POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Architectural exploration of KeyRing self-timed processors

MICKAEL FIORENTINO
Département de génie électrique

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Microélectronique

Septembre 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Architectural exploration of KeyRing self-timed processors

présentée par **Mickael FIORENTINO**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Jean-Pierre DAVID, président
Yvon SAVARIA, membre et directeur de recherche
Claude THIBEAULT, membre et codirecteur de recherche
Pierre LANGLOIS, membre
Zeljko ZILIC, membre externe

À mon père...

REMERCIEMENTS

En premier lieu, je remercie chaleureusement mes directeurs de thèse Yvon Savaria et Claude Thibeault pour leur soutien indéfectible durant les 5 années de mon doctorat. J'ai bénéficié d'une très grande liberté dans mes directions de recherches et dans mes choix éditoriaux ; je les remercie tout particulièrement d'avoir toujours encouragé et appuyé mes décisions.

Merci aux membres du jury ainsi qu'à Tom Awad d'Octasic d'avoir pris le temps de lire et d'évaluer cette thèse de doctorat.

Mes remerciements s'adressent ensuite à tous mes collègues du GRM pour leur bienveillance à mon égard et leur enthousiasme quotidien. Je tiens à remercier particulièrement Omar et Érika avec qui j'ai noué des liens d'amitié qui dépassent de loin le cadre du laboratoire. Je leur dois beaucoup, et je leur serai éternellement reconnaissant d'avoir été à mes côtés. Merci à Michel, Mathieu, Thibaut, Jefferson et Thomas pour nos franches rigolades et nos débats animés. Un grand merci également à Réjean pour sa disponibilité sans faille, mais aussi et surtout pour son amitié sincère.

Je profite de cette occasion pour remercier mes amis, Céline, Anders, Sabrina, Bastien, Jeanne, Arthur, Alexy, et Jean-Baptise avec qui j'ai partagé cette tranche de vie à Montréal et que je n'oublierai jamais.

Enfin, je remercie toute ma famille, en particulier ma sœur et mon père, pour leur soutien inconditionnel et leur amour salvateur. Surtout, merci à Juliette, ma compagne, mon amie, ma confidente, l'amour de ma vie ; merci à elle de rendre ma vie merveilleuse.

RÉSUMÉ

Les dernières décennies ont vu l'augmentation des performances des processeurs contraintes par les limites imposées par la consommation d'énergie des systèmes électroniques : des très basses consommations requises pour les objets connectés, aux budgets de dépenses électriques des serveurs, en passant par les limitations thermiques et la durée de vie des batteries des appareils mobiles. Cette forte demande en processeurs efficients en énergie, couplée avec les limitations de la réduction d'échelle des transistors—qui ne permet plus d'améliorer les performances à densité de puissance constante—, conduit les concepteurs de circuits intégrés à explorer de nouvelles microarchitectures permettant d'obtenir de meilleures performances pour un budget énergétique donné. Cette thèse s'inscrit dans cette tendance en proposant une nouvelle microarchitecture de processeur, appelée *KeyRing*, conçue avec l'intention de réduire la consommation d'énergie des processeurs.

La fréquence d'opération des transistors dans les circuits intégrés est proportionnelle à leur consommation dynamique d'énergie. Par conséquent, les techniques de conception permettant de réduire dynamiquement le nombre de transistors en opération sont très largement adoptées pour améliorer l'efficience énergétique des processeurs. La technique de *clock-gating* est particulièrement usitée dans les circuits synchrones, car elle réduit l'impact de l'horloge globale, qui est la principale source d'activité. La microarchitecture KeyRing présentée dans cette thèse utilise une méthode de synchronisation décentralisée et asynchrone pour réduire l'activité des circuits. Elle est dérivée du processeur AnARM, un processeur développé par Octasic sur la base d'une microarchitecture asynchrone *ad hoc*. Bien qu'il soit plus efficient en énergie que des alternatives synchrones, le AnARM est essentiellement incompatible avec les méthodes de synthèse et d'analyse temporelle statique standards. De plus, sa technique de conception ad hoc ne s'inscrit que partiellement dans les paradigmes de conceptions asynchrones. Cette thèse propose une approche rigoureuse pour définir les principes généraux de cette technique de conception ad hoc, en faisant levier sur la littérature asynchrone. La microarchitecture KeyRing qui en résulte est développée en association avec une méthode de conception automatisée, qui permet de s'affranchir des incompatibilités natives existant entre les outils de conception et les systèmes asynchrones. La méthode proposée permet de pleinement mettre à profit les flots de conception standards de l'industrie microélectronique pour réaliser la synthèse et la vérification des circuits KeyRing. Cette thèse propose également des protocoles expérimentaux, dont le but est de renforcer la relation de causalité entre la microarchitecture KeyRing et une réduction de la consommation énergétique des processeurs, comparativement à des alternatives synchrones équivalentes.

La principale contribution de cette thèse est le développement conjoint de la microarchitecture KeyRing et de la méthode de conception automatisée, qui permet l'exploration architecturale de circuits KeyRing. En particulier, cette dissertation présente *i)* une définition rigoureuse de la microarchitecture KeyRing ; *ii)* la conception de circuits KeyRing, incluant plusieurs processeurs de complexité croissante, ainsi que leurs alternatives synchrones utilisées comme points de comparaison pour l'évaluation des performances et de la consommation d'énergie ; *iii)* des méthodes de conception dédiées, et des contraintes temporelles adaptées, rendant les circuits KeyRing compatibles avec les outils de synthèse et de vérifications temporelles standards ; et *iv)* l'élaboration de protocoles expérimentaux permettant de générer des résultats reproductibles.

Un premier processeur, appelé *Mini-Mips*, basé sur la technique de conception asynchrone ad hoc d'Octasic, a été conçu en suivant une approche empirique. Son étude met au jour les principes de conception à l'origine de la microarchitecture KeyRing. Le Mini-Mips implémente une version simplifiée du jeu d'instruction MIPS, et cible une implémentation sur plateforme d'émulation matérielle FPGA, où il est comparé avec une alternative synchrone classique (pipeline à 5 étages). Des expériences réalisées sur la plateforme FPGA—permettant de contrôler l'exécution de programmes de test sur les processeurs cibles tout en observant leur consommation d'énergie—valident le fonctionnement du Mini-Mips. Cette première expérience met également en lumière les limitations de l'approche empirique, en particulier s'agissant de la méthode de conception. Ces limites sont repoussées avec la microarchitecture KeyRing et la méthode de conception automatisée qui y est associée.

Les Key Unit (KU)—circuits responsables de la génération décentralisée d'horloges asynchrones—ainsi que le protocole KeyRing—un arrangement de KU ayant une topologie toroïdale—sont au coeur de la microarchitecture KeyRing. Des définitions sont dérivées de l'analyse du graphe modélisant le protocole KeyRing. Elles permettent de prédire des propriétés des circuits KeyRing (niveau de parallélisme, performances) à partir des paramètres de la microarchitecture. Le modèle est également le support de définitions formelles pour les contraintes temporelles des circuits KeyRing. Ces contraintes sont décrites au format SDC, et exploitées par le flot de conception standard. Spécifiquement, les circuits KeyRing sont synthétisés en technologie ASIC 65 nm en utilisant les outils de conception de Synopsys. La méthode de conception automatisée proposée dans cette thèse est validée en simulation post-synthèse, et par l'analyse des rapports de l'analyse temporelle.

La microarchitecture KeyRing et sa méthode de conception automatisée sont valorisées par la conception du processeur *KeyV*, qui implémente la variante RV32IM complète du jeu d'instructions RISC-V. La comparaison de deux microarchitectures de KeyV—deux variantes

de la microarchitecture KeyRing—illustre les compromis de KeyRing entre résilience aux violations de contraintes temporelles, et performances. Un protocole expérimental basé sur les simulations post-synthèse des programmes de référence Dhrystone et Coremark, associées à l'analyse de puissance qui prend en compte l'activité des circuits en simulation, permet de comparer le processeur KeyV avec une alternative synchrone classique, appelée SynV (6 étages de pipeline), synthétisée avec et sans clock-gating. Les résultats montrent d'abord que les netlists de KeyV exécutent les programmes de références en simulation post-synthèse sans violations des contraintes temporelles. L'évaluation des performances et de la consommation d'énergie montre ensuite que KeyV possède une efficience énergétique entre celle de SynV avec clock-gating et celle de SynV sans clock-gating.

ABSTRACT

Over the last years, microprocessors have had to increase their performances while keeping their power envelope within tight bounds, as dictated by the needs of various markets: from the ultra-low power requirements of the IoT, to the electrical power consumption budget in enterprise servers, by way of passive cooling and day-long battery life in mobile devices. This high demand for power-efficient processors, coupled with the limitations of technology scaling—which no longer provides improved performances at constant power densities—, is leading designers to explore new microarchitectures with the goal of pulling more performances out of a fixed power budget. This work enters into this trend by proposing a new processor microarchitecture, called *KeyRing*, having a low-power design intent.

The switching activity of integrated circuits—*i.e.* transistors switching on and off—directly affects their dynamic power consumption. Circuit-level design techniques such as clock-gating are widely adopted as they dramatically reduce the impact of the global clock in synchronous circuits, which constitutes the main source of switching activity. The KeyRing microarchitecture presented in this work uses an asynchronous clocking scheme that relies on decentralized synchronization mechanisms to reduce the switching activity of circuits. It is derived from the AnARM, a power-efficient ARM processor developed by Octasic using an *ad hoc* asynchronous microarchitecture. Although it delivers better power-efficiency than synchronous alternatives, it is for the most part incompatible with standard timing-driven synthesis and Static Timing Analysis (STA). In addition, its design style does not fit well within the existing asynchronous design paradigms. This work lays the foundations for a more rigorous definition of this rather unorthodox design style, using circuits and methods coming from the asynchronous literature. The resulting KeyRing microarchitecture is developed in combination with Electronic Design Automation (EDA) methods that alleviate incompatibility issues related to *ad hoc* clocking, enabling timing-driven optimizations and verifications of KeyRing circuits using industry-standard design flows. In addition to bridging the gap with standard design practices, this work also proposes comprehensive experimental protocols that aims to strengthen the causal relation between the reported asynchronous microarchitecture and a reduced power consumption compared with synchronous alternatives.

The main achievement of this work is a framework that enables the architectural exploration of circuits using the KeyRing microarchitecture. Specifically, this work presents *i)* a rigorous definition of the KeyRing microarchitecture; *ii)* the design of KeyRing circuits, including several processors of increasing complexity, in combination with their synchronous alterna-

tives that can be used as baseline for performance and power consumption comparisons; *iii)* custom design flows and timing constraints enabling KeyRing circuits to benefit from timing-driven optimization and verification capabilities of standard EDA tools; and *iv)* the elaboration of experimental protocols that permit to generate reproducible results.

A first processor, called *Mini-Mips*, is designed with an Octasic-style asynchronous microarchitecture using an empirical approach. Its study allows to uncover the main design principles that are then used to define the KeyRing design style. The Mini-Mips implements a simplified version of the MIPS Instruction Set Architecture (ISA) and targets an Field Programmable Gate Array (FPGA) emulation platform. It is compared with a classical 5-stage synchronous pipeline microarchitecture having comparable characteristics. An EDA method is proposed for the timing-driven implementation of the asynchronous Mini-Mips on FPGA. It is validated by experimental results conducted on the emulation platform, which allows to control the execution of benchmarks on the processors while monitoring their power consumption. This first design attempt also demonstrates shortcomings of the design, and the EDA method, which are addressed in the KeyRing microarchitecture and the definitive EDA method.

At the core of the KeyRing microarchitecture are the Key Units (KUs), a self-timed clock controller, and the KeyRing protocol, an organization of KUs having a toroidal mesh topology. Definitions derived from the analysis of the KeyRing graph allow to predict the performances of KeyRing circuits, by establishing links between the parameters of the circuit, the level of parallelism, and the speed of self-timed clocks. Formal definitions of the timing constraints in KeyRing circuits are proposed. Their implementation in the standard SDC format is leveraged by a novel EDA method enabling the timing-driven synthesis of KeyRing circuits using industry-standard design flows. In particular, KeyRing circuits are synthesized using a 65 nm ASIC design kit and Synopsys EDA tools. The proposed EDA method is validated with post-synthesis timing simulations and the analysis of STA reports.

The KeyRing microarchitecture and the associated EDA method is showcased with the design of complete RV32IM RISC-V processor, called *KeyV*. Two KeyV microarchitectures are compared, illustrating trade-offs between robustness to timing-violations and performances. An experimental protocol based on post-synthesis timing simulations of the standard Dhrystone and Coremark benchmarks, combined with activity-aware power analysis, is proposed to compare KeyV processors with a 6-stage synchronous pipeline alternative, called *SynV*, synthesized with and without clock-gating. Results first show that the KeyV netlists successfully execute complex benchmarks in post-synthesis simulations without timing violations. Performance and power consumption evaluations then show that KeyV has a power efficiency in between SynV with clock-gating and SynV without clock-gating.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuit
BD	Bundled-Data
BM	Burst Mode
BFS	Breadth First Search
CAD	Computer Aided Design
CCS	Composite Current Source
CISC	Complex Instruction Set Computing
CLB	Configurable Logic Block
CSR	Control Status Register
CTS	Clock Tree Synthesis
DC	Design Compiler
DE	Delay Element
DI	Delay-Insensitive
DPM	Dual Port Memory
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
EDA	Electronic Design Automation
EU	Execution Unit
FIFO	First In First Out
FSM	Finite State Machine
FPGA	Field Programmable Gate Array
GALS	Globally Asynchronous Locally Synchronous
GCC	GNU Compiler Collection
GPDK	General Purpose Design Kit
HDL	Hardware Description Language
ILP	Instruction Level Parallelism
IoT	Internet Of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
KU	Key Unit

LCS	Local Clock Set
LUT	Look Up Table
LSU	Load Store Unit
MIPS	Millions Instructions Per Second
MMACS	Millions of Multiply-and-Accumulate operations per Second
NRZ	Non Return to Zero
NoC	Network on Chip
NTC	Near Threshold Computing
PaR	Place and Route
PC	Program Counter
PCB	Printed Circuit Boards
PL	Programmable Logic
PS	Processing System
PVT	Process, Voltage, and Temperature
QDI	Quasi-Delay-Insensitive
RAW	Read After Write
RF	Register File
RISC	Reduced Instruction Set Computing
RT	Relative Timing
RTC	Relative Timing Constraint
RTL	Register Transfer Level
RZ	Return to Zero
SDC	Synopsys Design Constraints
SDD	Small-Delay Defect
SDF	Standard Delay Format
SNUG	Synopsys User Group
SoC	System On Chip
STA	Static Timing Analysis
STG	Signal Transition Graph
TDP	Thermal Design Power
TLM	Transaction Level Modeling
TU	Token Unit
VLSI	Very Large Scale Integration
WAR	Write After Read
XBM	eXtended Burst Mode
XDC	Xilinx Design Constraints

CHAPTER 1 INTRODUCTION

1.1 Context and Motivations

Improving the energy efficiency of Very Large Scale Integration (VLSI) circuits is the main driving force of the microelectronic industry [1]. As the most ubiquitous components of information systems, microprocessors are representative of this trend. Their wide range of applications require different trade-offs in the performance *vs.* power consumption space. On one hand, processors designed for wireless sensors found in the Internet Of Things (IoT) typically have ultra-low power requirements. On the other hand, processors designed for computational intensive tasks, as found in personal computers and servers, favor high performances. In between, processors designed for mobile and embedded devices are instances where the performance/power trade-offs are the most aggressively explored. Indeed, in recent years, mobile processors have benefited from tremendous improvements in performances while keeping their power envelope within tight boundaries, as dictated by the requirements of the mobile devices market: passive cooling, and day-long battery life [2]. Technologies, architectures, and design methods that enhance the energy efficiency of microprocessors are beneficial for the entire design space—low-power processors can have better performances, and high-performance processors can consume less power. Given the scale at which processors are deployed, even minor improvements can have huge impacts. Apart from power and performances, other challenges have been identified by the International Technology Roadmap for Semiconductors (ITRS), which include dealing with increased variability, aging, scalability, and fault rates, as the integration density of VLSI circuits continues to rise.

1.1.1 Overview of energy-aware innovations in microprocessors

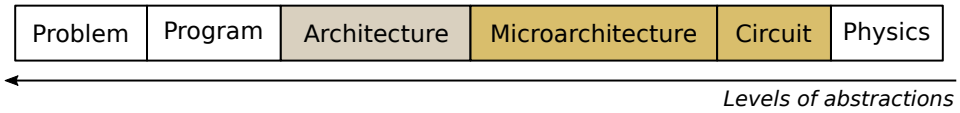


Figure 1.1 Levels of abstraction involved in solving problems with microprocessors [3]

Patt presents in [3] a view of microprocessors in abstraction layers, as depicted in Figure 1.1. Problems, formulated by algorithms, are eventually solved in microprocessors by flows of electrons, controlled by an arrangement of transistors. The discipline of microprocessor design lie between the *program* and the *physics* abstractions. Architectures (also known as

Instruction Set Architectures (ISAs)) are the specifications of processors, which are implemented in microarchitectures by means of circuits. Each of these abstraction layers has an impact on the performance and the power consumption of microprocessors. For decades, improvements in performances of microprocessors were led by innovations in transistors architectures and manufacturing processes. But the sustained increase in transistor switching rates, as we moved from one transistor technology to the next, also led to an overall growth in power consumption [1]. For example, the *PowerPC 601*, launched by IBM in 1993, has a Thermal Design Power (TDP) of 6,5 W and operates at 75 MHz, while the *Power7*, released by IBM in 2010, has a TDP of 200 W and operates at 3,5 GHz [4]. Around the 2010's, the industry hit a power-wall—increasing the switching rates further would lead to TDPs that could not be properly handled by most systems. In the mean time, the semiconductor manufacturing industry—that is, the companies, called *foundries*, that fabricate the chips from silicon wafers based on circuits design—was facing technological challenges that slowed down Moore's law. New transistor architectures and innovative manufacturing processes require important investments, which results in Application Specific Integrated Circuits (ASICs) having ever increasing production costs. These two factors—the power wall and the manufacturing costs—have been the source of renewed interests in microarchitectural and circuits innovations, which today have a greater influence on the overall growth in energy efficiency of microprocessors. In this work, we focus on the microarchitectural aspect of microprocessors.

CMOS VLSI circuits consume most of their energy in switching transistors. That is, the *dynamic* energy consumption due to transitions ($0 \rightarrow 1$ and $1 \rightarrow 0$), has a greater influence than the *static* energy consumption due to on- and off-currents dissipations while the transistors are idle. The power $P_{dynamic}$ required per transistor depends on the capacitive load C_{load} it drives, the voltage V, and the frequency of the transitions F [5]:

$$P_{dynamic} \propto C_{load} \times V^2 \times F \tag{1.1}$$

Dynamic power is first impacted by innovations in transistor architectures and manufacturing processes. Voltages have dropped from 5 V down to 1 V, and the load capacitance has decreased as we moved from one transistor technology node to the next, reducing power consumption. Higher integration density of transistors, coupled with higher switching frequencies, have, as a first approximation, increased power consumption as much as they have improved performances, thus having a moderated impact on energy. Alternative circuits logic style have been proposed as a way to reduce dynamic power or improve performances. For example, *dynamic logic* styles—such as domino logic, C^2MOS, or pass-transistor logic, which were widely used in asynchronous circuits—were proposed as energy-efficient alterna-

tives to the static CMOS logic style [6, 7]. But CMOS logic remains the *de-facto* choice for semi-custom design methodologies of VLSI circuits, because they compare favorably with respect to circuit speed and layout efficiency, and are robust against transistor downsizing and voltage scaling which allow efficient performance and power optimizations [8]. Indeed, in the last decade, the CMOS scaling paradigm has changed from "*speed hungry*" to "*power thrifty*" [9]. For example, operating transistors in the sub-threshold regime—a circuit design technique, called Near Threshold Computing (NTC) [10]—is of prevailing importance today. NTC is widely adopted in the ultra-low power design space because it is extremely effective to reduce energy, by scaling voltages down to 0,6 V, despite inducing performance penalties. An influential research project conducted at the ETH Zurich, called PULP, have recently demonstrated the effectiveness of this technique with a System On Chip (SoC) achieving 1 GOPS within a 10 mW power envelope [11].

On the opposite side of the microprocessor abstraction view, programs and architectures as well have an important impact on dynamic power. ISAs designed for the typical case, favoring fewer, and simpler, instructions—a paradigm known as Reduced Instruction Set Computing (RISC)—, are more energy efficient than Complex Instruction Set Computing (CISC) architectures alternatives [5]. Although CISC architectures can produce shorter programs, having more instructions that are more complex leads to microarchitectures with longer datapaths and bigger control structures, which results in an overall increase in power due to both a higher number of transistors and a higher switching activity. Compilers, often designed with microarchitectural aspects in mind, allow to optimize programs to reduce their size and improve their performance, thus improving the overall energy efficiency of the system.

Another key component enabling energy-aware designs is automation. Advances in software development, coupled with improvements of processors performances, enabled the advent of Electronic Design Automation (EDA) in the late 1980's [12]. Hardware Description Languages (HDLs), such as VHDL and Verilog, were created in combination with simulation engines, logic synthesis tools, and placement and routing algorithms, to automate the conception of VLSI circuits from high-level models. Computer Aided Design (CAD) tools have been responsible for a paradigm shift in the way integrated circuits are designed. Indeed, high-level modeling and simulations allow to design circuits that are more complex and less error prone; the synthesis of these models into optimized gate level circuits improves performance and power consumption, while lowering time-to-market; timing verifications and formal equivalence improve reliability, *etc.* For example, synthesis tools automatically size and organize the gates in a circuit according to user-defined constraints. Optimization algorithms balance constraints of conflicting purposes (*e.g.* performance and power)—for instance by choosing gates of varying sizes and drive strength—, to find the optimal design point.

Dynamic power can be reduced with system-level innovations, the most widely adopted of which are Dynamic Voltage Scaling (DVS) and clock-gating [13]. The DVS technique allows to alter the voltage of a circuit, or part of a circuit, as a function of its computation requirements. It is most effective in large circuits, where the chip can be split in independently powered island. The voltage of each island is dropped down when no computations are performed, thus saving power. The frequency of the processing islands can also be altered to improve the power savings, in which case the technique is referred to as Dynamic Voltage and Frequency Scaling (DVFS). The clock-gating technique only affects the switching frequency parameter F. Instead of altering the transistors switching rate, which, as we have seen, has a limited impact on energy, the idea is to limit the number of transistors switching, during the computation of a task, to the one being used for that computation. Clock-gating is performed by advantageously placing dedicated clock-gating cells on the clock-tree; a process mostly automated with modern EDA tools. These cells, dynamically controlled by the circuit, can block a clock signal reaching a group of registers, thus preventing the logic driven by these registers from switching. In large circuits, it can have huge impacts on the power consumption, while having virtually no impact on performances. The DVFS and the clock-gating techniques are often used in combination. When used in smaller circuits, however, the power consumption overhead due to the added logic required to implement both techniques can dominate over the savings they induce by operating.

Finally, dynamic power is affected by the Register Transfer Level (RTL) design paradigm used in the system. There are two main competing branches: the *synchronous* design paradigm, in which data are sampled at a fixed rate using a global synchronization signal called a clock, and the *asynchronous* design paradigm, in which data are sampled at a dynamic rate using local synchronizations mechanisms [7, 14, 15]. Both design paradigms were born at the same time, in the 1950's, from the requirements of sequential circuits implementing Finite State Machines (FSMs)—that is, the need for synchronization methods allowing to update the logical states of FSMs, while preventing glitches due to propagation delays through combinational logic and feedback loops. When computers occupied the space of entire rooms, the delays involved in moving data favored designing with an asynchronous approach (the ILIAC is a famous example of such early processors built from asynchronous design principles). As computers scaled down to Printed Circuit Boards (PCBs), first, and to VLSI circuits then, in the 1970's, the delays decreased enabling synchronous circuits to thrive. The synchronous paradigm started to dominate, mainly because it is inherently simpler than its asynchronous competitor. The rise of EDA took place while the synchronous paradigm was mainstream. Over the years, the synchronous paradigm has driven the evolution of CAD tools towards meeting its design needs, thus solidifying its dominance.

1.1.2 Synchronous and asynchronous design paradigms

The synchronous paradigm is efficient. Having a single clock signal, with a fixed periodicity, dictating the pace of the circuit facilitates the design and eases automation. On one hand, the simplicity of synchronous designs enables repeatable structures, and generic communication channels that favor *design reuse* [16]. The different building blocks of a big circuit can be built and assembled independently, using synchronizers if the clocks frequencies do not match [17]; the knowledge of tow modules native frequency is enough to synchronize their contents. This had a great influence on the emergence of companies specializing in the design of reusable modules, called Intellectual Property (IP), intended to be used as part of bigger circuits. On the other hand, the simplicity of the timing conditions ensuring the correct operation of synchronous circuits—the rate at which logic states can be updated must be within the bounds of the clock frequency—are easy to enforce and validate. These timing conditions are translated into design constraints, which fuel EDA tools optimization algorithms and verification engines. This dual outcome of the synchronous paradigm simplicity—common design practices favoring design reuse, and simple timing requirements enabling the extensive use of design automation—is the backbone of its resilience and long-term dominance [1].

However, the synchronous paradigm suffers from performance and power limitations due to the distribution of the clock [18]. First, the distribution of a global clock signal throughout a VLSI chip—often performed using Clock Tree Synthesis (CTS)—becomes more difficult as the density of integration of VLSI circuits increases. Indeed, as circuits carry on with Moore's law, the relative distance between synchronization elements increases, and wire delays have a greater impact on the total delay of the logic. The *skew* induced by longer clock distribution networks—that is, the maximum delay between the occurrences of a clock edge in different locations of the circuit—has limited the performances of VLSI circuits to the point where, passed a certain complexity, it is now impossible to use a single clock signal for the entire design without seriously altering its performances. Furthermore, this trend is accentuated by the difficulty to achieve *timing closure*—that is, validating the correct operation of the circuit from a timing perspective—in face of increased Process, Voltage, and Temperature (PVT) variations induced by technology scaling. This is especially pronounced with synchronous circuits, which must account for the worst-case delays in the circuit to size the clocks. Then, in addition to the performance penalties induced by clock skews, the scaling of the clock distribution network has resulted in important power consumption issues. A famous example illustrating the impact of the clock tree on the power consumption of a microprocessor is the Alpha 21164. In a 1998 publication [19], the authors uncover the power breakdown of this high-performance processor, showing that the clocks are responsible of over 40 % of the total.

The asynchronous paradigm is elegant. Unlike the theory of synchronous circuits design, which is quite simple, the theoretical aspects of asynchronous circuits design is still an active research area. This field, which has seen significant development since the late 1980's, is derived from both the first asynchronous FSMs of the 1950's, with the influential work of Huffman and Muller, and the theory of concurrency developed in computer science. The formal models of the asynchronous paradigm, which are conceptualized at a level of abstraction close to fundamental mathematics, not only have direct implications when applied to real circuits, but also play a significant role in arousing the interest of the research community. Asynchronous theories are elegant—the mathematical term for beautiful—, and their study should be pursued out of sheer curiosity, which is the most noble aspect of research.

On a more practical note, asynchronous designs have the potential to address the shortcomings of synchronous circuits, with respect to power consumption, PVT variations, and noise, while having competitive performances and proving robust to faults and aging [20]. Indeed, asynchronous circuits use distributed synchronization mechanisms, as opposed to the centralized clock used in synchronous systems, which support, by design, the modular compositions of variability-tolerant systems. This allows to adapt dynamically the activity of the circuit based on the computation requirements of the task being processed, without extensive instrumented power management. The inherent elasticity of asynchronous circuits is the source of their versatility, which makes them suitable candidates for a variety of use cases [21]. In the high-performance design space, designers use the elasticity of fine-grain asynchronous pipelines to achieve high throughputs. Two projects, both published in 1997, have famously attempted to demonstrate the superiority of asynchronous microprocessors: the asynchronous MIPS R3000 from the California Institute of Technology [22], and the AMULET, an asynchronous ARM developed in the University of Manchester [23]. Of interest is also the development of RAPPID [24], an asynchronous instruction length decoder for the Intel Pentium II processor, which achieved a 3× better throughput, 2× better latency, while consuming 2× less power within the same area footprint as the fastest commercial synchronous alternative fabricated on the same CMOS 0,25 µm process. In the low-power design space, designers exploit the distributed synchronization schemes of asynchronous systems to save power by reducing the switching activity of the circuit. A recent example illustrating this use-case is the asynchronous MSP430—a 16-bit Digital Signal Processor (DSP) from Texas Instruments (TI)—developed in the University of Utah in 2017, which exhibits a minimum of 5× energy per instruction improvement over a synchronous alternative [25]. Finally, asynchronous circuits have proved successful in some niche applications, such as contactless secured smartcards used in banking, or passport applications, where, in addition to being robust to the voltages variations of the energy source, their properties are also exploited

to counter hardware attacks. Some companies have successfully made the asynchronous paradigm an industrial reality. To name a few that are still active today: *Achronix* develops Field Programmable Gate Arrays (FPGAs) based on high-speed asynchronous logic; *Fulcrum*—now part of Intel—produces high-speed asynchronous networking chips; *Tiempo* designs variability-tolerant asynchronous circuits for secured smartcards; and *Octasic* develops communication systems based on their custom asynchronous DSPs [15, 20].

However, it is clear that the adoption of asynchronous circuits has been limited, and that the vast majority of the circuits designed today still rely on the synchronous paradigm. This can be explained by a combination of factors [7, 15]. First, asynchronous design styles are plentiful. In contrast with the synchronous paradigm, in which a fairly small number of design guidelines are agreed-upon [16], there is a wide range of asynchronous design styles, which vary with respect to data encoding, synchronization protocols, circuits logic styles, *etc.* Although this is the result of a healthy and productive research environment—many avenues are explored, which uncover new design opportunities—, it has the side effect of preventing design reuse. Paradoxically, most asynchronous design styles, when taken alone, surpass the synchronous paradigm in terms of modularity (for example, it is possible to communicate between circuits in separated chips without synchronizers). But it is difficult—if possible at all—to interface asynchronous circuits having different design styles. This limited design reuse is worsened by the dominance of synchronous circuits, as commercial IPs are likely to be synchronous, making their integration within an otherwise asynchronous system more complex. In addition, having multiple asynchronous design styles, each with a steep learning curve, does not play in favor of a growing designer base. The second factor affecting the adoption of asynchronous circuits is their lack of support by standard EDA tools, and their limited compatibility with conventional design and verification flows. Indeed, the leaders in the EDA industry (*Cadence*, *Synopsys*, and *Mentor* in the ASIC world, and *Xilinx* and *Altera*—now part of Intel—in the FPGA world) never invested in asynchronous technologies. In addition, most asynchronous design styles rely on specific cells—the Muller C-element is a good example—that are typically not part of the standard cells provided by the foundries as part of a transistor technology design kit. Although this is manageable for companies having the resources to design their own standard cells, or for researchers staying at the prototyping phase, it is a major issue for smaller companies. If they were to commit to a non standard cells based asynchronous design style, they would have to reinvest the resources in designing these cells for each new transistor technology node, which, in addition to the need for custom design flow, becomes a competitive liability. That is why, alternative asynchronous design styles and implementation flows are actively explored to overcome these limitations. An important part of this work fits with these trends.

Alternative asynchronous design styles—also known as *asynchronous templates*—that solely rely on standard cells have been proposed in the past two decades. The most influentials of them are reviewed in chapter 2. Alternative design flows have followed two paths [15, 26, 27]. The first path is concerned with designing custom EDA environments, some of which are still in active development today. These include custom HDLs, dedicated synthesis tools, and even FPGAs targeting asynchronous circuits prototyping. But the multiplicity of asynchronous design styles contributes to spreading the efforts put in these directions. Although some asynchronous EDA tools are currently used in an industrial context (the Tiempo custom synthesis flow is a known example), they are reserved for niche usage, and may arguably perform worse than their synchronous counterparts. The second path that is extensively explored is concerned with finding new methods that can bridge the gap between the asynchronous paradigm and industry-standard, synchronous-oriented, EDA tools. The automated implementation of an asynchronous circuit using standard CAD tools and conventional design flows is of varying complexity depending on the asynchronous design style used in the system of interest. In particular, the abilities of synchronous EDA tools to optimize a circuit such that it meets the performance requirements set by the designer in the form of timing constraints—such optimizations are said to be *timing-driven*—, and to verify that, under the worst case scenario, the circuit is exempt of error due to the propagation delay of signals through the logic—a verification performed by means of Static Timing Analysis (STA)—are difficult to exploit with asynchronous designs. The research in this direction is of utmost interest because timing-driven optimizations, and STA, are key components of modern EDA tools effectiveness.

In practice, asynchronous circuits which tend to follow fairly purist design approaches are developed in relative isolation. Nevertheless, globally synchronous systems still suffer from strong limitations as technologies continue to scale down. A solution that has emerged, and that has been widely adopted in the past decade to design SoCs of important complexity, is based on an hybrid approach called Globally Asynchronous Locally Synchronous (GALS) [14, 20, 28]. It relies on asynchronous design principles to build interfaces between locally clocked synchronous processing modules—they interact asynchronously with a network through asynchronous/synchronous interfaces or pausible clocks—, thereby taking advantages of both worlds. GALS designs benefit from the design reuse of synchronous IPs, and the flexibility provided by elastic asynchronous interconnects. However, this design style is more adapted for multi-core designs; the synchronous/asynchronous discussion still prevails at the core level. This work deals with asynchronous RTL design techniques for the design of processor cores, in combination with EDA methods that enable timing-driven optimizations and verifications.

1.1.3 Octasic ad hoc approach towards reducing power consumption

In this context, Octasic—a Montreal based company that has specialized in wireless and media processing systems based on their in-house DSPs—is developing a custom asynchronous design style with a low-power design intent. When they started the development of their DSPs, in the early 2000's, the strategy of Octasic to tackle the problem of the power consumption was to get rid of the clocks, which, as we have seen, represents a significant portion—close to half—of the total power budget in a synchronous VLSI circuit. Naturally, they considered the asynchronous paradigm. But the shortcomings of asynchronous circuits with respect to their integration with standard design flows were the main limitations that prevented Octasic from fully adopting the asynchronous paradigm. In particular, the need for designing custom cells specific to asynchronous circuits—typically not provided as part of standard design kits—was the biggest obstacle. Thus, instead of adopting existing asynchronous design techniques, they developed their own design style, that solely relies on standard cells, in combination with a dedicated design flow based on in-house CAD tools [29].

This led to the *Opus* family of asynchronous DSP cores: the Opus1 (used in the OCT1010) was released in 2008; the Opus2 (used in the OCT2200) was released in 2011; and the Opus3 (used in the OCT3032W) was released in 2018. They are used commercially and deployed in production telecommunication equipments sold by multiple companies worldwide. The details of their custom asynchronous design methodology was first revealed in two patents [30, 31]—filled in 2009 and respectively issued in 2012 and in 2014—, and the Opus2 microarchitecture was uncovered in a publication that appeared in the 2012 edition of the International Symposium on Asynchronous Circuits and Systems (ASYNC). In this publication, Octasic claims that the Opus2 DSP core delivers 2000 Millions of Multiply-and-Accumulate operations per Seconds (MMACS) with a power efficiency of 21 MMACS/mW when operating at 1 V. When compared in a similar 90 nm CMOS technology node, it is reported to be 3× better than the TI C64x+ synchronous DSP alternative—advertised by TI as the most power-efficient DSP at the time—, which delivers 4000 MMACS with a reported power efficiency of 7 MMACS/mw at 1 V. Moreover, it is of interest to note that the Opus DSPs have enjoyed a commercial success that is often cited as an example to illustrate the benefits of using asynchronous circuits [7, 20]. Finally, since the release of their publications, other companies have developed processors based on very similar design principles [32, 33].

Following the success of the Opus DSP cores, Octasic decided to leverage its original asynchronous design technique to build a general-purpose processor targeting embedded applications. A project started in 2012 as a research collaboration between Octasic, École de Technologie Supérieures (ETS), and Polytechnique Montreal, with the goal of building a proces-

sor, called the *AnARM*, capable of achieving performances comparable to those of the ARM Cortex-A7—one of the most power-efficient core used in mobile SoCs at the time [34]—in a smaller power envelope. To achieve this goal, the idea was to build upon the asynchronous subsystems that were originally developed for the Opus DSPs. The AnARM implements the ARMv7 ISA, although it does not include the *Thumb* and *NEON* ISA extensions for simplicity. The circuit was designed and fabricated using the STMicroelectronics 28 nm FD-SOI technology, which was the most advanced transistor technology available that allowed prototyping at the time of the design. Post-fabrication results have been published in the 2019 edition of ASYNC [35].

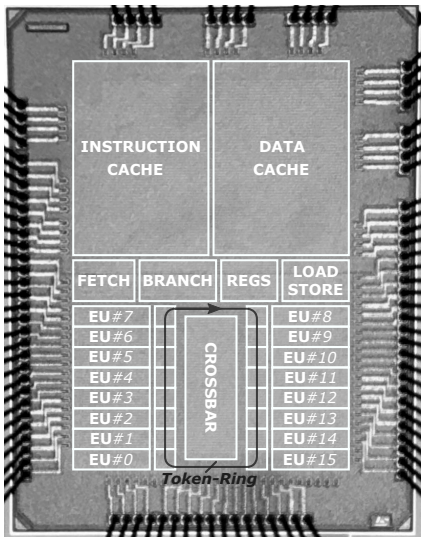


Figure 1.2 The *AnARM* die micrograph with an overlay representing its top-level organization. The core size is 0,78 mm × 1,13 mm (0,88 mm^2) [35].

Figure 1.2 shows the die micrograph of the AnARM, with an overlay summarizing its top-level organization. In contrast with most synchronous or asynchronous processors, the AnARM makes no use of pipelining to exploit the inherent parallelism of a sequence of instructions. Instead of pipelining, the AnARM computes multiple instructions concurrently in multiple *multicycle* Execution Units (EUs). A typical EU provides all the functions that would normally be found at different stages of a pipeline, including instruction decoding, arithmetical and logical operations, *etc.* The duration of each cycle in an EU is modulated according to the delay of the instruction being executed—much like what is done with many asynchronous schemes—using Delay Elements (DEs). Clocks are created locally, using pulse generators, to trigger the registers in the EUs datapaths. This allows to use flip-flops as the main storage elements in the circuit, which eases the integration with standard EDA flows and test methods. The AnARM is composed of 16 EUs that are linked to one another via a crossbar

switch. The central section spanning the core comprises the register file (REGS), the memory access (LOAD/STORE), the instructions fetch (FETCH), and the branch prediction (BRANCH) sub-systems. These resources are shared among EUs, and their access are regulated using the *token-ring*, as illustrated in Figure 1.2. It serves the dual purpose of arbitrating the access to the shared resources, ensuring that a resource access is only granted to one EU at a time, and of synchronizing the transactions between resources and EUs. Instructions are issued to EUs in sequence following the program order. They are then processed out-of-order, before being completed in-order. Hence, the AnARM is an out-of-order processor with in-order instructions issue, out-of-order execution, and in-order completion.

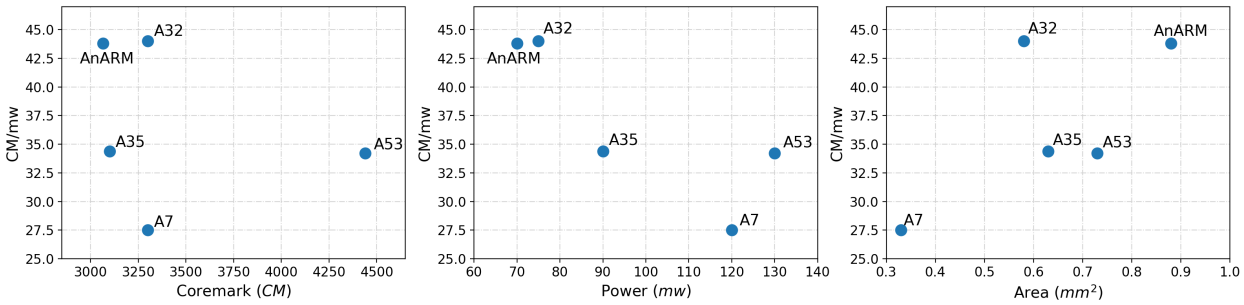


Figure 1.3 Comparison of the AnARM performance with synchronous ARM cores in the 28 nm CMOS technology node [35].

Figure 1.3 compares the AnARM with synchronous ARM cores implemented with comparable 28 nm CMOS technologies [35]. The processors are compared in terms of performance (the scores come from the Coremark (CM) benchmark), power consumption (the methods used to evaluate power are undisclosed in the references), and area. Each metric is plotted against the power efficiency (the performance-power ratio expressed in CM/mW), such that the relative ordering of the processors are always the same on the Y axis, and vary only on the X axis. The A7 (2011) [34] was the baseline when the project started. The A53 (2013) and A35 (2015) are the A7 successors [36]. Finally, the A32 (2017) is the most power-efficient ARM core in the Cortex-A family [37]. The AnARM achieves the design goal of 43,8 CM/mW. It is 1.6× more power-efficient than the A7 (27,5 CM/mW). It delivers 3 066 CM, close to the A35 and the A32, but consumes only 70 mW. It provides the same power efficiency as the A32, at a slightly lower performance/power design point. Also, it is worth mentioning that the AnARM achieves 1 806 Dhrystone MIPS (DMIPS) while consuming 80 mW (22,7 DMIPS/mW). Finally, the AnARM core occupies 0,88 mm^2, compared with 0,33 mm^2 for the A7 and 0,73 mm^2 for the A53. This area overhead is expected due to the replication of computing resources in EUs.

The performance and the power consumption results of the Opus2 and of the AnARM suggest that the Octasic asynchronous design style leads to power efficient circuits. However, the comparison of complex SoCs is always a difficult task when circuits vary in features and design intent, while being implemented with different EDA flows, and fabricated using different CMOS technologies (albeit being at the same node). In addition, the reported advantages of Octasic processors compared to synchronous alternatives are mitigated by the opacity of the methods employed to estimate the results. Thus, the superiority of the Octasic design style [35] over the synchronous paradigm in terms of power efficiency has not been properly established yet. Such a demonstration would require more reliable experimental protocols. Nevertheless, it is manifest that the Octasic original asynchronous design techniques play a central role in the power-savings strategy employed in their processors. Furthermore, the power-efficiency figures exhibited by the Opus2 and the AnARM provide a clear indication that these processors are amongst the most power-efficient in their category.

But the Octasic asynchronous design principles are somewhat unorthodox. To paraphrase Michel Laurence, author of the Opus2 publication [29], they are the result of an empirical development process, rather than a research-oriented approach. The Octasic asynchronous design style is derived from *ad hoc* solutions—that is, solutions adapted for very specific needs—to the problem of eliminating the clock-tree to reduce the power consumption of the processors. But these ad hoc design principles lack of a theoretical background that would provide the generality and the justifications required to form a design paradigm. In addition, we will make the case in the following that the Octasic design style [35] diverges from existing asynchronous design styles. In particular, it does not belong to the conventional *elastic* asynchronous paradigm. The reason is that it does not use the elastic channel abstraction—a notion discussed in section 2.1—characteristic of most asynchronous designs. Instead, it adopts new methods to exploit parallelism among processing elements (which also differ from traditional pipelining), in which the self-timed clocking mechanism is tightly coupled with the arbitration of resources access. Finally, despite relying solely on standard cells, and despite using most of the design practices that are commonly found in synchronous circuits—for example using flip-flops as the main storage elements—, the Octasic design style [35] shares the shortcomings of asynchronous circuits with respect to their limited integration with standard EDA tools. Indeed, at first glance, the use of self-timed clocks makes the Octasic design style [35] incompatible with STA engines, and timing-driven optimization algorithms, of synchronous CAD tools. This prevents standard synthesis and Place and Route (PaR) tools from performing the optimization steps that automatically size and organize the circuit based on performance requirements, and prevents the use of standard STA tools when verifying the timing of the circuits, ensuring that it is correctly working under the worst case scenario.

1.2 Objectives and Contributions

This work took place within the broader context of the AnARM project. The AnARM processor has first served as a proof of concept to showcase the Octasic ad hoc asynchronous design technique, by comparing its characteristics (performance, power consumption, area) with other general purpose synchronous processors, as discussed in the previous section, and as reported in [35]. But the AnARM also provided a context to propose a novel architecture for asynchronous caches in [38], a new model for DVS in [39], and original test methods in [40–42]. In this work, we further explore the design trade-offs of the Octasic asynchronous design style uncovered with the Opus2 and the AnARM experiments.

We have seen that the Opus2 and the AnARM processors are more power-efficient than their synchronous alternatives, although we reckon with the limitations of the methodology leading to this conclusion. Consequently, this work relies on the research hypothesis that the Octasic asynchronous design principles lead to power-efficient circuits. In this work, we aim to strengthen the causal relation between the reported design style and a reduced power consumption, compared with the synchronous paradigm. To support this goal, we propose experimental protocols based on the design of a series of asynchronous processors, derived from the Octasic asynchronous design principles, that are compared with their synchronous alternatives. In particular, we first report the design of an asynchronous Mini-Mips processor and its synchronous counterpart. They implement a small subset of the MIPS ISA, and are implemented on a Xilinx *7-series* FPGA. We then present the *KeyV* asynchronous processors, and compare their characteristics with the *SynV* synchronous alternatives. They implement the 32-bit variant of the RISC-V ISA (RV32IM) [43], and are synthesized with a 65 nm CMOS technology from TSMC. Their performances and power consumption are evaluated and compared, in simulations, using the *Dhrystone* and the *Coremark* benchmarks.

Furthermore, we have seen that the Octasic asynchronous design style relies on ad hoc principles, that were created as the result of an empirical development process. Hence, this rather unorthodox design style does not fit well within the existing synchronous and asynchronous design paradigms. In this work, we revisit the Octasic asynchronous design techniques, using circuits and methods coming from the asynchronous paradigm, with the objective of laying the foundations for a more rigorous definition of this unorthodox design style. The goal is to formally define generic design principles that can provide the theoretical background necessary to support a new paradigm. These principles should assist with the decision-making process of a design, and support the methods that enable design automation. Based on formalized design principles inspired by the Octasic design style [35] proposed in this thesis, we propose the *KeyRing* microarchitectures.

Here, it is important to emphasize that, in contrast with the out-of-order microarchitectures of the Opus2 and the AnARM, the KeyRing processors developed in this work are based on in-order microarchitectures. Indeed, as it will be discussed in Chapter 4, in the process of building the principles of the KeyRing design style, the decision was made to consider simpler microarchitectures first. Thus, the design of the KeyRing microarchitectures presented in this work result from an attempt to answer the question: How would a processor be designed, based on the Octasic asynchronous design style, if it was in-order ?

Finally, we have seen that the Octasic asynchronous design style was created with the intent to comply with common design practices. In particular, it solely relies on standard cells. However, it still has a limited compatibility with standard EDA tools, which is mainly due to its clocking mechanism conflicting with the timing analysis, and timing optimizations, capabilities of the tools. In this work, we intend to further lower the barrier with standard EDA tools. To this end, the KeyRing design style is made compatible with timing-driven optimizations, and the Static Timing Analysis, capabilities of standard EDA tools. This is achieved using a combination of circuit design techniques and EDA methods.

To summarize, the main objective of this work can be stated as follows:

- Develop a framework for the architectural exploration of processors derived from the KeyRing design style.

To achieve this main objective, we propose to complete the following secondary objectives:

- Define the formal principles behind the KeyRing microarchitecture that would serve as the basis for a generic design paradigm.
- Design KeyRing processors of reasonable—yet realistic—complexity, in combination with their synchronous alternatives.
- Develop custom design flows, and propose new circuits, that enable the timing-driven implementation and verification of KeyRing processors using standard EDA tools.
- Elaborate experimental protocols that allow to produce reproducible results and make fair comparisons between KeyRing processors and their synchronous alternatives, in terms of area, performance, and power consumption.

Each of these objectives led to the proposed solutions and methods that are claimed as contributions. Specifically, the contributions of this work are as follows:

- A deep analysis of the Octasic asynchronous design style, as implemented in the Opus2 and the AnARM processors.
- KeyRing, a generic microarchitecture derived from the Octasic asynchronous design style and novel circuit structures inspired from the asynchronous paradigm.
- A theoretical framework, based on a graph representation of the KeyRing microarchitecture, that lays the foundation for the KeyRing design paradigm, and that serves as the basis for balancing designs trade-offs, and enabling timing-driven EDA flows.
- A method to perform the Static Timing Analysis of KeyRing systems with standard EDA tools, based on the timing relations derived from the theoretical model, that enables timing-driven optimizations with synthesis tools, and timing verifications with STA engines.
- A prototyping platform enabling the comparison of KeyRing processors with their synchronous alternatives on FPGA, that can measure the power consumption of the processors while they execute a benchmark.
- An ASIC development framework enabling the comparison of KeyRing processors with their synchronous alternatives, that can evaluate the performances and the power consumption of the processors while they execute a benchmark.
- Multiple implementations of the KeyRing microarchitecture, including Mini-Mips, a simple KeyRing processor implementing a subset of the MIPS ISA, and *Mini-Spim*, its synchronous alternative, implemented on FPGA; *KeyBonacci*, a sequential KeyRing circuit implementing the Tribonacci algorithm [44]; *KeyV*, a series of RISC-V processors, and *SynV*, their synchronous (clock-gated and non-clock-gated) alternatives, implemented with a timing-driven EDA flow using a 65 nm ASIC technology.

As of the date of writing this dissertation, the following papers related to this work have been published, or are about to be submitted:

- *Self-timed circuits FPGA implementation flow* [45] published in NEWCAS (2015). This paper reports preliminary work discussed in Chapter 2.
- *A practical design method for prototyping self-timed processors using FPGAs* [46] published in ISCAS (2016). This paper reports early research that led to the implementation methodology proposed in Chapter 3.

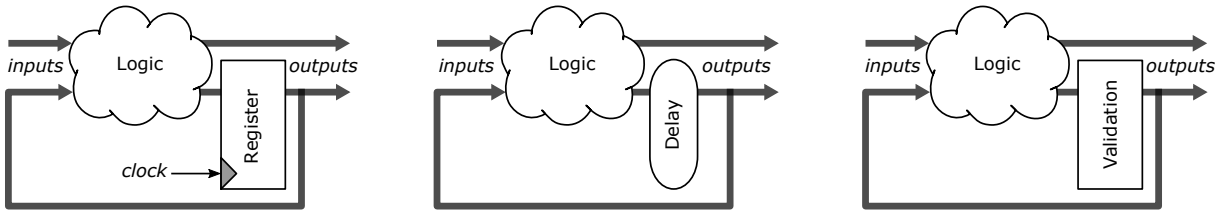
- *FPGA implementation of Token-based Self-timed processors: A case study* [47] published in NEWCAS (2017). This paper is a condensed version of Chapter 3.
- *AnARM: A 28 nm Energy Efficient ARM Processor Based on Octasic Asynchronous Technology* [35] published in ASYNC (2019). This paper reports the performances of the AnARM, most of which was reused in the introduction of this dissertation.
- *Introducing the KeyRing Self-Timed Microarchitecture and Timing-Driven Design Flow* submitted in TVLSI in November 2020. This paper summarizes the contributions and the results reported in Chapter 4 and in Chapter 5.

