtual clocks follows the causal structure of the handshake network, enabling the STA engine to derive accurate timing paths and slack

values. An important aspect of GCP is its ability to identify and isolate architectural loops, such as those found in sequential-ring circuits or iterative feedback pipelines, as well as local combinational loops within handshake controllers. Without such loop isolation, STA engines tend to propagate generated clocks indefinitely, leading to infinite or unstable slack values. GCP strategically disables non-functional timing arcs, prunes loop-back paths, and stabilizes propagation to yield correct, finite timing information. Furthermore, the method correctly handles conditional behavior through selective activation of handshake branches, enabling the STA engine to analyze only one active path at a time in circuits involving MUX, DEMUX, fork, join, and passive controllers. This per-path analysis is particularly important in architectures employing multiple selection-delay channels, such as frequency-adaptive pipelines or multi-mode processors. By fully leveraging native STA commands, GCP achieves accurate and loop-safe timing analysis for asynchronous circuits, providing consistent results regardless of placement randomness or routing variations.

The second contribution, the MDPC method, addresses the complementary challenge of timing closure through physically aware delay insertion. Traditional asynchronous FPGA flows often rely on hand-crafted delay chains or heuristics that place delay elements arbitrarily near control-path logic. However, because FPGA placement heavily influences routing delays, manually inserted delay lines frequently fail to match data path timing precisely or result in unstable timing behavior across synthesis iterations. MDPC introduces a novel placement strategy rooted in morphological image analysis to determine optimal delay locations. The method begins by representing implemented circuit regions as geometric patterns on the FPGA grid. Using dilation, erosion, and subtraction operations, MDPC identifies the physical boundary between datapath logic and control-path logic, forming a candidate region where delay elements would have the greatest likelihood of producing predictable timing behavior. Once candidate regions are extracted, MDPC performs slack-guided refinement: for each candidate tile, a temporary delay element is inserted, and slack is extracted. The candidate producing a slack value closest to the target requirement is selected as the optimal insertion site. This iterative evaluation continues until the desired control delay is achieved. Because MDPC relies directly on timing information obtained after placement and routing, its choices reflect the true physical characteristics of the design, rather than inferred or estimated delay behavior. As a result, MDPC achieves deterministic timing closure without requiring repeated design iterations or post-layout manual editing. This single-pass property marks a substantial departure from previous methodologies and significantly improves both design efficiency and precision.

The unified GCP + MDPC framework forms a complete asynchronous FPGA design flow compatible with conventional FPGA toolchains. The flow begins with HDL design entry using standard two-phase handshake primitives such as Click and its phase-decoupled variants. After initial synthesis produces a technology-mapped netlist, GCP generates timing constraints that accurately reflect handshake dependencies, loop structures, and selection behaviors. Placement and routing proceed under these constraints, ensuring that timing relationships remain stable across implementation. MDPC then inserts delay elements within the physically meaningful boundary regions guided by slack feedback. Final STA confirms that all setup and hold requirements are satisfied, after which the standard toolchain generates the FPGA bitstream. Throughout this process, no modifications to the underlying tools are required, and the entire flow is reproducible across routing seeds and device families.

The effectiveness of the proposed framework is demonstrated through its application to five asynchronous RISC-V processor pipelines, including linear, selective, frequency-adaptive, reduced, and combined styles, each representing different architectural structures and handshake characteristics. These designs encompass a broad range of behavioral patterns, such as pure feed-forward pipelines, pipelines incorporating conditional selection-delay channels, adaptively controlled timing paths, reduced-latency data paths, and hybrid architectures that integrate multiple asynchronous mechanisms. Experimental evaluation on Xilinx FPGA platforms using Coremark confirms that the GCP + MDPC flow achieves functional correctness across all implementations while substantially improving timing accuracy, performance, and energy efficiency compared with conventional iterative flows. Across the evaluated pipeline styles, processing speed improved by up to 13.50%, and energy consumption decreased by as much as 11.59%. Timing behavior remained stable, with slack deviations consistently within 1.45%. Furthermore, the flow significantly reduced hardware overhead, lowering LUT usage for delay elements by 3.76–5.56×. Collectively, these results highlight the practical advantages of the proposed

methodology and establish its suitability for scalable and deterministic asynchronous processor design on FPGA platforms.

Beyond quantitative gains, the experimental evaluation also highlights the conceptual advantages of the proposed framework. By semi-automating both timing analysis and delay placement, the methodology eliminates the historically difficult and error-prone steps of manual constraint editing and delay tuning. Because delay insertion is guided by physically grounded geometric analysis, MDPC maintains timing robustness across routing or synthesis variations, significantly improving reproducibility. At the same time, the GCP method transforms asynchronous STA into a representation that mainstream FPGA tools can naturally interpret, enabling designers to analyze complex handshake pipelines without custom scripts or external analyzers. Building upon these capabilities, this dissertation establishes a comprehensive, deterministic, and FPGA-compatible design flow for asynchronous BD circuits. GCP resolves long-standing limitations in modeling loops, conditional paths, and multi-channel timing behavior, while MDPC complements it by automating delay insertion in a single implementation pass through geometry-aware extraction and slack-driven refinement. Together, these techniques bridge the conceptual gap between self-timed design principles and the synchronous assumptions embedded within commercial FPGA tools. Through extensive validation across multiple asynchronous RISC-V pipeline architectures, the combined framework demonstrates that self-timed processors can be implemented with consistent timing, reduced resource usage, improved performance, and enhanced energy behavior, all within unmodified FPGA toolchains, showing that the gap between asynchronous theory and FPGA practice can be substantially narrowed and positioning asynchronous computation for broader use in future low-power, high-efficiency, and adaptive digital systems.

Preface

This thesis was carried out at the Graduate School of Engineering under the supervision of Professor Makoto Iwata from April 2023 to March 2026. The research focuses on developing an automated design flow for asynchronous bundled-data (BD) circuits on FPGA platforms, integrating the Generated Propagation Clocks (GCP) and Morphological Delay-Placement Constraint (MDPC) methods to achieve efficient and reliable implementation.

I would like to express my deepest gratitude to Professor Makoto Iwata for his continuous guidance, insightful discussions, and unwavering encouragement throughout my doctoral studies. I am also sincerely grateful to my Co-Supervisors, Professor Kazutoshi Yokoyama and Professor Kiminori Matsuzaki, for their valuable suggestions, constructive feedback, and academic support. My sincere appreciation is extended to the committee members, Professor Masayoshi Tachibana and Professor Yukio Mitsuyama, for their careful review, thoughtful comments, and kind advice, which greatly improved the quality of this thesis.

I would also like to gratefully acknowledge the Royal Thai Government Scholarship for its generous financial support throughout my doctoral program. My sincere thanks are extended to Kasetsart University for granting me the opportunity to pursue my doctoral studies and for its continuous institutional support. Finally, I extend my heartfelt appreciation to my laboratory members, family, and friends for their understanding, encouragement, and constant support, which made this work possible.

Valeeprakhon Tamnuwat

Table of Contents

Abstract I
Preface IV
Table of Contents V
Table of Figures VII
Chapter 1 Introduction 1
1.1 Research Objectives 2
1.2 Proposed Solution Overview 2
1.3 Dissertation Organization 3
Chapter 2 Asynchronous BD Circuits on FPGA and Their Existing Design Flows 4
2.1 Two-Phase Bundled-Data Protocol 5
2.2 Click-based Handshake Controllers 6
2.2.1 Handshake Components in Click-Based Systems 6
2.3 Delay Elements 7
2.3.1 Matched Delay Elements 8
2.3.2 Selection Delay Elements 8
2.4 Asynchronous Pipeline Architecture 8
2.5 Static Timing Analysis 10
2.5.1 Setup and Hold Time Requirements 10
2.5.2 Slack Calculation and Timing Margins 11
2.5.3 Critical Path Analysis 11
2.5.4 Timing Analysis in Two-Phase Asynchronous Bundled-Data Circuits 11
2.6 FPGAs 13
2.7 FPGA Circuit Design Flow 14
2.8 Asynchronous Circuit Design Flow 15
2.8.1 Iterative Synthesis Design Flow 16
2.8.2 Single-Pass Synthesis Design Flow 20
2.8.3 Comparison of Asynchronous Design Flows 21
2.9 Delay Placement Approaches 25
2.9.1 Iterative Synthesis Approaches 25
2.9.2 Engineering-Change-Order (ECO) Based Approaches 25
2.9.3 Limitation of Existing Delay Placement Approaches 26
Chapter 3 Generated Clock Propagation Constraint 28
3.1 Motivation 29
3.2 Proposed Static Timing Analysis Constraint 29
3.3 Example of STA for Timing Path Extraction 31

3.3.1 Iterative Fibonacci Pipeline: Handling An Architectural Loop 31
3.3.2 Conditional Handshake Circuit: Passive Controller Interpretation 31
3.3.3 Linear Pipeline with Selection Delay: Multi-Channel Timing Semantics 32
3.4 Example Applications 40
3.4.1 FIR Filter Circuit: Linear Pipeline 40
3.4.2 Fibonacci (FIBO) Generator: Sequential Ring Pipeline 41
3.4.3 AES Encryption: Iterative Ring Pipeline 42
3.5 Comparative Evaluation with Existing Methods 43
Chapter 4 Morphological Delay Placement Constraint 45
4.1 Motivation 46
4.2 Proposed Delay Placement Constraint 48
4.3 Computational Complexity 51
4.4 Comparative Evaluation with Existing Methods 51
Chapter 5 Experiment on RISC-V Processors 56
5.1 Pipeline Style Literature Review 57
5.2 RISC-V Processor Architecture 57
5.3 Pipeline-Style Implementations 58
5.3.1 Linear Pipeline 58
5.3.2 Selective Pipeline 58
5.3.3 Frequency-Adaptive Pipeline 59
5.3.4 Combined Pipeline 59
5.3.5 Reduced Pipeline 59
5.4 Benchmarking Methodology 63
5.5 Experimental Results 63
5.6 Limitation Analysis of Pipeline Styles 70
5.7 Pipeline-to-Application Mapping 70
Chapter 6 Discussion and Conclusion 72
6.1 Discussion of Integration of GCP and MDPC 72
6.1.1 Design Efficiency and Determinism 72
6.1.2 Quantitative Improvements and Resource Behavior 72
6.1.3 Scalability and Pipeline Diversity 73
6.1.4 Limitations and Future Considerations 73
6.2 Conclusion 73
6.3 Future works 74
Appendices 75
References 83

Table of Figures

Figure 2.1. Structure of a two-phase BD pipeline stage. 5

Figure 2.2. Logic structures of the standard Click and the Phase-Decoupled Click element. 6

Figure 2.3. Asynchronous BD pipeline. 7

Figure 2.4. Linear pipeline structure. 9

Figure 2.5. Sequential ring pipeline structure. 9

Figure 2.6. Iterative ring pipeline structure. 9

Figure 2.7. General setup and hold timing of synchronous circuits. 10

Figure 2.8. Timing constraints of the phase-decoupled pipeline. 12

Figure 2.9. Simplified architecture of a modern FPGA. 13

Figure 2.10. General FPGA design flow using EDA tools. 14

Figure 2.11. The iterative synthesis flow. 17

Figure 2.12. Single-pass synthesis flow. 20

Figure 3.1. The structure of the FIBO circuit, which applies STA by using the GCP. 32

Figure 3.2. The example of the STA command of GCP on the FIBO circuit, which has architecture loop issues. 33

Figure 3.3. The structure of the conditional handshake circuit. 34

Figure 3.4. The example of the STA command of GCP on the passive conditional handshake pipeline. 35

Figure 3.5. The structure of a linear pipeline with a selection delay. 37

Figure 3.6. The example of the STA command of GCP on the selection delay. 38

Figure 3.7. The example of a timing report. 39

Figure 3.8. Block diagram of the FIR filter asynchronous circuit. 41

Figure 3.9. Block diagram of the FIBO asynchronous circuit. 42

Figure 3.10. Block diagram of the AES asynchronous circuit. 43

Figure 4.1. The MDPC process. 47

Figure 4.2. The dilation process for shrinking the boundary region. 48

Figure 4.3. The erosion process for shrinking the boundary region. 48

Figure 4.4. The subtraction process for extracting the boundary region. 49

Figure 4.5. Pseudo code of MDCP. 50

Figure 4.6. Comparison of the average worst slack value of every timing path at different target slack values. 52

Figure 4.7. Comparison of the worst slack value distributions at different target slack values. 52

Figure 4.8. Comparison of LUTs usage for inserting delay at different target slack values. 53

Figure 4.9. Comparison of LUTs usage in average for inserting delay elements. 53

Figure 4.10. Comparison of execution time. 54
Figure 4.11. Comparison of energy. 54
Figure 5.1. Architecture of an asynchronous RISC-V processor with a hazard unit and an asynchronous controller. 57
Figure 5.2. Linear pipeline structure. 60
Figure 5.3. Selective pipeline structure. 61
Figure 5.4. Frequency-adaptive pipeline structure. 62
Figure 5.5. Combined pipeline structure. 63
Figure 5.6. Reduced pipeline structure. 63
Figure 5.7. Comparison of the power consumption of each power distribution usage based on 1500 CoreMark. 65
Figure 5.8. Comparison of execution time based on 1500 CoreMark of different pipelines. 66
Figure 5.9. Comparison of energy usage based on 1500 CoreMark of different pipelines. 66
Figure 5.10. Comparison of area usage based on 1500 CoreMark of different pipelines. 67
Figure 5.11. Comparison of EDAP usage based on 1500 CoreMark of different pipelines. 67
Figure 5.12. Execution time comparison for different testbenches among pipeline styles. 68
Figure 5.13. Energy consumption comparison for different testbenches among pipeline styles. 68
Figure 5.14. Area normalization comparison for different testbenches among pipeline styles. 69
Figure 5.15. Comparative radar chart of normalized metrics (execution time, energy, area, and EDAP). 69
Figure 5.16. Comparative radar chart of normalized EDAP. 69
Figure A.1. The structure of the delay element aligns with the column pattern. 78
Figure A.2. The structure of the delay element aligns with the row pattern. 78
Figure A.3. The structure of the delay element aligns with the row-column pattern. 79

Table of Tables

Table 2.1. Timing sequence of a BD channel. 5

Table 2.2. Common handshake components in Click-based asynchronous BD circuits. 7

Table 2.3. Key architectural components of an FPGA. 14

Table 2.4. Comparison of synchronous, iterative, asynchronous, and single-pass asynchronous design flows. 22

Table 2.5. Summarizes representative STA methods for asynchronous BD circuits. 23

Table 2.6. Comparison of limitations in existing delay placement approaches. 27

Table 3.1. Timing paths of the FIR circuit with final setup and hold timing results. 41

Table 3.2. The comparison results between synchronous and asynchronous FIR circuits implemented using GCP. 41

Table 3.3. Timing paths of the FIBO circuit with final setup and hold timing results. 42

Table 3.4. The comparison results between synchronous and asynchronous FIBO circuits implemented using GCP. 42

Table 3.5. Timing paths of the AES circuit with the final setup and hold timing result. 43

Table 3.6. The comparison results between synchronous and asynchronous AES circuits implemented using GCP. 43

Table 3.7. Comparison of Timing-Analysis Methods for Asynchronous BD Circuits. 44

Table 4.6. Comparison of Delay-Placement and Design-Flow Methods. 55

Table 5.1. Pipeline-To-Application Design Map for Asynchronous RISC-V Pipelines. 71

Table A.1. Common handshake components, their structures, and symbols. 75

Table A.2. Summary of Representative Asynchronous Processor Implementations. 80

Table A.3. Comparison of the worst slack value distributions at different target slack values. 83

Table A.4. Comparison of LUTs usage for inserting delay at different target slack values. 83

Table A.5. Comparison of the execution speed of RISC-V. 83

Table A.6. Comparison of the energy consumption of RISC-V. 84

Table A.7. Execution time comparison for different testbenches among pipeline styles. 84

Table A.8. Energy consumption comparison for different testbenches among pipeline styles. 85

Table A.9. Area normalization comparison for different testbenches among pipeline styles. 85

Table A.10. EDAP comparison for different testbenches among pipeline styles. 86

Chapter 1
Introduction

Modern digital systems increasingly demand high performance, low power consumption, and stable operation under variations in process, voltage, and temperature. Conventional synchronous circuits rely on a global clock to coordinate events across the chip, but this paradigm is becoming harder to maintain as system complexity grows. Clock distribution incurs significant power overhead, timing closure becomes more difficult in large designs, and global synchronization ultimately limits scalability.

Asynchronous circuits address these issues by eliminating the global clock and relying instead on local handshaking protocols to manage data flow between pipeline stages. Each stage operates independently, using request and acknowledge transitions to coordinate transfers. This self-timed behavior provides several advantages: modularity that allows stages to be designed and verified independently, event-driven energy efficiency, resilience to PVT variations, and natural adaptability that allows each stage to run at its own workload-dependent speed.

Among asynchronous design styles, the BD protocol stands out for its practicality. Unlike multi-rail encodings, BD maintains a standard single-rail data path similar to synchronous systems, while relying on the bundling constraint to ensure timing correctness. By inserting suitable delay elements in the control path, designers ensure that control transitions occur only after data has stabilized. This approach allows BD circuits to be implemented using conventional FPGA tools and standard cell libraries, making them well-suited for rapid prototyping.

Despite these benefits, two key challenges hinder the widespread use of asynchronous BD circuits on FPGA platforms. The first challenge arises from tool limitations: commercial FPGA flows are optimized for synchronous designs, and their STA engines cannot directly interpret handshake dependencies. Therefore, existing BD methodologies rely on manual constraint editing and repeated synthesis iterations, requiring considerable designer expertise and yielding unpredictable timing convergence. The second challenge concerns the precise alignment between control-path and data-path delays. For correct operation, the control path must be sufficiently slower than the data path, but achieving this on FPGAs is difficult because delay resolution is tied to discrete LUT and routing resources. Small placement variations can introduce large delay differences, making manual tuning error-prone and iterative approaches inconsistent. This issue becomes even more pronounced in advanced architectures, such as frequency-adaptive pipelines, where multiple delay channels must be individually characterized.

To address these limitations, this dissertation proposes a semi-automated design framework that integrates two complementary methodologies. The Generated Clock Propagation (GCP) method enables accurate timing analysis by mapping handshake dependencies into virtual clocks, allowing standard FPGA STA tools to analyze linear, sequential-ring, and iterative-ring architectures while correctly handling architectural loops, local controller loops, conditional paths, and multiple selection-delay channels. The Morphological Delay-Placement Constraint (MDPC) method automates delay insertion using geometry-aware placement, applying morphological boundary detection and slack-driven refinement to achieve precise control-data alignment in a single synthesis pass. Together, these methods provide deterministic timing closure, reduce design effort, and improve both performance and energy efficiency compared with conventional iterative flows.

1.1 Research Objectives

This dissertation aims to establish a unified and semi-automated methodology that integrates asynchronous design principles with existing FPGA tools and workflows. By combining accurate timing analysis with physically aware delay placement, the goal is to provide a deterministic and reproducible flow for implementing complex asynchronous systems. The proposed framework is intended not only to simplify the design process but also to broaden the practical feasibility of self-timed processors and adaptive computing architectures on modern FPGA platforms. The research objectives are as follows:

1. To develop an FPGA-compatible design flow for asynchronous BD circuits

   The first objective is to construct a systematic methodology that operates fully within commercial FPGA environments, specifically Xilinx Vivado, without relying on custom timing analyzers or tool modifications. The flow must remain compatible with standard EDA processes while supporting the unique requirements of asynchronous BD designs.

2. To establish the GCP method for accurate timing analysis using standard tools

   This objective focuses on developing the GCP method, which interprets handshake signals as virtual propagation clocks. The method must enable accurate setup and hold verification through standard STA commands while handling complex circuit topologies, including local and architectural loops, conditional paths, and multi-channel selection delay structures.

3. To propose the MDPC method for single-pass, resource-efficient delay placement

   The third objective is to design the MDPC methodology, which performs delay insertion in a single synthesis pass. Using morphological boundary detection and slack-guided refinement, the method must identify optimal delay locations, reduce resource usage, improve timing accuracy, and ensure deterministic convergence independent of designer experience.

4. To implement and compare multiple asynchronous RISC-V pipelines using the proposed flow

   This objective validates the overall framework by applying it to several asynchronous RISC-V pipeline styles, linear, selective, frequency-adaptive, reduced, and combined. The implementations must demonstrate functional correctness, timing closure, and robustness across different architectural configurations.

5. To evaluate design trade-offs of asynchronous RISC-V pipelines in performance, energy, and area efficiency

   The final objective is to analyze and quantify the performance, energy behavior, and resource usage of the implemented pipelines. This includes examining architectural trade-offs, providing application-oriented design guidelines, and comparing the results with synchronous baselines as well as other asynchronous methodologies.

1.2 Proposed Solution Overview

To address the challenges discussed in the previous section, this dissertation proposes a comprehensive design framework built on two complementary methodologies that together enable semi-automated and reliable implementation of asynchronous circuits on FPGA platforms.

1. Generated Clock Propagation (GCP) for Timing Analysis

   The GCP method reframes timing analysis by interpreting each handshake event as a local clock domain. It begins with virtual clock propagation, defining each request transition as a generated clock whose phase relationships propagate through the handshake network. Loop isolation then detects and separates both architectural loops, such as those in iterative-ring circuits, and local loops inside handshake controllers. The method also resolves conditional paths by analyzing passive controllers, including MUX, DEMUX, fork, and join elements, through selective path activation. For circuits using

selection delay, each delay channel is analyzed individually using per-channel timing constraints. Importantly, all steps operate entirely within standard FPGA tools, relying solely on native STA commands without requiring any modification to the toolchain.

2. Morphological Delay Placement Constraint (MDPC) for Delay Management

   MDPC automates delay insertion through a geometry-aware placement strategy. It begins with boundary detection, applying morphological operations, dilation, erosion, and subtraction, to identify candidate regions around module boundaries for delay placement. Slack-guided refinement then evaluates each candidate location by temporarily inserting a delay element and measuring the resulting slack. The location that brings the slack closest to the target value is permanently selected, and the process iterates to construct delay chains as needed. Because MDPC continuously refines placement based on real timing feedback, it achieves deterministic timing closure within a single synthesis pass.

3. Integration and Workflow

   The combined GCP + MDPC flow proceeds through a structured series of steps. The process begins with HDL design entry, describing the asynchronous circuit using explicit handshake controllers. Initial synthesis preserves the handshake structure while generating the logic netlist. GCP then produces timing constraints based on the handshake topology, enabling conventional placement and routing with timing-aware optimization. MDPC is followed by performing automated boundary analysis and iterative delay placement. Final timing verification ensures that all setup and hold constraints are satisfied. Once timing closure is confirmed, bitstream generation produces the configuration file used to program the FPGA.

1.3 Dissertation Organization

To address the research objectives described above, this dissertation is organized to progress from fundamental concepts toward practical implementation and evaluation. Each chapter builds on the previous one, introducing the theoretical background, design methodologies, and experimental results that form the basis of the proposed framework. The structure is intended to guide readers from the principles of asynchronous BD circuits to the complete realization of automated FPGA-based asynchronous processors.

Chapter 2: introduces the fundamentals of asynchronous bundled-data circuits, provides an overview of FPGA architecture relevant to their implementation, and asynchronous design flows.

Chapter 3: presents the GCP method for automated and loop-safe timing analysis using standard FPGA tools.

Chapter 4: details the MDPC method for single-pass, geometry-aware delay placement.

Chapter 5: demonstrates the proposed framework through a comprehensive case study of asynchronous RISC-V pipelines, including performance, energy, and area evaluation.

Chapter 6: discusses the integration of GCP and MDPC, summarizes key insights, and reflects on the broader implications of the proposed methodology.

Chapter 7: concludes the dissertation and outlines potential directions for future research.

Chapter 2
Asynchronous BD Circuits on FPGA and Their Existing Design Flows

Asynchronous bundled-data (BD) circuits represent a class of digital systems that operate without a global clock, using localized handshaking for synchronization between stages. Each communication channel consists of a single-rail data path paired with request and acknowledge signals, which together regulate data transfers across the circuit [1]. This decentralized timing style allows each stage to react to actual computation and propagation conditions, offering benefits in modularity, scalability, and energy efficiency compared with conventional clocked designs [2].

The bundled-data approach relies on the bundling constraint, which requires the control-path delay to be greater than or equal to the corresponding data-path delay. When this condition is satisfied, the circuit guarantees proper sequencing of data movement even without a global clock. In practice, handshake controllers such as Click and phase-decoupled Click elements generate local synchronization pulses that orchestrate data capture and release. These controllers serve as the building blocks for a wide range of asynchronous pipelines, including linear, sequential-ring, and iterative-ring architectures. Implementing such circuits on FPGAs, however, introduces specific challenges because commercial design tools are optimized for synchronous systems built around a global clock [3].

This chapter lays the foundational concepts required to understand asynchronous bundled-data (BD) circuits and their implementation on FPGA platforms. Section 2.1 introduces the two-phase BD protocol, which governs event-driven communication through request–acknowledge signaling. Section 2.2 presents Click-based handshake controllers and their phase-decoupled variants, followed by Section 2.2.1, which describes the functional roles of individual handshake components used to construct complex pipelines. Section 2.3 examines delay elements, matched delays, and selection delays that enforce the bundling constraint and maintain correct timing relationships between data and control paths. Section 2.4 outlines major asynchronous pipeline architectures, including linear, sequential-ring, and iterative-ring organizations. Section 2.5 reviews static timing analysis (STA) fundamentals and explains setup/hold constraints, slack margins, critical paths, and timing analysis techniques specialized for two-phase BD circuits. Sections 2.6 and 2.7 provide an overview of FPGA architectures and the conventional FPGA circuit design flow. Section 2.8 extends this flow to asynchronous circuits, detailing iterative synthesis, single-pass synthesis methodologies, and their conceptual differences. Finally, Section 2.9 surveys delay-placement approaches, beginning with iterative synthesis-based methods, establishing the background for the automated techniques introduced in later chapters.

2.1 Two-Phase Bundled-Data Protocol

In asynchronous digital systems, data transfers are coordinated through local handshaking rather than a global clock. The BD protocol employs a single-rail data path alongside request and acknowledge control signals, which together regulate communication between adjacent stages [4]. When valid data are available, the sender toggles the request signal, and the receiver responds by toggling the acknowledge signal once the data have been captured. This simple mechanism provides reliable point-to-point synchronization while avoiding the complexity of multi-rail encodings [5].

Central to the BD style is the bundling constraint, which requires the control-path delay to be greater than or equal to the corresponding datapath delay. Ensuring this relationship guarantees that control transitions occur only after data signals have stabilized, preventing premature sampling. In practice, this constraint is satisfied by inserting matched delay elements in the control path. The two-phase, or transition-signaling, version of the BD protocol was introduced in Sutherland's Micropipelines [6]. In this scheme, every transition, rising or falling, on the request or acknowledge lines represents an event. Unlike the four-phase protocol, which requires explicit return-to-zero phases, the two-phase approach completes each transfer with only two transitions, improving throughput and reducing unnecessary switching. This efficiency makes two-phase signaling well-suited for high-performance asynchronous pipelines [6].

A complete handshake cycle unfolds as follows: the sender first places stable data on the data path and toggles the request line to indicate availability. The receiver captures the data and then toggles the acknowledge line. After detecting this acknowledge transition, the sender prepares the next data item. Each request–acknowledge transition pair completes one handshake, ensuring stable data capture and adherence to the bundling constraint [7].

Table 2.1. *Timing sequence of a BD channel showing how the request and acknowledge signals coordinate each data transfer.*

Step	Description	Signal State
1	The sender places valid data on the bus.	Data valid
2	The sender toggles **Req** to indicate that new data are available.	Req ↑/↓
3	The receiver captures the data and responds by toggling **Ack**.	Ack ↑/↓
4	After detecting the **Ack** transition, the sender prepares the next data item.	Req ready

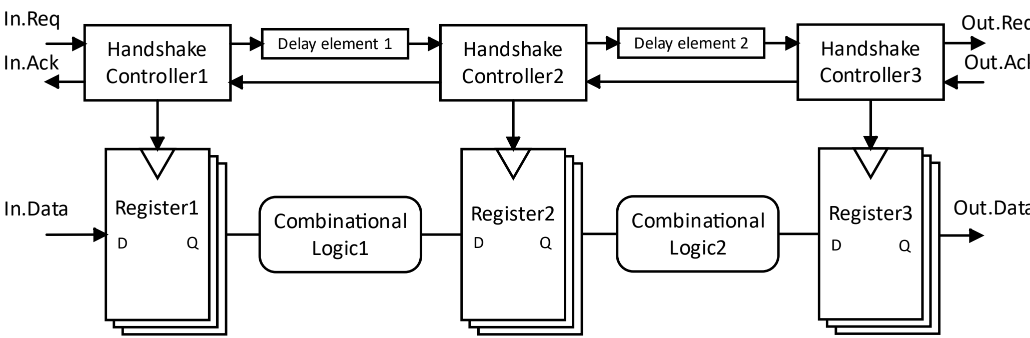


Figure 2.1. *Structure of a two-phase BD pipeline stage. The data path consists of a register (DFF) followed by combinational logic, while the control path uses request and acknowledge signals together with a matched delay element to ensure that the bundling constraint is satisfied.*

Table 2.1 summarizes the timing sequence of a BD communication channel. In this protocol, the request and acknowledge signals coordinate each data transfer while ensuring that the control-path delay satisfies the bundling constraint. As a result, the receiving register always samples stable data at the moment a handshake

event occurs. In a two-phase pipeline stage, the handshake controller monitors these transitions and generates a local timing pulse, commonly referred to as the Fire signal, which triggers data capture in the register. Figure 2.1 illustrates the structure of a two-phase BD pipeline stage and highlights the interaction between the control and data paths. The data path consists of a register and a block of combinational logic, while the control path relies on request and acknowledge signals together with a matched delay element to maintain correct timing behavior [8].

2.2 Click-based Handshake Controllers

The Click element serves as a fundamental two-phase handshake controller in asynchronous BD circuits. It effectively replaces the global clock by generating a short firing pulse whenever a valid handshake event occurs [9]. Internally, a Click controller is built from simple logic components, typically an XOR gate, an AND gate, and a single flip-flop. The XOR gate detects transitions between the request and acknowledge signals, while the flip-flop records the current phase. When Req ≠ Ack, the XOR output asserts the Fire signal, which enables the local data register and toggles the outgoing handshake lines. The logic structure of the standard Click element is shown in Figure 2.2. To improve initialization behavior and enhance robustness in cyclic pipelines, Mardari et al. introduced the Phase-Decoupled Click element [10]. In contrast to the original Click, which uses a single flip-flop to couple input and output phases, the Phase-Decoupled version employs two independent flip-flops, one storing the input phase and the other storing the output phase. This decoupling prevents phase conflicts in sequential-ring and iterative-ring pipelines, enabling deterministic token initialization during startup.

During each handshake, the Phase-Decoupled Click operates in a manner similar to the standard version. The XOR gate detects a phase difference (Req ≠ Ack) and generates a *Fire* pulse, which triggers the data register to capture incoming data. The controller then toggles Out.Req and In.Ack, completing the handshake. Although both variants behave identically under steady-state operation, the Phase-Decoupled Click provides improved initialization control and greater stability in ring-based pipelines, while maintaining throughput comparable to the original design.

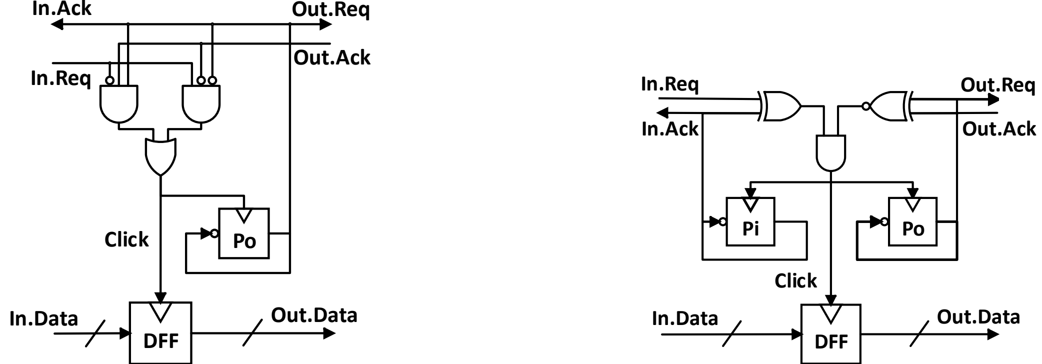


Figure 2.2. *Logic structures of (a) the standard Click element and (b) the Phase-Decoupled Click element.*

2.2.1 Handshake Components in Click-Based Systems

Phase-Decoupled Click–based controllers are modular and can be composed with a variety of functional elements to form complete asynchronous pipeline stages. Each element manages its own handshake behavior according to its role in the data path, and together these components govern the direction, synchronization, and sequencing of data tokens within a BD system [11]. Table 2.2 summarizes the commonly used handshake components and their functions, while Appendix Table 1 provides their circuit structures and symbols.

Table 2.2. Common handshake components in Click-based asynchronous BD circuits.

Component	Function	Data Flow
Source	Generates initial request signals and injects data tokens into the pipeline.	Input → Pipeline
Handshake Register	Stores and forwards data when input and output conditions are satisfied.	Forward
Fork	Duplicates one input token to multiple outputs and waits for all acknowledgments to return.	Divergent
Join	Waits for tokens from all inputs before issuing a single output request.	Convergent
Merge	Selects one of several inputs and forwards it to the output.	Selective
MUX / DEMUX	Routes data conditionally under handshake control.	Conditional
Sink	Receives final data tokens and returns acknowledgments to close the handshake chain.	Pipeline → Output

2.3 Delay elements

Delay elements play a crucial role in BD circuits by aligning the arrival of control signals with the availability of valid data. This alignment prevents premature latching and ensures that operations proceed in the correct sequence, thereby preserving functional correctness [12]. Because data requires time to propagate through combinational logic, delay elements are inserted into request and acknowledge paths to ensure that control transitions occur only after the corresponding data signals have stabilized. Delay elements in BD pipelines are generally classified into matched delays and selection delays. Both serve to synchronize control signals with data readiness, but they differ in their placement strategy and timing behavior. As illustrated in Figure 2.3, a matched delay (MD) provides a fixed delay that approximates the data path latency, while a selection delay (SD) incorporates multiple delay channels and uses a multiplexer to choose the appropriate delay depending on the operation being performed. This mechanism allows pipelines to adapt dynamically to varying computational latencies and supports more flexible timing behavior.

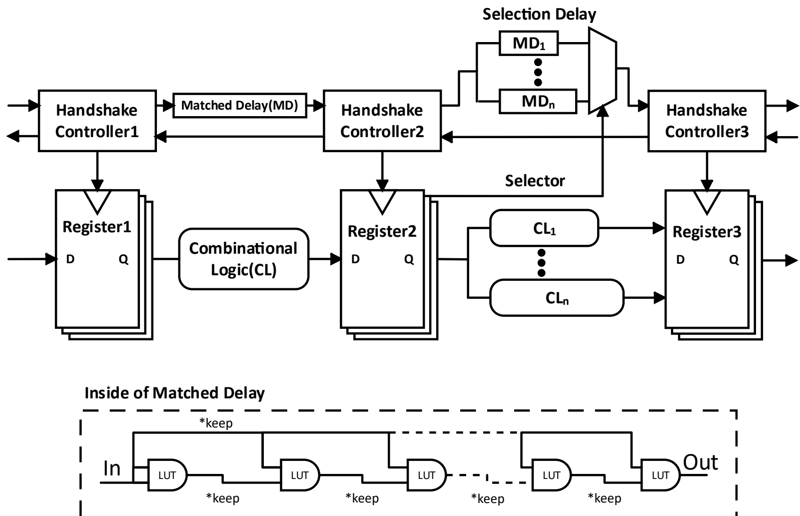


Figure 2.3. Asynchronous BD pipeline showing matched delay and selection delay. The selection unit uses a multiplexer to choose one of several delay channels based on operation latency.

