Received: 1 February 2021 | Revised: 4 April 2021 | Accepted: 5 May 2021

DOI: 10.1049/cdt2.12032

ORIGINAL RESEARCH PAPER

Introducing *KeyRing* self-timed microarchitecture and timing-driven design flow

Mickael Fiorentino[1] | **Claude Thibeault**[2] | **Yvon Savaria**[1]

[1]Department of Electrical Engineering, Polytechnique Montreal, Montreal, Quebec, Canada

[2]Department of Electrical Engineering, Ecole de Technologie Superieure, Montreal, Quebec, Canada

Correspondence

Mickael Fiorentino, Department of Electrical Engineering, Polytechnique Montreal, Montreal, QC H3T 1J4, Canada.
Email: mickael.fiorentino@polymtl.ca

Funding information

Canadian Network for Research and Innovation in Machining Technology, Natural Sciences and Engineering Research Council of Canada

Abstract

A self-timed microarchitecture called *KeyRing* is presented, and a method for implementing KeyRing circuits compatible with a timing-driven electronic design automation (EDA) flow is discussed. The KeyRing microarchitecture is derived from the AnARM, a low-power self-timed ARM processor based on ad hoc design principles. First, the unorthodox design style and circuit structures are revisited. A theoretical model that can support the design of generic circuits and the elaboration of EDA methods is then presented. Also addressed are the compatibility issues between KeyRing circuits and timing-driven EDA flows. The proposed method leverages relative timing constraints to translate the timing relations in a KeyRing circuit into a set of timing constraints that enable timing-driven synthesis and static timing analysis. Finally, two 32-bit RISC-V processors are presented; called KeyV and based on KeyRing microarchitectures, they are synthesized in a 65 nm technology using the proposed EDA flow. Postsynthesis results demonstrate the effectiveness of the design methodology and allow comparisons with a synchronous alternative called SynV. Performance and power consumption evaluations show that KeyV has a power efficiency that lies between SynV with clock-gating and SynV without clock-gating.

1 | INTRODUCTION

The high demand for energy-efficient computing coupled with the current limitations of technology scaling—which no longer provides improved performance at constant power densities—is leading designers to explore new microarchitectures to pull more performance out of a constrained power budget [1]. Classical approaches to improve the energy efficiency of very large-scale integration circuits consist of reducing supply voltages, gaining efficiency through specialization, and minimizing switching activity. Clock-gating methods are widely adopted, as they dramatically reduce the impact of the global clock in synchronous circuits, which constitutes the main source of switching activity. Alternatively, asynchronous circuits benefit from reduced switching activities, as they rely on decentralized self-timed clocking schemes, and may provide more flexible alternatives than their synchronous counterparts in the quest for energy efficiency [2–5]. However, the adoption of asynchronous circuits has been limited by multiple factors [4, 5]: (i) the lack of support for asynchronous designs by standard EDA flows, especially with respect to timing; (ii) the limited compatibility of asynchronous systems with conventional verification and test methods; and (iii) the limited design reuse, which is mainly due to the plethora of asynchronous design styles. This work follows a broader research project about the *AnARM*, an energy-efficient ARM processor based on the Octasic self-timed design style [6–8]. The AnARM was fabricated with a 28 nm technology, as reported in [9]. It first served as a proof of concept to be compared with other general-purpose processors and provided a context for a novel self-timed cache architecture [10], a new model for dynamic voltage scaling [11], and original test methods [12]. We revisit the original Octasic self-timed design style using circuits and methods coming from the asynchronous literature, with the

This work was funded by the Canadian Network for Research and Innovation in Machining Technology, Natural Sciences and Engineering Research Council of Canada (NSERC).

objective of lowering the barrier with timing-driven EDA flows.