1.3 Work Organization

This dissertation is organized as follows. In chapter 2, a review of the literature first compares the synchronous, asynchronous, and Octasic design style [35], before reviewing state-of-the-arts methods for the implementation of asynchronous circuits using standard EDA tools. A first prototype of processor—the Mini-Mips—based on Octasic asynchronous design principles is presented in chapter 3. It targets FPGA implementation, using a first EDA method attempting to enable timing-driven synthesis and Static Timing Analysis. It is compared with a synchronous alternative, using on-board measurements of performances and power consumption. The practical experiments uncovered with the Mini-Mips are used as the basis for the KeyRing microarchitecture and theoretical model presented in chapter 4. The KeyRing microarchitecture is evaluated with a generic circuit implementing the Tribonacci algorithm. This simple circuit is then synthesized with an ASIC design flow based on the second EDA method proposed in this work, that is more suitable with timing-driven optimizations and STA of KeyRing circuits using standard EDA tools. Building on the KeyRing microarchitecture, and its associated timing-driven design flow, two instances of RISC-V KeyRing processors—the KeyV processors—are presented in chapter 5, in combination with an improved EDA flow that account for the added complexity of the circuits. They are implemented in a 65 nm CMOS technology, and compared post-synthesis with synchronous alternatives—the SynV—, with and without clock-gating, in terms of area, performance, and power-consumption. Finally, conclusions and recommended directions for future research are listed in chapter 6.

CHAPTER 2 LITERATURE REVIEW

2.1 Synchronous and Asynchronous Circuits



(a) synchronous (b) asynchronous (*burst mode*) (c) asynchronous (*speed independent*)

Figure 2.1 Main sequential circuits synchronization paradigms

Synchronous logic designs rely on the simplifying assumptions that all signals are binary, and that time is discrete. The former allows to use Boolean logic to formally describe logic constructs, and the latter provides a convenient way to deal with *hazards*. Asynchronous logic designs are also based on Boolean logic, but remove the assumption that time is discrete. The original separation between the synchronous and the asynchronous logic design paradigms comes from their fundamentally different approaches towards building sequential circuits that can accommodate hazards [48]. A hazard—an unwanted *glitch* on a signal—comes from the dynamic operation of signals due to the delays of gates and wires in a circuit. For the reliable operation of any sequential circuit, both external (inputs) and internal (feedback loops) signals must remain stable and free of hazards while outputs and internal states are being updated. Figure 2.1 shows the three main synchronizations schemes that have been adopted in sequential circuits to meet this requirement. The vast majority of VLSI circuits use a design paradigm derived from one of these synchronization schemes.

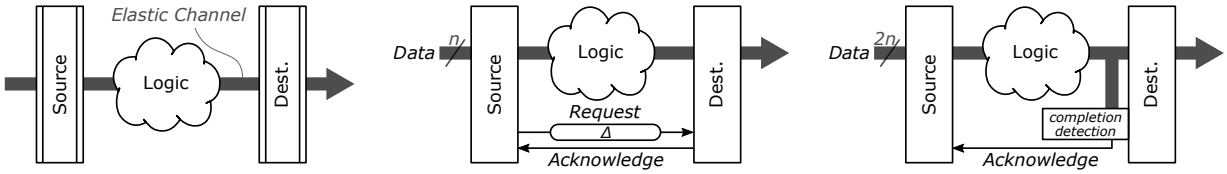
2.1.1 Synchronization paradigms

Synchronous circuits (Figure 2.1a) use registers to hold state signals, and a periodic control signal—the clock—to sample their updates. Reliable operation only requires that the signals be stable and free of hazards in a time window around the active phase of the clock, an interval characterized by the *setup* and *hold* time parameters of the registers [1]. Only the clock, and the asynchronous reset, must be kept free of hazards [49].

Asynchronous circuits can be separated in two categories [50]. The first category (Figure 2.1b) uses Delay Elements on the feedback paths to ensure that signals have settled, in response to updated states or new inputs, before updating internal states and outputs. It is based on the *bounded delay* model, which assumes that the time it takes for a circuit to reach a stable state is bounded. Reliable operation require that the signals are free of hazards, and that the DEs match the longest delay of the logic. Thus, the restrictions on the environment are formulated as an absolute time requirement. Legacy implementations of this design style—called Huffman style *fundamental mode* circuits—have important limitations. They allow only one state change by cycle, and they use one-hot encoding schemes to prevent multiple state bits to change simultaneously. Modern implementations—Burst Mode (BM) and eXtended Burst Mode (XBM) circuits—alleviate most of these limitations. The bounded delay model is an active research area which exceeds the scope of this work.

The second category (Figure 2.1c) uses validation mechanisms to enable the update of the circuit internal states and outputs. It is based on the *unbounded delay* model, which makes no assumptions about the time it takes for a circuit to settle in response to new inputs. Reliable operations require that the recipient of a signal informs the sender when it has received the information. Thus, the restrictions on the environment are formulated as causal relations between signal transitions. For this reason, these circuits are said to be *speed-independent* (they are also known as input-output mode, or Muller style) [48]. In practice, circuits that behave correctly regardless of delays are difficult to design. A practical relaxation of this requirement was proposed by Seitz in 1980 [51] and popularized in Mead and Conway [52], which states that asynchronous circuits with speed independent characteristics under local timing constraints are said to be *self-timed*. In this work, we study self-timed circuits.

VLSI circuits—both synchronous and asynchronous—can be designed following either a *semi-custom* or a *full-custom* design methodology [1, 53]. Semi-custom methodologies favor time to market and design reuse. They rely on *static* CMOS standard cells and are supported by mature EDA tools and standardized verification and test methods. Full-custom methodologies, on the other hand, trade design time for improved performances, area, and power consumption, by enabling all sorts of circuits optimizations techniques—transistor-level optimizations and manual control over the placement and the routing of each cell—that are not possible with a semi-custom approach. In particular, *dynamic logic* styles—such as domino logic, C^2MOS, or pass-transistor logic—have been largely explored in full-custom designs, especially within the asynchronous community [6, 7]. The semi-custom methodology is the preferred way of designing synchronous circuits [1], and asynchronous circuits can also be designed with this approach [20]. In this work, the focus is on asynchronous circuits that can be designed with static CMOS logic styles using a semi-custom design methodology.



(a) Abstraction (b) Bundled-Data (c) Delay-Insensitive

Figure 2.2 Asynchronous elastic channels

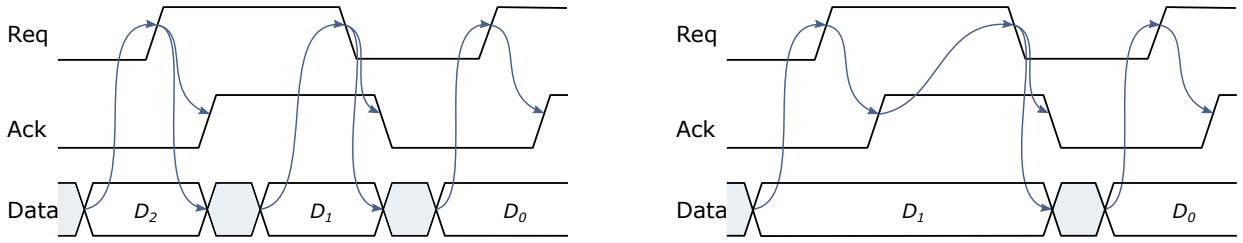
2.1.2 The elastic channel abstraction

The primary communication method used in speed-independent circuits is based on *handshaking* protocols. Basically, a sender first requests a transaction with a receiver, then, when it is ready, the receiver acknowledges back. These communication protocols are best modeled by the *elastic channel* abstraction, as represented in Figure 2.2a, and formalized in [21]. The elastic channel is composed of data lines, and additional control signals for handshaking. There are two main methods for operating transactions on the elastic channel [20]. The first method is called Bundled-Data (BD) (Figure 2.2b)—the validity of the transaction is indicated by a *matched delay*, that must be greater than the delay in the logic—, and the second method is called Delay-Insensitive (DI) (Figure 2.2c)—the validity of the transaction is directly indicated by the data using special circuitry to perform *completion detection*. Note that alternative protocols using *pulse-based* handshaking [15], or synchronous handshaking (also called *latency insensitive*) [21], have also been proposed.

BD circuits implement the asynchronous elastic channel with the coarsest level of granularity [21]. They use independent *request* (Req) and *acknowledge* (Ack) wires, with the requirement that the data must settle strictly before the request signal reaches the receiver end. In practice, this is achieved by sizing a DE on the request path such that the propagation delay of the request signal exceeds the worst-case delay of the data through the logic. Thus, BD circuits violate the strict requirements of speed independence, but meet the relaxed constraints of self-timed circuits. The validity tag provided by the request signal is analogous to the clock used in synchronous circuits, albeit it is local rather than global. The reliable operation of BD circuits requires that the data signals are stable and free of hazards in a time window around the active phase of the request signal. Hence, data encoding can be the same as with synchronous datapaths, which lowers the barrier with standard EDA flows (*i.e.* commercial synthesis tools can be used to compile the datapaths of BD circuits from high-level descriptions), and results in optimized datapaths that occupy less space, and are less power hungry than hazard-free logic.

DI circuits embed the validity of the data directly in their encoding, using the *transition signaling* convention. With transition signaling, at least two wires are required to transfer a data bit, with the wire on which a transition occurs determining the value being transmitted. The simplest form of DI encoding is called *dual-rail* (as opposed to the *single-rail* encoding used in BD and synchronous datapaths), because it uses two wires for each bit. That is, a legal codeword is either 01 (indicating that a 0 is transmitted) or 10 (indicating that a 1 is transmitted). Alternative DI codes (*e.g.* 1-of-4, m-of-n, *etc.*) have been proposed to explore the trade-offs in coding efficiency, area overhead, and dynamic power [15]. In the absence of a clock, or a dedicated request signal, the validity of the data at the receiver end is determined by using special circuitry dedicated to completion detection. The reliable operation of DI circuits relies on hazard-free logic, which should prevent the completion detection circuitry from issuing false positives. Note that *pure* DI circuits are very limited due to the strict requirements resulting from the unbounded delay assumptions. Practical implementations of DI circuits—called Quasi-Delay-Insensitive (QDI)—rely on the relaxed timing assumption that the wires at each fanout point must have roughly equal delays (this is called the *isochronic fork* assumption). DI circuits implement the elastic channel with the finest levels of granularity [21]. Regardless of the propagation delay and skew of the data lines, the receiver can unambiguously identify when every bit is valid by sensing the completion of the data on each bit. As a result, DI circuits deliver self-regulated performances that can dynamically adapt to the chip operating variability, and provide great resilience to PVT variations. However, the need for encoding the data using dual-rail hazard-free logic creates an area overhead compared to BD circuits, and prevents DI circuits from being compatible with standard synthesis tools and conventional verification and test methods [20].

The elastic channel can be implemented by two common handshaking protocols—*2-phase* and *4-phase*—as illustrated in Figure 2.3 [48]. In the 2-phase protocol—also known as Non Return to Zero (NRZ)—the Req line toggles to initiate a transaction, followed by a toggle on the Ack line to validate the transaction. In the 4-phase protocol—also known as Return to Zero (RZ)—the Req and Ack signals are initially at zero. The sender initiates a transaction by asserting the Req line, and the receiver responds by asserting the Ack line. Then, both Req and Ack are deasserted to validate the transaction. Both protocols are widely used, regardless of the data encoding (BD or DI) used to design the circuit (although the 2-phase protocol is generally used in BD circuits, and efficient QDI circuits use a 4-phase protocol) [14]. The 4-phase protocol benefits from reseting the control lines to a unique state after each transaction, which allows completion detection to be done with *level sensitive* circuitry. By contrast, the 2-phase protocol requires more complex *transition sensitive* circuitry to be able to initiate a transaction in both the rising and the falling edges of the control lines. On the other



(a) 2 phases (b) 4 phases

Figure 2.3 Handshaking protocols

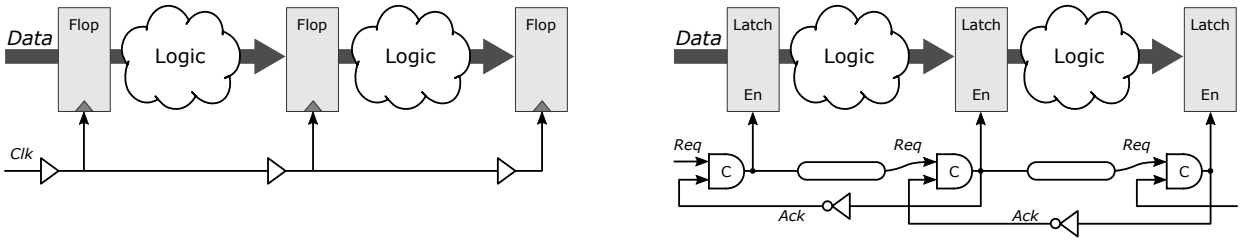
hand, the 4-phase protocol requires two round-trip communications per transaction, while the 2-phase protocol only requires one round-trip communication per transaction [15].

The various implementations of the handshake protocols, and the encoding scheme used in the circuit, result in asynchronous circuits that are radically different (*e.g.* a dual-rail 4-phase QDI circuit *vs.* a single-rail 2-phase BD circuit). Yet, when looking at the circuits from an abstract perspective—the elastic channel abstraction—it appears that the choice of handshaking protocol and circuit encoding style can be viewed as implementation decisions that are made independently from the overall structure and operation of the circuit [48]. This characteristic—strengthened by a recent work showing that the main asynchronous templates are interchangeable under *naturalized communication* [54]—, combined with the observation that most asynchronous circuits rely on an elastic channel [7, 20], makes the elastic channel abstraction the main asynchronous design paradigm.

2.1.3 Pipelining

Pipelining is a fundamental design technique used in sequential circuits to increase the level of parallelism and improve the throughput, and it is widely adopted in all modern VLSI circuits [6]. The basic principles of pipelining are straightforward, as illustrated in Figure 2.4: complex logic blocks are subdivided into smaller blocks, separated by registers. Using the synchronous paradigm (Figure 2.4a), the registers of the pipeline—typically flip-flops—are controlled by a clock. Using the asynchronous paradigm (Figure 2.4b), by contrast, the interaction between neighboring registers—typically latches—are coordinated by elastic channels.

In a synchronous pipeline, the data advance from one stage to the next on the active phase of the clock. On first approximation, all stages transfer their data to their neighbor at the same time (in reality, uncertainties induced by the clock distribution network, represented by buffers in Figure 2.4a, alter the relative arrival time of the clock edge to the registers [55]).

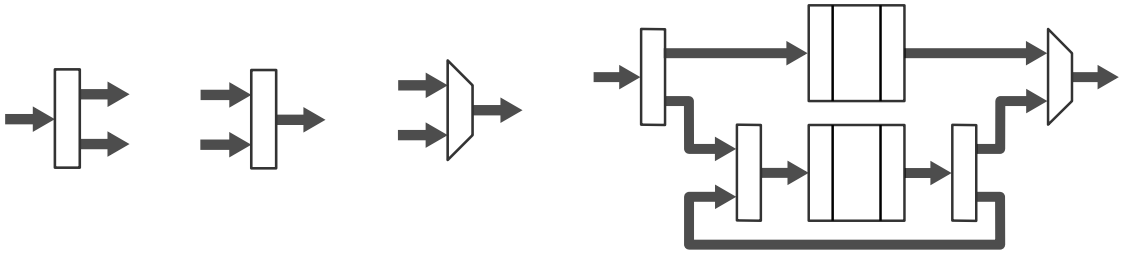


(a) synchronous (b) asynchronous (Bundled-Data)

Figure 2.4 Simple linear pipelines implementations

The longest path delay through a stage logic must be inferior to the clock period to ensure the correct operation of the system. Hence, a properly designed synchronous pipeline typically has stages with *balanced* delays. In an asynchronous pipeline, the transport of data from one stage to the next is orchestrated by the elastic channel on a per-stage basis: stage i can accept new data if stage $i-1$ is available, and if stage $i+1$ has finished its computation. Hence, stages of widely *unbalanced* delays can be assembled without severely impacting the system latency and throughput. In addition, the elastic channel gives the ability to asynchronous pipelines, unlike their synchronous counterparts, to dynamically adjust the amount of active stages as a function of inputs rates. Thus, the throughput changes dynamically. Consequently, the switching activity and the dynamic power consumption resulting from processing data in stages are generated on demand, only when a computation is needed. From this perspective, the handshake controllers achieves results similar to fine-grain, built-in, clock gating.

Figure 2.4b illustrates the basic principles of asynchronous pipelines based on a simple 4-phase BD implementation called the Muller pipeline [48]. The handshake controller relies on a special gate—the Muller *C-element*—to orchestrate the handshake protocol. The C-element is a state-holding element, similar to a latch, which outputs a 0 when both inputs are 0, a 1 when both inputs are 1, and holds its previous state otherwise. As a result, a $0 \rightarrow 1$ transition on the output indicates that both inputs are 1, and a $1 \rightarrow 0$ transition indicates that both inputs are 0. This characteristic makes of the C-element a fundamental component for the implementation of handshaking, which involves cyclic acknowledgement of $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions. In the Muller pipeline, local control signals generated by handshaking drive the *enable* inputs of the latches in a carefully controlled interlocked manner. When the pipeline is full, the states of the C-elements are $(1, 0, 1)$. Thus, only every other latch is storing data (similar to a synchronous circuit using one-phase level-sensitive latch based clocking). We review asynchronous pipelines that achieve better performances than this basic example in the following section.



(a) fork (b) join (c) merge (d) non-linear pipeline

Figure 2.5 Non-linear asynchronous pipeline components and example implementation

Figure 2.4 represents simple *linear* pipelines, that is, pipelines having the behavior of First In First Out (FIFO) circuits. To build complex circuits involving iterative computations—*i.e. non-linear* pipelines—the basic asynchronous channel must be enhanced with additional components [48]. Figure 2.5 shows an example of such non-linear asynchronous pipeline based on the elastic channel abstraction. It uses the basic components *fork*, *join*, and *merge*. Fork (Figure 2.5a) and join (Figure 2.5b) components allow to handle parallel threads of computation. A fork component is used to connect the output of one elastic source to the input of several elastic destinations, and a join component is used in the opposite case, when several independent channels need to be synchronized. A merge component (Figure 2.5c) synchronizes multiple, mutually exclusive, input channels to one output channel. Other components exists, such as *mux* and *demux*, *arbitration*, as well as complex function blocks [48].

Note that, in such non-linear asynchronous pipelines, the local handshaking allows to organize the flow of data in many different ways—a characteristic that is often referred to as the *composability*, *modularity*, or *flexibility* of asynchronous (elastic) circuits. Each asynchronous design style that uses the elastic channel abstraction must implement these basic components. In particular, we will see that the asynchronous design styles that aim to be compatible with a standard EDA flow must provide a design for these components based on standard-cells, and must integrate them as part of an implementation and verification flow.

2.2 Bundled-Data Asynchronous Design Templates

An asynchronous system can be viewed as a group of basic blocks—called *templates*—that communicate with each other through elastic channels [54]. This view is best adapted for BD circuits, in which the template—composed of a storage element and a handshake controller—is placed around the datapath. The template encapsulates the design constraints

(*e.g.* the timing constraints) to meet the requirements set by its environment, much like with a synchronous design. The first advantage of template-based design is the ease of manual design using a data-flow approach: once the template is designed, the rest of the circuit can be built by assembling them with standard combinational logic. The second advantage of template-based design is the improved compatibility with synchronous CAD tools. In fact, this approach is the preferred way of designing asynchronous circuits with standard EDA flows [7]. Many templates have been proposed in the literature. We first present one of the most influential, *Micropipeline*, uncovered by Sutherland in its 1989 Turing Award lecture [56]. Then, we present *Mousetrap* [57], and *Click Elements* [58], which are the most compatible with a synchronous-oriented design flow.

2.2.1 Micropipeline

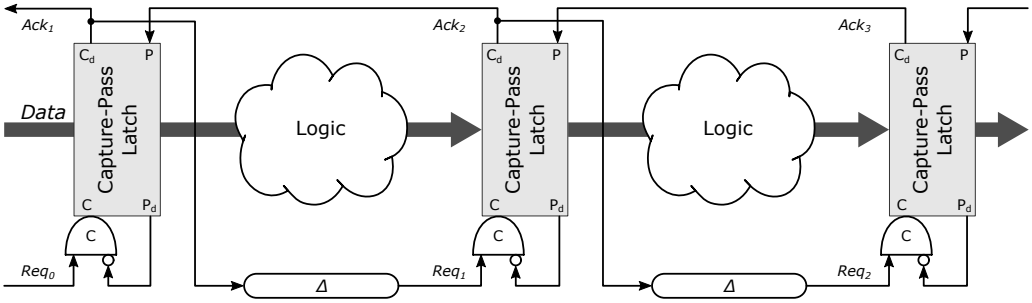


Figure 2.6 Micropipeline

Sutherland's Micropipeline [56] is illustrated in Figure 2.6. It uses a 2-phase BD handshaking protocol with a DEs matching the worst-case delays of the single-rail combinational logic blocks. The storage elements are event-controlled registers, called *Capture-pass Latches*, which respond symmetrically to rising and falling transitions on their inputs (C: *capture*, C_d: *capture done*, P: *pass*, P_d: *pass done*). The pipeline operates according to the *capture-pass* protocol, derived from the Muller pipeline (Figure 2.4b). Initially, all signals are 0, and the latches are all transparent. New data entering the pipeline first arrive at the leftmost register and pass directly through all the channels. Hence, the Micropipeline initially forms a flow-through combinational path [6]. At each stage, the Req signal toggles two events. First, a $0 \rightarrow 1$ transition on the C input of the latch—called the *forward synchronization*—makes the latch opaque, thereby capturing the data. Then, a $0 \rightarrow 1$ transition on C_d output of the latch—called the *backward synchronization*—occurs after an internal delay, indicating the completion of the operation to the predecessor latch trough the Ack signal. This event induces a $0 \rightarrow 1$ transition on the P input of the predecessor latch, effectively making it

transparent again. Consequently, a $0 \rightarrow 1$ transition on the P_d output of the latch occurs after an internal delay, which half-enables the C-element, making it ready to receive a new request. This 2-phase protocol works similarly with $1 \rightarrow 0$ transitions. Note that the internal delays ($C \rightarrow C_d$ and $P \rightarrow P_d$) are sized to satisfy the hold time requirement of the latch.

The Micropipeline publication triggered a resurgence of research activity in asynchronous designs, and paved the way for future BD circuits [15]. Although Micropipelines require specialized components—C-elements and capture-pass latches—that are typically not part of design kit standard-cells, their Bundled-Data operation allows registers to filter out logic hazards within the combinational logic, much like a synchronous circuit. Thus, standard synthesis tools can be used to synthesize combinational logic from high-level descriptions, which is the first step towards designing asynchronous circuits using a synchronous-oriented design flow.

2.2.2 Mousetrap

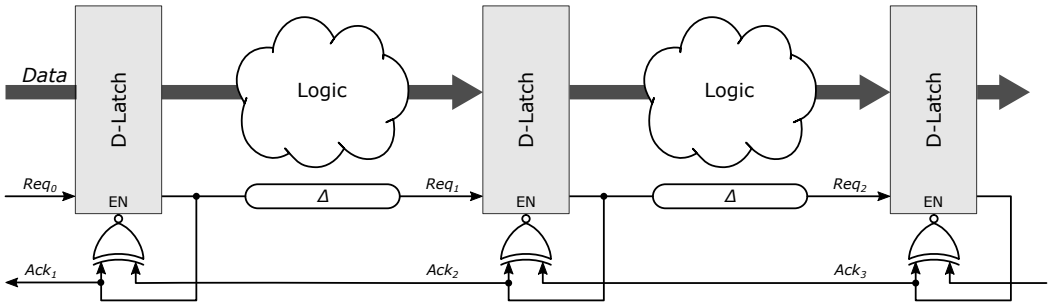


Figure 2.7 Mousetrap

Mousetrap was developed at Columbia University in 2001, with the goal of designing a high-performance BD asynchronous pipeline that could support the use of a standard cell methodology [57, 59]. Indeed, this BD asynchronous template uses single-rail combinational logic and standard D-latches as storage elements, that are synchronized with a 2-phase handshake protocol implemented with simple control circuits based on XNOR gates, as shown in Figure 2.7. The capture-pass protocol—forward and backward synchronization—is similar to that of Micropipeline, but uses simpler signaling. Initially, all signals are at 0, and thus all latch *enable* signals (EN) are at 1. Similarly to Micropipeline, all latches are initially transparent, forming a flow-through combinational path. When new data arrives at the leftmost register, it passes through all the channels. At stage i, the Req_i signal toggles (e.g. $0 \rightarrow 1$) and travels along the DE matching the delay of the longest path through the logic. When it reaches stage $i+1$, it first passes through a 1-bit latch (represented in Figure 2.7 as part of

the data D-latches), and then triggers a $1 \rightarrow 0$ transition at the output of the XNOR gate, thereby making the D-latches opaque and preventing data to be overwritten. This latched Req_i signal is used as both the Ack_{i+1} signal that acknowledges the transaction with stage i, causing the D-latches of stage i to become transparent again, and as the Req_{i+1} signal that request a new transaction with stage $i+1$. This effectively indicates that stage i has safely stored new data, and that it is ready to overwrite the data of stage $i+1$.

The correct operation of a Mousetrap pipeline depends on a hold constraint ensuring that stage i has fully captured its current data before stage $i-1$ overwrites them. This constraint is easily met by adding sufficient delay to the Ack_i signal. The relatively lightweight control circuitry allows the pipeline to achieve high throughput [6]. Although it relies on single D-latches separating adjacent stages, every stage can hold a distinct data item, thereby providing 100 % storage capacity, similar to what can be achieved in a synchronous design with a two-phase non-overlapping clocking scheme. This asynchronous design template has been very influential as it provides a simple way of implementing asynchronous BD pipelines using standard synchronous combinational logic, and handshake controllers based on logic gates that can be found in any standard-cell libraries (*i.e.* D-latches and XNOR gates). However, this standard-cell methodology is only applicable for linear pipelines. Indeed, more complex asynchronous structures required for non-linear pipelines organizations—fork, join, merge, *etc.*—still require C-elements [57]. Mousetrap was used in a variety of research projects, including a GALS multiprocessor implemented in 90 nm that can achieves up to 10× better energy efficiency with comparable performances than globally synchronous alternatives [60].

2.2.3 Click Elements

The Click Elements asynchronous template was introduced in 2010 as part of a new asynchronous design flow developed by Philips [58] (although it was first released in a 2009 patent [61]). It implements a 2-phase handshake protocol with a Bundled-Data encoding, using only logic gates and flip-flops that are available in any standard-cell libraries. In contrast with Mousetrap, the Click template uses edge-triggered flip-flops for both control and data, with the intended goal of being more in line with the conventional way of designing synchronous circuits. Figure 2.8 shows a Click implementation of a BD linear asynchronous pipeline. Initially, all signals are at 0, and the flips-flops are opaque. In contrast with the previous two templates, the Click pipeline is not initially a flow-through combinational path; in this respect it resembles a synchronous pipeline. The intent is to save energy by preventing data from rippling through the pipeline prematurely [54]. When the Req_i signal toggles (*e.g.* a $0 \rightarrow 1$ transition) at stage i, it initiates the rising edge of the clock at the output of the

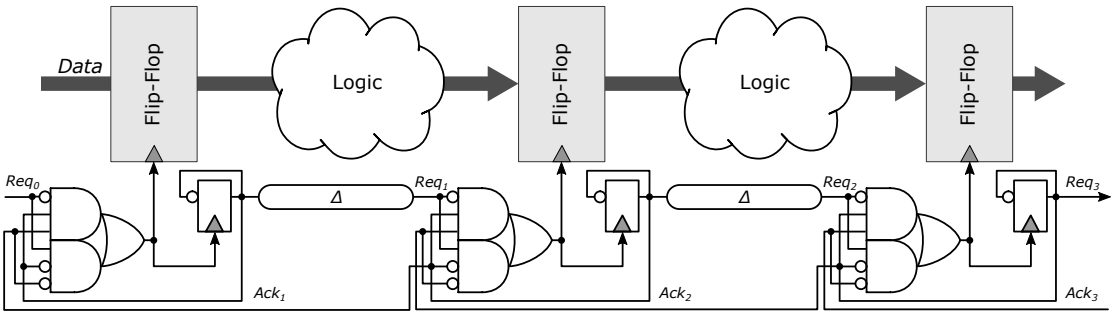


Figure 2.8 Click Elements

phase conversion circuitry—composed of two AND gates and one OR gate—, which triggers the control flip-flop holding the state of the handshake controller, as well as the registers on the datapath. The controller state is used in three places: *i)* it forms the Req_{i+1} edge that travels along the DE matching the delay of the stage $i+1$ combinational logic; *ii)* it is used through the feedback path of stage i, in the phase conversion circuitry, to initiate the falling edge of the clock; and *iii)* it is used as acknowledgement for stage $i-1$ through the Ack_i signal. The correct operation of this circuit relies on the generation of well defined clock pulses. This implies that the clock pulse width be larger than the minimum pulse width defined in the technology library—it can be checked by STA, and enforced by sizing the gates of the phase conversion circuit—, and that hazards in the handshake controller be avoided. Since all handshake signals are driven by flip-flops, it is safe to consider them glitch-free [58].

The use of an edge-triggered flip-flop on the control feedback loop of the Click circuit, instead of a D-latch in Mousetrap and a C-element in Micropipeline, facilitates timing analysis and eases the automatic insertion of test structures using standard CAD tools [58]. In contrast with Mousetrap, the Click template contains an implementation of each basic block for non-linear elastic pipelining—i.e. fork, join, merge—that solely relies on standard-cells. Since it was uncovered, the Click template is the preferred choice for implementing asynchronous BD circuits using standard EDA tools and synchronous FPGAs [62]. Most notably, it was used in the design of a 60 mm^2 neuromorphic manycore processor, implementing a spiking neural network with on-chip learning, that was fabricated in Intel's 14-nm process [63]. Interestingly, the basics of the Click template, as illustrated in Figure 2.8, were fist proposed in a 1995 Intel patent [64], and very similar circuit structures were proposed in a 2008 work from the University of British Columbia, for the implementation of an asynchronous QDI interconnect compatible with the standard EDA flow [65].

2.3 Octasic *ad hoc* Asynchronous Design Style

In this section, we uncover the general principles of the Octasic asynchronous design style based on an analysis of the AnARM processor. The AnARM was introduced in a 2019 ASYNC publication [35], but its underlying design principles are the same as the Opus2 DSP, which were first reported by Laurence in 2012 [29]. We present the AnARM architecture with an emphasis on parallelism and resource sharing. First, we try to place the AnARM within the asynchronous paradigm, as defined in Section 2.1. Then, we present the AnARM as an out-of-order processor, which does not rely on pipelining to exploit parallelism among instructions. This analysis is the basis from which we elaborate the principles of the KeyRing design style in Chapter 4.

2.3.1 Overview

The AnARM is composed of functional modules, called Execution Units (EUs), and shared resources, such as a Register File (RF), a Program Counter (PC), a Load/Store Unit *etc.* as depicted in Figure 2.9. EUs operate asynchronously from one another; the overlapping of instruction execution in different EUs allows to exploit parallelism in a program sequence. EUs and shared resources communicate between each other through the Crossbar Switch (CS). In contrast with elastic circuits, the CS in the AnARM does not include any synchronization mechanism. Shared resources may be synchronous, in which case they are interfaced with the CS using FIFOs (not represented in Figure 2.9). In the AnARM, this is the case for example for the Instruction Fetch and the Load/Store units, which include caches that may require many cycles to be accessed. Note that self-timed caches were developed for the AnARM [38], but they were not included in the final version [35].

Similar to most processors, instructions in the AnARM are decomposed in steps (*Fetch*, *Decode*, *Execute*,...), and parallelism among instructions is exploited by overlapping the execution of multiple instruction steps. The number of instructions that can be executed simultaneously defines the level of parallelism that a processor can achieve. Microarchitectural implementations of Instruction Level Parallelism (ILP) usually rely on *pipelining*, where a pipeline refers to a sequence of stages composed of the logic associated with an instruction step followed by a memory element, as presented in Section 2.2. A new instruction may enter the pipeline as soon as the first stage is available. In the case of single-issue in-order execution, ILP is at its maximum when each stage of the pipeline processes a step of different instructions. By contrast, the AnARM exploits ILP by overlapping the execution of multiple instructions in different *multicycle* EUs (similar to the *multicycle processor* described by

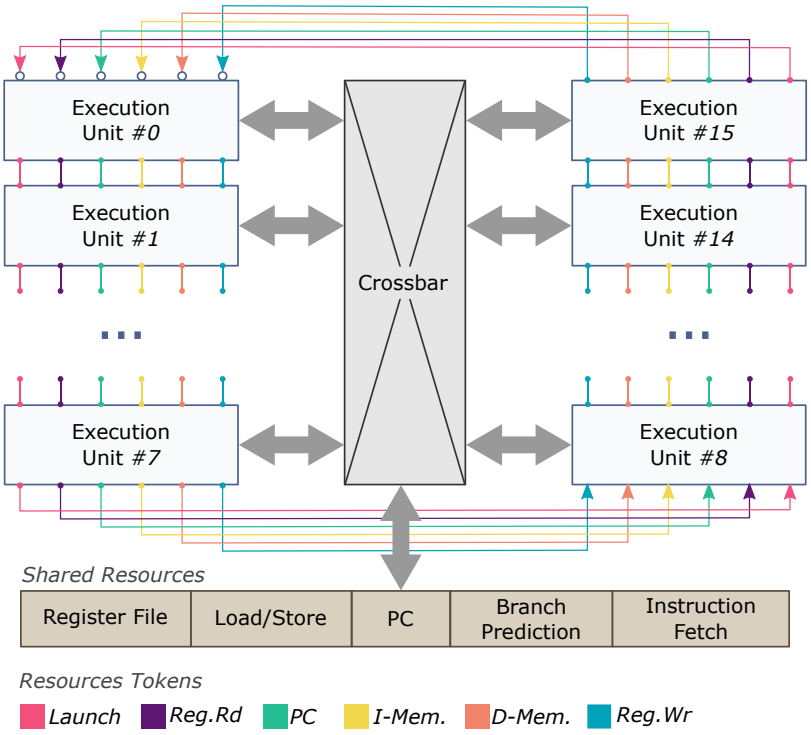
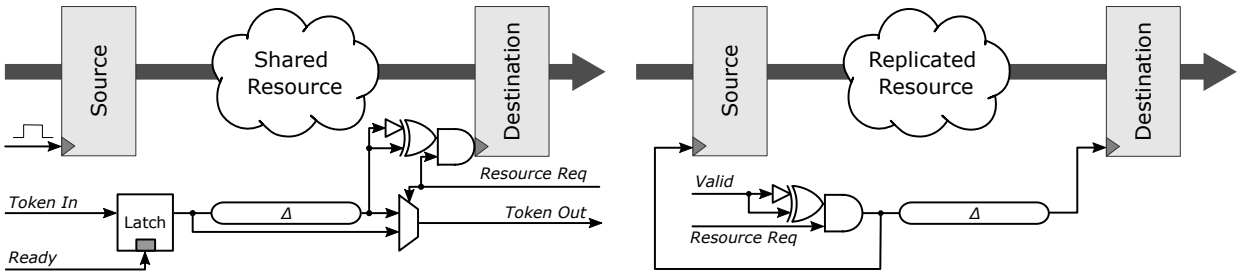


Figure 2.9 AnARM processor top-level organization

Hennessy & Patterson [5]). Each self-timed stage is a basic processing element as depicted in Figure 2.10. A new instruction may enter the EU only when the last stage has completed its execution. Hence, each EU contains a maximum of one instruction at any given time. In the case of single-issue in-order execution, ILP is at its maximum when each EU is processing one instruction. This organization implies the replication of the logic stages for each EU, which creates an area overhead, as reported in [35].

2.3.2 Instruction Level Parallelism

Exploiting Instruction Level Parallelism alters the normal execution flow of a program by exposing data dependencies and resources access order between nearby instructions to the hardware [5]. In the AnARM, structural hazards that would arise from concurrent access to shared resources by multiple EUs are addressed using *Resources Tokens*. In Octasic's terminology, a *token* is a signal edge that propagates through EUs to arbitrate the access to shared resources [29]. In Figure 2.9, tokens are represented by colored arrows, and their associated resources are indicated in the legend. The principles of arbitration with resources tokens in the AnARM are as follows [31]: when a token enters an EU, if the EU requests a transaction with the associated resource, it is retained in a DE until the transaction completes.



(a) (b)

Figure 2.10 Typical EU stages used with (a) shared resources, and (b) replicated resources

This is called *consuming the token*. If the EU does not request the transaction, the token is directly released. A token travels along the EUs in sequence, as shown in Figure 2.9, thereby granting access to its associated resource one EU at a time. After a token is released by the last EU, it is inverted before looping back to the first EU. This organization of resources tokens in *rings* serves the dual purpose of granting access to shared resources and of synchronizing the transactions between a resource and the associated EU stage.

Figure 2.10 represents the architecture of typical EU stages in the AnARM. The first circuit template (Figure 2.10a) is used with *shared resources* to compute instructions *in-order*, using the token-based synchronization mechanism. A token is being consumed (*i.e.* a resource access is granted) as long as it propagates through the DE, which matches the delay of the shared resource access path. The *capture* clock, generated from the delayed token, triggers the destination register. If an EU does not request access to a resource (*Resource Req.* signal is low), the DE is bypassed. The source register may represent the destination register of a resource, or the destination register of another EU stage. It is triggered by the *launch* clock, generated either by the same token coming from a different EU, or by another token coming from a different stage. The second circuit template (Figure 2.10b) is used with *replicated resources* to compute instruction *out-of-order*. Replicated resources can be operated concurrently in multiple EUs. Hence, the AnARM is an out-of-order processor with in-order instructions issue, out-of-order execution, and in-order completion. In contrast with shared resources, these replicated resources have no need for access arbitration, and thus have no need for dedicated tokens. Instead of matching the datapath with a delayed token signal, it is matched with a delayed pulse signal used for both the launch and the capture clock [29]. In practice, replicated resources in the AnARM are ALUs. They can be used as soon as the operands they need to operate are ready.

Let us consider the timing conditions that should be valid for the correct operation of a

typical EU stage (Figure 2.10a). In the case where both the launch clock and the capture clock are generated by the same token (assuming that the latch is transparent), the setup constraint has the same definition as in BD circuits (see Section 2.5.1). However, in the case where the launch clock is generated by a token different from the capture clock, the setup constraint may vary by an unpredictable amount equal to the arrival time difference between the two tokens. The role of the latch in the controller, controlled by the *Ready* signal, is to block the input token until the source register is triggered, which ensures that the setup constraint does not change. Interestingly, hold conditions, which in BD circuits are associated with the backward propagation of acknowledgement signals, have no direct correspondence in an EU stage of the AnARM. Indeed, because EUs are multicycle, stage_{i+1} can accept data from stage_i without acknowledgement signal. Thus, we could conclude that there are no hold constraints to be considered in the Octasic asynchronous design style. However, we will show in Chapter 4 that hold conditions do exist, and we will explicit their definitions.

The AnARM microarchitecture does not belong in the elastic design paradigm. Indeed, although the basic processing elements used in the AnARM, as shown in Figure 2.10, resemble BD circuits—they both use a single-rail data encoding scheme and storage elements are timed with self-generated clocks, with cycles duration adjusted as a function of the computations delay using Delay Elements—, at the system level they do not use the elastic channel abstraction. As previously stated, this *ad hoc* design style was developed following empirical design principles stemming from an attempt to reduce power consumption by eliminating the global clock, rather than trying to fit in any design paradigm [29]. Consequently, we define the AnARM microarchitecture as being simply self-timed, in accordance with the broad definition presented in Section 2.1.

In the AnARM, the clocking mechanism and the ILP organization are tightly coupled. Indeed, each stage in an EU is controlled by a different token, and their ordering defines the order in which instruction steps are executed. For example, updating the PC (*PC* token) must be performed prior to accessing the instruction memory (*I-Mem* token). Similarly, a new instruction cannot be launched (*Launch* token) before the register write operation of the previous instruction in the EU is completed (*Reg.Wr* token). *Ready* signals distributed across EUs stages allow to order resources tokens by controlling their propagation in the EUs through the latches. These control signals, generated by the EUs, carry the information that one or more stages has completed its execution. Hence, we can deduce the conditions for initiating a transaction in an EU stage: *i)* the associated resource token has arrived; meaning that a transaction involving the same resource in the previous EU has finished, and *ii)* the *Ready* signal is enabled; meaning that transactions involving dependent tokens in the same EU have finished.

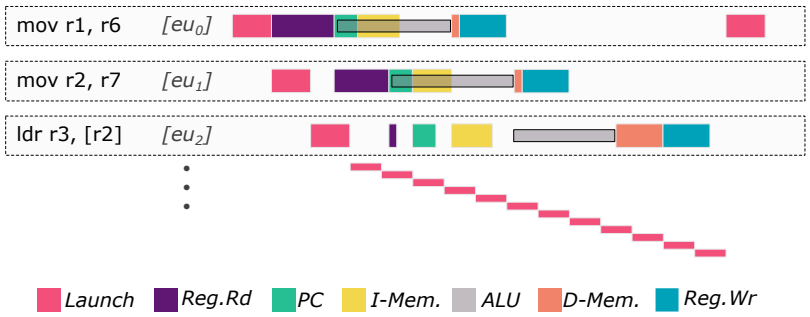


Figure 2.11 Illustration of Instruction Level Parallelism in the AnARM

Figure 2.11 shows how ILP is achieved by allocating instructions steps in EUs stages. A given EU processes the i^{th} instruction modulo N, where N represents the number of EUs. Colored rectangles represent instruction steps, with their length representing the stage delay, and their color representing the associated token. Notice that delays vary from one stage to another—much like with BD pipelines—with very short rectangles representing the case where a stage is bypassed. Horizontally, steps are processed in a sequence that follows the dependency of tokens in an EU (*i.e*, *Launch* $\rightarrow$ *Reg.Rd* $\rightarrow$ *PC* ...). Vertically, steps are processed in a sequence that follows the propagation of tokens across EUs (*i.e*, *Launch*: $\text{EU}_0 \rightarrow \text{EU}_1 \rightarrow \text{EU}_2$ ...). The *Execution* steps, which rely on replicated ALU resources (not associated with a token), can be completed out-of-order; the *Execution* step of the i^{th} instruction can finish before the *Execution* step of the i^{th}_{-1} instruction is completed. Results are written back in the RF in-order using the *Reg.Wr* token. Data dependencies between nearby instructions are handled by preventing the *Execution* step in the processing EU to start before the operands are ready, and by forwarding data between all EUs using the CS as soon as results are available. For example, Figure 2.11 shows that the third instruction depends on data coming from the previous two instructions. The operands come from the CS, thus the *Reg.Rd* token is bypassed and the *Execution* step does not start before the last *Execution* step is completed.

In contrast with existing literature [29–31], this analysis has emphasized the different levels of parallelisms in the AnARM. The first level of parallelism (in-order) is achieved by concurrently operating *multicycle* EUs with shared resources. The second level of parallelism (out-of-order) is achieved by *replicating* resources in EUs. We use this original point of view to build the KeyRing microarchitecture in Chapter 4.

2.4 Design Automation

We have seen that the Octasic asynchronous design style uses circuits similar to those found in BD asynchronous designs. In addition, it relies on a semi-custom design methodology based on standard-cells that aims to be compatible with the standard EDA flow. However, the Octasic design style [35] does not support timing-driven synthesis and Static Timing Analysis, which are the backbone of the standard design flow, as we will see in this section. Indeed, Octasic circuits are mainly designed at the gate level, and timing verifications are performed using a custom simulation environment [29]. An important objective of this work is to propose new circuits and methods to lower the barrier between the Octasic design style [35] and the standard EDA flow. The key to achieve this goal is timing. In the following, we review state-of-the-art methods for the integration of asynchronous circuits with the standard design flow, with an emphasis on Static Timing Analysis.

2.4.1 The standard synchronous design flow

The inherent simplicity and efficiency of the synchronous paradigm enables common design practices and favors design reuse, which is particularly well adapted for the semi-custom design methodology based on standard-cells. The benefits of the synchronous paradigm are responsible for its widespread adoption, and have lead many to advocate for a KISS (*Keep It Strictly Synchronous*—derived from the well known *Keep It Simple, Stupid*) approach [1]. First, hazards, races, metastability, and noise issues are all addressed by having the clock sample the registers at fixed time intervals [49]. Because there is no need for redundant circuitry to suppress hazards, synthesis algorithms can be simpler, and datapaths can occupy less space and consume less power. Then, synchronous operations allow to separate the *functional validation* from the *timing verification*. Functional validation consists in checking the correct behavior of the system against its specification at the algorithmic level—*i.e.* without taking delays into consideration—using cycle-based simulation. On the other hand, timing verification consists in exhaustively checking that the timing constraints of the circuits are met—*i.e.* ensuring that the signals through logic blocks have the time to settle within the time intervals set by the clock period and the delays of the clock distribution network—using Static Timing Analysis. Moreover, this strict separation between functionality and delay of the logic enable timing-driven synthesis. That is, optimization algorithms can size and organize the gates of the datapath to meet the timing constraints without altering its behavior. Finally, synchronous circuits are simpler to test. Today, established methods for circuit testing (fault grading, automatic insertion of test structures, and automated test vector generation), including test equipment, suppose synchronous operations [1].

2.4.2 Asynchronous design flows

The key aspects of the standard EDA flow—semi-custom methodology based on timing-driven synthesis of high-level models mapped to static CMOS standard cells, decoupling of the functional validations from the timing verifications, and testability—are, at first glance, incompatible with asynchronous design methods [1]. As we said, these incompatibilities come from the fundamental differences between the synchronous and the asynchronous timing approaches (cycle-based *vs.* event-driven), and the dominance of the synchronous paradigm which shaped the development of EDA [7].

Standard synthesis tools and HDLs cannot be used to compile automatically an asynchronous high-level description into an optimal RTL netlist [20]. Indeed, they cannot deal with hazard-free logic and elastic channels communication protocols, and they cannot map these channels to an appropriate handshake circuit template. For instance, standard synthesis tools do not support the compilation of C-elements from high-level descriptions, and most design kits do not provide them as part of their standard-cell libraries. The alternative is either to design and map the C-element manually—static CMOS C-element cell designs have been studied [66], and FPGA implementations have been proposed [67, 68]—, or to design control circuits based on standard-cells that fill its role—this is the goal of the Click Elements [58] and the Mousetrap [57] BD asynchronous templates, and it was also done for QDI circuits [65].

In addition, the functional validation and the timing verification of asynchronous circuits are combined, because the functionality of the elastic channel is tightly coupled with its timing. The first consequence is that behavioral simulation must take timing into account. Rather than fast cycle-based behavioral simulations, asynchronous circuits rely on slower event-driven simulations (simulation issues are further discussed in Chapter 4). The second, and most important, consequence is that the level of compatibility between standard STA engines and asynchronous circuits is low. Thus, instead of using STA tools, timing verifications are often performed in simulations, where, in contrast to STA, the exhaustive check of delay races in the circuit is a difficult and time consuming process. As a result, timing closure is longer to achieve and less reliable. Moreover, the timing-driven synthesis of asynchronous circuits with standard EDA tools—*i.e.* optimizing the circuits to meet performance targets defined as timing constraints—is also hindered by this native incompatibility.

There are two main avenues for automating the design of asynchronous circuits [15, 26, 27]. The first relies on alternative EDA flows—dedicated synthesis and verification tools—, and the second relies on the standard EDA flow, either with conversion algorithms, or with circuits and methods that address asynchronous circuits incompatibilities with the synchronous flow.

At the elastic channel abstraction level, asynchronous circuits are often modeled by Petri nets [69]—directed graphs describing concurrency and choice—, in particular Signal Transition Graphs (STGs) [20]. Rather than representing the system as a succession of states, this model represents asynchronous circuits as a partially ordered sequence of events. The most widely used tool for synthesizing asynchronous control circuits from their STG specifications is called *Petrify* [70]. It was developed by the *Universitat Politècnica de Catalunya* since 1999, and was released in the public domain. Earlier and influential design flows were the *Caltech Synthesis Method* and the *Philips Tangram tool flow*. The former was developed in the 1980s to produce QDI circuits—including the MiniMIPS processor [22]—from high-level models specified in the Communicating Sequential Process (CSP) language. The latter—used industrially by Phillips during the 2000s—produces BD circuits from high-level descriptions, also in CSP, with extensive support for testability. In 2010, it evolved into the Haste language and compiler, in combination with the Click Elements design template, and a novel design flow that aims to improve testability and support by synchronous tools [58]. It is also worth mentioning Balsa [71], a public-domain version of the Tangram compiler. During the same period, Tiempo developed an asynchronous design and verification flow, based on a dedicated synthesis tool called Asynchronous Circuit Compiler (ACC), which uses SystemVerilog descriptions (with some extensions to model elastic channels) to compile QDI circuits [20]. Finally, a promising avenue for asynchronous automation is WORKCRAFT, which provides a framework for the specification, simulation, synthesis, and analysis of speed independent asynchronous circuits [72]. In addition, it provides an extensible front-end that can be interfaced with a variety of existing tools, such as Petrify.

An alternative approach to using high-level representations with dedicated synthesis tools, is to design part of the circuit manually and use standard synthesis tools to create the rest of the netlist [26]. This approach is well adapted for BD asynchronous templates, in which the combinational logic can be compiled from high-level HDL descriptions, but it was also employed with Delay-Insensitive circuits [65, 73, 74]. With the template-based approach, two categories of methods are being explored.

The first category of methods—called the *desynchronization*, or *resynthesis*, design flows [75, 76]—uses conversion algorithms. It starts with the synthesis of a synchronous specification, then converts the synchronous netlist into an equivalent asynchronous BD circuit by replacing the registers with an asynchronous template (flip-flops are replaced with latches, and the clock tree is replaced with the handshake control network). Most notably, a detailed tutorial published in the 2006 edition of the Synopsys User Group (SNUG) [77] presented an asynchronous synthesis flow supporting conventional HDL specifications using Synopsys tools, which relies on dedicated micropipeline standard-cell libraries. More recently, an

automated tool flow called Proteus [78], based on the desynchronization design flow [75], was developed by Fulcrum Microsystems (now part of Intel) to produce high-performance asynchronous Ethernet switches. Other conversion algorithms in the same spirit were also proposed [76, 79].