2.3.1 Matched Delay Elements

A matched delay aligns the control-path delay with the longest data path delay within a pipeline stage, thereby ensuring that the setup-time constraint is satisfied. On FPGAs, matched delays are typically implemented using cascaded LUTs configured as buffers. The standard approach constructs a chain of LUTs with carefully controlled relative placement on the FPGA fabric to achieve stable and reproducible delay characteristics [13]. Each LUT in the chain is implemented as a single-input buffer, specifically, a LUT1 initialized with the truth table *"10"*. This configuration ensures that the LUT performs no logical transformation and functions purely as a delay element. Achieving consistent delay values depends heavily on the precise physical placement of these LUTs. The *rloc* (relative location) attribute allows designers to specify the relative slice positions using Cartesian coordinates (X#Y#), thereby constraining the spatial layout of the LUT chain. A common implementation places the LUTs in a single column, typically at positions X0Y0, X0Y1, X0Y2, and so forth. The Y-index follows a deliberate pattern (0, 1, 0, 1, 0, 1, 0, 1, 2, 3, …) so that consecutive LUTs occupy different physical slices. This alternation introduces additional routing delay between LUTs due to the interconnect wires, producing longer and more predictable delay values compared with placing all LUTs in the same slice. The spatial organization of these LUT-based delay chains strongly influences routing delay, congestion, and timing reproducibility. Three primary placement strategies are commonly adopted, each offering different trade-offs:

1. Row Pattern (RP)
   LUTs are placed horizontally along a single CLB row, as illustrated in Appendix Figure 1. This minimizes vertical routing delay but can increase congestion when many delay elements share the same row [14], [15], [16], [17], [18], [19], [20].
2. Column Pattern (CP)
   LUTs are stacked vertically across CLB columns, as shown in Appendix Figure 2. This arrangement produces uniform routing paths but may introduce larger inter-stage distances [10].
3. Row-Column Pattern (RCP)
   As depicted in Appendix Figure 3, this pattern alternates between horizontal and vertical placement, combining the uniformity of CP with the compactness of RP. It often provides more predictable timing behavior in multi-stage pipelines [21].

2.3.2 Selection Delay Elements

While matched delays ensure safe data capture, they require every instruction to adhere to the longest combinational delay within the stage, which limits overall throughput. Selection delays address this limitation by providing multiple matched-delay branches that can be chosen at runtime. This structure allows each stage to adapt its completion time to the actual operation latency. Each delay channel is tuned to match a particular operation type, enabling fast operations to complete earlier, while slower operations use longer delay paths [11], [22], [23]. The selection delay architecture consists of a multiplexer that chooses among several parallel delay channels based on control signals derived from the instruction type or decode stage. This dynamic timing adaptation allows the pipeline to avoid unnecessary stalls and achieve better-than-worst-case performance.

Despite their advantages, selection delays significantly complicate timing verification because each channel introduces different propagation characteristics. Static timing analysis must consider all possible delay paths and ensure that each satisfies the bundling constraint under its corresponding operational conditions. Furthermore, the multiplexer and its selection logic introduce additional delays and potential timing hazards that must be incorporated into the analysis. These challenges motivate the automated verification and placement methodologies proposed in later chapters, which systematically evaluate and optimize selection delays across all modes of operation.

2.4 Asynchronous Pipeline Architecture

Asynchronous BD circuits can be organized into several pipeline architectures depending on how control and data flow are structured. The three fundamental forms, linear pipelines, sequential rings, and iterative rings, represent increasing levels of structural complexity and serve as templates for constructing larger

asynchronous systems. All architectures use phase-decoupled Click elements as handshake controllers to coordinate data transfers and ensure proper initialization.

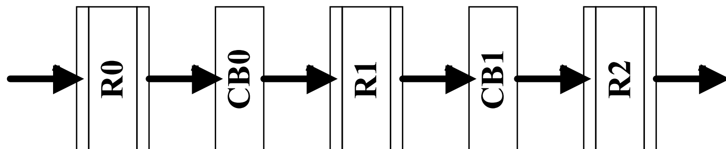


Figure 2.4. Linear pipeline structure.

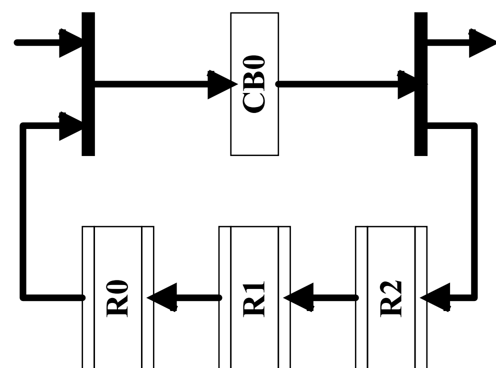


Figure 2.5. Sequential ring pipeline structure.

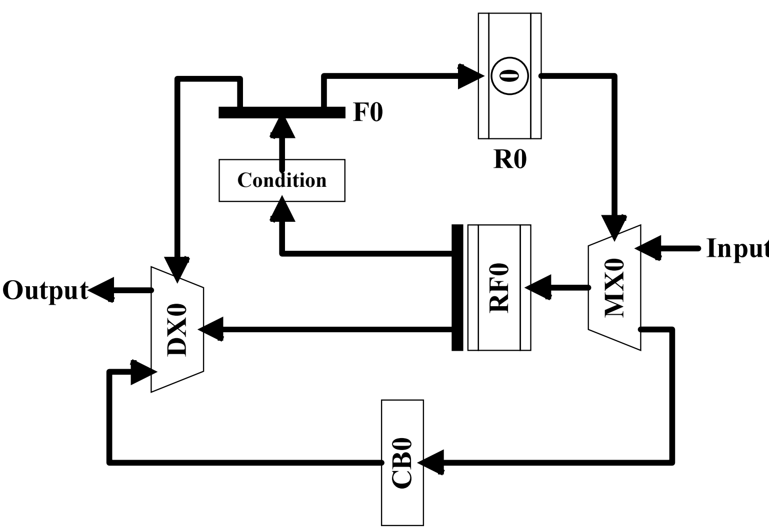


Figure 2.6. Iterative ring pipeline structure.

1. Linear pipeline

   The linear pipeline is the simplest form, consisting of a feed-forward chain in which data move sequentially from one stage to the next. Each stage contains a register controller, a block of combinational logic, and local request–acknowledge channels connecting it to its neighbors. Because the structure contains no feedback paths, timing analysis is relatively straightforward, only the setup and hold constraints between adjacent stages must be satisfied. Proper insertion of delay elements along the request paths ensures the bundling constraint and stable pipeline operation [11]. Figure 2.4.

illustrates the topology of a linear BD pipeline, showing the sequence of register–logic–register stages connected through local handshaking channels.

2. Sequential ring pipeline

   The sequential ring extends the linear pipeline by feeding the output of the last stage back into the first, forming a closed-loop data path suitable for cyclic or recurrent computations. Fork and join controllers are typically used to distribute and merge data within the loop. Although this structure improves resource utilization, the presence of a feedback path introduces architectural loops that must be handled carefully during timing analysis to avoid propagation errors [11]. Figure 2.5. shows the organization of a sequential ring, highlighting the circular flow of data and the feedback connection between the final and initial stages.

3. Iterative ring pipeline

   The iterative ring architecture introduces conditional execution paths within the cyclic structure. Multiplexer (MX) and demultiplexer (DX) controllers determine whether data continues circulating or exits the loop, enabling iterative and conditional behavior. This architecture combines feedback with branching control, making it the most complex among BD pipeline forms. It often contains passive controllers and shared logic blocks, requiring additional static timing considerations to manage unrelated or multi-path timing dependencies. Delay elements are tuned according to the worst-case slack across all timing paths, and local loops are selectively disabled to maintain consistent timing behavior [11]. Figure 2.6. illustrates the structure of an iterative asynchronous BD ring, emphasizing the conditional feedback managed by the multiplexer and demultiplexer elements.

2.5 Static Timing Analysis

Understanding timing analysis is essential for implementing correct asynchronous BD circuits on FPGA platforms. While synchronous circuits rely on a global clock and its associated timing constraints, asynchronous BD circuits must instead consider the relative timing between data and control paths. This section introduces the fundamental timing concepts that underpin both synchronous and asynchronous design methodologies.

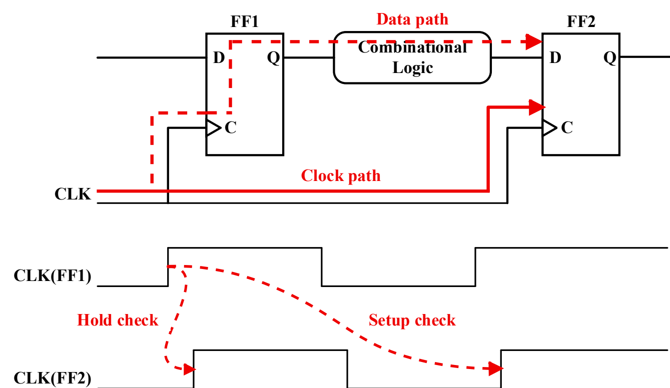


Figure 2.7. General setup and hold timing of synchronous circuits.

2.5.1 Setup and Hold Time Requirements

Setup and hold times define the intervals during which data must remain stable relative to a clock or control transition [24]. The setup time (T_{setup}) specifies how long data must be valid before the active clock edge so that the flip-flop can capture a stable value [25]. The hold time (T_{hold}) specifies how long data must remain stable after the clock edge to avoid corrupting the captured value [25]. Figure 2.7 illustrates these timing windows. For a synchronous flip-flop, the setup and hold constraints are expressed as:

$$T_{arrival_data} + T_{setup} < T_{clock_edge} \tag{2.1}$$

$$T_{clock_edge} + T_{hold} < T_{arrival_data} + T_{contamination} \tag{2.2}$$

where $T_{arrival_data}$ represents the latest time at which data arrives, and $T_{contamination}$ represents the earliest time at which data may change.

2.5.2 Slack Calculation and Timing Margins

Slack indicates the margin by which a timing constraint is satisfied. Positive slack means that the requirement is met, whereas negative slack indicates a timing violation. The hold slack must be positive to ensure data stability, and unlike setup constraints, hold violations cannot be corrected by adjusting the clock frequency [25].

For setup timing, slack is computed as:

$$Slack_{setup} = T_{required} - T_{arrival} = \left(T_{clock_{period}} - T_{setup}\right) - (T_{clk-Q} + T_{logic} + T_{routing}) \quad (2.3)$$

For hold constraints, the calculation differs:

$$Slack_{hold} = T_{arrival} - T_{required} = \left(T_{clk-Q} + T_{contamination}\right) - T_{hold} \quad (2.4)$$

where $T_{required}$ represents the latest time at which data must arrive to satisfy setup requirements, and $T_{arrival}$ represents the actual arrival time considering all path delays.

2.5.3 Critical Path Analysis

The critical path is the longest combinational delay between sequential elements. In synchronous systems, it determines the maximum operating frequency, while in asynchronous BD circuits, it defines the minimum delay-element size required to satisfy the bundling constraint [26]. Identifying the critical path involves examining all possible routes through the circuit and summing the delays along each path. The path with the largest cumulative delay becomes the critical path and sets the timing requirement that all other paths must meet [26]. Critical path analysis becomes more involved when a design includes conditional logic, multiple clock domains, or feedback loops. False paths, those that cannot be activated during normal operation, must be excluded to avoid unnecessarily restricting the design. Multi-cycle paths, where data legitimately takes more than one cycle to propagate, must also be handled carefully to prevent incorrect timing violations [27]. Static timing analysis tools automate these tasks by constructing a timing graph in which nodes correspond to sequential elements and edges represent combinational paths with associated delays. Graph traversal algorithms compute arrival times across the graph, highlighting paths whose slack is close to zero or negative, thereby identifying potential timing bottlenecks [28].

2.5.4 Timing Analysis in Two-Phase Asynchronous Bundled-Data Circuits

In a two-phase BD pipeline, data transfers are coordinated by request and acknowledge signals, where every transition, either rising or falling, represents a handshake event. Each stage contains a storage register, combinational logic, a handshake controller (such as a phase-decoupled Click element), and delay elements on the control path. The goal of timing analysis is to ensure that data signals remain valid throughout every handshake transition. An asynchronous BD circuit divides a computation into multiple pipeline stages, each consisting of a control path and a data path. The control path uses handshake controllers to regulate the movement of data tokens, while the data path includes combinational logic for computation and flip-flops for storing results. Request and acknowledge wires connect the controllers between stages, and delay elements align the timing of these signals with the completion of the logic operations [29].

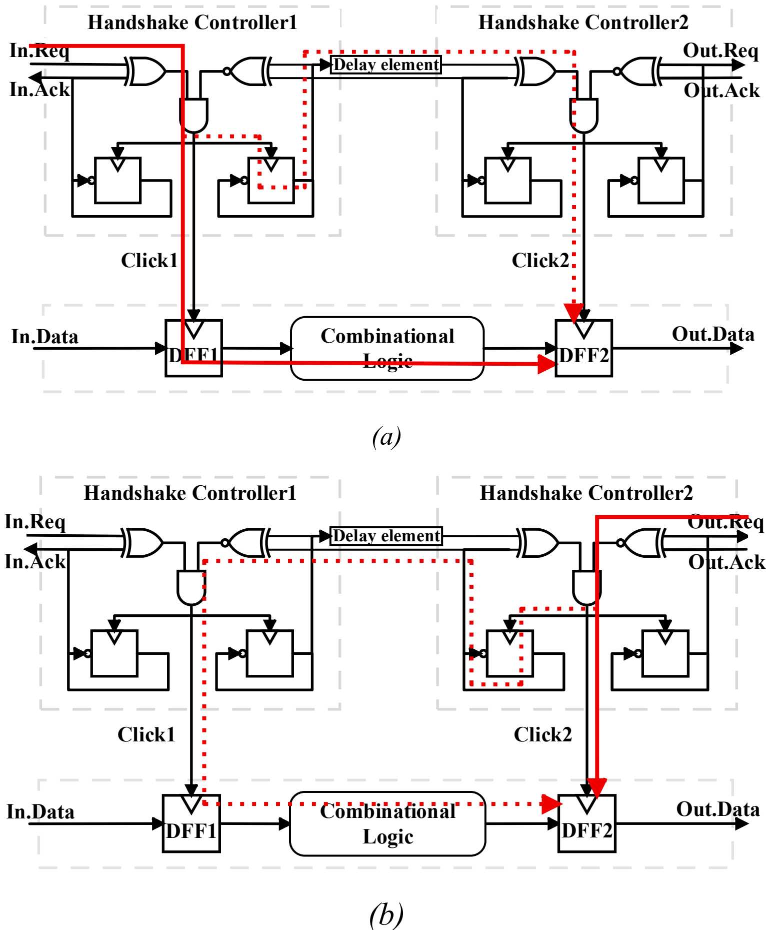


Figure 2.8. *Timing constraints of the phase-decoupled pipeline: (a) setup timing path, and (b) hold timing path.*

As illustrated in Figure 2.8, a phase-decoupled Click controller generates a local pulse (Click2) when the input request signal toggles. This pulse enables the downstream flip-flops (DFF2) to capture the result from the combinational block. Once the data capture is complete, the acknowledgment is asserted to inform the upstream stage. For the pipeline to operate correctly, the bundling constraint must be satisfied: data must be stable at the moment of capture and remain valid long enough afterwards. STA checks timing violations on these paths by examining setup and hold requirements [17].

Setup constraint: the minimum delay on the control path must be greater than the maximum delay of the data path, so that the capture event occurs only after data becomes valid.

Hold constraint: the data path must remain stable for a sufficient interval after capture to avoid metastability or premature switching.

These relationships are given by:

$$T_{DE} + T_{click2} + T_{skew2} > T_{CL,max} + T_{setup} + T_{skew1} \quad (2.5)$$

$$T_{hold} + T_{skew2} < T_{click1,min} + T_{CL,min} + T_{clk-Q,min} + T_{skew1} \quad (2.6)$$

where T_{DE} is the latency of a delay element, T_{click} is the delay of the combinational logic in the handshake controller, T_{skew} is the clock skew of a DFF, T_{CL} is the combinational logic delay, T_{setup} is the setup time of a DFF, T_{hold} is the hold time of a DFF, and $T_{clk\text{-}Q}$ is the maximum propagation time.

Because the contamination delay of a typical DFF is usually larger than its hold time, additional delay elements are not typically required on the acknowledgment path [30]. However, as circuits become more

complex, the logic required to generate the local fire pulse may introduce non-negligible delays that must be accounted for when verifying setup and hold constraints. To ensure stable operation across all stages, request-path delays must be tuned so that the control path consistently exceeds the data path delay by the required margin. For large asynchronous BD designs, an automated and reliable timing-analysis and delay-insertion methodology is therefore essential, an issue addressed by the frameworks introduced in the following chapters.

2.6 FPGAs

A Field-Programmable Gate Array (FPGA) is a reconfigurable integrated circuit that allows designers to implement custom hardware after fabrication by loading a configuration bitstream. Unlike fixed-function ASICs, FPGAs support rapid prototyping, hardware reuse, and broad design exploration, making them well suited for experimenting with asynchronous BD circuits. Internally, an FPGA consists of several hierarchical resources connected through a programmable routing network [31]. Figure 2.9 provides an overview of a modern FPGA architecture, while Table 2.3 summarizes its key components.

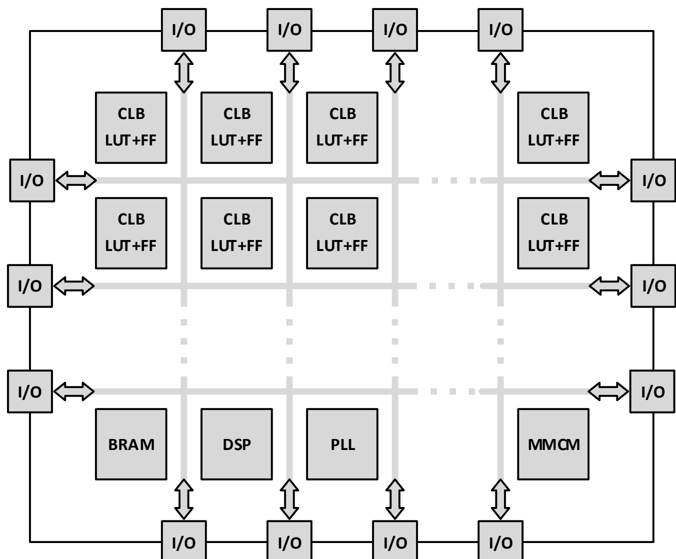


Figure 2.9. *Simplified architecture of a modern FPGA showing configurable logic blocks (CLBs), routing network, on-chip memories, DSP blocks, and I/O interfaces.*

Earlier FPGA development was supported by the Integrated Synthesis Environment (ISE), which was later replaced by Xilinx Vivado. Vivado offers improved timing analysis, more advanced routing algorithms, and better automation for modern families such as Zynq and UltraScale. In this dissertation, Xilinx FPGA platforms are used as the target hardware due to their widespread academic adoption, stable toolchain support, and compatibility with the proposed asynchronous BD design flow. Although FPGAs provide high flexibility for handshake-based architectures, their clock-centric nature introduces specific challenges for asynchronous design. Signal delays are determined by a discrete LUT and routing resources, limiting the granularity with which delay balancing can be achieved. Commercial tools focus primarily on clocked static timing analysis, which makes relative timing and minimum-delay constraints more difficult to express. Minor placement changes can significantly alter delay characteristics, affecting compliance with the bundling constraint. Furthermore, implementing matched and selection delays often requires additional LUTs, increasing resource usage. Despite these challenges, FPGAs remain an effective platform for validating asynchronous BD architectures thanks to their reconfigurability, rapid verification capability, and suitability for prototyping complex handshake pipelines prior to ASIC deployment [31].

Table 2.3. Key architectural components of an FPGA.

No.	Component	Description	Function
1	Configurable Logic Block (CLB)	Contains lookup tables, flip-flops, and fast carry chains	Implements combinational and sequential logic
2	Programmable Interconnect	Network of routing switches and wires connecting CLBs and other blocks	Provides flexible routing paths
3	Block RAM (BRAM) / Distributed RAM	On-chip memory arrays	Used for buffers, FIFOs, and lookup tables
4	DSP Blocks	Dedicated multiply–accumulate units	Accelerate arithmetic and signal-processing operations
5	Clocking Resources (PLL/MMCM)	Generate and distribute low-skew clock signals	Provide synchronization in synchronous regions
6	I/O Blocks	Configurable input/output cells with delay elements and serializers	Interface with external devices and support timing adjustments

2.7 FPGA Circuit Design Flow

The conventional FPGA design flow transforms an HDL description into a fully implemented hardware circuit through a series of steps, including synthesis, placement, routing, and verification [32], [33]. Figure 2.10 illustrates this standard sequence, which is typically managed by commercial electronic design automation (EDA) tools. Although the flow is highly optimized for clocked systems, it requires several adaptations to accommodate the local-timing behavior of asynchronous BD circuits.

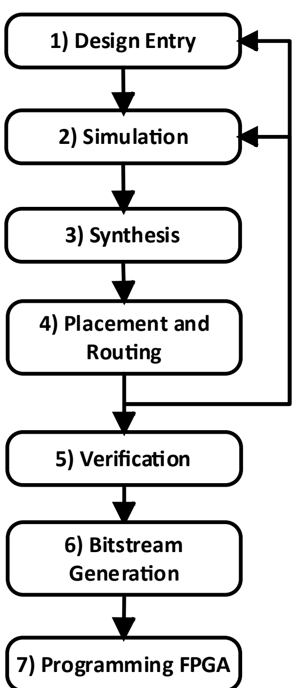


Figure 2.10. General FPGA design flow using EDA tools.

1. Design Entry
   The process begins by describing the circuit in VHDL or Verilog at the RTL level, defining both the functionality and the interface structure. A well-organized hierarchy helps preserve timing-critical modules during synthesis. Design constraints, such as clocking requirements, pin assignments, and floor planning directives, are specified in XDC or SDC format to guide the toolchain.

2. Simulation
   Functional simulation verifies that the HDL behaves as intended using testbenches. Behavioral simulation validates logic correctness, whereas post-synthesis or post-route simulation incorporates back-annotated delays to reflect physical timing characteristics before programming the FPGA.

3. Synthesis
   The validated HDL is translated into a technology-mapped netlist composed of LUTs, flip-flops, carry chains, and block RAMs. During synthesis, optimization heuristics restructure logic to meet timing and area objectives while honoring design attributes such as *"DONT_TOUCH"* or *"KEEP_HIERARCHY"* for timing-sensitive modules.

4. Implementation
   Implementation maps the logical design to physical FPGA resources. Placement assigns physical locations to logic elements, and routing establishes signal connections through the programmable interconnect. Placement quality significantly influences achievable performance and routing congestion.

5. Verification
   Post-implementation checks include design rule checks (DRC) and static timing analysis (STA). STA verifies that setup and hold constraints are met for all clock domains. If violations are detected, designers adjust constraints or floorplanning strategies and repeat implementation until timing closure is achieved.

6. Bitstream Generation
   The final bitstream encodes the contents of LUTs, routing resources, and I/O configurations. Loading this bitstream into the FPGA completes the implementation cycle.

Although this flow is efficient for synchronous circuits, it assumes the presence of global clocks and edge-triggered timing references. Asynchronous BD circuits do not rely on a global clock; instead, timing is defined through relative delays between data and control paths. As a result, additional steps, such as handshake modeling, virtual clock generation, and delay-element placement, must be incorporated to extend the conventional flow. The following sections describe these adaptations and how they support asynchronous BD design on FPGA platforms.

2.8 Asynchronous Circuit Design Flow

The asynchronous FPGA design flow retains the main stages of the conventional process but augments them with handshake-based timing management and delay-alignment procedures. Instead of relying on a global clock edge, each stage signals completion through request and acknowledge transitions. The asynchronous process can be summarized as follows:

1. Handshake-Based Modeling
   The design is partitioned into data paths, consisting of combinational logic and registers, and control paths composed of handshake controllers. Common controllers include Click, Phase-Decoupled Click, Source, Sink, Fork, Join, MUX, and DEMUX [11]. Each controller dictates when data is captured and when the next stage may proceed, forming the foundation of the BD timing discipline [34].

2. Constraint Formulation

In the absence of a global clock, relative-timing constraints replace traditional setup and hold checks. Commands such as "create_generated_clock", "set_false_path", and "set_clock_groups -asynchronous" are used within STA tools to encode handshake dependencies and isolate unrelated paths [17-19], [34].

3. Synthesis and Implementation
   HDL modules are synthesized into FPGA primitives. Handshake controllers and delay elements are preserved using constraints that prevent unwanted optimization. During placement, spatial locality between control and data path logic is emphasized to reduce routing imbalance and improve timing consistency [17-19] [34].

4. Delay Matching and Verification
   Programmable delay elements are inserted along control paths so that request transitions occur only after data stabilization, satisfying the bundling constraint. STA verifies timing correctness by ensuring that the relative delays between data and control paths meet setup and hold requirements [10, 17-19], [22], [34].

5. Functional Simulation
   Post-route simulation with back-annotated delays validates proper handshake sequencing and checks for deadlock-free operation and correct token propagation [10, 17-19], [22], [34].

Therefore, designing asynchronous BD circuits on FPGAs extends the conventional FPGA flow by replacing global-clock coordination with localized request–acknowledge handshaking. Although the process still follows the familiar steps of synthesis, placement, routing, and static timing analysis, asynchronous implementations introduce additional requirements such as handshake modeling, virtual-clock generation, loop handling, and delay-element placement. Because BD timing depends on relative delays rather than a global reference clock, these relationships must be explicitly captured to ensure functional correctness after physical implementation. Within this context, two major timing-closure strategies have emerged for asynchronous BD circuits on FPGA platforms: iterative synthesis flows and single-pass synthesis flows. These approaches differ fundamentally in how delay alignment is achieved, how timing constraints are constructed, and how designers interact with the toolchain.

2.8.1 Iterative Synthesis Design Flow

The iterative synthesis design flow is one of the earliest and most commonly used approaches for implementing asynchronous BD circuits on FPGA platforms. Its defining characteristic is the reliance on repeated cycles of synthesis, place-and-route, STA, and manual delay adjustment until all bundling constraints are satisfied, as illustrated in Figure 2.11. Because asynchronous circuits do not share a global clock reference, timing correctness depends entirely on the relative alignment between data path and control-path delays. Consequently, timing closure rarely occurs in a single implementation run, making iterative refinement the default strategy in early asynchronous FPGA design methodologies.

In a typical iterative process, the design is synthesized and routed using standard FPGA tools. STA is then used to detect mismatches in which the request (Req) transition arrives before the corresponding data path signals have stabilized. When such violations occur, the designer manually adjusts the LUT-based delay elements, modifies placement constraints, or inserts additional buffers to increase the control-path delay. The updated design is then reimplemented, rerouted, and reanalyzed. This cycle repeats until every stage in the pipeline satisfies both setup and hold requirements. Although conceptually straightforward, the approach can be time-consuming and unpredictable because routing variations between iterations often alter delay values, undoing earlier adjustments and necessitating further refinement. Two representative iterative methodologies, Adaptive Delay Matching (ADM) and the Propagation Timing Constraint (PTC) framework, illustrate how iterative STA is applied in practice.

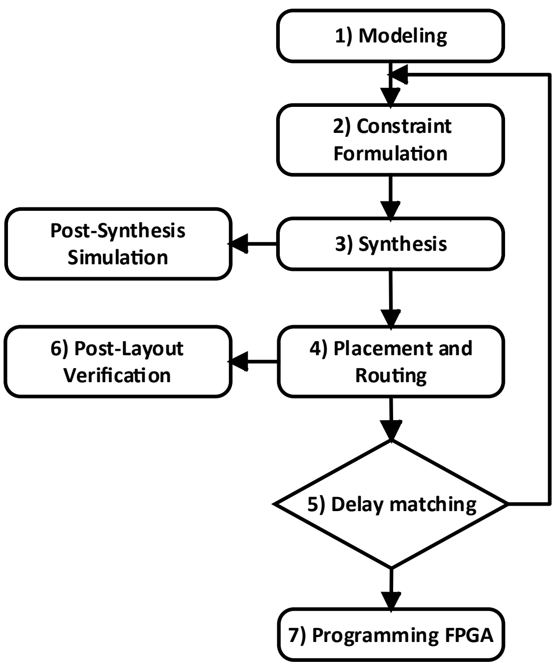


Figure 2.11. The iterative synthesis flow requires multiple implementation and verification cycles before timing closure is achieved.

2.8.1.1 Adaptive Delay Matching (ADM)

ADM was originally proposed for Click-based asynchronous BD circuits implemented in ASIC flows [18]. The key idea is to treat the *fire* signals generated by Click elements as local clocks and to perform delay matching by specifying minimum-delay constraints between these clocks. Delay lines are then inserted between stages so that the control-path latency satisfies the setup requirement relative to the longest data path delay. In the original formulation, a pipeline with *n* stages introduces *n* delay variables. All stages initially share the same delay value, typically a small non-zero number (e.g., 0.1 ns) used as a seed for synthesis. A representative fragment of the synthesis script from the original work can be written as:

```
1. # set variables
2. set var(1) 0.1
3. set var(2) 0.2
4. set_min_delay $var(1)../..
5. set_min_delay $var(2)../..
6. create_clock -name CK1 -period 30 -waveform {0.2} [get_pins fire_1]
7. create_generated_clock -name CK2 -source [get_clocks fire1] – edges {1 3 5} -edge_shift {$var(1) $var(1) $var(1)}
[get_pins fire_2]
8. create_generated_clock -name CK3 -source [get_clocks fire1] – edges {1 3 5} -edge_shift {$var(2) $var(2) $var(2)}
[get_pins fire_3]
```

Using these constraints, the synthesis tool (e.g., Design Compiler) interprets *fire* signals as phase-related local clocks and optimizes the combinational logic accordingly. Timing paths between consecutive fire signals are analyzed so that the delay-matching condition is enforced under worst-case PVT corners. After placement and routing, parasitic extraction is performed and STA is run in a dedicated tools. The script iterates over all reported timing paths using commands such as "for_in_collection" and "report_timing" to obtain slack values for each stage. If a stage exhibits a measured path delay of t_{path}, the corresponding delay variable is updated to:

$$t_{\text{new}} = t_{\text{path}} + t_h \tag{3.1}$$

where t_h is a guard margin (approximately 10% of the path delay in the original work).

The updated delay values are then written back into the synthesis script via new "set_min_delay" constraints, and the circuit is resynthesized. This synthesis–STA loop continues until all stages satisfy the timing requirements with the desired margin. In ASIC environments, ADM provides a fully automated delay-matching mechanism because tools natively support "set_min_delay" on arbitrary nets. However, commercial FPGA tools generally do not allow minimum-delay constraints on internal control signals, and the physical delay of LUT-based buffers depends heavily on routing. As a result, when ADM is applied to FPGAs, designers must manually emulate minimum delays by inserting LUT chains or modifying placement constraints instead of relying on "set_min_delay". This breaks the automation of the original method and often turns ADM into a manual, iterative tuning process, limiting its practicality for large asynchronous BD pipelines.

2.8.1.2 Propagation Timing Constraint (PTC)

The PTC method was introduced to overcome the limitations of minimum-delay-based approaches [17], such as ADM, by offering a systematic and FPGA-compatible timing model for Click-based asynchronous BD circuits. Instead of enforcing explicit minimum delays, PTC models each handshake completion as a virtual local clock and propagates this timing event across pipeline stages using "create_clock" and "create_generated_clock". This approach allows the STA engine to reason about control-path timing without relying on unsupported minimum-delay constraints, enabling accurate verification of both in-order and out-of-order handshake behavior. In PTC, the fire signal of the first stage is defined as the root clock. A virtual clock is created for the initial handshake event:

```
1. # Root virtual clock for the first handshake event
2. create_clock -name G0 -period 3 [get_pins RF_0/click_inferred_i_1/O]
```

This root clock is propagated stage by stage. Each subsequent fire pulse is modeled as a generated clock whose edges are derived from the preceding stage. An example XDC fragment is:

```
1. # Propagate request/acknowledge events across stages
2. create_generated_clock -name G1 -divide_by 1 -source [get_pins RF_0/phase_c_reg/C] [get_pins
RF_0/phase_c_reg/Q] -master G0 -add
3. create_generated_clock -name G2 -divide_by 1 -source [get_pins J_0/phase_reg/C] [get_pins J_0/phase_reg/Q] -
master G1 -add
```

The -add option is essential: it accumulates propagated handshake events rather than overwriting the existing clock definition, allowing the STA engine to maintain temporal causality across multiple Click controllers. To prevent analysis of non-functional combinational loops within Click elements, PTC disables these internal arcs:

```
1. set_disable_timing -from I0 -to O [get_cells S1/phase_a_i_1]
2. set_disable_timing -from I0 -to O [get_cells S1/phase_b_i_1]
```

PTC also supports multi-cycle acknowledgment behavior, which is important in selective pipelines, MUX/DEMUX structures, and other cases where handshakes may complete out of order. According to the referenced rule, if stage m acknowledges a request issued by stage n, the acknowledgment may legally occur up to $(m - n)$ cycles later. Modeling this behavior prevents the STA engine from reporting false timing violations for handshakes that legitimately propagate through multiple intermediate stages before producing an acknowledgment. For conditional paths, such as those passing through Fork/MUX or Join/DEMUX components, PTC combines generated clocks with asynchronous clock groups to isolate unrelated timing domains and avoid cross-domain interference. An example constraint specification is shown below:

```
1. # Setup: acknowledgment may occur 2 cycles late
2. set_multicycle_path 2 -setup -from [get_clocks G3] -to  [get_clocks G1]
3. set_clock_groups -asynchronous -group {G0}  -group {G1 G2 G3}
```

Once all propagated clocks and timing exceptions have been defined, STA is executed. During this phase, the timing engine evaluates each handshake channel by examining the source clock delay, the control-path delay, the data-path delay, and the destination clock delay:

```
1. report_timing_summary
2. report_timing -from [get_clocks G0] -to [get_clocks G1]
3. report_timing -from [get_clocks G1] -to [get_clocks G2]
4. report_timing -from [get_clocks G2] -to [get_clocks G3]
```

Based on the STA results, the designer adjusts the LUT-based delay elements near each Click controller to correct any remaining mismatches. Although PTC provides much higher accuracy and better FPGA compatibility than ADM, it still requires multiple synthesis and routing iterations because delay balancing remains sensitive to routing variation. As a result, PTC improves the iterative process but does not eliminate the need for repeated STA-guided refinement cycles. Despite these advances, iterative flows share several fundamental drawbacks. They demand considerable design effort due to repeated compilation steps, their convergence is often unpredictable because routing randomness can easily disrupt carefully tuned delay values, and their scalability is limited for designs containing many handshake stages. These limitations collectively motivated the development of automated and deterministic single-pass methodologies.