Octasic developed a custom self-timed design style for the *Opus* family of digital signal processors (DSPs) [6] with the intent for a low-power design. The commercial success of these DSPs is often cited as an example to illustrate the benefits of using asynchronous circuits [2, 4], and other processors have been developed based on their design principles [13, 14]. However, as reported in [6], this design style is derived from ad hoc solutions that resulted from an empirical development process rather than a research-oriented approach. In the following, we will show that it stands out from the standard asynchronous design paradigm, as it does not rely on elastic channel abstraction [5]. Moreover, we will show that the ad hoc self-timed microarchitecture of the AnARM exploits instruction-level parallelism (ILP) out of order. We here lay the foundations for a more rigorous definition of this unorthodox design style by introducing a theoretical model that allows for balancing design trade-offs and serves as the basis for a timing-driven electronic design automation (EDA) flow. We propose the name *KeyRing* for this self-timed design style. Note that this work focuses on *in-order* KeyRing systems. First, this design decision is aligned with our goal of building basic principles that can support more complex approaches (such as out-of-order operations). Second, it allows us to depart from the original Octasic design style, thus avoiding the pitfall of duplicating their design without understanding important design trade-offs.

When Octasic started to develop their DSPs in 2004, the lack of well-suited EDA flows and the need to design custom cells dedicated to asynchronous circuits (e.g. C-elements), were the main limitations that prevented full adoption of a standard asynchronous paradigm [6]. But since then, the asynchronous design literature has reported many advances with standard EDA flows that have lowered the barrier for leveraging commercial static timing analysis (STA) engines and timing-driven synthesis tools. The most influential research in this area is the click elements template [15], relative timing constraint (RTC) formalism [16–18], and local clock set (LCS) methodology [19, 20]. This work proposes the KeyRing microarchitecture in combination with a design method inspired by this influential research that makes KeyRing circuits compatible with standard timing-driven EDA flows. Specifically, the proposed method allows timing-driven synthesis tools to optimize KeyRing circuits for performance while preventing setup and hold-time violations.

Results reported in Ref. [9] show that the AnARM achieves performance comparable to that of ARM *Cortex A* processors in a smaller power envelope (compared in a 28 nm technology node). Earlier experiments presented in [6] have shown that the Octasic *Opus2* DSP core is more power-efficient than the TI *C64+* synchronous alternative (compared in a 90 nm technology node). However, comparison with complex systems on a chip is always a difficult task when circuit features and design intent vary while being implemented with different EDA tools using various transistor technologies. Hence, this work aims to strengthen the causal relation between the reported KeyRing microarchitecture and reduced power consumption compared with synchronous alternatives. To support this goal, we propose an experimental protocol that maximizes the impact of the microarchitecture on performance and power consumption comparisons between KeyRing processors and synchronous alternatives.

We developed RISC-V processors, called *KeyV*, showcasing the KeyRing microarchitecture and the proposed EDA flow. These processors implement the RV32IM 32-bit variant of the RISC-V instruction set architecture (ISA) [21], and are synthesized with a 65 nm technology from TSMC. In the spirit of drawing up fair comparisons, we have developed synchronous alternatives to KeyV, called *SynV*—one with and one without clock-gating—that reuse most of the modules used in KeyV, implement the same RV32IM ISA, and are synthesized with the same technology using the same EDA tools. Performance and power consumption evaluations are performed postsynthesis using the *CoreMark* benchmark [22].

We begin by taking a new look at the AnARM design style from the perspective of ILP and resource sharing (Section 2). That review highlights how the AnARM clocking mechanism is involved in the arbitration of shared resource access and in ILP implementation, which to the best of our knowledge has never been shown before. We show that shortcomings of the AnARM design style with respect to timing can be alleviated with circuits and methods from the standard asynchronous paradigm. Backed by analysis of the AnARM, we propose the novel KeyRing microarchitecture based on a template inspired by click elements [15] (Section 3). We propose an abstract model of the microarchitecture that reveals key characteristics of KeyRing systems and allows derivation of timing relations using relative timing constraints. We use this model as the foundation of a timing-driven EDA flow compatible with Synopsys tools (Section 4). Finally, we present the case of KeyV (Section 5), a RISC-V processor based on the proposed KeyRing microarchitecture, which is implemented using a 65 nm technology. KeyV is compared with SynV with and without clock-gating in terms of performance, area, power consumption, and synthesis run time to highlight the trade-offs of the KeyRing design style. To summarize, this paper focuses on studying the KeyRing microarchitecture and improving its integration with the conventional timing-driven EDA flow. Our proposed EDA methodology enables unbiased comparison of the KeyV self-timed KeyRing RISC-V processor with its synchronous counterparts, which sets a baseline for future research. Note that optimizations of the KeyV processor for energy efficiency may be possible but are outside the scope of the paper. The specific contributions of this work are as follows:

- Deep analysis of the Octasic ad hoc self-timed design style as implemented in the AnARM
- In-order KeyRing microarchitecture adapted from the original out-of-order Octasic design style and a template inspired by click elements [15]

- A KeyRing microarchitecture model that reveals the timing relations of the self-timed clocking mechanism and allows performance prediction
- A method to perform STA of KeyRing systems based on the timing relations derived from the model, enabling the use of standard EDA flows (i.e. timing-driven synthesis and STA)
- KeyV RISC-V (RV32IM) processors designed with the proposed KeyRing microarchitecture and synthesized with the reported timing-driven EDA flow
- Analysis of KeyRing microarchitecture trade-offs based on comparisons of KeyV with SynV synchronous alternatives in terms of performance, area, power consumption, and synthesis run time

This work is available at https://github.com/mickaelfiorentino/keyv.

2 | THE AnARM PROCESSOR

The AnARM was introduced in Ref. [9], but its underlying design principles were first reported by Laurence in Ref. [6]. Additional details are also presented in several patents [7, 8, 14]. This section summarizes the principles of the AnARM design style from an original point of view that highlights key characteristics not reported in previous works. We first show, in Section 2.1, that the reported design style does not belong to the *elastic* design paradigm [5]. This claim is supported by a discussion on the taxonomy used throughout this paper. We then present the AnARM architecture in Section 2.2, with an emphasis on ILP and resource sharing. It shows that the AnARM is an out-of-order processor that does not rely on pipelining to exploit ILP. This analysis is the basis from which we elaborate on the principles of the KeyRing design style in *Section* 3.

2.1 | Taxonomy

We use the term *self-timed* in a broad sense to refer to any circuits that are not globally synchronous. Specifically, we refer to Seitz's chapter on system timing in Mead and Conway [23], which defines self-timed systems as interconnections of processing elements performing computational steps in sequences and for which the time required to perform a computation is determined locally by the delays imposed by the processing elements and interconnection delays between them.

The most ubiquitous forms of self-timed circuits are *asynchronous elastic circuits*, as formalized in Ref. [5]. Elasticity is a design paradigm in which elastic channels manage timing variations (within the circuit and between the circuit and its environment). An elastic channel synchronizes the transactions between a sender and a receiver using a *handshake protocol*, often implemented with *request* and *acknowledge* signals [24]. Despite having very different implementations—from latency insensitive synchronous templates to delay insensitive templates by way of bundled data (BD) templates—elastic circuits share the characteristics of the elastic channel abstraction: common building blocks (*fork*, *join*, *merge* etc.), properties (such as *composability*), and formal models (e.g. to predict performance) [5, 24, 25].

The AnARM microarchitecture does not belong to the asynchronous elastic design paradigm. Indeed, although the basic processing elements used in the AnARM, as shown in Figure 1, resembles 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 (DEs)—, 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 [6]. Consequently, we define the AnARM microarchitecture as being simply self-timed in accord with the broad definition presented in this section.

2.2 | Microarchitecture

The AnARM is composed of functional modules, called execution units (EU), and shared resources, such as a register file (RF), program counter (PC), load store unit (LSU) etc., as depicted in Figure 2. EUs operate asynchronously; the overlapping of instruction execution in different EUs allows the parallelism in a program sequence to be exploited. EUs and shared resources communicate with one another through the