The second category uses adapted circuits and EDA methods to synthesize and implement asynchronous circuits description much like regular synchronous designs. In this case, the controllers are designed at the gate-level, and a set of design constraints (e.g. *dont_touch* directives) are used to prevent the synchronous synthesis tool from automatically removing them. A first approach uses no optimizations in the synthesis flow and *floorplanning constraints* in the physical implementation flow to manage the timing requirement of asynchronous circuits. By constraining the placement of cells, routing becomes more predictable, which limits the variability of delay. It was employed for the design of an asynchronous BD processor on FPGA in [80], using the *Hard Macro* method in Xilinx tools as a way of isolating DEs and combinational logic between incremental synthesis steps. A similar method based on *hierarchical design* was proposed in a work preceding this thesis to implement an Octasic-like Execution Unit stage on FPGA [45]. Another method, based on relative location (RLOC) constraints, was used in [81] to minimize internal delays in a QDI circuit implemented on FPGA, by placing neighboring cells close to each other. A similar approach was also used in [65], using the *createRegion* command in Cadence Encounter, to bound timing variations in a QDI circuit. However, these *floorplanning-based* methods are neither practical nor scalable, and lead to poor performances. Indeed, the full capabilities of synchronous tools—synthesis optimizations and timing verifications—can only be exploited by adequately defining the timing constraints of the circuit. Thus, a second approach uses timing constraints to enable the timing-driven synthesis of BD asynchronous circuits. Using these methods the design—that is, the gate-level controllers and the high-level description of the datapath—can be automatically synthesized, optimized, and verified based on the timing requirements of the circuit. However, because timing verifications and performance optimizations rely on Static Timing Analysis, the efficiency of these *timing-based* methods depends on the level of integration of the asynchronous design with the STA tool. The Static Timing Analysis of asynchronous circuits with standard EDA tools is an on-going research topic that has produced significant results very recently, thanks to the combined work of Stevens [82]—with the Relative Timing Constraint (RTC) formalism—, and of Gimenez [83]—with the Local Clock Set (LCS) methodology—, which we leverage in this work. It is discussed in more depth in the following section.

2.5 Static Timing Analysis

The timing analysis of a digital design either refers to timing simulations, in which the analysis is carried out *dynamically* by observing the response of the system to stimuli applied on its inputs, or to Static Timing Analysis, in which the analysis is carried out *statically* and does not depend upon stimuli being applied [84]. A timing simulation validates the functionality as well as the timing of a circuit, by simulating its netlist in combination with its timing file—the industry standard is the Standard Delay Format (SDF) [85]—, which contains the delays of each cell and wires in the circuit as well as timing checks (*e.g.* setup, hold). However, it lacks in quality, because the verification coverage depends on test vectors producing the stimuli, and in efficiency, because it is resource intensive and time consuming. On the other hand, the Static Timing Analysis *exhaustively* checks the timing paths of the circuit without taking functionality into account. It is much faster than timing simulations, and scales well with the increasing complexity of VLSI circuits. STA is thus the preferred way to achieve timing closure [1, 84].

Both timing simulations and STA derive the timing of a circuit from the timing information that are provided by the standard-cells library. Each cell in the library contains a timing *characterization file*—usually written in the *liberty* (.lib) format—, which provides timing models for various instances of the cell in the design environment. These models are obtained from (SPICE) circuit simulations that reflect different scenario of the cell operation [84]. Different models exist with a trade-off between accuracy and size that influence simulation and STA time (the standards are the NLDM, CCS, and ECSM models). Note that similar models are available for power information. In addition to providing accurate timing information, a characterization file also defines the timing relation within a cell (*e.g.* setup and hold checks for a sequential cell). These relations, and timing information, are exploited to produce SDF files, and are used by STA tools in combination with timing constraints.

The Static Timing Analysis uses timing constraints—the standard of the industry is the Synopsys Design Constraints (SDC) format—to describe the interactions of events in the circuit (*e.g.* a clock definition, an input delay, *etc.*) within the STA tool. It builds the timing information of the circuit from the combined knowledge of the timing constraints (SDC file), the circuit organization (netlist), and the internal relations and timing information of each cell (characterization file). Thus, the timing analysis of a circuit using STA is only as good as its timing constraints. The STA engine is solicited throughout the steps of the standard design flow to enforce—through timing-driven optimizations—and verify—with STA reports—the requirements set by the timing constraints.

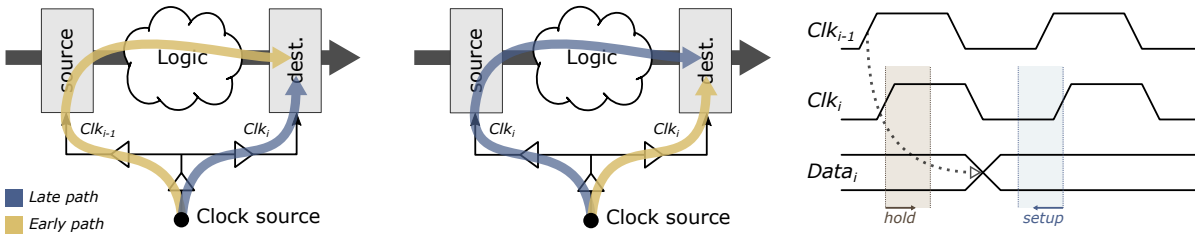
However, as we have seen, STA tools have been designed with the synchronous paradigm in mind. Thus, the lack of support for asynchronous designs by STA tools results in a poor compatibility between asynchronous circuits and standard EDA flows (see Section 2.4). Nevertheless, the Static Timing Analysis of asynchronous BD circuits is possible, although it is not trivial. In this section, we first review the basic timing constraints of a typical synchronous and BD asynchronous design, and we show the limitations of standard STA tools with respect to asynchronous circuits. Then, we review the methods at the state of the art that are employed to alleviate these limitations.

2.5.1 Timing constraints of synchronous and BD asynchronous designs

Figure 2.12 and Figure 2.13 represent the typical case for the evaluation of the timing constraints in a synchronous and a BD asynchronous pipeline stage. In both cases, the correctness of the circuit is checked against *setup* and *hold* timing constraints [21]. Setup and hold checks compare the delay of two competing paths: one that should be *early* (E_{path}) and another that should be *late* (L_{path}), represented in Figure 2.12 and Figure 2.13 with yellow and blue arrows respectively. In STA terminology, a timing path begins at a *startpoint*—represented as a thick dot in Figure 2.12 and Figure 2.13—and ends at an *endpoint*. The timing path between a startpoint and the data pin of a register is called the *launch* path, and the timing path between a startpoint and the clock pin of a register is called the *capture* path. In a synchronous circuit the startpoint is identified as the clock source, while in a BD asynchronous circuit it is identified inside the handshake controller and depends on its specific implementation [86]. Using this representation, setup and hold constraints can be differentiated by the orientation of the inequality, which determine through which path the signals should arrive first. Hence, timing constraints have the following form [21]:

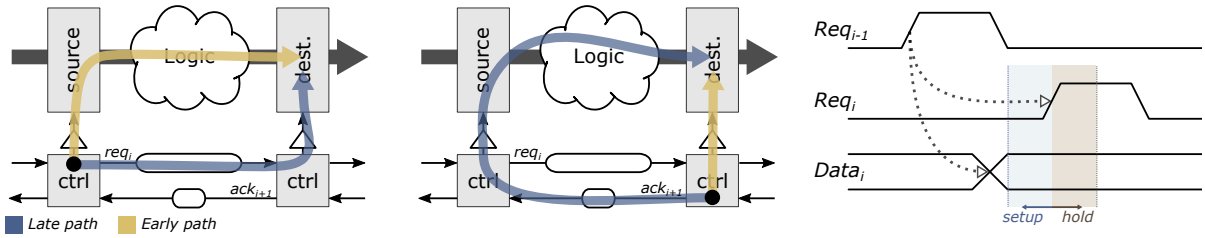
$$\delta_{min}(L_{path}) > \delta_{max}(E_{path}) \tag{2.1}$$

where δ_{min} and δ_{max} respectively represent the minimum and maximum delays of the paths. From the startpoint, the launch path first reaches the clock input of the source register ($\delta^{start \to source}$) through the clock tree, then passes through the combinational logic to reach the destination register ($\delta^{source \to dest}$). In BD systems, for the hold analysis (Figure 2.13b), this path goes through the acknowledgement line. The capture path starts from the startpoint then reaches the clock input of the destination register ($\delta^{start \to dest}$) through the clock tree. In BD systems, for the setup analysis (Figure 2.13a), this path goes through the request line along the Delay Element. In both the synchronous and the BD cases, the *skew* measures the delay difference of the clock distribution network between the launch and the capture paths.



(a) setup (b) hold (c) timing diagram

Figure 2.12 Timing of a typical synchronous pipeline stage



(a) setup (b) hold (c) timing diagram

Figure 2.13 Timing of a typical Bundled-Data asynchronous pipeline stage

$$T_{clk} + \delta_{min}^{start \to dest} > \delta_{max}^{start \to source} + \delta_{max}^{source \to dest} + t_{setup} + t_{margin} \tag{2.2}$$

$$\delta_{min}^{start \to dest} > \delta_{max}^{start \to source} + \delta_{max}^{source \to dest} + t_{setup} + t_{margin} \tag{2.3}$$

The setup constraint guarantees that the launch path is *faster* than the capture path—*i.e.* the signals through the launch path should arrive *early* at the endpoint—when they are evaluated with a maximum delay (δ_{max}) and a minimum delay (δ_{min}) analysis (respectively) to account for the worst case scenario. Applying this reasoning to a synchronous system leads to Relation (2.2), while applying it to an asynchronous BD system leads to Relation (2.3). These relations are the same, except that the synchronous one accounts for the clock period (T_{clk}) between the launch event and the capture event. In STA terminology, this is called a *one-cycle path*, which is the default configuration for setup analysis in standard STA tools. By contrast, the BD system uses a *zero-cycle path*, because data are launched and captured within the same cycle: the edge used to launch the data from the source register is also used to capture them, after being delayed in the Delay Element, in the destination register.

$$\delta_{min}^{start \to source} + \delta_{min}^{source \to dest} + t_{hold} + t_{margin} > \delta_{max}^{start \to dest} \tag{2.4}$$

Similarly, the hold constraint guarantees that the launch path is *slower* than the capture path—*i.e.* the signals through the launch path should arrive *late* at the endpoint—when they are evaluated with a minimum delay (δ_{min}) and maximum delay (δ_{max}) analysis (respectively) to account for the worst case scenario. Relation (2.4) expresses the hold constraint for both the synchronous and the BD asynchronous system. Indeed, in both cases the launch and the capture events are evaluated within the same cycle. The default configuration for hold analysis in standard STA tools is zero-cycle path. Note that the t_{margin} term is a margin to account for clock uncertainties (*e.g.* the jitter), and the t_{setup} and t_{hold} terms refer to the setup and hold intrinsic parameters of the registers.

$$slack = \text{data required time} - \text{data arrival time} \tag{2.5}$$

In STA terminology, the delay of the signals through the launch path is called the *data arrival time*, and the delay of the signals through the capture path is called the *data required time*. The *slack* (see Relation (2.5)) measures the amount of time by which the timing constraint is met. Its definition is the same for setup and hold timing checks. However, in accordance with Relation (2.1), the arrival (required) time must be evaluated with a maximum (minimum) delay analysis for the setup, and the opposite is true for the hold.

2.5.2 Relative Timing Constraints

The timing of asynchronous circuit is best modeled using a formal verification formalism called Relative Timing (RT), which was developed by Stevens in the early 2000's [87]. This formalism, which is not restricted to any particular design style, allows to explicitly specify the effect of delays in a circuit, in terms of assertions on the relative ordering of events. In particular, the RT design methodology was employed to design RAPPID [24], an asynchronous instruction length decoder that achieved impressive results, as discussed in Chapter 1. When used in the context of integrating BD asynchronous designs with the standard EDA flow, the RT formalism allows to create and prove correct necessary constraints—called Relative Timing Constraints (RTCs)—that support timing driven synthesis, physical design, and Static Timing Analysis [82]. RTCs formally define the timing requirements that a circuit must satisfy to operate correctly using the following relation [83, 88]:

$$pod \mapsto poc_{early} \prec poc_{late} - \varepsilon \tag{2.6}$$

It specifies that the consequences of an event occurring at the *point-of-divergence* (*pod*) must reach the *point-of-early-convergence* (poc_{early}) before reaching the *point-of-late-convergence*

(poc_{late}) with a margin ε. *Internal* RTCs define the timing relations that are local to a component (such as a register). *Protocol-level* RTC define the timing relations that exist between adjacent stages having control paths dependencies, such that are typically found in BD elastic channels where the control paths span over adjacent stages through DEs. The interested reader may find detailed classifications of RTCs in [82, 83].

In STA terminology, a *pod* is a startpoint, while a *poc* is an endpoint. Thus, the *pod* is the *last common point* between the launch and the capture paths; it corresponds to the thick dot in Figure 2.12 and Figure 2.13. In contrast with the timing constraints defined in the previous section, RTCs *explicitly* define the startpoint from which the constraint is derived. A RTC translates to a setup condition if the capture event should reach the *poc after* the launch event. Similarly, a RTC translates to a hold condition if the capture event should reach the *poc before* the launch event [82, 86]:

$$ctrl_i \mapsto D_{i+1}^{max} \prec C_{i+1}^{min} - \varepsilon \tag{2.7}$$

$$ctrl_i \mapsto C_{i}^{max} \prec D_{i}^{min} - \varepsilon \tag{2.8}$$

The setup condition (2.7) specifies that an event starting from the startpoint $ctrl_i$—the i^{th} event from the clock source in a synchronous system, and an event on the i^{th} controller in a BD system—should trigger an event D_{i+1}^{max} on the data pin of the destination register (through the longest path) before the clocking event C_{i+1}^{min} occurs on its control pin (through the shortest path). Similarly, the hold condition (2.8) specifies that the event starting from the startpoint $ctrl_i$ should induce the event C_i^{max} before the event D_i^{min} occurs.

2.5.3 Integration of BD asynchronous circuits in standard STA tools

To ensure the compatibility between an asynchronous circuit and the standard EDA flow, timing constraints must be derived from the circuit, and be supported throughout the stages of the synthesis, physical, and verification flows [21, 26]. Given the similarities between synchronous and BD timing constraints, as defined in the previous sections, it seems trivial to make BD asynchronous circuits compatible with standard STA tools. However, multiple issues complicate the specification of BD timing constraints in standard STA tools; the correct definition and translation of RTCs into SDC commands is not trivial [86]. Note that developing the point of view from which the similarities between synchronous and BD timing constraints can be observed is an important part of the process to addressing these issues.

1. *Clock definitions*—Standard STA tools rely on *clock definitions* (`create_clock` SDC command) to activate setup and hold timing checks: A clock—defined on a startpoint

with a fixed period T_{clk}—reaches the clock input of all the registers it is connected to [89]. However, the irregular propagation of events through DEs in BD asynchronous circuits is not efficiently captured with fixed-period clock definitions.

2. *Timing loops*—The elastic channel is intrinsically composed of combinational loops with the request and the acknowledgement lines. Standard STA tools must break these loops in order to perform the static analysis. But the automatic and arbitrary cut of these timing loops by the tool results in disabling timing paths that are valid, and that should be accounted for in the verification process. Similar timing loops also exist within the handshake controllers, but the problem varies depending on its specific implementation.

3. *Startpoint*—In contrast with a synchronous system, in which the startpoint is a known and fixed port in the circuit (*i.e.* the clock source), the irregular placement of startpoints within the controllers in BD asynchronous circuits cannot be automatically retrieved by the STA tools. Without startpoint definition, the timing constraints cannot be properly derived. Thus, they must be described manually within the STA tool without breaking any valid timing path.

4. *Timing driven synthesis*—The STA engine is solicited during the synthesis operation to provide timing information feedbacks, as the optimization algorithms alter the circuits to meet the timing constraints. During this iterative process, the timing relationship between the datapath and the control path should be preserved to allow the synthesis tool to concurrently optimize control and data paths. In the case of BD asynchronous circuits, this relationship is difficult to preserve because the control path, which includes Delay Elements, may vary as much as the combinational logic it is supposed to match.

The first approaches to tackle these issues were mainly empirical, and did not rely on the RT formalism. The work reported in [90] first proposed to define *virtual clocks*—clock definitions not associated to a source—in a latch-based BD circuit to constrain the combinational logic and enable synthesis optimizations. In addition, each DE is constrained with `set_min_delay` and `set_max_delay` SDC constraints to validate the delay matching with the associated combinational logic. This method addresses only part of the limitations listed above: points (2) and (3) are not dealt with, and points (1) and (4) are only partially addressed with virtual clock definitions and *min/max* delay constraints. Although these constraints allow some degree of optimization of the datapath, the fixed period of the virtual clock, and the fixed value used with the *min/max* constraints, is not adapted to deal with the evolution of DEs timing during the synthesis operation. In the first part of this thesis we use a similar empirical approach, as reported in [46, 47] and detailed in Chapter 3. However, instead of using

min/max delay constraints on control paths, we define a set of *generated clocks*, derived from virtual clocks, to model the propagation of events in the control path of an Octasic-like Execution Unit. The resulting timing paths are then defined as zero-cycle paths to account for the non-periodic characteristics of bundled data operations (see Section 2.5.1), using `set_multicycle_path` SDC constraints. These zero-cycle clock constraints enable better optimizations of the datapath, and facilitate Static Timing Analysis, but still lack in efficiency. Indeed, similar to the previous method, these constraints should be updated during the synthesis flow to reflect the actual delays of the control paths, which prevent efficient timing driven synthesis. The first method developed in combination with RTCs is reported in [82,88]. It was patented in 2015 [91], and was successfully used to design an energy-efficient asynchronous MSP430 DSP in 2017 [25]. In contrast with the previous methods, this approach is more generic. It relies on formal verification (Relative Timing) to systematically derive the necessary timing constraints of the circuit, using `set_min_delay` and `set_max_delay` SDC constraints, and explicitly disable timing loops with `set_disable_timing` SDC commands. According to Relations (2.7) and (2.8), the *min/max* constraints are respectively used to constrain the control (*resp.* data) and the data (*resp.* control) paths for the setup (*resp.* hold) analysis. Some tools have been developed to automate the process of deriving RTCs from a sequential system and create the associated min/max delay constraints, such as ARTIST [92] or ACDC [93]. However, similar to the previous methods, this approach does not fully benefit from the synthesis tool ability to explore the design space with timing-driven optimizations, because the constraints must be updated during the flow, based on successive extractions of the timing results, which should be provided for all PVT corners. Finally, the Local Clock Set (LCS) methodology [83,86], uncovered in the 2018 edition of ASYNC, gets the best from these previous works by interpreting the RTC formalism with clock constraints, improving the compatibility with standard EDA tools. First, it exhaustively defines the RTCs of a BD template. Then, for each RTC, a set of clocks constraints are advantageously defined to describe the propagation of events, in a way that preserves the timing relationship between data and control paths, allowing the synthesis tool to concurrently optimize them. In addition, clock constraints defined at specific points of the controllers solves the STA tool inability to manage combinational loops without disabling any relevant timing path. The definitive EDA flow presented in Chapter 4 adapts the LCS methodology to the case of KeyRing circuits.

In the next chapter, we present the Mini-Mips processor experiment that can be seen as an intermediate step toward the development of the KeyRing microarchitecture and of the definitive EDA flow.

CHAPTER 3 MINI-MIPS: IMPLEMENTING A SIMPLE OCTASIC-STYLE ASYNCHRONOUS PROCESSOR ON FPGA

In this chapter, we study the Octasic design style [35] using an empirical approach based on reverse engineering. This study is motivated by three main goals: *i)* uncover the design principles that can be drawn up from designing a simple general purpose processor using an Octasic-style asynchronous microarchitecture; *ii)* propose circuits and methods for the implementation of Octasic-style asynchronous microarchitectures on FPGA, using synchronous-oriented EDA tools; and *iii)* compare the design trade-offs and the power efficiency of the proposed Octasic-style asynchronous microarchitecture with a synchronous alternative. To this end, we propose to design and compare two processor microarchitectures—one with a classical synchronous 5-stage pipeline microarchitecture, similar to the one described in the book *Computer Organization and Design* [94], and the other with an Octasic-style microarchitecture, inspired from Octasic 2012 ASYNC publication [29], and the 2012 and 2014 patents [30, 31]—that implement a simplified version of the MIPS Instruction Set Architecture [95], called *Mini-Mips*. These two processors target an FPGA-based emulation platform, which provides the support for monitoring their performances and power consumption.

First, Section 3.1 discusses the Mini-Mips ISA. Then, Sections 3.2, 3.3, and 3.4 explore the practical aspects of implementing the Mini-Mips ISA using synchronous and Octasic-style asynchronous microarchitectures, with an emphasis on Instruction Level Parallelism. Section 3.6 presents the hardware emulation platform and the experimental protocol for comparing the processors performance and power consumption. Section 3.5 deals with the processors implementation on FPGA. In particular, it addresses the issues of implementing asynchronous building blocks (such as Delay Elements) within the FPGA fabric, and tackles the problems of timing—*i.e.* Static Timing Analysis and timing-driven implementation—with standard EDA tools. Section 3.7 first compares post-implementation timing simulations with STA results, and then analyzes the measures taken on the emulation platform. Finally, Section 3.8 provides a feedback on this first design attempt. In particular, it discusses the limits of the proposed timing method, presents post-implementation timing simulation issues, and highlights imprecisions in the results provided by the FPGA-based emulation platform, which, combined, suggest that an ASIC design flow is better suited for this type of study.

Finally, note that this chapter is an extended version of an article published in 2017 [47], and that it is the result of preliminary research conducted on related topics which was also published [45, 46].

3.1 Mini-Mips Architecture

The Mini-Mips ISA is a simplified version of the original MIPS ISA [95]. We chose to simplify the original specification to facilitate the design of the datapath, and thus focus on the Octasic-style asynchronous microarchitecture. The Mini-Mips ISA makes use of all three MIPS 32-bit instruction formats (R-type, I-type, J-type), and addressing modes (register, immediate, displacement). Only 10 instructions were kept from the original MIPS ISA: `add`, `sub`, `and`, `or`, `jr`, `j`, `beq`, `addi`, `lw`, and `sw`. As a result, compiling code written in high-level languages (such as C), or in MIPS assembly, using existing compilers is difficult as it would generate many instructions that are not implemented in the Mini-Mips. Thus, instead of using standard compilers, the compilation of programs for the Mini-Mips is performed by a custom *Tcl* script. It simply converts a program written in Mini-Mips assembly into the correct sequence of hexadecimal 32-bit words. The resulting file (*init_imem.hex*) is then used as an initialization file for the instruction memory. The syntax and the semantics of the Mini-Mips assembly is derived from the standard MIPS assembly [95].

3.2 Top-level and common building blocks

Figure 3.1 shows the top-level view of the Mini-Mips processor, targeting Xilinx ZYNQ *7-series* FPGAs, in which the memories are interfaced with the *core* component—having either the synchronous or the Octasic-style asynchronous microarchitecture—and AXI buses that make the link with the processing element of the ZYNQ device for programming and monitoring purposes (see Section 3.6). In accordance with the Mini-Mips ISA, memories are byte-addressable and in little-endian mode. The *instruction memory* (IMEM) is read-only, whereas the *data memory* (DMEM) provides read and write accesses. Both IMEM and DMEM use a Dual Port Memory (DPM) component—one memory space is shared between two independent access ports, each with its own clock, address, and data lines—which target BRAM blocks in Xilinx 7-series FPGAs. The instruction memory uses a 10-bit address port driven by the Program Counter (PC), and a 32-bit data port interfaced with the core to provide the next instruction to be computed. It is clocked by the *axi_clock* signal for AXI transactions, and by the *imem_clock* signal for transactions with the core. Similarly, the data memory is composed of a 10-bit address port, a 32-bit read port, and a 32-bit write port interfaced with the *Load/Store* unit in the core. The write operation is performed synchronously with the processor when the *write_enable* signal is set. It is clocked by the *axi_clock* signal for AXI transactions, and by the *dmem_clock* signal for transactions with the core. When using the synchronous microarchitecture, both the *imem_clock* and *dmem_clock* signals are

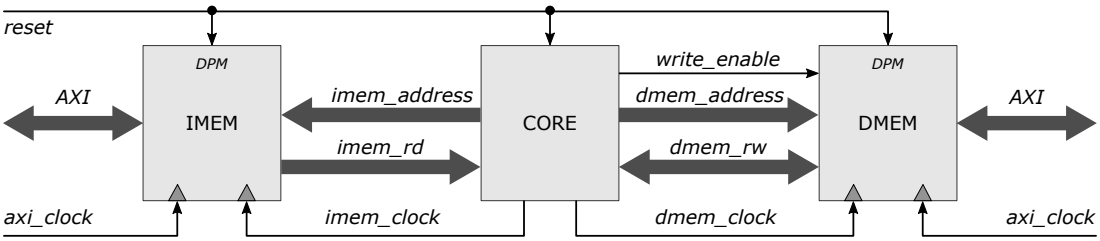


Figure 3.1 Mini-Mips Top-Level: memory interfaces with the core and AXI bus

driven by the global clock signal used in the core; using the Octasic-style asynchronous microarchitecture, they are driven by independent self-generated clock (see Section 3.4). In simulation, the instruction memory is initialized by a file (*init_imem.hex*), as discussed in the previous section. Likewise, the data memory content is dumped in a file (*dump_dmem.hex*) at the end of the program execution. During FPGA emulation, the content of the instruction memory is initialized, and the content of the data memory is dumped at the end the program execution, through the AXI interface (see Section 3.6).

The core is composed of building blocks that are common to both the synchronous and the Octasic-style asynchronous microarchitectures. These modules—Program Counter (PC), Register File (RF), and Arithmetic and Logic Unit (ALU)—are standard synchronous modules composed of a combinational logic cloud and one register stage (standard edge-triggered flip-flops). Having modules shared between both microarchitectures is possible thanks to the *pulse-based* self-timed clocking scheme used in the Octasic-style asynchronous microarchitecture (see Section 3.4). Similar to the case of the memory interfaces, the modules are driven by the global clock when used in the synchronous microarchitecture, and by their own self-generated clock when used in the Octasic-style asynchronous microarchitecture.

3.3 The synchronous 5-stage pipeline microarchitecture

The synchronous version of the Mini-Mips processor is a textbook implementation of the Mini-Mips ISA using a 5-stage pipeline as described in *Computer Organization and Design—MIPS Edition* [94]. It is depicted in Figure 3.2. In the following, we summarize the aspects of this standard synchronous microarchitecture that we think are relevant for the comparisons with the Octasic-style asynchronous microarchitecture in Section 3.4.

The execution of each instruction is divided into five stages—Instruction Fetch (*IF*), Instruction Decode (*ID*), Execute (*EX*), Memory Access (*ME*), Write-Back (*WB*)—, each having a one clock cycle latency that overlaps in time, thereby implementing Instruction Level Paral-

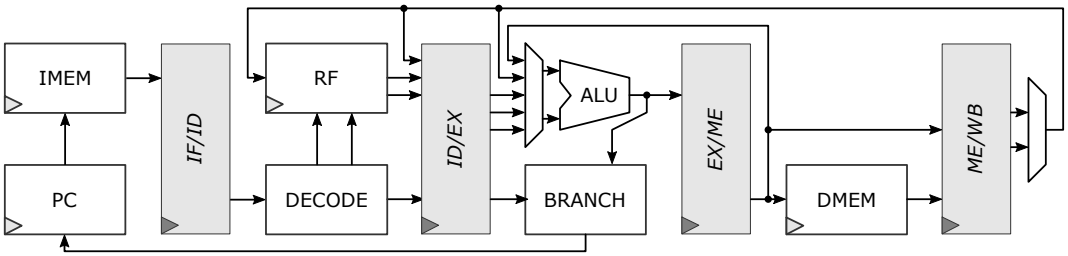


Figure 3.2 Overview of the Mini-Mips synchronous 5-stage pipeline microarchitecture

lelism as shown in Figure 3.3. Hazards arise in the synchronous microarchitecture because of the overlapping in time of instructions execution in the pipeline. In the synchronous Mini-Mips, the three types of hazards are handled as follow:

- **Structural hazard**—They may occur when some combination of instructions cannot be accommodated because of resource conflicts. Memory access conflicts—instruction i tries to access the memory in the *ME* stage while instruction $i+3$ tries to access the memory in the *IF* stage—is avoided by having separated data and instruction memories (see Figure 3.2). Similarly, the RF may have access conflict when instruction i tries to write to a register in the *WB* stage while instruction $i+3$ tries to read the same register in the *ID* stage. To prevent this, the Register File has a write-back buffer which allows to read a register that is being written during the same cycle.

- **Data hazards**—They arise when an instruction depends on the result of a previous instruction. Because the Mini-Mips cores process instructions *in-order*, only Read After Write (RAW) hazards can occur, that is, when instruction $i+1$ and/or $i+2$ use at least one operand register that is being used by instruction i as a destination register. To avoid data hazards, two design choices have been made: *i)* When the result of instruction i is known at the *EX* stage (all instructions except `lw`), it is fed back from the *EX/MEM* registers to both the *EX* and *ID* stages. *ii)* When the result of instruction i is not known until the end of *MEM* stage (`lw` only), the result is fed back from the *ME/WB* register to both the *EX* and *ID* stage and the processor is *stalled* for one cycle. In the synchronous microarchitecture, stalls are handled during the *ID* stage. The PC and the *IF/ID* registers *enable* signals are set to 0 (they hold their current value) and the *ID/EX* register *reset* signal is set to 1, such that the instruction currently in the *ID* stage does not alter the architectural state of the processor in the next stages. Instructions further along in the pipeline are processed normally.

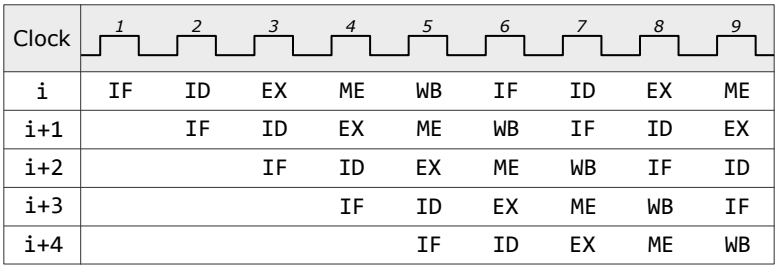


Figure 3.3 Instruction Level Parallelism in the synchronous 5-stage pipeline

- **Control hazards**—They arise from the overlapping in time of instructions that alter the Program Counter. Instructions following a `beq`, `j`, or `jr` instruction in the program enter the pipeline before the decision to change the PC has been taken. In case of a jump or a taken branch, those instructions must be *flushed* out of the pipeline to ensure the correct operation of the program. The number of instructions to be flushed in that case depends on where the decision to change the PC is taken. In the synchronous microarchitecture, branches and jumps are resolved during the *ID* stage, which limits the number of flushed instruction to one. An instruction is flushed by reseting the *ID/EX* register, such that it is seen as a `nop` by the rest of the stages. Resolving a branch consist in comparing the value of the operand registers: the branch is taken if both operands are equal, an operation performed in the ALU. The branch and jump flags are interfaced with the PC from the *EX* stage along with the target address signal.

3.4 The Octasic-style asynchronous microarchitecture

The Octasic-style asynchronous microarchitecture of the Mini-Mips ISA presented in this section is the result of an empirical development process, in which successive iterations of the design were tested and evaluated, using a *trial-and-error* approach, prior to settling on a final version. Instead of presenting the various stages of the design process, this section presents the *final* version of the processor but emphasizes, when necessary, on the choices and the trade-offs that have led to the proposed microarchitecture. The main guidelines that were followed for the design of the Octasic-style asynchronous core are: *i)* it must be *equivalent*—in features and in performance—to the synchronous 5-stage pipeline microarchitecture; *ii)* it must make use of the main asynchronous *ad hoc* design principles from Octasic, as described in Section 2.3; *iii)* it must target FPGA implementation. The following details the specifications for the design of the Octasic style asynchronous Mini-Mips core stemming from these guidelines:

- **In-order operation**—In contrast with the Opus2 and the AnARM microarchitectures, which execute instructions out-of-order (see Section 2.3), the Mini-Mips core should process them in-order. This requirement ensures that the asynchronous core is equivalent to its synchronous alternative. But most importantly, it allows to explore the design trade-offs of in-order Octasic-style asynchronous cores, which was never done before. In addition, this requirement is also in line with the goal of uncovering generic design principles from the Octasic *ad hoc* design style. Indeed, before studying an out-of-order microarchitecture it seems relevant to first study its in-order mode of operation.

- **Performances**—The asynchronous Mini-Mips should have performances comparable to its 5-stage pipeline synchronous alternative. This requirement ensures that the power consumption comparisons are *fair*. To this end, we first decided to use 4 Execution Units, although the final design uses only 3 EUs, as explained in the following.

- **Design reuse**—The asynchronous core should be able to reuse the modules designed for the synchronous microarchitecture. This requirement is also in the spirit of drawing up fair comparisons between the two microarchitectures. It is made possible by the pulse-based Octasic-style asynchronous scheme which allows to use flip-flops in datapaths.

- **FPGA implementation**—The Octasic-style asynchronous core should be compatible with standard FPGAs. Hence, special asynchronous control circuits—Delay Elements, Token Units—should be adapted to fit within FPGAs fabric.

3.4.1 Overview

Figure 3.4 shows the final version of the Octasic-style asynchronous Mini-Mips core top-level organization. It is composed of 3 *multicycle* Execution Units of 6 stages, and uses 6 *tokens—I, R, A ,D, W, P*—that are managed in Token Units (TUs) to orchestrate Instruction Level Parallelism and generate self-timed clocks. The crossbar-switch allows EUs to communicate with one another, along with the *shared resources—i.e.* PC, RF, ALU, IMEM, and DMEM. These resources, which are common with the synchronous microarchitecture (see Section 3.2), are shared across EUs and their access is arbitrated by the *token-ring*. Each token within the token-ring controls a stage of the multicycle EUs and is associated with a resource access, with an exception made of the W token, which does not control the access to any resource, as discussed in the following. When an EU requests a resource access at a given stage—*i.e.* when the associated token passes through the TU of that stage—, a local clock is generated

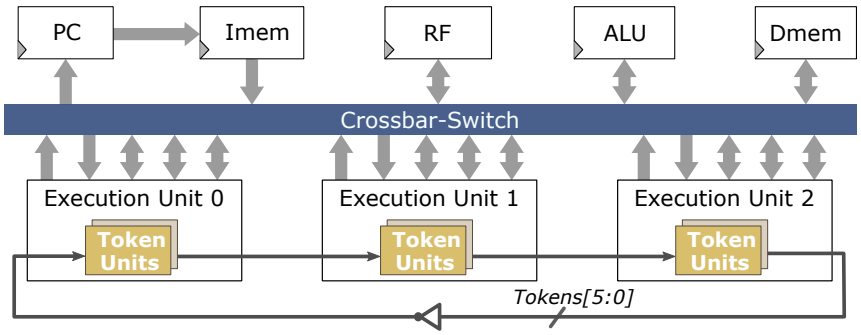


Figure 3.4 Octasic-style asynchronous Mini-Mips core top-level organization

to trigger the registers holding the data in both the resource and the EU stage. For instance, the I_0 clock, issued when the I token passes through the TU of the I stage in EU_0, triggers the instruction memory and the registers of the I stage. Before reaching the clock pins of the IMEM and the stage registers, the I_0 clock is first delayed through a DE, which matches the delay of the IMEM access time, as detailed in the following. A `not` gate ensures a change of state of a token after it has passed through each TU, which allows to differentiate request cycles. Resources are triggered by different clocks, depending on the stage at which they are operated. Clocks coming from the same stage from different EUs are OR-ed together before going to a resource. For example, the I clock triggering the IMEM module is driven by I_0 `or` I_1 `or` I_2. Note that a synchronous clock is also used for the performance counters. Finally, in addition to generating the local clocks for each stage, the TUs also generate control signals that are used in the crossbar to guide the data coming from shared resources to the appropriate EU.

Figure 3.5 represents a multicycle EU interfaced with the crossbar. Instruction computation is decomposed in sequential stages holding intermediate results in registers, much like regular pipelined microarchitectures. However, unlike conventional pipelining, an EU processes an instruction through each stage before it issues the next. Compared with the synchronous single-issue 5-stage pipelined microarchitecture (Section 3.3), where ILP is obtained by simultaneously processing multiple instructions steps in different stages, the Octasic-style asynchronous microarchitecture exploits ILP by simultaneously processing multiple instructions across EUs. Instructions are decomposed in 6 stages, each of which is associated with a token. At each stage, local clocks issued from TUs are delayed through Delay Elements. DEs are sized such that their respective delay exceed the associated resource access logic delay. The token-ring, composed of 3 EUs of 6 stages, issues 18 self-timed clocks. Each EU processes one instruction at a time. Using a different token at each stage, they accesses a different

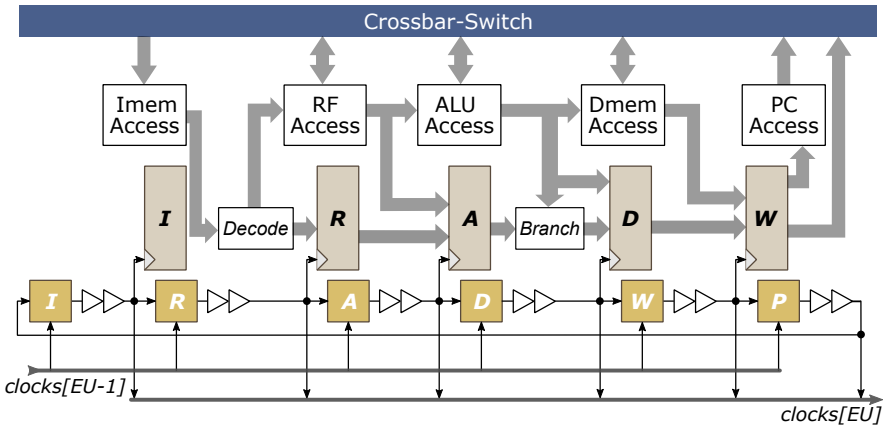


Figure 3.5 Multicycle Execution Unit

resource simultaneously while respecting the program order.

EUs stages perform the following operations:

- **I** (*IMEM access*)—The instruction word pointed by the PC is fetched from the instruction memory, which is triggered by the clock issued from the *I* TU.
- **R** (*RF access*)—The instruction is decoded, and the RF is triggered by the clock issued from the *R* TU. Operands are read from the RF, and the result of the previous instruction processed by the EU (stored in the *W* registers) is written back to the RF.
- **A** (*ALU access*)—An operation is performed in the ALU and the result is stored in the ALU registers by the clock issued from the *A* TU.
- **D** (*DMEM access*)—In case of a load (`lw`) or a store (`sw`) instruction, the data memory is accessed using the clock issued from the *D* TU. Using this clock, the result of the ALU is stored in the *D* registers. The branch decision is also computed at this stage. It is transmitted to the other EUs through the crossbar, while the targeted address is transmitted to the PC at the *P* stage.
- **W** (*Write back*)—Either the ALU result or the data memory content is stored in the *W* registers, using the clock issued from the *W* TU. The write-back result, and its associated address, are not written to the RF using the *W* clock to avoid a structural conflict in the RF. Instead, data that are forwarded back to the RF from the *W* stage are saved using the *R* clock.
- **P** (*PC access*)—The PC is accessed using the clock issued from the *P* TU. The address it generates is used to point to the next instruction to be processed in the EU. In case

of a branch or a jump instruction, the targeted address saved at the D stage is used.

3.4.2 Instruction Level Parallelism

Exploiting ILP alters the normal execution flow of a program by exposing data dependencies and resources access order between nearby instructions to the hardware. In the single issue pipelined microarchitecture, as described in Section 3.3, hazards arise because of the overlapping of instructions across pipeline stages. Using the Octasic-style asynchronous microarchitecture, hazards arise because of the overlapping of instructions across EUs, as illustrated in Figure 3.6. To alleviate these hazards, the crossbar-switch is responsible for sharing data among EUs and resources, and the token-ring is responsible for resolving resources access conflicts and preserving the program order.

- **Structural hazards**—They would arise from concurrent access to the RF (reads at the R stage and writes at the W stages), or from concurrent accesses to memory. Similar to the synchronous microarchitecture, the latter is prevented by having distinct instructions and data memories (IMEM and DMEM are respectively accessed at the I and D stages by dedicated clocks). The RF, on the other hand, can be the source of more complex structural conflicts. Indeed, it can only be triggered by one clock, whereas it should be triggered by the R and W clocks for read and write accesses. One solution consists in *OR-ing* R and W clocks together; another solution consists in using either the R or the W clocks for both read and write accesses. We chose the second solution as it works regardless of the stages organization, by taking advantage of the multicycle nature of EUs. Although an instruction result is not written back until the next instruction is at the R stage, data in the W stage are not overwritten. Moreover, in case of data dependencies, data can be forwarded from W stages between EUs using the crossbar.

- **Data hazards**—They occur when at least one operand used by an instruction in an EU is being used as a destination register by another instruction in another EU. They are addressed by forwarding data between EUs in the crossbar. Operands addresses are compared with the write-back addresses from other EUs, and data are forwarded in case of a match. If a result is not yet available when the RF is being accessed, the A stage is *stalled—i.e.* the clock issued from the A TU is not released—until a result is released by the producer EU. As opposed to using a synchronous pipeline, in which the stall time is a fixed number of clock cycles, using the Octasic-style asynchronous

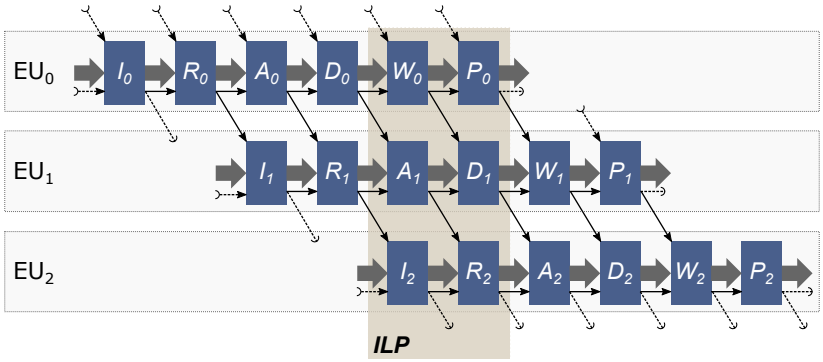


Figure 3.6 Organization of EU stages in the Octasic-style asynchronous microarchitecture. Black arrows represent dependencies between stages.

microarchitecture, the stall time depends only upon the time left for the required result to be available (see Section 3.4.3).

- **Control hazards**—They would arise each time an instruction modifies the PC. Similar to the synchronous microarchitecture, the Octasic-style asynchronous core uses a *predicted-not-taken* branching strategy, and the branch outcome is computed during the *A* stage. Thus, two instructions are being processed in the first two EUs while the branch decision is being processed in the third EU. When the branch (or jump) is taken, control signals are sent through the crossbar switch to flush these two instructions from their respective EU. The next instruction, resulting from a taken branch or a jump, is processed in the same EU that detected it.

The token-ring is an organization of Token Units orchestrating the generation of self-timed clocks across EU stages. The level of parallelism that can be achieved depends on: *i)* the number of EUs and the number of stages per EU; and *ii)* the stage organization between EUs, as illustrated in Figure 3.6. In practice, stage dependencies are enforced with control signals—represented in Figure 3.6 with black arrows—preventing the release of tokens in TUs. The TU architecture is detailed in Section 3.4.3.

Figure 3.7 represents the ILP implementation trade-offs using the Octasic-style asynchronous microarchitecture when the number of EUs and the stage organization vary (the number of EU stages is set to 6). As a simplifying assumption, let us consider that each stage has a duration of one cycle. Figure 3.7a represents the ILP obtained with 4 EUs having a *shifted-by-one* stages organization—stage s of EU_i depends on stage s of EU_{i-1}—, which corresponds to the "natural" way of organizing the stages, where stages follow one another in EUs in a fashion similar to the synchronous pipeline (see Figure 3.3). That is, an EU can access a resource as soon as the previous EU has completed its transaction with this resource.

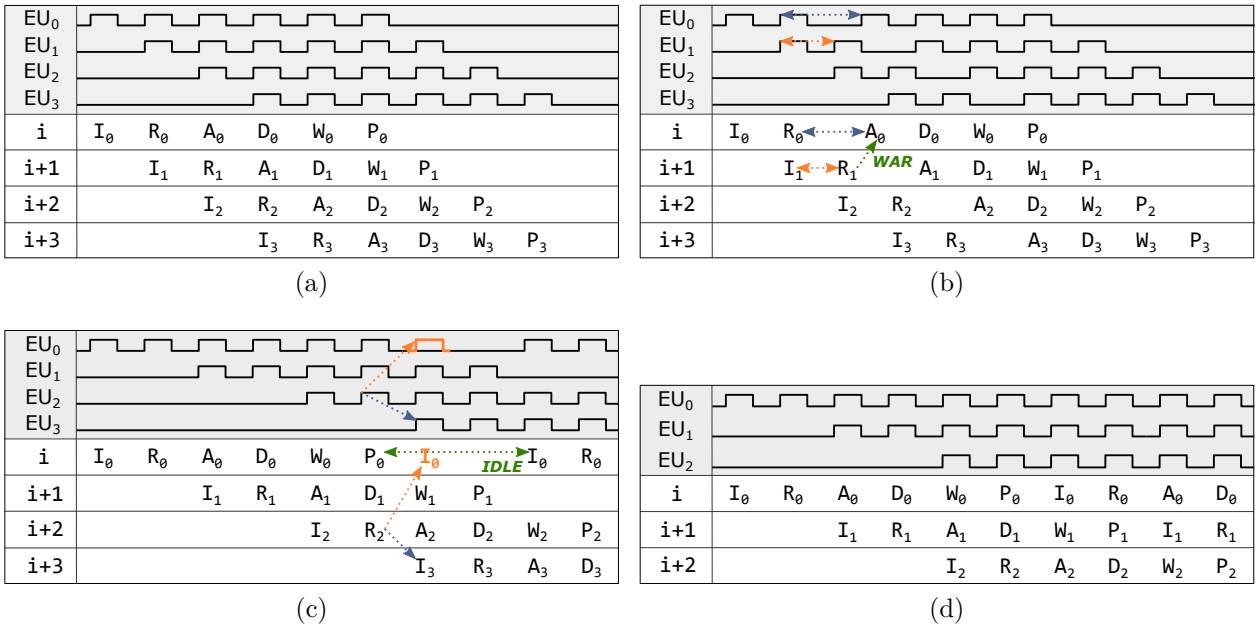


Figure 3.7 ILP implementation trade-offs using the Octasic-style asynchronous microarchitecture. A *shifted-by-one* stage organization (a) leads to WAR hazards (b). A *shifted-by-two* stage organization solves them (c) at the expense of performance (d).

However, when the cycle duration of different stages varies, as illustrated in Figure 3.7b—the A clocks have a longer delay than the other clocks—, this stage organization exhibits Write After Read (WAR) hazards. Let us take the example of the Register File access. When the R_0 clock triggers the RF, the operands on the RF bus are updated and used in the A stage of EU_0. The operands, or their outcome, are then saved in the A stage register using the A_0 clock. Concurrently with the A_0 clock, the R_1 clock reaches the clock pin of the RF, which may trigger a write operation in the RF. If this write operation overwrites the value of the registers used as operands in EU_0 before the A_0 clock has reached its registers—a situation that arises when the R clocks are faster than the A clocks, as illustrated in Figure 3.7b—the A stage of EU_0 would be exposed to a WAR hazard, thus saving the wrong values. This situation can be generalized to any stage interacting with a shared resource where the clocks delays vary. To prevent these WAR hazards, we implemented a *shifted-by-two* stages organization, in which a stage s in EU_i must wait for the completion of stage $s+1$ in EU_{i-1}, as represented in Figure 3.7c. Organizing the flow of instructions this way ensures that any data computed by a resource is saved in an EU stage register before it can be altered by the following EU. In Chapter 4, we show that this resource sharing issue, assimilated here to a WAR hazard, is actually a hold violation which, instead of being addressed architecturally, can be addressed with timing constraints and a timing-driven synthesis method.

Using the shifted-by-two stages organization generates new problems, as illustrated in Figure 3.7c. In this case, having four Execution Units is inefficient: the *same* level of parallelism can be obtained with only three EUs. Indeed, using this stages organization, the I_0 stage could start after the R_2 stage. Instead, with four EUs, R_2 controls I_3, and I_0 depends on R_3, which creates an idle time in EU_0 equivalent to two cycles, as represented in Figure 3.7c. This means that instruction $i + 3$, processed in EU_3 in a four EUs configuration, could start at the same cycle if it was processed in EU_0 in a three EUs configuration, such as it is shown in Figure 3.7d (in which case instruction i is equivalent to instruction $(i + 3)$ modulo 3). Consequently, the final version of the Octasic-style asynchronous Mini-Mips core uses a microarchitecture composed of 3 EUs of 6 stages with a shifted-by-two stages organization. Using this microarchitecture, a new clock is issued for each stage every 6 cycles. Thus, a resource is triggered once every 6 cycles by each EU, which results in having the resources triggered once every other cycle overall.

3.4.3 Token Unit

Figure 3.8 shows the Token Unit circuit used in the Octasic-style asynchronous Mini-Mips core. It is directly inspired from the TU developed by Octasic [31], as represented in Figure 2.10a, and works as follow. Note that, in the following, we use the e, s index notation to refer to signals coming from the stage s of EU e.

The *latch* controls the propagation of the token coming from the previous EU ($token_{e-1,s}$) with the *pass* signal. When it becomes transparent, the token passes and reaches the next TU. A new token-cycle begins after a token has passed through each TU and has been inverted, as represented in Figure 3.4. The *pulse generator*, composed of a XOR gate and a buffer, uses the token change of state to generate the local clock $clk_{e,s}$. The width of the clock pulse depends on the delay of the buffer, which can be appropriately sized by the synthesis tool to match the minimum pulse width of the flip-flops. In contrast with the TU architecture described in [31], in which the Delay Element is placed on the token path, the TU in the Mini-Mips uses the DE on the clock path. This facilitates the specification of the timing constraints (see Section 3.5.2), but incurs an increase in the dynamic power consumption, as the clock pulse ($0 \rightarrow 1 \rightarrow 0$) doubles the activity in the DE compared with the token (either $0 \rightarrow 1$ or $1 \rightarrow 0$). The circuits enforcing the conditions for letting tokens pass in TUs are not described in the Octasic public literature [29–31]. In the Mini-Mips, they are derived from the expected behavior of the processor. From Figure 3.7 we can deduce that, in the Mini-Mips microarchitecture, a given stage s in a given EU e, can start—*i.e.* the latch in that stage TU can release the token—when *i)* the previous stage $s - 1$ in the same EU e, and *ii)* the next

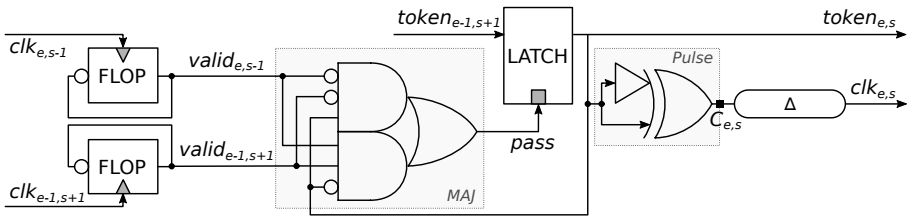


Figure 3.8 Token Unit in the Mini-Mips

stage $s+1$ in the previous EU $e-1$, are completed. The first condition translates the normal behavior of multicycle EUs: a given stage can only start when the previous stage has finished. The second condition results from the shifted-by-two stages organization used to implement the ILP in the Mini-Mips, as described in Section 3.4.2. The *valid* signals enforcing these two conditions—$valid_{e,s-1}$ and $valid_{e-1,s+1}$ respectively—are generated by toggle flip-flops using self-timed clocks—$clk_{e,s-1}$ and $clk_{e-1,s+1}$ respectively—generated from other TUs. Toggle flip-flops are needed here because of the multicycle nature of EUs. Indeed, since a stage receives only one clock pulse per token cycle, signals set during this stage cannot be reset until the next token cycle. Thus, the circuit enforcing the conditions for letting tokens pass in TUs must make use of the *valid* signals phase rather than their value, much like in 2-phase handshake protocols (see Section 2.1.2). This circuit, composed of two AND gates and one OR gate, behaves as a majority gate (MAJ): when both *valid* signals are in the same state the *pass* signal is asserted, thus letting the token pass, and when the $token_{e,s}$ signal is fed back, the *pass* signals is de-asserted, thus closing the latch. Note that this MAJ circuit is used for similar purposes in the Click Elements template [58], as described in Section 2.1.3.