2.8.1.3 Other Technique

Beyond PTC, several iterative-synthesis approaches [23] attempt to reduce manual retuning by relying on fixed propagation templates or static handshake-event modeling. Template-based schemes assign predetermined delay margins to Click controllers and matched-delay components, ensuring correctness under worst-case timing conditions. However, these templates cannot accurately capture delay divergence introduced by conditional execution paths. This divergence, commonly referred to as selection delay, occurs when a request may traverse alternative computation paths, such as ALU, MUL, or VALU units, or multiple delay channels within a Delay Selection Unit (DSU). Because these paths differ in logic depth and routing complexity, fixed-delay templates must conservatively match the slowest path, leading to unnecessary performance loss. Handshake-event modeling provides a more expressive alternative by representing each firing event as a generated clock that propagates through the asynchronous pipeline. Timing causality is encoded directly through STA constraints. For example:

```
1. # Base firing event clock
2. create_clock -name F0 -period 4.0 [get_pins S0/fire]
3. # Propagated firing event at next stage
4. create_generated_clock -name F1 -source [get_pins S0/fire] -divide_by 1 [get_pins S1/fire]
5. # Optional propagated firing event at another stage
6. create_generated_clock -name F2 -source [get_pins S1/fire] -divide_by 1 [get_pins S2/fire]
7. # Suppress inactive conditional branch through MUX/DEMUX
8. set_false_path -from [get_pins MUX/out1] -to [get_pins JOIN/in0]
9. # Isolate unrelated handshake-event domains
10. set_clock_groups -asynchronous -group {F0} -group {F1 F2}
11. set_multicycle_path 2 -setup -from [get_clocks F2] -to [get_clocks F1]
12. # Timing verification between handshake-event clocks
13. report_timing -from [get_clocks F0] -to [get_clocks F1] -max_paths 10 -nworst 1
```

While this class of techniques captures handshake causality more precisely than template-based methods, it still faces difficulties when modeling selection delays introduced by DSUs. Because a request may propagate through either a short or a long delay channel, depending on the operation, STA must evaluate each channel separately. If both channels are checked against the fast timing constraint, the slow channel inevitably produces false setup violations. To avoid this, the slow channel must be constrained explicitly:

```
1. set_max_delay 4.4 -from [get_pins DSU/in_req] -to  [get_pins DSU/out_req] -datapath_only
```

These constraints emulate the behavior of the Click controller, which only issues a handshake completion when the *selected* delay channel finishes its operation. They also ensure that STA reflects the dynamic timing behavior of the DSU more accurately. Despite being more expressive than fixed-delay templates or simple

generated-clock propagation, these techniques still require evaluating each delay channel individually. Consequently, pipelines containing many conditional handshake paths often demand multiple synthesis–implementation cycles, causing design effort to scale poorly for large or heterogeneous asynchronous processors.

2.8.2 Single-Pass Synthesis Design Flow

The single-pass synthesis design flow addresses the fundamental limitations of iterative timing-closure methodologies by eliminating the need for repeated synthesis, routing, and manual delay refinement, as illustrated in Figure 2.12. Rather than adjusting control-path delays through multiple rounds of post-implementation STA, the single-pass approach derives all necessary timing constraints, loop-breaking rules, and delay-placement requirements before synthesis and implementation begin. This transforms timing closure for asynchronous BD circuits into a deterministic process with far greater reproducibility and significantly reduced design effort. A major advantage of the single-pass paradigm is that delay elements are introduced algorithmically during constraint generation, rather than through trial-and-error tuning. Delay lines or minimum-latency targets are determined analytically from expected data path behavior and are inserted automatically during synthesis. STA is performed only once, not to refine delays, but to verify that the predefined constraints correctly enforce the bundling relationship. Consequently, single-pass methods achieve predictable convergence and maintain stable timing behavior across independent builds.

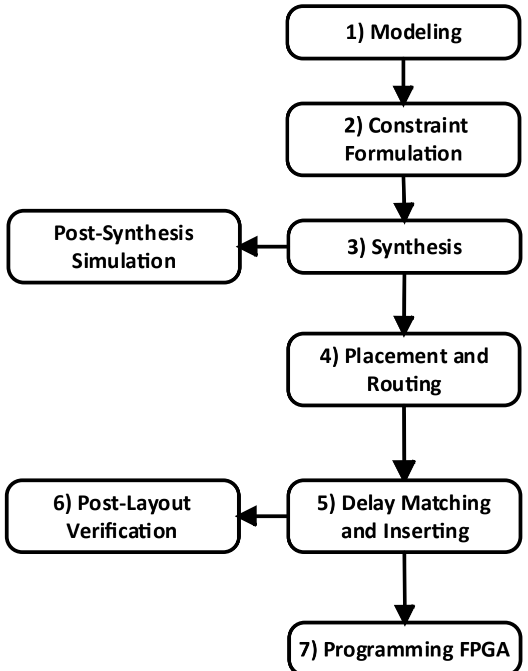


Figure 2.12. *Single-pass synthesis flow integrates constraint generation, timing verification, and delay insertion within a single automated process.*

2.8.2.1 Backward Delay Propagation Constraint (BDPC)

BDPC performs a stage-by-stage timing analysis to determine the required delay elements without explicitly modeling phase relationships between propagated clocks. The method defines a master clock at the final pipeline stage using the acknowledgment signal and then propagates timing backward through generated clocks. To facilitate this propagation, BDPC employs "set_clock_tree_exception" to decouple clock trees, "set_disable_timing" to remove misleading timing arcs, and "set_case_analysis" to handle non-unate logic. A simplified STA example illustrating this backward timing derivation is shown below:

```
1. # Virtual Ack clock at Stage 2 (final stage)
2. create_clock -name ACK2 -period 5.0 [get_pins S2/ack]
3.
4. # Decouple the virtual Ack clock tree from the physical clock network
5. set_clock_tree_exception -ignore [get_clocks ACK2]
6.
7. # Disable misleading timing arcs in the Stage-2 controller
8. set_disable_timing -from [get_pins S2/req_reg/Q] -to [get_pins S2/ack_comb/D]
9.
10. # Resolve non-unate control logic by forcing the select signal
11. set_case_analysis 0 [get_pins S2/non_unate_sel]
12.
13. # Backward-propagated virtual Req event for Stage 1
14. create_generated_clock -name REQ1_BWD -source [get_pins S2/ack] -divide_by 1 [get_pins S1/req]
15.
16. # Required control-path delay to satisfy the backward constraint
17. set_min_delay 1.0 -from [get_pins S1/req] -to  [get_pins S2/ack]
```

After placement and routing, slack values reported by STA are used to identify timing violations. Buffers are then manually inserted to resolve these issues, and the process is repeated until all slack values become positive and timing closure is achieved. Although BDPC reduces the need for detailed RTL-level timing modeling, it introduces a substantial post-layout workload because it depends on manual ECO operations. Furthermore, BDPC is incompatible with FPGA toolchains, as commands such as "set_clock_tree_exception" are not supported. Finally, BDPC cannot accommodate selection delays, because acknowledgment-based clock propagation does not capture the dynamic control behavior (e.g., multiplexers) present in selection-delay structures along request paths.

2.8.3 Comparison of Asynchronous Design Flows

Table 2.4 illustrates the conceptual differences between iterative and single-pass asynchronous flows. In an iterative flow, the design cycles repeatedly through synthesis and STA until timing closure is achieved. In contrast, the single-pass flow integrates constraint generation, timing verification, and delay placement into one continuous automated process. This progression, from conventional synchronous design, to asynchronous design requiring manual iterative refinement, and finally to automated single-pass asynchronous synthesis, represents a significant conceptual and practical advance. It enables asynchronous systems to approach the predictability and reproducibility of synchronous designs while retaining their inherent strengths in modularity and energy efficiency. Although several methods have been developed to support STA of asynchronous bundled-data circuits, each presents fundamental limitations that hinder practical deployment on FPGA platforms. ADM models handshake events as locally generated clocks with fixed phase offsets and uses "set_min_delay" to enforce timing margins. However, FPGA toolchains generally do not support net-specific minimum-delay constraints, making ADM infeasible in practice. Furthermore, ADM's reliance on fixed phase relationships prevents it from modeling dynamic or data-dependent timing behavior such as adaptive delays.

PTC improves FPGA compatibility by propagating handshake events as generated clocks and using "set_multicycle_path" to define inter-stage timing dependencies. While effective for fixed-structure pipelines, PTC still requires iterative adjustment of delay elements, and its assumption of uniform phase propagation limits its ability to model selection delays or data-dependent control behavior. BDPC increases modeling fidelity by propagating timing requirements backward from the final stage, pruning invalid paths using "set_disable_timing" and "set_case_analysis". Although more accurate than ADM or PTC, BDPC relies heavily on manual ECO operations to insert or adjust delay elements after routing. This post-layout workload is labor-intensive and error-prone. In addition, BDPC provides limited support for cyclic or feedback-driven topologies such as rings and token-based loops, which are widely used in asynchronous systems. Table 2.5 summarizes the key limitations of these existing timing-analysis approaches. Collectively, these restrictions prevent any prior

method from achieving automated, single-pass, and loop-safe STA for asynchronous BD circuits on FPGA platforms. The Generated Clock Propagation (GCP) method was developed to address these gaps through a unified and automated constraint-generation framework.

Table 2.4. *Comparison of synchronous, iterative asynchronous, and single-pass asynchronous design flows.*

Aspect	Conventional FPGA (Synchronous)	Iterative Asynchronous Flow	Single-Pass Asynchronous Flow
Timing Reference	Global clock domains	Local handshake	Local handshake
Constraint Formulation	"create_clock", "set_input_delay"	"create_clock", "create_generated_clock"	"create_clock", "create_generated_clock"
Timing Closure Method	STA with one pass per clock	STA + manual delay adjustment (RTL-level)	STA + manual delay adjustment (post-layout)
Convergence Behavior	Deterministic	Multi-round, designer-dependent	One-round, deterministic
Automation Level	Full (built into tools)	Low – manual constraint editing	Low – manual post-layout
Tool Compatibility	Native support	Partial – unsupported min-delay constraints	Partial – uses only supported commands
Reproducibility	High	Low – RTL-dependent	Low – layout-dependent
Scalability	High for clocked systems	Partial – scales to multi-stage pipelines	Partial – scales to multi-stage pipelines

Table 2.5. summarizes representative STA methods for asynchronous BD circuits, focusing on their analytical principles, tool compatibility, and inherent limitations related to STA.

Method	Key Concept	Main STA Constraints	Automation	FPGA Compatibility	Supported Architecture	Limitations
ADM	Models each request as a local clock with a fixed phase offset; enforces minimum delay between control and datapath signals.	"create_clock", "create_generated_clock -edges, set_clock_groups", "set_min_delay"	Manual constraint definition	Low – "set_min_delay" unsupported for internal control paths	Linear pipelines	Cannot model variable or data-dependent timing; no support for non-unate logic or cyclic paths; infeasible on FPGA due to missing min-delay support
PTC	Propagates handshake timing forward using generated clocks; uses multicycle constraints to model inter-stage timing.	"create_clock", "create_generated_clock", "set_clock_groups", "set_multicycle_path", "set_false_path"	Manual constraint definition	High – all commands supported	Linear and conditional pipelines	Assumes uniform phase propagation; limited accuracy for adaptive or selection paths; requires multiple STA iterations
BDPC	Performs backward STA from the final stage; disables non-functional feedback arcs and handles non-unate logic.	"create_generated_clock", "set_disable_timing", "set_case_analysis", "set_clock_tree_exception"	Manual constraint definition	Low – set_clock_tree_exception unsupported on FPGA	Linear and conditional pipelines	Sensitive to cyclic dependencies; partial support for multi-domain systems; relies on manual ECO operations

Other Techniques [23]	Uses constraint recipes for adaptive delay; performs per-channel STA without fixed phase offsets.	"create_generated_clock", "set_multicycle_path", "set_max_delay", "set_false_path"	Manual constraint definition	High – uses only supported commands	Linear, conditional, and selection-delay pipelines	Requires separate STA per delay channel; limited detection of cyclic or multi-path dependencies; assumes static timing for inactive paths

2.9 Delay Placement Approaches

Accurate delay placement is essential to ensure that asynchronous BD circuits operate reliably across process, voltage, and temperature (PVT) variations [28]. To address this requirement, several design flows have been proposed that insert and tune delay elements based on static timing analysis feedback. These flows differ in methodology, level of automation, and their compatibility with FPGA implementation tools. Broadly, existing delay-placement approaches fall into two categories:

1. Iterative synthesis approaches, which refine delay values through repeated cycles of synthesis, place-and-route, and STA until timing closure is reached.
2. Engineering Change Order (ECO)–based approaches, which perform post-layout delay tuning by manually inserting buffers or LUTs after routing.

Although both categories have contributed to advancing asynchronous design automation, they remain limited in scalability, reproducibility, and their ability to support more advanced architectures such as frequency-adaptive or combined asynchronous pipelines.

2.9.1 Iterative Synthesis Approaches

The earliest FPGA-based implementations of asynchronous circuits emerged from the iterative synthesis paradigm. Designers followed a repetitive loop beginning with RTL modeling, then proceeding through synthesis, placement, and routing, static timing analysis, and manual delay adjustment before starting the cycle again [18]. Each iteration aimed to refine the control-path delay until all handshake channels satisfied their setup and hold requirements. In these flows, delay elements were inserted and adjusted using timing intuition or rough initial estimates. After each implementation run, the timing report exposed mismatches between data and control paths, prompting further delay tuning or buffer insertion. Although this approach allowed fine-grained control, convergence was slow and unpredictable; each iteration could unintentionally disturb previously balanced paths. As a result, successful timing closure often depended heavily on designer experience and detailed knowledge of asynchronous behavior.

A notable technique developed under this framework was ADM [18]. ADM analyzed each handshake controller independently, gradually adjusting the delay element until request and acknowledgment transitions consistently bracketed valid data. Designers relied on repeated simulations and STA reports to determine whether the inserted delays provided adequate safety margins. While ADM worked reasonably well for small designs, its strong dependence on manual tuning limited its ability to scale. To improve timing consistency across pipeline stages, the PTC [17] method extended the iterative concept by propagating timing dependencies forward from the first stage through the pipeline. Each stage inherited the timing characteristics of its predecessor, improving inter-stage coherence. However, PTC still required multiple synthesis runs, especially when dealing with conditional handshake paths or bypass logic. As asynchronous designs evolved, frequency-adaptive and selection-delay pipelines introduced parallel delay channels that adapt to operation latency. These architectures increased the verification burden significantly, as each channel required separate measurement and tuning. Although placing delay elements near relevant computational logic helped balance routing, the overall process grew increasingly complex and labor-intensive.

Overall, iterative synthesis approaches provided foundational insight into the timing behavior of asynchronous pipelines and enabled early FPGA prototypes. However, their reliance on repeated compilation, manual tuning, and designer expertise makes them impractical for large or complex circuits. These limitations motivated the development of automated, deterministic design flows capable of single-pass timing closure.

2.9.2 Engineering-Change-Order (ECO)–Based Approaches

As asynchronous circuits increased in scale and architectural complexity, designers began searching for alternatives to the long, repetitive cycles of traditional iterative synthesis. This need led to the adoption of engineering-change-order (ECO)–based delay-placement techniques, which operate directly on the post-layout netlist rather than re-synthesizing the entire design. Instead of rebuilding the circuit, delay adjustments were

applied by inserting or removing small buffers at carefully selected points in the routed netlist, allowing designers to fine-tune the timing balance between request–acknowledge paths.

Among the ECO-driven methods, the BDPC [35] approach became particularly influential. BDPC defines a timing reference at the final acknowledgment stage and propagates this requirement backward through the pipeline. At each stage, designers inspect whether data arrive too early or too late relative to the backward-propagated acknowledgment. When an imbalance is detected, LUT-based buffers are manually inserted near the affected handshake controller. This backward traversal preserves causal ordering, ensuring that each request transition aligns safely with its predecessor's acknowledgment. ECO-based techniques provide two notable benefits. First, they eliminate repeated synthesis and routing cycles, since all modifications occur after placement and can be verified immediately using STA. Second, they offer fine-grained physical control: designers can choose exact LUT locations and routing segments to achieve the desired delay. These qualities make ECO approaches effective for quick prototyping or small circuits needing targeted tuning.

However, the reliance on manual inspection remains a major limitation. Identifying optimal insertion points requires detailed knowledge of the routed design and careful analysis of each timing path. As circuits incorporate more handshake branches, passive controllers, or selection-delay structures, this manual process becomes increasingly time-consuming and prone to errors. Moreover, each ECO modification introduces risk, small routing differences between runs or across engineers can lead to inconsistent or non-reproducible timing behavior.

2.9.3 Limitation of Existing Delay Placement Approaches

Although iterative and ECO-based approaches have supported asynchronous circuit design on FPGAs, both remain constrained by manual effort and limited scalability. Iterative flows require repeated synthesis and timing verification, where each implementation round can disturb previously balanced paths. This dependence on human adjustment makes convergence slow and unpredictable, especially for pipelines with numerous delay elements.

ECO-based methods reduce the number of iterations but introduce manual editing of the routed netlist. While this allows for precise placement, it demands a detailed inspection of routing paths and risks inconsistent results across different engineers or tool versions. As designs grow to include multiple handshake branches or selection-delay channels, manual insertion becomes cumbersome and prone to errors.

Neither approach integrates seamlessly with commercial FPGA tools nor supports adaptive timing behavior. Their reliance on trial and error prevents the production of reproducible results and hinders the implementation of large-scale solutions. These limitations underscore the need for an automated, deterministic workflow that integrates physical awareness with systematic delay management.

Table 2.6. Comparison of Limitations in Existing Delay Placement Approaches.

Aspect	Iterative Synthesis Approaches	ECO-Based Approaches
Timing Closure Process	Requires repeated synthesis → STA → delay-adjustment loops	Performed post-layout through manual buffer insertion
Automation Level	Low – relies on designer-driven tuning	Low – requires manual inspection and netlist editing
Convergence Behavior	Unpredictable – routing changes can invalidate previous results	Manual – depends on visual confirmation after each ECO
Tool Compatibility	Partial FPGA support; some constraints unsupported	Compatible but lacks automation mechanisms
Scalability	Limited for large pipelines with many delay elements	Difficult for multi-branch handshake networks
Accuracy and Reproducibility	Sensitive to iteration variation and routing instability	Non-reproducible across users or tool versions
Support for Adaptive / Selective Delays	Requires per-channel tuning	No systematic support for conditional paths
Overall Limitation	Time-consuming and non-deterministic	Labor-intensive and error-prone
Representative Methods	ADM, PTC, GCP	BDPC

Chapter 3
Generated Clock Propagation Constraint

STA is essential for verifying the timing relationships between data and control signals in asynchronous BD circuits. Unlike synchronous systems that rely on a global clock, BD circuits use request and acknowledge transitions to coordinate timing behavior. This event-driven nature often causes conventional FPGA timing tools to misinterpret control-path dependencies, propagate timing through unintended loops, or incorrectly analyze conditional and selection-based paths, resulting in inaccurate or incomplete timing reports.

The Generated Clock Propagation [36, 37] (GCP) methodology overcomes these limitations by interpreting handshake events as virtual clocks and propagating them along the actual request–acknowledge dependencies. GCP reconstructs causal timing relationships using only native STA commands, isolates both local and architectural loops, and prunes inactive branches in conditional or multi-channel structures. In doing so, it enables FPGA STA engines to evaluate asynchronous circuits deterministically while preserving strict adherence to the bundling constraint.

Section 3.1 introduces the motivation behind the GCP method and outlines the challenges that motivate a new timing-analysis strategy for asynchronous BD circuits. Section 3.2 presents the proposed static timing-analysis constraint, describing how GCP captures handshake behavior using generated clocks, isolates unrelated timing paths, and reconstructs valid bundled-data timing relationships. Section 3.3 provides detailed examples of timing-path extraction using GCP, covering three representative cases: an iterative Fibonacci pipeline with an architectural loop, a conditional handshake circuit containing passive controllers, and a linear pipeline employing selection-delay channels. Section 3.4 demonstrates the application of GCP to practical asynchronous designs, including a linear FIR filter pipeline, a sequential-ring Fibonacci generator, and an iterative-ring AES encryption circuit. Finally, Section 3.5 presents a comparative evaluation between GCP and existing STA approaches, highlighting improvements in loop handling, compatibility with FPGA tools, and the determinism of timing analysis.

3.1 Motivation

STA for asynchronous two-phase BD circuits requires a fundamentally different treatment from synchronous designs. In the absence of a global reference clock, BD circuits rely entirely on local request and acknowledge transitions to coordinate data transfers. These transitions encode the causal ordering between stages, yet commercial FPGA timing tools, built around explicit clock domains, cannot interpret handshake-driven timing directly. As a result, when asynchronous circuits are mapped onto FPGA fabrics, STA often misinterprets feedback paths, merges unrelated logic, or propagates timing through cycles without termination, ultimately producing incomplete or incorrect timing reports. These issues are further amplified by architectural constraints inherent to FPGAs. LUT-based delay elements exhibit coarse, placement-dependent delay variations, and routing resources introduce non-uniform propagation latencies. Key constraints such as "set_min_delay" are not supported on arbitrary control signals, making classical delay-matching techniques like ADM [18] impractical. Existing asynchronous STA approaches also face structural limitations: PTC [17] assumes uniform phase propagation and struggles with complex conditional paths, while BDPC [35] relies heavily on manual disable-timing rules and cannot fully resolve cyclic dependencies. Consequently, none of these techniques can achieve timing closure in a single synthesis pass, and all require significant manual expertise.

GCP is motivated by the need for an automated, deterministic, and FPGA-compatible STA methodology capable of handling the complete range of asynchronous architectures and timing behaviors. Specifically, GCP addresses the following STA-critical scenarios:

- Iterative rings, where feedback paths repeatedly circulate data for computations such as Fibonacci or iterative FIR pipelines, often causing STA to report unbounded register-to-register paths.
- Sequential rings, where circulating control tokens maintain continuous activity, making it difficult for conventional STA tools to determine an unambiguous timing origin.
- Passive controllers, including forks, joins, multiplexers, and demultiplexers, that introduce conditional and non-unate paths, frequently resulting in false cycles or re-convergent fanout issues during timing analysis.
- Selective-delay pipelines, where multiple data-dependent delay channels exist, causing STA to misinterpret inactive channels or incorrectly flag multi-cycle violations.
- General STA hazards, such as local loops, architectural loops, false combinational feedback, and unate ambiguity, each capable of triggering infinite timing propagation or invalid timing checks if not explicitly resolved.

By propagating virtual clocks along real request–acknowledge dependencies, GCP reconstructs the causal timing behavior of asynchronous circuits in a form that FPGA STA engines can analyze reliably. The method isolates each asynchronous domain, suppresses redundant or cyclic timing paths, and uses generated clocks to enforce correct handshake sequencing. Through this transformation, GCP bridges the gap between asynchronous handshake semantics and synchronous timing analysis, enabling loop-safe, deterministic timing verification even in large and structurally complex self-timed systems. This foundation enables single-pass timing closure without manual tuning and provides the analytical basis for the MDPC delay-placement method introduced in the next chapter.

3.2 Proposed Static Timing Analysis Constraint

GCP transforms asynchronous handshake behavior into a structured hierarchy of virtual clocks that commercial STA engines can interpret directly. Instead of modifying circuit structure or relying on unsupported minimum-delay constraints, GCP treats each handshake firing event as a virtual clock edge and propagates this timing reference across the request–acknowledge network. Through selective disabling of problematic arcs and systematic isolation of handshake domains, GCP resolves architectural loops, local-loop hazards, and false timing paths caused by conditional or data-dependent behavior. This section introduces the core ideas behind GCP and outlines its constraint-generation process.

In a two-phase BD controller, the firing event marks the exact instant when downstream registers capture valid data. GCP interprets this firing transition as a virtual timing reference and models it as a generated clock. The request of the first active controller becomes the master clock, and each subsequent controller emits a divide-by-1 generated clock derived from its phase-transition flip-flop. These clocks share the same period and phase, encoding the causal order of handshake events in a synchronous STA-compatible form.

The GCP flow generates STA constraints after synthesis, placement, and routing, and systematically removes timing anomalies that commonly appear in asynchronous BD circuits, especially those involving feedback paths, conditional routing, or data-dependent behavior. The overall procedure consists of five stages.

1. Handshake Extraction

   The post-layout netlist is analyzed to identify all handshake controllers and their Req/Ack connections. Each element is classified as either an active controller (e.g., Click or Phase-Decoupled Click) or a passive controller (e.g., MUX, DEMUX, Fork, Join). These relationships are used to construct a directed handshake graph that captures the underlying communication structure.

2. Clock Creation and Propagation

   A root virtual clock is created at the request input of the first active controller using "create_clock". For each downstream controller, GCP generates a corresponding virtual clock using "create_generated_clock" with an identical frequency and zero-phase offset. These clocks emulate event transitions along the handshake network, allowing asynchronous timing behavior to be evaluated within the synchronous STA framework.

3. Clock Grouping and Loop Handling

   To prevent infinite timing propagation and false violations, GCP detects and resolves loop-related issues at multiple levels.

   - Local loops, such as internal feedback between phase_in and phase_out signals within a single controller, are removed by disabling the corresponding arcs using "set_disable_timing".
   - Architectural loops, which arise from feedback cycles in sequential or iterative rings, are identified through the handshake graph. GCP disables the returning arc at the loop boundary so that STA evaluates only the forward timing path.
   - Passive controllers may replicate or merge handshake signals without generating new events, often creating misleading dependencies. GCP temporarily disables arcs on non-active branches and groups virtual clocks only through the active path. For controllers with multiple outputs, each branch is analyzed independently by enabling one branch at a time while suppressing the others.

   These steps ensure a loop-free timing graph and consistent slack computation across the entire circuit.

4. Selection-Delay Fixing

   Selective or frequency-adaptive pipelines often contain multiple matched-delay channels, and standard STA engines cannot analyze all channels concurrently. GCP therefore evaluates each channel individually:

   - Non-selected channels are disabled using "set_disable_timing".
   - The active channel receives its own virtual clock domain, and "set_multicycle_path 0" aligns its control and data delays precisely.
   - Each remaining channel is activated in turn while the others are disabled.
   - To evaluate additional channels without re-running synthesis, "reset_timing" is used after each insertion to clear prior STA results and establish a clean timing environment for subsequent analysis.
   - After all channels are analyzed, GCP compares slack values and assigns delay elements according to their measured latencies. The longest channel becomes the reference, while shorter channels are tuned to maintain uniform slack margins.

This per-channel analysis guarantees correct timing verification for dynamic delay-selection structures and prevents false setup violations caused by inactive branches.

5. Timing Report Generation

   Once all constraints are applied, GCP performs a full STA run. Commands such as "report_timing_summary" and "report_timing" evaluate setup and hold slacks along each handshake path defined by the generated clocks. Data and control domains are checked separately to confirm that the bundling constraint is met, and unconstrained or pruned paths are highlighted automatically to ensure complete timing coverage. The resulting slack summaries provide quantitative confirmation of timing correctness throughout the design.

Through this structured sequence, GCP eliminates local-loop and architectural-loop hazards, isolates asynchronous handshake domains, and resolves the complexities introduced by selection-delay structures. The generated XDC constraints allow commercial STA tools to verify timing in a deterministic, single-pass manner, without any manual edits, across a wide range of asynchronous BD architectures, including conditional pipelines and ring-based systems.

3.3 Example of STA for Timing Path Extraction

This section presents practical examples illustrating how GCP resolves timing-analysis challenges in representative asynchronous BD circuits. Three circuit structures are examined: 1) an iterative Fibonacci pipeline that contains an architectural loop, 2) a conditional handshake pipeline incorporating passive controllers, and 3) a linear pipeline with selection-delay channels requiring multi-cycle timing semantics. These examples show how GCP reconstructs valid timing paths, prunes false or cyclic dependencies, and expresses asynchronous behavior in a form suitable for accurate static timing analysis on FPGA platforms.

3.3.1 Iterative Fibonacci Pipeline: Handling an Architectural Loop

The Fibonacci (FIBO) pipeline includes a global feedback connection that creates an architectural loop, as illustrated in Figure 3.1. A conventional STA engine repeatedly propagates timing through this loop because no explicit clock boundary terminates the returning path, resulting in unbounded traversal and unstable slack values. GCP resolves this by first establishing a clear timing reference. A "create_clock" directive defines the master request clock at the entry controller, and "create_generated_clock" propagates this timing through the forward path of the pipeline. To avoid unintended interactions between the master and propagated clocks, "set_clock_groups -asynchronous" is applied to isolate them into separate timing domains. Once these domains are defined, GCP detects the feedback edge and disables the returning timing arc using "set_disable_timing", as shown in Figure 3.2. This effectively collapses the architectural loop into a single forward traversal, allowing STA to evaluate only one iteration of the pipeline instead of repeatedly expanding the feedback path. Additionally, each Click controller contains small internal loops arising from its phase-decoupled timing logic. Although these loops are functionally necessary, they do not represent valid timing dependencies. GCP disables these internal arcs as well, ensuring that STA observes only the genuine request-to-register path required for setup and hold verification.

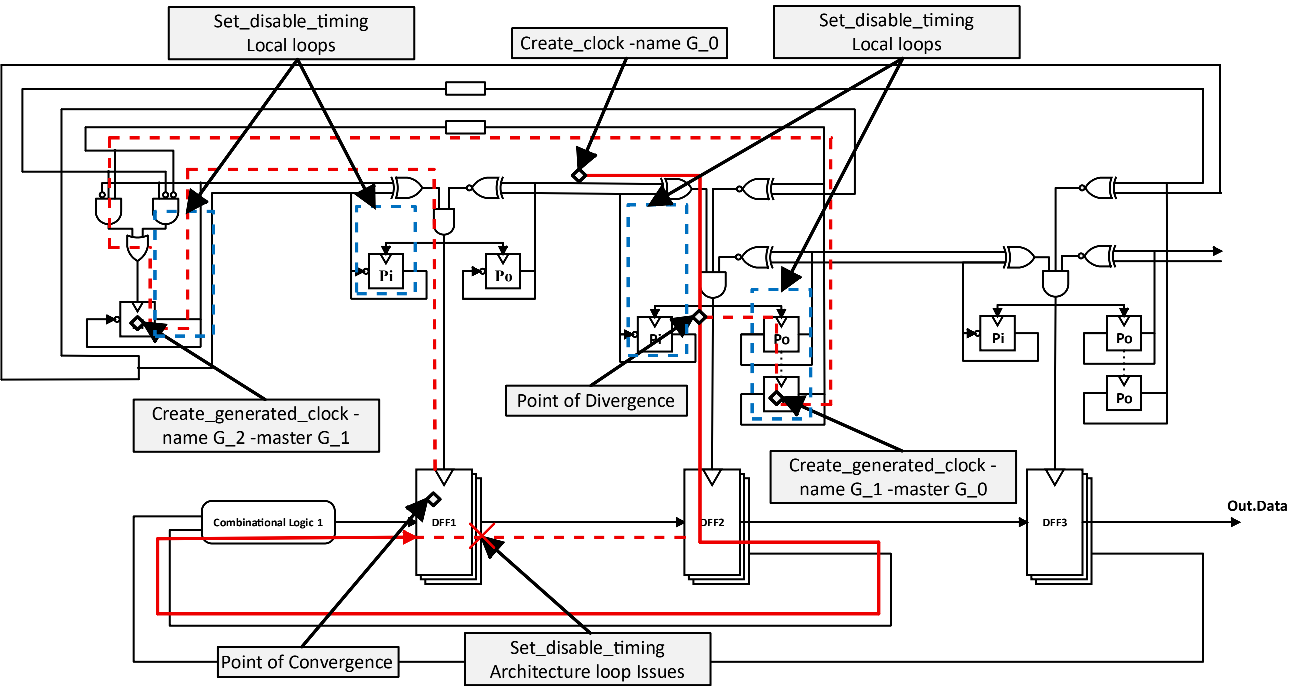
3.3.2 Conditional Handshake Circuit: Passive Controller Interpretation

A conditional handshake pipeline that uses a DEMUX–MUX pair, shown in Figure 3.3, illustrates how passive controllers introduce ambiguity during timing analysis. Without additional constraints, the STA engine propagates the master clock through every fanout branch of the DEMUX and allows the paths to reconverge at all MUX inputs, creating timing routes that never occur in actual execution. GCP resolves this by first defining the controlling request clock using "create_clock", and then propagating this timing through the intended handshake path with "create_generated_clock". To prevent interference between branches, "set_clock_groups -asynchronous" is applied to separate the active channel from the inactive ones. Next, GCP identifies and disables unused DEMUX and MUX arcs via "set_disable_timing", as shown in Figure 3.4. Once these non-active paths are pruned, the generated clock propagates only along the selected handshake route, eliminating false reconvergence and restoring a correct, branch-specific timing structure. With these constraints in place,

STA can accurately evaluate the bundling constraint for the active dataflow path without being misled by conditional or inactive handshake branches.

3.3.3 Linear Pipeline with Selection Delay: Multi-Channel Timing Semantics

The structure in Figure 3.5 extends the linear BD pipeline by incorporating a selection-delay unit that provides multiple delay channels, each representing a different latency configuration. Under default STA behavior, these channels are interpreted as simultaneously active, causing the timing engine to propagate timing through all branches. This leads to pessimistic slack values and false setup-time violations because STA evaluates paths that never occur during actual operation. GCP resolves this by explicitly defining separate timing domains and enforcing per-channel exclusivity. The analysis begins by assigning the master handshake clock through "create_clock", followed by "create_generated_clock" to propagate timing through the intended delay channel. All non-selected channels are then pruned using "set_disable_timing", which removes their arcs from the timing graph and ensures that STA interprets the selection-delay structure as a single, deterministic timing path. The constraint sequence is illustrated in Figure 3.6. Once the timing graph is restricted to the active branch, "report_timing" yields the true bundled-data slack for that channel, and the example timing report in Figure 3.7 confirms that only the selected delay path influences the computed setup margin. For designs with multiple selection channels, GCP applies "reset_timing" and reuses the same constraint set for each channel without requiring re-synthesis. The resulting per-channel slack values guide the delay-placement procedure, where candidate LUT positions are evaluated and the optimal delay is selected. This channel-specific interpretation enables accurate multi-channel timing semantics and supports deterministic single-pass analysis even in pipelines with complex selection-delay structures.

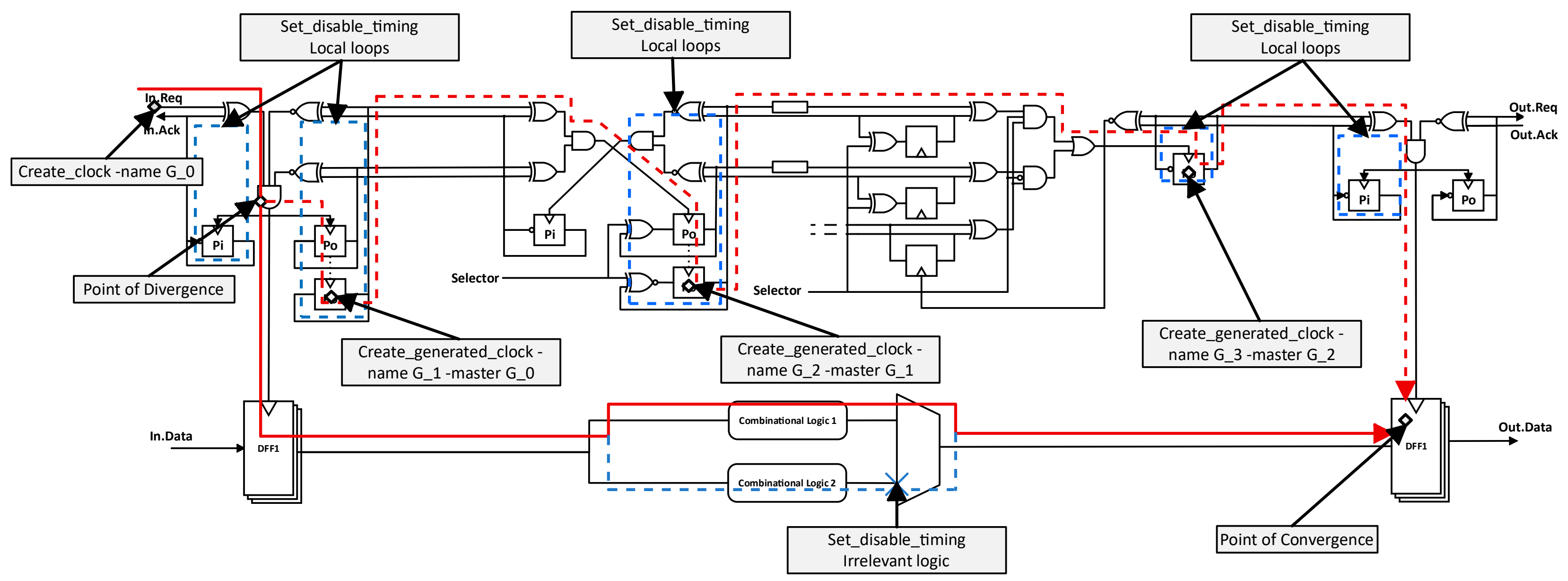


- set_clock_groups -asynchronous -group {G_0 G_1 G_2}
- set_multicycle_path 0

Figure 3.1. The structure of the FIBO circuit, which applies STA by using the GCP.