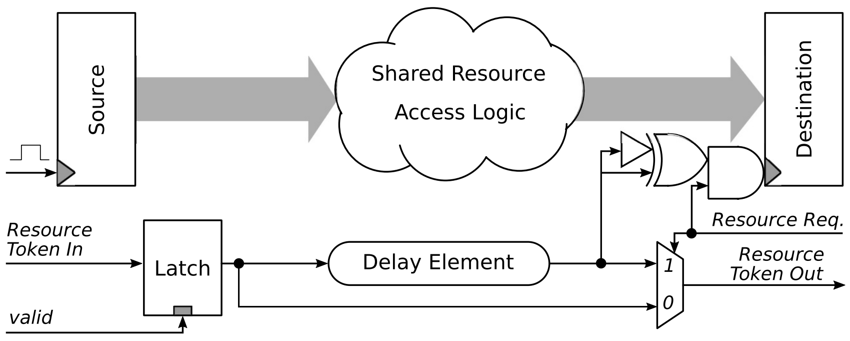

FIGURE 1 Basic processing element used in the AnARM, representing one stage of a *multicycle* execution unit

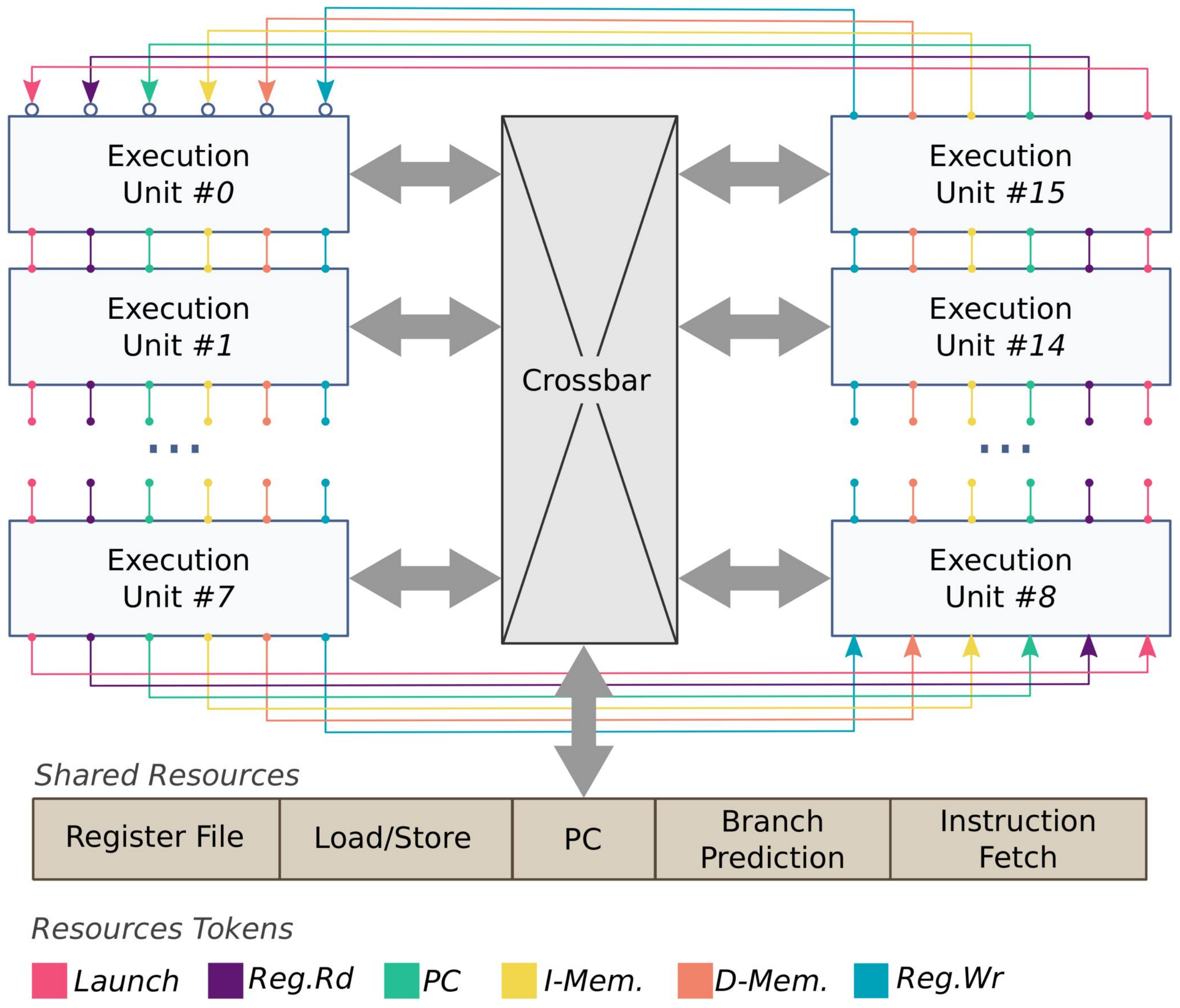

FIGURE 2 AnARM processor overview

crossbar switch (XBS). Note that in contrast with many asynchronous circuits, the XBS in the AnARM does not include any synchronization mechanism.

2.2.1 | Multicycle operations

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. Microarchitectural implementations of ILP usually relies on pipelining, where a new instruction may enter the pipeline as soon as the first stage is available. In 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* of Hennessy & Patterson [26]). Each self-timed stage is a basic processing element, as depicted in Figure 1. 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 single-issue in-order execution, ILP is at its maximum when each EU processes one step of an instruction.

2.2.2 | Resource tokens

Structural hazards that would arise from concurrent access to shared resources by multiple EUs are addressed using *resource tokens*. In Octasic's terminology, tokens are signals associated with shared resources that asynchronously propagate through EUs [6]. In Figure 2, tokens are represented by coloured arrows, and their associated resources are indicated in the legend. The principles of arbitration with tokens in the AnARM are as follows [8]: 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 is completed. If the EU does not request the transaction, the token is directly released. A token travels along EUs in sequence (see Figure 2), thereby granting resource access to one EU at a time. This organization of resource tokens in *rings* serves the dual purpose of granting access to shared resources and synchronizing transactions between a resource and the associated EU stage.

2.2.3 | EU stage

Figure 1 represents typical EU stage architecture in the AnARM and illustrates the synchronization mechanism using resource tokens. A token is being consumed (i.e. resource access is granted) as long as it propagates through the DE matching the delay of the resource access path. The local clock (*capture* clock) generated from the delayed token triggers the destination register. The source register may represent the destination register of a resource or the destination register of another EU stage. It is triggered by a local clock (*launch* clock) generated either by the same token coming from a different EU or by another token coming from a different stage. Because EUs are multicycle, stage$_{i+1}$ accepts data from stage$_i$ without returning an acknowledgement.