Figure 3.9 shows the Delay Element architecture used in the Mini-Mips. It is composed of N stages—DE_0 to DE_{N-1}—, which can be sized to match any circuit delay. The *effective* length of the DE can be altered by the *opcode* signal, which allows to dynamically adjust an operation duration according to the instruction being processed. The *opcode* signal uses a *thermometer code* encoding: when the N bits are equal to 0 the clock signal travels along all the stages, from the *i_clk* input back to the *o_clk* output. If bit i is set to 1, then all stages from i to $N-1$ are bypassed. Using this DE architecture presents two main advantages compared with the default variable length DE presented in [30], where different delays are selected among multiple DEs by an N input MUX. First, the size of the DE increases linearly with the number of stages: adding one stage increases the delay by a fixed amount, and increases the size of the *opcode* signal by one bit. Then, a signal does not propagate through the DE stages passed the bypass stage, thus reducing dynamic power consumption of fast operations, compared with slower ones. However, although having a *variable length* DE is

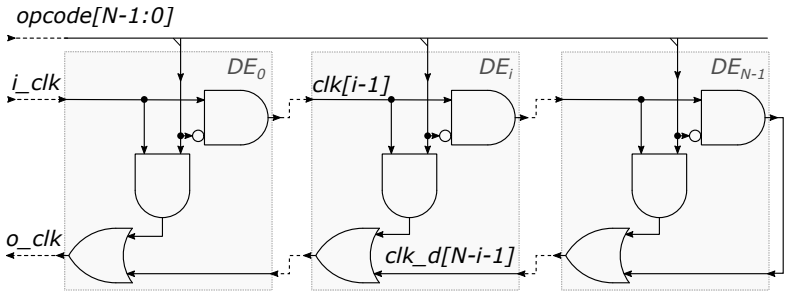


Figure 3.9 Delay Element in the Mini-Mips

a characteristic of the Octasic design style [35], as described in [30], the Mini-Mips does not make use of it—this DE architecture is used with fixed opcode values—because we have found that it adds a serious overhead to the definition of timing constraints.

To summarize, the Octasic-style asynchronous Mini-Mips microarchitecture differs from both standard synchronous and asynchronous-elastic paradigms. First, Instruction Level Parallelism is not implemented using pipelining. Rather, it relies on the coordination of multicycle EUs, in which the stages depends on one another in an organized network, as illustrated in Figure 3.6. Then, having a self-timed clocking scheme in multicycle EUs, which do not use handshaking protocols, distinguishes the Octasic-style asynchronous microarchitecture from a synchronous pipeline, as described in Section 3.3, as well as from an asynchronous elastic pipeline, as described in Section 2.1. Furthermore, the Octasic-style asynchronous Mini-Mips microarchitecture also differs from the original Octasic design style [35] because it uses an in-order mode of operation, as opposed to the out-of-order mode of operation employed in both the AnARM [35] and the Opus2 [29] processors. Finally, the Mini-Mips core performs the delay matching using DEs on the clocks paths, whereas the original Octasic design style [35] performs delay matching using DEs on the tokens paths [30, 31].

3.5 FPGA Implementation Methodology

The Mini-Mips cores presented in the previous sections—the synchronous microarchitecture, and the Octasic-style asynchronous microarchitecture—target a prototyping platform based on Xilinx 7-series FPGAs (see Section 3.6). These devices rely on the *Vivado Design Suite* software [96] for logical synthesis, Place and Route, Static Timing Analysis, bitstream generation, and FPGA programming. Also note that Vivado uses the Xilinx Design Constraints (XDC) format [97] for the definition of timing constraints, which is derived from the standard SDC format.

The implementation of the Mini-Mips synchronous microarchitecture on FPGA is standard,

```
attribute DONT_TOUCH of clk  : signal is "TRUE";
attribute DONT_TOUCH of clk_d: signal is "TRUE";
--
DE_i : LUT6_2
generic map (INIT => X"ECECECECOA0A0A0A")
port map (O6 => clk_d(N-i),
          O5 => clk(i),
          I0 => clk(i-1),
          I1 => clk_d(N-i-1),
          I2 => opcode(i),
          I3 => '1', I4 => '1', I5 => '1');
```

Figure 3.10 VHDL code snippet of a DE stage mapped into one LUT of 7-series FPGAs

thus it will not be discussed further. However, as we have seen in Section 2.4, the implementation of asynchronous circuits on FPGAs using standard EDA is challenging. First, asynchronous-specific modules cannot be automatically synthesized from high-level descriptions. We propose to design the Token Unit and the Delay Element at the gate level, and to manually map them onto the FPGA fabric (Section 3.5.1). Then, standard timing verification and timing-driven synthesis methods, which are the backbone of the standard design flow, are not directly compatible with asynchronous circuits. We propose a set of timing constraints (Section 3.5.2), in combination with an EDA flow (Section 3.5.3), that enable the timing-driven synthesis of the Octasic-style asynchronous Mini-Mips core on FPGA, and allows to use Static Timing Analysis to perform timing verifications.

3.5.1 Gate level design of asynchronous modules

The Token Unit and the Delay Element modules exhibit a dynamic behavior which prevent them from being automatically compiled from high-level descriptions using Vivado. Since these modules are not part of 7-series FPGAs primitive components, they cannot be inferred by the synthesis engine [98]. Thus, they must be designed at the gate level, and `DONT_TOUCH` directives must be used to prevent their automatic removal by synthesis optimizations.

Figure 3.10 shows the VHDL code snippet of a generic DE stage, composed of two AND gates and one OR gate as represented in Figure 3.9. To further optimize the DE implementation on Xilinx Configurable Logic Blocks (CLBs), we have manually mapped a DE stage into one *LUT6_2* Look Up Table (LUT) component using a dedicated configuration [99]. Relative location constraints [97] can then be used to force DE stages to be mapped to adjacent LUTs and CLBs. Figure 3.11 shows the VHDL code snippet of a typical EU stage. It can be seen as a synchronous pipeline stage (*STAGE* process) in which the clock is locally generated in

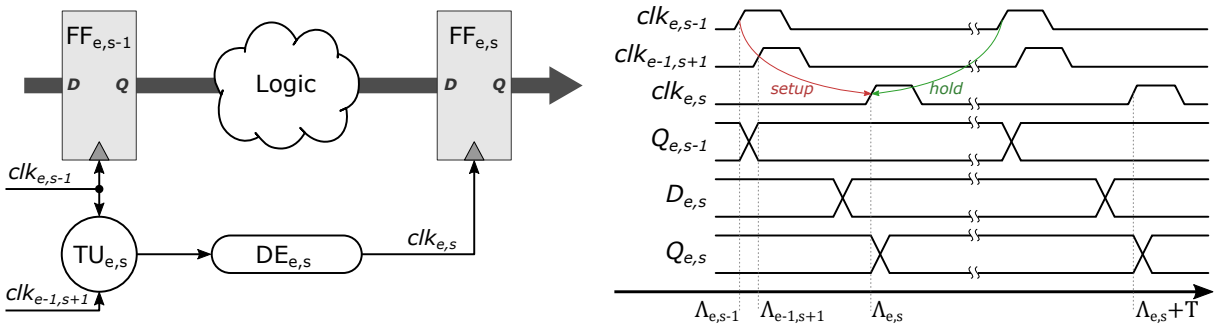
```
TU:token_unit
generic map (DE_LENGTH => PULSE_WIDTH)
port map (i_token => token(e-1),
          o_token => token(e),
          i_tog_a => clk_d(e-1,s+1),
          i_tog_b => clk_d(e,s-1),
          o_clk   => clk(e,s));
DE:delay_element
generic map (DE_LENGTH => MATCH_DELAY)
port map (i_opcode => opcode,
          i_clk    => clk(e,s),
          o_clk    => clk_d(e,s));
STAGE:process(clk_d(e,s), rst)
begin
  if rst = '1' then
    -- Reset registers
  elsif rising_edge(clk_d(e,s)) then
    -- Update registers
  end if;
end process stage;
```

Figure 3.11 VHDL code snippet of a typical EU stage

the TU (*TU* component), and locally timed in the DE (*DE* component). Using this model eases the design of Octasic-style asynchronous processors by lowering the barrier with the synchronous mentality, and facilitates their integration with standard synthesis engines.

3.5.2 Timing constraints

Static Timing Analysis plays an essential role in two phases of the standard FPGA implementation flow: *i)* during the timing verification phase, where it determines timing margins using slack metrics and detects timing violations; and *ii)* during the Place and Route phase, where timing-driven optimizations influence the placement and the routing of the system (in contrast with the ASIC design flow, the timing has little influence on circuit synthesis on FPGAs, because the gate sizes are determined by the FPGA fabric). Although the Octasic-style asynchronous Mini-Mips microarchitecture uses flip-flops—edge-triggered flip-flops are better suited for STA than level sensitive latches—, its self-timed clocking scheme is difficult to translate into *timing constraints* that would be properly interpreted by synchronous-oriented STA engines. Figure 3.12 shows the clocking scheme of a typical Execution Unit stage. Figure 3.12a represents the EU stage generic circuit model, which shows the timing paths for the data transfer between two registers and illustrates the local clock dependencies in TUs. Figure 3.12b represents the associated timing diagram, where the source clock, $clk_{e,s-1}$, triggers the source data, $Q_{e,s-1}$, from the source register and the destination clock,



(a) EU stage (b) Timing diagram

Figure 3.12 Clocking scheme of a typical Octasic-style asynchronous EU stage

$clk_{e,s}$, saves the propagated data, $D_{e,s}$, into the destination register. Although the timing constraints of a typical Octasic-style asynchronous EU stage are different from those of a synchronous pipeline stage, they are still based on setup and hold relationships, as discussed in Section 2.5.

The hold constraint of EU stages, as defined in Section 2.5, are always met. Indeed, as illustrated in Figure 3.12b, the hold constraint protects against data overwriting by preventing the next source clock ($clk_{e,s-1}$, at the next token cycle) to send data that could be captured by the active destination clock ($clk_{e,s}$). However, following the chain of clock dependencies, the next source clock cannot fire before the active destination clock, which architecturally prevents hold conditions from being violated. In Chapter 4 we will see that this definition of the hold constraint is incomplete: other hold conditions exist, which must be appropriately constrained. In the remaining of this chapter, we only consider hold constraints defined between successive edges of a clock separated by a token cycle (see Figure 3.12b), which are always met. The setup constraint ensures that data launched by the source clock are safely captured by the destination clock. The *setup slack* ($S_{e,s}$) of a typical EU stage is defined as the difference between the destination clock arrival time, $\Lambda_{e,s}$, and the source clock arrival time, $\Lambda_{e,s-1}$, plus the propagation delay through the combinational logic (CL), $\delta^{\text{cl}}_{e,s}$:

$$S_{e,s} = \Lambda_{e,s} - \left(\Lambda_{e,s-1} + \delta^{\text{cl}}_{e,s}\right) \tag{3.1}$$

This definition of the slack highlights the impact of the clock dependencies on timing, where a clock arrival time $\Lambda_{e,s}$ is defined as the sum of the arrival time of the latest dependent clock and the delay of the DE $\delta^{\text{de}}_{e,s}$:

$$\Lambda_{e,s} = \delta^{\text{de}}_{e,s} + \max\left(\Lambda_{e,s-1}, \Lambda_{e-1,s+1}\right) \tag{3.2}$$

Thus, matched delays should not only be tuned based on the combinational logic worst-case delay, but should also include timing information from dependent clocks. Merging relation (3.1) and (3.2) results in the following conditional slack definition:

$$S_{e,s} = \begin{cases} \delta^{\text{de}}_{e,s} - \delta^{\text{cl}}_{e,s} \text{ if } \Lambda_{e,s-1} \geq \Lambda_{e-1,s+1} \\ \delta^{\text{de}}_{e,s} + (\Lambda_{e-1,s+1} - \Lambda_{e,s-1}) - \delta^{\text{cl}}_{e,s} \text{ else} \end{cases} \tag{3.3}$$

A complementary way of looking at this is to consider that predicting the delay between two consecutive clocks (*i.e* a stage delay $\delta^{\text{stage}}_{e,s}$) consists in determining the arrival time of dependent clocks in addition to evaluating the DE delay. Thus, the delay between two clocking events is a recursive function of dependent clocks arrival time:

$$\delta^{\text{stage}}_{e,s} = \delta^{\text{de}}_{e,s} + \max\left(\Lambda_{e,s-1}, \Lambda_{e-1,s+1}\right) - \Lambda_{e,s-1} \tag{3.4}$$

As we said in Section 2.5, common strategies to implement asynchronous circuits using synchronous EDA use Relative Timing Constraints [25,82,87,91,93]. The RTC formalism allows to *i)* define the timing constraints of an asynchronous design; and *ii)* translate these constraints into standard SDC commands compatible with conventional STA tools, which enable both timing-driven optimizations and timing verification. Intuitively, the main incompatibility between synchronous and asynchronous timing schemes (as found in BD and Octasic-style asynchronous circuits) comes from their different approaches towards delay matching: the former uses a periodic clock signal, thus the constraint is relative to a *constant*—the clock period has a fixed value—while the later uses a delayed pulse signal, thus the constraint is relative to a *variable*—the delay of the DE varies with implementation steps. The basic idea behind the translation of RTCs, as defined in Section 2.5.2, into standard SDC commands is to decouple data and control relations by splitting the constraint: the ordering of two competing paths arrival time is constrained by comparing the delay of each path to a constant. A condition such that $\delta_{\text{data}} < \delta_{\text{control}}$—signals through the datapath must arrive before those of the control path—becomes $\max(\delta_{\text{data}}) \leq \Delta$, and $\min(\delta_{\text{control}}) > \Delta$, where Δ represents a constant delay and plays a role similar to a clock period. These two conditions are then translated into the following SDC commands: `set_max_delay` Δ `-through data` and `set_min_delay` Δ `-through control`. When considered individually, each stage of the Octasic-style asynchronous Mini-Mips could be timed with this approach. However, these constraints are not able to capture the effect of the clocks dependencies on the stage delays, as defined in Relation 3.4, and thus fail to properly reflect the global timing of the system. Instead, we propose to use a set of *clock definitions* as the basis of the timing constraints.

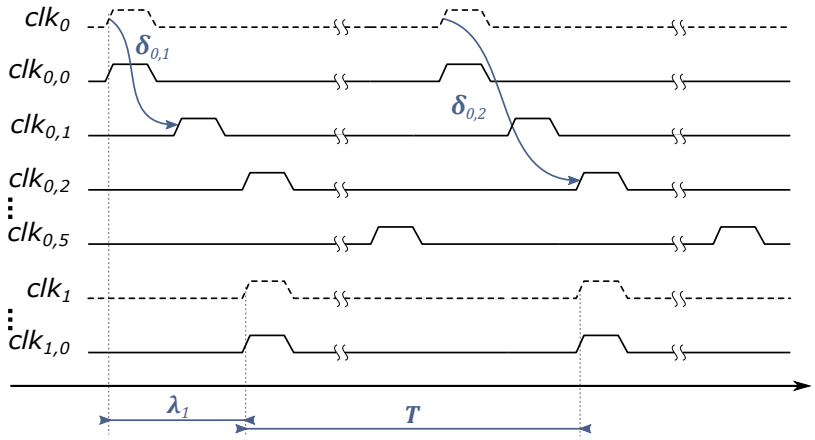


Figure 3.13 Timing diagram showing how the clocks in the Octasic-style asynchronous Mini-Mips are defined in the timing constraints

Indeed, we have found that they fit better with the timing conditions that we have derived from the typical Octasic-style asynchronous EU stage, as depicted in Figure 3.12. The idea is to provide to the STA tool an ensemble of clock definitions that corresponds to the actual self-timed clocks organization in the system.

The proposed timing constraints are described in Algorithm 1, and are illustrated by the timing diagram of Figure 3.13. For each EU, a *reference* clock (labelled clk_e) is defined as a *virtual clock*—*i.e.* a clock that is not tied to an actual pin or port in the circuit—using the `create_clock` XDC command. Reference clocks are illustrated in Figure 3.13 using dashed lines. For each EU stage, *local* clocks (labeled $clk_{e,s}$) are defined relatively to their associated reference clock using the `create_generated_clock` XDC command. Local clock definitions are tied to the pulse generator output pin of each TU (see Figure 3.8), labeled $C_{e,s}$. They are represented in Figure 3.13 using plain lines. The first reference clock, clk_0, is arbitrarily defined as the starting point from which all the other clocks definitions are derived. The next reference clock, clk_1, starts after an interval λ_1 calculated from clk_0 using the GET_DELAY function, *etc.* Similarly, the clock period T is determined by using the GET_DELAY function between two consecutive events of a local clock (*e.g.* $C_{0,0}$), and local clocks are shifted from their respective reference clock by a value $\delta_{e,s}$ determined with the GET_DELAY function. This function uses the STA engine to compute the delay between clocking events, from the delay of the clock path through their respective DEs, using Relation (3.3).

The remaining part of Algorithm 1 is responsible for modifying the setup and hold relationships of the clocks, which should reflect the timing scheme defined in Figure 3.12. This is the role of the `set_multicycle_path` and the `set_case_analysis` constraints. A

Algorithm 1: Timing constraints of the Octasic-style asynchronous Mini-Mips core

Data: Let T and P be the clocks period and pulse width, and $C_{e,s}$ be a TU clock pin.

Function *SET_CONSTRAINTS* **is**
  $T \leftarrow$GET_DELAY$(C_{0,0}, C_{0,0})$
  foreach e *in* EUs **do**
    $\lambda_e \leftarrow e = 0?\ 0 :$ GET_DELAY$(C_{0,0}, C_{e,0})$
    create_clock -name clk_e -period T -waveform $\{\lambda_e,\ \lambda_e + P\}$
    foreach s *in* $STAGES$ **do**
      $\delta_{e,s} \leftarrow s = 0?\ 0 :$ GET_DELAY$(C_{e,0}, C_{e,s})$
      create_generated_clock -name $clk_{e,s}$ -source clk_e -shift $\delta_{e,s}$ [get_pin $C_{e,s}$]
      set_multicycle_path -setup -end 0 -rise_from $clk_{e,s-1}$ -rise_to $clk_{e,s}$
      set_multicycle_path -setup -end 0 -fall_from $clk_{e,s-1}$ -fall_to $clk_{e,s}$
      set_case_analysis $opcode_{e,s}$ -from $clk_{e,s}$
    end
  end
end

`set_multicycle_path` constraint alters the number of clock cycles normally required to propagate data from a source register to a destination register. By default, sequential datapaths are timed using a *one-cycle* configuration. That is, one clock period separates the launch edge, which triggers data from a source register, from the capture edge, which triggers data into a destination register. By contrast, a typical Octasic-style asynchronous EU stage should be timed using a *zero-cycle* configuration. That is, the capture edge should be evaluated within the same period as the launch edge, as illustrated in Figure 3.12. These timing exceptions adjust the edge used for setup analysis to zero cycle after the first launch edge, which ensures that the datapath is constrained between the first edge of the launch clock and the first edge of the capture clock, that is, the delayed version of the launch clock. Note that, although hold conditions are always met, rise-to-rise and fall-to-fall edges are specified to ensure that the hold relationships are correct. A `set_case_analysis` constraint applies logic constants on pins to restrict signals propagation through the design. Using this command, a signal can be declared as having a specific value to the timing engine. When applied to the *opcode* signal of the DE (see Figure 3.9) it allows to configure the effective DE length from the perspective of the timing analysis engine.

Part of the modifications made to the Octasic-style asynchronous Mini-Mips, compared with the original Octasic design style [35], result from the need to being compatible with this combination of timing constraints. Indeed, in the original Octasic design style [35], DEs are used on the tokens paths, and the delayed version of the tokens are used to generate local clocks. However, this prevents the multicycle exception from begin properly used, as matched delays are not on the clock propagation paths. By contrast, in the Mini-Mips, local clocks

are generated using non-delayed version of the tokens, and DEs are used on the clocks paths. Finally note that Algorithm 1 is implemented, using the *Tcl* language, as part of the overall implementation flow.

3.5.3 Implementation flow

The SET_CONSTRAINTS procedure (Algorithm 1) translates the timing conditions of the Octasic-style asynchronous Mini-Mips core in a set of timing constraints that serves two purposes: *i)* reflecting the clocks organization of the core; and *ii)* adapting the setup and hold relationships of each sequential stage in the design. However, in contrast with a synchronous design, in which the timing constraints are consistent throughout the stages of the implementation flow, the timing information of an Octasic-style asynchronous design must be updated after each implementation phase. Indeed, the values of λ_e and $\delta_{e,s}$, which are computed from the delay of the timing path between two clock source pins in the circuit using STA, must be updated after synthesis, placement, and routing, to reflect the updated delays of the timing paths after each implementation step. Figure 3.14 illustrates the proposed implementation flow that accounts for this particularity. It computes the SET_CONSTRAINTS procedure after each of the following steps:

1. **Elaboration**—The elaboration step consists in compiling the design and building a first circuit from the HDL description using generic gates. After this step, a procedure called INIT_TIMING initializes the timing information of the clocks using user-defined values instead of STA results.

2. **Synthesis**—The synthesis step performs the mapping of generic gates within the FPGA fabric. The design techniques developed in Section 3.5.1 are the most useful during this step. Then, the SET_CONSTRAINTS procedure updates the clocks definitions based on timing information evaluated by a procedure called UPDATE_TIMING, which rely on STA results. Note that, at this stage, routing delays are not yet taken into account. Thus, the clocks timing information are far from being complete, as the routing delay on FPGA can account to up to 50 % of the total delay of a path. Finally, a global timing analysis performed on the system allows to evaluate slack metrics for each EU stage. When a timing constraint is not met, a DE stage is added to the failing clock path, and the implementation process is restarted.

3. **Placement**—First, floorplan is defined by the designer. However, unlike floorplanning-based implementation strategies (see Section 2.5.1), in which delay variations are contained by constraining the relative placement of cells, the proposed implementation flow

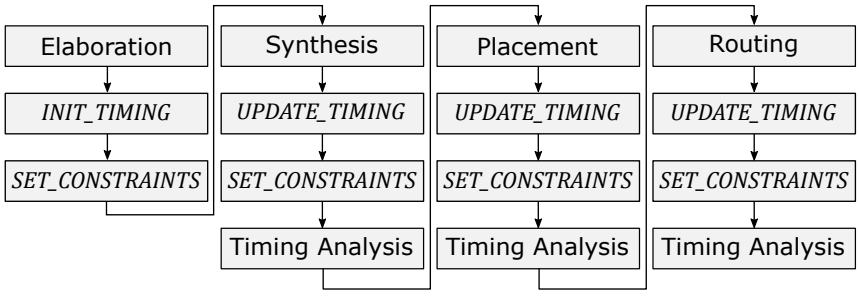


Figure 3.14 Octasic-style asynchronous Mini-Mips FPGA implementation flow

only uses area constraints as an additional optimization option. The relative placement of cells is automatically chosen by the PaR engine (except for the DE stages) which tries to minimize slack metrics defined for each EU stage.

4. **Routing**—Similarly, the PaR engine uses the post-placement clocks definitions, updated by the SET_CONSTRAINTS procedure, to perform a timing-driven routing of the system. Note that the routing of DEs are not minimal. Instead, these routes are chosen for each DE based on an evaluation of the routing delay that minimizes the slack metrics defined for each EU stage.

3.6 Prototyping Platform

As mentioned in the previous sections, the targeted hardware emulation platform for the Mini-Mips processors are Xilinx 7-series FPGAs. Among the many Xilinx FPGAs boards available, we selected the *ZC*706 board [100] as the prototyping platform of choice. As illustrated in Figure 3.15, this board has two key features that can be leveraged to enable runtime performance and power consumption monitoring of the Mini-Mips processors:

1. **Hardware/Software interface**—The ZC706 board is built around a ZYNQ SoC, composed of two parts: a Programmable Logic (PL) side, which is a regular 7-series FPGA; and a Processing System (PS) side, which is an ARM Cortex A9 processor. Both sides of the SoC can communicate with each other through AXI interfaces, shared cache memories, and interrupt lines. It allows hardware implemented on the PL to interact with a software stack running on the PS.

2. **Power supply control and monitoring**—The ZC706 hosts a *power management system* based on the Texas Instruments *UCD90120A* power supply sequencer and monitor [100]. In addition to controlling the power supply of the PS and the PL sides of the

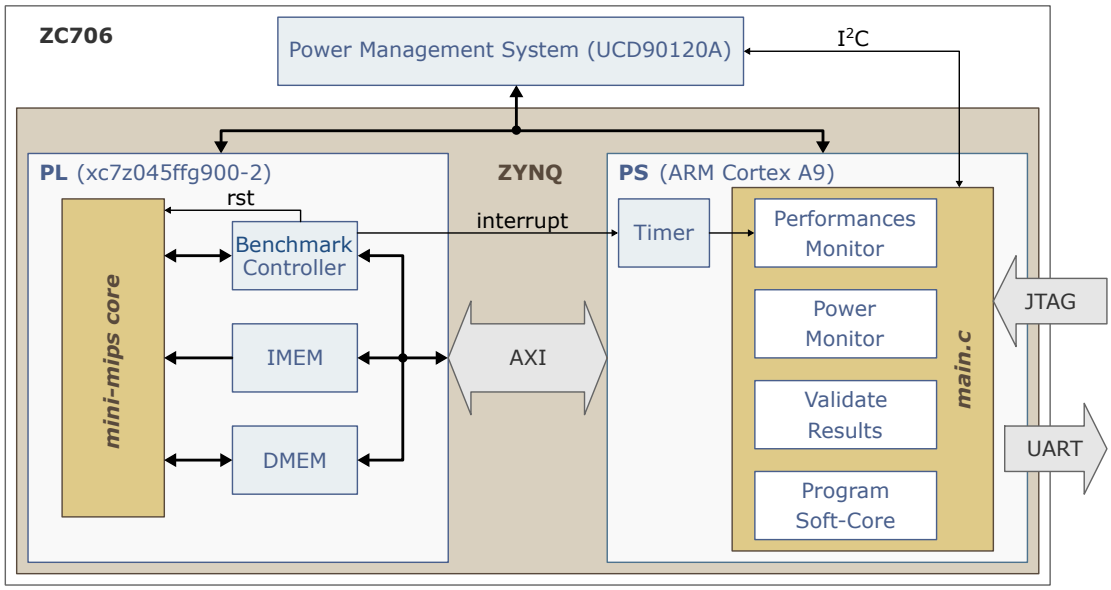


Figure 3.15 Prototyping platform for the Mini-Mips cores based on a Xilinx ZC706 board

ZYNQ, this power management system allows to monitor a set of power supply units through online sensors for voltage, current and temperature measurement. It is interfaced with the PS through an I^2C interface, that allows to communicate the samples directly to a software stack running on the PS.

We used these features in the context of prototyping the Mini-Mips cores. They facilitate the programming and the monitoring of the Mini-Mips while running the Fibonacci benchmark. As represented on Figure 3.15, on the PL side of the ZYNQ resides the Mini-Mips core—either with the synchronous or the Octasic-style asynchronous microarchitecture—and the memories (IMEM and DMEM). The *benchmark controller* is a synchronous system responsible for controlling and monitoring the Mini-Mips by communicating with the software stack running on the PS using the AXI interface. It is synchronized with the Mini-Mips using asynchronous FIFOs (not represented on Figure 3.15). On the PS side of the ZYNQ resides the software stack responsible for the overall coordination of operations. The *program soft-core* function fill the IMEM memory with the sequence of instructions of the compiled benchmark program. This sequence of instruction is first compiled on a host computer, as described in Section 3.1, then added to the *program soft-core* function as a string, and finally compiled as part of the software stack running on the PL. Programming the Mini-Mips consists in parsing the instruction sequence string and filling the IMEM memory with it through the AXI interface. The *validate results* function simply computes the Fibonacci algorithm on the PS and compares the results with the benchmark running on the Mini-Mips core.

The *performances monitor* function uses the internal timer of the ARM core on the PS to measure the duration of the benchmark execution on the Mini-Mips cores implemented on the PL. During the benchmark execution, the *power monitor* function records the power consumption of the PL by communicating with the power management system using the I^2C interface. Finally, all of these functions continuously communicate their information with a host computer, during the execution of the benchmark, using the UART interface.

The Mini-Mips cores are implemented using the *out of context* synthesis flow of Xilinx [98]. The goal is to decouple the synthesis and implementation flow of the Mini-Mips from those of the rest of the hardware implemented on the PL (*i.e.* the memories and the benchmark controller). In particular, the iterative process through which the Octasic-style asynchronous Mini-Mips is implemented, as described in Section 3.5, can be ran independently from the rest of the design. During the implementation of the rest of the design, the Mini-Mips is seen as a black-box. IMEM and DMEM are true dual port RAMs, that are interfaced on one side with the Mini-Mips, and on the other side with the AXI interface. The experimental protocol is composed of three steps. In *step 1*, the compiled benchmark program is loaded into the Mini-Mips. In *step 2*, the benchmark is started on the Mini-Mips by releasing the *reset* signal. During this time, power consumption values are gathered using the power management system. When the benchmark ends, an interrupt is sent to the PS to trigger step 3. Finally in *step 3*, memories are read by the PS to validate results and compute the performance figures.

Algorithm 2 describes the program running on the PS. The Mini-Mips core—either with the synchronous or the Octasic-style asynchronous microarchitecture—is first placed in *PROGRAM* mode, using the `set_mode()` function. A compiled version of the Fibonacci benchmark, is loaded into the instruction memory using the AXI interface. The Mini-Mips is then placed in *RUN* mode, and the benchmark starts running. The start time is taken with the `get_time()` function, using the PS timer. As long as the benchmark is running on the Mini-Mips, the power consumption of the PL is measured with the `power_monitor()` function, using the power management system. When the benchmark has finished running, an interrupt is sent from the benchmark controller to the PS, that ends the power monitoring loop. The number of instructions completed by the Mini-Mips while running the benchmark is sent from the Mini-Mips to the PS by the benchmark controller through the AXI interface. It is used along with the execution time to determine the Mini-Mips performances. Then, the Mini-Mips is placed in *SLEEP* mode, that is, the reset signal is enabled. The power consumption of the PL is monitored for a time proportional to the execution time using the `sleep_run_ratio` variable. The Mini-Mips is placed in *RUN* and *SLEEP* modes alternatively 10 times before the end of the program.

Algorithm 2: Benchmark control and Mini-Mips monitoring

```
1  sleep_run_ratio ← 1
2  set_mode(PROGRAM)
3  for i = 0 to 10 do
4      set_mode(RUN)
5      t0 ← get_time()
6      repeat
7          power_monitor()
8      until dut_interrupt()
9      t1 ← get_time()
10     run_time ← t1 − t0
11     perf ← get_inst_count() / run_time
12     set_mode(SLEEP)
13     sleep_time ← 0
14     repeat
15         power_monitor()
16         t2 ← get_time()
17         sleep_time ← t2 − t1
18     until sleep_time = run_time × sleep_run_ratio
19 end
```

3.7 Experimental Results

This section presents the results obtained from post-implementation (*i.e.* post Place and Route) timing analysis, and from runtime measurements of the Mini-Mips performances and power consumption. First, Section 3.7.1 compares the FPGA resources utilization of both microarchitectures. Then, Section 3.7.2 focuses on validating the proposed timing-driven implementation methodology, using the Vivado EDA software, as presented in Section 3.5.2, by comparing post-implementation timing simulations with STA reports. In particular, we compare the clock timing scheme observed in a timing simulation, with the clocks timing scheme obtained by applying the SET_CONSTRAINTS procedure (Algorithm 1) with the STA tool. In addition, we show the benefits of using the proposed timing-driven implementation flow, compared with a non timing-driven implementation flow, by evaluating the clocks slacks and the performances of the Octasic-style asynchronous core after each implementation step. Finally, Section 3.7.3 compares the Mini-Mips—synchronous *vs.* Octasic-style asynchronous—experimental results obtained from runtime measurements of the processors performance and power consumption when running the Fibonacci benchmark on the hardware emulation platform, as presented in Section 3.6.

Table 3.1 Resources utilization summary

Microarch.		**SLICES**	**LUTs**	**FLOPs**	**BRAMs**
Synchronous	#	585	1 296	1 371	2
	%	1,07	0,54	0,31	0,37
Octasic-style	#	1 050	2 497	1 912	2
	%	1,92	0,90	0,44	0,37

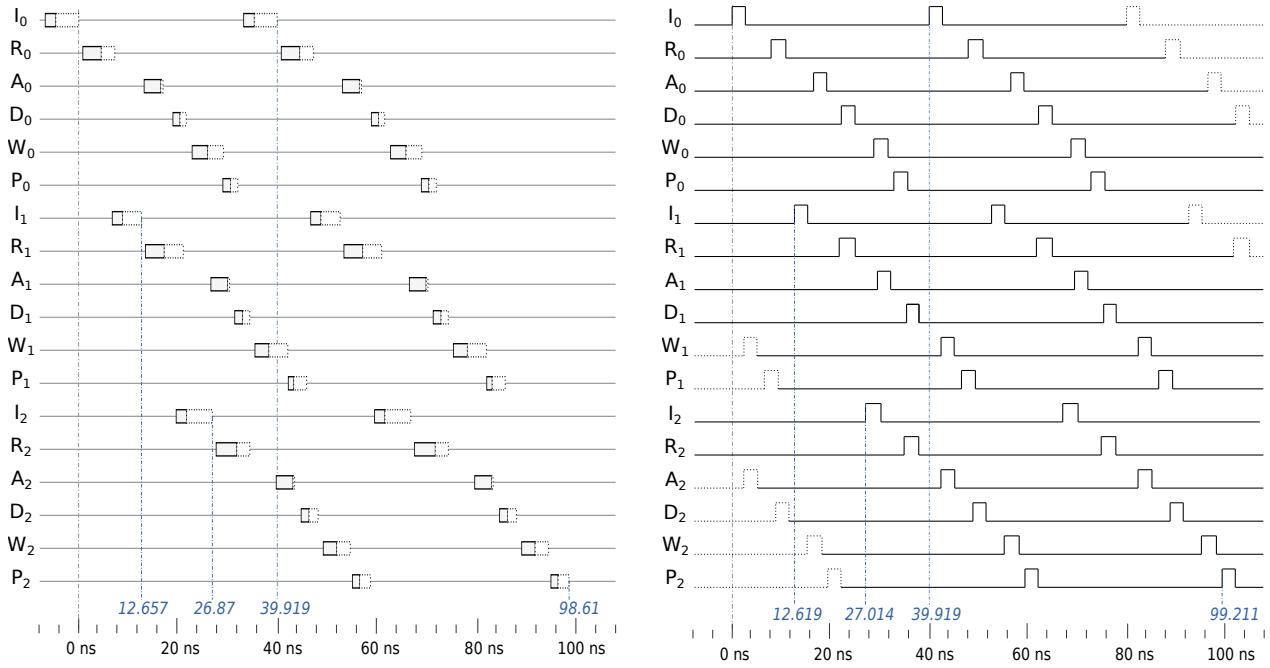
3.7.1 Resource utilization

Table 3.1 is a summary of the FPGA resource utilization. It shows that both microarchitectures use a small amount of the overall resources available, with less than 2 % of the slices being occupied by the core logic. It also shows that both microarchitectures use two Block RAMs (a 1 024 kB instruction and data memory are used in both cases). Finally, it shows that the Octasic-style asynchronous microarchitecture uses almost 2× more logic than its synchronous counterparts. This is explained by the inherent redundancy of stage registers, and associated control logic, across Execution Units. Note also that the space occupied by the clock generation mechanism (token-ring) is an overhead that does not have its counterpart (PLL) evaluated in the synchronous statistics.

3.7.2 Post-implementation timing analysis

Figure 3.16 shows experimental results generated after implementing the Octasic-style asynchronous Mini-Mips processor using the proposed implementation flow with Vivado. They compare the timing scheme of the clocks as seen by the Static Timing Analysis engine using the proposed timing constraints (Figure 3.16a), with the propagation of clocks signals as obtained in a timing simulation (Figure 3.16b). Both figures have the same x and y axes to ease the comparison. Each row represents an EU stage clock, and the origin (0 ns) coincides in both cases with the arrival time of the I_0 clock. Comparing STA with a timing simulation is relevant to evaluate the accuracy of the proposed design flow because the timing information comes form the same source (the standard-cells timing characterization file) in both cases.

- **Static Timing Analysis**—Figure 3.16a shows a timing diagram illustrating the Static Timing Analysis of the Octasic-style asynchronous Mini-Mips using the proposed set of timing constraints. Each bar represents a stage delay and is composed of two parts: the combinational logic delay (plain section), and the slack (dashed section). Each row is composed of two identical bars, that are separated from each other by the period T of the system. Note that only the setup slack is shown because, for the reasons mentioned in section 3.5.2, the hold constraint is always met.



(a) Static Timing Analysis

(b) Post-implementation simulation

Figure 3.16 Post-implementation STA and timing simulation results of the Octasic-style asynchronous Mini-Mips core implemented on a Xilinx 7-series FPGA

- **Post-implementation simulation**—Figure 3.16b depicts the post-implementation timing simulation of the Mini-Mips clocks. It shows how the self-timed clocks actually interact and behave in the circuit. This snapshot captures a very small window of the simulation, which has been shifted in order to make the rising edge of the I_0 clock coincide with the origin at 0 ns. It shows at least two edges for each clock, separated by the period T of the system. Note that clock edges that belong to the previous or the next token cycle are represented with dashed line.

We observe that the representation of the clocks using STA results and the timing simulation of the clocks signals exhibit very similar patterns. This indicates that the clock organization of the Octasic-style asynchronous Mini-Mips, as wired in the circuit, is accurately interpreted by the Vivado timing engine using the proposed set of timing constraints. It is confirmed by a quantitative comparison of selected clock arrival times (namely, I_0, I_1, I_2, and P_2), which are represented in Figure 3.16 with dashed cursors. Indeed, the period T of the system, which is evaluated as the arrival time of the second edge of the I_0 clock, is the same in both cases ($T = 39.919$ ns). Although other clocks exhibit slightly different values, their relative differences only vary between 0,3 % for I_1 and 0,6 % for P_2. These differences may be due to

the way delays are distributed in the SDF file and interpreted in simulation. (Here we refer to *negative-timing checks* which are discussed in more depth in Chapter 4).

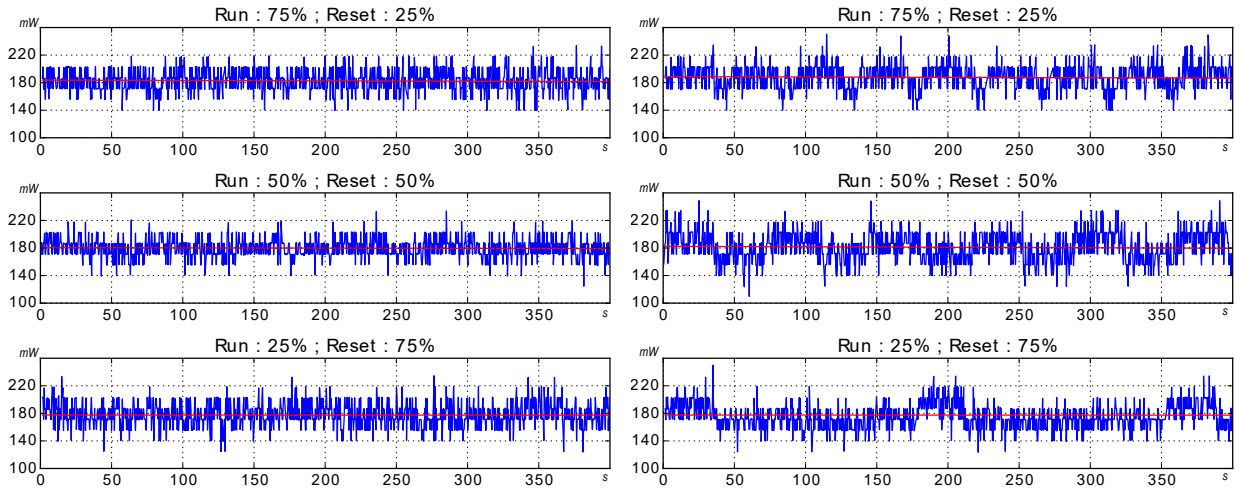
Figure 3.17 compares the STA results obtained after each implementation step—synthesis, placement, routing—using the proposed timing-driven implementation flow, with the results obtained using a non timing-driven implementation flow. Similarly with Figure 3.16, each bar represents a stage delay, composed of the combinational logic (CL) delay and the slack. However, Figure 3.17 hides the clocks organization, and only reports absolute values. The slack percentage, indicated at the top of each bar, along with the value of the period T of the system, provides additional information to analyze the influence of the proposed implementation flow on timing and performance. Both the timing-driven and the non timing-driven implementation flows were applied on the same processor design, using the same DE configurations. The proposed timing constraints were used in both cases to generate the results, as the clocks timing information cannot be accurately evaluated without it. However, in contrast with its non timing-driven alternative, the timing-driven implementation flow enforces design constraints using optimizations during placement and routing, with an emphasis on timing and performances.

In accordance with these explanations, we observe identical results post-synthesis, which then differ post-place and post-route. Indeed, as mentioned previously, the timing constraints have no impact during the synthesis step of an FPGA flow—as opposed to an ASIC flow—because the sizing of the gates is fixed by the FPGA fabric. As a general observation, also note that in contrast with a synchronous system, the overall performance of the system evolves with the implementation process. As mentioned, this is due to the fact that the clock management system is affected by the implementation as much as the rest of the circuit. Hence, the value of T increases as new timing information is added by the post-place and the post-route steps. The post-synthesis and the post-place results have no timing violations in both the timing-driven and the non timing-driven cases. However, the non timing-driven post-route results show three timing violations (A_0, A_1 and A_2 stages), whereas the timing-driven results exhibit none. This indicates that the proposed implementation methodology has successfully taken advantage of the timing-driven capabilities of the Vivado PaR timing-driven optimization algorithms to meet the targeted timing specifications. Moreover, in addition to meeting timing constraints, the proposed timing-driven implementation flow also improves the performance of the system with respect to the non timing-driven implementation. Indeed, we observe a 0 % improvement post-synthesis (same T values), a 2,7 % improvement post-place (T values differ by 1 ns), and a 10,2 % improvement post-route (T values differ by 4,1 ns). Finally note that for some clocks the slack and/or the stage delay obtained with the non timing-driven method is better than the one obtained with the timing-driven method. This

Figure 3.17 STA results showing the benefits of the reported timing-driven implementation methodology compared with a non timing-driven implementation flow

indicates that the balance of delays across stages is not trivial, but is rather conditional on improving the overall performance of the system, and meeting its timing specifications.

To summarize, by comparing the timing scheme of the clock definitions reported by Static Timing Analysis, with the clock signals observed in the timing simulation, Figure 3.16 provides both qualitative and quantitative evidence that the proposed timing constraints are accurate. Our implementation methodology leverages these timing constraints in combination with timing-driven directives for placement and for routing in order to meet the timing specifications and improve the performance of the system. By comparing STA results (slacks, stages delays, and system period) obtained after synthesis, placement, and routing using the timing-driven implementation flow, with the results obtained using the non timing-driven alternative, Figure 3.17 provides evidence that the proposed implementation methodology meets these expectations.



(a) Synchronous (b) Octasic-style Asynchronous

Figure 3.18 Power consumption of the Mini-Mips processors while running the Fibonacci benchmark (time in s, power in mW). The platform alone consumes approximately 160 mW.

3.7.3 Runtime performance and power consumption

Using the prototyping platform described in Section 3.6, we monitored the runtime performances and power consumption of both processors when running the Fibonacci benchmark in a loop, as described in Algorithm 2. The Octasic-style asynchronous Mini-Mips reaches performances similar to the synchronous microarchitecture with a 80 MHz clock frequency. Indeed, the Octasic-style asynchronous microarchitecture achieves 53,26 Millions Instructions Per Second (MIPS), while the synchronous microarchitecture achieves 58,02 MIPS. Figure 3.18a and Figure 3.18b show the average (red line), and the dynamic (blue line) power consumption of the synchronous and the Octasic-style asynchronous microarchitecture respectively. They result from the program depicted in Algorithm 2. Three sets of measurements have been realized for each microarchitecture, with a different *RUN/SLEEP* ratio of 25 %, 50 % and 75 %. The largest synchronous microarchitecture average power consumption was observed at 75 % run time with 182,5 mW. It drops to 179,5 mW at 50 % run time and 177,2 mW at 25 % run time. The Octasic-style asynchronous microarchitecture average power consumption is maximum at 75 % run time with 187,6 mW. It drops to 180,4 mW at 50 % run time and 173,4 mW at 25 % run time. These results confirm what we expected from the Octasic-style asynchronous microarchitecture. The power consumption drop in reset mode can be explained by the token-ring mechanism. When the reset signal is enabled, latches in TUs provide steady signals to their outputs, which prevents self-timed clocks from being released. It can be seen as a built-in clock-gating mechanism.

According to these results, when run-time dominates, the Octasic-style asynchronous microarchitecture consumes more power. When run-time and sleep-time are equal, both microarchitectures have a similar power/performance ratio. When the activity is dominated by sleep-time, the Octasic-style asynchronous microarchitecture consumes less power on average than its synchronous counterparts. However, these results must be analyzed with care. Despite the efforts put to make fair comparisons between both microarchitectures, some aspects are biased in favor of one microarchitecture or the other. Indeed, on one hand, because the synchronous microarchitecture does not implement clock-gating, its activity decreases less in sleep mode than the Octasic-style microarchitecture, in which the token-ring plays a similar role as clock gating. On the other hand, the clock generation mechanism of the Octasic-style asynchronous microarchitecture in Token Units is part of the measured power consumption, whereas the clock generation mechanism of the synchronous microarchitecture (PLL) is excluded from these measurements, as it does not use the same power supply in the prototyping platform. Moreover, despite the fact that self-timed clocks are locally generated, they still use the FPGA clock-trees to drive the registers. One has to keep in mind that FPGA fabric and tools are designed towards optimizing synchronous system performances. Despite our efforts to make the Octasic-style asynchronous design compliant with this platform, it remains a somewhat unfair environment for such designs. Nevertheless, being able to use FPGAs as a fast prototyping environment to perform in depth functional verification of Octasic-style asynchronous processors, that provide performance and power consumption insights, is a more significant achievement than the absolute results themselves.

3.8 Limitations

The Mini-Mips design experiments have provided interesting results, such as the in-order design trade-offs, the timing-driven implementation methodology, and the FPGA emulation platform. At the same time, designing the Mini-Mips processors has enabled to uncover important limitations of the Octasic-style asynchronous microarchitecture and its associated design flow. In the following, we summarize the achievements of the Mini-Mips project and we discuss its limitations in more depth. We use the experience gained from designing the Mini-Mips to address some of these limitations in Chapter 4, where we introduce a more formal definition of the Octasic-style asynchronous microarchitecture that we call *KeyRing*, and in Chapter 5, where we use the KeyRing microarchitecture and its associated design flow to design RISC-V processors.

3.8.1 Design

Designing the Mini-Mips using an empirical approach, based on the reverse engineering of the Opus2 and the AnARM processors, has greatly improved our understanding of Octasic design principles. The additional constraint of designing an in-order processor—although in-order processors are less complex than out-of-order processors—has allowed to take design decisions that diverge from the original Octasic design style [35], thus avoiding the pitfall of copying the design without understanding the reasons behind important design trade-offs. Indeed, as in-order operations are usually the foundation of out-of-order operations, we believe that designing the in-order Mini-Mips has contributed to uncover the underlying design principles of Octasic processors. Yet, many questions still remain—starting with the link between the number of Execution Units, the number of tokens, the organization of tokens dependencies within the token-ring, and the processor performances—, and it is clear at this point that the empirical approach used in the Mini-Mips project has reached its limits to answer them. We address this limitation in Chapter 4, where we introduce the KeyRing microarchitecture as a generalization of the Octasic-style asynchronous microarchitecture, using a more theoretical approach. In particular, we will explore the trade-offs of token-rings having varying sizes and organizations in generic sequential circuits.

Then, in the process of designing Token Units (see Section 3.4.3), we have found that alternative architectures may be worth exploring. Indeed, the combination of the toggle flip-flops and the latch to control the propagation of the token can be simplified, as we observed that the state variations of these state-holding elements are symmetrical. Thus, the KeyRing circuits studied in the following will also include the evaluation of alternative TUs architectures.

Moreover, with the experience gathered from designing the Mini-Mips, we reckon that FPGAs are not particularly well adapted to evaluate the characteristics of the Octasic-style asynchronous microarchitecture. First, since FPGAs fabric do not include asynchronous primitives, the resulting design is far from being optimal. In particular, the lack of Delay Elements and the structure of Xilinx CLBs leave the designer with limited choices: we have tried using LUTs—but a LUT is essentially a memory, how good a DE can it emulate ?—and carry-chains—but their unit delay is too small compared to typical combinational logic delays, resulting in very long DEs. In addition, we were never sure if the clocks traveling through LUT-based DEs used the dedicated clock routes of the FPGA. If not, then the clock structure of asynchronous circuits on FPGA may create routing congestions and lead to suboptimal designs. Cumulated, these issues result in an important disadvantage of the Octasic-style asynchronous Mini-Mips processor over its synchronous alternative, which hinders the relevance of their comparison. In addition, we have seen in Section 3.7.3 that the power con-

sumption measures on the prototyping platform are not precise enough to produce definitive conclusions about the energy-efficiency of the Octasic-style asynchronous microarchitecture. Using an ASIC design flow, most of these limitations would not exist. Indeed, typical ASIC kits contain enough standard-cells to produce efficient asynchronous circuits—despite having to replace specific gates, as seen in Section 2.1—, and ASIC design flows and power estimation tools are sufficiently advanced to produce relevant results that are more precise than what we achieved with our emulation platform. That is why, the KeyRing circuits proposed in the following chapters are based on the *TSMC65GP* 65 nm ASIC design kit.

Finally, we believe that the Mini-Mips ISA is too small. The idea behind using a small subset of the MIPS ISA was to reduce the complexity of the processors in order to focus on the particularities of the Octasic-style asynchronous microarchitecture, and to develop the implementation flow. Although this goal was achieved—using a full ISA from the beginning may have been too big of a challenge—, new processors designed with the proposed KeyRing microarchitecture must be based on a full ISA in order to produce meaningful results. Indeed, from todays standards, a processor developed in a research project must be able to implement a full ISA, supporting standard compilers and being able to execute standard benchmarks, such as Dhrystone [101] or Coremark [102]. The KeyV KeyRing processors presented in Chapter 5 implement the RV32IM variant of the RISC-V ISA, which satisfies these specifications.