Figure 3.2. The example of the STA command of GCP on the FIBO circuit, which has architecture loop issues.

```
 1. reset_timing
 2.
 3. # create master and generated clock
 4. create_clock -name G_0 -period 3 [get_pins RF_0/click_inferred_i_1/O]
 5. create_generated_clock -name G_1 -divide_by 1 -source [get_pins RF_0/phase_c_reg/C] [get_pins
RF_0/phase_c_reg/Q] -master G_0  -add
 6. create_generated_clock -name G_2 -divide_by 1 -source [get_pins J_0/phase_reg/C] [get_pins J_0/phase_reg/Q] -
master G_1 -add
 7. set_clock_groups -asynchronous -group {G_0 G_1 G_3}
 8.
 9. # local loop issues handled
10. set_disable_timing -from I0 -to O [get_cells RF_0/phase_a_i_1]
11. set_disable_timing -from I0 -to O [get_cells RF_0/phase_b_i_1]
12. set_disable_timing -from I0 -to O [get_cells RF_0/phase_c_i_1]
13.
14. # local loop issues handled
15. set_disable_timing -from I0 -to O [get_cells R_0/phase_in_i_1]
16. set_disable_timing -from I0 -to O [get_cells R_0/phase_out_i_1]
17.
18. # local loop issues handled
19. set_disable_timing -from I0 -to O [get_cells J_0/phase_i_1]
20. set_disable_timing -from I0 -to O [get_cells J_0/click_inferred_i_1]
21. set_disable_timing -from I1 -to O [get_cells J_0/click_inferred_i_1]
22.
23. # architecture loop issues handled
24. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[0]]
25. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[1]]
26. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[2]]
27. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[3]]
28. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[4]]
29. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[5]]
30. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[6]]
31. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[7]]
32. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[8]]
33. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[9]]
34. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[10]]
35. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[11]]
36. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[12]]
37. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[13]]
38. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[14]]
39. set_disable_timing -from C -to Q [get_cells R_0/data_sig_reg[15]]
40.
41. # handle non-uate
42. set_clock_sense -positive [get_pins J_0/click_inferred_i_1/O]
43. set_clock_sense -positive [get_pins RF_0/click_inferred_i_1/O]
44. set_clock_sense -positive [get_pins R_0/click_inferred_i_1/O]
45.
46. set_clock_groups -asynchronous -group {G_0 G_1 G_2}
47. set_multicycle_path 0  -setup -to [all_clocks]
48. report_timing -setup -no_report_unconstrained -file "setup.rpt"
```



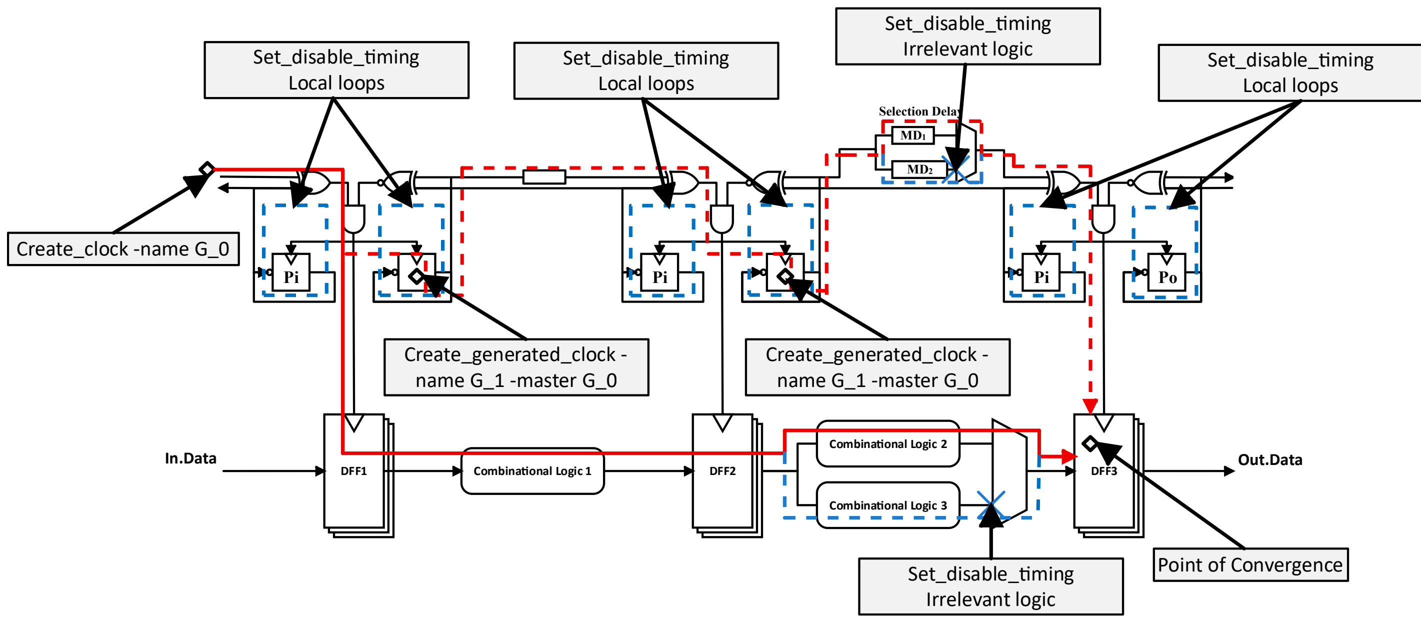
- set_clock_groups -asynchronous -group {G_0 G_1 G_2}
- set_multicycle_path 0

Figure 3.3. *The structure of the conditional handshake circuit, which applies STA by using the GCP method.*

Figure 3.4. The example of the STA command of GCP on the passive conditional handshake pipeline.

```
 1. reset_timing
 2.
 3. # create master and generated clock
 4. create_clock -name G_0 -period 0.001 [get_pins RF_0/in_req]
 5. create_generated_clock -name G_1 -divide_by 1 -source [get_pins RF_0/phase_c_reg/C] [get_pins
RF_0/phase_c_reg/Q] -master G_0 -add
 6. create_generated_clock -name G_2 -divide_by 1 -source [get_pins DX_0/phase_b_reg/C] [get_pins
DX_0/phase_b_reg/Q] -master G_1 -add
 7. create_generated_clock -name G_3 -divide_by 1 -source [get_pins MX_0/phase_c_reg/C] [get_pins
MX_0/phase_c_reg/Q] -master G_2 -add
 8.
 9. # local loop issues handled
10. set_disable_timing -from I0 -to O [get_cells RF_0/phase_c_i_1]
11. set_disable_timing -from I0 -to O [get_cells RF_0/phase_b_i_1]
12. set_disable_timing -from I0 -to O [get_cells RF_0/phase_a_i_1]
13.
14. # local loop issues handled
15. set_disable_timing -from I0 -to O [get_cells DX_1/phase_b_i_1]
16. set_disable_timing -from I0 -to O [get_cells DX_1/phase_b_i_2]
17. set_disable_timing -from I1 -to O [get_cells DX_1/phase_b_i_1]
18. set_disable_timing -from I1 -to O [get_cells DX_1/phase_b_i_2]
19.
20. # local loop issues handled
21. set_disable_timing -from I0 -to O [get_cells DX_1/phase_a_i_1]
22. set_disable_timing -from I0 -to O [get_cells DX_1/phase_a_i_2]
23. set_disable_timing -from I1 -to O [get_cells DX_1/phase_a_i_2]
24. set_disable_timing -from I0 -to O [get_cells DX_1/phase_c_i_1]
25. set_disable_timing -from I1 -to O [get_cells DX_1/phase_c_i_1]
26.
27. # Disable un-analysis channel
28. set_disable_timing -from I2 -to O [get_cells DX_1/phase_a_i_2]
29. set_disable_timing -from I3 -to O [get_cells DX_1/phase_a_i_2]
30.
31. # local loop issues handled
32. set_disable_timing -from I0 -to O [get_cells MX_0/phase_c_i_1]
33. set_disable_timing -from I0 -to O [get_cells MX_0/phase_b_i_1]
34. set_disable_timing -from I1 -to O [get_cells MX_0/phase_b_i_1]
35. set_disable_timing -from I0 -to O [get_cells MX_0/phase_a_i_1]
36. set_disable_timing -from I1 -to O [get_cells MX_0/phase_a_i_1]
37. set_disable_timing -from I0 -to O [get_cells MX_0/click_ack_inferred_i_1]
38. set_disable_timing -from I1 -to O [get_cells MX_0/click_ack_inferred_i_1]
39. set_disable_timing -from I1 -to O [get_cells MX_0/click_ack_inferred_i_1]
40. set_disable_timing -from I2 -to O [get_cells MX_0/click_req_inferred_i_1]
41. set_disable_timing -from I3 -to O [get_cells MX_0/click_req_inferred_i_1]
42. set_disable_timing -from I5 -to O [get_cells MX_0/click_req_inferred_i_1]
43.
44. # Disable un-analysis channel
45. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[0]_INST_0]
46. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[1]_INST_0]
47. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[2]_INST_0]
48. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[3]_INST_0]
49. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[4]_INST_0]
50. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[5]_INST_0]
51. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[6]_INST_0]
52. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[7]_INST_0]
53. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[8]_INST_0]
54. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[9]_INST_0]
55. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[10]_INST_0]
56. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[11]_INST_0]
```

```
57. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[12]_INST_0]
58. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[13]_INST_0]
59. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[14]_INST_0]
60. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[15]_INST_0]
61.
62. # local loop issues handle
63. set_disable_timing -from I0 -to O [get_cells R_0/phase_in_i_1]
64. set_disable_timing -from I0 -to O [get_cells R_0/phase_out_i_1]
65.
66. # handle non-uate
67. set_clock_sense -positive [get_pins RF_0/click_inferred_i_1/O]
68. set_clock_sense -positive [get_pins DX_0/phase_a_i_2/O]
69. set_clock_sense -positive [get_pins DX_0/phase_b_i_2/O]
70. set_clock_sense -positive [get_pins MX_0/click_req_inferred_i_1/O]
71. set_clock_sense -positive [get_pins R_0/click_inferred_i_1/O]
72.
73. set_clock_groups -asynchronous -group {G_0 G_1 G_2 G_3}
74. set_multicycle_path 0  -setup -to [all_clocks]
75. report_timing -setup -no_report_unconstrained -file "setup.rpt"
```



- set_clock_groups -asynchronous -group {G_0 G_1 G_2}
- set_multicycle_path 0

Figure 3.5. *The structure of a linear pipeline with selection delay, which applies STA by using the GCP method.*

Figure 3.6. *The example of the STA command of GCP on the selection delay.*

```
 1. reset_timing
 2.
 3. # create master and generated clock
 4. create_clock -name G_0 -period 0.001 [get_pins R_0/in_req]
 5. create_generated_clock -name G_1 -divide_by 1 -source [get_pins R_0/phase_out_i_1/C] [get_pins
R_0/phase_out_i_1/Q] -master G_0 -add
 6. create_generated_clock -name G_2 -divide_by 1 -source [get_pins R_1/phase_out_i_1/C] [get_pins
R_1/phase_out_i_1/Q] -master G_1 -add
 7.
 8. # local loop issues handled
 9. set_disable_timing -from I0 -to O [get_cells R_0/phase_in_i_1]
10. set_disable_timing -from I3 -to O [get_cells R_0/fire_INST_0]
11.
12. # local loop issues handled
13. set_disable_timing -from I0 -to O [get_cells R_1/phase_in_i_1]
14. set_disable_timing -from I3 -to O [get_cells R_1/fire_INST_0]
15.
16. # local loop issues handled
17. set_disable_timing -from I0 -to O [get_cells R_2/phase_in_i_1]
18. set_disable_timing -from I3 -to O [get_cells R_2/fire_INST_0]
19.
20. # Disable un-analysis channel
21. #set_disable_timing [get_pins DSU_1/delay_element_DSU_EX_0/*]
22.
23. # Disable un-analysis channel
24. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[0]_INST_0]
25. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[1]_INST_0]
26. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[2]_INST_0]
27. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[3]_INST_0]
28. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[4]_INST_0]
29. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[5]_INST_0]
30. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[6]_INST_0]
31. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[7]_INST_0]
32. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[8]_INST_0]
33. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[9]_INST_0]
34. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[10]_INST_0]
35. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[11]_INST_0]
36. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[12]_INST_0]
37. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[13]_INST_0]
38. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[14]_INST_0]
39. set_disable_timing -from I1 -to O [get_cells MUX/outC_data[15]_INST_0]
40.
41. # handle non-uate
42. set_clock_sense -positive [get_pins R_0/click_inferred_i_1/O]
43. set_clock_sense -positive [get_pins R_1/click_inferred_i_1/O]
44. set_clock_sense -positive [get_pins R_2/click_inferred_i_1/O]
45.
46. set_clock_groups -asynchronous -group {G_0 G_1 G_1}
47. set_multicycle_path 0  -setup -to [all_clocks]
48. report_timing -setup -no_report_unconstrained -file "setup.rpt"
```

Figure 3.7. The example of a timing report.

```
1. Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
2. ------------------------------------------------------------------------------------------------------------
3. | Tool Version : Vivado v.2022.1 (win64) Build 3526262 Mon Apr 18 15:48:16 MDT 2022
4. | Date         : Tue Apr  1 07:36:40 2025
5. | Host         : DESKTOP-29E3E9D running 64-bit major release  (build 9200)
6. | Command      : report_timing -setup -no_report_unconstrained -file
D:/Circuit/async_FIBO_verilog/STA/setup_RF0_to_R0.rpt
7. | Design       : Fib
8. | Device       : 7z020-clg400
9. | Speed File   : -1  PRODUCTION 1.12 2019-11-22
10. ------------------------------------------------------------------------------------------------------------
11.
12. Timing Report
13.
14. Slack (MET) :             0.480ns  (required time - arrival time)
15.   Source:                 RF_0/data_reg_reg[7]/C
16.                             (rising edge-triggered cell FDCE clocked by STU1  {rise@0.000ns fall@1.500ns period=3.000ns})
17.   Destination:            R_0/data_sig_reg[15]/D
18.                             (rising edge-triggered cell FDCE clocked by STU1_2  {rise@0.000ns fall@1.500ns
period=3.000ns})
19.   Path Group:             STU1_2
20.   Path Type:              Setup (Max at Fast Process Corner)
21.   Requirement:            0.000ns  (STU1_2 rise@0.000ns - STU1 rise@0.000ns)
22.   Data Path Delay:        1.076ns  (logic 0.538ns (50.021%)  route 0.538ns (49.979%))
23.   Logic Levels:           4  (CARRY4=3 LUT2=1)
24.   Clock Path Skew:        1.504ns (DCD - SCD + CPR)
25.     Destination Clock Delay (DCD):    1.807ns = ( 4.807 - 3.000 )
26.     Source Clock Delay      (SCD):    0.303ns
27.     Clock Pessimism Removal (CPR):    0.000ns
28.   Timing Exception:       MultiCycle Path   Setup -end   0
29.
30.     Location             Delay type                Incr(ns)  Path(ns)    PBlock              Netlist Resource(s)
31.   -------------------------------------------------------------------    -------------------------------------------
32.                         (clock STU1 rise edge)       0.000     0.000 r
33.     SLICE_X1Y19          LUT3                         0.000     0.000 r  pblock_1            RF_0/click_inferred_i_1/O
34.                          net (fo=19, routed)          0.303     0.303                        RF_0/click
35.     SLICE_X3Y19          FDCE                                          r  pblock_1           RF_0/data_reg_reg[7]/C
36.   -------------------------------------------------------------------    -------------------------------------------
37.     SLICE_X3Y19          FDCE (Prop_fdce_C_Q)         0.175     0.478 r  pblock_1            RF_0/data_reg_reg[7]/Q
38.                          net (fo=2, routed)           0.538     1.016                        CL_0/inB_data[7]
39.     SLICE_X2Y19          LUT2 (Prop_lut2_I1_O)        0.056     1.072 r  pblock_1
CL_0/outC_data[4]_INST_0_i_1/O
40.                          net (fo=1, routed)           0.000     1.072                        CL_0/outC_data[4]_INST_0_i_1_n_0
41.     SLICE_X2Y19          CARRY4 (Prop_carry4_S[3]_CO[3])
42.                                                       0.143     1.215 r  pblock_1            CL_0/outC_data[4]_INST_0/CO[3]
43.                          net (fo=1, routed)           0.000     1.215                        CL_0/outC_data[4]_INST_0_n_0
44.     SLICE_X2Y20          CARRY4 (Prop_carry4_CI_CO[3])
45.                                                       0.050     1.265 r  pblock_1            CL_0/outC_data[8]_INST_0/CO[3]
46.                          net (fo=1, routed)           0.000     1.265                        CL_0/outC_data[8]_INST_0_n_0
47.     SLICE_X2Y21          CARRY4 (Prop_carry4_CI_O[3])
48.                                                       0.114     1.379 r  pblock_1            CL_0/outC_data[12]_INST_0/O[3]
49.                          net (fo=1, routed)           0.000     1.379                        R_0/in_data[15]
50.     SLICE_X2Y21          FDCE                                          r  pblock_1           R_0/data_sig_reg[15]/D
51.   -------------------------------------------------------------------    -------------------------------------------
52.
53.                         (clock STU1_2 rise edge)     0.000     0.000 r
54.     SLICE_X1Y19          LUT3                         0.000     0.000 r  pblock_1            RF_0/click_inferred_i_1/O
55.                          net (fo=19, routed)          0.262     0.262                        RF_0/click
56.     SLICE_X3Y19          FDPE (Prop_fdpe_C_Q)         0.141     0.403 r  pblock_1            RF_0/phase_c_reg/Q
```

```
57.                    net (fo=3, routed)          0.121     0.525                 J_0/inB_req
58.   SLICE_X0Y19         LUT3 (Prop_lut3_I2_O)       0.045     0.570 r  pblock_1          J_0/click_inferred_i_1/O
59.                    net (fo=1, routed)          0.311     0.881                 J_0/click
60.   SLICE_X0Y19         FDCE (Prop_fdce_C_Q)        0.164     1.045 r  pblock_1          J_0/phase_reg/Q
61.                    net (fo=3, estimated)       0.198     1.243                 CL_0/delay_req/d
62.   SLICE_X2Y17         LUT1 (Prop_lut1_I0_O)       0.045     1.288 r  pblock_1          CL_0/delay_req/delay0/O
63.                    net (fo=1, estimated)       0.171     1.459                 R_0/in_req
64.   SLICE_X3Y19         LUT4 (Prop_lut4_I0_O)       0.045     1.504 r  pblock_1          R_0/click_inferred_i_1/O
65.                    net (fo=18, routed)         0.304     1.807                 R_0/click
66.   SLICE_X2Y21         FDCE                                 r  pblock_1          R_0/data_sig_reg[15]/C
67.                    clock pessimism             0.000     1.807
68.   SLICE_X2Y21         FDCE (Setup_fdce_C_D)       0.051     1.858    pblock_1          R_0/data_sig_reg[15]
69.   -------------------------------------------------------------------
70.                    required time                         1.858
71.                    arrival time                         -1.379
72.   -------------------------------------------------------------------
73.                    slack                             0.480
```

3.4 Example Applications

Static timing analysis provides numerical timing information for a mapped asynchronous circuit, but these values must be organized and interpreted before they can meaningfully guide delay placement and pipeline verification. This section introduces the reporting framework used in this work to extract, classify, and analyze timing paths that are relevant to bundled-data correctness. It explains how timing information is collected from Vivado, how handshake-specific paths are separated from general clock domains, and how these structured reports are used to evaluate bundling accuracy, selection-delay alignment, and loop-safety conditions in complex asynchronous pipelines.

3.4.1 FIR Filter Circuit: Linear Pipeline

A finite impulse response (FIR) filter is a widely used digital signal-processing component. In this study, an 11-tap FIR circuit with 16-bit inputs and coefficients was implemented using a fully parallel architecture mapped onto a linear asynchronous BD pipeline. In this organization, Register+Fork (RF) controllers distribute input samples to the multiplier blocks, while Joint+Register (JR) controllers collect and forward partial results into the adder network, as illustrated in Figure 3.8.

This circuit was chosen to evaluate whether the proposed design flow can operate effectively even when multiple valid timing paths are detected. During synthesis, the hierarchy and logical structure of the design were preserved, and both the combinational blocks and handshake controllers were placed inside PBLOCKS to maintain consistent physical organization during implementation. Although the FIR pipeline does not include architectural loops or shared combinational-block hazards, it still presents challenges in terms of path extraction and delay interpretation. Among the thirty-one setup paths reported by the timing analyzer, only twenty-one corresponded to true bundled-data timing paths and were therefore used to determine the required delay-element latencies.

GCP generated twenty-one constraint files, each containing the specific timing rules needed to evaluate the latency of a corresponding delay element. Table 3.1 summarizes the setup and hold slack values across three refinement rounds. In the first round, all setup paths exhibited negative slack because no delay elements were present, while all hold paths remained positive. After inserting delay elements in the second round, several setup paths achieved positive slack, with all hold paths still meeting the constraint. In the third round, adjusting the inserted delays resulted in positive slack across every setup path, and all hold paths continued to satisfy timing requirements. Without placement constraints, achieving fully positive slack would have required approximately five rounds.

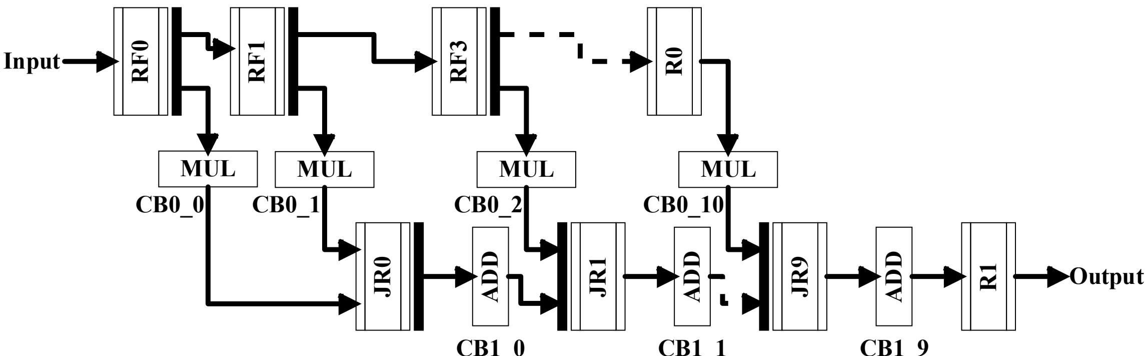


Figure 3.8. *Block diagram of the FIR filter asynchronous circuit.*

Table 3.1. *Timing paths of the FIR circuit with final setup and hold timing results.*

No	Timing path	Setup timing (Round)			Hold timing
		1st	2nd	3rd	
1	RF1→CB0_1→JR0	-2.19	1.37	0.43	0.71
2	RF2→CB0_2→JR1	-4.80	1.34	0.64	1.10
3	RF3→CB0_3→JR2	-4.83	1.42	0.90	1.17
4	RF4→CB0_4→JR3	-3.55	0.28	0.75	0.71
5	RF5→CB0_5→JR4	-2.59	0.57	0.47	0.47
6	RF6→CB0_6→JR5	-5.12	0.53	0.73	1.35
7	RF7→CB0_7→JR6	-5.43	1.00	0.81	0.76
8	RF8→CB0_8→JR7	-4.08	0.54	0.71	0.84
9	RF9→CB0_9→JR8	-4.16	0.43	0.47	1.45
10	R0→CB0_10→JR0	-5.66	1.49	0.93	1.09
11	JR0→CB1_0→JR1	-1.98	0.27	0.79	0.84
12	JR1→CB1_1→JR2	-1.35	0.10	0.56	0.82
13	JR2→CB1_2→JR3	-1.12	0.86	0.52	0.65
14	JR3→CB1_3→JR4	-1.06	0.17	0.35	0.63
15	JR4→CB1_4→JR5	-2.26	1.10	0.71	1.03
16	JR5→CB1_5→JR6	-1.34	0.04	0.43	0.95
17	JR6→CB1_6→JR7	-1.66	1.27	0.31	0.82
18	JR7→CB1_7→JR8	-1.58	0.60	0.51	0.98
19	JR8→CB1_8→JR9	-1.52	0.97	0.93	0.52
20	JR9→CB1_9→R1	-1.75	1.05	0.74	0.55

Table 3.2. *The comparison results between synchronous and asynchronous FIR circuits implemented using GCP.*

Type	Area (10^{-3})	Latency (ns)	Energy (nJ)
Synchronous	5.76	89.05	9.89
Asynchronous	7.52	97.08	10.58
Improvement	-23.44%	-8.27%	-6.52%

3.4.2 Fibonacci (FIBO) Generator: Sequential Ring Pipeline

The Fibonacci (FIBO) generator shown in Figure 3.9 has been discussed previously in the literature, but no accompanying STA or delay-matching methodology was provided. Although the circuit uses only a single ADD block to combine values from two Register+Fork (RF) controllers, its structure is significantly more complex than a linear pipeline because it forms a sequential ring with two nested loops—an inner ring (RF0→J0→CL0→R0) and an outer ring (RF1→J0→CL0→R0→RF0). This configuration presents all major STA challenges, including architectural loops, shared combinational blocks, and multiple overlapping timing dependencies.

A phase-decoupled Click controller was used in this implementation to support deterministic ring initialization. Each RF controller receives an initial token: RF0 is initialized with a value of 1, and both RF0

and RF1 are assigned an initial output phase of 1 according to the ring's initialization policy. The GCP analysis identified four timing paths in the circuit, although only two were required to determine the delay-element latency, as summarized in Table 3.3. GCP automatically resolved the architectural loop by detecting the feedback formed by paths such as RF0→J0→CL0→R0 and R0→RF0. Because the CL0 block is shared across two timing paths, the delay-element latency was selected based on the path with the strictest requirement.

After inserting a delay element in the second round, all setup paths achieved positive slack. The circuit is small and includes only a single delay element, so even without explicit placement management, two rounds were sufficient to achieve timing closure. Table 3.4 presents a comparison between synchronous and asynchronous implementations of the FIBO circuit constructed under the GCP methodology.

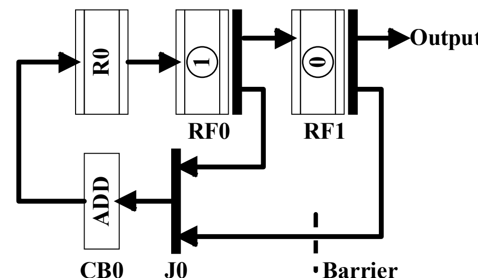


Figure 3.9. Block diagram of the FIBO asynchronous circuit.

Table 3.3. Timing paths of the FIBO circuit with final setup and hold timing results.

No	Timing path	Setup timing			Hold timing
		1st	2nd	3rd	
1	RF0→J0→CB0→R0	-0.63	0.60	-	0.34
2	RF1→J0→CB0→R0	-0.71	0.48	-	0.48

Table 3.4. The comparison results between synchronous and asynchronous FIBO circuits implemented using GCP.

Type	Area (10^{-3})	Latency (ns)	Energy (nJ)
Synchronous	0.61	3.75	0.41
Asynchronous	0.92	7.56	0.81
Improvement	-33.87	-50.40%	-49.38%

3.4.3 AES Encryption: Iterative Ring Pipeline

The Advanced Encryption Standard (AES) implements a symmetric-key encryption algorithm that operates on fixed-size data blocks and supports 128-, 192-, and 256-bit keys. Each encryption round applies a sequence of operations—SubBytes, ShiftRows, MixColumns, and AddRoundKey—while KeyExpansion generates the round keys. For AES-128, a total of 11 rounds are executed, with the final round omitting the MixColumns step.

Typical asynchronous BD implementations adopt an iterative ring architecture controlled by a passive multiplexer that tracks the round count. However, this approach generally duplicates the SubBytes, ShiftRows, and AddRoundKey combinational blocks, resulting in large area overhead. In this study, the architecture was refined by incorporating passive multiplexer and demultiplexer controllers to manage the encryption rounds, enabling a fully conditional if–else structure. This reduces combinational duplication while preserving correct round sequencing. Figure 3.10 shows the schematic of the 128-bit AES asynchronous BD circuit. The ADD module performs both AddRoundKey and KeyExpansion, SUBSH handles SubBytes and ShiftRows, and MIX performs MixColumns. Of the 11 rounds, the first nine use ADD, SUBSH, and MIX; the tenth uses ADD and SUBSH; and the last round uses only ADD. Initialization involves three rings, with MX0→RF0→CL0→F0→R0 serving as the initial controller path, where R0 begins with a value of 0 and an output phase initialized to 1.

This circuit was selected to evaluate the proposed design flow under complex timing interactions and multiple STA challenges. Using the GCP framework, nine out of fourteen extracted timing paths were used to determine the required delay-element latency. GCP automatically identified and resolved an architectural loop in the feedback formed by RF0→CL0→F0→R0 and R0→MX0→RF0. As summarized in Table 3.6, three pairs

of timing paths, (1,2), (4,5), and (7,8), share combinational blocks. In each pair, the delay-element latency was selected based on the path with the stricter requirement.

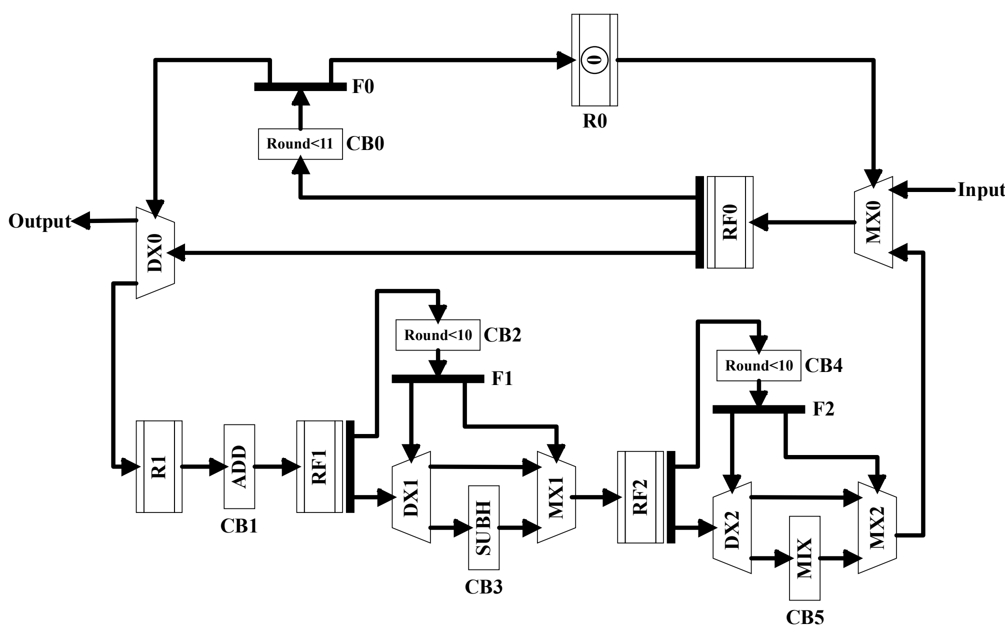


Figure 3.10. *Block diagram of the AES asynchronous circuit.*

Table 3.5. *Timing paths of the AES circuit with the final setup and hold timing result.*

No	Timing path	Setup timing			Hold timing
		1st	2nd	3rd	
1	RF0→CL0→F0→R0	-3.26	-1.21	0.49	2.79
2	RF0→CL0→F0→DX0	-3.98	-1.40	0.79	2.05
3	R1→CL1→RF1	-7.21	0.21	0.66	1.21
4	RF1→CL2→F1→DX1	-5.35	-0.71	0.59	2.60
5	RF1→CL2→F1→MX1	-3.09	-0.91	0.59	3.17
6	RF1→DX1→CL3→MX1→RF2	-6.37	1.26	0.51	0.89
7	RF2→CL4→F2→DX2	-3.73	-1.52	0.48	2.33
8	RF2→CL4→F2→MX2	-3.05	-1.88	0.72	3.21
9	RF2→DX2→CL5→MX2→MX0→RF0	-0.23	0.53	0.48	1.50

Table 3.6. *The comparison results between synchronous and asynchronous AES circuits implemented using GCP.*

Type	Area (10^{-3})	Latency (ns)	Energy (nJ)
Synchronous	27.54	290	39.76
Asynchronous	36.94	308.9	37.38
Improvement	-25.45	-6.12%	+6.37%

3.5 Comparative Evaluation with Existing Methods

Table 3.7 compares the proposed GCP method with existing timing-analysis frameworks, including ADM [18], PTC [17], BDPC [35], and other FPGA-oriented approaches [23], based on their analytical principles and FPGA compatibility. GCP achieves complete and deterministic STA coverage for two-phase BD circuits across all three architectural styles: linear, sequential ring, and iterative ring. Unlike ADM and PTC, which require multiple synthesis and routing iterations, GCP consistently achieves positive slack across all timing paths within three synthesis rounds. Its rule-based deactivation of timing arcs enables reliable handling of loop detection and passive-controller behavior, providing a scalable solution for FPGA-based asynchronous design.

A key advantage of GCP is its ability to detect shared combinational blocks and select delay-element latencies based on the strictest requirement among overlapping timing paths, a capability not supported by prior methods. As a result, the proposed approach offers higher automation, broader applicability, and improved timing accuracy on modern FPGA platforms. GCP's support for iterative and sequential rings further distinguishes it from previous STA methods. Earlier approaches, such as BDPC and PTC, were unable to fully resolve cyclic dependencies because they lacked an automatic mechanism to isolate feedback loops. In contrast, GCP constructs an acyclic clock-propagation graph through asynchronous clock-group separation and targeted false-path disabling, ensuring loop-safe analysis.

In iterative rings, GCP verifies setup and hold correctness across feedback paths without inflating delay margins, enabling efficient use of FPGA resources. In sequential rings, partial propagation preserves event causality and prevents STA from reporting timing violations along the return-token path. These capabilities make GCP suitable not only for conventional linear pipelines but also for more complex designs such as recursive arithmetic units, control-token networks, and self-timed schedulers.