2.2.4 | Timing conditions

Let us consider the timing conditions that should be valid for the correct operation of a typical EU stage. When the same token generates both the launch clock and the capture clock (assuming that the latch is transparent), the *setup* condition is the same as with BD circuits [19]: the delay on the token path should exceed the delay on the data path from the source register to the destination register. However, when the launch clock is generated by a token different from the capture clock, the setup condition may vary by an unpredictable amount equal to the arrival time difference between the two tokens. The role of the latch controlled by the *valid* signal is to block the input token until the source register is triggered. It ensures that the setup condition is satisfied as long as the DE delay matches the data path delay. *Hold* conditions found in BD circuits—associated with the backward propagation of control signals through the acknowledgement line—have no direct correspondence here. However, we show in Section 3 that hold conditions exist and explicitly define them.

2.2.5 | Instruction-level parallelism

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

So far, we have described the *in-order* operations of the AnARM based on the arbitration of shared resource access using tokens. But the AnARM computes instructions *out-of-order* using arithmetical and logical units (ALUs) distributed across all EUs. In contrast with shared resources, these *replicated* resources have no need for access arbitration and thus do not require tokens. Replicated resources can be used as soon as the operands they need to operate are ready. The subsequent stages may depend on the results that replicated resources produce, which are handled similarly to other dependent stages using *valid* signals. Replicated resources operated concurrently enable another level of parallelism: the AnARM is an out-of-order processor with in-order instructions issue, out-of-order execution, and in-order completion.