3.8.2 Simulations

Simulating the Mini-Mips processors has highlighted the difficulties of simulating asynchronous circuits when compared to synchronous circuits, both in behavioral and in timing simulations.

Behavioral simulations of asynchronous circuits are more difficult to perform due to the lack of global synchronization signal. Indeed, as opposed to synchronous circuits, in which the clock is modeled in the test bench and fed to the circuit as an external signal, the clocks in asynchronous circuits are locally generated and timed, thus delays in the clocks paths must be taken into account to properly generate the clocking scheme of the circuit. This is at the root of two issues. First, behavioral models of asynchronous circuits must include either *i)* a gate level description of the control path—Token Unit and Delay Element in the Octasic-style asynchronous Mini-Mips—, in addition to the associated standard-cell library that includes timing models which should be properly loaded into the simulator (VHDL VITAL models are usually used for this purpose), or *ii)* an alternative description of the control path that models the delays using high-level timing constructs (such as `dest <= transport source after 1`

`ns;` in VHDL). This creates an overhead in the design process, and slows simulations down. Then, the behavioral models of asynchronous circuits may be malfunctioning when all DEs do not have the same delays. Indeed, as behavioral simulations assume no propagation delays in the circuit, clock edges are supposed to reach all the registers in design at the exact same time at each cycle. Otherwise—*i.e.* if the occurrence of clocks triggering the registers of the datapath vary (such as it is the case when DEs of adjacent stages have different delays), while the combinational logic has no propagation delay—signals in feedback paths may be overwritten, similarly to hold violations.

In contrast with behavioral simulations issues, which are easily addressed, timing simulations issues are more complex and involve more advanced concepts. Indeed, Section 3.7 does not show post-implementation simulations of the Octasic-style asynchronous Mini-Mips because timing simulations issues could not be solved. Basically, the problem comes from an incompatibility between the timing checks defined in the FPGA standard-cell library and the clocking scheme of the Octasic-style asynchronous Mini-Mips. Figure 3.19 shows how timing checks are performed in timing simulations, as described in the IEEE Standard for Verilog Hardware Description Language [103]. The setup & hold timing check values define a timing violation window with respect to the reference signal edge, *i.e.* the clock signal. Data shall remain stable during the timing window specified by the setup & hold values, otherwise a timing violation cause the registers to output a `X`. Timing checks are part of standard-cells library models written in verilog—using the `$setuphold` and the `$recrem` verilog tasks—, while the timing values are back-annotated by the simulator from the SDF file produced by a Static Timing Analysis tool. A positive value for both setup and hold times results in having the timing window straddling the clock edge, as represented in Figure 3.19. Negative values are allowed as long as their sum is positive [104]. The timing check algorithm of a simulator (Modelsim in our case) determines the proper delay values of the data and clock paths reaching a sequential element by evaluating the setup and hold timing check: a late data signal requires a negative hold value; conversely a late clock signal requires negative setup value. Depending on the STA tool and the options used to produce the SDF file, the DEs delays of asynchronous circuits are typically accounted for using negative timing checks. Given the large delay of DEs, the sum of the setup and the hold timing check values are often negative (*i.e.* the violation window does not exists). In this case, the negative timing check algorithm of the simulator will iteratively alter the timing checks by zeroing their values until it converges to a solution [104]. However, since this effectively alters the delay of the clock or the data path, there are good chances that the sequential element issue a timing error when it is triggered. The solution to this problem is to have a standard-cell library that properly handles negative timing checks, and to produce an SDF file such that values in timing

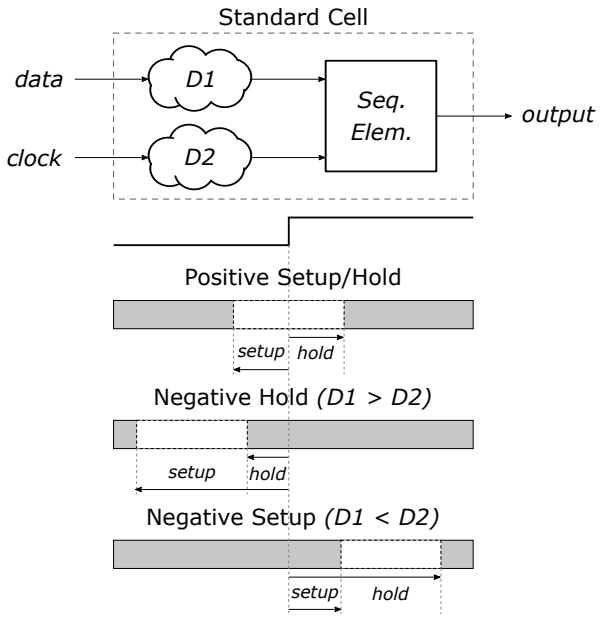


Figure 3.19 Setup & Hold timing checks intervals used in timing simulations [103]

checks always overlap. We have failed to achieve this using Xilinx FPGA timing libraries and STA tool, which is why timing simulations of the Mini-Mips post-implementation netlist has always failed. However, we have succeeded in making timing simulations of post-synthesis Octasic-style asynchronous circuits work using the TSMC65GP ASIC design kit, and the STA engine of the *Synopsys Design Compiler* synthesis tool, as described in Chapter 4.

3.8.3 Implementation methodology

The implementation methodology presented in Section 3.5—the combination of the timing constraints and the implementation flow—has allowed to perform the Static Timing Analysis of the Octasic-style asynchronous Mini-Mips. It has enabled *i)* the Place and Route algorithms to be aware of the core internal timing; and *ii)* to perform the timing verification of the core using STA rather than using timing simulations, which, to the best of our knowledge, was never accomplished before. Although the timing simulations issues uncovered in the previous section have prevented the validation of the STA results using post-implementation simulation timing checks, we used the information of the self-timed clocks timing simulation—not altered by the aforementioned issues—to validate the proposed timing constraints and STA method, as presented in Section 3.7.2. In addition, the final validation of the implementation methodology comes from the successful execution of the Fibonacci benchmark running on the Octasic-style asynchronous Mini-Mips core implemented on the hardware emulation

platform, which was a big achievement.

Yet, the Mini-Mips implementation methodology has many flaws. First, as was already mentioned, the timing checks definitions developed in Section 3.5.2 are incomplete. In Chapter 4, we develop a timing model for KeyRing circuits which provide extended setup and hold definitions. In particular, we present new hold definitions that allow to develop in-order KeyRing circuits having a *shifted-by-one* EUs stages organization. The *shifted-by-two* stages organization used in the Octasic-style asynchronous Mini-Mips (see Section 3.4.2), which limits the performances of the core, is used as a way to architecturally prevent these unconstrained hold violations having an effect. Moreover, having better setup and hold definitions enable improved timing constraints, which result in better STA and improved support for timing-driven optimizations. Then, in addition to having incomplete timing checks definitions, the method used to translate these definitions into XDC timing constraints is impractical. Indeed, similar to the method based on `set_min/max_delay` constraints, the timing constraints must be updated during the flow based on timing information gathered after each implementation step (synthesis, placement, and routing), as illustrated in Figure 3.14. However, to fully take advantage of the timing driven capabilities of the tools, timing information should be continuously updated during the timing optimizations phases, as both the clock path and the datapath delays are continuously being altered during this process. In Chapter 4, we develop a new design flow based on improved timing constraints, in which the datapath in constrained relatively to the clock path—instead of being constrained relatively to a constant value that is being updated periodically—, that allow the synthesis engine to concurrently optimize the clock paths and the datapaths using always up-to-date timing information.

CHAPTER 4 MODELING THE KEYRING MICROARCHITECTURE

In this chapter, we present the foundation of the KeyRing self-timed microarchitecture, derived from the analysis of the original Octasic design style [35] in Chapter 2, and from the Mini-Mips processor experiment detailed in Chapter 3. We propose the *Key Unit (KU)* as an alternative implementation of the Token Unit used in the Mini-Mips (Section 4.2), and the *KeyRing* protocol—a self-timed organization of multicycle Execution Units—as the basic structure of in-order KeyRing systems (Section 4.3). This new circuit template and the formal definition of the KeyRing protocol are leveraged by a novel timing model, which enables the implementation of KeyRing systems with timing-driven EDA flows (Section 4.4). These proposals are supported throughout this chapter by a generic representation of in-order sequential KeyRing circuits. As a practical example of such generic circuits, we have developed the Tribonacci system—a sequential circuit implementing the Tribonacci algorithm [44] in three stages—using the KeyRing microarchitecture (Section 4.1).

The KeyRing circuits and the timing models presented in this chapter target an ASIC design flow. As explained in Section 3.8, standard FPGAs have a restricted support for asynchronous circuits, which limits the extent of the comparisons that can be drawn up with synchronous alternatives. Although ASIC EDA tools are usually regarded as being the most advanced, they are also more complex. Most importantly, the fundamental difference between FPGA and ASIC design flows comes from the fact that FPGAs fabrics are composed of basic components and routing resources that have fixed sizes and locations. As a result, FPGA design flows have less room for optimizations than their ASIC counterparts, in which standard-cells exist in many sizes, and where the placement and the routing of the circuit is only constrained by the circuit specifications. Here, we only focus on the *front-end*—synthesis—part of the design flow, and we leave the *back-end*—Place and Route—for future works. This choice is motivated by the fact that, as opposed to FPGA design flows, the synthesis step of an ASIC design flow fully takes advantage of timing-driven optimizations (*e.g.* gate sizing), which allows to showcase the contributions of this work. We first used the General Purpose Design Kit (GPDK) 45 nm ASIC design kit from Cadence, in combination with the Cadence *Genus* synthesis tool, before moving to the TSMC65GP 65 nm ASIC design kit from TSMC, in combination with the Synopsys Design Compiler (DC) synthesis tool. The reasons to change of technology and of synthesis tool are: *i)* the GPDK45 kit cannot be manufactured, which may weaken the significance of the results compared with those of a "real" kit; and *ii)* we have found that the Synopsys DC synthesis tool better handles our timing-driven design flow, and, at the time of writing, the GPDK45 design kit is not compatible with Synopsys tools.

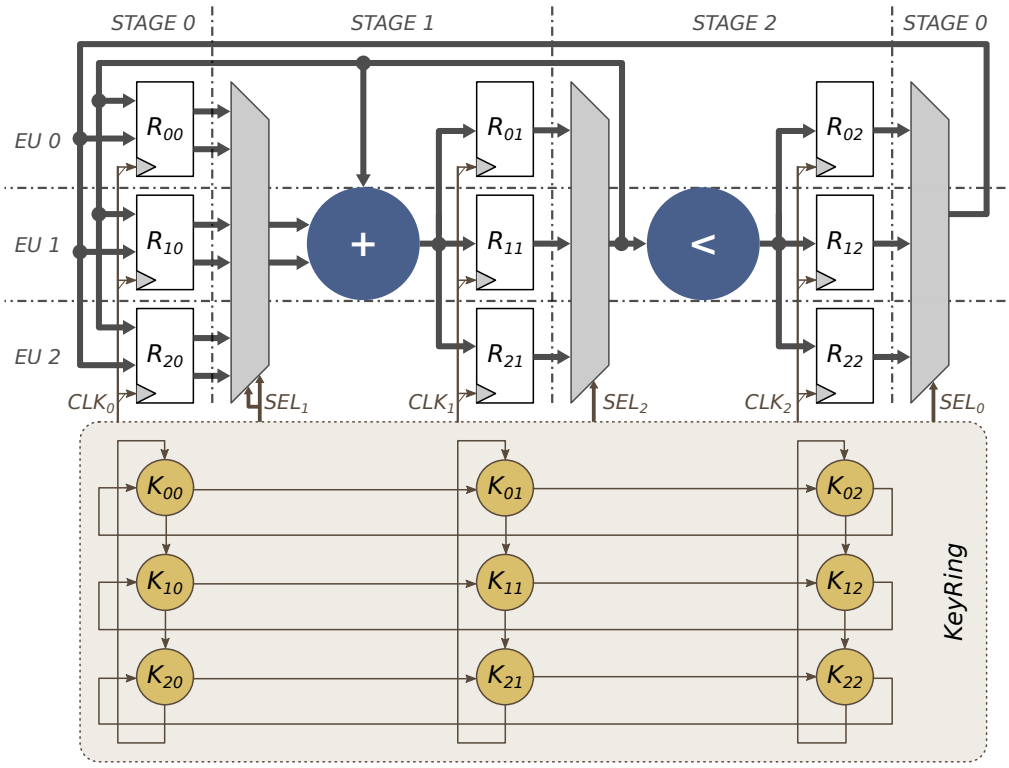


Figure 4.1 KeyRing sequential circuit implementing the Tribonacci sequence

4.1 The Tribonacci Circuit

The Tribonacci circuit presented in this section is an implementation of the Tribonacci algorithm [44] using the KeyRing microarchitecture. Although the Keyring microarchitecture is detailed in the following sections, we chose to first present the Tribonacci circuit as it will help support generic considerations with a practical example. In addition, this section only presents the general microarchitecture of the circuit, without going into the details of its implementation. This circuit was designed based on the experience gathered with the Mini-Mips, and attempts to answer the question:

How can we design a generic sequential circuit using the Octasic design style [35] ?

Using the general ideas behind the Octasic asynchronous design style, as understood from the Mini-Mips experiment, we propose to design the equivalent of a generic three-stage pipelined state machine that implements the Tribonacci algorithm. With the Tribonacci experiment, our broader intent is to develop the KeyRing microarchitecture, and to propose new circuits and design method to integrate KeyRing circuits with a standard timing-driven EDA flow. The resulting design style will then be used to design KeyRing processors in Chapter 5.

The Tribonacci sequence [44] is similar to the famous Fibonacci sequence, but it sums the three previous elements of the sequence instead of summing only the two previous elements, as shown in Relation (4.1). The reason why we preferred the Tribonacci sequence over the Fibonacci sequence is because a sequential system that adds three elements in sequence is naturally composed of three stages, which results in a more generic circuit than a two-stage system.

$$F_n = \begin{cases} 0 \text{ when } n = 0 \\ 1 \text{ when } n = 1 \\ 1 \text{ when } n = 2 \\ F_{n-3} + F_{n-2} + F_{n-1} \text{ when } n > 2 \end{cases} \tag{4.1}$$

Figure 4.1 shows the microarchitecture of the Tribonacci KeyRing circuit, which is composed of 3 EUs comprising 3 stages. We use the same label notation as with the Mini-Mips, where the label (e, s) refer to stage s of EU e. The top half of the figure represents the datapath, where $R_{e,s}$ refer to the registers (flip-flops) of stage (e, s). For example, $R_{0,1}$ represents the registers of the second stage in the first EU. The two blue circles represent combinational resources, whose accesses are shared between EUs. The "+" resource is an adder, which is used to compute the next Tribonacci element, and the "<" resource is a comparator, which is used to end the calculation after a user-defined value has been reached. The muxes are equivalent to the crossbar-switch in the Mini-Mips. They allow to sequentially select which EU stages drive the shared resources, based on one-hot encoded selection signals produced by the KeyRing circuit. For example, with this Keyring organization, the muxes would concurrently select R_{20} at the first stage, $R_{1,1}$ at the second stage, and $R_{0,2}$ at the third stage. The Tribonacci sequence is computed by adding data saved in the first stage, coming from the second and third stages (F_{n-2} and F_{n-3} elements of the sequence), and data fed back from the second stage, resulting from the last computation (F_{n-1} element of the sequence).

The bottom half of the figure represents the KeyRing—the circuit responsible for generating the self-timed clocks—, where $K_{e,s}$ refers to the Key Unit associated with stage (e, s). Each KU generates the local clock that triggers its associated registers, as well as the selection signals used in the datapath muxes that sequentially select EU stages. As shown in Figure 4.1, KUs in the KeyRing are linked to one another using a *shifted-by-one* stage organization. That is, a given Key Unit $K_{e,s}$ waits for the completion of the previous stage in the same EU ($K_{e,s-1}$), and the completion of the same stage in the previous EU ($K_{e-1,s}$), before firing its clock. The KeyRing organization is formalized in Section 4.3.

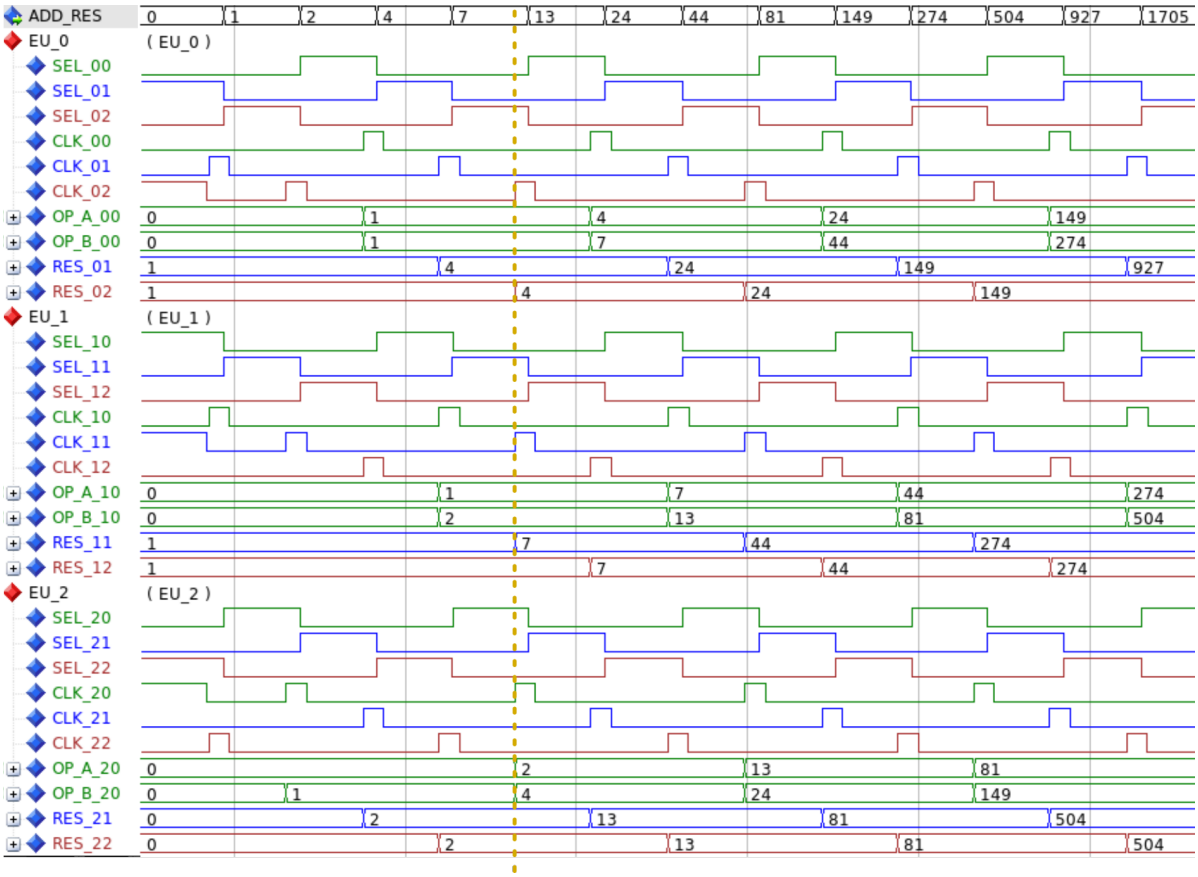
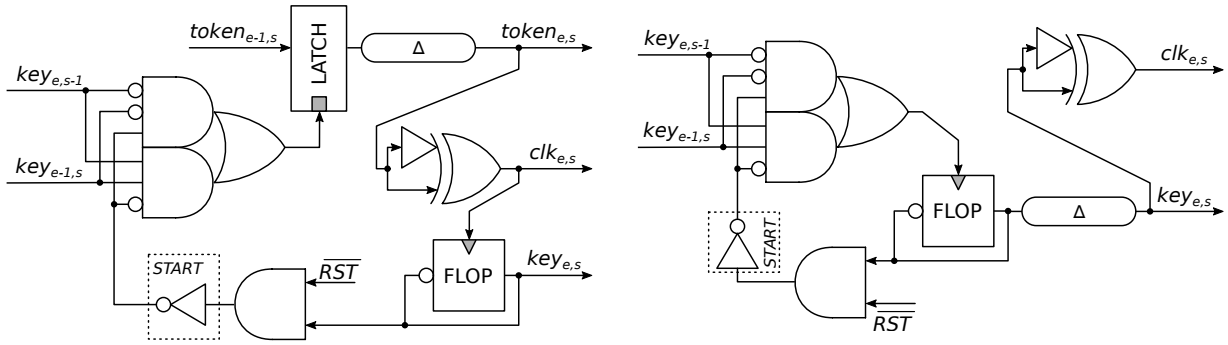


Figure 4.2 Execution flow of the Tribonacci KeyRing circuit

Figure 4.2 is a simulation snapshot showing the execution flow of the Tribonacci KeyRing circuit. It showcases a typical, yet simple, example of Instruction Level Parallelism implementation using the KeyRing microarchitecture. Signals are grouped by EUs, and each stage is associated with a color: green, blue, and red, respectively for the first, second, and third stages of each EU. The `SEL_es` and `CLK_es` signals are the selection and clock signals produced by the KeyRing, the `OP_A/B_e0` signals are the operand from the $R_{e,0}$ registers, the `RES_e1` signals are the result of the add operation saved in the $R_{e,1}$ registers, and the `RES_e2` signals are those saved in the $R_{e,2}$ registers. Each EU processes one value of the sequence every three iterations. For instance, `EU_0` processes the values $1, 4, 24$, while `EU_1` processes the values $1, 7, 44$, and `EU_2` processes the values $0, 2, 13$. EUs process values in sequence. For example, when looking at the registers of the second stage (`RES_e1`), we can see that the Tribonacci values $F_4 = 4$, $F_5 = 7$, and $F_6 = 13$, are respectively processed by `EU_0`, `EU_1`, and `EU_2`. The yellow dashed cursor in Figure 4.2 highlights an example of ILP in the circuit, where clocks `CLK_02`, `CLK_11`, and `CLK_20` fires concurrently, triggering the $R_{0,2}$, $R_{1,1}$, and $R_{2,0}$ stage registers respectively.



(a) First version (ku_1) (b) Second version (ku_2)

Figure 4.3 Key Unit design iterations

4.2 Key Units

Before we dive into the formal definitions of in-order KeyRing circuits in Section 4.3, we shall first broach the subject of the Key Unit design. Indeed, as discussed in the previous chapter, the Token Unit used in the Mini-Mips—derived from the AnARM and the Opus2 designs, as reported in [30, 31]—can be improved. Figure 4.3 shows the first two KU design iterations that attempt to improve the TU, and Figure 4.4 shows the final version used in the Tribonacci circuit and in the KeyV processors. In this section, we discuss the design trade-offs of Key Units, which involve multiple considerations, including design efficiency and robustness, power consumption, and compatibility with timing-driven EDA flows. This discussion is supported by the design and the synthesis of three KeyRing circuits used in the Tribonacci system—*i.e.* as represented on the bottom half of Figure 4.1—, each of which using a different version of the KU. The three KeyRing have been synthesized independently from the rest of the Tribonacci circuit using the GPDK45 ASIC design kit. Post-synthesis power consumption results have been evaluated—with SAIF activity files extracted from timing simulations—using the *Genus* synthesis tool, and are summarized in Figure 4.5.

The first iteration of the KU design (ku_1—Figure 4.3a) differs from the original TU (see Figure 3.8) in two respects. First, the Delay Element is placed back on the *token* line, as in the original Octasic TU. This choice is motivated by the fact that this configuration consumes less power than the configuration where the DE is placed on the *clock* line, as the *clock* toggle twice as much as the *token*. Second, and most importantly, the two toggle flip-flops generating the *valid* signals at the input of the TU have been replaced by one toggle flip-flop at the output of the KU. As the clock issued from each TU within the KeyRing is linked to two TUs, it triggers two toggle flip-flops activating two *valid* signals. Thus, these

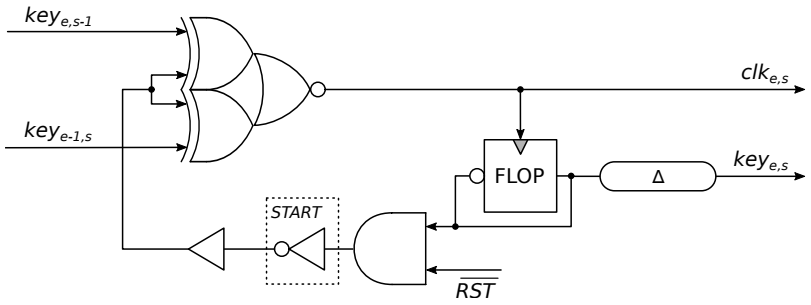


Figure 4.4 Key Unit final version (`KU_3`)

two *valid* signals carry redundant information. The KU design removes this redundancy in the KeyRing: when a clock is issued from a source KU, instead of making two signals toggle in the two destination KUs, as it is the case with TUs, only one signal toggles in the source KU and is distributed to the two destination KUs. We call this signal a *key* to contrast with *token*, used in the Octasic terminology, and with *request* and *acknowledgement*, used in the BD asynchronous terminology. Note that this naming convention has given its name to the KeyRing microarchitecture.

The second iteration of the KU design (`KU_2`—Figure 4.3b) is a simplification of the first version of the KU which results in an important conceptual change. Indeed, we have removed the *token* signal and its latch altogether. This design decision was motivated by the observation that, in the KeyRing based on the `KU_1` architecture, the *token* and the *key* signals carry redundant information (the completion of a stage). In practice, a *key* would toggle whenever its associated *token* passes through the latch and the DE. Since the *token* is inverted before going through the same KU, *key* and *token* signals in each KU of the KeyRing have the same, or mirrored (depending on their respective initialization values), waveforms. Hence, in the `KU_2` architecture, the output of the majority gate directly triggers the toggle flip-flop instead of opening the latch, and the DE and the pulse generator are placed on the *key* line instead of the *token* line. In addition to simplifying the KU itself, this architecture also simplifies the KeyRing architecture as it limits the link between KUs to the *key* signals only. As expected, Figure 4.5 shows that the `KU_2` architecture consumes less power than the `KU_1` architecture, with gains in both switching power (less signals toggle as a result of removing the token) and internal power (less on-current is drawn from the circuit as a result of removing the latch). Also note that the switching activity of the DE is the same when the DE is on the *key* line than when it is on the *token* line, as both signals have the same switching frequencies.

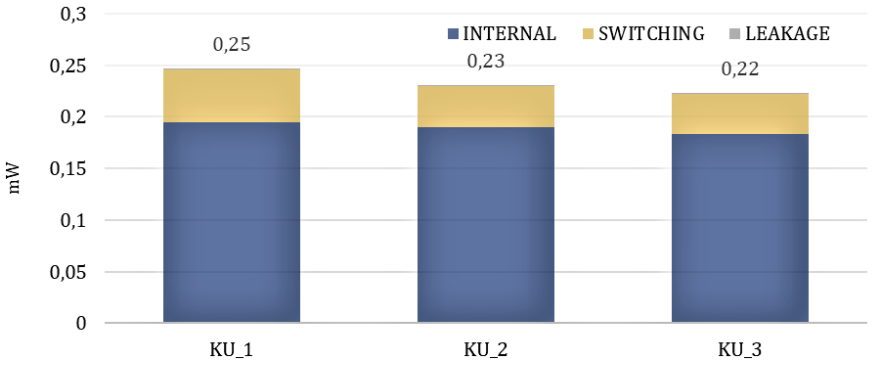


Figure 4.5 Comparison of the KeyRing power consumption when implemented using the three Key Unit architectures (results evaluated post-synthesis with the GPDK45 ASIC design kit).

Finally, Figure 4.4 shows the final version of the KU design (`KU_3`), which we use in the KeyV processors. This architecture differs from the `KU_2` design in two respects: First, the majority gate architecture has been modified. It now uses two 2-input XOR gates and one NOR gate instead of the two 3-input AND gates (and associated inverters) and one OR gate. This new architecture does not change the functionality of the majority gate—a $0 \rightarrow 1$ transition occurs when the two *key* signals coming from other KUs are in the same state, and a $1 \rightarrow 0$ transition occurs when the *key* on the feedback path toggles its state—but it reduces the number of gates, and simplifies the wiring, which results in a more compact circuit. Next, the local *clock* signal is directly taken from the output of the majority gate, instead of being generated by a dedicated pulse generator. Notice in Figure 4.3 that the buffer used on the pulse generator to adjust the width of the clock is now on the feedback path of the *key*. In addition to reducing the size of the circuit, this modification also simplifies the timing as only one clock is used to trigger both the internal state of the KU and the registers of the datapaths. Moreover, this modification involves another conceptual change: the clock pulse is now generated before the DE, and not after, as it was the case with the previous architectures. Consequently, the DE of a given KU should not match the delay of its associated EU stage, but instead should match the worst-case delay of the two EU stages associated with the destination KUs. Again, as expected, Figure 4.5 shows that the `KU_3` architecture consumes less power than the `KU_2` architecture. The gains are due to the new majority gate architecture—which uses less gates and has a lower number of inputs resulting in reduced switching and internal power—and to the removal of the pulse generator. However, the power consumption is not the only criterion that have weighted in favor of the `KU_3` architecture. Indeed, with a simpler design, we find this architecture to be also more robust and more elegant.

In summary, the Key Unit used in KeyRing circuits operates as follows (see Figure 4.4): A

toggle flip-flop stores a state signal ($key_{e,s}$), and a majority gate—composed of a pair of XOR gates and a NOR gate—implements a phase conversion mechanism that generates the local clock ($clk_{e,s}$). The rising edge of the clock toggles the state of the *key*, which, through the feedback path, initiates the falling edge of the clock. A new cycle can begin when the two input *keys*—asynchronously generated by other KUs—and the local *key* have toggled and are in the same state. The following conditions need to be satisfied for the correct operation of the circuit: *i)* the inputs must remain stable during the active phase of the clock. This is ensured by construction of the KeyRing protocol (see Section 4.3); and *ii)* the clock pulse width must be larger than the minimum pulse width defined in the technology library, which is enforced by sizing the buffer on the feedback path. Compared with the original TU, using only *key* signals as input to the majority gate, instead of a mix of *token* and *valid* signals, is less prone to errors. Similarly, having only one state holding element instead of three (or two in the `KU_1` architecture) is more robust. Moreover, in the process of elaborating the Mini-Mips implementation flow, we have found that the latch in the TU limits the definition of timing constraints. To put it simply, the STA engine cannot easily propagate timing events through latches, as the propagation of such events depends on both the clock and the data inputs (as opposed to flip-flops in which an event on the output is solely triggered by the clock input). Thus, this KU architecture—and it is confirmed in the following—is also more compatible with standard timing-driven EDA flows. Finally, note that the inverter on the feedback path of the KU is present only for specific stages in the KeyRing, as defined in Section 4.3.1. They allow the KeyRing to start after the reset sequence.

Interestingly, the KU architecture is very similar to the Click Elements template [58] (see Figure 2.8). The main difference comes from the design of the majority gate, which is similar to the Click implementation [54], and accounts for the specificities of the KeyRing protocol. As mentioned in Chapter 2, this circuit structure was also independently proposed by Intel [64], and by Quinton [65]. In addition to bridging the gap between the KeyRing microarchitecture and standard elastic asynchronous circuits, these similarities allow us to leverage the work being done around the Click Elements—asynchronous template regarded as being the most compatible with timing-driven EDA flows—, with respect to the implementation of asynchronous circuits using standard CAD tools. In particular, we propose in the following a timing-driven implementation flow that is inspired from the work of Gimenez with the Local Clock Set methodology [83,86], in which Relative Timing Constraints definitions of BD asynchronous circuits—including those implemented with the Click Elements template—are used as the basis of a synthesis flow that fully support timing-driven optimizations (see Section 2.5.3).

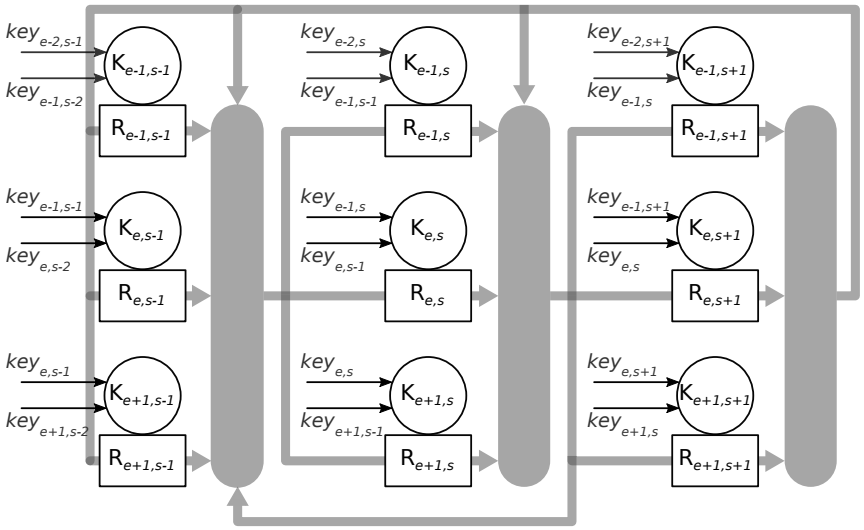


Figure 4.6 A KeyRing circuit is a two-dimensional grid of Key Units ($K_{e,s}$), linked together by keys ($key_{e,s}$), controlling a set of registers ($R_{e,s}$).

4.3 In-Order KeyRing Systems

As for the AnARM and the Mini-Mips, KeyRing systems do not use the channel abstraction of elastic circuits. Instead, KeyRing circuits use an organization of KUs linked together by *keys* in rings—that we simply call the KeyRing—, which orchestrates the generation of self-timed clocks across EU stages. The *keys* arbitrate the access to shared resources and their states bound the activity of EUs stages. An EU stage is activated by its KU when dependent *keys* (coming from other KUs) have triggered the local clock. The ordering of KUs in the KeyRing is tightly coupled with the level of parallelism exposed by the system. Yet, the link between the ILP of a KeyRing circuit and the organization of KUs is not clearly established. In this section, we propose a model of KeyRing systems—limited to the in-order mode of operation—which allows to construct any sequential in-order KeyRing circuit having a predetermined level of parallelism. This model is used to predict the performances of KeyRing circuits, and is leveraged in Section 4.4 to support our proposed timing-driven EDA flow.

Although it was already presented, the Tribonacci circuit detailed in Section 4.1 is a direct implementation of the concepts developed in this section. In the following, we use an abstracted model of in-order KeyRing circuits (see Figure 4.6) from which the Tribonacci circuit was inspired. In addition, in the following the Key Unit refers to the `ku_3` architecture, as represented on Figure 4.4.

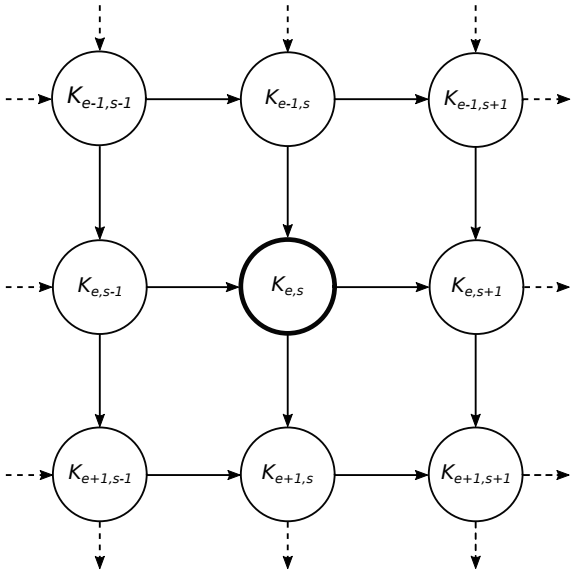


Figure 4.7 The KeyRing protocol is modelled by a graph having a toroidal mesh topology with wrap-around connections at the edges.

4.3.1 General principles

In this section, we show that the KeyRing microarchitecture is suitable for implementing any sequential system that would otherwise be implemented with a pipeline. Figure 4.6 represents a generic KeyRing circuit equivalent to a 3-stage pipeline, and Figure 4.7 is an abstracted view of the KUs organization. It is composed of 3 EUs (columns) of 3 stages each (rows), where $R_{e,s}$ represent the registers of the datapath, and $K_{e,s}$ their associated Key Units. The KeyRing protocol is modeled by the graph of Figure 4.7, where nodes represent KUs and arcs represent *keys*. Interestingly, this interconnection topology of processing elements has previously been studied in the context of multicomputer networks, as reported in [105]. Here we reuse some of the notations originally proposed in this publication.

Let $G = (K, L)$ be the graph of Figure 4.7, such that $K \in E \times S$ and $L \in E^2 \times S^2$, where E is the number of Execution Units and S is the number of stages per EU. G is a *toroidal mesh*—that is, a graph in which the nodes can be embedded on a torus such that no edges are crossing—which consists of a two-dimensional grid of EUs with wrap-around connections at the edges. For any node (e, s), $e \in \{0, ..., E-1\}$, $s \in \{0, ..., S-1\}$, connections are defined as follow:

$$\begin{aligned}(e, s) &\leftarrow (e, \langle s-1 \rangle_S), (\langle e-1 \rangle_E, \langle s+(\alpha-1) \rangle_S) \\ (e, s) &\rightarrow (e, \langle s+1 \rangle_S), (\langle e+1 \rangle_E, \langle s-(\alpha-1) \rangle_S)\end{aligned} \tag{4.2}$$

where $\langle x \rangle_X$ signifies that x is considered *modulo* X, which allows to account for the wrap-around connections at the edges, and where $\alpha \in \{1, ..., S\}$ is the parameter representing the dependency shift between two stages of successive EUs. Hence, a *shifted-by-one* KeyRing organization (case of the Tribonacci circuit) corresponds to $\alpha = 1$, while a *shifted-by-two* KeyRing organization (case of the Mini-Mips) corresponds to $\alpha = 2$. Note that Figure 4.6 and Figure 4.7 show the specific case where $\alpha = 1$.

Definition 1. An *in-order* KeyRing system is such that no stage is computed concurrently in more than one EU.

Relation (4.2) describes KeyRing organizations in which computations are performed *in-order*, as defined in Definition 1. A value of $\alpha = 1$ corresponds to the case where a stage in a given EU (e, s) can start when the stage in the previous EU $(e - 1, s)$ has finished, which maximizes parallelism. On the opposite end, a value of $\alpha = S$ corresponds to the case where the first stage of a given EU $(e, 0)$ can start when the last stage of the previous EU $(e-1, S-1)$ has finished, which minimizes parallelism. Indeed, the level of parallelism of an in-order KeyRing system depends on the number of EUs (E), the number of stages per EU (S), and on the relation between dependent stages across successive EUs (α). Let $F(e, s)$ be the relation between these parameters, such that all the stages (e, s) having the same value of $F(e, s)$ are computed concurrently.

$$F(e, s) = \langle s + \alpha e \rangle_S \tag{4.3}$$

Definition 2. The *level of parallelism* P of an in-order KeyRing system is the number of elements in the set of all (e, s), $e \in \{0, ..., E - 1\}$, $s \in \{0, ..., S - 1\}$, having the same value of $F(e, s)$. It is equal to the number of EUs: $P = E$.

From these definitions, we propose a necessary condition on the KeyRing parameters (E, S and α) that a KeyRing system should satisfy to operate in-order.

Proposition 1. *An in-order KeyRing system satisfies* $\alpha E = \lambda S$*, with* $\lambda \in \{1, ..., \alpha\}$.

Proof. By definition of an in-order KeyRing system (Definition 1), a given stage s has the same value of $F(e, s)$ in only one EU e. From eq.(4.3), by taking advantage of the symmetry of the graph, this can be expressed for the first stage as: $F(E, 0) = F(0, 0) \Rightarrow \langle \alpha E \rangle_S = 0 \Rightarrow \alpha E = \lambda S, \lambda \in \mathbb{N}^+$. Moreover, KeyRing organizations where $\lambda > \alpha$ have more EUs than stages per EU, which is incompatible with Definition 1. Thus $\lambda \leqslant \alpha$. □

KeyRing systems having more stages per EU than EUs ($S > E$) are compatible with Definition 1. However, these configurations are suboptimal. For example, a KeyRing with ($E = 3, S = 6, \alpha = 2$), as used in the Mini-Mips (see Section 3.4), has the same level of parallelism as a KeyRing with ($E = 3, S = 3, \alpha = 1$) despite having twice as many stages per EU. The *optimal* level of parallelism of an in-order KeyRing system is a configuration in which the level of parallelism is maximized while the number of EUs and stages per EU are minimized. It is obtained when $E = S$. As we will see in the following, although KeyRing circuits having $E < S$ are suboptimal in terms of parallelism, and thus reach poor performances, they need less timing constraints to be correctly implemented, which results in reduced synthesis time and more robust circuits. The Tribonacci circuit presented in Section 4.1 uses a KeyRing with ($E = 3, S = 3, \alpha = 1$). In Chapter 5, we present two *KeyV* processors, one with a ($E = 3, S = 6, \alpha = 2$) KeyRing, and the other with a ($E = 6, S = 6, \alpha = 1$) KeyRing. They are both compared with a synchronous processor (*SynV*) having a 6 stages pipeline.

Finally, in addition to the organization of KUs in the KeyRing, which define the level of parallelism in the circuit, we must also consider the practical aspects of operating the KeyRing. In particular, starting the KeyRing after a reset sequence can be quite challenging. This issue is addressed by answering the following question:

What initial values should the KUs have to start the KeyRing after reset ?

Indeed, if all the flip-flops are initialized with the same value—either 0 or 1—, then all *keys* will have the same value when the reset is released, and no clock will be issued. Thus, some KUs in the KeyRing should be initialized with a different value, but which ones ? We first addressed this issue using a trial-and-errors approach. In this process, we have found that most configurations do not permit the KeyRing to start, either because too few pulses are issued, or because the wrong pulses are issued concurrently. In both cases, the wave of clocks quickly vanishes and the circuit rests idle. The solution is to only invert the initial value of one set of concurrent KUs. Using Relation (4.3), this can be expressed as follows:

$$K_{e,s} \xleftarrow{\text{reset}} \begin{cases} not(\text{INIT}) \text{ if } F(e,s) = \gamma \\ \text{INIT otherwise} \end{cases} \tag{4.4}$$

where $\gamma \in \{0, S-1\}$, and INIT is the (boolean) initial value. When the reset is released, the *keys* from the starting KUs will generate clocks in subsequent stages that will trigger other KUs until the whole KeyRing has started. The inverter on the feedback path of the KU architecture is only present on the starting KUs to account for the inverted logic during normal operation.

4.3.2 Predicting Performance

Performance of a sequential circuit is bounded by the level of parallelism and the rate of the computations. In the previous section, we have elaborated a model that defines the level of parallelism as a function of the KeyRing parameters. In this section, we evaluate the pace at which the KeyRing can be operated. In a typical synchronous system, the speed is determined by a global clock—usually generated externally—characterized by a given frequency and by uncertainties coming from the clock distribution network. In a KeyRing system, by contrast, the speed is determined by local clocks—generated and timed internally—characterized by their individual frequencies, and by uncertainties coming from their respective clock trees. Moreover, in a synchronous system, the clock base frequency is independent from the implementation flow; only the uncertainties of the clock tree are affected by the implementation. In contrast, because the clocks in a KeyRing system are timed by DEs, their frequencies are constantly being altered during the implementation flow.

Definition 3. The period $T(e, s)$ of a clock in a KeyRing system is the delay separating the occurrence of two pulses generated by a given KU $K_{e,s}$.

As opposed to synchronous systems, in which the clock period corresponds to the delay between adjacent stages (adjusted with the clock skew), the periods of the self-timed clocks in a KeyRing system should not be confused with the delay between stages due to the multicycle nature of Execution Units. Hence, this metric is independently defined:

Definition 4. The delay Δ_{src}^{dest} between a source stage (src) and a destination stage ($dest$) in a KeyRing system is the delay separating the occurrence of the destination clock, generated by KU K_{dest}, from the occurrence of the source clock, generated by KU K_{src}.

In synchronous pipelines, the delay between two stages that are separated by multiple stages is simply the sum of each stage latency expressed as a multiple of the clock period (assuming no stalls). In a KeyRing system, by contrast, the delay between two stages cannot simply be expressed as the sum of each stage latency because of the interdependence of KUs, as modeled by the KeyRing graph (Figure 4.7). Let δ_{src}^{dest} be the latency between two *adjacent* nodes—src and $dest$—in the KeyRing graph, which can be evaluated with STA. It includes the delay of the DE, and the delays of the clock distribution network (contributing to the clock skew), including the delay from the source KU to the DE, and from the DE to the destination KU. As illustrated by the KeyRing graph, the instant at which a KU fires its clock recursively depends on the arrival time of its predecessors *keys*. To determine the *effective delay* between two clocking events in the KeyRing—*i.e.* the delay between two stages in the KeyRing as per Definition 4—this recursivity should be accounted for.

Algorithm 3: Determine the *effective* delay between a source (K_{src}) and a destination (K_{dest}) Key Unit.

Data: Let G be the graph of Figure 4.7, $A[K_{e,s}]$ the forward adjacency lists of a node $K_{e,s} \in G$, μ and ν be tables, and Γ be a stack

```
1  Function EFFECTIVE_DELAY(K_src, K_dest) is
2      Initialize(Γ, μ, ν)
3      push(Γ, K_src)
4      while Γ ≠ ∅ do
5          K_{e,s} ← pop(Γ)
6          ν(K_{e,s}) ← μ(K_{e,s})
7          foreach K_{i,j} ∈ A[K_{e,s}] do
8              μ(K_{i,j}) ←max(μ(K_{i,j}), μ(K_{e,s}) + δ^{i,j}_{e,s})
9              if K_{i,j} has not been reached yet then
10                 push(Γ, K_{i,j})
11             end
12         end
13     end
14     return μ(K_src) − ν(K_dest)
15 end
```

To this end, we propose a *longest-path algorithm* (Algorithm 3) which computes the effective delay between two clocking events by finding the longest path between two nodes of the KeyRing graph (Figure 4.7). It is inspired from the Breadth First Search (BFS) algorithm and the Dijkstra algorithm [106], which are well known methods for solving similar problems. Each arc of the graph is weighted by δ_a^b, which have been defined as the latency between two adjacent nodes a and b. Starting from the source node, the algorithm traverses the graph by exploring the forward adjacency list of each node. Each step of the way, it computes the maximum cumulative distance from the source. When applied to a directed graph, the BFS algorithm traverses each arc exactly once [106]. Given the symmetry of the graph, this means that the cumulative distance is evaluated twice for each node, the retained value being the largest.

Algorithm 3 can be used to determine the delay Δ_{src}^{dest} between two stages in a KeyRing system (4.5), by computing the effective delay between a source (src) and a destination ($dest$) KU. Similarly it can determine the period $T(e, s)$ of a clock (4.6), by computing the effective delay between two clocking events of the same KUs.

$$\Delta_{src}^{dest} = \text{EFFECTIVE\_DELAY}\left(K_{src}, K_{dest}\right) \tag{4.5}$$

$$T(e, s) = \text{EFFECTIVE\_DELAY}\left(K_{e,s}, K_{e,s}\right) \tag{4.6}$$

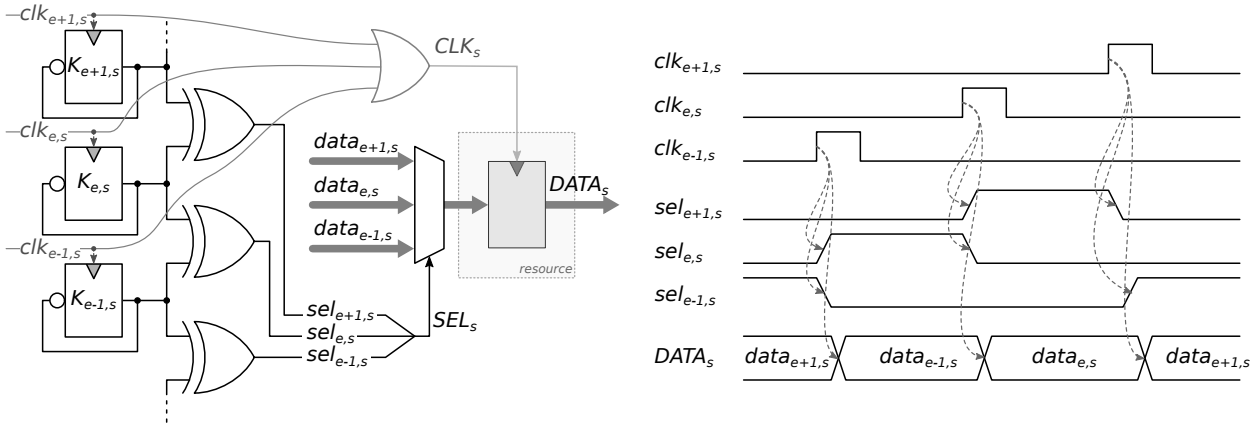
Proposition 2. *All the clock periods in a KeyRing system have the same value T, for a given set of Delay Elements configuration:* $\forall(e,s), T(e,s) = T$.

Proof. From Relation (4.6), $T(e,s)$ is computed with Algorithm 3 by traversing the graph from node (e,s) back to itself. The algorithm reaches each node twice, choosing the longest path at each step. Thus, regardless of the starting point, the computation of $T(e,s)$ traverses the longest cycle of the graph. □

In practice we have indeed observed that the evaluation of every clock period using Algorithm 3 and STA results, as well as in post-synthesis timing simulations, returns the same value. It is important to note that this is true regardless of the DEs configuration; even if DEs have very different delays, the period of every clock in the KeyRing is the same. Intuitively, this is explained by the graph topology. Note that clocks period do vary when DEs configuration are dynamically adjusted. In this case, the system first passes through a transient state during which the clock periods are temporarily not all equals, before reaching a new steady state in which all the clock periods have the same, new, value.

4.3.3 Arbitrating the access to shared resources

We have seen that the exploitation of parallelism in KeyRing systems greatly differs from classical pipelining as it relies on the concurrent use of multicycle EUs. In particular, it differs with respect to the way resources are accessed (*e.g.* the adder or the comparator in the Tribonacci circuit (Figure 4.1)). In an in-order synchronous pipeline, as long as a resource is used in only one stage s, its access needs not be arbitrated: the resource access is implicitly always assigned to this stage. In an in-order KeyRing system by contrast, multiple stages—*i.e.* $(e-1,s), (e,s), (e+1,s)$—compete to access the same resource. The arbitration of resources access between EUs stages is performed by the combination of *i)* the ordering of clocks generation in the KeyRing, allowing to trigger the resource in sequence; and *ii)* the arbitration circuit of Figure 4.8, allowing to dynamically select which EU stage should drive the resource. Data coming from each of the competing stages are connected to a MUX driving the input of the resource. This MUX is controlled by the SEL_s signal composed of selection signals formed by XORing the *keys* of neighboring EUs, as represented in Figure 4.8a. Thus, the SEL_s signal is *one-hot* due to the circuit topology, as illustrated in Figure 4.8b. The resource may have registered outputs, in which case the self-timed clocks of each competing stages are ORed together to trigger the resource register. The resource may also be combinational, such as in the Tribonacci circuit, in which case its output is captured by the register of an EU stage using its dedicated clock.