Table 3.7. *Comparison of Timing-Analysis Methods for Asynchronous BD Circuits.*

Methods		**PTC [17]**	**BDPC [35]**	**Other Technique [23]**	**ADM [18]**	**GCP**
Platform		FPGA	ASIC	ASIC	ASIC	FPGA
Asynchronous circuit		BD	BD	BD	BD	BD
Structure		-Linear	-Linear	-Linear	-Linear	-Linear -Sequential ring -Iterative ring
Delay identification		Slack-based calculation	Slack-based calculation	all timing path extraction	Min/Max delay	Slack-based calculation
STA issues	**Local loop**	✓	✓	✓	×	✓
	Architecture Loop	×	×	×	×	✓
	Passive controller	✓	✓	✓	×	✓
	Sharing a combinational block	×	×	×	×	✓
	Selection delay	×	×	✓	×	✓

NA no acknowledgment*

Chapter 4
Morphological Delay Placement Constraint

Delay elements are fundamental to ensuring correct event sequencing in asynchronous BD circuits, where the absence of a global clock places timing correctness entirely on the relationship between data-path delays and control-path delays. Achieving precise delay alignment on FPGA platforms remains challenging because commercial EDA tools cannot directly model relative-delay constraints, and routing variability introduces unpredictable timing behavior. Existing flows, such as iterative synthesis and post-route ECO editing, therefore rely heavily on manual intervention, which leads to slow convergence, inconsistency across implementations, and poor scalability in complex pipelines containing multiple handshake branches or selection-delay channels.

The Morphological Delay-Placement Constraint (MDPC) addresses these challenges by introducing a geometry-aware, single-pass methodology for determining both the placement and length of delay elements. MDPC identifies candidate regions using morphological boundary detection, evaluates timing behavior at each location through slack-guided refinement, and selects delay placements that minimize hardware overhead while satisfying the bundling constraint. By integrating spatial analysis with real timing feedback, MDPC eliminates the dependency on repeated synthesis loops or manual buffer insertion and provides deterministic timing closure within standard FPGA toolchains.

This chapter presents the technical background and rationale behind the MDPC for automated delay insertion on FPGA platforms. Section 4.1 introduces the motivation for the proposed method, outlining the challenges of manual and iterative delay placement in asynchronous BD circuits. Section 4.2 describes the proposed delay-placement constraint in detail, explaining how MDPC performs boundary detection, spatial analysis, and slack-guided refinement to determine optimal delay locations and lengths. Section 4.2.1 then analyzes the computational complexity of the method, characterizing its scalability and practicality for large asynchronous designs. Finally, Section 4.3 provides a comparative evaluation with existing delay-placement approaches, highlighting MDPC's advantages in automation, timing accuracy, resource efficiency, and FPGA compatibility. Together, these sections establish the basis for a deterministic and physically grounded delay-placement strategy tailored for modern asynchronous pipelines.

4.1 Motivation

The MDPC method was developed to address the inherent limitations of both iterative and ECO-based delay-placement approaches. Its objective is to deliver an automated, single-pass process that integrates seamlessly with FPGA toolchains while maintaining precise control-path alignment. MDPC determines both the placement and the required latency of each delay element by combining spatial analysis with rule-based placement constraints. This eliminates manual tuning, reduces unnecessary hardware overhead, and ensures reproducible results across various asynchronous pipeline styles. In asynchronous bundled-data circuits, the timing behavior of each handshake controller depends critically on where delay elements are inserted and how long these delays must be. Earlier approaches required designers to choose delay locations manually, often relying on intuition, inspection of the routed layout, or repeated trial-and-error iterations. As asynchronous systems evolved toward larger designs and more sophisticated pipeline styles, such as frequency-adaptive or combined pipelines, this manual approach became increasingly impractical.

MDPC is grounded in the idea that each asynchronous module can be treated as a spatial region on the FPGA floorplan. Within this region lie natural boundary zones, interfaces between data-path logic and handshake controllers, where delay elements can be inserted most effectively. These boundaries inherently provide optimal locations for aligning the timing of request and data signals. To identify these regions, MDPC leverages morphological operations, a class of geometric transformations originally used in image processing. By applying dilation, erosion, and subtraction to the spatial representation of the asynchronous module, the method automatically extracts the boundary area where delay placement is most effective. Combined with timing feedback, MDPC then determines the appropriate delay length for each element without introducing redundant buffers or requiring manual intervention.

The MDPC is built upon four key design principles: slack optimization, resource efficiency, placement proximity, and iteration reduction. Together, these principles define how delay elements are selected, positioned, and tuned to achieve precise timing alignment in asynchronous bundled-data circuits.

1. Slack optimization

   MDPC incrementally inserts delay elements until the control path reaches the desired positive slack margin. This ensures that request transitions arrive only after the data-path computation stabilizes, avoiding both under-delayed and over-delayed configurations [15], [17], [18], [19], [20], [21] [23], [35], [38]. By converging to a controlled slack threshold, the method maintains stable handshake timing across all stages without introducing unnecessary margins.

2. Resource efficiency

   To minimize hardware overhead, particularly important on FPGA fabrics, MDPC introduces only the minimal number of LUT-based delay elements required to satisfy the timing constraint. This selective insertion avoids redundant buffering and keeps control paths compact, preserving LUT and routing resources for functional logic [15], [20], [21], [35], [38].

3. Placement proximity

   The effectiveness of a delay element depends heavily on its physical proximity to the associated combinational logic and handshake controller [15], [20]. MDPC uses morphological boundary detection to identify peripheral zones around an asynchronous module where delays can be placed most effectively. Restricting placement to these regions improves timing predictability, reduces routing variability, and simplifies debugging and verification.

4. Iteration reduction

   By combining geometric boundary analysis with automated delay tuning, MDPC achieves timing closure within a single synthesis pass. This removes the need for repeated synthesis, routing, and manual refinement cycles that characterize traditional flows [15], [20]. The result is a deterministic and highly reproducible delay-placement process.

Taken together, these principles make MDPC a structured and fully automated approach for implementing delay placement in asynchronous BD circuits, enabling accurate timing alignment without iterative tuning or manual intervention.

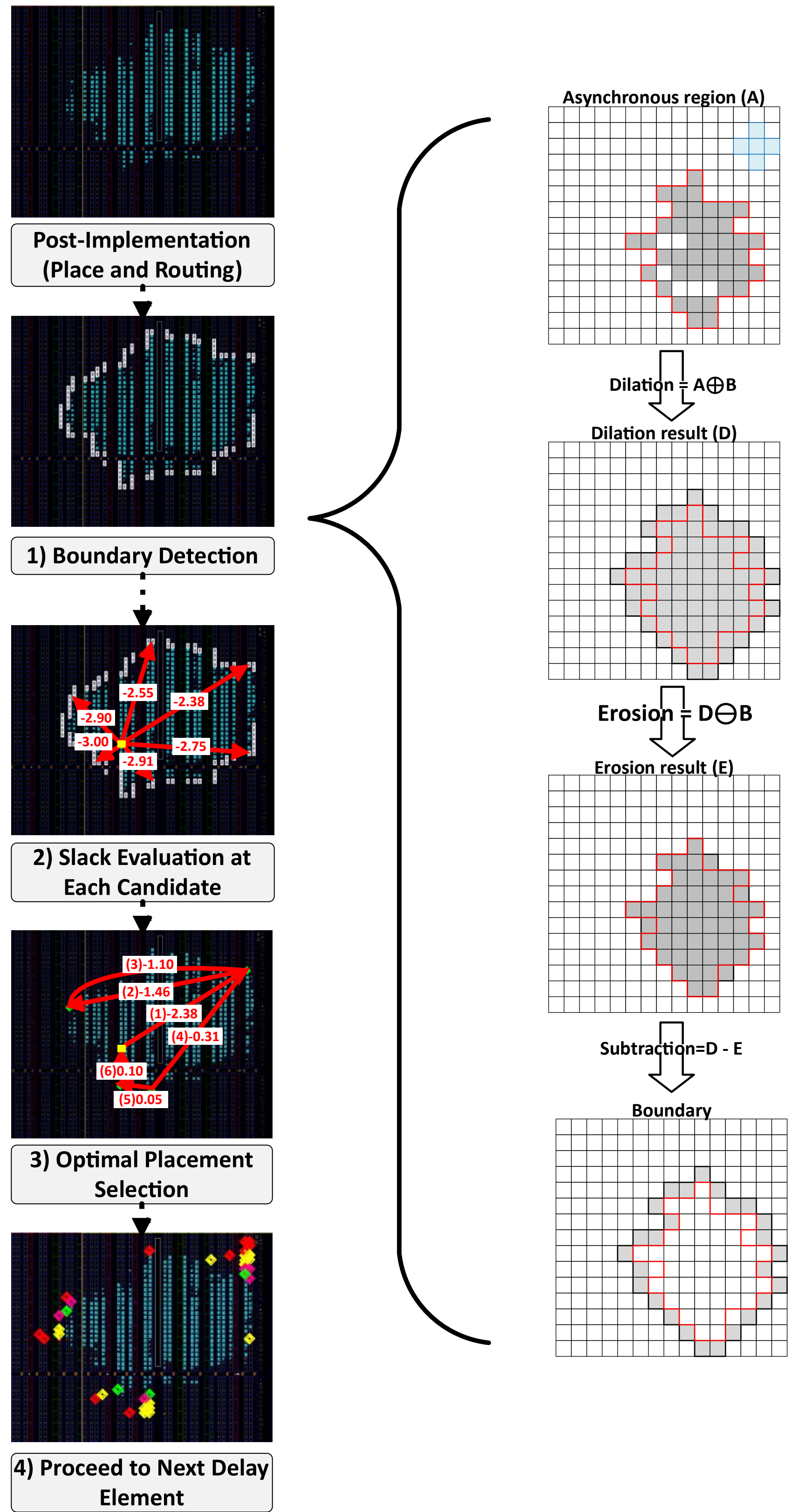


Figure 4.1. The MDPC process, which consists of four main steps and boundary detection to extract the border region of the asynchronous module.

4.2 Proposed Delay Placement Constraint

Based on these principles, MDPC is implemented as a four-step automated process that performs delay placement within a single synthesis pass, as illustrated conceptually in Figure 4.1. The flowchart highlights how MDPC transitions systematically from boundary detection to slack evaluation, optimal placement selection, and progression to subsequent delay elements.

1. Boundary Detection

   The first step identifies a set of candidate placement sites, Boundary[n], located near the periphery of the asynchronous module where the current delay element, Delay[j], can be inserted. This region is extracted using morphological dilation, erosion, and subtraction [44]. The process of boundary detection can be expressed in Figure 4.2-4.4.

   Dilation is a fundamental operation in mathematical morphology used to expand or grow the boundary of a set (or object) based on a predefined structuring element.

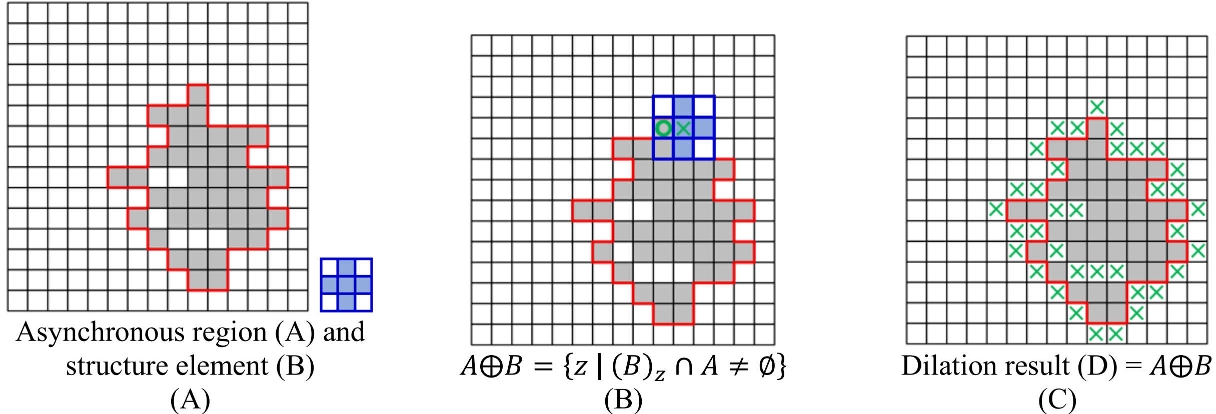


Figure 4.2. The dilation process for shrinking the boundary region, (A) the asynchronous region and its structure element, (B) the dilation condition, (C) the dilation results with mark point locations.

   Erosion is a fundamental operation in mathematical morphology used to shrink or reduce the boundary of a set (or object) according to a predefined structuring element.

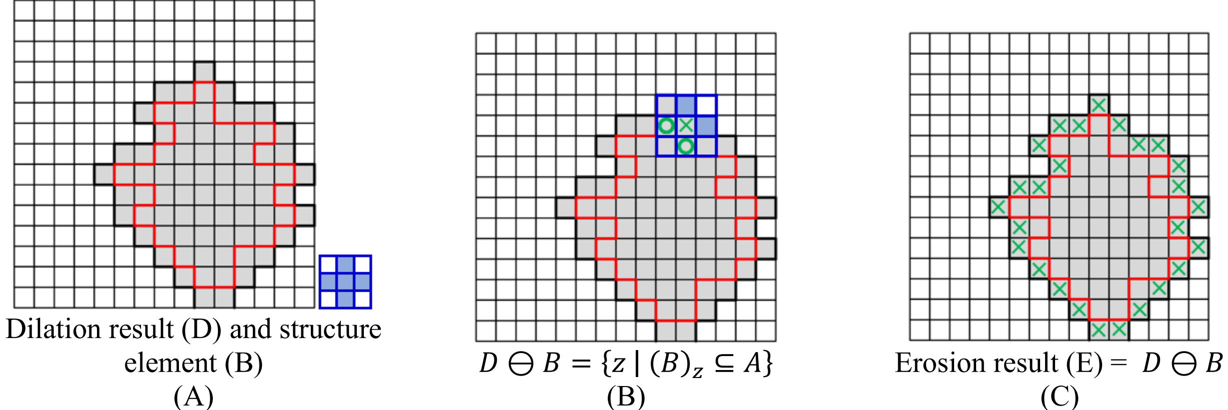
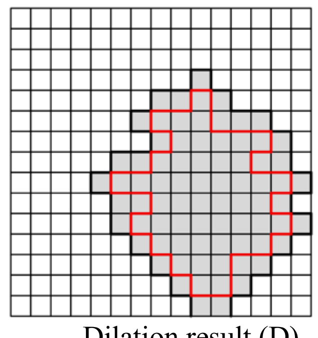
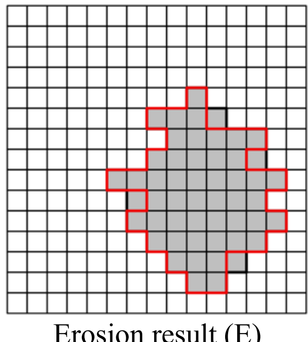


Figure 4.3. The erosion process for shrinking the boundary region, (A) the dilation region and its structure element, (B) the erosion condition, (C) the erosion results with mark point locations.

   Subtraction, also called set difference, is a morphological operation used to extract the difference between two sets. It removes elements of one set from another and is commonly used for boundary extraction.





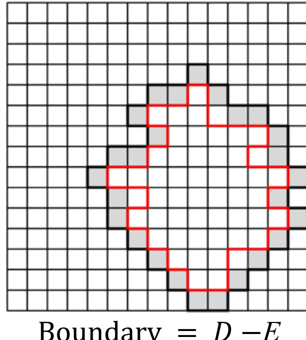


Figure 4.4. *The subtraction process for extracting the boundary region, (A) the dilation region, (B) the erosion result, (C) the boundary of the asynchronous region.*

2. Slack Evaluation

   For each candidate site Boundary[i], MDPC temporarily inserts a single-LUT delay cell. A LUT is created using "create_cell -reference LUT1" and configured with "set_property INIT 2'h2". The cell is then placed using "place_cell", and if multiple LUTs already exist, the new element is connected through "connect_net -net". After placement, STA is executed and the slack value Slacks[i] is recorded. The temporary cell is removed with "unplace_cell", placed at Boundary[i+1], and the new slack Slacks[i+1] is measured. This place–measure–remove loop continues exhaustively for all n boundary sites.

3. Optimal Placement Selection

   After evaluating all candidate sites, MDPC selects the best placement Boundary[k] based on the following criteria:

   $$MIN\{Target - Slacks[k]\} \leq Error \quad (4.4)$$

   If no site meets this tolerance, the candidate that provides the greatest improvement is chosen:

   $$Target - Slacks[k] = MAX \quad (4.5)$$

   The selected site then receives the actual LUT chain of the required length. If no candidate satisfies the target slack range, the boundary region is expanded and the search repeats from Step 1.

4. Progression to the Next Delay Element

   Once Delay[j] is finalized, the algorithm proceeds automatically to Delay[j+1]. Steps 1–3 are repeated until all delay elements required by the pipeline are inserted and verified.

To prevent interference during evaluation, unrelated asynchronous modules are temporarily removed from the layout using "unplace_cell [get_cells $cells]". After delay placement is completed, delay cells and relevant modules are locked using "set_property IS_BEL_FIXED" or "IS_LOC_FIXED", and the remaining modules are restored with "place_design". Figure 4.5 provides the pseudocode for MDPC. By combining geometric boundary detection with automated timing feedback, MDPC determines delay locations and lengths efficiently, minimizes redundant buffering, and ensures deterministic single-pass convergence, making it well suited for FPGA-based asynchronous pipelines.

Figure 4.5. *Pseudo code of MDCP.*

```
1. INPUT:
2.    A         // region of current asynchronous module
3.    B         // structural element (for morphology)
4.    Target    // desired positive slack
5.    Error     // acceptable tolerance
6.    DelayList  // list of delay elements to insert: Delay[1..M]
7.
8. PROCEDURE MDPC_Delay_Placement(A, B, Target, Error, DelayList):
9.
```

```
10.  // 0) Preparation: isolate current async module
11.  Unplace_Unrelated_Modules()
12.
13.  FOR j ← 1 TO M DO                   // iterate over each delay element Delay[j]
14.
15.    REPEAT
16.      // ----------------------------------------------------------------
17.      // 1) Boundary Detection (morphological)
18.      // ----------------------------------------------------------------
19.      D ← Dilate(A, B)            // D = A ⊕ B   (Eq. 3)
20.      E ← Erode(A, B)             // E = D ⊖ B   (Eq. 4)
21.      Boundary ← D − E            // Boundary = D − E  (Eq. 5)
22.
23.      // Boundary = {Boundary[1], ..., Boundary[n]}
24.      CandidateSites ← Enumerate_Points(Boundary)
25.      Slacks ← empty list
26.
27.      // ----------------------------------------------------------------
28.      // 2) Slack Evaluation (place–STA–unplace for each candidate)
29.      // ----------------------------------------------------------------
30.      i ← 1
31.      FOR each site S IN CandidateSites DO
32.        // create and place 1-LUT delay candidate for this site
33.        LUT ← Create_LUT1_Delay()            // equivalent to create_cell ... set_property init 2'h2
34.        Place(LUT, S)                        // equivalent to place_cell
35.
36.        Connect_To_Previous_Delay_If_Needed(LUT)  // equivalent to connect_net ...
37.
38.        Run_STA()
39.        Slacks[i] ← Read_Slack_For_Delay_Path()
40.        Unplace(LUT)                         // equivalent to unplace_cell
41.        i ← i + 1
42.      END FOR
43.
44.      // ----------------------------------------------------------------
45.      // 3) Optimal Placement Selection
46.      // ----------------------------------------------------------------
47.      // Find candidate k whose slack is closest to Target from below/within tolerance
48.      k ← ArgMin_Abs(Target − Slacks[1..n])
49.
50.      IF Abs(Target − Slacks[k]) ≤ Error THEN
51.        // good placement found
52.        FinalSite ← CandidateSites[k]
53.        FinalLUT  ← Create_LUT1_Delay()
54.        Place(FinalLUT, FinalSite)
55.        Lock(FinalLUT)                       // is_bel_fixed / is_loc_fixed
56.        PlacementOK ← TRUE
57.      ELSE
58.        // not yet within tolerance → choose best site (max slack) and repeat
59.        k ← ArgMax(Slacks[1..n])             // Eq. (7): Target − Slacks[k] = MAX
60.        TempSite ← CandidateSites[k]
61.        TempLUT  ← Create_LUT1_Delay()
62.        Place(TempLUT, TempSite)
63.        Lock(TempLUT)
64.        // re-run from Step 1 to refine boundary w.r.t. updated layout
65.        PlacementOK ← FALSE
66.      END IF
67.
68.    UNTIL PlacementOK = TRUE
69.
```

```
70.        // go to next delay element
71.     END FOR
72.
73.     // 4) Finalization
74.     Lock_All_Async_Modules()            // set_property is_bel_fixed / is_loc_fixed
75.     Restore_And_Place_Remaining_Design()  // place_design
76.
77. END PROCEDURE
78.
79.
80. // --- Helper operations (conceptual) ---
81.
82. FUNCTION Dilate(A, B):
83.     // return { z | (B)_z ∩ A ≠ ∅ }
84. END FUNCTION
85.
86. FUNCTION Erode(A, B):
87.     // return { z | (B)_z ⊆ A }
88. END FUNCTION
89.
90. FUNCTION Create_LUT1_Delay():
91.     // create 1-LUT configured as delay cell
92. END FUNCTION
```

4.3 Computational Complexity

The computational efficiency of MDPC can be assessed by examining both its runtime behavior and auxiliary memory usage. For each delay element, the method evaluates S candidate boundary locations generated through morphological analysis, which incurs a cost of O(S). Because MDPC applies a place–evaluate–remove cycle to every candidate, the total runtime for *n* delay elements grows proportionally to O(n × S).

If the selected delay chain ultimately contains L LUT buffers, the complete computational cost becomes:

$$O(n \times S \times L) \quad (4.6)$$

and the corresponding auxiliary memory requirement is:

$$\mathrm{O}(\mathrm{S} + \mathrm{n} \times \mathrm{L}) \quad (4.7)$$

Both runtime and memory usage scale linearly with the number of delay elements, the number of boundary candidates, and the length of the resulting delay chain. This linear growth ensures that MDPC remains computationally practical even for large asynchronous pipelines, enabling efficient delay placement on modern FPGA platforms without compromising scalability or automation.

4.4 Comparative Evaluation with Existing Methods

To validate the effectiveness of the proposed MDPC method, a comparative evaluation was performed against conventional delay-placement strategies. The study considered an ECO-based single-pass flow using three physical placement patterns, Row Pattern (RP), Column Pattern (CP), and Row–Column Pattern (RCP), as well as an iterative synthesis flow configured with RP alignment. All approaches used the same timing-constraint set generated by the MDPC framework to ensure a fair and consistent comparison. The experiments employed the frequency-adaptive RISC-V pipeline described in Chapter 5, which represents the most demanding case because it contains the largest number of delay elements. The pipeline includes two selection-delay components with four and three channels, respectively, along with two fixed matched delays. This configuration provides a comprehensive assessment of both matched and selection-delay structures. The worst slack of each corresponding timing path was extracted, as these values determine the critical margins for setup correctness. The objective was to align these worst slack values as closely as possible with the predefined targets of 0.2 ns, 0.5 ns, 0.8 ns, and 1.0 ns.

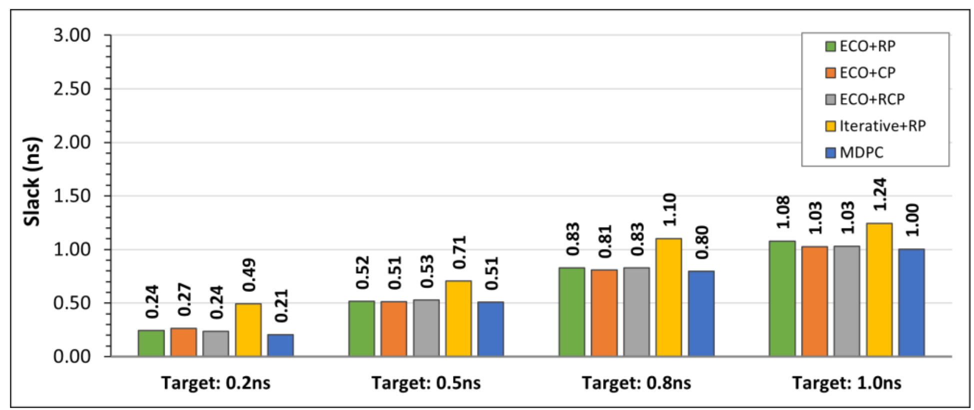


Figure 4.6. *Comparison of the average worst slack value of every timing path at different target slack values.*

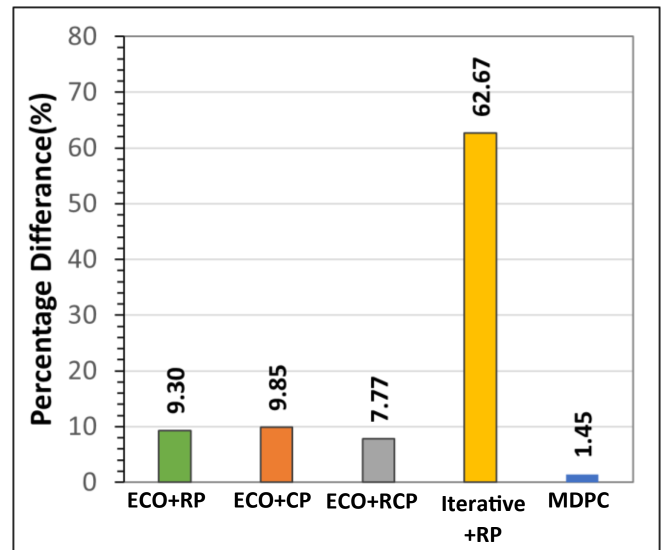


Figure 4.7. *Comparison of the worst slack value distributions at different target slack values.*

1. Slack Accuracy

   Table 4.6 highlights that MDPC provides significantly higher slack precision than conventional flows. MDPC consistently achieves worst-slack values close to the target margins, 0.206 ns, 0.508 ns, 0.797 ns, and 1.004 ns, while iterative delay placement shows much higher variation, with average values of 0.495 ns, 0.706 ns, 1.101 ns, and 1.245 ns. Overall, MDPC maintains a slack error of only 1.45%, whereas conventional approaches exhibit deviations of 62.67%. This high precision stems from MDPC's feedback-driven boundary exploration, in which each candidate boundary site is evaluated and inserted incrementally until the measured slack converges to the target. In contrast, iterative and ECO-based methods rely on coarse timing adjustments that often fail to converge reliably, leading to fluctuating and unpredictable timing outcomes.

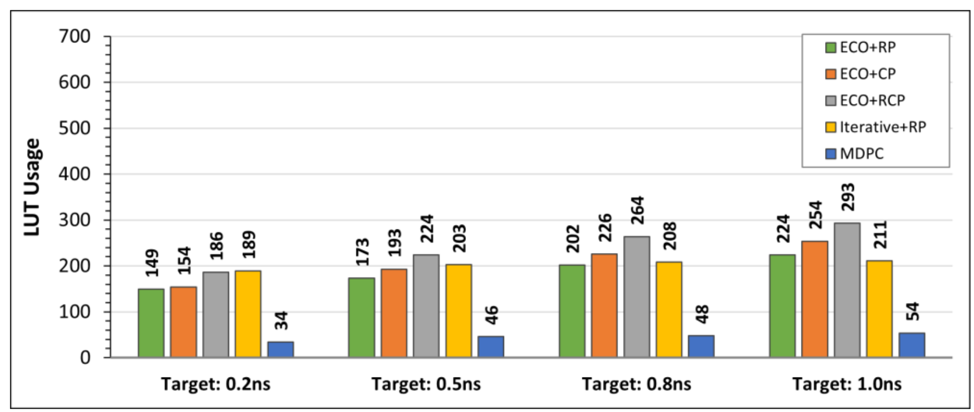


Figure 4.8. *Comparison of LUTs usage for inserting delay at different target slack values.*

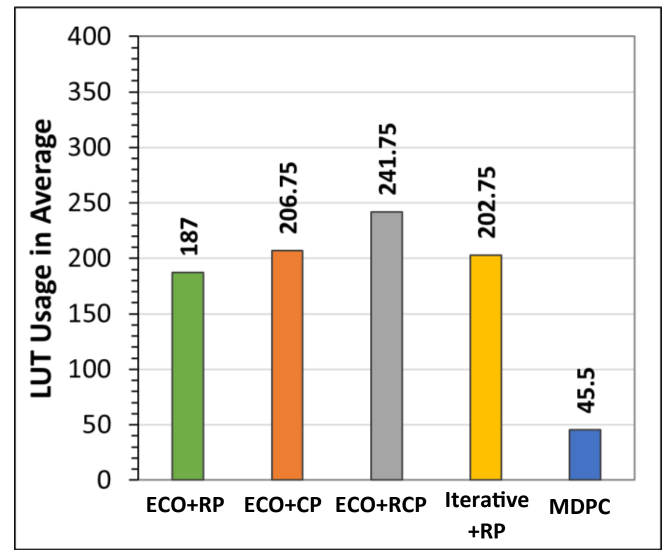


Figure 4.9. *Comparison of LUTs usage in average for inserting delay elements.*

2. Resource Utilization

    As shown in Table 4.8, MDPC offers the lowest LUT overhead among all evaluated methods. Morphological boundary detection restricts the search space to physically relevant regions, and the incremental slack-evaluation mechanism ensures that only the minimum number of delay elements is inserted. Across all experiments, MDPC reduces LUT consumption by approximately 3.76–5.56× compared with ECO-based and iterative flows. Among the alternative methods, ECO-RCP achieves moderately good accuracy but introduces the highest hardware cost due to redundant buffer insertions. By contrast, MDPC's selective placement strategy eliminates unnecessary logic and preserves compact routing structures.

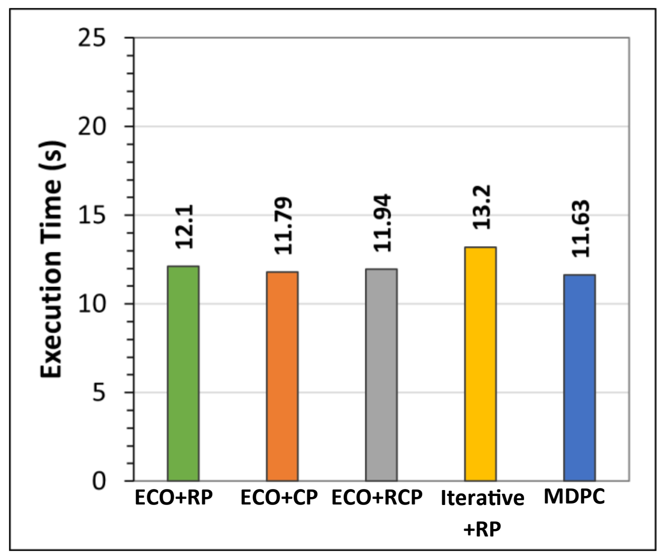


Figure 4.10. *Comparison of execution time based on 1500 CoreMark usage in average for inserting delay elements.*

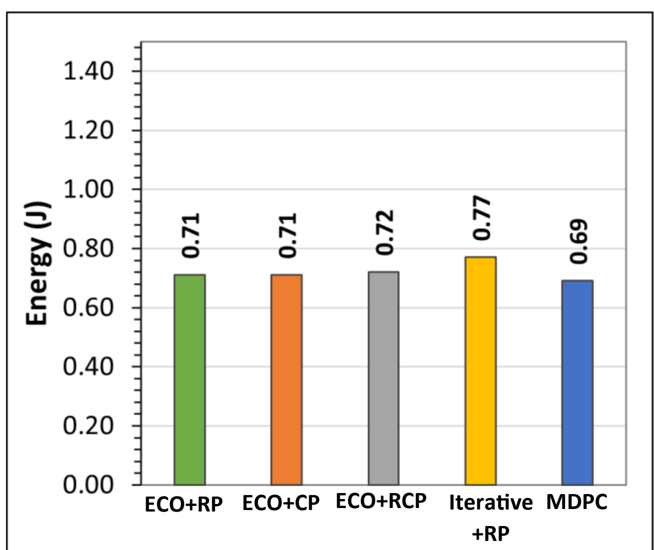


Figure 4.11. *Comparison of energy usage based on 1500 CoreMark usage in average for inserting delay elements.*

3. Performance and Energy Impact

Figures 4.10 and 4.11 summarize the effects of delay-placement quality on performance and energy. Designs produced using MDPC show consistent improvements with 1500 iterations of Coremark, with execution-time reductions of 1.38-13.50% and energy reductions of 2.89–11.59%. These gains arise from accurate slack alignment, which minimizes idle handshake cycles and shortens average token propagation time. Additionally, by avoiding excessive delay elements, MDPC reduces both dynamic switching activity and leakage power. Although some ECO-based approaches may achieve similar slack accuracy, their higher LUT usage results in greater power consumption. MDPC therefore provides a balanced improvement across timing accuracy, performance, and energy efficiency.

4. Existing Methods Comparison

Table 4.6 compares MDPC with established asynchronous design flows including PTC, ADM, BDPC, and the method described in [23]. All flows are compatible with standard EDA toolchains; however, only MDPC, BDPC, and the method in [23] align closely with synchronous design practices,

as they rely on post-layout STA-driven adjustment rather than RTL-level modifications. Regarding pipeline support, all flows can handle linear pipelines. Selective pipelines are supported by MDPC, PTC, BDPC, and the method in [23], but not by ADM. Only MDPC and the method in [23] support advanced styles such as frequency-adaptive, combined, and reduced pipelines, demonstrating greater scalability for performance-oriented designs.

In terms of timing closure, MDPC and BDPC require a single synthesis iteration, whereas PTC and ADM depend on iterative RTL refinement, and the method in [23] requires multiple passes due to selection-delay evaluation. MDPC provides automated delay insertion and completes the full design process for the frequency-adaptive pipeline in 18.6 minutes, whereas BDPC relies on manual ECO steps and PTC/ADM require repeated RTL updates. For delay identification, MDPC, BDPC, PTC, and the method in [23] use propagated-clock analysis, which extracts slack directly without assuming phase offsets between generated clocks. ADM, by contrast, employs fixed phase shifts that cannot represent multi-channel selection-delay behavior, limiting its applicability to advanced pipelines. Platform compatibility further distinguishes the flows. MDPC and PTC are optimized for FPGA implementations, whereas BDPC and ADM are primarily oriented toward ASIC flows. The method in [23] supports both FPGA and ASIC platforms, offering flexibility across implementation targets.

Overall, MDPC provides an efficient, FPGA-oriented design methodology that supports all pipeline styles, fully automates delay insertion, and achieves timing closure in a single synthesis iteration. Its tight integration with synchronous design practices and robust scalability make it a practical and effective solution for implementing modern asynchronous processors across diverse architectural configurations.

Table 4.6. *Comparison of Delay-Placement and Design-Flow Methods.*

Comparison Aspects		**MDPC**	**BDPC**	**PTC**	**ADM**	**[23]**
Supported by EDA tools		✓	✓	✓	✓	✓
Similarity with synchronous flow		High	High	Medium	Medium	High
Supported pipeline	**Linear**	✓	✓	✓	✓	✓
	Selective	✓	✓	✓	×	✓
	Frequency-Adaptive	✓	×	×	×	✓
	Combined	✓	×	×	×	✓
	Reduced	✓	×	×	×	✓
Delay length identification		NPCs	NPCs	NPCs	Phase shift	NPCs
Synthesis iteration		Single	Single	Multiple	Multiple	Multiple
Platform		FPGAs	ASIC	FPGAs	ASIC	ASIC/FPGAs
Delay placement		Automatic	Manual	Manual	Manual	Manual
Delay placement accuracy		High	Medium	Low	Low	Medium
Delay insertion time usage (min)		~18.6	NA	NA	NA	NA

Chapter 5
Experiment on RISC-V Processors

The evaluation of BD circuits requires a systematic methodology that links the proposed design flow to practical processor implementation. While earlier chapters introduced the Generated Propagation Clocks (GCP) and Morphological Delay-Placement Constraint (MDPC) techniques individually, their combined effectiveness must be demonstrated on a real processor to validate timing accuracy, resource efficiency, and performance scalability. The RISC-V architecture provides an ideal test platform for this purpose due to its modular instruction-set structure, availability of open-source toolchains, and suitability for FPGA-based prototyping.