Figure 3 shows how ILP is achieved by allocating instructions steps in EUs stages. A given EU processes the i^{th} instruction modulo E, where E represents the number of EUs. Coloured rectangles represent instruction steps, with their length representing the stage delay and their colour 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

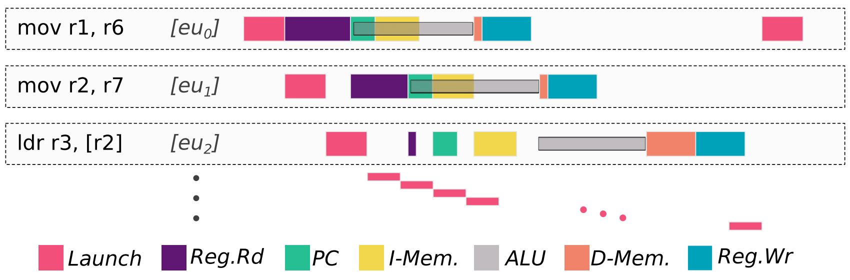

FIGURE 3 Illustration of instruction-level parallelism in the AnARM

bypassed. Horizontally, steps are processed in a sequence that follows the dependency of tokens in an EU (i.e. *Launch* → *Reg.Rd* → *PC* …). Vertically, steps are processed in a sequence that follows the propagation of tokens across EUs (i.e. *Launch*: $EU_0 \rightarrow EU_1 \rightarrow EU_2$ ...). The *Execution* steps, which rely on replicated ALU resources, can be completed out-of-order because they are not associated with a token: the *Execution* step of the i^{th} instruction can finish before the *Execution* step of the $(i-1)^{th}$ instruction is completed. Results are written back to the RF sequentially using the *Reg.Wr* token. Data dependencies between nearby instructions are handled by preventing the *Execution* step in the processing EU from starting before the operands are ready and by forwarding data between all EUs using the XBS as soon as results are available. For example, Figure 3 shows that the third instruction depends on data coming from the previous instruction in the r2 register. The operands come from the XBS, 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 [6–8], 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 different EUs.

3 | KeyRing MICROARCHITECTURE

In this section, we present the KeyRing self-timed microarchitecture derived from the analysis of the AnARM in Section 2. We propose the *KeyRing protocol*, an arrangement of EUs in a two-dimensional toroidal mesh topology, as the elementary organization of KeyRing systems and the *KeyRing template* as an alternative implementation of the basic processing element used in the AnARM (Figure 1). This new circuit template and the formal definition of the KeyRing protocol also ease integration with timing-driven EDA flows (see Section 4). Here, the focus is on the first level of parallelism (in-order), while formalizing, analysing, and implementing the second level of parallelism (out-of-order) is left for future works.

3.1 | KeyRing template

The need to modify the AnARM template (see Figure 1) comes from the following: (i) The tokens and *valid* signals carry redundant information. That is, one or more stages have completed their execution. (ii) The tokens and *valid* signals use different signalling conventions. Tokens use transition signalling, while valid signals use level signalling. (iii) The AnARM template is less compatible with a timing-driven methodology because latches are used on the clock paths.

Figure 4 shows the proposed alternative to the AnARM template based on a Key unit controller (KU). A KU is an adaptation of the click elements circuit [15] that produces a local clock for the registers in the data path and a state signal used to replace both the tokens and the *valid* signals. We call it a *Key* to contrast it with *token* in the AnARM and *request* and *acknowledge* in BD circuits. Similarly to a token, a Key uses transition signalling and is delayed through a DE to match a stage logic delay. Like *valid* signals, Keys are generated by flip-flops and are involved in the control of other Keys. Our proposed processing element (Figure 4) operates as follows: A toggle flip-flop stores a state signal ($K_{e,s}$), and a pair of XOR gates implements the phase conversion that generates the local clock ($C_{e,s}$). 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 begins 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 must be satisfied for correct circuit operation: (i) The inputs must remain stable during the active phase of the clock. This is ensured by construction of the KeyRing protocol. (ii) The clock pulse width must be larger than the minimum pulse width defined in the technology library. This can be enforced by sizing the buffer on the feedback path. Compared with the original Click Elements [15], the main difference comes from adaptation of the phase conversion mechanism inspired by the implementation reported in [25]. This difference accounts for the specificities of the KeyRing protocol. It is worth noting that similar circuits were independently proposed by Traylor in [27] and Quinton in [28].