(a) circuit (b) waveform

Figure 4.8 Arbitration of resources access between Execution Units stages

The crossbars in the Mini-Mips and KeyV processors (see Section 5.4) are mostly composed of such one-hot MUXes, working in coordination with the KeyRing to dynamically guide the correct data from EUs to shared resources. The need for arbitration is a clear overhead of in-order KeyRing systems compared with synchronous pipelines. However, it has the potential to scale well with out-of-order operations. As out-of-order operation in synchronous systems significantly increases their complexity, this overhead may be reduced in that case.

4.4 Static Timing Analysis

In this section, we develop a set of timing constraints for KeyRing systems that allow to leverage Static Timing Analysis and timing-driven synthesis using standard EDA tools. The proposed timing constraints are built in two steps: first, formal definitions of the constraints using Relative Timing Constraints are elaborated in Section 4.4.1; then, a practical implementation of these constraints using the SDC format is proposed in Section 4.4.2. In the following, we only consider KeyRing systems having fixed DEs configurations. Indeed, although the perspective of dynamically adjusting the size of DEs as a function of the operation being processed is appealing in terms of performance and power consumption, such dynamic behavior is incompatible with the current state of the design methodology. As we have seen in the previous section, because each stage delay has an impact on the whole system, the timing constraints can only be met for a given set of DEs configuration. Synthesizing a KeyRing system that simultaneously meet the constraints of multiple DEs configurations exceeds the scope of this work.

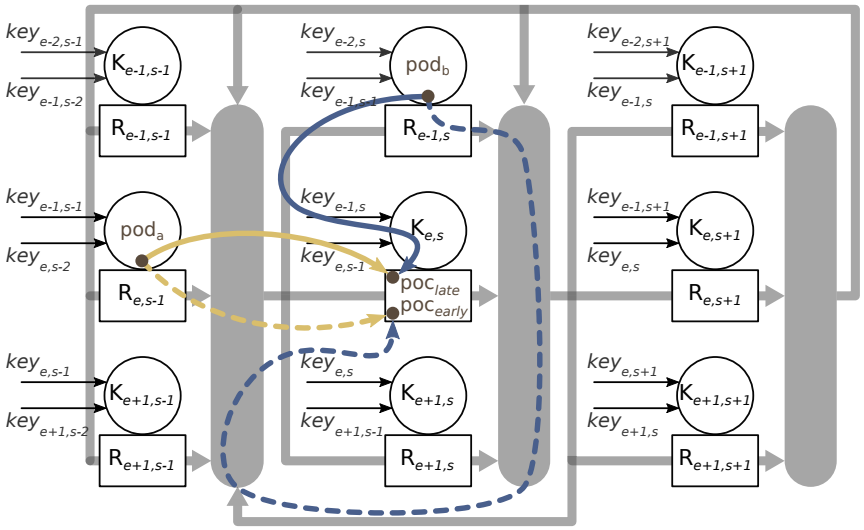


Figure 4.9 Protocol-level *setup* RTCs definitions in a KeyRing circuit

4.4.1 Relative Timing Constraints of KeyRing systems

Relative Timing Constraint is a formalism developed by Stevens [87, 88, 92] which allows to define the timing requirements that a circuit must satisfy to operate correctly. The basic principles of the RTC formalism have been presented in Section 2.5.2. In summary, a RTC is of the form: $pod \mapsto poc_{early} \prec poc_{late} - \varepsilon$; which specifies that the consequences of an event occurring at the *point-of-divergence* (*pod*) must reach the *point-of-early-convergence* (poc_{early}) before reaching the *point-of-late-convergence* (poc_{late}) with a margin ε. In this section, we use RTCs to define the timing requirements of a generic KeyRing system.

Figure 4.9 and Figure 4.10 depict the generic KeyRing circuit with an overlay representing the protocol level RTCs definitions spanning over multiple EU stages. These definitions are shown for a KeyRing having $E = 3, S = 3, \alpha = 1$, which can be used as a template to be adapted with any type of KeyRing circuit. The point-of-divergence serves as reference for the definitions of RTCs. In a synchronous circuit, as relations (2.7) and (2.8) suggest, it is usually defined as the origin of the main clock. Thus, it can be defined implicitly as it is the same for every RTC. In a KeyRing circuit, or in an asynchronous BD circuit, by contrast, it varies with each stage, thus it must be explicitly defined. In KeyRing circuits, finding the *pods* is further complicated by the recursive dependencies of KUs. In practice, a *pod* can be localized as the last common point between the launch and the capture paths. That is, a KU at the intersection of a launch path—from dependent KUs to the data input of the destination registers—and a capture path—from dependent KUs to the clock input of of the destination registers—is the *pod* of a RTC.

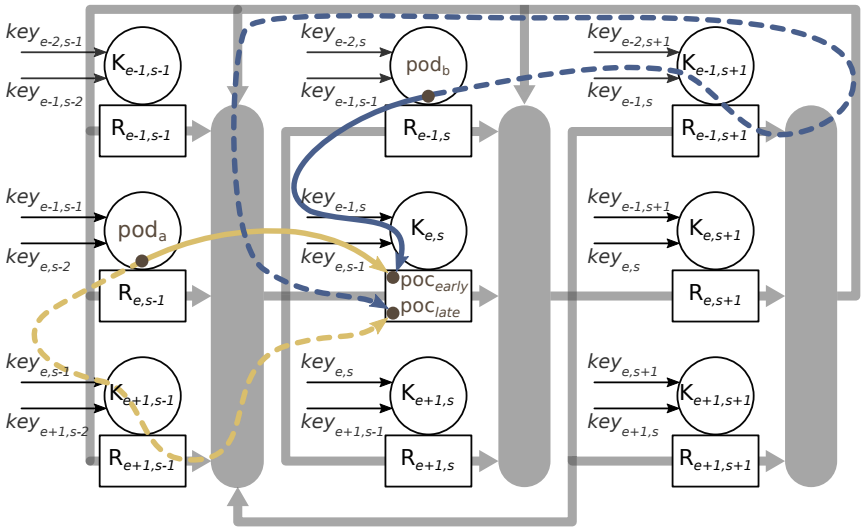


Figure 4.10 Protocol-level *hold* RTCs definitions in a KeyRing circuit

A specificity of KeyRing circuits is that each stage (e, s) has two setup and two hold RTCs. The first RTC comes from the dependent stage $(e, s-1)$ —pod_a in Figure 4.9 and 4.10—, while the second RTC comes from the dependent stage $(e-1, s)$ —pod_b in Figure 4.9 and 4.10. In the following, we denote $C_{e,s}$, and $D_{e,s}$ the clock pin and the data pin (respectively) of register $R_{e,s}$. Moreover, note that Figure 4.9 and Figure 4.10 represent launch paths with dashed lines, and capture paths with plain lines.

$$\text{setup (a): } C_{e,s-1} \mapsto D^{max}_{e,s} \prec C_{e,s} - \varepsilon \tag{4.7}$$

$$\text{setup (b): } C_{e-1,s} \mapsto D^{max}_{e,s} \prec C_{e,s} - \varepsilon \tag{4.8}$$

$$\text{hold (a): } C_{e,s-1} \mapsto C_{e,s} \prec \{C_{e+1,s-1}, D^{min}_{e,s}\} - \varepsilon \tag{4.9}$$

$$\text{hold (b): } C_{e-1,s} \mapsto C_{e,s} \prec \{C_{e-1,s+1}, D^{min}_{e,s}\} - \varepsilon \tag{4.10}$$

Setup RTCs are defined by relations (4.7) and (4.8), and are represented in Figure 4.9. The capture path starts at $C_{e,s-1}$ (*resp.* $C_{e-1,s}$) and reaches $C_{e,s}$ from the DE of $K_{e,s-1}$ (*resp.* $K_{e-1,s}$) through the shortest path. The launch path starts at $C_{e,s-1}$ (*resp.* $C_{e-1,s}$) and reaches $R_{e,s}$ from $R_{e,s-1}$ (*resp.* $R_{e-1,s}$) and the combinational logic through the longest path. Hold RTCs are defined by relations (4.9) and (4.10), and are represented in Figure 4.10. The capture path starts at $C_{e,s-1}$ (*resp.* $C_{e-1,s}$) and reaches $C_{e,s}$ from the DE of $K_{e,s-1}$ (*resp.* $K_{e-1,s}$) through the longest path. The launch path starts at $C_{e,s-1}$ (*resp.* $C_{e-1,s}$), then reaches $C_{e+1,s-1}$ (*resp.* $C_{e-,s+1}$) from the DE of $K_{e,s-1}$ (*resp.* $K_{e-1,s}$), and reaches $R_{e,s}$ from $R_{e+1,s-1}$ (*resp.* $R_{e-1,s+1}$) and the combinational logic through the shortest path.

Setup RTCs in Keyring circuits are similar to those in BD circuits [83, 86]. On the other hand, hold RTCs are different. Indeed, the hold conditions in a typical asynchronous BD stage is associated with the feedback path of the acknowledgement signal. The lack of acknowledgement signal path in a typical multicycle Execution Unit may mislead the designers into thinking that KeyRing systems are exempt of hold conditions altogether, as discussed in Section 2.3 and in Section 3.4. Hold conditions in KeyRing circuits are indeed hidden if the focus is on a single EU stage, but they are revealed when looking at the interaction of multiple EU stages (see Figure 4.10). Hold conditions express the necessity of capturing data in a stage register before it is overridden by new data. In a synchronous pipeline, hold violations occur when the incoming data of a register are overridden by the data produced from another register, launched by an event clk_i, before they are captured by the same event clk_i. KeyRing circuits are indeed immune to this type of violations because EUs are multicycle. In a KeyRing circuit, hold violations happen in a stage (e, s) when the incoming data of register $R_{e,s}$ are overridden by the data produced from a concurrent stage, from register $R_{e+1,s-1}$ (launched by the clock event on $C_{e+1,s-1}$), or from register $R_{e-1,s+1}$ (launched by the clock event on $C_{e-1,s+1}$), before they are captured by the clock event on $C_{e,s}$.

Last but not least, we must consider *multi-pods timing paths*. Although RTCs are by definition relative to a common point-of-divergence, multi-pods timing paths refer to the cases where data launched from pod_a (pod_b) is being captured by a clock starting from pod_b (pod_a). These cases exist for both setup and hold conditions, and should thus be constrained appropriately. The solution that comes to mind is to define additional RTCs starting from a common ancestor in the KeyRing graph, but unfortunately it is not conclusive. Indeed, multi-pods timing paths are only activated when the capture path from pod_b (pod_a) is longer than the capture path from pod_a (pod_b). This situation is actually in favor of meeting the setup condition, as delaying the arrival time of the clock on the capture path increases the likelihood of satisfying the setup constraint. Thus, no additional RTC need to be defined to constrain multi-pods setup timing paths. On the other hand, this situation works against meeting the hold condition, as the clock on the capture path must arrive early to satisfy hold constraints. But trying to define a new RTC from parent KUs only shifts the problem one stage up in the KeyRing graph. Regardless of the stage at which we start the RTC, there will always be a race between pod_a and pod_b. Hence, no static method would permit to fully constrain multi-pods hold timing paths. In practice, as long as the clock definitions used to translate both RTCs (*i.e.* Relation (4.7) and (4.8), or Relation (4.9) and (4.10)) belong to the same clock group, inter-clock timing paths are automatically constrained by the STA engine (see Section 4.4.2). The solution that we propose to safely constrain multi-pods timing paths is thus to define inter-clock hold margins.

4.4.2 Timing driven synthesis flow

In order for the RTCs defined in the previous section to serve the purpose of constraining the design in a timing-driven design flow, and to correctly report the setup and hold slacks available for each EU stage using standard STA reports, they must be translated into a set of timing constraints using the SDC format that the tools interprets appropriately. As already mentioned, this task is not trivial [82, 83, 86–88]. This section deals with it for the case of KeyRing circuits. An important contribution of the Local Clock Set methodology [83, 86] is to show that synchronous EDA tools already deal with RTCs. As discussed in Section 2.5, internal RTCs are typically defined for each component in a timing characterization file. To deal with timing, EDA tools expect that clock constraints be defined at every clock pin of the registers, in order to derive the setup and hold timing conditions of each stage from the internal RTCs defined in the timing characterization file of the registers cell. In the following, we leverage this knowledge in combination with an informed use of SDC commands to properly convert the KeyRing RTCs into a set of timing constraints.

Algorithm 4 shows the proposed set of timing constraints that makes KeyRing circuits compatible with a timing-driven design flow. Similarly to the LCS methodology, we use clock constraints as the basis of our method. As mentioned, the main advantage of using clock constraints—using the `create_clock` and the `create_generated_clock` SDC commands—is that it allows to use synchronous EDA tools as they are intended to, thus enabling full support for timing-driven optimizations. Note that Algorithm 4 targets Synopsys tools, and that the associated analysis may not be valid for another tool suite. In particular, we failed to make this method work in the same manner using Cadence tools.

Root clocks—The clocks $c_{e,s}$ and $k_{e,s}$, defined on the *clock* pin ($C_{e,s}$) and the *key* pin ($K_{e,s}$, the output of the toggle flip-flop in Figure 4.4) of each KU are used *i)* to break architectural loops that the STA engine finds by traveling along the KeyRing (here we take advantage of the fact that a clock constraint definition break any timing path crossing its source point); and *ii)* to generate the *launch* and the *capture* clocks in the design (here we use the property of generated clock constraints which are able to propagate events through breakpoints). Using this combination of clock constraints for each KU is enough to disable all the timing cycles in the KeyRing, and provides support for the rest of the constraints. Note that this reveals an advantage of the KU architecture (see Figure 4.4) over the TU architecture (see Figure 3.8), which requires a reduced number of clock constraints to disable all timing cycles.

Launch & Capture clocks—The remaining clock constraint definitions, derived from the root clocks, are meant to be used by the timing engine to effectively constrain the datapaths. Two groups of clocks are defined, respectively for setup and for hold timing checks: the *setup*

clocks translate RTCs (4.7) and (4.8), while the *hold* clocks translate RTCs (4.9) and (4.10). For each RTC, a *launch* and a *capture* clock is defined for each *poc*: the *setup-launch* clocks (noted *sl*) and the *setup-capture* clocks (noted *sc*) are respectively used for poc_{early} and for poc_{late}. Similarly, the *hold-launch* clocks (noted *hl*) and the *hold-capture* clocks (noted *hc*) are respectively used for poc_{late} and for poc_{early}. The *pods* are automatically determined by the tool as the last common point between the launch clock and the capture clock paths. That is, one of the root clock $c_{e,s}$.

Margins—Margins are added to the timing paths by applying an uncertainty surplus delay between every pair of launch and capture clocks (using the `set_clock_uncertainty` SDC command). Similar margins are used in synchronous design flow to account for the deviations of the fabricated circuit from the timing models. In addition, we add an increased margin between crossed pair of hold launch and capture clocks. This strengthen inter-clock hold constraints to safely constrain multi-pods timing paths (see Section 4.4.1).

Exceptions—Various exceptions are applied to the clocks depending on their role. First, root clocks are excluded from the timing analysis with false paths definitions (using the `set_false_path` SDC commands), as they are not intended to constrain the datapaths. Then, inter-clock timing interactions between selected groups of clocks are disabled (using the `set_clock_groups -asynchronous` SDC commands). Every group of four setup (*resp.* hold) clocks generated for each stage (e, s) are added to the $setup_{e,s}$ (*resp.* $hold_{e,s}$) group after being defined. Only the inter-clock interactions between clocks of a same group are allowed: *i)* setup clocks do not interact with hold clocks, and *ii)* clocks defined at stage (e, s) do not interact with clocks defined at other stages. Finally, the timing paths coming from (*resp.* reaching) capture clocks (*resp.* launch clocks) are defined as false paths, because capture clocks (*resp.* launch clocks) do not launch (*resp.* capture) data.

Miscellaneous—The default behavior of synchronous tools, with respect to the way they deal with clocking events, should be altered to be suitable with KeyRing circuits. First, clocks should be propagated (using the `set_propagated_clock` SDC command). Propagating clocks allows the timing engine to account for the contributions of the clock path logic delay to the skew. In our case, propagating clocks is mandatory to include the control path delay—including the DEs delay, the KUs logic delay, *etc.*—in the computation of the launch and capture clocks latencies. Second, the ordering of active clock edges that are considered for the timing analysis should be modified. Indeed, by default setup checks are performed between two successive active edges of the clock, separated by the clock period, and the hold checks are performed between the same active edges. However, for circuits with bundled data operations (such as asynchronous BD and KeyRing circuits), the same clock edge generates

the launch and the capture events. Thus, *i)* both setup and hold checks should be performed between the same active edge of the clock, and *ii)* the constraints should reflect the relative arrival time of the launch and the capture clocks regardless of their respective period. This can be achieved by defining the paths between launch and capture clocks as *zero-cycle path* (using the `set_multicycle_path 0` SDC command). Having propagated launch and capture clocks, with setup and hold timing checks performed at zero-cycle, allows to adequately use the control path (including the DE) delay as a constraint for the associated datapath. Finally, the DE lengths are defined in the timing engine by setting a fixed value to the control pins with a *case analysis* (using the `set_case_analysis` SDC command). Although the length of each DE is fixed with constants defined in the RTL, the timing engine does not propagate the value of these constants unless explicitly specified by a case analysis.

4.4.3 The KeyRing Tcl library

This section deals with the practical implementation of the KeyRing timing constraints (see Algorithm 4) using standard, Tcl based, EDA tools. In particular, we present a Tcl library that was devoloped with the goal of facilitating and generalizing the deployment of KeyRing timing constraints. Figure 4.11 is a code snippet illustrating the basic ideas behind this Tcl library. In practice, this library is defined as a *class*—using the built-in `oo::` Tcl package—that can be used in any Tcl script used in the design flow, including the timing constraints.

The library provides the `KeyRing` class, which allows to create KeyRing objects. Each object is defined by the KeyRing parameters—the number of EUs E (`E`), the number of stages per EUs S (`S`), and the dependency shift parameter α (`D`)—and a data structure, `KUS`, representing the KeyRing graph of Figure 4.7. The following example shows the creation of the `KeyRing` object used for the Tribonacci circuit, with $(E = 3, S = 3, \alpha = 1)$:

```
set KeyRingTribo [KeyRing create "tribonacci" 3 3 1]
```

The `KUS` data structure is a Tcl dictionary, which keys are the KUs of the KeyRing. For each KU, connections to neighboring KUs are defined in the `neighbor` array, according to the definitions of relation (4.2). Manipulating `KeyRing` objects is facilitated by a collection of *methods* interacting with the data structure. In the following example, each KU is displayed next to its right hand side neighbor.

```
foreach k [$KeyRingTribo get_kus] {puts {$k:[$KeyRingTribo get_neighbor $k r]}}
```

Algorithm 4: KeyRing timing constraints pseudo code

 # Root clocks
foreach (e, s) *in KeyRing* **do**
    create clock $c_{e,s}$ on $C_{e,s}$
    generate clock $k_{e,s}$ from $c_{e,s}$ on $K_{e,s}$
end
 # Setup launch & capture clocks
foreach (e, s) *in KeyRing* **do**
    generate clock $sl^{e,s}_{e,s-1}$ from $c_{e,s-1}$ on $C_{e,s-1}$
    generate clock $sc^{e,s}_{e,s-1}$ from $k_{e,s-1}$ on $C_{e,s}$
    generate clock $sl^{e,s}_{e-1,s}$ from $c_{e-1,s}$ on $C_{e-1,s}$
    generate clock $sc^{e,s}_{e-1,s}$ from $k_{e-1,s}$ on $C_{e,s}$
    add these clocks to the $setup_{e,s}$ group
end
 # Hold launch & capture clocks
foreach (e, s) *in KeyRing* **do**
    generate clock $hl^{e,s}_{e,s-1}$ from $k_{e,s-1}$ on $C_{e+1,s-1}$
    generate clock $hc^{e,s}_{e,s-1}$ from $k_{e,s-1}$ on $C_{e,s}$
    generate clock $hl^{e,s}_{e-1,s}$ from $k_{e-1,s}$ on $C_{e-1,s+1}$
    generate clock $hc^{e,s}_{e-1,s}$ from $k_{e-1,s}$ on $C_{e,s}$
    add these clocks to the $hold_{e,s}$ group
end
 # Margins
add uncertainty to all capture clocks
increase the uncertainty for inter-clock hold capture clocks
 # Exceptions
define false paths from/to root clocks
define groups of clocks as asynchronous
define false paths from capture clocks
define false paths to launch clocks
 # Miscellaneous
propagate all clocks
define zero-cycle path from launch to capture clocks
set case analysis on the DEs

```
oo::class create KeyRing {
  variable KUS E S D
}
oo::define KeyRing {
  constructor {e s d} {
    set E $e; set S $s; set D $d;
    for {set e 0} {$e < $E} {incr e} {
      for {set s 0} {$s < $S} {incr s} {
        dict set KUS K${e}${s} neighbor(l) K[expr $e%$E][expr ($s-1)%$S]
        dict set KUS K${e}${s} neighbor(r) K[expr $e%$E][expr ($s+1)%$S]
        dict set KUS K${e}${s} neighbor(u) K[expr ($e-1)%$E][expr ($s+$D-1)%$S]
        dict set KUS K${e}${s} neighbor(d) K[expr ($e+1)%$E][expr ($s-$D+1)%$S]
      }
    }
  }
  method get_name {} {
    return [string trim [self] ::]
  }
  method get_kus {} {
    return [dict keys [dict get $KUS]]
  }
  method get_neighbor {k d} {
    return [dict get $KUS $k neighbor($d)]
  }
  method create_root_clocks {} {
    ...
  }
  method create_launch_capture_clocks {} {
    ...
  }
}
```

Figure 4.11 Code snippet from the `KeyRing.tcl` package showing the KeyRing data structure

CHAPTER 5 KEY-V: ARCHITECTURAL EXPLORATION OF KEYRING RISC-V PROCESSORS WITH A TIMING-DRIVEN ASIC DESIGN FLOW

In this chapter, we present the design of RISC-V microprocessors based on the KeyRing microarchitecture—called *KeyV*—and their implementation using the timing-driven design methodology proposed in Chapter 4. Similarly to the Mini-Mips experiment (see Chapter 3), our goal is to explore the design trade-offs of the KeyRing microarchitecture, to demonstrate the effectiveness of our proposed EDA methodology, in a realistic scenario. Here, we reckon with the limitations encountered with the Mini-Mips, namely the unsuitability of the FPGA platform and the restriction of the Mini-Mips ISA. First, KeyV processors implement the *RV32IM* variant of the RISC-V ISA, which is supported by a complete toolchain. The software ecosystem supports the compilation of standard benchmarks, such as Dhrystone [101] and Coremark [102], targeting KeyV processors. It is presented in Section 5.1. Then, KeyV processors are implemented with the *TSMC65GP* 65 nm ASIC design kit, using Synopsys EDA tools. In Section 5.2 we present the EDA ecosystem supporting the implementation and the performance/power evaluation of KeyV processors. We chose *not* to compare the KeyV processors with other RISC-V processors in the literature. Indeed, as discussed in Section 2.3 about the AnARM [35] and the Opus2 [29], and in Section 3.1 about the Mini-Mips, it is difficult to evaluate the benefits of using a given design style when the circuits in balance vary in features and design intent, and are implemented with different technologies using different EDA flows. Instead, we propose *SynV*—our own synchronous alternative to KeyV—, which uses an in-order 6-stage pipeline. It is presented in Section 5.3. SynV allows drawing up fair comparisons with KeyV in terms of microarchitecture, synthesis flow, performance, area, and power consumption. To this end, it reuses most of the modules used in KeyV, implements the same variant of the ISA, and is synthesized with the same ASIC technology using the same EDA tools. In addition, SynV is synthesized with and without clock-gating. KeyV processors are presented in Section 5.4. The first KeyV processor, called KeyV_{362}, is an adaptation of the Mini-Mips using the RISC-V ISA and the KeyRing microarchitecture. It implements the same level of parallelism as the Mini-Mips using a $(E = 3, S = 6, \alpha = 2)$ KeyRing organization. The second KeyV processor, called KeyV_{661}, achieves the same level of parallelism as the SynV processor (twice that of KeyV_{362}) using a $(E = 6, S = 6, \alpha = 1)$ KeyRing organization. Further explanations on the microarchitectural details of KeyV_{362} and KeyV_{661}—in particular with respect to ILP—are provided in section 5.4.1. Finally, Section 5.5 compares the post-synthesis results of the four processors: SynV with and without clock gating, and KeyV with two different KeyRing organizations.

5.1 Software Ecosystem

This section presents the software ecosystem supporting the compilation and the execution of programs on the KeyV and SynV processors, including the RISC-V ISA, the associated toolchain, and the firmware designed in combination with the top level architecture of the processors.

5.1.1 RISC-V ISA

The Mini-Mips ISA has a very limited set of instructions, which is too small to support the use of a toolchain—a compiler and standard libraries—, and prevents the execution of programs written in a high level languages such as C or C++, including standard processor benchmarks. Instead of standard benchmarks, the performance of the Mini-Mips is evaluated with a very small program written in assembly that computes the Fibonacci series, which does not satisfy to the high standards requirement of modern computer engineering research.

Hence, the KeyV (and SynV) processors implement the RISC-V Instruction Set Architecture [43]. We chose RISC-V for several reasons [107]. First, unlike most other architectures, it is open-source. Thus, it comes without licenses concerns, and it is free from the fate of corporations. We believe that an open-source ISA is a desirable starting point to build a novel processor microarchitecture in a research environment. Indeed, in addition to facilitating the continuation of the project in the future, this choice also increases the visibility of the project within the research community, as the RISC-V trend is growing in universities as well as in industries. Then, the RISC-V ISA is designed to be stable and modular. It is built around a base specification, called RV32I, which can be extended. RV32I is small yet it supports a full software stack and should never change in the future [107]. This enables very small and low power implementation of RISC-V, such as KeyV. Finally, the RISC-V ISA emphasizes the isolation between the architecture and the microarchitectures, which is a welcomed feature when using unorthodox microarchitectures, such as KeyRing.

We chose to implement the `M` extension (RV32IM)—adding support for integer multiplication and division (mul/div)—in addition to the base specification. This allows to implement mul/div operations in hardware instead of using routines, which greatly improves the performance of the processors, as such operations are often used in benchmarks, especially in Coremark. In addition, it will allow us to compare the implementation of the mul/div FSMs using a synchronous and a KeyRing microarchitecture.

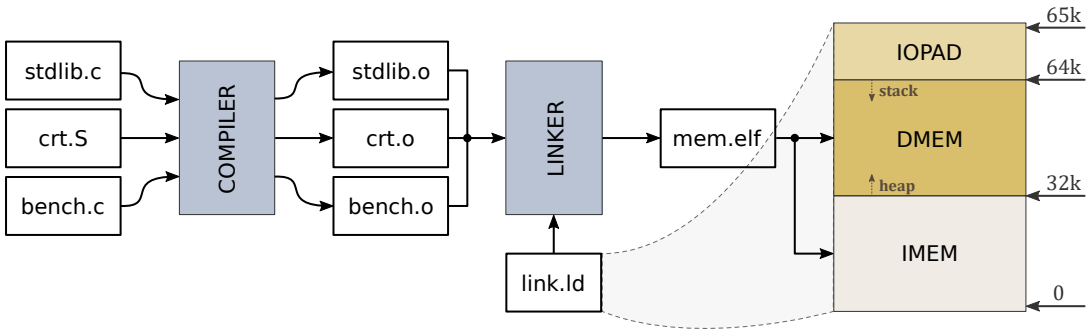


Figure 5.1 Software environment used for the compilation of benchmarks into RV32IM binaries targeting KeyV and SynV processors

5.1.2 Toolchain

Figure 5.1 shows the software ecosystem that enables the compilation of benchmark programs into RV32IM binaries targeting KeyV and SynV processors using the GNU Compiler Collection (GCC). The benchmark program is represented by the `bench.c` file. The `crt.S` file contains startup code that is executed after reset to call the main benchmark program. The `stdlib.c` file contains standard C functions that can be used by the benchmark (*e.g.* `malloc()`, `printf()`, *etc.*). These files are first compiled into independent binaries, and then linked to produce the final executable (`mem.elf`). The `link.ld` linker script specifies how the different code sections should be mapped, in the final binary, into the memory of the targeted processor. Memories are instantiated in the simulation environment as VHDL models: IMEM and DMEM share a 64kB space, while IOPAD uses a dedicated memory instance of 1kB (see Section 5.2). As the intended use of KeyV processors is limited to a simulation environment, and as the only programs to be executed are the benchmarks, it was easier to just write simplified versions of the `crt.S` and `stdlib.c` files rather than using those that are provided as part of the toolchain. Our custom versions of `stdlib.c` provides minimal support to the benchmarks, including simple procedures to interact with the memories (allocate memory on the stack and on the heap) and the performance counters (return the current values of the cycle and instruction counters). In addition, it contains an alternative to `printf()` that is adapted to the simulation environment. Instead of being displayed on a screen, messages output from programs are written into a dedicated space in memory—IOPAD in Figure 5.1—which is monitored by the simulation environment (see Section 5.2). Finally, the startup code in `crt.S` is best illustrated in Figure 5.2. After reset, the processor fetches instructions starting from the `start` label, and then call the `main()` function. When it returns, it first jumps to the `exit` trap before looping indefinitely in the `stop` procedure. This infinite loop is detected by the simulation environment, which ends the simulation (see Section 5.2).

```
start:
    beqz sp,reset
exit:
    csrr a0,mcause
    sw a0, 0(_IOPAD_START)
    j stop
reset:
    la sp, _STACK_START
    jal main
    ebreak
stop:
    j stop
```

Figure 5.2 Code snippet from `crt.S` illustrating how programs start and end

5.1.3 Benchmarks

The RISC-V software stack allows standard CPU benchmarks to be executed on the KeyV and SynV processors. Not only they allow to grade the performances of the processors, they also enable their functional verifications: first in the design phase, and then in post-synthesis timing simulations. Indeed, and this is especially true for the case of KeyV processors, a benchmark can uncover new timing errors in post-synthesis simulations that may come from shortcomings in the synthesis flow, or in the design, that were not caught by the Static Timing Analysis. The more complex the benchmark, the more combination of instructions are being executed, and the more paths are activated in the processor, uncovering new potential design issues. We discuss the complementary use of timing simulations and STA in more details in Section 5.2. We use four benchmarks of increasing complexity:

- *Basic*: This benchmark is provided as part of the GCC RISC-V toolchain. It is written in assembly and its purpose is to test all the instructions of the ISA. We used a modified version accounting for the specificities of the KeyV microarchitecture.
- *Fibo*: This benchmark is a simple C program computing the Fibonacci series, yet it is more complex than *basic* as it uses all the standard library functions.
- *Dhrystone*: This is the legacy CPU benchmark used to grade the performances of processors since the mid 1980's. It has many limitations, but it has the advantage of being simple, and running relatively quickly in timing simulations [101].
- *Coremark*: In many respects, this benchmark is the successor of Dhrystone. It is the most complex and complete benchmark, but it takes longer to run in simulation [102].

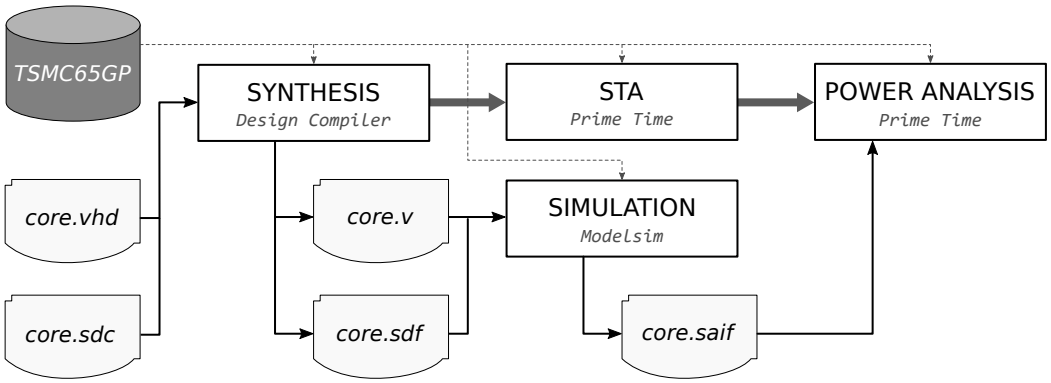


Figure 5.3 EDA environment used for the implementation, verification, and performance/power evaluation of KeyV and SynV processors

5.2 EDA Ecosystem

Unlike with the the Mini-Mips experiment, which relies on an FPGA emulation platform (see Section 3.6), we chose to assess the characteristics of the KeyV and SynV processors using an ASIC design kit and EDA environment to perform their implementation, simulation, verification, and power analysis. Although hardware emulation presents some advantages over simulations—it enables faster execution of benchmarks, and it allows results to be measured on real hardware instead of being evaluated in simulation—, its shortcomings in the context of prototyping asynchronous circuits—namely the limitations of the synchronous-oriented FPGA fabric, and the noise in the power measurements (see Section 3.8.3)—have led us to favor an ASIC design flow. Moreover, in addition to evaluating performance/power figures, our goal is also to demonstrate the effectiveness of the proposed timing-driven synthesis flow. In this respect, ASIC EDA tools are more suitable than their FPGA counterparts.

Figure 5.3 shows the EDA environment used for the implementation (synthesis), the verification (functional simulation and STA), and the performance and power consumption evaluation of the KeyV and SynV processors using the TSMC65GP ASIC design kit. In this diagram, *core* represents either the KeyV or the SynV processor. This typical EDA flow relies on Synopsys Design Compiler (DC) 2019.03 for the synthesis, Synopsys Prime Time 2019.03 for the STA and power analysis steps, and Mentor Graphics Modelsim 10.7 for the simulations. First, the processor (`core.vhd`) is synthesized using the TSMC65GP library targeting the requirements of the timing constraints (`core.sdc`), which are then verified using STA. Then, the netlist (`core.v`) and the associated SDF file (`core.sdf`) are used for the timing simulation of the processor, which provides a record of its activity (`core.saif`). Finally, this activity file is used to evaluate the power consumption of the processor.

Algorithm 5: Synthesis flow used for KeyV and SynV processors (Design Compiler)

```
    # Elaboration
 1  source init_tsmc65gp.tcl                    # Initialize TSMC65GP ASIC design kit
 2  analyze -format vhdl [design_sources]       # Compile design sources
 3  elaborate                                   # Elaborate the core
    # Timing constraints
 4  if Design is SynV then
 5      read_sdc synv.sdc
 6  end
 7  if Design is KeyV then
 8      read_sdc keyv.sdc
 9  end
    # Timing-driven synthesis
10  compile                                     # Synthesis
11  set_fix_hold [all_clocks]                   # Fix hold violations
12  compile -incremental_mapping                # Incremental synthesis
```

5.2.1 Synthesis flow

Algorithm 5 outlines the synthesis flow used for both KeyV and SynV processors, using DC. This standard synthesis flow starts with the initialization of the TSMC65GP design kit, which includes loading the timing libraries and setting the operating conditions. We used Composite Current Source (CCS) timing models, and we set the operating conditions to the typical-case (NCCOM) at TT/1V/25°C. Then, the VHDL description of the core is first compiled and elaborated, before applying the timing constraints. The SynV timing constraints are standards, targeting a global clock of 500MHz base frequency. The KeyV timing constraints are based on the KeyRing timing constraints (see Algorithm 4), and will be discussed in more depth in Section 5.4.5. Note that the only differences between the KeyV and SynV synthesis flows are the timing constraints. Indeed, as opposed to the Mini-Mips timing constraints, which need to be reloaded after each implementation step (see Figure 3.14), the KeyRing timing constraints need to be defined only once, enabling the use of a standard design flow. Following the timing constraints is the synthesis operation (`compile` command). Using configurations not shown in Algorithm 5, the synthesis optimizes the design for timing, meeting the requirements set in the timing constraints. During this timing-driven synthesis, DC fixes setup violations, but does not prevent hold violations. Indeed, in a standard design flow, hold violations are usually addressed after the Clock Tree Synthesis step, in the back-end. In a synchronous design, hold violations rarely appear post-synthesis. However, hold constraints are an important part of the KeyRing timing constraints, as discussed in Section 4.4, and must be addressed during the synthesis step. Hence, the final step of the synthesis flow is an incremental synthesis which optimizes the design in order to fix hold violations.

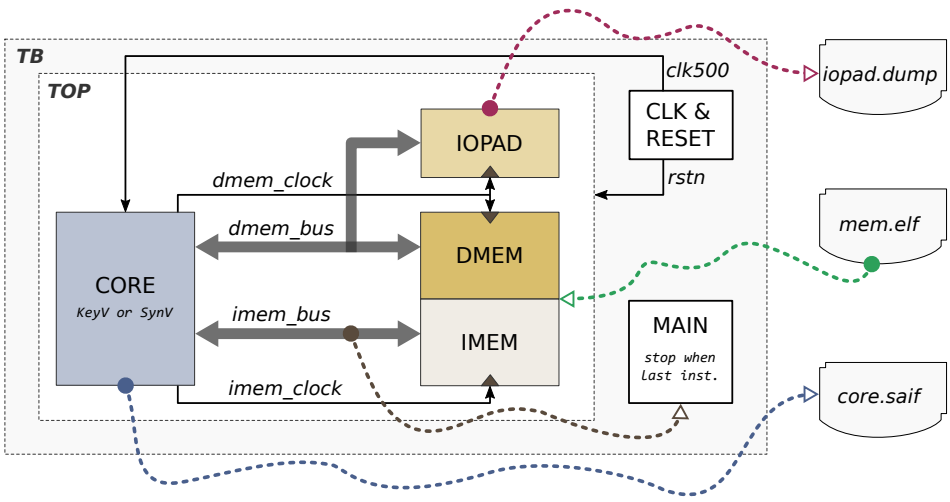


Figure 5.4 Overview of the simulation environment used to assess the performances and record the activity of KeyV and SynV processors while executing a benchmark

5.2.2 Simulation flow

Figure 5.4 shows the simulation environment used to evaluate the performances and record the activity of KeyV and SynV processors while executing a benchmark. Similarly to the Mini-Mips, the KeyV and SynV processors are synthesized without memories (as discussed, this work primarily focuses on the impact of the KeyRing microarchitecture on the *core*). Memories are implemented as VHDL models having an ideal latency of one cycle, and are interfaced with the *core* in a top-level wrapper, called TOP in Figure 5.4. The instruction memory (IMEM) and the data memory (DMEM) share the memory space of a 64kB Dual Port Memory. They are interfaced with the core through the `imem_bus` and the `dmem_bus` signals, which are composed of 16-bit address lines and 32-bit data lines. The scratchpad memory (IOPAD) uses a separate 1kB DPM instance that is interface with the *core* using a 10-bit address line and 32-bit data lines coming from the `dmem_bus` signals. Programs may write (read) data to (from) DMEM or IOPAD using the same signals: the appropriate memory is selected by the address, which value is correctly set in the program during its compilation according to the memory map defined in the linker script, as detailed in Section 5.1. At the beginning of the simulation, the compiled benchmark program (`mem.elf`—see Section 5.1) is loaded into the IMEM memory. While the benchmark is being computed, the activity of the *core* is recorded in the `core.saif` file. The simulation ends when the infinite loop instruction (see Figure 5.2) is detected on the IMEM data bus. Finally, the IOPAD memory is dumped into the `iopad.dump` file, which contains the data outputted by the benchmark, including performance results.

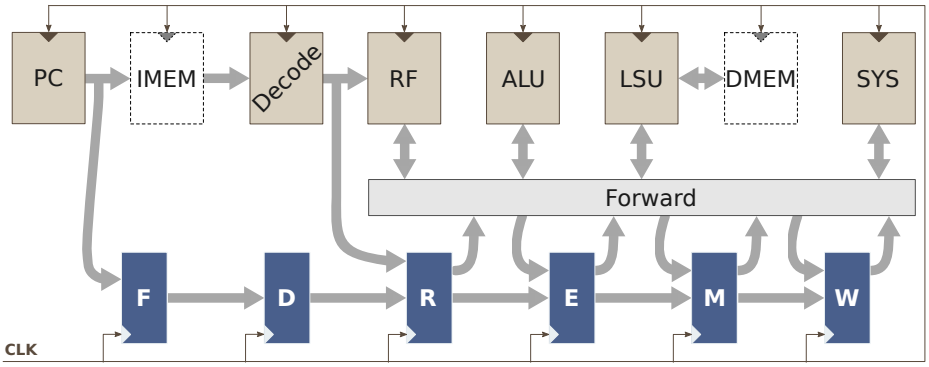
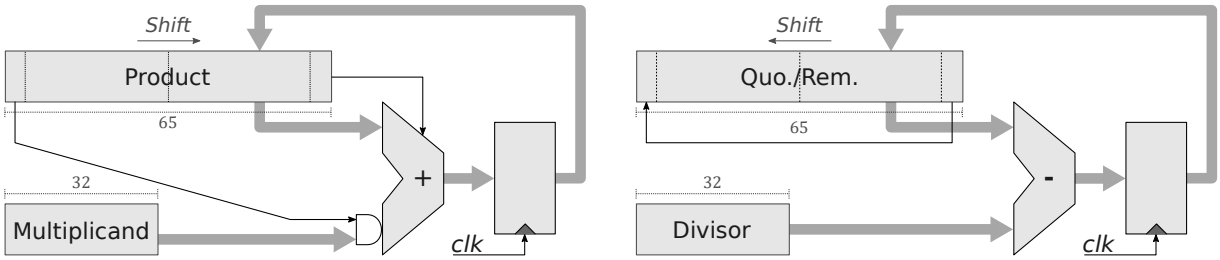


Figure 5.5 Overview of the SynV 6-stage pipeline microarchitecture

5.3 SynV: RISC-V Processor Based on a 6-stage Synchronous Pipeline

This section deals with the *SynV* processor, our own synchronous alternative to KeyV. This work focuses on making a *fair* comparison between the processors, in order to uncover, among the characteristics of the KeyV processors, those that are attributable to the KeyRing microarchitecture. In that spirit, SynV implements the RV32IM variant of the RISC-V ISA, and it is synthesized with the same EDA flow than KeyV, as detailed in Section 5.1. The SynV microarchitecture relies on an in-order single-issue synchronous pipeline of 6 stages. It contains sequential modules having a 1 cycle latency, which are reused in KeyV. It also contains a multiplier and divider submodules in the ALU having a 32 cycles latency, which, with the notable exception of the ad hoc clocking mechanism, are also reused in KeyV. The SynV cores are synthesized with and without clock-gating, and are constrained with a single synchronous clock targeting a 500MHz clock frequency. They will serve as reference points to compare the design trade-offs of KeyV cores in terms of microarchitecture, synthesis flow, performance, area, and power consumption. Detailed results are presented in Section 5.5.

Figure 5.5 presents the top-level view of the SynV microarchitecture. It is composed of 6 concurrent stages: *Fetch* (`F`), *Decode* (`D`), *Register Read* (`R`), *Execute* (`E`), *Memory* (`M`), and *Register Write* (`W`). This pipeline depth was chosen as it provides the same level of parallelism ($P = 6$) as the KeyV_{661} core (see Section 5.4). In the spirit of drawing up fair comparisons between SynV and KeyV cores, the modules are designed as FSMs having a 1-cycle latency—they are composed of combinational logic clouds implementing the desired functions, and have their outputs registered—, which allows them to be reused in both cores: in SynV, the modules use the global clock, while in KeyV, each module uses a different self-timed clock. Note that the memories are represented with dashed lines to illustrate the fact that they are not part of the core.



(a) Multiplication (b) Division

Figure 5.6 32-bit integer multiplication & division 32-cycle FSMs (taken from [5])

As with the Mini-Mips, sequential modules perform the basic operations of the processor: the Program Counter (PC) controls the address bus of the instruction memory; the Decode module decodes RV32IM instructions; the Register File (RF) is a 32×32 bits array of registers with two read ports and one write ports; the Arithmetic and Logic Unit (ALU), which operates on two 32-bit operands, contains an adder, a shifter, and logical submodules, as well as a multiplier and a divider submodule; the Load Store Unit (LSU) is interfaced with the data memory; and finally the System (SYS) module is responsible for handling system tasks, including Control Status Register (CSR) instructions which provide access to the performance counters. The multiplier and divider (mul/div) submodules of the ALU are presented in Figure 5.6. They both operate on 32-bit integers, using a 32-cycle FSM which basic principles are described in the computer arithmetic section of [5]: the multiplier computes the product of two integers using an *add* and a *right-shift* operation at each cycle; the divider implements a *non-restoring* division algorithm, which computes the quotient and the remainder using a *sub* and a *left-shift* operation at each cycle.

Because the pipeline is 6 stages deep, data hazards are handled by forwarding data from the `W`, `M`, and `E` stages to the `R` stage, as shown in Figure 5.5. Also note that structural hazards in the RF are addressed with a write-back buffer, which allows to read and write a register at the same cycle. As with the Mini-Mips (see Section 3.3), the SynV pipeline may be stalled for 1 cycle in case a data hazard is produced by a load operation. In addition, the pipeline is stalled for 32 cycles while a mul/div operation is being computed in the ALU. Branches and jumps are processed at the `E` stage. Thus, as a result of taken branches or jumps, three instructions in the pipeline are flushed. The same *predict-not-taken* branch strategy used in the Mini-Mips is also employed in SynV, although here the penalty is three cycles instead of one. This microarchitecture is far from being optimal, and many improvements could be brought to its design. However, as long as KeyV processors implements similar design decisions (branch penalty, stalls, *etc.*) the role of SynV is fulfilled.

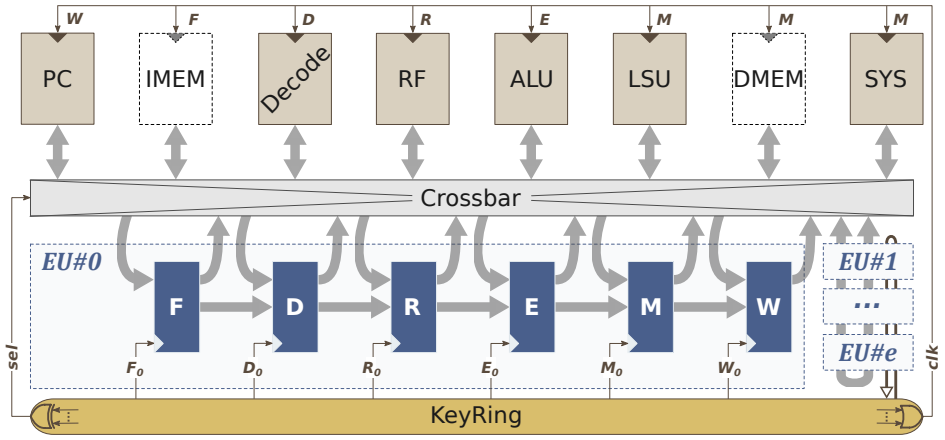


Figure 5.7 Overview of KeyV processors microarchitecture

5.4 KeyV: RISC-V Processors Based on the KeyRing Microarchitecture

In this section, we present the design of KeyV, in-order RISC-V processors implementing the RV32IM variant of the ISA using the KeyRing design principles developed in Chapter 4. First, the microarchitecture of KeyV processors is presented in Section 5.4.1. Figure 5.7 is an overview of this microarchitecture, which shows the main characteristics of the processors. A first version—called KeyV_{362}—is proposed as an adaptation of the Mini-Mips to the RV32IM specification based on the KeyRing microarchitecture. It uses a $(E = 3, S = 6, \alpha = 2)$ KeyRing configuration, which provides a $P = 3$ level of parallelism. A second version of KeyV—called KeyV_{661}—is proposed as an improvement over the KeyV_{362} microarchitecture. It uses a $(E = 6, S = 6, \alpha = 1)$ KeyRing configuration, which provides a $P = 6$ level of parallelism, equal to the level of parallelism achieved with the 6-stage pipeline of SynV. Then, Section 5.4.2 presents modifications brought to the KeyRing, and to the KU architecture, to handle stalls in KeyV. We show that stalls are the source of hazards in the KeyV_{661} microarchitecture, and we propose solutions to alleviate them. Section 5.4.3 shows the synchronization of the KeyRing with synchronous modules, and Section 5.4.4 details the implementation of the mul/div submodule FSMs using an inner-KeyRing. Finally, Section 5.4.5 presents the practical adaptation of the KeyRing timing constraints to the case of KeyV processors. In particular, we propose additional constraints for the inner-KeyRing used in the mul/div submodules, and its seamless integration with the main KeyRing.

5.4.1 KeyV$_{362}$ & KeyV$_{661}$ microarchitectures

Figure 5.7 depicts the basic principles of the microarchitecture of KeyV processors. Execution Units in KeyV are composed of 6 stages—*Fetch* (`F`), *Decode* (`D`), *Register Read* (`R`), *Execute* (`E`), *Memory* (`M`), and *Register Write* (`W`)—, which, in addition to the sequential modules, are the same as in SynV. In addition to generating self-timed clocks, and orchestrating the stages concurrency across EUs, the KeyRing also generates control signals to arbitrate the access to shared resources—*i.e.* the sequential modules—in the crossbar (see Section 4.3). Resources are triggered by different clocks depending on the stage at which they are operated. Clocks coming from the same stage from different EUs are ORed together before going to a resource, as illustrated in Figure 5.7, and as discussed in Section 4.3.3. For example, the `F` clock triggering the IMEM memory is driven by $\texttt{F}_0$ `or` $\texttt{F}_1$ `or ... or` $\texttt{F}_e$. Consequently, the PC, which provides the address to IMEM, is clocked by the `W` clock, and the Decode module, which decodes the instruction word coming from IMEM, is clocked by the `D` clock. Following this reasoning, the ALU is clocked by the `E` clock, and the LSU, DMEM, and SYS modules are clocked by the `M` clock. There is a special case to consider for the RF: a concurrent access at the `R` and `W` stages by different EUs may cause a structural conflict between their clocks. To solve it, similarly than with the Mini-Mips, the RF is only clocked by `R` clocks, and the result of instruction i is written back to the RF at the `R` stage of instruction $i + 1$. One-hot selection signals—noted `sel` on Figure 5.7—coming from the KeyRing arbitrate the access of each EU stage to the modules. For example, when the E_0 clock is released, the MUXes in the crossbar guide the data coming from the R_1 stage to the ALU until the E_1 clock is released, after which they guide the data coming from the R_2 stage until the E_2 clock is released and so on. Data issued from the modules are distributed by the crossbar to appropriate stages in all Execution Units, and saved in their respective registers with their dedicated clock. For example, data computed in the ALU with the `E` clock, using operands provided by the R_1 stage, are saved in the M_1 stage register. The crossbar is also responsible for forwarding data between EUs, and distributing control signals. In KeyV, microarchitectural hazards, caused by the overlapping of instructions across EUs, are addressed as follows:

- *Data hazards*—They would occur when at least one operand used by an instruction in an EU is being used as a destination register by a preceding instruction in another EU. In KeyV, similarly than with the Mini-Mips, they are addressed with the crossbar: the operands addresses are compared with the destination addresses of preceding instructions, and data are forwarded in case of a match. If a result is not available, the `R` KU may be stalled until the producer EU has finished its computation. The detail of the stall mechanism is presented in Section 5.4.2.