Implementing asynchronous RISC-V processors on FPGA platforms demands tight coordination between handshake-based control, combinational data paths, and delay-placement constraints. In this study, multiple pipeline organizations, linear, selective, frequency-adaptive, reduced, and combined, are realized to explore the full design spectrum of asynchronous BD architectures. Each pipeline style exhibits distinct timing behavior: linear pipelines follow straightforward token progression; selective and frequency-adaptive pipelines incorporate multi-channel delay structures that vary with instruction latency; reduced pipelines streamline data paths to lower control complexity; and combined pipelines integrate diverse handshake structures within one processor. Mapping these pipelines onto FPGA fabric requires consistent placement of handshake controllers, insertion of matched or selection-delay elements, and careful isolation of timing domains to guarantee correct event sequencing. By applying GCP for timing-path reconstruction and MDPC for automated delay insertion, all pipeline variants can be implemented reproducibly, enabling direct comparison across styles under identical design conditions.

This chapter presents a comprehensive experimental study of asynchronous RISC-V processors implemented on FPGA platforms. Section 5.1 reviews the literature on pipeline styles in asynchronous processor design, establishing the architectural background for the pipelines evaluated in this work. Section 5.2 describes the adopted RISC-V processor architecture and its handshake-based organization. Section 5.3 details the implementation of five pipeline styles, including linear, selective, frequency-adaptive, combined, and reduced, and explains how each organization maps its control and data paths onto FPGA fabric. Section 5.4 outlines the benchmarking methodology used to evaluate timing accuracy, performance, energy behavior, and hardware efficiency. Section 5.5 presents the experimental results across all pipeline variants, highlighting differences in slack precision, execution time, energy consumption, and resource usage. Section 5.6 analyzes the architectural limitations and design trade-offs associated with each pipeline style, while Section 5.7 discusses how these processor organizations map to different application domains. Together, these sections demonstrate the feasibility, scalability, and practical benefits of the proposed asynchronous design methodologies across diverse RISC-V pipeline architectures.

5.1 Pipeline Style Literature Review

The development of asynchronous processors has produced a variety of pipeline organizations, each reflecting different trade-offs among performance, energy efficiency, and hardware cost. Early prototypes such as FAM [40], NSR [41], and TITAC [42], established the foundations of self-timed processing and demonstrated the viability of asynchronous execution on real hardware. The linear pipeline represents the most basic and widely adopted form. It transfers data sequentially through handshake-controlled stages, offering structural simplicity but enforcing worst-case operation latency across all instructions. Representative implementations include AMULET1–3 [43], [44], [45], ASYNMPU [46], TITAC-2 [47], MIPS R3000 [48], RISC-V [49], Lutonium [50], YUPPIE [51] , SAMIPS [52], and FPGA-based processors [14], [15], [16], [53], [54].

To improve energy efficiency, the selective pipeline introduces conditional bypassing to deactivate unused modules, reducing switching activity and dynamic power. This pipeline style has been adopted in several architectures, including asynchronous RISC-V [17], AEM32 [55], A8051 [56], RISC32-A [57], and superscalar RISC-V [58].The frequency-adaptive pipeline enhances performance by aligning handshake timing with instruction-dependent delays, allowing faster operations to complete earlier. This concept first appeared in MSP430 [22] and was later extended to vector-capable asynchronous RISC-V designs [23].The reduced pipeline simplifies control and shrinks area by eliminating infrequently used registers. Recent FPGA-based RISC-V [59] processors have demonstrated that this style can achieve a balanced combination of resource efficiency and energy reduction. Finally, the combined pipeline merges frequency-adaptive and selective techniques into a hybrid architecture that offers both timing flexibility and energy savings. This approach has been implemented in superscalar asynchronous RISC-V processors [23], where multi-channel delay selection and conditional bypass logic operate together to adapt dynamically to instruction behavior.

Although these works collectively illustrate the rich design space of asynchronous processors, cross-study comparisons remain difficult due to differences in benchmarks, tool flows, and implementation methodologies. Table 2 in the Appendix summarizes representative processors across multiple generations, highlighting the diversity of pipeline styles, handshake protocols, and hardware platforms considered in the comparative analysis of this chapter.

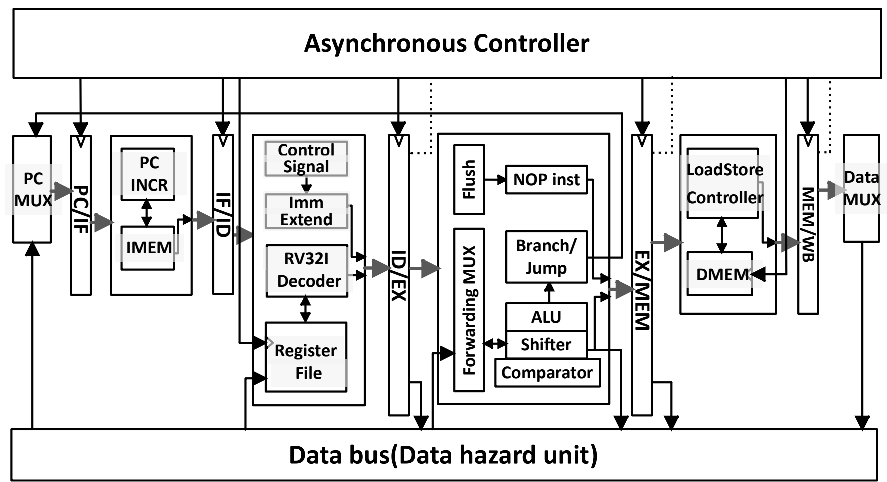


Figure 5.1. Architecture of an asynchronous RISC-V processor with a hazard unit and an asynchronous controller, which generates a fire signal to control register memory and register file.

5.2 RISC-V Processor Architecture

A conventional five-stage RISC-V pipeline, comprising Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write-Back (WB), is used as the architectural baseline. In the IF stage, instructions are fetched from the instruction memory (IMEM) using the program counter (PC). The ID stage decodes the instruction, generates the required control signals, and reads operands from the register file. The EX stage handles arithmetic and logical operations and computes memory addresses. The MEM stage performs load and store operations through the data memory (DMEM), and the WB stage writes the final result

back to the register file. The implementation in this study follows the official RISC-V specifications in [60] and [61].

In the asynchronous realization, the global clock is replaced by locally generated clocks derived from two-phase BD handshake controllers. Each stage is divided into a control path and a data path. The control path employs handshake elements, such as register, source, sink, MUX, and DEMUX controllers, to coordinate data transfers through request (Req) and acknowledge (Ack) signals. The data path contains the combinational logic blocks and flip-flops responsible for computation and storage. Matched and selection delay elements are inserted along the control path to align handshake transitions with data stabilization, ensuring safe sequencing and preventing premature latching. A hazard-detection unit resolves data dependencies by managing forwarding and stalling between interacting stages.

Once valid data and handshake conditions are satisfied, each asynchronous controller issues a local *fire* signal to update the registers, allowing pipeline stages to proceed at their natural pace. This modular, handshake-driven organization enables elastic execution without relying on a global clock. Figure 5.1 presents the overall architectural framework.

5.3 Pipeline-Style Implementations

This part implements multiple asynchronous RISC-V pipeline styles to demonstrate how different architectural organizations behave when mapped onto an FPGA using a unified self-timed processor framework. The evaluation covers linear, selective, sequential-ring, and iterative-ring pipelines, each constructed on the same RISC-V baseline while preserving their characteristic control structures and handshake interactions. By realizing these designs under identical conditions, the implementations reveal how pipeline topology, control intensity, and feedback structure shape the overall behavior of each self-timed processor variant.

5.3.1 Linear Pipeline

The linear pipeline represents the most fundamental and historically established asynchronous processor architecture. Data tokens propagate through a fixed sequence of handshake-controlled stages, with each stage initiating its operation only after receiving a valid request and issuing an acknowledgment upon completion. Matched delay elements are inserted along the request lines to satisfy setup timing constraints, ensuring that the control signal is delayed until the corresponding data are stable. In the asynchronous RISC-V implementation, register handshake controllers manage inter-stage communication, while source and sink controllers define pipeline boundaries. The logical clocks for the register file and DMEM are generated through dedicated handshake controllers, maintaining temporal alignment. Although conceptually simple, the linear pipeline is constrained by the worst-case delay of each stage, forcing all instructions to execute at the speed of the slowest operation. Consequently, it serves primarily as a baseline reference for evaluating more advanced pipelines. Figure 5.2 presents a linear pipeline structure and its operation timing.

5.3.2 Selective Pipeline

The selective pipeline introduces adaptivity by bypassing inactive stages based on instruction-dependent control signals. This approach prevents unnecessary switching activity, thereby improving energy efficiency. Two common variants exist: the stage-adaptive skipping controller, which merges handshake operations across skipped stages, and the conditional handshake controller, which employs MUX/DEMUX elements to redirect control flow dynamically. The present study adopts the conditional handshake controller because it is compatible with the two-phase BD protocol. In this organization, the MEM stage can be conditionally bypassed when no memory access is required. A DEMUX controller forwards data from the EX/MEM register either to the DMEM or directly to the register file, while a MUX controller determines the active control path. Matched delay elements are inserted on the logical clock path to maintain synchronization. When bypassing is active, the register write-back delay is shorter than that of memory access, yielding significant energy savings. However, the MUX/DEMUX handshake controllers introduce additional latency; thus, overall speedup depends on the frequency of stage bypass events. Figure 5.3 presents selective pipeline structure and its operation timing.

5.3.3 Frequency-Adaptive Pipeline

The frequency-adaptive pipeline enhances timing flexibility by assigning variable delay lengths according to instruction latency. Instead of forcing all operations to adhere to a single worst-case delay, each stage completes once its actual processing time elapses. This is accomplished using selection delay elements—multiplexed sets of matched delays, chosen dynamically based on the operation type. In the asynchronous RISC-V implementation, the EX and MEM stages exhibit the largest latency variations and therefore employ selection delays with multiple channels (four for the EX stage and three for the MEM stage). The IF and ID stages retain fixed matched delays for simplicity. Arithmetic operations that require longer propagation times select longer delay paths, whereas lightweight logical instructions follow shorter paths. This fine-grained adaptation allows the pipeline to approximate "better-than-worst-case" performance while maintaining handshake correctness. However, additional selection logic slightly increases area and control complexity. Figure 5.4 presents the frequency adaptive pipeline structure and its operation timing.

5.3.4 Combined Pipeline

The combined pipeline merges the benefits of selective and frequency-adaptive schemes. It integrates conditional stage bypassing with dynamic delay selection, allowing both energy and timing optimization. A DEMUX controller can bypass the MEM stage when it is idle, while the EX stage uses selection delay channels for instruction-dependent timing. This design achieves balanced improvements across performance and power without excessive complexity. Logical clocks for DMEM and the register file are generated through MUX/DEMUX controllers, as in the selective pipeline. Meanwhile, adaptive delay control in the EX stage follows the frequency-adaptive principle. Although the combined architecture introduces moderate control overhead, it offers steady overall gains in speed and energy efficiency. Figure 5.5 presents the combined pipeline structure and its operation timing.

5.3.5 Reduced Pipeline

The reduced pipeline simplifies the processor by removing rarely used registers, thereby lowering hardware cost and control complexity. In the asynchronous RISC-V processor, the MEM/WB register is eliminated, allowing results to flow directly from the EX/MEM register or DMEM to the register file. This reduces the number of storage elements and handshake controllers but extends the critical path, slightly reducing pipeline parallelism. To compensate for timing variations, selection delays are retained in the EX and MEM stages. A MUX ensures the correct routing of results depending on the instruction type. The logical clocks for the register file and DMEM are generated by a dedicated handshake controller, eliminating the need for additional matched delays. This configuration provides the best area efficiency and balanced EDAP performance among all evaluated styles. Figure 5.6 presents the reduced pipeline structure and its operation timing.

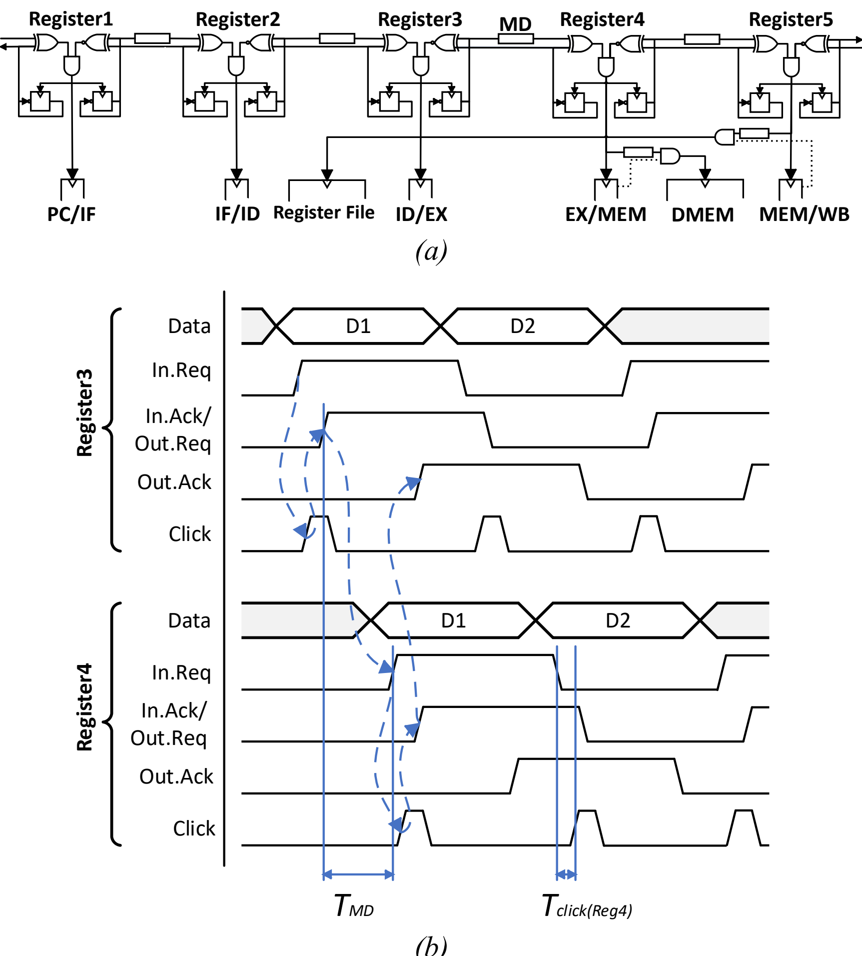
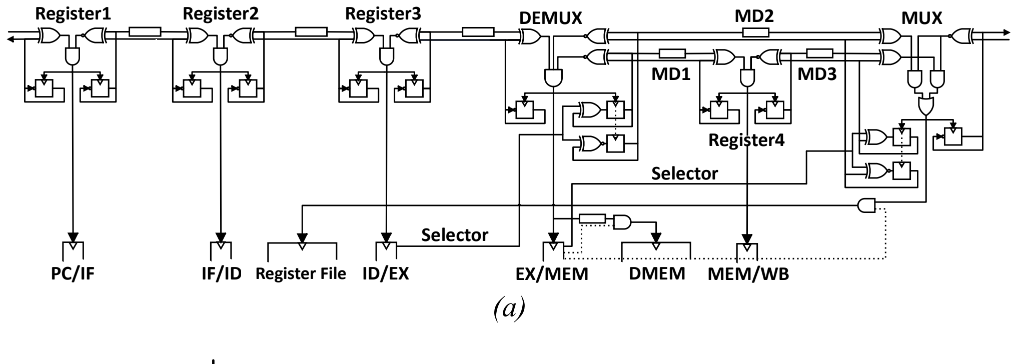


Figure 5.2. *Linear pipeline: (a) Pipeline structure, (b) Operational timing diagram between Register 3- and Register 4-handshake controllers. The setup constraint of the control path can be expressed as* $T_{MD}+T_{click(Reg4)}+T_{skew(clk\text{->}EX/MEM)}$.



(a)

(b)

Figure 5.3. *Selective pipeline: (a) Pipeline structure, (b) Operational timing diagram between DEMUX and MUX handshake controllers during register write-back and memory read operations. The setup constraint of DEMUX and Register 4 handshake controller can be expressed as* $T_{MD1}+T_{click(Reg4)}+T_{skew(clk\text{->}MEM/WB)}$. *The setup constraint of the DEMUX and MUX handshake controller can be expressed as* $T_{MD2}+T_{click(MUX)}+T_{skew(clk\text{->}Reg_File)}$. *The setup constraint of Register4 and MUX handshake controller can be expressed as* $T_{MD3}+T_{click(MUX)}+T_{skew(clk\text{->}Reg_File)}$.

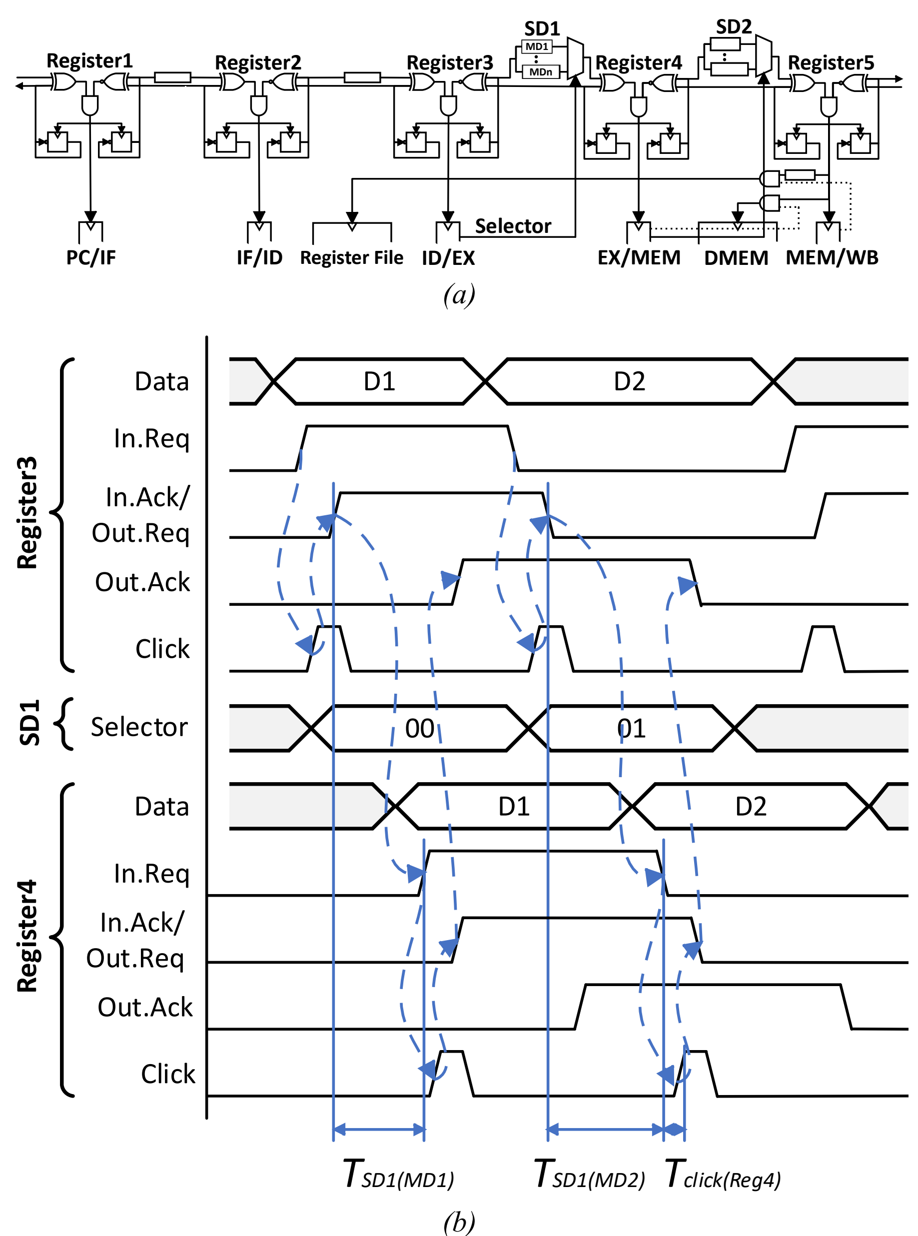


Figure 5.4. *Frequency-adaptive pipeline: (a) Pipeline structure, (b) Operational timing diagram of Register 3 and Register 4 handshake controllers. The setup constraint of the control paths that pass through DSU1 can be expressed as* $T_{SD1(MD(i))}+T_{click(Reg4)}+T_{skew(clk\text{->}EX/MEM)}$.

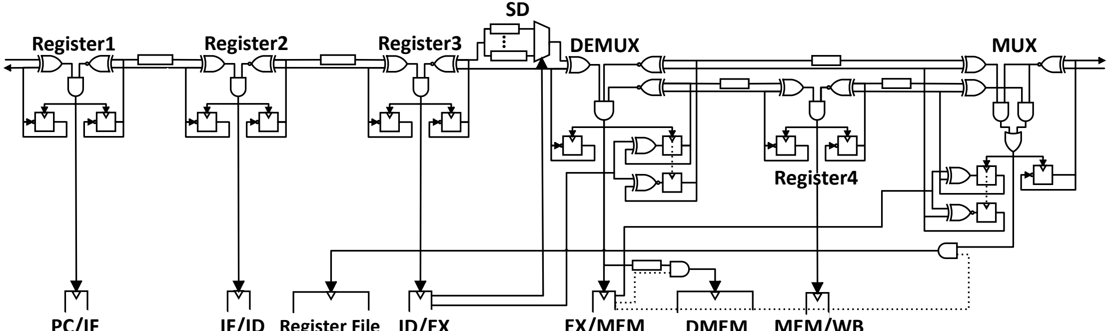


Figure 5.5. *Combined pipeline structure.*

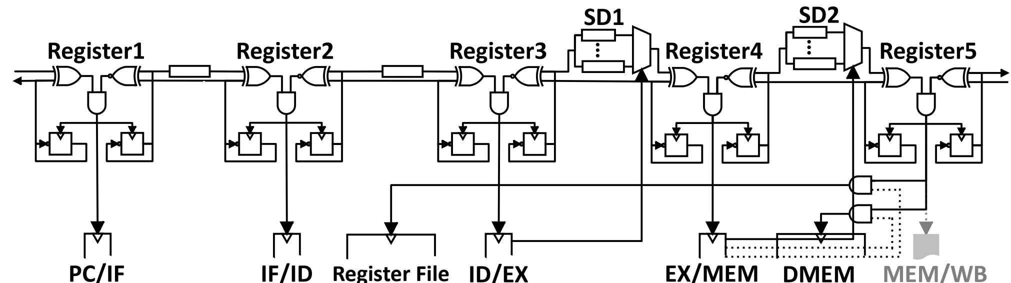


Figure 5.6. *Reduced pipeline structure.*

5.4 Benchmarking Methodology

In selecting benchmarks for this study, careful attention is given to balancing representativeness, feasibility, and comparability. Many earlier studies on asynchronous design relied on standard CPU benchmarks, such as Dhrystone and CoreMark, which are widely recognized and reported in MIPS or CoreMark scores. For example, Dhrystone was employed in the AMULET family and TITAC-2 to validate instruction throughput, thereby enabling direct comparison with synchronous systems. Although useful for evaluating raw speed, it mainly targets integer operations and lacks workload diversity. Similarly, SPEC2000 has been used in designs such as AEM32 to validate large-scale performance, but its complexity and resource demands make it impractical for FPGA-based prototyping. In addition to general-purpose tests, domain-specific workloads have been adopted to highlight the applicability of asynchronous processors in specialized contexts. For instance, AES encryption was evaluated in RISC32-A to measure the cryptographic performance of IoT applications, while SenseBench was applied on the MSP430 to assess energy efficiency in sensor-based workloads. Although these studies demonstrate application-level versatility, they do not capture the broad mix of control- and data-intensive behaviors required for fair architectural comparison.

To provide a realistic and industry-recognized performance reference, this study first evaluates the processor using CoreMark, a widely adopted embedded benchmark developed by the Embedded Microprocessor Benchmark Consortium (EEMBC). CoreMark generates a mixed-instruction workload consisting of control flow operations, arithmetic computations, memory accesses, and list processing, thereby reflecting practical embedded processing behavior. Unlike synthetic micro-kernels that isolate specific computational characteristics, CoreMark captures overall system-level performance under representative instruction diversity. Therefore, it serves as a comprehensive baseline for evaluating real-world execution behavior before analyzing pipeline-specific characteristics.

Before applying them, the RISC-V implementation was verified with the official riscv-arch-test suite to ensure ISA compliance [62].

5.5 Experimental Results

Five asynchronous RISC-V processors with different pipeline styles were implemented using the MDPC-based design flow and were compared with a synchronous baseline. The synchronous was set to 130 MHz,

which is the maximum stable frequency achieved under timing closure. Coremark with 1,500 iterations was used to measure the execution time, energy consumption, area utilization, and EDAP. By analyzing the results across these benchmarks, this section highlights the characteristic strengths and limitations of each pipeline style and discusses how different architectural organizations affect execution time, energy efficiency, and resource utilization. The EDAP, defined in Equations 5.1 and 5.2, is also employed as a composite metric to capture the overall efficiency of each design, thereby enabling a balanced comparison across pipeline styles.

$$EDAP=E\times D\times A \tag{5.1}$$

$$Area = \sqrt{\left(\frac{LUT}{LUT_{cap}}\right)\times\left(\frac{FF}{FF_{cap}}\right)} \tag{5.2}$$

where E denotes energy consumption, D denotes execution delay (execution time), and A denotes the FPGA area computed using Equation (11).

1. Performance evaluation

Every asynchronous pipeline demonstrated lower overall power consumption than the synchronous design. This reduction primarily originates from the elimination of the global clock tree and the adoption of a handshake-based control protocol. In synchronous architectures, a high-speed global clock generated by the Mixed-Mode Clock Manager (MMCM) continuously toggles regardless of data activity, contributing significantly to dynamic power consumption. In contrast, asynchronous pipelines activate only when valid tokens propagate through the stages, effectively suppressing unnecessary switching activity. As illustrated in Figure 5.7, the removal of clock-driven distribution networks, particularly the MMCM-related clock power, substantially reduces dynamic power overhead.

As shown in Figure 5.8, among the pipeline styles, the frequency-adaptive pipeline consistently delivered the highest performance gains, achieving speedups of 1.11-18.96%. These improvements stem from its ability to align stage delays with instruction latencies, thereby preventing simple operations from being constrained by the worst-case paths. However, this flexibility comes at the cost of additional selection logic, resulting in modest area overheads of 2.32–9.30%. Despite this, the pipeline maintained strong EDAP improvements of up to 176.46%.

The selective pipeline achieved the greatest energy savings, reducing energy consumption by 2.98–192.54% across the benchmarks. Its efficiency stems from bypassing inactive stages, which prevent unnecessary data transfers and suppress switching activity along both the data and control paths. With fewer toggling signals and less dynamic power wasted on unused modules, overall energy use is lowered while area utilization is also reduced by about 2.50–7.50%. Although selective control introduces small latency penalties in some workloads, the EDAP improvements remain significant and reach up to 154.05%.

The reduced pipeline provided the most balanced improvements. By eliminating seldom-used registers, it consistently lowered area utilization by 2.56–7.69% and improved EDAP up to 154.05%.

The combined pipeline offered moderate yet steady benefits by integrating stage bypassing with adaptive timing. It reduced energy consumption up to 184.06%, incurred minimal area overheads of 2.56–7.69%, and improved EDAP up to 154.05%. Although it did not outperform the selective or frequency-adaptive pipelines individually, it consistently provided improvements across multiple metrics.

By contrast, the linear pipeline consistently underperformed. Execution slowdowns by 2.94–13.24%, area overheads reached 13.64-28.21%, and EDAP up to 84.31%. These results confirm that the linear pipeline, although simple and historically important, is best retained as a baseline reference.

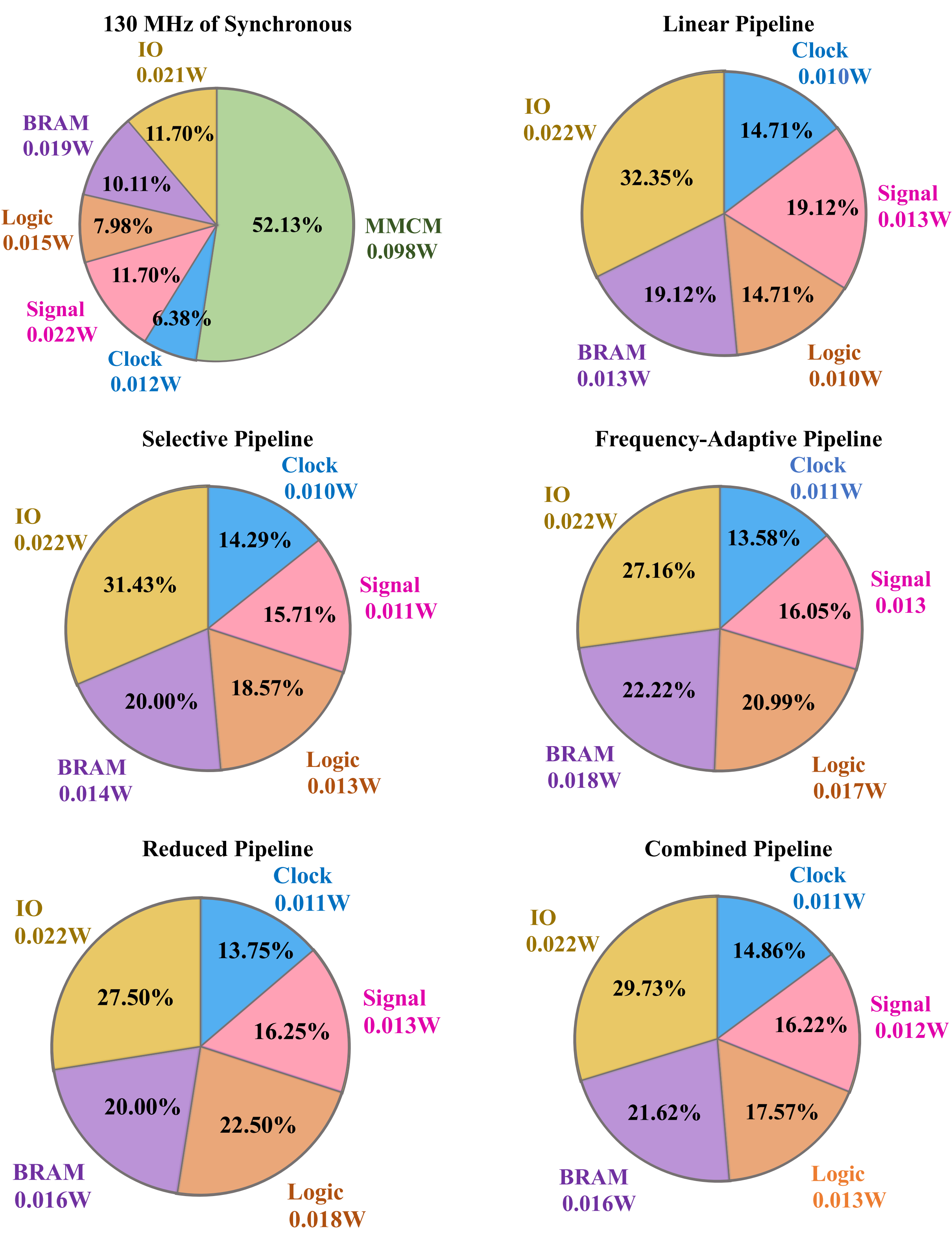


Figure 5.7. *Comparison of the power consumption of each power distribution usage based on 1500 CoreMark.*

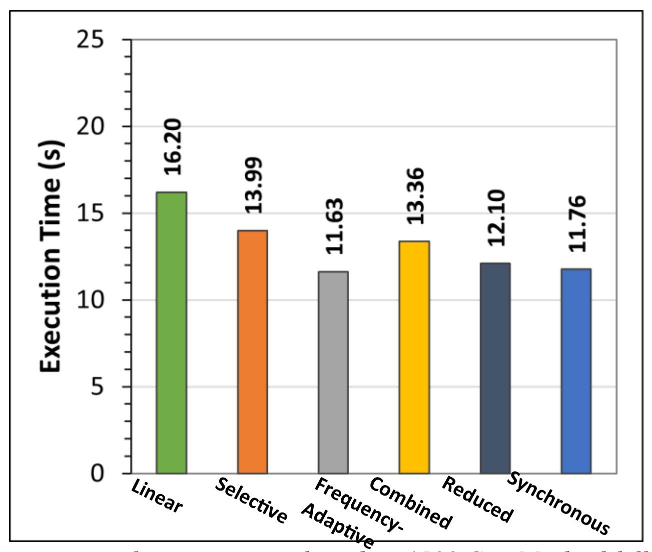


Figure 5.8. *Comparison of execution time based on 1500 CoreMark of different pipelines.*

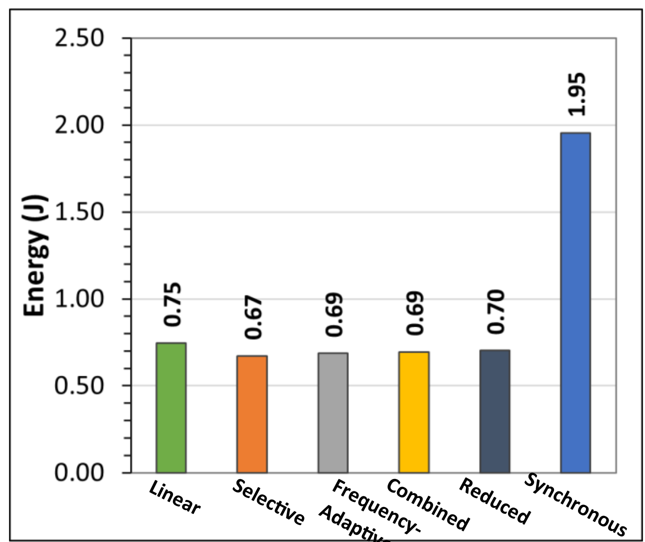


Figure 5.9. *Comparison of energy usage based on 1500 CoreMark of different pipelines.*

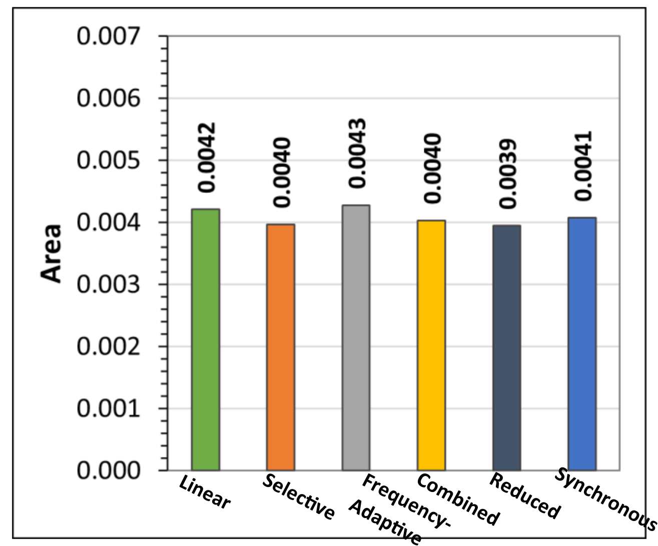


Figure 5.10. *Comparison of area usage based on 1500 CoreMark of different pipelines.*

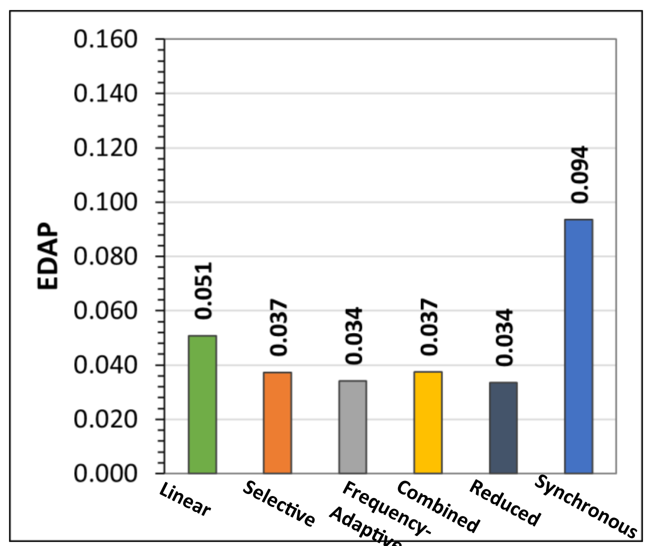


Figure 5.11. *Comparison of EDAP usage based on 1500 CoreMark of different pipelines.*

2. Computational behaviors

This study adopts a balanced set of lightweight kernels, including bubble sort, matrix multiplication, Fibonacci, and factorial, to provide representative and comparable workloads for processor evaluation. These benchmarks encompass a wide range of computational behaviors, thereby enabling a fair and meaningful comparison across different pipeline styles.