As for the AnARM, KeyRing systems do not use the channel abstraction of elastic circuits. The KeyRing is an organization of KUs linked together by Keys in rings (hence the name), which orchestrates the generation of self-timed clocks across EU stages. Keys arbitrate the access to shared resources, and their states bound the activity of EU 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.

3.2 | KeyRing protocol

The KeyRing microarchitecture is suitable for implementing any sequential system that would otherwise be implemented with a pipeline. Figure 5a represents a generic KeyRing circuit equivalent to a three-stage pipeline. It is composed of three EUs (columns) of three stages each (rows). $R_{e,s}$ represent the registers of the data path, $K_{e,s}$ are their associated KUs, and $key_{e,s}$ are the Keys. The KeyRing is modelled by the graph of Figure 5b, 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 [29]. Here, we reuse some of the notations and definitions proposed in [29].

Let $G = (U, K)$ be the graph of Figure 5b such that $U \in \xi \times \zeta$ and $K \in \xi^2 \times \zeta^2$, where ξ is a set of E identical EUs, each comprising a set ζ of S stages. G is a toroidal mesh that

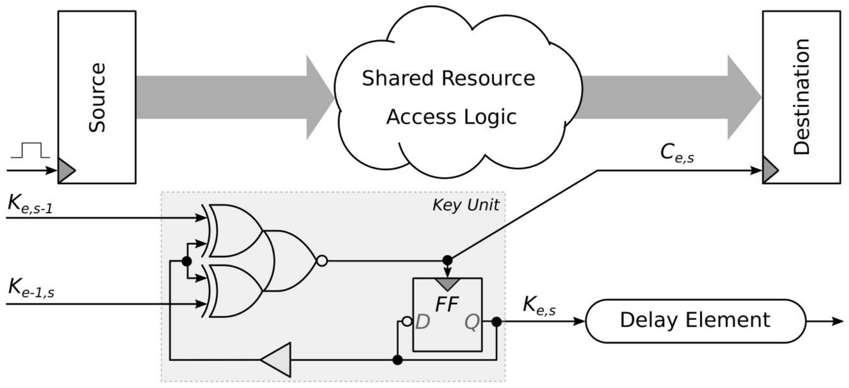

FIGURE 4 Processing element used in KeyRing systems representing one stage of a multicycle execution unit controlled by the proposed Key unit