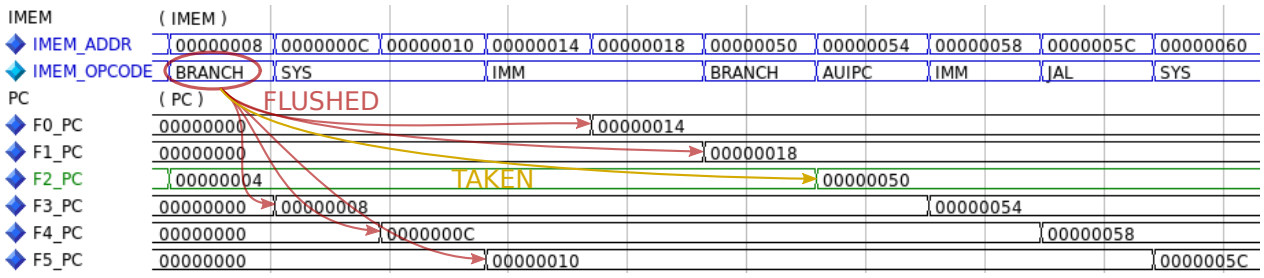
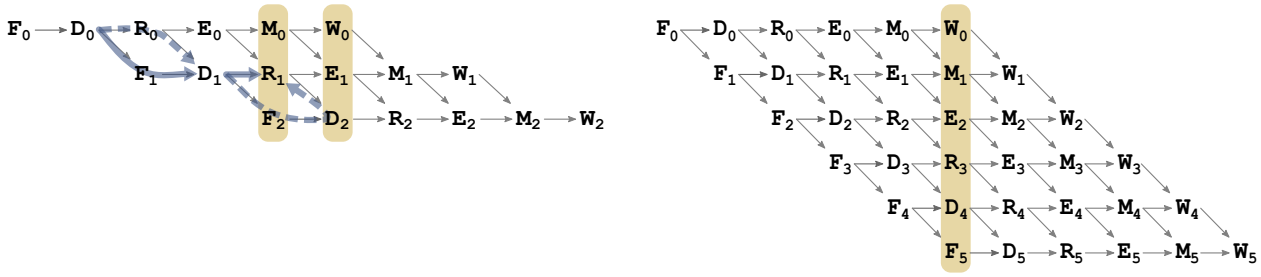


Figure 5.8 Branches: all but the branching EU are flushed (illustrated with KeyV_{661}).

- *Control hazards*—They would arise when an instruction modifies the PC (branches and jumps). In KeyV, they are handled by flushing out instructions from EUs. As SynV, KeyV uses a simple *predicted-not-taken* branching strategy, where the branch outcome is computed during the `E` stage and exposed from the `M` register. Every EUs but the one responsible for the transfer are flushed, and the branching EU processes the destination instruction. Figure 5.8 shows the branching mechanism in KeyV (it is illustrated with the KeyV_{661} microarchitecture). The BRANCH instruction is fetched by EU_2 at address `0x04`. The following instructions fetched by $\text{EU}_{3,4,5,0,1}$ are flushed, and the target instruction is fetched at address `0x50` by EU_2.

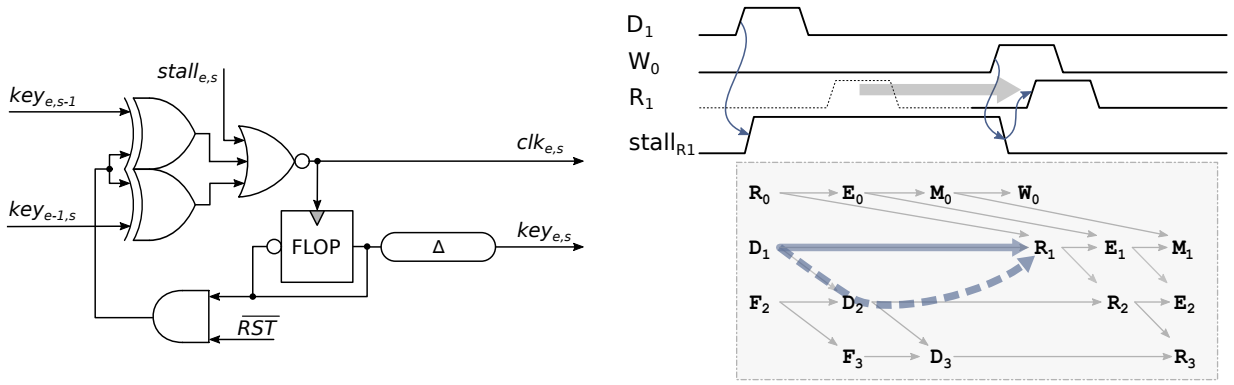
Reusing the modules in KeyV and SynV improves the relevance of the comparisons between the KeyRing and the synchronous microarchitecture. Moreover, it also suggests improved design reuse capabilities of synchronous IP blocks in KeyRing circuits, compared with standard asynchronous circuits. If the synchronous block is a simple combinational module with registered outputs, requiring only one clock edge to be updated (*e.g.* the main modules in KeyV), then it can directly be clocked by a self-timed clock generated by the KeyRing. No additional synchronization is required. On the other hand, if the synchronous block needs multiple cycles to produce a result, two methods can be employed to synchronize its content with the main KeyRing: *i)* The module can be clocked by a synchronous clock and synchronized with the KeyRing using a standard synchronizer. This method is demonstrated in Section 5.4.3 to synchronize the results of the performance counters—which, for obvious reasons, use a synchronous clock—with the `M` clock in the SYS module. *ii)* The module can be clocked by a dedicated $(1, 1, 1)$ inner-KeyRing and synchronized with the main KeyRing by a dedicated synchronizer. This method is demonstrated in Section 5.4.4 with an inner-KeyRing that is used to produce the 32-cycles required by the mul/div FSMs. Additional timing constraints are required for the inner-KeyRing and for the associated synchronizer, as discussed in Section 5.4.5. Note that in both cases the main KeyRing can be stopped while the module is computing a result using the stalling mechanism presented in Section 5.4.2.



(a) KeyV$_{362}$ (b) KeyV$_{661}$

Figure 5.9 Comparison of Instruction Level Parallelism in KeyV processors

Figure 5.9 illustrates the main differences between the KeyV$_{362}$ and the KeyV$_{661}$ microarchitectures. Following the design principles developed in Chapter 4, these two microarchitectures correspond to two possible KeyRing configurations for a fixed number of stages $S = 6$: 3 or 6 EUs. KeyV$_{362}$ has the same KeyRing configuration as the Mini-Mips. It is composed of 3 EUs of 6 stages with a dependency shift between stages $\alpha = 2$, which provides an ILP of 3. Although EUs have 6 stages, only 3 of them can be exploited concurrently because of the dependency shift—`M`, `R`, `F`, or `W`, `E`, `D`—, as shown in Figure 5.9a. The resulting level of parallelism achieved by KeyV$_{362}$ is half that of the SynV processor. KeyV$_{661}$ is composed of 6 EUs of 6 stages with a dependency shift between stages $\alpha = 1$, which provides a level of parallelism of 6, equal to that of SynV. Indeed, similarly to the 6-stage pipeline, 6 EU stages can be exploited concurrently, as shown in Figure 5.9b. The reason for considering the KeyV$_{362}$ microarchitecture is its resilience against hold violations. In Section 3.4.2 (*Mini-Mips*), we have assimilated these hold violations to Write After Read hazards because they involve the arbitration of shared resources between EUs stages. Then, we have shown in Section 4.4.1 (*KeyRing*) that these WAR hazards are better defined as protocol-level hold RTCs (see Figure 4.10), which can be constrained as standard hold constraints. Figure 5.9 illustrates these hold constraints between `D` and `R` stages in KeyV processors with dashed and plain blue arrows. Using the terminology from Section 4.4.1, dashed arrows represent the launch paths, which should arrive late, while plain arrows represent the capture path, which should arrive early. Signals on the launch path should not reach their destination before those on the capture path. For example, data exposed from the D_1 stage to the RF should be clocked by the R_1 clock before data from the D_2 stage is updated and directed to the RF by the crossbar. In the KeyV$_{661}$ microarchitecture, since `R` and `D` stages are concurrent, this hold condition can only be enforced through timing constraints, as proposed in Section 4.4.2. In the KeyV$_{362}$ microarchitecture by contrast, this hold condition is architecturally enforced through KUs dependencies in the KeyRing, thus it is always met.



(a) KU Architecture (b) R_1 stall example ($KeyV_{661}$)

Figure 5.10 Implementation of *stalls* in KeyV

5.4.2 Handling stalls

Stalling a processor may be required to address data dependencies. In SynV, a 1-cycle stall is applied to the pipeline when instruction i depends on the result produced by instruction $i-1$ via a load operation. The detection of the data dependency at the E stage leads to flushing the `E` registers and stalling all preceding stages at the next cycle, which allows the result to be forwarded between the `M` stage and the `E` stage. In KeyV, stalls are handled differently by making use of the self-timed characteristics of the clocks. As discussed in Section 5.4.1, data dependencies are addressed by forwarding data between EUs in the crossbar: instruction results are exposed to the crossbar from the `W` stage registers, and forwarded operands are saved in the `R` stage registers. If a result is not available when the requesting EU reaches the `R` stage, then the `R` KU is stalled, preventing the release of the `R` clock until the producer EU reaches the `W` stage. Figure 5.10 shows the implementation of stalls in KeyV. Modifications brought to the KU architecture (see Figure 4.4 for comparisons) introduce an additional input signal, $stall_{e,s}$, directly connected to the NOR gate, which delays the release of the clock as long as it is active. As a result, the activation of dependent KUs is delayed, which has a snowballing effect on the whole KeyRing. The set and reset of the $stall_{e,s}$ signal must be properly performed to prevent the KeyRing from staying idle, and to avoid glitches. Stalling in KeyV is illustrated in Figure 5.10b. A data dependency between EU_1 and EU_0 is detected in the D_1 stage, which activates the $\texttt{stall}_{R1}$ signal, and prevents the release of the R_1 clock. The $\texttt{stall}_{R1}$ signal is deactivated when EU_0 reaches the `W` stage, releasing the R_1 clock and enabling the capture of the forwarded data in the R_1 stage registers. Note that the typical duration of stalls are about twice as long in $KeyV_{661}$, where `R` and `W` stages are two stages apart, as in $KeyV_{362}$, where they are only one stage apart (see Figure 5.9).

Key Units with stalling capabilities are used in KeyV for the R and E stages only. Stalling the R stage is required to address data dependencies between EUs, as discussed, while stalling the E stage is used to stop the KeyRing while the mul/div FSMs are being used (see Section 5.4.4). Indeed, since the mul/div FSMs last for 32 cycles, stalling the E stage of any EU eventually stops the whole KeyRing—due to KUs dependencies—until the mul/div operation is finished. The stalling mechanism used in KeyV presents some advantages compared with synchronous stalls used in SynV. First, there are no needs for flushing instructions. Indeed, in SynV stalls consist in stopping the pipeline upstream, which requires the flushing of the instruction present in the stalling stage. By contrast, in KeyV stalls essentially rely on delaying the release of a clock, which causes no additional data movement in the circuit. Second, it does not increases the clocking activity. Indeed, stalls in SynV introduce additional clock edges, which contribute to the clocking activity (although this effect is of lesser importance in clock-gated designs). In KeyV, by contrast, because the release of the clock is delayed, stalls cause no additional clocking activity.

However, stalls in KeyV also have drawbacks. First, the $stall_{e,s}$ signal exposes the KeyRing to glitches, as it comes from the crossbar. In the KeyV design, we mitigated their impact by making sure that the data dependency detection circuitry stays stable during the R clock active time frame. Although they do not rise any errors in timing simulations, such glitches may be the cause of reliability issues in the circuit, and thus should be properly addressed in future works. Second, introducing stalls in the KeyRing changes the KUs organization, as modeled by the dependency graph of Figure 4.7. In particular, some of the timing conditions used to constrain timing paths in KeyV, as discussed in Section 5.4.1 and in Section 4.4.1, are completely disrupted by the unpredictable delays introduced by the stalls. Setup conditions actually benefit from stalls. Indeed, delaying the arrival time of the capture clock is in favor of meeting the setup condition. In other words, the worst case scenario that should be constrained by the setup condition occurs when stalls are not involved. Thus, there are no need for additional setup constraints to handle stalls. On the other hand, stalls work against meeting hold conditions. Indeed, hold conditions are violated when the capture path is longer than the launch path, a situation that is more likely to occur when stalling. Figure 5.10b shows how the ILP is altered when the $\mathtt{R}_1$ stage is stalled. In particular, it depicts the hold RTC, with the plain arrow representing the capture path and the dashed one representing the launch path. In this example, when the $\mathtt{D}_1$ clock is released it triggers the Decode module—which outputs the decoded instruction from EU_1—and it initiates the race between $\mathtt{D}_1 \rightarrow \mathtt{R}_1$ (capture path) and $\mathtt{D}_1 \rightarrow \mathtt{D}_2 \rightarrow \mathtt{R}_1$ (launch path). When $\mathtt{D}_2$ is released, it triggers the Decode module, replacing the decoded instruction from EU_1 by the decoded instruction from EU_2. If the $\mathtt{R}_1$ stage has not captured the data latched in the Decode module by the

$\texttt{D}_1$ clock before it is replaced, the information is lost (Write After Read). Here, it is clear that the stall exacerbates hold violations, with $\texttt{R}_1$ arriving so late that the Decode module has had the time to be updated by $\texttt{D}_2$, $\texttt{D}_3$, and even $\texttt{D}_4$, before the information is captured by $\texttt{R}_1$. Because the stall delay is unpredictable, and because it is so long compared to the delays involved in the hold constraints—a stall may last for up to 3 cycles (3 DE)—, this *architectural hold* cannot be addressed with a timing constraint. A first solution consists in using the KeyV_{362} microarchitecture, where the release of $\texttt{D}_2$ depends on the prior release of $\texttt{R}_1$, which protects against these stall-induced WAR hazards. However, we have seen that it implies important performance penalties. Another solution, which we have used in KeyV_{661}, is to move the decode logic from the Decode module into each Execution Unit. That way, decoded instructions cannot be overwritten by another EU. This solution addresses the problem not by removing the hold violation due to stalls—this cannot be done in KeyV_{661}—, but by preventing it to have an effect on the state of the processor. The main drawback of this method is that it introduces an area overhead: the decode logic that was *shared* between EUs is now *replicated* across all EUs. To mitigate this overhead, we have only replicated the part of the decode logic that must be processed at the D stage, and left the rest in the shared Decode module, which is now triggered by the R clock instead of the D clock.

5.4.3 Synchronization with synchronous modules

When designing using a KeyRing microarchitecture, one often realizes that clock edges are scarce. While it should work in favor of improving energy efficiency by reducing the clocking activity—after all, minimizing the number of clock edges is the main goal of this ad hoc clocking scheme—, it also has the drawback of limiting the realm of possibility. In designing the KeyV processors, we found two situations where the KeyRing was a limiting factor. First, when reusing an IP requiring multiple cycles to produce a result. This is the case of the mul/div FSMs, which require 32 cycles to complete. Second, when the module must be clocked at a fixed pace. This is the case of the cycle counter, which is used in the SYS module to measure time by counting the number clock ticks, given that their frequency is fixed and known. As mentioned previously, the first case may be addressed in two ways: either *i)* by using an external synchronous clock and synchronizing the result with one of the self-timed clock generated by the KeyRing (this method is explored in this section); or *ii)* by using a dedicated ($E = 1, S = 1, \alpha = 1$) KeyRing—we call that an *inner-KeyRing*—that is synchronized with the main KeyRing using a dedicated synchronizer (this method is explored in Section 5.4.4). In the second case, however, there are no alternatives to using a synchronous clock, as the frequency of self-timed clocks generated by a KeyRing are neither fixed nor known.

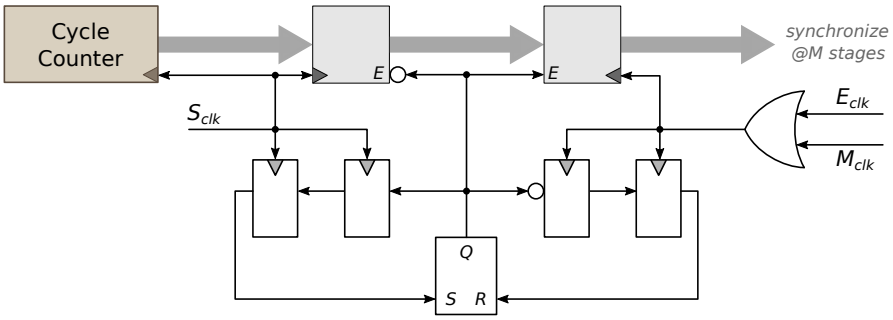


Figure 5.11 Synchronization of the (synchronous) cycle counter with the KeyRing

The cycle counter is a 64-bit counter which increments its value at each cycle since reset. Its goal is to provide a measure of time, which is why it must be clocked synchronously from a source having known characteristics. Accessing the current value of the cycle counter is performed in a program via the CSR instruction `rdcycle`, which returns the number of clock ticks that were issued since reset. Benchmarks typically call this instruction twice, once before the execution of the main program and once after. We then use the difference between these two measurements multiplied by the known period of the clock to estimate the execution time of the benchmark. Figure 5.11 shows the synchronization circuit that we used to interface the cycle counter with EUs in KeyV. The cycle counter uses an external synchronous clock (S_{clk} in Figure 5.11), targeting a 500 MHz frequency as in SynV. It is part of the SYS module, which is clocked by the `M` clock (M_{clk} in Figure 5.11). The role of the synchronizer is thus to safely save the content of the cycle counter into a register clocked by the `M` clock, given that S_{clk} and M_{clk} belong to two mutually-asynchronous clock domains. To this end we used a *shared latch synchronizer* inspired from [17]. It relies on a signaling SR latch, which outputs a control signal Q that is set (S) by the producer and reset (R) by the consumer. This control signal is carefully synchronized using two flip-flops in both the producer and the consumer clock domains to handle metastability. Here the producer is the synchronous cycle counter, and the consumer is the SYS module. Since we need two clock edges two synchronize the control signal, we ORed the `E` clock with the `M` clock on the consumer side. This synchronization scheme works well to safely transmit data between a synchronous module and a KeyRing system, in situations where it is acceptable to only transmit part of the data from one side to the other (otherwise, a FIFO would be required), such as it is the case with the cycle counter. It is particularly well adapted to KeyRing circuits because the handshake signaling between producer and consumer is directly handled by the SR latch, avoiding the overhead of generating such control signals in EUs and guiding them through the crossbar, with appropriate timing constraints.

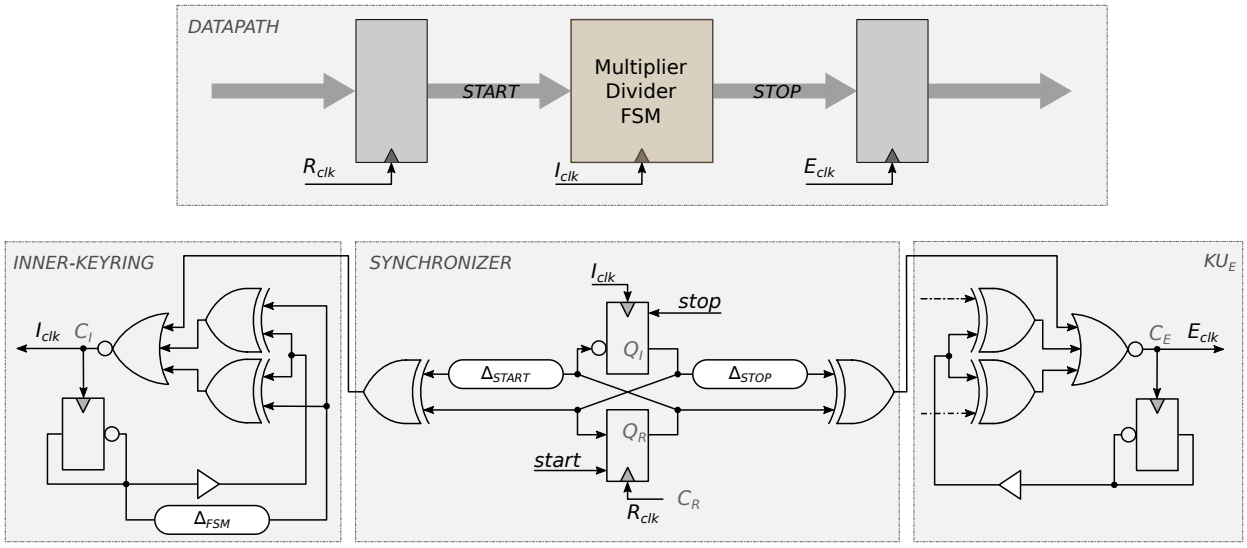


Figure 5.12 Proposed circuits to clock the mul/div FSMs using a $(1, 1, 1)$ inner-KeyRing, and to synchronize it with the main KeyRing

5.4.4 Multiplier/Divider submodule inner-KeyRing

In Section 5.3 we have seen that the multiplier and divider submodules of the ALU are FSMs that require 32 clock cycles to complete. In KeyV, instead of using an external synchronous clock to generate the 32 clock cycles, we propose an *inner-KeyRing*. This inner-KeyRing uses the same Key Units as those used in the main KeyRing, but its architecture is based on a $(E = 1, S = 1, \alpha = 1)$ configuration. Figure 5.12 shows the inner-KeyRing architecture along with the synchronization circuitry that allows to integrate it within the main KeyRing. It also shows the datapaths around the mul/div FSMs to provide a context for the requirements of the synchronizer. As in SynV, operands for the mul/div operations are provided from the `R` stage—through the *START* path—, and the result is saved in the ALU register—through the *STOP* path—at the `E` stage. In contrast with SynV, in which preceding stages in the pipeline are stalled during a mul/div operation, in KeyV the `E` clock is only released when the mul/div operation is finished. The inner-KeyRing generates the `I` clock signal (I_{clk}) that clocks the mul/div FSMs. It is composed of only one KU with stalling capabilities (see Figure 5.10a), which is connected to itself through one Delay Element (Δ_{FSM} on Figure 5.12). Both *key* inputs of the KU are connected to the feedback wire coming from the DE, such that clock pulses are released in a synchronous like manner: the rising edge of the clocks are separated by a period equal to the propagation delay through the Δ_{FSM} DE, and this delay should match the worst case delay in the mul/div FSMs. The *stall* input of the KU is used to start and stop the inner-KeyRing, along with the mul/div FSMs.

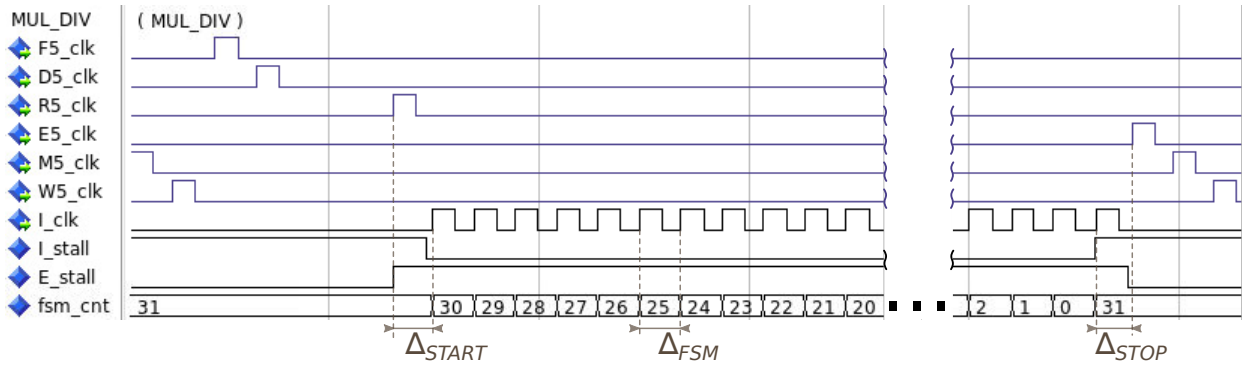


Figure 5.13 Simulation snapshot showing the use of the inner-KeyRing to generate the local clock for the mul/div FSM while stalling the main KeyRing

The synchronizer controls the *stall* inputs of both the inner-KeyRing and the `E` KUs of the main KeyRing, such that their activation are mutually exclusive. When a mul/div instruction is decoded at the `D` stage of an EU, it sets the *start* flag to 1. When the `R` clock is released, the state of the associated flip-flop toggles, which *i)* immediately stalls the `E` KU, preventing the release of the `E` clock until the mul/div operation is finished; and *ii)* starts the inner-KeyRing after a delay Δ_{START}, matching the worst-case delay of the logic between the `R` stage and the mul/div FSMs. By means of successive dependencies in the main KeyRing, stalling one of the `E` KU leads to halting all the KUs until the stall signal is released. Similarly, after 32 cycles the *stop* flag is set and the state of the associated flip flop is toggled by the `I` clock. This *i)* immediately stalls the inner-KeyRing, stopping its activity until the next mul/div instruction; and *ii)* deasserts the stall input of the `E` KU after a delay Δ_{STOP}, matching the worst case delay of the logic between the mul/div FSMs and the `E` stage. Figure 5.13 shows a simulation snapshot illustrating the execution of a mul/div operation in EU_5 of the $KeyV_{661}$ processor. When `R5_clk` is released, the `E_stall` signal is asserted, preventing the release of `E5_clk`, and the `I_stall` signal is deasserted after the Δ_{START} delay, starting the release of `I_clk` from the inner-KeyRing for 32 cycles. Note that `I_clk` has a period of Δ_{FSM}. At the end of the mul/div operation, the `I_stall` signal is asserted, preventing the release of `I_clk` and stopping the inner-KeyRing, and the `E_stall` signal is deasserted after the Δ_{STOP} delay, enabling the capture of the result by the ALU registers clocked by `E5_clk`.

Compared with using an external synchronous clock and a shared latch synchronizer, as presented in Section 5.4.3, the main advantage of using the inner-KeyRing is its seamless integration with the main KeyRing. Indeed, the start and stop of the main KeyRing, and of the inner-KeyRing, is performed by the synchronizer without metastability. However, it implies more complex timing constraints, as will be detailed in the following Section.

Algorithm 6: KeyV$_{661}$ timing constraints pseudo code

```
    # KeyRing objects
 1  create a KeyRing object with a (6,6,1) configuration for the main KeyRing
 2  create a KeyRing object with a (1,1,1) configuration for the inner-KeyRing
 3  foreach KeyRing object do
 4      define the DE configurations of each KU
 5      define the endpoints of each KU
 6  end
    # KeyRing timing constraints
 7  foreach KeyRing object do
 8      apply the KeyRing timing constraints as defined in Algorithm 4
 9  end
    # Synchronizer timing constraints
10  apply the synchronizer timing constraints as defined in Algorithm 7
    # Cycle counter
11  create clock S_clk on cycle counter with a 500MHz frequency and add it to its own group
```

5.4.5 Timing constraints

The timing constraints used for the KeyV processors are derived from the KeyRing timing constraints defined in Section 4.4. Both KeyV$_{362}$ and KeyV$_{661}$ use similar timing constraints, but those used in KeyV$_{661}$ are more difficult to enforce, particularly hold constraints as discussed in Section 5.4.1. Thus, in the following we focus on the timing constraints targeting the KeyV$_{661}$ microarchitecture. The overall structure of these timing constraints are presented in Algorithm 6. They rely on the KeyRing Tcl library (see Section 4.4.3), which provides convenient methods to deal with KeyRing timing models. Thus, the first step consists in creating and configuring KeyRing objects: one for the main KeyRing with a $(6, 6, 1)$ configuration, and one for the inner-KeyRing with a $(1, 1, 1)$ configuration. Configurations specific to the circuit—DE configurations and endpoints pin names that are used for the constraints definitions—are then defined for each object. Once these configurations are performed, the KeyRing timing constraints detailed in Algorithm 4 can be generically applied for both the main KeyRing and the inner-KeyRing. Indeed, one of the advantages of the KeyRing timing model proposed in Section 4.4, and its implementation in the KeyRing Tcl library, is that it allows to define the timing constraints of any KeyRing circuit in a generic way. Here we make use of this generality in two manners: first, by having generic timing constraints definitions for both the main KeyRing and the inner KeyRing; and second, by having generic timing constraints definitions for both the KeyV$_{362}$ and the KeyV$_{661}$ microarchitectures. Now that the paths in the main KeyRing (EUs and shared resources) and in the inner-KeyRing (mul/div FSMs) are constrained, the last paths left to constrain are those crossing KeyRing domains (synchronizer), and those in the synchronous cycle counter.

Algorithm 7: Synchronizer timing constraints pseudo code

```
    # Root clocks
1   foreach e in EUs do
2       generate clock start_e from R_e on Q_R
3       generate clock stop_e from I on Q_I
4   end
    # Start setup launch & capture clocks
5   foreach e in EUs do
6       generate clock start_sl_e from R_e on C_{e,R}
7       generate clock start_sc_e from start_e on C_I
8       add these clocks to the start group
9   end
    # Stop setup launch & capture clocks
10  foreach e in EUs do
11      generate clock stop_sl_e from I on C_I
12      generate clock stop_sc_e from stop_e on C_{e,E}
13      add these clocks to the stop group
14  end
    # Exceptions
15  define groups of clocks as asynchronous
16  define false paths from/to root clocks
17  define false paths from capture clocks
18  define false paths to launch clocks
    # Misc.
19  propagate all clocks
20  define zero-cycle path from launch to capture clocks
21  set case analysis on the DEs
22  set uncertainty to capture clocks
```

The synchronizer timing constraints are depicted in Algorithm 7. Their definitions are based on the same principles that was used for the KeyRing timing constraints (see Algorithm 4). Basically, each path is constrained between a *launch* and a *capture* clock, with both clocks originating from a common source that the STA engine is able to retrieve. Note that the notations used in Algorithm 7 refer to the Figure 5.12. Here, we have two paths to constrain: the *START* path—from the `R` clock to the `I` clock—and the *STOP* path—from the `I` clock to the `E` clock—, and these paths should be constrained for each EU. Moreover, since both paths are only used once each time a mul/div instruction is processed (see Figure 5.13), there is no need for hold constraints. We first define *root clocks* that are intended to be used as reference clocks for the definition of launch and capture clocks: for each EU we define one root clock ($start_e$) for the START path on the output of the start flip-flop (Q_R), and another one ($stop_e$) for the STOP path on the output of the stop flip-flop (Q_I). For the START paths—from each EU `R` stage to the mul/div modules—the launch clocks ($start\_sl_e$)

are defined from the R clock of each EU stage (R_e) on the clock pin of each `R` KU ($C_{e,R}$), while the capture clocks ($start\_sc_e$) are defined from the root clocks ($start_e$) on the clock pin of the inner-KeyRing `I` KU (C_I). For the STOP paths—from the mul/div modules to each EU `E` stage —clocks are defined in the same manner, symmetrically. Then, similarly to what is done in the KeyRing timing constraints (see Algorithm 4), each group of clocks—*start* and *stop*—is defined as mutually asynchronous with the others to prevent inter-clock constraints; root clocks are excluded from timing analysis, and launch (capture) clocks are prevented from capturing (launching) timing paths, using *false paths* definitions. Clocks are *propagated* such that the constraints account for the contribution of the clock paths delays (including the DEs delays), and setup and hold timing checks are defined as *zero-cycle* paths, which allows to adequately use the START (STOP) clock path delay as a constraint for the START (STOP) datapath. Finally, DEs lengths are set in the timing engine with *case analysis* directives, and setup margins are added using *uncertainty* directives. Coming back to the KeyV timing constraints (Algorithm 6), we conclude by defining a 500MHz synchronous clock for the cycle counter (S_{clk}). It is added to its own clock group, that is set to be mutually asynchronous with all other clock groups in the design.

In summary, this section has presented the design and the associated timing constraints of KeyV processors. They implement the RV32IM RISC-V ISA, with EUs of 6 stages and shared resources that are common with the synchronous SynV processor, thus maximizing the relevance of their comparisons in the following Section. KeyV processors share a common microarchitecture which is based on the KeyRing design style developed in chapter 4. The KeyV_{362} variant uses a $(3, 6, 2)$ KeyRing configuration, which presents the advantage of being resilient against hold violations. However, it only achieves half the level of parallelism that is achieved by SynV. The KeyV_{661} variant uses a $(6, 6, 1)$ KeyRing configuration, achieving the same level of parallelism as SynV. However, hold conditions must be enforced through timing constraints using timing-driven synthesis. Stalls in KeyV consists in delaying the release of a clock, which dynamically disrupts the KeyRing organization. In KeyV_{661}, they may cause architectural hold violations (WAR hazards) that cannot be addressed with timing constraints. Instead of fixing them, we prevented these hold violations from having an effect on the states of the processor. We then proposed two methods to interface KeyV with synchronous IPs. First, the cycle counter, clocked by a 500MHz external clock, is synchronized with EU stages through a shared latch synchronizer. Second, the mul/div FSMs, clocked by a $(1, 1, 1)$ inner-KeyRing, is seamlessly integrated within the main KeyRing using a dedicated synchronizer, and a set of timing constraints. Finally, we have seen that KeyV processors are generically constrained, using the KeyRing timing constraints and its associated Tcl library.

5.5 Experimental Results

This section presents experimental results obtained from the synthesis, and the post-synthesis simulations, of KeyV and SynV processors following the experimental protocol detailed in Section 5.2. The synthesis was performed by Synopsys Design Compiler, the STA and the power analysis were performed by Synopsys Prime Time, and simulations were performed by Mentor Graphics Modelsim. All the results presented in this section have been generated with the TSMC65GP 65 nm ASIC design kit at the typical PVT corner (TT/1V/25°C). The synthesis runs were executed on a host computer having 32GB of RAM using 16 threads of an Intel Xeon E5-2640 clocked at 2,5 GHz. First, Section 5.5.1 presents the Static Timing Analysis of the KeyV processors. These results are the most important, as they contribute to demonstrate the effectiveness of the proposed KeyRing timing model and synthesis flow, which is the main contribution of this thesis. Then, Section 5.5.2 and Section 5.5.3 respectively compare KeyV and SynV processors in terms of area and performances. In particular, we compare the benchmark scores, and the power consumption, achieved by the processors from the post-synthesis timing simulation of the Dhrystone and the Coremark benchmarks. The following four netlists are compared:

- SynV : The synchronous processor presented in Section 5.3 with a 6-stage pipeline, synthesized with a 500 MHz clock frequency target.
- SynV_{cg} : A clock-gated version of SynV also targeting a 500 MHz clock frequency.
- KeyV_{362} : The KeyV processor presented in Section 5.4 with a $(E = 3, S = 6, \alpha = 2)$ KeyRing configuration, synthesized with the DEs lengths defined in Table 5.1.
- KeyV_{661} : The KeyV processor presented in Section 5.4 with a $(E = 6, S = 6, \alpha = 1)$ KeyRing configuration, synthesized with the DEs lengths defined in Table 5.1.

Note that the DEs configurations of Table 5.1 represent the *effective length* of each DE: every DE has a length of 30 DE units, which is then altered with control signals. A given stage has the same configuration across all EUs. The `I` stage represents all the DEs of the Inner-KeyRing and the associated synchronization circuit (see Section 5.4.4).

Table 5.1 KeyV DEs configurations

Core	`F`	`D`	`R`	`E`	`M`	`W`	`I`
KeyV_{362}	8	10	14	16	24	18	30
KeyV_{661}	15	15	15	15	15	15	30

Table 5.2 Summary of KeyV and SynV synthesis results

Core	#clocks	Synthesis time	Area	Cycle time
SynV	2	$\sim$ 5 min	0,44 mm^2	2 ns
SynV$_{cg}$	2	$\sim$ 10 min	0,40 mm^2	2 ns
KeyV$_{362}$	212	$\sim$ 100 h	0,69 mm^2	$\sim$ 3,55 ns
KeyV$_{661}$	534	$\sim$ 150 h	0,98 mm^2	$\sim$ 3,09 ns

Table 5.2 provides a summary of the KeyV and SynV processors synthesis results, which are made explicit in the following sections. First, it shows that the number of clock definitions used to constrain the KeyV processors are excessively more important than those used for the SynV processors. This is expected given that the proposed KeyRing timing constraints heavily rely on clock definitions (see Section 4.4). A detailed analysis of the KeyV processors timing is presented in Section 5.4.5. Similarly, the synthesis run time is also excessively more important for KeyV processors than for SynV processors (note that the reported synthesis time are approximated from multiple synthesis run). This also is clarified in Section 5.4.5. Then, the area reported for each core shows a reduced size of SynV processors compared with KeyV processors, which is expected due to the overhead of replicating datapaths in EUs, and the added area of the KeyRing and the crossbar. Section 5.5.2 provides a deeper analysis based on the area breakdown by modules for each core. Finally, Table 5.2 list the cycle time of each core. In the case of SynV processors, it simply corresponds to the clock period. Note that the 500 MHz clock frequency is not a maximum achievable by the SynV design, but rather a baseline that we thought was adequate. In the case of KeyV processors on the other hand, the cycle time value corresponds to the average delay between EU stages, which provides an information comparable to the clock period in a synchronous circuit (see Section 4.3.2). The detailed clocks information of KeyV processors are given in Section 5.4.5, and a more complete picture of the processors performance, including power efficiency, is drawn in Section 5.5.3. Most importantly, results presented in Section 5.5.3 constitute the evidence that post-synthesis simulations of KeyRing processors netlists, generated by timing-driven synthesis, was successfully executed. As with the Mini-Mips experiment, we use timing simulations to verify the validity of the timing constraints through STA. Although timing simulations cannot exhaustively verify timing violations in the circuit, we used it in the process of improving our timing constraints to uncover the timing violations missed by STA. The more complex the benchmark, the more timing violations can be uncovered. We eventually succeeded in executing Coremark on KeyV$_{661}$, which ran for 2h38 in simulation without any timing violations.

5.5.1 Timing

Figure 5.14 and Figure 5.15 show timing reports from the Static Timing Analysis of the KeyV_{661} processor. Specifically, Figure 5.14 shows the setup analysis (*max path*) of the paths between the $\texttt{D}_0$ launch clock and the $\texttt{R}_0$ capture clock, while Figure 5.15 shows the hold analysis (*min path*) of the paths between the $\texttt{R}_5$ launch clock and the $\texttt{R}_0$ capture clock. These reports should be analyzed in light of the KeyV_{661} ILP representation in Figure 5.9b, in addition to the protocol-level RTCs definitions of Figure 4.9 (setup) and of Figure 4.10 (hold). Note that these reports use the naming convention defined in Chapter 4, which is based on the graph representation of KeyRing systems in Figure 4.7. For example, in the `K_main_02_setup_left_launch` clock name, the `K_main` part stands for *main KeyRing* (it makes the distinction from the inner-KeyRing which has its own set of clocks), the `02` part refers to the EU stage (e, s) indexes (here it corresponds to the $\texttt{R}_0$ stage), the `setup` parts refers to the type of analysis (*max*, *min*) this clock has been defined for, the `left` part stands for the direction in the KeyRing graph from which the paths originates (here it corresponds to the $\texttt{D}_0$ stage), and finally the `launch` part makes the distinction between launch and capture clocks (here, the launch clock is defined on the $\texttt{D}_0$ KU while the capture clock is defined on the $\texttt{R}_0$ KU). In accordance with the standard definitions of setup and hold slacks in timing analysis (see Relations 2.5), the first part of these timing reports details the computation of the *data arrival time* from delays in the launch path, while the second part details the computation of the *data required time* from delays in the capture path. The reports are organized in four columns: the `Point` column represents pins in the netlist, the `Inst` column represents the instances to which the pins belong, the `Incr` column is the delay contributed by each timing path, and the `Path` column contains the cumulative delay of the path formed by the succession of timing paths. The `r/f` label is an information about the unateness (rising/falling) of the associated timing arc. Note that these reports have been slightly altered to improve their readability. For example, some lines providing redundant information were removed and their respective delays were added to the following line. This is notably the case for the Delay Element, where we only show the input and output pins instead of each DE unit as in the original report.

The organization of these setup and hold timing reports demonstrates the effectiveness of the proposed timing constraints based on the KeyRing microarchitecture and its associated timing model. Indeed, there are multiple aspects of these timing reports that have a direct correspondence with the KeyRing timing model developed in Chapter 4. The fact that setup and hold KeyRing RTCs are accurately translated in the timing reports produced by a standard EDA tool, as illustrated by Figure 5.14 and Figure 5.15, contributes to demonstrate the

validity of the proposed timing-driven design flow. The other contribution to this demonstration is the successful execution of post-synthesis timing simulations of KeyV processors, as will be discussed in the following.

The setup timing report corresponds to the setup RTC defined in Relation (4.7) applied to the $\mathtt{R}_0$ stage, while the hold timing report corresponds to the hold RTC defined in Relation (4.10) also applied to the $\mathtt{R}_0$ stage. In the setup (resp. hold) report, the *pod* is defined for both the launch and the capture clock on the clock pin of the $K_{e,s-1}$ (resp. $K_{e-1,s}$) KU, which here corresponds to the KU of the $\mathtt{D}_0$ (resp. $\mathtt{R}_5$) stage (`keyring/ku_01`, and `keyring/ku_52` respectively). Notice that both the launch and the capture timing paths start from the same startpoint at the same time (`0.00 ns`). Because timing constraints are essentially controlling the races between these paths, this is of primary importance. It is not self-evident either: it results from the proper definition of clocks, generated clocks, and multicycle-paths constraints (see Algorithm 4). There is a notable exception to this rule for the case of inter-clock hold violations (*i.e.* multi-pods timing paths, see Section 4.4.1). For those cases, we added a margin to ensure the correct behavior of the circuit by shifting the start of the capture timing paths by 0,5 ns. The launch and capture paths in the setup report can be followed on Figure 5.9b, Figure 5.9, and Figure 4.9. Starting from the $\mathtt{D}_0$ KU, the launch path reaches a register at the `D` stage of EU_0 (`eu_0/D_decode_reg`), which drives the data input of a register in the RF (`rf/fwd_a_reg/D`) through the crossbar (`xbs`). Concurrently, the capture path starts from the $\mathtt{D}_0$ KU, then reaches the $\mathtt{D}_0$ DE (`keyring/de_01`), which triggers the clock input of the same register in the RF (`rf/fwd_a_reg/CP`) through the $\mathtt{R}_0$ KU (`keyring/ku_02`). Similarly, the launch and capture paths in the hold report can be followed on Figure 5.9b, Figure 5.9, and Figure 4.10. Starting from the $\mathtt{R}_5$ KU, the launch path reaches the $\mathtt{E}_5$ KU (`keyring/ku_53`) through the $\mathtt{R}_5$ DE (`keyring/de_52`), which triggers a register in the ALU (`alu/alu_res_reg`) driving the data input of a register of the RF (`rf/regfile_reg[25][31]/D`) through EU_5 and the crossbar. Concurrently, the capture path starts from the $\mathtt{R}_5$ KU, then reaches the $\mathtt{R}_0$ KU (`keyring/ku_02`) through the $\mathtt{R}_5$ DE (`keyring/de_52`), which triggers the clock input of the same register in the RF (`rf/regfile_reg[25][31]/CP`).

For the KeyV_{661} microarchitecture there are 334 similar timing reports, reflecting all the setup and hold timing constraints associated with each stage of each EU, as well as the Inner-KeyRing and the synchronization circuit. Also note that a similar analysis is also applicable to the KeyV_{362} microarchitecture, for which there are 164 of such timing reports. Note that the number of timing reports is related to the number of clock definitions presented in Table 5.2: each report is based on a launch and a capture clock, with some exceptions (*e.g.* the Inner-KeyRing synchronization circuit). For each of these reports, the same correspon-

dence can be established between launch and capture paths and KeyRing RTCs. Not only these correspondences imply that standard timing reports correctly reflect the timing races involved in the circuit, thus enabling to safely rely on STA to perform the timing verification of KeyV processors, they also imply that timing-driven synthesis optimization algorithms appropriately optimize the circuits such that, for each path group, the data arrival time does not exceed the data required time (*i.e.* ensure a positive slack). Since they are not related to user defined values, the KeyRing constraints always reflect the actual races involved in the circuit accurately (to simplify: combinational logic *vs.* Delay Element), in particular during the synthesis timing optimization steps. The first claim (enabling timing verification by STA) is backed by the successful execution of post-synthesis timing simulations of KeyV processors (see Section 5.5.3), while the second claim (enabling timing-driven synthesis) is backed by evidences in the setup and hold timing reports of Figure 5.14 and Figure 5.15. Indeed, since the DE and the KUs logic are fixed by `dont_touch` directives, they cannot be altered by the synthesis engine. Instead, the synthesis alters the combinational logic delay to meet the timing constraints, similarly to what is achieved from a user-defined clock period constraint in a synchronous circuit. For example, the setup report shows that the `rf/fwd_a_reg` register uses a flip-flop cell with a drive of 4 (`EDFCNQD4`), which is faster than its basic alternative (`EDFCNQD0`). This cell has been automatically sized by the synthesis engine in order to meet the setup constraints. Another example in the setup report shows two instances of the `DEL0` cell. This *delay cell* is typically used to fix hold violations. It is automatically inserted in the capture paths of hold analysis to artificially increase the data required time in order to meet the hold condition. Unfortunately, capture paths of hold analysis may also be launch paths of setup analysis, and so the insertion of such artificial delays may hurt setup conditions. The synthesis tool finds a solution satisfying all the constraints. Given the number of clock definitions (see Table 5.2), and given that multiple of them overlap to constrain a given datapath—there are at least two competing setup and hold constraints for each EU stage, and up to five times more for shared resources (see Section 4.4.1)—the synthesis timing optimizer has difficulty finding such a solution. But it does, eventually. This explains the unreasonable amount of time (150+ hours) required to perform a single synthesis run of KeyV processors, whereas the SynV processor—which is of comparable complexity—is synthesized in a matter of minutes. Although the proposed design flow does enable timing-driven synthesis of KeyRing circuits, it is at the cost of a huge increase in synthesis time, resulting from the fact that the synthesis tool is *overconstrained*.

Figure 5.16 summarizes the setup and hold timing reports for the KeyV_{362} and the KeyV_{661} microarchitectures. A bar represents the required time of the timing path associated with a clock group. It is composed of two parts: the first part is the data arrival time and the

remaining part is the slack. Since each stage has two setup and two hold RTCs, we chose to represent only the worst case for each stage for better clarity. Clocks having a required time of 0 (no bar) represent the cases where there are no timing paths to be analyzed. In particular, notice that `D` stages in KeyV_{661} have no hold timing paths, which results from a design decision that prevents stalls from creating WAR hazards (see Section 5.4.2). First, the data required time of each clock (the length of each bar) should be put in perspective with the DEs length of Table 5.1. In KeyV_{362}, we used varying sizes for the DEs, with the intent to get the most out of each stage. In KeyV_{661}, by contrast, DEs are all configured identically. As discussed in Section 4.4.1, this choice in KeyV_{661} has been made to limit inter-clock hold violations, which facilitate timing optimizations (an issue that does not exists in KeyV_{362}). Indeed, observe that hold constraints are more tight than setup constraints. It gives an indication of the difficulty, for the synthesis tool, to fix hold violations. In practice, about 70 % of the synthesis run time is spent in the hold-fixing phase. Then, we can observe in Figure 5.16 a regularity in the setup and hold required time of a given stage across EUs. This is due to the fact that, for a given stage, DEs have the same configuration across EUs as shown in Table 5.1. Notice that there is a delay difference, for a given EU stage, between the setup required time and the hold required time. For example, in KeyV_{661}, the setup required time is on average 3,09 ns, while the hold required time is on average 3,24 ns. Although the paths—including the DEs—involved in the calculation are the same, the worst case scenario is not the same for setup and for hold. In addition, rising and falling edges do have the same propagation delays in most cells. We found that the best solution to limit the complexity of timing optimizations was to use cells having similar rising and falling delays in DEs, hence we mainly used clock tree buffers (*e.g.* `CKBD0`). Finally, the average setup required time for all clocks—a value that corresponds to the average delay between two neighboring EUs stages, and that constitutes a representative measure of the synthesis effort, similar to the clock period in a synchronous circuit—is of 3,55 ns for KeyV_{362} and of 3,09 ns for KeyV_{661}. Although these results have been difficult to obtain, they may certainly be improved further. However, given the time of each synthesis run—some of which took more than 200 h—they are the best that we could achieve.