1. Bubble sort is highly control-intensive (CI). It stresses control flow, branch prediction, and decision logic, and has been widely used in both standard and superscalar RISC-V designs [17], [23], [58], [59].
2. Matrix multiplication is data-intensive (DI). It evaluates computational efficiency and data-path utilization under arithmetic-dominant workloads and has been adopted in MIPS [15], MSP430 [22], and RISC-V [58].
3. Fibonacci involves recursive dependency depth (RDD). It examines pipeline forwarding capability and latency tolerance in dependency-rich execution and has been employed in MIPS [14], [54] and RISC-V [58].

4. Factorial performs iterative regularity (IR) multiplication with simple, predictable control. It tests loop handling efficiency and register reuse in regular, low-control workloads and has been evaluated in superscalar RISC-V processors [58].

To further clarify the characteristic strengths of each pipeline style, Figure 5.14 presents a radar chart showing average normalized results with min–max scaling, where higher values indicate greater improvement in execution time, energy, area, and EDAP. This visualization highlights the complementary nature of asynchronous pipeline styles. The frequency-adaptive pipeline provides the strongest execution time advantage, while the selective pipeline achieves the greatest energy and EDAP benefits. The reduced pipeline excels in area efficiency with balanced EDAP, and the combined pipeline delivers steady all-around improvements. By contrast, the linear pipeline consistently lags behind.

To explain the benchmark-derived behavior of each pipeline style in greater detail, Figure 5.15 shows a radar chart of normalized EDAP across four benchmark-driven indicators. The selective pipeline stands out in CI and DI because it saves energy and area, giving it the highest overall EDAP efficiency for control- and data-intensive workloads. The reduced pipeline rivals the speed of the frequency-adaptive pipeline while using less hardware, making its EDAP especially strong for RDD and IR workloads. Although the frequency-adaptive pipeline achieves the fastest execution, its added control cost keeps its overall EDAP below that of selective and reduced pipelines. The combined pipeline provides moderate, steady benefits, while the linear pipeline and synchronous remain the weakest across all indicators. These findings emphasize that no single pipeline dominates every aspect, highlighting the need to match pipeline style with application requirements, from high-performance arithmetic workloads to energy-constrained systems.

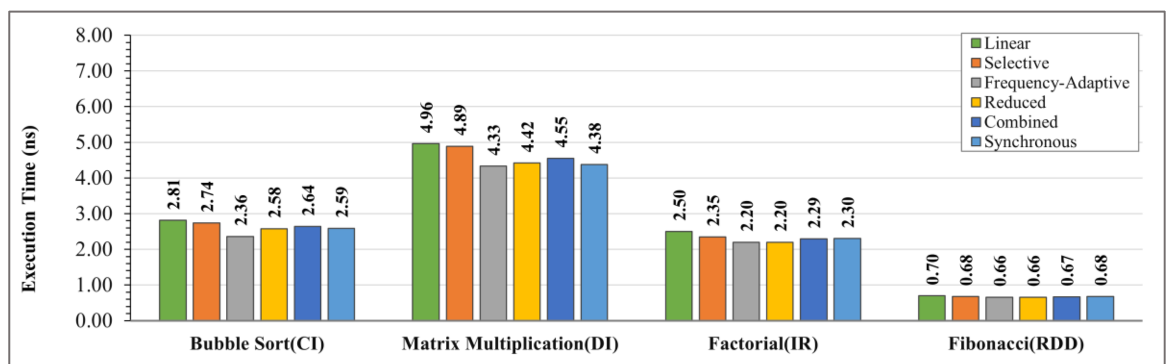


Figure 5.12. *Execution time comparison for different testbenches among pipeline styles.*

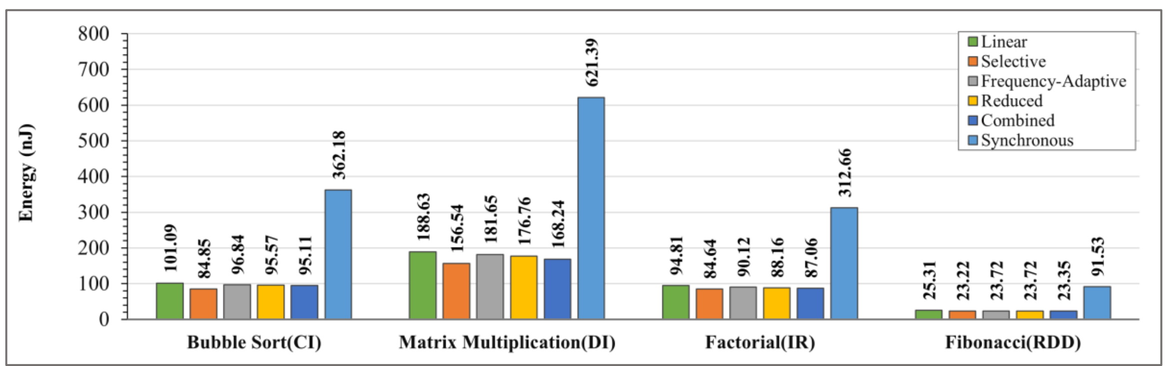


Figure 5.13. *Energy consumption comparison for different testbenches among pipeline styles.*

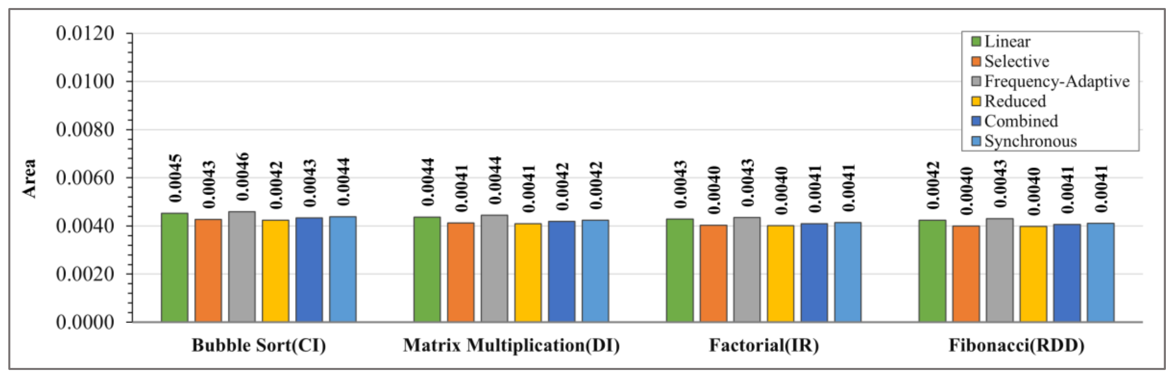


Table 5.14. *Area normalization comparison for different testbenches among pipeline styles.*

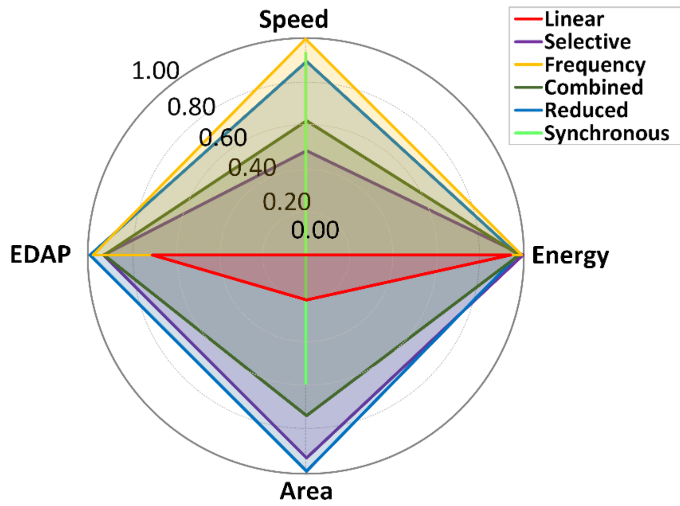


Figure 5.15. *Comparative radar chart of normalized metrics (execution time, energy, area, and EDAP) highlighting trade-offs among asynchronous pipeline styles.*

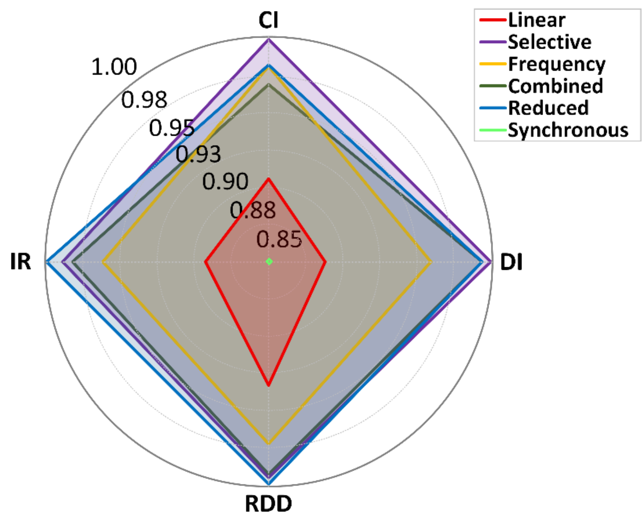


Figure 5.16. *Comparative radar chart of normalized EDAP with respect to computational behaviors of each benchmark, including control-intensive (CI), data-intensive (DI), recursion/dependency depth (RDD), and iteration regularity (IR), across different pipeline styles.*

5.6 Limitation Analysis of Pipeline Styles

Although asynchronous pipelines demonstrated clear advantages in execution time, energy efficiency, and EDAP across many workloads, several limitations emerged when they were compared with the synchronous baseline. These limitations are reflected not only in numerical results but also in architectural and implementation characteristics.

1. **Execution Slowdowns**

   Performance penalties in some asynchronous pipelines stem from structural bottlenecks. The linear pipeline relies on delay elements tuned to the worst-case paths, forcing all instructions to wait for the slowest operation, which can lead to slowdowns of up to 28.21%. The selective pipeline introduces additional latency from its MUX/DEMUX handshake controllers, resulting in penalties of approximately 16.87% when stage bypassing is infrequent. The conditional handshake controller splits into two paths. One path is shared for both memory read and memory write, and its timing is determined by the worst-case operation, typically the memory read. This constraint can limit overall memory access efficiency. Moreover, FPGA-based asynchronous designs often rely on conservative delay margins to guarantee correctness under process–voltage–temperature (PVT) variations, which can make them slower in practice, even when theoretical throughput is higher [63]. In contrast, synchronous designs take better advantage of EDA timing closure, allowing them to run closer to the true critical path [64].

2. **Area Costs**

   Area overheads are generally small but vary by pipeline style. The frequency-adaptive pipeline increases area by about 4.65% because it needs multiple delay channels and selector logic. The linear pipeline adds 2.38% due to duplicated handshake circuits. In contrast, the reduced pipeline saves 2.56% by removing seldom-used registers but adds some extra logic for selection delays to speed performance. Another factor is the use of the "DONT_TOUCH" attribute. Although necessary to preserve the integrity of handshake controllers and delay elements, it restricts optimization because the synthesis tool cannot merge, retime, or remove redundant, protected logic [65].

Overall, these limitations show that asynchronous pipelines are not always superior to synchronous designs on FPGAs. Their efficiency depends on architectural trade-offs, including tuning to worst-case delays, added control complexity, and longer critical paths can offset the benefits of self-timed operation. These findings highlight the importance of carefully matching pipeline styles with workload and platform characteristics.

5.7 Pipeline-to-Application Mapping

The diversity of asynchronous pipeline styles demonstrates that no single architecture can dominate across all application domains. Instead, each style addresses distinct design trade-offs and aligns with specific performance, power, and cost constraints. Table 5.5 summarizes their respective strengths, limitations, and representative use cases, providing practical guidance for selecting an appropriate style. The frequency-adaptive pipeline targets performance-oriented applications, such as DSP workloads, AI/ML accelerators, and edge servers, by exploiting average-case delays to improve throughput and reduce idle time between stages. The selective pipeline is designed for energy-constrained platforms, including IoT nodes and wearable devices. By bypassing inactive stages, it minimizes unnecessary switching activity, thereby lowering dynamic power and extending battery life. The reduced pipeline is well-suited for area- and cost-sensitive designs where hardware simplicity and minimal resource usage are critical. Typical examples include peripheral controllers, configurable IP blocks, and lightweight data-processing modules, where moderate but reliable performance is sufficient. The combined pipeline delivers moderate yet consistent improvements across energy, area, and performance, making it a practical choice for low-cost controllers and FPGA-based educational prototypes that need balanced trade-offs without complex optimization. Finally, the linear pipeline, though straightforward and easy to verify, functions best as a baseline for reference, teaching, and design verification rather than as a competitive option for modern high-efficiency applications.

Table 5.1. Pipeline-To-Application Design Map for Asynchronous RISC-V Pipelines.

Pipeline Styles	Strengths	Limitations	Best-Suited Applications
Frequency-Adaptive	• Achieves the highest speed performance • Adapts well to diverse workloads • Provides strong EDAP improvements	• Higher area overhead • Added control complexity • Energy efficiency may drop in computation-intensive tasks	High-throughput and performance-critical domains (arithmetic-heavy workloads, DSP pipelines, AI/ML accelerators, edge servers)
Selective	• Achieves strong energy savings • Reduces unnecessary switching • Provides area and EDAP improvements	• Extra overhead from MUX/DEMUX logic • Energy benefits depend on workload bypass • Limited speedup in some cases	Energy-sensitive applications (IoT, wearable devices, sensor nodes, battery-powered embedded systems)
Reduced	• Eliminates seldom-used registers • Simplifies control and reduces area • Provides balanced efficiency across metrics	• Longer critical path reduces overlap • Decreases pipeline parallelism • May limit peak performance	Mixed workloads requiring balanced trade-offs (peripheral controllers, general embedded processor, and lightweight data-processing modules)
Combined	• Integrates adaptive timing and selective bypass • Provides balanced trade-offs • Achieves steady EDAP improvements	• Inherits complexity from both styles • Moderate area overhead • Verification effort increases	Moderate balanced trade-offs (low-cost controllers and FPGA-based educational prototypes)
Linear	• Offers conceptual simplicity • Easy to implement and verify • Serves as baseline reference	• Constrained by worst-case delay • Often slower and less efficient • Larger area from handshake replication	Baseline/reference implementations, education, verification environments

Chapter 6

Discussion and Conclusion

This chapter synthesizes the overall findings of the research, discusses the technical and methodological implications of the proposed flow, and concludes with broader reflections on its significance and future directions. By integrating GCP and the MDPC, this work establishes the first automated and deterministic design methodology for asynchronous BD circuits on commercial FPGAs. The following sections discuss the impact of this framework in terms of flow determinism, design efficiency, quantitative behavior, scalability, and remaining challenges, before concluding with a summary of contributions and implications.

6.1 Discussion of Integration of GCP and MDPC

The combination of GCP and MDPC directly addresses the two central challenges in asynchronous FPGA design: timing-path extraction and delay placement. GCP transforms request–acknowledge transitions into hierarchical generated clocks, allowing standard STA engines to analyze setup and hold constraints, prune timing loops, and isolate meaningful timing paths across linear, sequential, and conditional handshake structures. MDPC extends this analytical layer by introducing a spatially guided placement technique that identifies boundary regions for delay elements, evaluates slack after hypothetical placements, and selects the site whose slack best satisfies the bundling constraint. Together, these mechanisms create a closed-loop flow in which GCP ensures timing correctness and MDPC guarantees physical realizability. This integration enables complete single-pass synthesis without manual constraint tuning, ECO adjustments, or iterative routing stabilization.

6.1.1 Design Efficiency and Determinism

Earlier methodologies such as ADM, PTC, and BDPC rely on RTL modifications or repeated synthesis cycles to rebalance delay elements. These iterative flows suffer from routing randomness and uncertain convergence, making timing closure dependent on designer experience rather than tool predictability. The proposed GCP + MDPC flow eliminates iteration entirely. GCP provides consistent path extraction because every handshake event is converted into a logically ordered generated clock. MDPC then places only the minimum required delay elements in positions that satisfy slack constraints with physical awareness. This results in deterministic timing reports, consistent layout behavior across implementation runs, and compatibility with native FPGA toolchains. Therefore, the flow replaces manual heuristics with a structured methodology that consistently converges in a single pass.

6.1.2 Quantitative Improvements and Resource Behavior

The experimental evaluation in Chapter 6 demonstrates that the proposed flow provides not only automation but also quantifiable improvements in precision, performance, and energy efficiency. Slack deviation is reduced to 1.45%, representing an approximately 10× improvement compared with BDPC- or PTC-based timing extraction. Because MDPC inserts delay elements only when necessary and chooses optimized spatial locations, the LUT usage dedicated to delay elements decreases by 3.76–5.56×, reducing the overhead typically associated with BD circuits. More accurate slack alignment leads to system-level improvements: execution speed increases by 1.38–13.50%, particularly in pipelines whose throughput depends on locally asynchronous conditions, such as frequency-adaptive designs. Energy consumption improves as well, with measured reductions of 2.89–11.59%, attributed to shorter handshake cycles and reduced routing overhead. These improvements confirm that the proposed methodology not only simplifies design but also enhances runtime efficiency.

6.1.3 Scalability and Pipeline Diversity

A major strength of the GCP + MDPC framework is its scalability across a wide range of asynchronous pipeline styles. Linear and selective pipelines benefit from low routing complexity and minimal control overhead, while frequency-adaptive and combined pipelines take advantage of the framework's ability to handle multi-path timing semantics and internal control feedback. Because MDPC evaluates each handshake controller's spatial boundary independently, the same placement mechanism applies regardless of whether a pipeline uses feed-forward, selective, ring-based, or adaptive timing structures. This scalability confirms that the framework is architecture-agnostic and can be extended beyond RISC-V processors, enabling its application to asynchronous dataflow engines, event-driven accelerators, and mixed-timing SoCs.

6.1.4 Limitations and Future Considerations

Although the proposed framework significantly advances asynchronous design automation, several limitations remain. First, FPGA routing granularity constrains fine-grained delay matching. While MDPC mitigates this by evaluating spatial boundaries, the discrete nature of LUT and routing delays still limits precision. Second, the methodology depends on the capabilities of vendor STA engines, which restrict direct modeling of extreme process–voltage–temperature conditions and limit user control over low-level delay estimation. Third, the current experiments focus on RISC-V pipelines; broader evaluations using dataflow accelerators, multi-clock SoCs, or dynamically reconfigurable systems could further validate generality. Future work should explore hybrid delay models that blend post-route characterization with analytical timing, portability to alternative FPGA platforms such as Intel or Lattice devices, and techniques for integrating partial reconfiguration to support adaptive asynchronous systems. Such convergence between timing disciplines could lead to new directions in low-power processor design, dynamic voltage–frequency scaling, and resilient computation under timing uncertainty. The approach may also influence educational and industrial design methodologies by encouraging the inclusion of asynchronous principles in mainstream EDA practice.

Despite its automation and robustness, several challenges remain. First, the granularity of FPGA routing resources limits the precision of fine-grained delay tuning. MDPC partially compensates for this limitation through boundary-based placement, but sub-LUT adjustments are still constrained. Second, the flow relies on existing vendor STA engines, which restricts direct control over delay modeling under extreme process, voltage, and temperature variations. Finally, while the current validation focuses on RISC-V pipelines, further evaluation with dataflow accelerators or mixed-timing SoCs could better explore the framework's generality. Future work should consider hybrid delay modeling at the post-route level, cross-platform portability (e.g., Intel or Lattice FPGAs), and partial reconfiguration support for adaptive asynchronous systems.

6.2 Conclusion

This dissertation successfully fulfills its goal of establishing a unified, FPGA-compatible methodology for implementing asynchronous bundled-data (BD) circuits, bridging the conceptual gap between self-timed design principles and mainstream FPGA workflows. By integrating accurate timing analysis with physically grounded delay placement, the proposed framework achieves deterministic synthesis behavior, reproducible timing closure, and practical applicability across multiple asynchronous processor architectures. The accomplishment of each research objective is summarized below.

First, the work presents a complete design flow for asynchronous BD circuits that operates entirely within commercial FPGA tools, specifically Xilinx Vivado. The flow requires no custom timing analyzers, external scripts, or modifications to vendor tools. Instead, it systematically leverages existing synthesis, placement, routing, and static timing analysis capabilities, enabling asynchronous designs to be realized using the same infrastructure as synchronous circuits. This ensures portability, repeatability, and compatibility with standard EDA practices.

Second, the dissertation establishes the GCP method as a precise and FPGA-friendly timing-analysis paradigm. By interpreting handshake events as virtual clock domains, GCP allows Vivado's STA engine to evaluate setup and hold margins automatically, even in circuits with feedback loops, conditional paths, or multi-channel selection delay structures. This resolves long-standing timing-analysis challenges in asynchronous systems and provides consistent timing reports without manual tuning.

Third, the MDPC method fulfills the objective of achieving single-pass, resource-efficient delay insertion. Through boundary detection and slack-guided refinement, MDPC identifies physically meaningful delay locations and minimizes the number of LUT-based delay elements required to satisfy the bundling constraint. Experimental results demonstrate substantial improvements in timing accuracy, LUT efficiency, and convergence determinism, indicating that MDPC overcomes the unpredictability and labor-intensiveness of iterative ECO-based flows.

Fourth, the applicability of the proposed flow is validated through the implementation of five asynchronous RISC-V pipeline styles: linear, selective, frequency-adaptive, reduced, and combined. All designs achieve functional correctness, consistent timing closure, and robust behavior across different handshake topologies. These implementations not only confirm the generality of the framework but also show that complex asynchronous architectures can be developed without specialized tools or manual intervention.

Finally, the experimental evaluation provides a comprehensive analysis of performance, energy, and area trade-offs among the asynchronous pipeline styles. The proposed flow enables accurate measurement of architectural behavior, revealing improvements in execution speed, energy efficiency, and delay-element utilization over conventional asynchronous methodologies. The insights gained offer practical guidance for selecting suitable pipeline styles for different classes of applications, from low-power embedded systems to adaptive data-processing workloads.

Overall, this work establishes an automated and deterministic foundation for designing asynchronous BD circuits on modern FPGA platforms. By achieving all research objectives, the dissertation demonstrates that self-timed processors and adaptive computing architectures can now be implemented, analyzed, and optimized within standard toolchains, opening new possibilities for scalable, energy-efficient, and timing-robust digital systems.

6.3 Future works

Future research can extend the proposed asynchronous design framework in several directions. First, the granularity limits of FPGA routing resources indicate a need for more precise delay management. While MDPC reduces sensitivity to placement variation, sub-LUT delay tuning remains restricted by the discrete nature of commercial FPGA fabrics. Future work may explore hybrid delay models that combine analytical timing with post-route characterization, enabling finer control over the bundling constraint under diverse routing patterns and PVT conditions.

Second, the current design flow is validated primarily on Xilinx devices. Extending the GCP and MDPC methodologies to alternative FPGA platforms, such as Intel and Lattice architectures, would improve portability and expose platform-dependent timing characteristics that may require refined constraint generation or modified placement heuristics. In addition, integrating partial reconfiguration could support adaptive asynchronous systems whose timing requirements change dynamically, allowing delay elements or local controllers to be reconfigured at runtime.

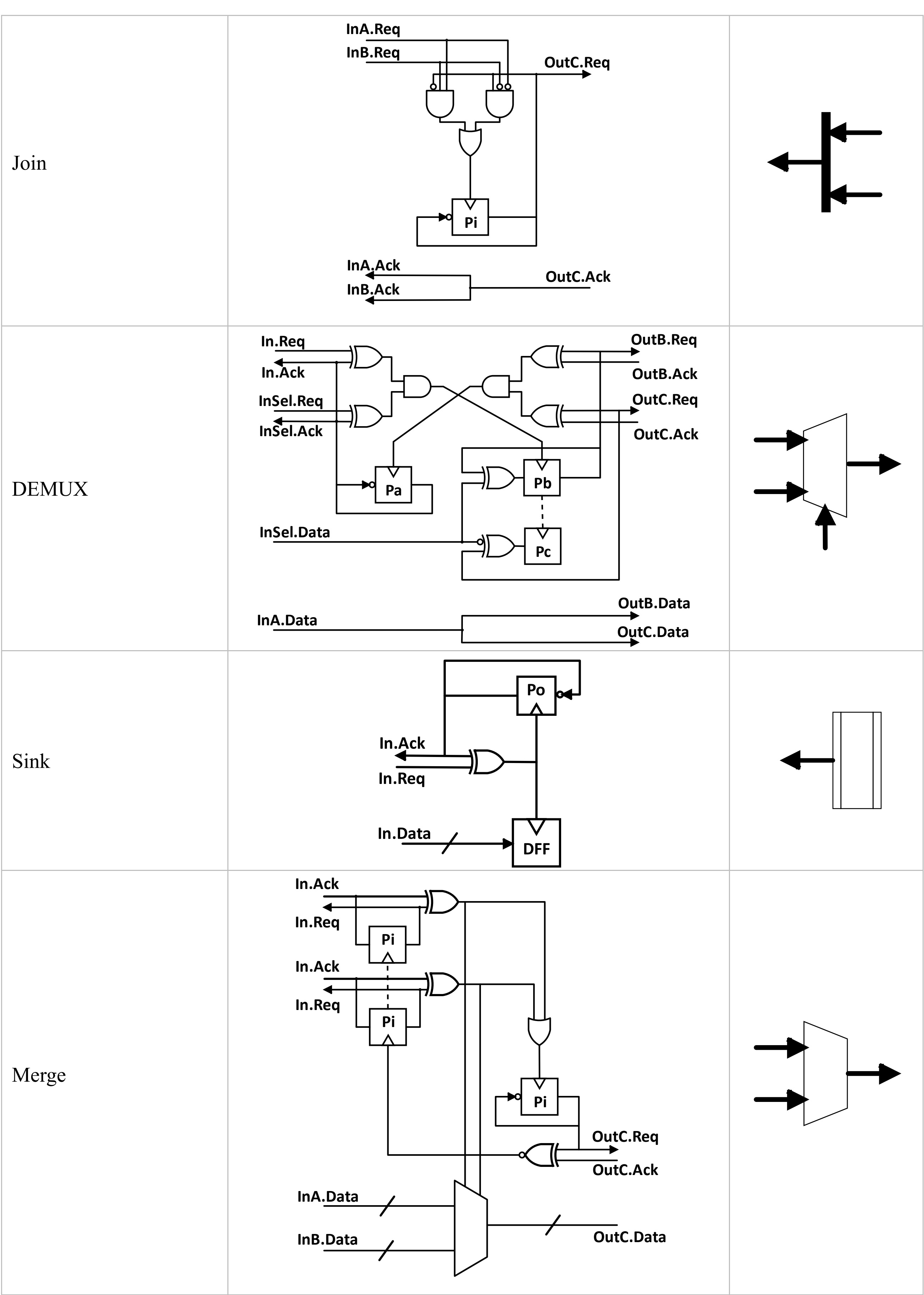
Third, broader architectural evaluations are needed to further demonstrate generality. Although the methodology successfully handles multiple RISC-V pipeline styles, future studies may investigate dataflow accelerators, event-driven computational kernels, and mixed-timing SoCs that combine asynchronous and synchronous subsystems. Such designs may reveal new opportunities for applying GCP in hierarchical or multi-clock environments and for extending MDPC to scenarios involving clustered or distributed delay channels.

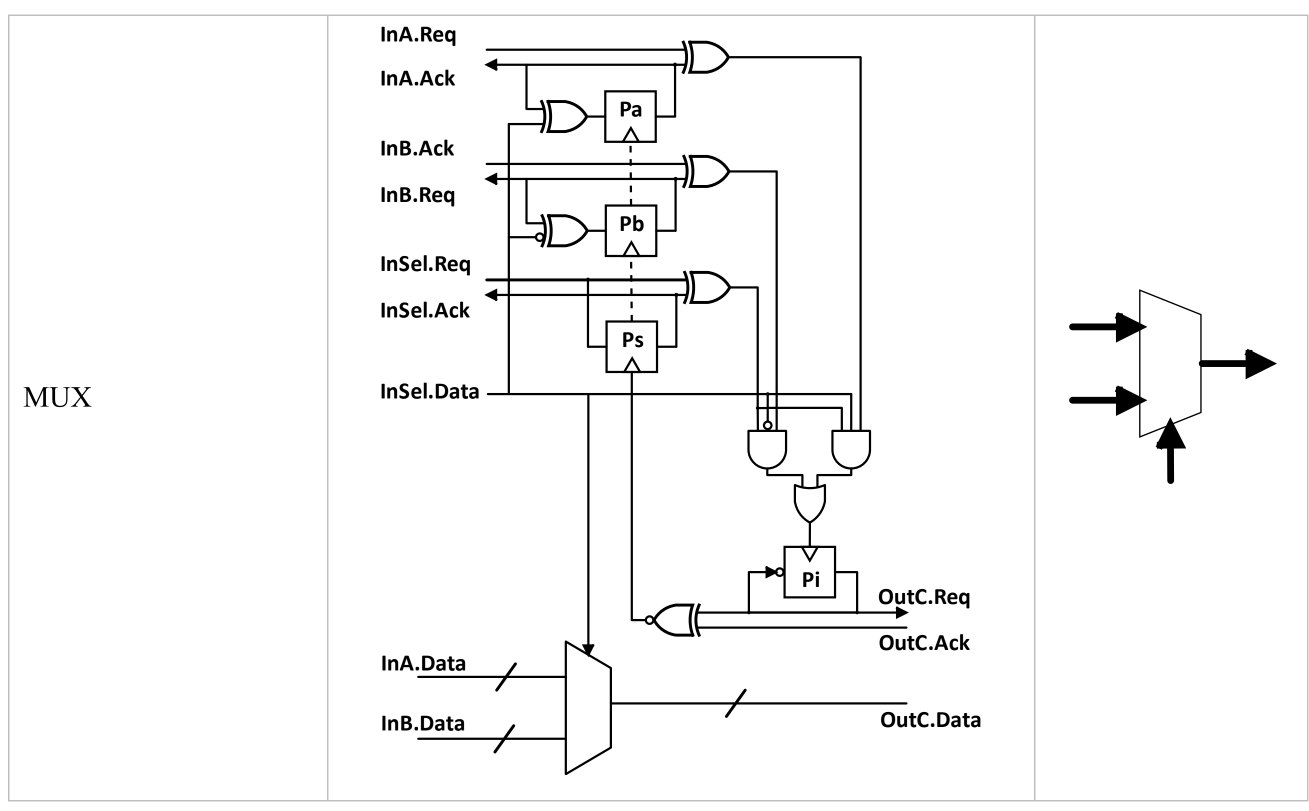
Finally, as I plan to continue my academic career as a faculty member at Kasetsart University, future research will expand toward advanced hardware system design, particularly in digital circuits and AI-oriented architectures. Building upon the asynchronous timing and FPGA prototyping expertise developed in this work, future efforts may focus on low-power AI accelerators, edge-computing hardware, and digital circuit design methodologies suitable for machine learning and intelligent systems. Integrating asynchronous principles with AI accelerator architectures could provide energy-efficient and workload-adaptive computation platforms. This direction not only advances academic research in digital hardware and AI systems but also supports practical engineering education and hardware innovation in Thailand.