Timing Report

```
****************************************
Design     : KeyV661
Library    : tcbn65gplustc_ccs (NCCOM)
Path Group : K_main_02_setup_left_capture
Path Type  : max
****************************************
```

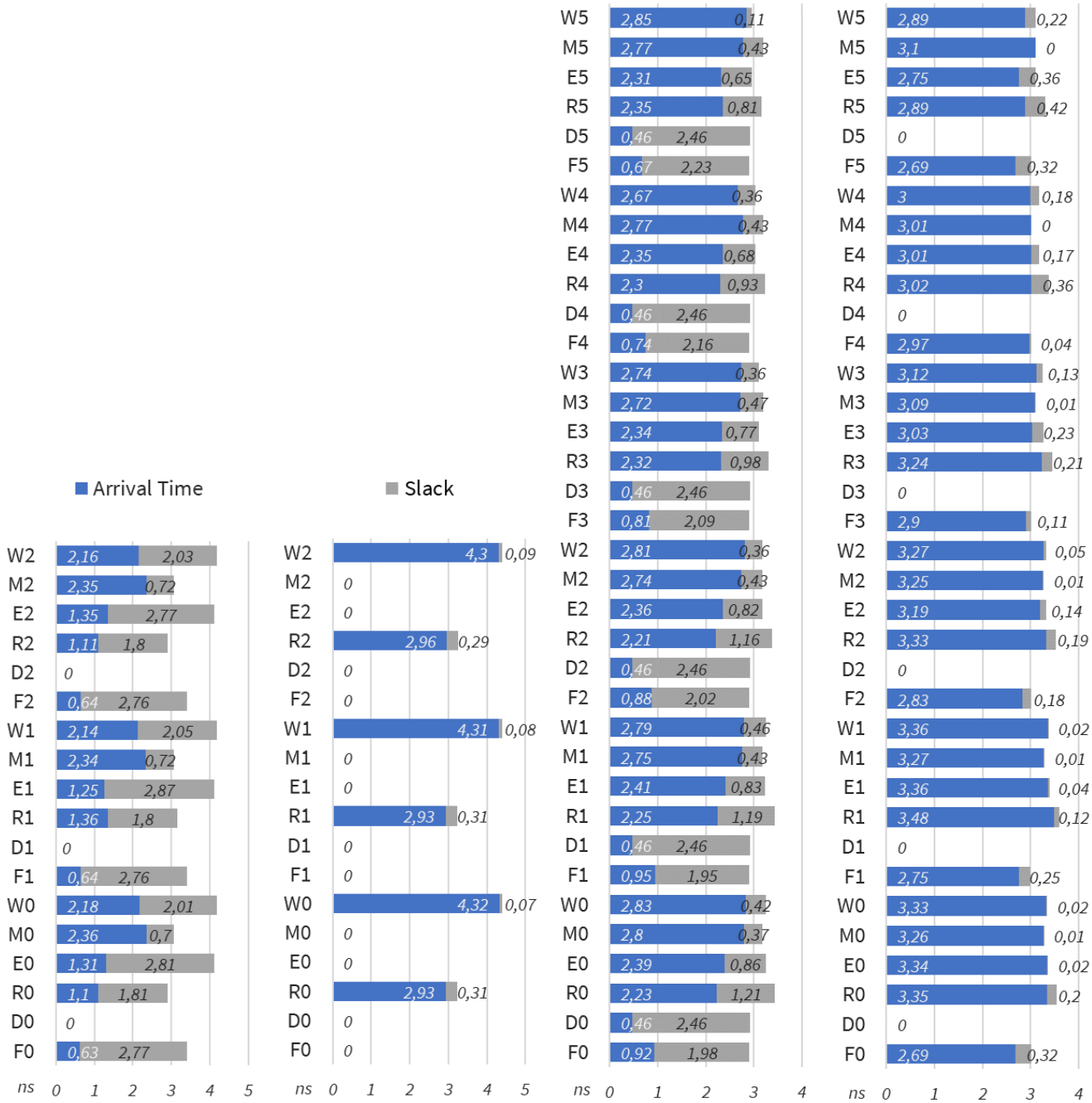
Point	Inst	Incr	Path	
clock K_main_02_setup_left_launch (rise edge)		0.00	0.00	
keyring/ku_01/clkb/Z	*CKBD0*	0.00	0.00	r
keyring/ku_01/o_ku[CLK]	*ku_E0_S1*	0.00	0.00	r
keyring/o_keyring[CLKS][0][1]	*keyring*	0.07	0.07	r
eu_0/i_clks[1]	*eu_0*	0.12	0.19	r
eu_0/D_decode_reg[RS1_ADDR][4]/CP	*DFCNQD1*	0.00	0.19	r
eu_0/D_decode_reg[RS1_ADDR][4]/Q	*DFCNQD1*	0.20	0.39	r
xbs/from_eu[0][TO_RF][ADDR_A][4]	*xbs*	0.00	0.39	r
xbs/U6204/Z	*AO22D0*	0.08	0.47	r
xbs/U7314/Z	*AO211D0*	0.07	0.53	r
xbs/U7313/Z	*DEL0*	0.78	1.31	r
rf/i_rf[ADDR_A][4]	*rf*	0.00	1.31	r
rf/U1953/ZN	*NR2D0*	0.04	1.35	f
rf/U1952/ZN	*IND4D0*	0.04	1.39	r
rf/U1933/Z	*AN4XD1*	0.09	1.49	r
rf/U58/Z	*AN4D0*	0.13	1.61	r
rf/U55/Z	*DEL0*	0.75	2.36	r
rf/fwd_a_reg/D	*EDFCNQD4*	0.00	2.36	r
data arrival time			**2.36**	
clock K_main_02_setup_left_capture (rise edge)		0.00	0.00	
keyring/ku_01/clkb/Z	*CKBD0*	0.00	0.00	r
keyring/ku_01/toggle/Q	*DFCSND1*	0.15	0.15	r
keyring/ku_01/keyb/Z	*CKBD0*	0.06	0.21	r
keyring/ku_01/o_ku[KEY]	*ku_E0_S1*	0.00	0.21	r
keyring/de_01/i_logic	*de_30*	0.00	0.21	r
keyring/de_01/o_logic	*de_30*	2.33	2.54	r
keyring/ku_02/i_ku[KEY_E]	*ku_E0_S2*	0.00	2.54	r
keyring/ku_02/xor_e/Z	*XOR2D0*	0.08	2.62	f
keyring/ku_02/nor/ZN	*NR3D0*	0.05	2.67	r
keyring/ku_02/clkb/Z	*CKBD0*	0.08	2.74	r
keyring/ku_02/o_ku[CLK]	*ku_E0_S2*	0.00	2.74	r
rf/i_clk	*rf*	0.51	3.25	r
rf/fwd_a_reg/CP	*EDFCNQD4*	0.20	3.45	r
inter-clock uncertainty		-0.10	3.35	
library setup time		-0.10	3.25	
data required time			**3.25**	
slack (MET) *data required time - data arrival time*			**0.89**	

Figure 5.14 Setup Timing Report

Timing Report

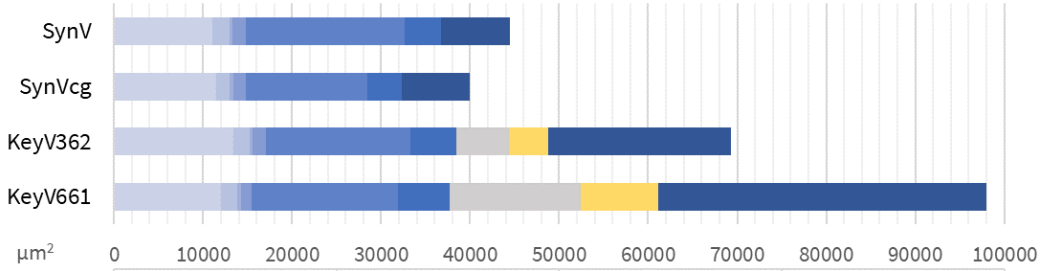
```
****************************************
Design     : KeyV661
Library    : tcbn65gplustc_ccs (NCCOM)
Path Group : K_main_02_hold_right_capture
Path Type  : min
****************************************

  Point                                  Inst              Incr       Path
  ------------------------------------------------------------------------------
  clock K_main_02_hold_right_launch (rise edge)          0.00       0.00
  keyring/ku_52/clkb/Z                   CKBD0             0.00       0.00 r
  keyring/ku_52/toggle/Q                 DFCSND1           0.15       0.15 r
  keyring/ku_52/keyb/Z                   CKBD0             0.06       0.21 r
  keyring/ku_52/o_ku[KEY]                ku_E5_S2          0.00       0.21 r
  keyring/de_52/i_logic                  de_30             0.00       0.21 r
  keyring/de_52/o_logic                  de_30             2.33       2.54 r
  keyring/ku_53/i_ku[KEY_E]              ku_E5_S3          0.00       2.54 r
  keyring/ku_53/xor_e/Z                  XOR2D0            0.08       2.61 f
  keyring/ku_53/nor/ZN                   NR3D0             0.05       2.66 r
  keyring/ku_53/clkb/Z                   CKBD0             0.08       2.74 r
  keyring/ku_53/o_ku[CLK]                ku_E5_S3          0.00       2.74 r
  alu/i_clk                              alu               0.14       2.88 r
  alu/alu_res_reg[31]/CP                 DFCNQD1           0.09       2.97 r
  alu/alu_res_reg[31]/Q                  DFCNQD1           0.17       3.14 r
  alu/o_alu[PORT_Z][31]                  alu               0.08       3.21 r
  eu_5/i_eu[FROM_ALU][PORT_Z][31]        eu_5              0.00       3.21 r
  eu_5/U466/ZN                           AOI222D0          0.06       3.27 f
  eu_5/U464/ZN                           ND2D1             0.08       3.35 r
  eu_5/o_eu[TO_RF][DATA_W][31]           eu_5              0.00       3.35 r
  xbs/from_eu[5][TO_RF][DATA_W][31]      xbs               0.09       3.44 r
  rf/i_rf[DATA_W][31]                    rf                0.00       3.52 r
  rf/regfile_reg[25][31]/D               EDFCNQD1          0.16       3.60 r
  data arrival time                                                   3.60
  ------------------------------------------------------------------------------
  clock K_main_02_hold_right_capture (rise edge)         0.00       0.00
  keyring/ku_52/clkb/Z                   CKBD0             0.00       0.00 r
  keyring/ku_52/toggle/Q                 DFCSND1           0.18       0.18 f
  keyring/ku_52/keyb/Z                   CKBD0             0.06       0.23 f
  keyring/ku_52/o_ku[KEY]                ku_E5_S2          0.00       0.23 f
  keyring/de_52/i_logic                  de_30             0.00       0.23 f
  keyring/de_52/o_logic                  de_30             2.40       2.63 f
  keyring/ku_02/i_ku[KEY_S]              ku_E0_S2          0.00       2.63 f
  keyring/ku_02/xor_s/Z                  XOR2D0            0.07       2.70 f
  keyring/ku_02/nor/ZN                   NR3D0             0.06       2.76 r
  keyring/ku_02/clkb/Z                   CKBD0             0.08       2.84 r
  keyring/ku_02/o_ku[CLK]                ku_E0_S2          0.00       2.84 r
  rf/i_clk                               rf                0.51       3.35 r
  rf/rf_reg[25][31]/CP                   EDFCNQD1          0.21       3.56 r
  inter-clock uncertainty                                  0.10       3.66
  library hold time                                       -0.07       3.59
  data required time                                                  3.59
  ------------------------------------------------------------------------------
  slack (MET) data required time - data arrival time                  0.01
```

Figure 5.15 Hold Timing Report



(a) KeyV$_{362}$ (setup) (b) KeyV$_{362}$ (hold) (c) KeyV$_{661}$ (setup) (d) KeyV$_{661}$ (hold)

Figure 5.16 KeyV Static Timing Analysis summary

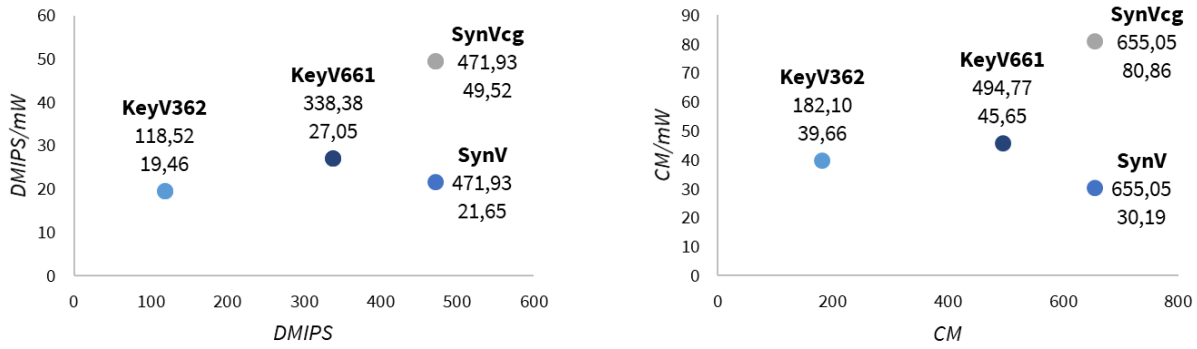


	KeyV661	KeyV362	SynVcg	SynV
alu	12019	13449	11522	10993
decode	1855	1778	1474	1954
lsu	363	363	386	386
pc	1233	1423	1463	1443
rf	16460	16322	13630	17859
sys	5817	5121	3837	4112
xbs	14711	5871	0	0
keyring	8655	4457	0	0
EUs/pipeline	36933	20484	7685	7686
Total	98046	69268	39997	44433

Figure 5.17 Area comparison of KeyV and SynV

5.5.2 Area

Figure 5.17 compares the area, broken down by modules, of KeyV and SynV processors. As they are evaluated post synthesis, these results only include the area of the cells. The shared modules—ALU, Decode, LSU, PC, RF, and SYS—, and the pipeline (SynV) and the EUs (KeyV) are depicted in a blue shading, while the modules that are unique to KeyV—the crossbar (XBS) and the KeyRing—are respectively depicted in gray and yellow. As expected, KeyV_{661} occupies the most space, followed by KeyV_{362}. There is a clear area overhead of KeyV processors due to the replicated logic in EUs, which occupy 4.8× (KeyV_{661}) and 2.6× (KeyV_{362}) more space than the SynV pipeline. There is also an added area due to the crossbar and the KeyRing. Interestingly, SynV_{cg} is smaller than SynV. Although clock-gating cells add some area to most modules, the clock-gating process in DC also performs optimizations that have resulted in a 1.3× decrease of the RF area. Because the modules are shared by all four processors (see Section 5.3), they globally occupy the same area which facilitates both the area and the power consumption comparisons. Yet, some modules occupy more space in KeyV than in SynV, most notably the ALU (*e.g.* the ALU in KeyV_{362} is 1.2× bigger than in SynV). One possible explanation may be the added area of hold-fixing cells insertion (*e.g.* `DEL0`), and the use of bigger cells in the adders to satisfy more constraining setup conditions.



(a) dhrystone (b) coremark

Figure 5.18 Performance and power efficiency comparison of KeyV and SynV

5.5.3 Performance

Figure 5.18 compares the performances and the power efficiency of the KeyV and SynV processors. The x axis represents the benchmark scores obtained from the post-synthesis simulation of the Dhrystone (Figure 5.18a) and Coremark (Figure 5.18b) benchmarks, and the y axis represents the score *per* mW, obtained from the power analysis of the processors based on the activity generated while running the benchmarks in timing simulations.

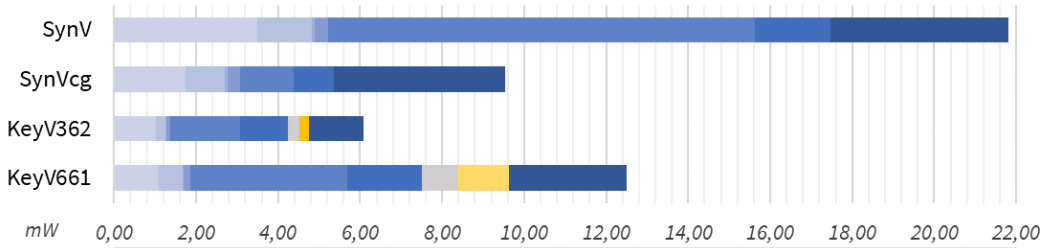
First and foremost, we must emphasize that the ability to obtain these results is a more important achievement than the relative performances of KeyV and SynV processors (they are discussed in the following nevertheless). Indeed, many obstacles had to be overcome to successfully run timing simulations of the KeyV processors netlists produced by standard timing-driven synthesis, most of which have been extensively discussed in this thesis. The main challenge comes down to performing timing simulation of KeyRing circuits that are consistent with Static Timing Analysis. In the process of designing the KeyV processors, and their associated timing constraints, a timing violation in post-synthesis simulation could be interpreted in two ways: either *i)* the timing constraints are incomplete, as a result the synthesis tool do not optimize the path responsible for the violation and the STA do not report it (the timing analysis can only be as good as the timing constraints); or *ii)* there are inconsistencies between the design and the assumptions made in STA, which result in timing violations occurring on paths that are supposed to be properly constrained. In the first case, the issue can be fixed by creating a new constraint, or by modifying the definition of an existing constraint, such that the path responsible for the violation is properly constrained. This was the most common source of errors. New errors of this kind would typically appear when the DEs were reduced in size to improve the processors performance.

Given the synthesis run time, we added configurable registers controlling each selection bit of the DEs, accessible from the top-level, that allowed us to configure the size of the DEs in the test-bench rather than using constants. This way, a faster incremental synthesis (with a proper case analysis) could be used to meet the timing constraints with DEs having a reduced size. Note that, since these registers occupy a lot of space and consume a lot of power, they have been subtracted from the area and power consumption calculations as they are not intended to be used in the definitive designs. In the second case, the only way to resolve the timing violations is to modify the design, and to restart the synthesis from scratch. A typical example of such design-induced timing violations is the stalling issue discussed in Section 5.4.2. More often than not, modifying the design to solve an issue of this kind would create a new timing violation of the first kind. To complicate things further, timing simulations could also be the source of errors, due to a combination of improper SDF generation, improper compilation of the ASIC timing libraries, and negative timing checks issue in Modelsim (see Section 3.8.2). In short, additional difficulties came from a deficit of trust in the results provided by the EDA tools that are supposed to assist in the design process.

Performance figures reported in Figure 5.18 show that KeyV_{362} is 4× less performing than SynV, while KeyV_{661} is only 1.3× less performing than SynV. The reduced performances of KeyV_{362} are mainly attributable to its level of parallelism, which, as discussed in Section 5.4.1, is half that of KeyV_{661} and SynV. The performances of KeyV_{661} are mainly limited by the DEs sizes reported on Table 5.1 (this is also applicable to KeyV_{362}). Indeed, as discussed in Section 5.4.5, KeyV_{661} has only reached an average stage delay of 3.09 ns, compared with the 2 ns stage delay of SynV. Very tight hold constraints—including artificial hold margins to address inter-clock hold violations—and extremely long synthesis runs were the two main factors that prevented us from further improving the performances of KeyV_{661}.

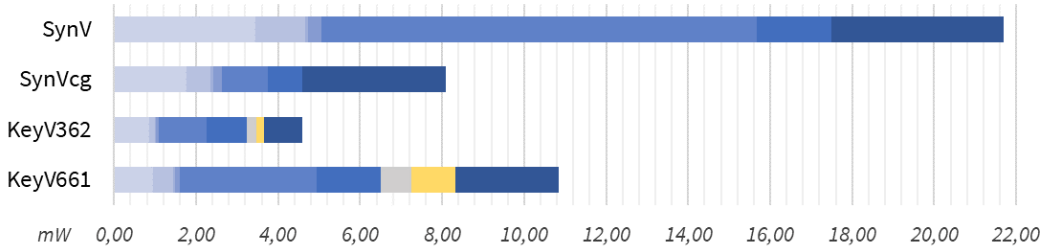
Figure 5.19 shows the power consumption of the KeyV and SynV processors, evaluated from the activity recorded during the execution of the Dhrystone (Figure 5.19a) and Coremark (Figure 5.19b) benchmarks, broken down by modules (we used the same representation as with the area in Figure 5.17). First of all, note that these power evaluations, unfortunately, do not account for the clock-tree. As this work only deals with synthesis, more precise evaluations resulting from the Clock Tree Synthesis and the Place and Route operations were not collected. PrimeTime does have a feature to estimate the power consumption of the clock-tree post-synthesis, which relies on the estimated activity of each clock defined in the circuit. This works fine with SynV, but generates unrealistic results with KeyV. Indeed, since we use a lot more clock definitions than there are clock lines in KeyV, their activity are summed by PrimeTime, which results in an overestimated clock-tree power consumption. Hence, we chose not to use this feature in our power evaluations, for both SynV and KeyV.

SynV is the most power hungry of the group, followed by KeyV_{661}, SynV_{cg}, and KeyV_{362}. The low power consumption of KeyV_{362} is mainly attributable to the reduced activity due to its poor performances. Indeed, in terms of power efficiency (see Figure 5.5.3), KeyV_{362} performs 1.1× worse than SynV with Dhrystone, and 1.3× better than SynV with Coremark. KeyV_{661}, with its improved performances, provides better results: it has a power efficiency that surpasses SynV by a factor of 1.25× with Dhrystone, and 1.5× with Coremark. The most power efficient core of the group, by far, is SynV_{cg}. With Dhrystone (Coremark), it is 2.29× (2.68×) more power efficient than SynV, and 1.83× (1.77×) more power efficient than KeyV_{661}. When looking at the details of Figure 5.19, it appears that the RF, the pipeline/EUs, and the ALU are the main sources of power consumption. In particular, the RF consumes a huge amount of power in SynV, and, interestingly, it is the main source of power reduction in KeyV and in SynV_{cg}. In KeyV, this reduced power consumption (9.05× for KeyV_{362} and 3.18× for KeyV_{661}) can be explained by a reduced activity of the RF due to *i)* a lower average frequency of operation due to reduced performances, and *ii)* a more controlled use of the clocks, with, for instance, the withholding of the `R` clock during stalls. In SynV_{cg}, the 9.46× reduction of the RF power consumption can be explained by the modifications brought to the RF by the clock-gating optimizations. Indeed, the netlist of SynV_{cg} shows that each register in the RF is controlled by a dedicated clock-gating cell. As a result, only the register selected by the address bus is actually clocked during a read or write operation in the RF, and thus dynamic power is only consumed by this register. By contrast, in every other microarchitectures (SynV or KeyV), the 32 registers of the RF are clocked—and dynamic power is consumed—for any read or write operation. Another interesting comparison can be drawn between the EUs in KeyV and the pipeline in SynV. Regardless of the area overhead induced by the replication of resources in the EUs compared with the pipeline (see Figure 5.17), EUs actually consume less power than the SynV pipeline, even clock-gated. As for the RF analysis, part of this reduced power consumption can be attributed to the reduced performances of KeyV, and part can be attributed to the reduced number of clock pulses released by the KeyRing. This analysis also works well with the ALU, in which the registers in the mul/div FSMs in KeyV are only activated when the inner-KeyRing is in use. In addition, the DE used in the inner-KeyRing is voluntarily long, such that the mul/div FSMs can be synthesized with slower, lower power, gates. The resulting performance degradation is balanced by the fact that mul/div operations in the benchmarks are rare. Finally, there are added power consumed by the crossbar and the KeyRing in KeyV, which do not find their equivalent in SynV. Although KeyV_{661} is not as power efficient as SynV_{cg}, its superiority over SynV suggests that this work is a step in the right direction. Future research may provide improved results, as discussed in Section 6.3.



	KeyV661	KeyV362	SynVcg	SynV
alu	1,09	1,02	1,74	3,49
decode	0,58	0,24	0,97	1,34
lsu	0,03	0,03	0,06	0,06
pc	0,15	0,08	0,29	0,34
rf	3,84	1,71	1,32	10,40
sys	1,81	1,15	0,98	1,84
xbs	0,88	0,29	0,00	0,00
keyring	1,24	0,24	0,00	0,00
EUs/pipeline	2,89	1,33	4,16	4,33
Total	12,51	6,09	9,53	21,80

(a) Dhrystone



	KeyV661	KeyV362	SynVcg	SynV
alu	0,94	0,85	1,77	3,44
decode	0,50	0,15	0,58	1,22
lsu	0,03	0,02	0,07	0,06
pc	0,13	0,07	0,21	0,34
rf	3,33	1,17	1,12	10,60
sys	1,57	0,99	0,84	1,82
xbs	0,76	0,22	0,00	0,00
keyring	1,08	0,19	0,00	0,00
EUs/pipeline	2,51	0,95	3,52	4,22
Total	10,84	4,59	8,10	21,70

(b) Coremark

Figure 5.19 Comparison of KeyV and SynV power consumption broken down by modules

CHAPTER 6 CONCLUSION

6.1 Summary

In this work, we have proposed the KeyRing asynchronous microarchitecture, and a framework for the architectural exploration of KeyRing circuits using standard EDA flows. The KeyRing microarchitecture is derived from the Octasic asynchronous design style, as implemented in the AnARM processor, which uses an ad hoc asynchronous clocking scheme to decrease the circuit activity and reduce its dynamic power consumption. However, the Octasic-style asynchronous microarchitecture is mostly incompatible with standard EDA flows, which restricts the design process, due to limited support for synthesis and timing verification. Addressing these limitations is complicated by the unorthodox design principles used in this microarchitecture, which do not fit well within existing asynchronous design paradigms, and prevent the adoption of innovations reported in the asynchronous literature. In this thesis, we laid the foundation for a rigorous definition of this design style, and we proposed state-of-the-art circuits and methods that make it compatible with industry-standard timing-driven synthesis and Static Timing Analysis.

In Chapter 1, we first presented the synchronous and asynchronous design paradigms, and the importance of their integration with EDA flows, which provided a context for the introduction of the Octasic-style asynchronous microarchitecture. We proposed an analysis of the AnARM, which showed that it is among the most power-efficient of its category, and justified our interest in its study. Despite the initial efforts put into building the Octasic design style [35] from standard ASIC cells, we detailed why it remains, for the most part, incompatible with standard EDA flows, in particular with timing-driven synthesis and Static Timing Analysis (we have seen in Chapter 3 that it presents some simulation issues as well). These fundamental limitations, combined with the lack of rigorous definition of this design style in the literature, were presented as the main subjects discussed in this dissertation.

In Chapter 2, we first drew a map of the most significant asynchronous design styles found in the literature, which allowed us to clearly establish that the Octasic design style [35] is out of the scope of standard asynchronous paradigms. Nevertheless, we found similarities with some asynchronous templates, built with EDA compatibility in mind—namely Mousetrap and Click Elements—, that we exploited in the KeyRing microarchitecture in Chapter 4. We then proposed an in-depth analysis of the Octasic-style asynchronous microarchitecture, as implemented in the AnARM. The most important aspects of the microarchitecture—multicycle Execution Units, token rings, and shared *vs.* replicated resources—were discussed to un-

cover the underlying principles of the Octasic asynchronous design style. In particular, we highlighted the unique methods employed to exploit Instruction Level Parallelism, both with an in-order and an out-of-order mode of operation. Finally, we presented state-of-the-art methods for the integration of asynchronous circuits within standard EDA flows. We focused on timing, as it is the most difficult aspect to address. Specifically, we detailed the Relative Timing Constraint formalism, and the Local Clock Set methodology, both of which were adapted to the case of the KeyRing microarchitecture in Chapter 4.

In Chapter 3, we presented the Mini-Mips experiment. For this preliminary work we used an empirical approach to design a simple in-order processor with the Octasic-style asynchronous microarchitecture, and to implement it on FPGA. The Mini-Mips was the support of our first timing-driven implementation methodology, targeting Xilinx Vivado. Combined with FPGA-specific design considerations, we detailed how this method—dedicated implementation flow combined with advanced timing constraints—enabled, for the first time, the implementation of an Octasic-style asynchronous processor on FPGA from a high-level description. We proposed an experimental protocol that enabled the monitoring of runtime performance and power consumption of the Mini-Mips while running a simple benchmark. Results compared with a classic 5-stage pipeline synchronous alternative were inconclusive, which suggested that an ASIC environment would be a more appropriate medium to analyze the power efficiency of asynchronous circuits. Moreover, we showed that the Mini-Mips and its associated implementation flow present shortcomings related to the lack of generality of the design, and incomplete timing definitions combined with impractical timing constraints. Nevertheless, designing the Mini-Mips allowed us to deepen our understanding of the Octasic design style [35], when applied to an in-order single issue scalar processor, as well as widen our comprehension of standard EDA environments and their native incompatibilities with asynchronous specifications. The KeyRing microarchitecture in Chapter 4, and the KeyV processors in Chapter 5, have benefited from the experience gained with this first experiment.

In Chapter 4, we proposed the KeyRing microarchitecture as a generic adaptation of the Octasic-style asynchronous microarchitecture to the case of in-order sequential systems. The main modifications were contributed to the self-timed clock management system, which was renamed KeyRing. The resulting circuit was shown to be more robust than the original, and more compatible with standard timing engines, while bridging the gap with the asynchronous paradigm. This new microarchitecture enabled us to build an abstract model of KeyRing circuits based on a graph, from which we derived generic properties and relations that link performances and ILP with microarchitectural parameters (*i.e.* number of EUs (E), number of stages per EU (S), and stage shift (α)). This abstracted representation of KeyRing circuits is also the backbone of our timing model. We proposed an adaptation of the RTC formalism

to the case of the KeyRing microarchitecture, which allowed us to rigorously define setup and hold timing conditions in KeyRing circuits. These complete timing definitions were then advantageously translated into practical timing constraints, using the SDC format, that are fully compatible with a standard synthesis flow: it benefits from standard timing-driven synthesis with Synopsys Design Compiler, and from standard Static Timing Analysis with PrimeTime. We also demonstrated that inter-clock hold conditions can only be properly constrained with margins, which limits the effective performances of the design. KeyRing design guidelines were showcased on an example circuit (Tribonacci), and the timing model, and its associated timing constraints, were implemented in a generic Tcl library.

In Chapter 5, we presented the design of the KeyV processors, implementing the RV32IM variant of the RISC-V ISA with the KeyRing microarchitecture, and their automatic synthesis using our proposed timing-driven design methodology. We first presented the software ecosystem, which supports the compilation of standard benchmarks—namely Dhrystone and Coremark—for the KeyV processors. We then presented the EDA environment supporting our experimental protocol based on the TSMC65GP 65 nm ASIC design kit, which provides reliable post-synthesis performances and power consumption evaluation of the processors while running the benchmarks. We developed two synchronous alternatives to KeyV based on a 6-stage pipeline—SynV (without clock-gating) and SynV_{cg} (with clock-gating)—that we used as baseline for performances and power consumption comparisons. We proposed two KeyV microarchitectures: KeyV_{362} ($E = 3, S = 6, \alpha = 2$), and KeyV_{661} ($E = 6, S = 6, \alpha = 1$). We showed that KeyV_{362} is more robust to timing violations (especially hold violations) at the cost of important performance degradations, while KeyV_{661} provides theoretical performances as high as SynV, but relies on more complex timing constraints. We then added stalling capabilities to KeyV, and we proposed architectural solutions to the WAR hazards that it induces. We also proposed two methods—one based on a shared-latch synchronizer, and the other based on an Inner-KeyRing—to operate synchronous FSMs and synchronize their result with KeyV. Timing constraints derived from the KeyRing methodology were used to perform the timing driven synthesis and STA of KeyV processors. The combination of setup and hold timing reports analysis with the successful execution of complex benchmarks in post-synthesis timing simulations demonstrated the validity of our proposed design methodology. Post-synthesis results first showed that synthesis runtime of KeyV processors is abnormally long, suggesting that they are overconstrained. Area reports clearly showed an overhead due to logic replication in EUs. Finally, power-efficiency figures showed that KeyV_{661} is superior to KeyV_{362} and SynV, but inferior to SynV_{cg}, which suggests that synchronous microarchitectures with clock-gating are still the better choice for in-order processors.

6.2 Limitations

The limitations of this work can be sorted into two categories: first, limitations of the KeyRing microarchitecture, and related circuits, as well as the limitations of the KeyRing timing model; second, limitations of the timing-driven synthesis flow, including the limitations of the timing constraints.

The first limitation of the KeyRing microarchitecture is its restricted applicability to in-order systems. Indeed, as already discussed in Section 3.8.1, we chose to focus on in-order systems first as they are the foundations on which out-of-order systems are built. Moreover, it allowed us to depart from the original Octasic design style [35], thus avoiding the pitfall of duplicating their design without understanding important design trade-offs. However, the Octasic-style asynchronous microarchitecture was originally intended for out-of-order operations. Although this work can be extended to support out-of-order systems, as will be discussed in the following section, we think that the in-order mode of operation is the main factor limiting the power efficiency of KeyV processors. In other words, the lack of support for out-of-order systems is a limitation to this work as it prevents showcasing better performances and power consumption figures. Furthermore, we showed in Section 4.4.1 that in-order KeyRing circuits cannot be fully constrained for inter-clock hold violations. Instead, we used a margin to permit the correct behavior of the circuit post-synthesis. However, this timing violation can only be partially verified in STA, weakening the reliability of the circuit. In addition, the added margin has an impact on the maximum performance of the circuit, and significantly slows the synthesis runtime.

Another limitation of the KeyRing microarchitecture is its limited support for stalls. Stalls were only introduced for the requirements of the KeyV processors, but they are the source of several issues. Since stall flags come from the datapath, they introduce serious risks of glitches to the KeyRing, making the design unreliable. In fact, the current design would certainly fail due to glitches occurring on the stall lines. In addition, we have seen in Section 5.4.2 that stalls cause microarchitectural hazards in KeyV_{661} (WAR hazards due to the lengthening of hold violations), causing additional restrictions to the datapaths organization. Stalls are also not included in the current KeyRing model. But since stall signals directly control the release of keys and clocks in KUs, they should be part of the model, such that the microarchitectural hazards they induce is formalized, and more generic solutions can be proposed.

The timing-aware implementation methodology of KeyRing circuits also suffers from some limitations. The first limitation is that it is only compatible with Synopsys tools. As mentioned, early experiment conducted with Cadence and Xilinx tools failed. At the time of

writing, and to the best of our knowledge, these limitations are due to the unsuitable timing engines provided with these tools, which is out of our control. Another limitation of the design flow is that, in its current form, it only supports synthesis. Indeed, the lack of back-end flow, in particular the part pertaining to the distribution of the clocks (Clock Tree Synthesis), is an important deficiency of this work. Although CTS and Place and Route steps were not performed in this work, mainly due to a lack of time, we think that the timing model that we proposed for the front-end will be compatible with the back-end. CTS should therefore be a priority for future works, which may lead to interesting results. Finally, another limitation of the synthesis flow is the exceptionally long runtime. As discussed in Section 5.5.3, it is likely due to the complex timing constraints that are required for KeyRing systems leading to an overconstrained design. It is expected that similar issues affect the timing-driven Place and Route steps.

To conclude, it seems clear that synchronous circuits are simpler and more versatile than KeyRing circuits, due to less complex design principles and a native support by standard EDA tools. A similar observation may be formulated for BD asynchronous circuits. Albeit relying on design principles that are arguably as complex as those of KeyRing, and being as poorly supported by standard EDA tools, BD asynchronous circuits offer unmatched modularity. Unfortunately, current power-efficiency results obtained with KeyV do not restore the balance in favor of KeyRing. Nevertheless, we strongly believe that future research should be conducted in the direction of improving the current KeyRing model, and extending it to the case of out-of-order operations, where better results are expected.

6.3 Future Research

"Ad-hoc approaches to clocking are just unfortunate leftovers from the early days of digital design that are incompatible with the requirements of VLSI. Instead, strive to do with as few clock domains as possible and strictly adhere to one synchronous clocking discipline within each such domain"—Hubert Kaeslin, Top-Down Digital VLSI Design [1].

This quotation from a renowned VLSI design book seems all the wiser after reading 144 pages of this dissertation, and suggests that future research on this topic would be ill-advised. Yet, if ad hoc approaches to clocking can contribute to the quest for extreme power efficiency, we think that they are worth exploring. New methods, such as those proposed in this thesis, would then be researched to make these ad hoc microarchitectures more compatible with the requirements of VLSI. Although the power efficiency results of the KeyRing microarchitecture were shown to be inferior to clock-gating, we think that they were good enough to justify

future research. Indeed, to be fair, clock-gating is the result of years of research, which is now globally adopted and integrated within industrial EDA tools. In addition, and as previously mentioned, comparisons with in-order processors may be in favor of synchronous implementations, as the Octasic design style [35] at the origin of the KeyRing microarchitecture was primarily developed for out-of-order systems.

Future research should first focus on consolidating the results obtained with KeyV after the Clock Tree Synthesis and the Place and Route implementation steps. Although we think the proposed timing constraints are compatible with the back-end, new challenges specific to the implementation flow will certainly appear. Then, issues introduced by the stalling mechanism should be addressed, starting with the risks of glitches. In addition, improvements to the KeyRing model that reckon with the introduction of stalls would lead to better performance predictions, and better timing constraints. Finally, the most important future research to conduct is the creation of an out-of-order KeyRing microarchitecture. We think that this thesis provides the foundations that can support such future research. It would consist in designing new KeyRing organizations enabling out-of-order operations, proposing modifications to the KeyRing model that reflect these new organizations, updating the relations that link performances with microarchitectural parameters, defining new timing relations and adapting them to practical timing constraints, and solving any new issues that would certainly manifest themselves in the process.

REFERENCES

[1] H. Kaeslin, *Top-Down Digital VLSI Design: From Architectures to Gate-Level Circuits and FPGAs*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[2] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2014, pp. 10–14.

[3] Y. Patt, "Requirements, bottlenecks, and good fortune: Agents for microprocessor evolution," *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1553–1559, Nov. 2001.

[4] "CPU DB - Looking At 40 Years of Processor Improvements | A complete database of processors for researchers and hobbyists alike." [Online]. Available: http://cpudb.stanford.edu/

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, Oct. 2011.

[6] S. M. Nowick and M. Singh, "High-Performance Asynchronous Pipelines: An Overview," *IEEE Design Test of Computers*, vol. 28, no. 5, pp. 8–22, Sep. 2011.

[7] S. M. Nowick and M. Singh, "Asynchronous Design—Part 1: Overview and Recent Advances," *IEEE Design Test*, vol. 32, no. 3, pp. 5–18, Jun. 2015.

[8] R. Zimmermann and W. Fichtner, "Low-power logic styles: CMOS versus pass-transistor logic," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 7, pp. 1079–1090, Jul. 1997.

[9] T. Skotnicki, C. Fenouillet-Beranger, C. Gallon, F. Boeuf, S. Monfray, F. Payet, A. Pouydebasque, M. Szczap, A. Farcy, F. Arnaud, S. Clerc, M. Sellier, A. Cathignol, J. Schoellkopf, E. Perea, R. Ferrant, and H. Mingam, "Innovative Materials, Devices, and CMOS Technologies for Low-Power Mobile Multimedia," *IEEE Transactions on Electron Devices*, vol. 55, no. 1, pp. 96–130, Jan. 2008.

[10] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[11] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beignè, F. Clermidy, P. Flatresse, and L. Benini, "Energy-Efficient Near-Threshold Parallel Computing: The PULPv2 Cluster," *IEEE Micro*, vol. 37, no. 5, pp. 20–31, Sep. 2017.

[12] S. Krolikoski, "Evolution of EDA standards worldwide," *IEEE Design Test of Computers*, vol. 28, no. 1, pp. 72–75, Jan. 2011.

[13] C. Isci, A. Buyuktosunoglu, C.-y. Cher, P. Bose, and M. Martonosi, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec. 2006, pp. 347–358.

[14] E. Beigne, P. Vivet, Y. Thonnart, J.-F. Christmann, and F. Clermidy, "Asynchronous Circuit Designs for the Internet of Everything: A Methodology for Ultralow-Power Circuits with GALS Architecture," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 4, pp. 39–47, 2016.

[15] S. M. Nowick and M. Singh, "Asynchronous Design—Part 2: Systems and Methodologies," *IEEE Design Test*, vol. 32, no. 3, pp. 19–28, Jun. 2015.

[16] P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*. Springer Science & Business Media, May 2007.

[17] R. Ginosar, "Fourteen ways to fool your synchronizer," in *Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings.*, May 2003, pp. 89–96.

[18] V. G. Oklobdzija, V. M. Stojanovic, D. M. Markovic, and N. M. Nedovic, *Digital System Clocking: High-Performance and Low-Power Aspects*. John Wiley & Sons, Mar. 2005.

[19] P. E. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, and R. L. Allmon, "High-performance microprocessor design," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 5, pp. 676–686, May 1998.

[20] A. Yakovlev, P. Vivet, and M. Renaudin, "Advances in asynchronous logic: From principles to GALS amp;amp; NoC, recent industry applications, and commercial CAD tools," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2013, pp. 1715–1724.

[21] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1437–1455, Oct. 2009.

[22] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and Tak Kwan Lee, "The design of an asynchronous MIPS R3000 microprocessor," in *Proceedings Seventeenth Conference on Advanced Research in VLSI*, Sep. 1997, pp. 164–181.

[23] J. Woods, P. Day, S. Furber, J. Garside, N. Paver, and S. Temple, "AMULET1: An asynchronous ARM microprocessor," *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 385–398, Apr. 1997.

[24] K. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Koi, C. Dike, and M. Roncken, "An asynchronous instruction length decoder," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 2, pp. 217–228, Feb. 2001.

[25] 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 (DATE), 2017*, Mar. 2017, pp. 794–799.

[26] J. Sparsø, "Current trends in high-level synthesis of asynchronous circuits," in *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, Dec. 2009, pp. 347–350.

[27] J. Cortadella, M. Galceran-Oms, M. Kishinevsky, and S. S. Sapatnekar, "RTL Synthesis: From Logic Synthesis to Automatic Pipelining," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2061–2075, Nov. 2015.

[28] P. Teehan, M. Greenstreet, and G. Lemieux, "A Survey and Taxonomy of GALS Design Styles," *IEEE Design Test of Computers*, vol. 24, no. 5, pp. 418–428, Sep. 2007.

[29] M. Laurence, "Introduction to Octasic Asynchronous Processor Technology," in *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, May 2012, pp. 113–117.

[30] T. Awad, M. Laurence, M. Filteau, P. Gervais, and D. Morrissey, "Clock signal propagation method for integrated circuits (ICs) and integrated circuit making use of same," US Patent US8 130 019B1, Mar., 2012. [Online]. Available: https://patents.google.com/patent/US8130019B1

[31] T. Awad, M. Laurence, M. Filteau, P. Gervais, and D. Morrissey, "Method for sharing a resource and circuit making use of same," US Patent US8 689 218B1, Apr., 2014. [Online]. Available: https://patents.google.com/patent/US8689218B1

[32] Q. Zhang, Y. Ge, W. Shi, T. Huang, and W. Tong, "Method and apparatus for asynchronous processor with a token ring based parallel processor scheduler," US Patent US20 150 074 680A1, Mar., 2015. [Online]. Available: https://patents.google.com/patent/US20150074680A1

[33] T. Huang, Y. Ge, Q. Zhang, W. Shi, and W. Tong, "Method and apparatus for asynchronous processor pipeline and bypass passing," US Patent US9 846 581B2, Dec., 2017. [Online]. Available: https://patents.google.com/patent/US9846581B2

[34] K. Krewell, "ARM Pairs Cortex-A7 With A15," *Microprocessor Report (MPR), The Linley Group*, Nov. 2011.

[35] M. Fiorentino, C. Thibeault, Y. Savaria, F. Gagnon, T. Awad, D. Morrissey, and M. Laurence, "AnARM: A 28nm Energy Efficient ARM Processor Based on Octasic Asynchronous Technology," in *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2019, pp. 58–59.

[36] L. Gwennap, "Cortex-A35 Extends Low End," *Microprocessor Report (MPR), The Linley Group*, Nov. 2015.

[37] M. Demler, "Cortex-A32 is First 32-bit ARMv8 CPU," *Microprocessor Report (MPR), The Linley Group*, Mar. 2016.

[38] L. C. Trudeau, G. Gagnon, F. Gagnon, C. Thibeault, T. Awad, and D. Morrissey, "A Low-Latency, Energy-Efficient L1 Cache Based on a Self-Timed Pipeline," in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, May 2015, pp. 17–18.

[39] M. Benyoussef, C. Thibeault, and Y. Savaria, "A Prediction Model for Implementing DVS in Single-Rail Bundled-Data Handshake-Free Asynchronous Circuits," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2019, pp. 1–5.

[40] O. A. Hasib, D. Crépeau, T. Awad, A. Dulipovici, Y. Savaria, and C. Thibeault, "Exploiting built-in delay lines for applying launch-on-capture at-speed testing on self-timed circuits," in *2018 IEEE 36th VLSI Test Symposium (VTS)*, Apr. 2018, pp. 1–6.

[41] O. Al-Terkawi Hasib, Y. Savaria, and C. Thibeault, "Multi-PVT-Point Analysis and Comparison of Recent Small-Delay Defect Quality Metrics," *Journal of Electronic Testing*, vol. 35, no. 6, pp. 823–838, Dec. 2019.

[42] O. A.-T. Hasib, Y. Savaria, and C. Thibeault, "Optimization of Small-Delay Defects Test Quality by Clock Speed Selection and Proper Masking Based on the Weighted Slack Percentage," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–13, 2020.

[43] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual Volume I: User-Level ISA," RISC-V, Tech. Rep., May 2017.

[44] "A000073 - OEIS." [Online]. Available: https://oeis.org/A000073

[45] M. Fiorentino, O. Al-Terkawi, Y. Savaria, and C. Thibeault, "Self-timed circuits FPGA implementation flow," in *2015 IEEE 13th International New Circuits and Systems Conference (NEWCAS)*, Jun. 2015, pp. 1–4.

[46] M. Fiorentino, Y. Savaria, C. Thibeault, and P. Gervais, "A practical design method for prototyping self-timed processors using FPGAs," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 1754–1757.

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

[48] J. Sparso and S. Furber, Eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer US, 2001.

[49] D. Messerschmitt, "Synchronization in digital system design," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 8, pp. 1404–1419, Oct. 1990.

[50] S. Hauck, "Asynchronous design methodologies: An overview," *Proceedings of the IEEE*, vol. 83, no. 1, pp. 69–93, Jan. 1995.

[51] C. L. Seitz, "SELF-TIMED VLSI SYSTEMS," *CALTECH CONFERENCE ON VLSI*, p. 11, 1979.

[52] C. Mead and L. Conway, *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[53] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2011.

[54] M. Roncken, S. M. Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland, "Naturalized Communication and Testing," in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, May 2015, pp. 77–84.

[55] E. Friedman, "Clock distribution networks in synchronous digital integrated circuits," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 665–692, May 2001.

[56] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989.

[57] M. Singh and S. M. Nowick, "MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, Jun. 2007.

[58] 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*, May 2010, pp. 3–14.

[59] M. Singh and S. Nowick, "MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines," in *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, Sep. 2001, pp. 9–17.

[60] M. N. Horak, S. M. Nowick, M. Carlberg, and U. Vishkin, "A Low-Overhead Asynchronous Interconnection Network for GALS Chip Multiprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 494–507, Apr. 2011.

[61] W. C. Mallon and A. M. G. Peeters, "Asynchronous pipeline controller," WO Patent WO2 009 066 238A1, May, 2009. [Online]. Available: https://patents.google.com/patent/WO2009066238A1/en

[62] 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)*, May 2019, pp. 9–18.

[63] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018.

[64] R. L. Traylor, "Self-timed data pipeline apparatus using asynchronous stages having toggle flip-flops," US Patent US5 386 585A, Jan., 1995. [Online]. Available: https://patents.google.com/patent/US5386585A/en

[65] B. R. Quinton, M. R. Greenstreet, and S. J. E. Wilton, "Practical Asynchronous Interconnect Network Design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 5, pp. 579–588, May 2008.

[66] M. Shams, J. Ebergen, and M. Elmasry, "Modeling and comparing CMOS implementations of the C-element," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 4, pp. 563–567, Dec. 1998.

[67] K. Maheswaran and V. Akella, "Hazard-Free Implementation of the Self-Timed Cell Set in a Xilinx FPGA," Computer Engineering Research Laboratory; Department of Electrical & Computer Engineering; University of California; Davis CA, Tech. Rep., 1994.

[68] Z. Hajduk, "Simple method of asynchronous circuits implementation in commercial FPGAs," *Integration*, vol. 59, pp. 31–41, Sep. 2017.

[69] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[70] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers," *IEICE TRANSACTIONS on Information and Systems*, vol. E80-D, no. 3, pp. 315–325, Mar. 1997. [Online]. Available: https://search.ieice.org/bin/summary.php?id=e80-d_3_315&category=D&year=1997&lang=E&abst=

[71] D. Edwards and A. Bardsley, "Balsa: An Asynchronous Hardware Synthesis Language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, Jan. 2002.

[72] D. Sokolov, V. Khomenko, A. Yakovlev, and D. Lloyd, "Design and Verification of Speed-Independent Circuits with Arbitration in Workcraft," in *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2018, pp. 30–31.

[73] A. Kondratyev and K. Lwin, "Design of Asynchronous Circuits by Synchronous CAD Tools," *Design Automation Conference (DAC'02)*, p. 4, 2002.

[74] R. Reese, M. Thornton, and C. Traver, "A coarse-grain phased logic CPU," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 788–799, Jul. 2005.

[75] 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, Oct. 2006.

[76] A. Saifhashemi, D. Hand, P. A. Beerel, W. Koven, and H. Wang, "Performance and Area Optimization of a Bundled-Data Intel Processor through Resynthesis," in *2014 20th IEEE International Symposium on Asynchronous Circuits and Systems*, May 2014, pp. 110–111.

[77] A. Smirnov and A. Taubin, "Synthesizing Asynchronous Micropipelines with Design Compiler," *SNUG*, p. 35, 2006.

[78] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC Flow for GHz Asynchronous Designs," *IEEE Design Test of Computers*, vol. 28, no. 5, pp. 36–51, Sep. 2011.

[79] P. D. Ferguson, A. Efthymiou, T. Arslan, and D. Hume, "Optimising Self-Timed FPGA Circuits," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, Sep. 2010, pp. 563–570.

[80] Y. Liu, G. Xie, P. Chen, J. Chen, and Z. Li, "Designing an asynchronous FPGA processor for low-power sensor networks," in *2009 International Symposium on Signals, Circuits and Systems*, Jul. 2009, pp. 1–6.

[81] S. Moore and P. Robinson, "Rapid prototyping of self-timed circuits," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*, Oct. 1998, pp. 360–365.

[82] K. Stevens, Y. Xu, and V. Vij, "Characterization of Asynchronous Templates for Integration into Clocked CAD Flows," in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, May 2009, pp. 151–161.

[83] G. Gimenez, J. Simatic, and L. Fesquet, "From Signal Transition Graphs to Timing Closure: Application to Bundled-Data Circuits," in *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. Hirosaki, Japan: IEEE, May 2019, pp. 86–95.

[84] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. New York: Springer, 2009.

[85] IEEE, "IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process," *IEEE Std 1497-2001*, pp. 1–80, Dec. 2001.

[86] 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)*, May 2018, pp. 110–118.

[87] K. Stevens, R. Ginosar, and S. Rotem, "Relative timing (asynchronous design)," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 1, pp. 129–140, Feb. 2003.

[88] J. V. Manoranjan and K. Stevens, "Qualifying Relative Timing Constraints for Asynchronous Circuits," in *2016 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2016, pp. 91–98.

[89] Synopsys, "Synopsys Timing Constraints and Optimization User Guide," Synopsys, Tech. Rep., 2019.

[90] N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, "A Fully-Automated Desynchronization Flow for Synchronous Circuits," in *2007 44th ACM/IEEE Design Automation Conference*, Jun. 2007, pp. 982–985.

[91] K. S. Stevens, "Relative timing characterization," US Patent US20 150 026 653A1, Jan., 2015. [Online]. Available: https://patents.google.com/patent/US20150026653A1/en?q=stevens&inventor=Kenneth&assignee=Stevens+Kenneth

[92] Y. Xu and K. S. Stevens, "Automatic synthesis of computation interference constraints for relative timing verification," in *2009 IEEE International Conference on Computer Design*, Oct. 2009, pp. 16–22.

[93] M. Gibiluka, M. T. Moreira, and N. L. V. Calazans, "A Bundled-Data Asynchronous Circuit Synthesis Flow Using a Commercial EDA Framework," in *2015 Euromicro Conference on Digital System Design*, Aug. 2015, pp. 79–86.

[94] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, Sep. 2013.

[95] MIPS, "MIPS32 Instruction Set I," MIPS, Tech. Rep., 2019. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00082-2B-MIPS32INT-AFP-06.01.pdf

[96] Xilinx, "Vivado Design Suite User Guide: Using the Vivado IDE," *Xilinx User Guide*, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug893-vivado-ide.pdf

[97] Xilinx, "Vivado Design Suite User Guide: Using Constraints," *Xilinx User Guide*, p. 197, 2019.

[98] Xilinx, "Vivado Design Suite User Guide: Synthesis," *Xilinx User Guide*, p. 293, 2019.

[99] Xilinx, "7 Series FPGA and Zynq-7000 SoC Libraries Guide (UG953)," *Xilinx User Guide*, p. 607, 2018.

[100] Xilinx, "ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC User Guide (UG954)," *Xilinx User Guide*, p. 103, 2019.

[101] "Dhrystone howto - CDOT Wiki." [Online]. Available: https://wiki.cdot.senecacollege.ca/wiki/Dhrystone_howto

[102] "CPU Benchmark – MCU Benchmark – CoreMark – EEMBC Embedded Microprocessor Benchmark Consortium." [Online]. Available: https://www.eembc.org/coremark/

[103] IEEE, "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, Apr. 2006.

[104] M. Graphics, "ModelSim SE User's Manual," *Modelsim User Guide*, p. 1367, 2015.

[105] K. Tang and S. Padubidri, "Diagonal and toroidal mesh networks," *IEEE Transactions on Computers*, vol. 43, no. 7, pp. 815–826, Jul. 1994.

[106] T. H. Cormen, T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction To Algorithms*. MIT Press, 2001.

[107] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*, 1st ed. Strawberry Canyon, 2017.