Appendix

Table A.1. *Common handshake components based on two-phase click element, their structures, and symbols, including of Source, Register, Fork, Join, DEMUX, Sink, MUX, and Merge.*

Components	Structure	Data Flow
Source	Po, Out.Req, Out.Ack, DFF, Out.Data	
Handshake Register	In.Req, In.Ack, Out.Req, Out.Ack, Pi, Po, Click, In.Data, DFF, Out.Data	
Fork	OutB.Req, In.Req, OutC.Req, OutB.Ack, OutC.Ack, InA.Ack, Pi	


Join
InA.Req
InB.Req
OutC.Req
Pi
InA.Ack
InB.Ack
OutC.Ack
DEMUX
In.Req
In.Ack
InSel.Req
InSel.Ack
OutB.Req
OutB.Ack
OutC.Req
OutC.Ack
Pa
Pb
Pc
InSel.Data
InA.Data
OutB.Data
OutC.Data
Sink
Po
In.Ack
In.Req
In.Data
DFF
Merge
In.Ack
In.Req
Pi
In.Ack
In.Req
Pi
Pi
OutC.Req
OutC.Ack
InA.Data
InB.Data
OutC.Data


MUX
InA.Req
InA.Ack
Pa
InB.Ack
InB.Req
Pb
InSel.Req
InSel.Ack
Ps
InSel.Data
Pi
OutC.Req
OutC.Ack
InA.Data
InB.Data
OutC.Data

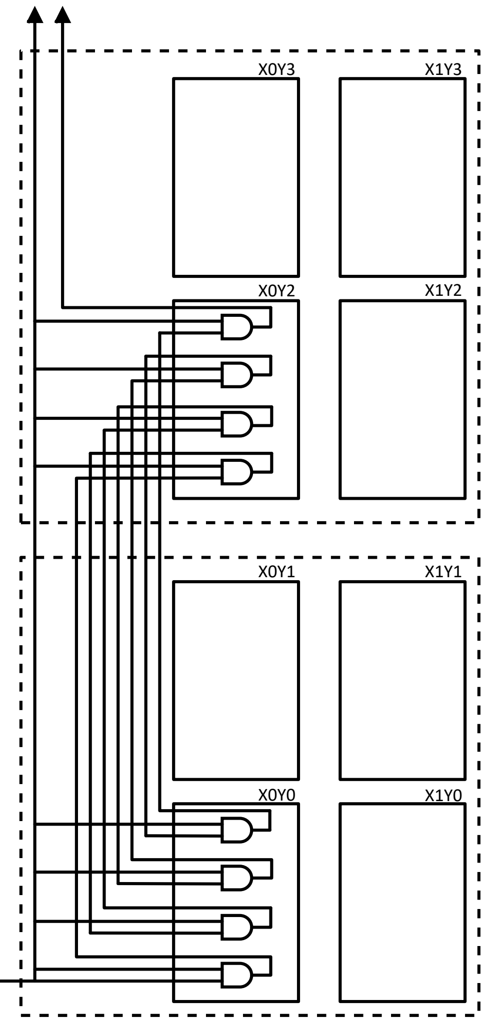


Figure A.1. *The structure of the delay element aligns with the column pattern.*

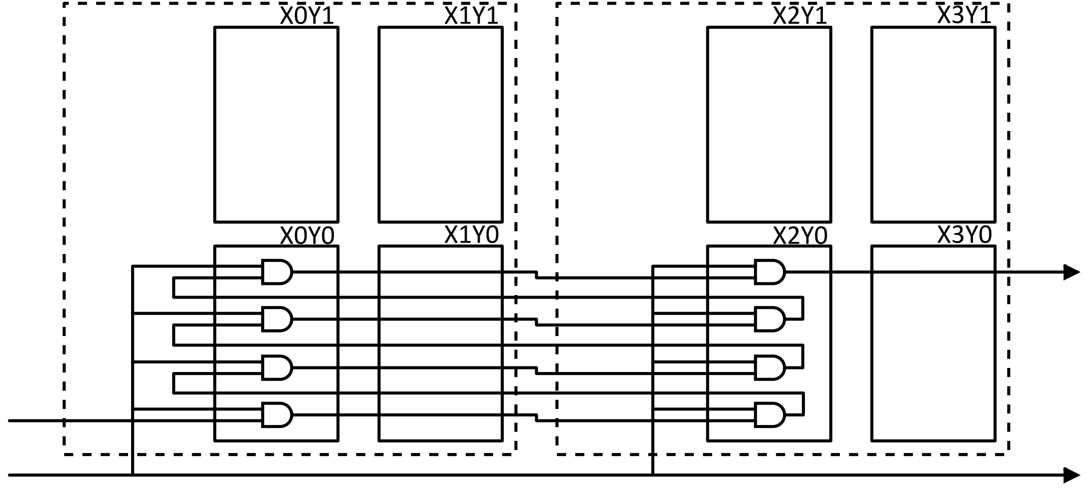


Figure A.2*. The structure of the delay element aligns with the row pattern.*

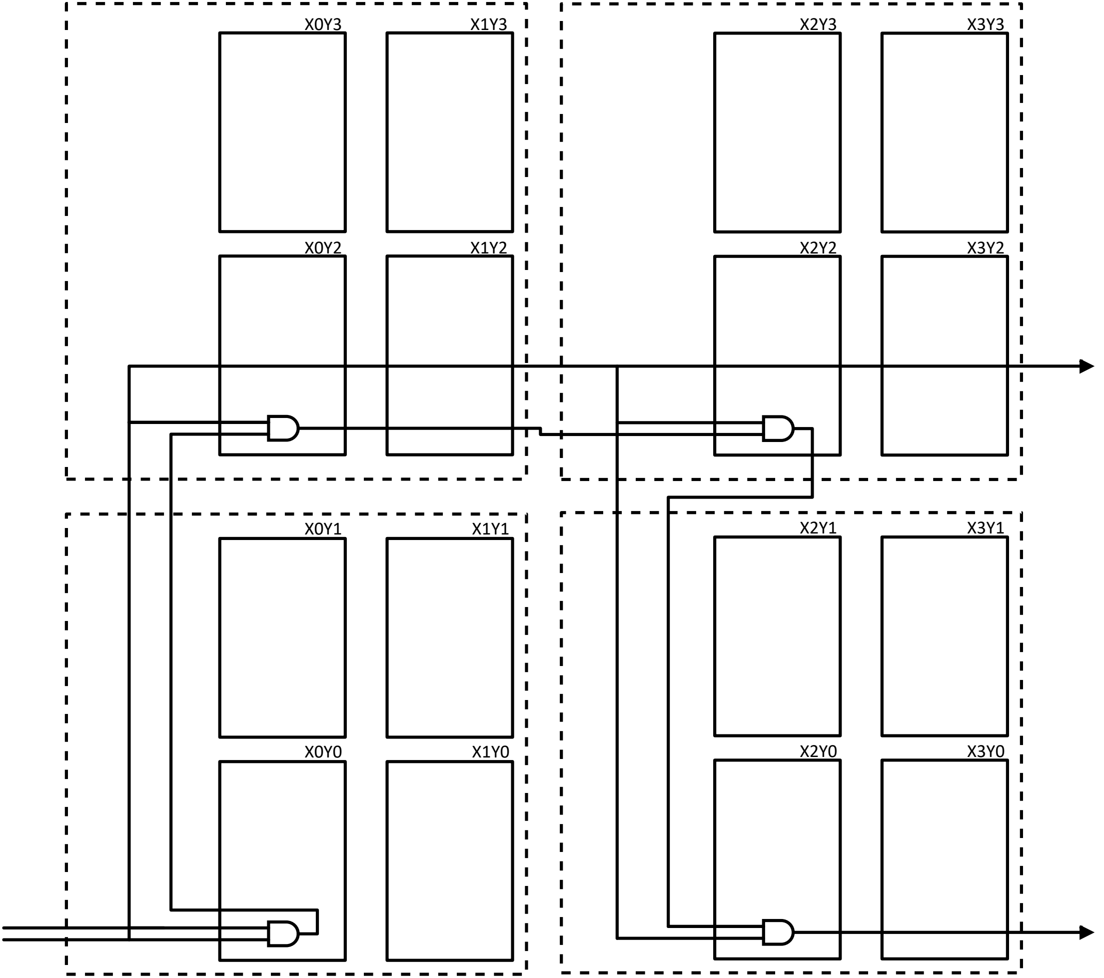


Figure A.3. *The structure of the delay element aligns with the row-column pattern.*

Table A.2. Summary of Representative Asynchronous Processor Implementations.

Processors	Year	Instructions	Pipeline stages	Pipeline Styles	Protocols	Platforms	Improvement (speed, power, resource usage) and Benchmark
FAM [40]	1992	32-bitRISC	4	Linear, Reduced	Four-phase BD	0.5 um CMOS	Imp: 300 MIPS, 71000 transistors
NSR [41]	1993	16-bit RISC	5	Linear	Two-phase BD	Actel FPGA 0.8 µm	Imp: 1.3 MIPS
TITAC [42]	1994	8-bit von Neumann	0	No pipeline	QDI	1 um CMOS	Imp: 11.2 MIPS, 212 mW, 22068 transistors
AMULET1 [43] (ARM6)	1997	32-bit RISC	6	Linear	Two-phase BD	GEC Plessey 1 µm CMOS	Imp: 20.5 kDhry, 152mW,58374 transistors Benchmark: Dhrystone
						GEC Plessey 0.7 µm CMOS	Imp: 40 kDhry, 58374 transistors Benchmark: Dhrystone
ASYNMPU [46]	1997	8/16-bit CISC	2-6	Linear	Two-phase BD	DLM 0.6 µm CMOS	Imp: 33MHz, 110 mW, 31750 transistors
TITAC-2 [47] (MIPS)	1997	32-bit RISC	5	Linear	QDI for the processor pipeline	0.5 µm CMOS	Imp: 52.3 MIPS, 2.11 W, 496,367 transistors Benchmark: Dhrystone
					Two-phase BD for cache memory		
MIPS R3000 [48]	1997	32-bit RISC	3	Linear	QDI	Hewlett-Packard 0.6 µm CMOS	Imp: 280 MIPS, 3.3V, 7W Imp: 150 MIPS, 2.0V, 1W
AMULET2e [44] (ARM7)	1999	32-bit RISC	6	Linear	Two-phase BD	0.5 µm CMOS	Imp: 42 MIPS, 94 mW, 454,000 transistors Benchmark: Dhrystone
AMULET3 [45] (ARM9)	2000	32-bit RISC	6	Linear	Two-phase BD	0.35 µm CMOS	Imp: 100 MIPS, 155mW, 113,000 transistors Benchmark: Dhrystone
Lutonium [50] (8051)	2003	8-bit CISC	11	Linear	QDI	TSMC 0.18 µm CMOS	Imp: 200 MIPS, 100 mW
YUPPIE [51]	2006	8-bit CISC	2	Linear	Two-phase BD	130 nm CMOS	Imp: 170 MIPS, 15 pJ
AEM32 [55] (ARM9)	2008	32-bit RISC	8	Selective	Four-phase BD	0.35 µm CMOS	Imp: 365 MIPS, 165mW, 125,000 transistors Benchmark: SPEC2000
A8051 [56]	2008	8-bit CISC	5	Selective	Four-phase BD	Hynix 0.35 µm CMOS	Imp: 84.2 MIPS, 36.3mW, 110,000 transistors Benchmark: Dhrystone

MIPS [14]	2012	32-bit RISC	5	Linear	Four-phase BD	Virtex-5 FPGA 65 nm	Imp: 53.2 MHz, 1316 LUTs, 2466 FFs
MIPS [15]	2014	32-bit RISC	5	Linear	Four-phase BD	Virtex-5 FPGA 65 nm	Imp: 140.8 MHz, 39.92 mW Benchmark: Fibonacci
						Spartan-6 FPGA 45 nm	Imp: 75.7 MHz, 15.85 mW Benchmark: Fibonacci
MIPS [16]	2016	32-bit RISC	5	Linear	Two-phase BD	Altera Cyclone IV FPGA 65 nm	Imp: 0.821us, 58.59mW, 2203 LUTs, 1319 FFs Benchmark: Match multiplication
							Imp: 63.487us, 74.89mW, 2203 LUTs, 1319 FFs Benchmark: Matrix multiplication
MIPS [53]	2017	32-bit RISC	6	Linear	Two-phase BD	Xilinx ZC706 FPGA 28 nm	Imp: 53.26 MIPS, 173.4 mW Benchmark: Fibonacci
MSP430 [22]	2017	16-bit RISC	2	Frequency-adaptive	Two-phase BD	IBM 65 nm CMOS	Imp: Avg. 15.417 us, 2477.8 pJ µW Benchmark: Math and Matrix multiplication
RISC-V [17]	2021	32-bit RISC	5	Selective	Two-phase BD	ZCU-102 FPGA 16 nm	Imp: Avg. 2.286 ns, 47 mW, 2544 LUTs, 1849 FFs Benchmark: Bubble sort, GCD, Arch-Test
RISC-V [49]	2021	32-bit RISC	6	Linear	Two-phase BD	TSMC 65 nm CMOS	Imp: 182.1 CoreMark, 4.6 mW Benchmark: CoreMark
MSP430 [54]	2022	16-bit RISC	5	Linear	Four-phase BD	Spartan3 FPGA 90 nm	Imp: 17.6 MHz, 59.79 mW Benchmark: SenseBench
SAMIPS [52]	2024	32-bit RISC	5	Linear	Four-phase BD	SGST 0.18 µm	Imp: Avg. 21.48 MIPS, 2142.32 MIPS/W, 169,634 transistors Benchmark: Dhrystone, Quicksort, Heapsort, ExcTest
Superscalar RISC-V [58]	2024	32-bit RISC	7	Selective	Two-phase BD	TSMC 65 nm CMOS	Imp: Avg. 250 MHz, 119.6 mW, 40,000 transistors Benchmark: Bubble, Matrix multiplication, Fibonacci, factorial, BGE
Superscalar RISC-V [23]	2024	32-bit RISC	7	Selective Frequency-adaptive	Two-phase BD	TSMC 65 nm CMOS	Imp: Avg. 250 MHz, 85 uw/MHz Benchmark: Bubble sort, val_mul, val_add, val_dot, val_cross

						ZCU-102 FPGA 16 nm	Imp: Avg. 120 MHz, 81mW, 12797 LUTs, 6366 FFs benchmark: Bubble sort, vector dot
RISC32-A [57]	2024	32-bit RISC	5	Selective	Two-phase BD	Artix-7 FPGA 28 nm	Imp: 300 ms, 0.232 J, 9545 LUTs, 7310 FFs Benchmark: Poling-based AES-encrypted
							Imp: 534 ms, 0.086 J, 9545 LUTs, 7310 FFs, benchmark: Interrupt-based AES-encrypted data
RISC-V [59]	2025	32-bit RISC	4	Selective, Reduced	Two-phase BD	Zynq7020 FPGA 28 nm	Imp: 13 mW, 1502 LUTs, 1276 FFs benchmark: Bubble sort, GCD, Fibonacci

Table A.3. *Comparison of the worst slack value distributions at different target slack values.*

Target slack	Results	ECO+RP	ECO+CP	ECO+RCP	Iterative+RP	MDPC
0.2	Average obtained slack (ns)	0.244	0.266	0.238	0.495	0.206
	Percentage Error	4.55%	7.45%	4.55%	21.01%	1.22%
0.5	Average obtained slack (ns)	0.517	0.514	0.528	0.706	0.508
	Percentage Error	0.98%	0.50%	1.46%	8.53%	0.50%
0.8	Average obtained slack (ns)	0.831	0.807	0.829	1.101	0.797
	Percentage Error	3.79%	0.91%	3.52%	31.66%	0.32%
1.0	Average obtained slack (ns)	1.078	1.025	1.028	1.245	1.004
	Percentage Error	7.52%	2.51%	2.80%	21.81%	0.39%

Table A.4. *Comparison of LUTs usage for inserting delay at different target slack values.*

Target slack	Methods	ECO+RP	ECO+CP	ECO+RCP	Iterative+RP	MDPC
0.2	LUT usage	149	154	186	189	34
	Improvement of MDPC (X)	4.38X	4.53X	5.47X	5.56X	-
0.5	LUT usage	173	193	224	203	46
	Improvement of MDPC (X)	3.76X	4.20X	4.87X	4.41X	-
0.8	LUT usage	202	226	264	208	48
	Improvement of MDPC (X)	4.21X	4.71X	5.50X	4.33X	-
1.0	LUT usage	224	254	293	211	54
	Improvement of MDPC (X)	4.15X	4.70X	5.43X	3.91X	-

Table A.5. *Comparison of the execution speed of RISC-V based on an adaptive-frequency pipeline by inserting delay with different methods.*

Programs	Methods	ECO+RP	ECO+CP	ECO+RCP	Iterative+RP	MDPC
Bubble Sort	Execution Time (us)	2.43	2.41	2.41	2.77	2.36
	Improvement of MDPC (%)	2.97%	2.12%	2.12%	17.37%	-
Matrix Multiplication	Execution Time (us)	4.46	4.41	4.41	5.07	4.33
	Improvement of MDPC (%)	3.00%	1.85%	1.85%	17.09%	-

Factorial	Execution Time (us)	2.27	2.24	2.24	2.58	2.20
	Improvement of MDPC (%)	3.18%	1.82%	1.82%	17.27%	-
CoreMark	Execution Time (s)	12.10	11.79	11.94	13.20	11.63
	Improvement of MDPC (%)	4.04%	1.38%	2.67%	13.50%	-

Table A.6. *Comparison of the energy consumption of RISC-V based on an adaptive-frequency pipeline by inserting delay with different methods.*

Programs	Methods	ECO+RP	ECO+CP	ECO+RCP	Iterative+RP	MDPC
Bubble Sort	Energy (nJ)	99.79	98.69	98.65	107.93	96.84
	Improvement of MDPC (%)	3.05%	1.91%	1.87%	11.45%	-
Matrix Multiplication	Energy (nJ)	191.61	189.50	189.42	202.80	181.65
	Improvement of MDPC (%)	5.48%	4.32%	4.28%	11.64%	-
Factorial	Energy (nJ)	92.87	91.84	91.80	100.46	90.12
	Improvement of MDPC (%)	3.05%	1.91%	1.86%	11.47%	-
Fibonacci	Energy (nJ)	24.44	24.16	24.16	27.02	23.72
	Improvement of MDPC (%)	3.03%	1.82%	1.82%	13.89%	-
CoreMark	Energy (n)	0.71	0.71	0.72	0.77	0.69
	Improvement of MDPC (%)	2.90%	2.90%	4.35%	11.59%	-

Table A.7. *Execution time comparison for different testbenches among pipeline styles.*

Programs	Results	Linear	Selective	Frequency	Combined	Reduced	Sync
Bubble Sort	Execution Time (us)	2.81	2.74	2.36	2.64	2.58	2.59
	Improvement over Sync (%)	-8.49%	-5.79%	8.88%	-1.93%	0.39%	-
Matrix Multiplication	Execution Time (us)	4.96	4.89	4.33	4.55	4.42	4.38
	Improvement over Sync (%)	-13.23%	-11.64%	1.14%	-3.88%	-0.91%	-
Factorial	Execution Time (us)	2.50	2.35	2.20	2.29	2.20	2.30
	Improvement over Sync (%)	-8.70%	-2.17%	4.35%	0.43%	4.35%	-
Fibonacci	Execution Time (ns)	0.70	0.68	0.66	0.67	0.66	0.68

	Improvement over Sync (%)	-2.94%	0.00%	2.94%	1.47%	2.94%	-
CoreMark	Execution Time (s)	16.20	13.99	11.63	13.36	12.10	11.76
	Improvement over Sync (%)	-37.76%	-18.96%	1.11%	-13.61%	-2.89%	-

Table A.8. *Energy consumption comparison for different testbenches among pipeline styles.*

Programs	Results	Linear	Selective	Frequency	Combined	Reduced	Sync
Bubble Sort	Energy (nJ)	101.09	84.85	96.84	95.11	95.57	362.18
	Improvement over Sync (%)	72.09%	76.57%	73.26%	73.74%	73.61%	-
Matrix Multiplication	Energy (nJ)	188.63	156.54	181.65	168.24	176.76	621.39
	Improvement over Sync (%)	69.64%	74.81%	70.77%	72.93%	71.55%	-
Factorial	Energy (nJ)	94.81	84.64	90.12	87.06	88.16	312.66
	Improvement over Sync (%)	69.68%	72.93%	71.18%	72.16%	71.80%	-
Fibonacci	Energy (nJ)	25.31	23.22	23.72	23.35	23.72	91.53
	Improvement over Sync (%)	72.35%	74.63%	74.08%	74.49%	74.08%	-
CoreMark	Energy (n)	0.75	0.67	0.69	0.69	0.70	1.95
	Improvement over Sync (%)	61.54%	65.64%	64.62%	64.62%	64.10%	-

Table A.9. *Area normalization comparison for different testbenches among pipeline styles.*

Programs	Results	Linear	Selective	Frequency	Combined	Reduced	Sync
Bubble Sort	Area	0.0045	0.0043	0.0046	0.0043	0.0042	0.0044
	Improvement over Sync (%)	-2.27%	2.27%	-4.55%	2.27%	4.55%	-
Matrix Multiplication	Area	0.0044	0.0041	0.0044	0.0042	0.0041	0.0042
	Improvement over Sync (%)	-4.76%	2.38%	-4.76%	0.00%	2.38%	-
Factorial	Area	0.0043	0.0040	0.0043	0.0041	0.0040	0.0041
	Improvement over Sync (%)	-4.88%	2.44%	-4.88%	0.00%	2.44%	-
Fibonacci	Area	0.0042	0.0040	0.0043	0.0041	0.0040	0.0041
	Improvement over Sync (%)	-2.44%	2.44%	-4.88%	0.00%	2.44%	-

CoreMark	Area	0.0042	0.0040	0.0043	0.0040	0.0039	0.0041
	Improvement over Sync (%)	-2.44%	2.44%	-4.88%	2.44%	4.88%	-

Table A.10. *EDAP comparison for different testbenches among pipeline styles.*

Programs	Results	Linear	Selective	Frequency	Combined	Reduced	Sync
Bubble Sort	EDAP	1.29	0.99	1.05	1.09	1.05	4.11
	Improvement over Sync (%)	68.73%	75.86%	74.48%	73.56%	74.55%	-
Matrix Multiplication	EDAP	4.09	3.16	3.49	3.20	3.20	11.53
	Improvement over Sync (%)	64.50%	72.62%	69.70%	72.20%	72.20%	-
Factorial	EDAP	1.01	0.80	0.86	0.82	0.78	2.98
	Improvement over Sync (%)	65.94%	73.06%	71.05%	72.57%	73.87%	-
Fibonacci	EDAP	0.075	0.063	0.067	0.064	0.062	0.255
	Improvement over Sync (%)	70.58%	75.29%	73.58%	75.12%	75.63%	-
CoreMark	EDAP	0.051	0.037	0.034	0.037	0.034	0.0940
	Improvement over Sync (%)	45.74%	60.64%	63.83%	60.64%	63.83%	-

References

[1] H. Saito, N. Hamada, N. Jindapetch, T. Yoneda, C. Myers, and T. Nanya, "Scheduling Methods for Asynchronous Circuits with Bundled-Data Implementations Based on the Approximation of Start Times," *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences,* vol. E90A, 2007, doi: 10.1093/ietfec/e90-a.12.2790.

[2] N. Hamada, Y. Shiga, T. Konishi, H. Saito, T. Yoneda, C. Myers, and T. Nanya, "A Behavioral Synthesis System for Asynchronous Circuits with Bundled-data Implementation," *IPSJ Transactions on System and LSI Design Methodology,* vol. 2, pp. 64-79, 2009, doi: 10.2197/ipsjtsldm.2.64.

[3] Y. Zhou, "Investigation of asynchronous pipeline circuits based on bundled-data encoding: Implementation styles, behavioral modeling, and timing analysis," *Tsinghua Science and Technology,* vol. 27, no. 3, pp. 559-580, 2022, doi: 10.26599/TST.2021.9010089.

[4] G. Miorandi, M. Balboni, S. M. Nowick, and D. Bertozzi, "Accurate Assessment of Bundled-Data Asynchronous NoCs Enabled by a Predictable and Efficient Hierarchical Synthesis Flow," in *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2017, pp. 10-17, doi: 10.1109/ASYNC.2017.22.

[5] J. V. Manoranjan and K. S. Stevens, "Qualifying Relative Timing Constraints for Asynchronous Circuits," in *2016 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2016, pp. 91-98, doi: 10.1109/ASYNC.2016.23.

[6] I. E. Sutherland, "Micropipelines," *Commun. ACM,* vol. 32, no. 6, pp. 720–738, 1989, doi: 10.1145/63526.63532.

[7] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A Designer's Guide to Asynchronous VLSI*. Cambridge, U.K.: Cambridge University Press, 2010, pp. 7-9.

[8] J. Zhang, H. Wu, W. Chen, S. Wei, and H. Chen, "Design and Tool Flow of a Reconfigurable Asynchronous Neural Network Accelerator," *Tsinghua Science and Technology,* vol. 26, no. 5, pp. 565-573, 2021, doi: 10.26599/TST.2020.9010048.

[9] A. Peeters, F. t. Beest, M. d. Wit, and W. Mallon, "Click Elements: An Implementation Style for Data-Driven Compilation," in *2010 IEEE Symposium on Asynchronous Circuits and Systems*, 2010, pp. 3-14, doi: 10.1109/ASYNC.2010.11.

[10] A. Mardari, Z. Jelčicová, and J. Sparsø, "Design and FPGA-implementation of Asynchronous Circuits Using Two-Phase Handshaking," in *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2019, pp. 9-18, doi: 10.1109/ASYNC.2019.00010.

[11] J. Sparsø, *Introduction to Asynchronous Circuit Design*. DTU Compute, Technical University of Denmark, 2020.

[12] G. Heck, L. S. Heck, A. Singhvi, M. T. Moreira, P. A. Beerel, and N. L. V. Calazans, "Analysis and Optimization of Programmable Delay Elements for 2-Phase Bundled-Data Circuits," in *28th International Conference on VLSI Design*, 2015, pp. 321-326, doi: 10.1109/VLSID.2015.60.

[13] J. N. Lassen, "FPGA Prototyping of Asynchronous Networks-on-Chip," Technical University of Denmark, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, DK-2800 Kgs. Lyngby, Denmark, 2008. [Online]. Available: http://www.compute.dtu.dk/English.aspx

[14] J.-G. Lee and M.-H. Oh, "Asynchronous Circuit Designs on an FPGA for Targeting a Power/Energy Efficient SoC," *IEICE Transactions on Electronics,* vol. E97.C, no. 4, pp. 253-263, 2014, doi: 10.1587/transele.E97.C.253.

[15] M. N. a. H. S. Jukiya Furushima, "Design of an Asynchronous Processor with Bundled-data Implementation on a Commercial Field Programmable Gate Array," *Informatica,* vol. 40, pp. 399–408, 2016.

[16] Z. Shin and M. H. Oh, "Design and Implementation of Asynchronous Processor on FPGA," *IEEE Access,* vol. 10, pp. 118370-118379, 2022, doi: 10.1109/ACCESS.2022.3220660.

[17] Z. Li, Y. Huang, L. Tian, R. Zhu, S. Xiao, and Z. Yu, "A Low-Power Asynchronous RISC-V Processor With Propagated Timing Constraints Method," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 68, no. 9, pp. 3153-3157, 2021, doi: 10.1109/TCSII.2021.3100524.

[18] H. Wu, Z. Su, J. Zhang, S. Wei, Z. Wang, and H. Chen, "A Design Flow for Click-Based Asynchronous Circuits Design With Conventional EDA Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 40, no. 11, pp. 2421-2425, 2021, doi: 10.1109/TCAD.2020.3038337.

[19] S. Yoshikawa, S. Sannomiya, M. Iwata, A. Sato, and H. Nishikawa, "EDA-oriented FPGA Circuit Design Method for Four-phase Bundled-data Circular Self-timed Pipeline," *Journal of Information Processing,* vol. 31, pp. 495-508, 2023, doi: 10.2197/ipsjjip.31.495.

[20] T. Otake and H. Saito, "A Design Method for Designing Asynchronous Circuits on Commercial FPGAs Using Placement Constraints," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,* vol. E103.A, no. 12, pp. 1427-1436, 2020, doi: 10.1587/transfun.2020VLP0006.

[21] A. Motaqi, M. Helaoui, S. AghliMoghaddam, and M. R. Mosavi, "Detailed implementation of asynchronous circuits on commercial FPGAs," *Analog Integrated Circuits and Signal Processing,* vol. 103, no. 3, pp. 375-389, 2020, doi: 10.1007/s10470-020-01602-3.

[22] D. Bhadra and K. S. Stevens, "Design of a low power, relative timing based asynchronous MSP430 microprocessor," in *Design, Automation & Test in Europe Conference & Exhibition*, 2017, pp. 794-799, doi: 10.23919/DATE.2017.7927097.

[23] L. Zhou, S. Xiao, H. Wang, J. Wang, Z. Xu, B. Wang, and Z. Yu, "Better-Than-Worst-Case: A Frequency Adaptation Asynchronous RISC-V Core With Vector Extension," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 32, no. 6, pp. 1045-1057, 2024, doi: 10.1109/TVLSI.2024.3375350.

[24] H. A. Balef, H. Jiao, J. P. d. Gyvez, and K. Goossens, "An analytical model for interdependent setup/hold-time characterization of flip-flops," in *2017 18th International Symposium on Quality Electronic Design (ISQED)*, 2017, pp. 209-214, doi: 10.1109/ISQED.2017.7918318.

[25] N. Maheshwari and S. S. Sapatnekar, *Timing Analysis and Optimization of Sequential Circuits*. Springer Science+Business Media New York, NY, 1999.

[26] R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2 ed. Pearson, 2015.

[27] C. Intel, "Timing Analyzer User Guide, Quartus Prime Handbook," Intel Corporation, 2022.

[28] P. Saxena and C. R. Visweswariah, "Design Automation for Timing Closure," *Proceedings of the IEEE,* vol. 103, no. 11, pp. 2131-2151, 2015, doi: 10.1109/JPROC.2015.2475396.

[29] G. Gimenez, A. Cherkaoui, G. Cogniard, and L. Fesquet, "Static Timing Analysis of Asynchronous Bundled-Data Circuits," in *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2018, pp. 110-118, doi: 10.1109/ASYNC.2018.00036.

[30] X. Lin, E. Dagless, and A. Lu, "Technology mapping of LUT based FPGAs for delay optimisation," in *Field-Programmable Logic and Applications*, Berlin, Heidelberg, W. Luk, P. Y. K. Cheung, and M. Glesner, Eds., 1997: Springer Berlin Heidelberg, pp. 245-254.

[31] K. Ian, T. Russell, and R. Jonathan, *FPGA Architecture: Survey and Challenges*. now, 2008, p. 1.

[32] V. Bhardwaj, *FPGA Design Flow and CAD*. 2021.

[33] X. (AMD), *Vivado Design Suite User Guide: Design Flows Overview (UG892)*. 2023.

[34] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, "Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 10, pp. 1904-1921, 2006, doi: 10.1109/TCAD.2005.860958.

[35] L. Zhou, S. Xiao, H. Wang, J. Wang, Z. Xu, B. Wang, and Z. Yu, "Toward Efficient Asynchronous Circuits Design Flow Using Backward Delay Propagation Constraint," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 32, no. 10, pp. 1852-1863, 2024, doi: 10.1109/TVLSI.2024.3418769.

[36] T. Valeeprakhon, Z. Zhang, and M. Iwata, "An Efficient Design Flow for Iterative Asynchronous Bundled-Data Circuits on FPGA," in *2024 9th International Conference on Integrated Circuits, Design, and Verification (ICDV)*, 2024, pp. 171-176, doi: 10.1109/ICDV61346.2024.10616447.

[37] T. Valeeprakhon, Z. Zhang, and M. Iwata, "Efficient FPGA Implementation of Asynchronous BD AES Circuits Based on Iterative Ring Structure. ," in *The 8th International Symposium on Frontier Technology (ISFT)* 2024.

[38] A. Mardari, Z. Jelčicová, and J. Sparsø, "Design and FPGA-implementation of Asynchronous Circuits Using Two-Phase Handshaking," in *25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2019, pp. 9-18, doi: 10.1109/ASYNC.2019.00010. [Online]. Available: https://ieeexplore.ieee.org/stampPDF/getPDF.jsp?tp=&arnumber=8850673&ref=

[39] P. Soille, "Erosion and Dilation," in *Morphological Image Analysis: Principles and Applications*, P. Soille Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 63-103.

[40] K. R. Cho, K. Okura, and K. Asada, "Design of a 32-bit fully asynchronous microprocessor (FAM)," in *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, 1992, vol. 2, pp. 1500-1503, doi: 10.1109/MWSCAS.1992.271009.

[41] E. Brunvand, "The NSR processor," in *Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*, 1993, vol. 1, pp. 428-435, doi: 10.1109/HICSS.1993.270622.

[42] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura, "TITAC: design of a quasi-delay-insensitive microprocessor," *IEEE Design & Test of Computers,* vol. 11, no. 2, pp. 50-63, 1994, doi: 10.1109/54.282445.

[43] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple, "AMULET1: an asynchronous ARM microprocessor," *IEEE Transactions on Computers,* vol. 46, no. 4, pp. 385-398, 1997, doi: 10.1109/12.588033.

[44] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver, "AMULET2e: an asynchronous embedded controller," in *Proceedings Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997, pp. 290-299, doi: 10.1109/ASYNC.1997.587182.

[45] S. B. Furber, D. A. Edwards, and J. D. Garside, "AMULET3: a 100 MIPS asynchronous embedded processor," in *International Conference on Computer Design*, 2000, pp. 329-334, doi: 10.1109/ICCD.2000.878304.

[46] J. M. C. Tse and D. P. K. Lun, "ASYNMPU: a fully asynchronous CISC microprocessor," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1997, vol. 3, pp. 1816-1819, doi: 10.1109/ISCAS.1997.621499.

[47] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya, "TITAC-2: an asynchronous 32-bit microprocessor based on scalable-delay-insensitive model," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, 1997, pp. 288-294, doi: 10.1109/ICCD.1997.628881.

[48] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and L. Tak Kwan, "The design of an asynchronous MIPS R3000 microprocessor," in *Proceedings Seventeenth Conference on Advanced Research in VLSI*, 1997, pp. 164-181, doi: 10.1109/ARVLSI.1997.634853.

[49] M. Fiorentino, C. Thibeault, and Y. Savaria, "Introducing KeyRing self-timed microarchitecture and timing-driven design flow," *IET Computers & Digital Techniques,* vol. 15, no. 6, pp. 409-426, 2021, doi: 10.1049/cdt2.12032.

[50] A. J. Martin, M. Nystrom, K. Papadantonakis, P. I. Penzes, P. Prakash, C. G. Wong, J. Chang, K. S. Ko, B. Lee, E. Ou, J. Pugh, E. V. Talvala, J. T. Tong, and A. Tura, "The Lutonium: a sub-nanojoule asynchronous 8051 microcontroller," in *Ninth International Symposium on Asynchronous Circuits and Systems*, 2003, pp. 14-23, doi: 10.1109/ASYNC.2003.1199162.

[51] L. Necchi, L. Lavagno, D. Pandini, and L. Vanzago, "An ultra-low energy asynchronous processor for wireless sensor networks," in *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2006, pp. 8 pp.-85, doi: 10.1109/ASYNC.2006.9.

[52] Q. Z. a. G. Theodoropoulos, "SAMIPS: A Synthesised Asynchronous Processor," *ArXiV,* vol. abs/2409.20388, pp. 1-40, 2024.

[53] S.-J. Lee, D.-Y. Lee, Y.-W. Ko, and J.-G. Lee, "Asynchronous Circuit Design on an FPGA: MIPS Processor Case Study," in *Convergence and Hybrid Information Technology*, Berlin, Heidelberg, G. Lee, D. Howard, D. Ślęzak, and Y. S. Hong, Eds., 2012: Springer Berlin Heidelberg, pp. 480-487.

[54] M. Fiorentino, Y. Savaria, and C. Thibeault, "FPGA implementation of Token-based Self-timed processors: A case study," in *15th IEEE International New Circuits and Systems Conference (NEWCAS)*, 2017, pp. 313-316, doi: 10.1109/NEWCAS.2017.8010168.

[55] J.-H. Lee, S.-S. Lee, and K.-R. Cho, *Asynchronous ARM Processor Employing an Adaptive Pipeline Architecture*. 2007, pp. 39-48.

[56] J. H. Lee, Y. H. Kim, and K. R. Cho, "A Low-Power Implementation of Asynchronous 8051 Employing Adaptive Pipeline Structure," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 55, no. 7, pp. 673-677, 2008, doi: 10.1109/TCSII.2008.921589.

[57] M. A. Yong, K. M. Mok, W. K. Lee, S. K. Teoh, and D. C. K. Wong, "RISC32-A: A Low-Power Asynchronous IoT Processor on FPGA With Adaptive Pipeline Structure," *IEEE Internet of Things Journal,* vol. 12, no. 9, pp. 12669-12685, 2025, doi: 10.1109/JIOT.2024.3521365.

[58] H. Wang, S. Xiao, L. Zhou, J. Wang, Z. Xu, B. Wang, and Z. Yu, "LAC: A Novel Lightweight Asynchronous Controller With Optimized Phase Shift," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 71, no. 8, pp. 3920-3924, 2024, doi: 10.1109/TCSII.2024.3369643.

[59] H. Huang, J. Zhang, D. Huo, Q. Sun, W. Rhee, and H. Chen, "An Asynchronous RISC-V Processor Utilizing a Chisel-Based Desynchronization Flow," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025, pp. 1-5, doi: 10.1109/ISCAS56072.2025.11043907.

[60] K. A. Andrew Waterman, SiFive Inc.,. E. D. 2CS Division, University of California. (2019). The RISC-V Instruction Set Manual Volume I: Unprivileged ISA.

[61] S. L. Harris and D. Harris, "Digital Design and RISC-V Computer Architecture Textbook," in *ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, 2021, pp. 1-5, doi: 10.1109/WCAE53984.2021.9707615.

[62] N. G. a. M. Karasek. "RISC-V Architecture Test SIG." github. https://github.com/riscv-non-isa/riscv-arch-test (accessed 15/01, 2024).

[63] D. Bertozzi, G. Miorandi, A. Ghiribaldi, W. Burleson, G. Sadowski, K. Bhardwaj, W. Jiang, and S. M. Nowick, "Cost-Effective and Flexible Asynchronous Interconnect Technology for GALS Systems," *IEEE Micro,* vol. 41, no. 1, pp. 69-81, 2021, doi: 10.1109/MM.2020.3002790.

[64] K. L. Chang, J. S. Chang, B. H. Gwee, and K. S. Chong, "Synchronous-Logic and Asynchronous-Logic 8051 Microcontroller Cores for Realizing the Internet of Things: A Comparative Study on Dynamic Voltage Scaling and Variation Effects," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems,* vol. 3, no. 1, pp. 23-34, 2013, doi: 10.1109/JETCAS.2013.2243031.

[65] I. AMD, *Vivado Design Suite User Guide: Synthesis (UG901)*, 2025.