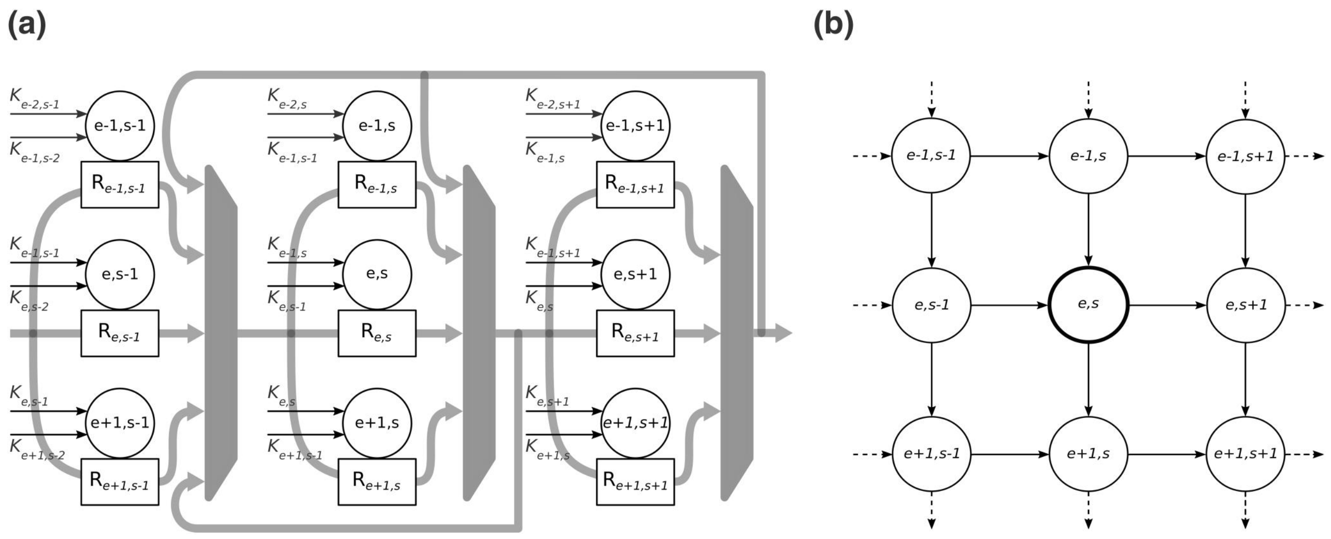

FIGURE 5 A KeyRing circuit (a) is a two-dimensional grid of Key units linked together by Keys controlling a set of registers ($R_{e,s}$). The KeyRing protocol is modelled by a graph (b) having a toroidal mesh topology with wraparound connections at the edges. The represented KeyRing has $E = S = 3$ and $\alpha = 1$

consists of a two-dimensional grid of EUs with wraparound connections at the edges. For any node $(e, s), e \in \{0, \ldots, E-1\}, s \in \{0, \ldots, S-1\}$, connections are defined as

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

where $\langle x\rangle_X$ signifies that x is considered *modulo X*, which allows accounting of the wraparound connections at the edges, where $\alpha \in \{1, \ldots, S\}$ represents the dependency shift between two stages of successive EUs. Note that Figure 5 shows the specific case where $E = 3$, $S = 3$, $\alpha = 1$.

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

Equation (1) defines KeyRing organizations in which computations are performed in order according to Definition 1. A value of $\alpha = 1$ corresponds to the case where a stage (e, s) in a given EU 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 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:

$$F(e,s) = \langle s + \alpha e\rangle_S \tag{2}$$

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), $e \in \{0, \ldots, E-1\}$, and $s \in \{0, \ldots, S-1\}$ 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 in-order operation.

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

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 Equation (2), 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 ($E = 3$, $S = 6$, $\alpha = 2$) has the same level of parallelism as a KeyRing with ($E = 3$, $S = 3$, $\alpha = 2$) despite having twice as many stages per EU. The *optimal* configuration of an in-order KeyRing system is one in which a given level of parallelism is obtained with the minimum number of EUs and stages per EU. It is obtained when $E = S$. However, we will see that KeyRing circuits having $E < S$ need fewer timing constraints to be correctly implemented, which results in reduced synthesis time and more robust circuits. In *Section* 5, we present two *KeyV* processors: the first uses an ($E = 3$, $S = 6$, $\alpha = 2$) KeyRing ($p = 3$), while the other uses an ($E = 6$, $S = 6$, $\alpha = 1$) KeyRing ($p = 6$).

3.3 | Performance

The performance of a sequential circuit is bounded by the level of parallelism and the speed of the computations. In a typical synchronous system, the speed is determined by a global clock—usually generated externally—characterized by a given frequency and uncertainties 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 frequencies and the uncertainties coming from their respective clock trees. Moreover, in a synchronous system, the clock base frequency is independent of 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 (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 EUs. Hence, this metric is independently defined.

Definition 4 The delay Δ_{src}^{dst} between a source KU (src) and destination KU (dst) is the delay separating the occurrence of the destination clock, generated by (dst), from the occurrence of the source clock, generated by (src).

In synchronous pipelines, the delay between two stages separated by multiple stages is simply the sum of each stage delay 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 delay because of the interdependence of KUs, as modelled by the KeyRing graph (Figure 5b). Let δ_{src}^{dst} be the delay between two *adjacent* nodes ((src) and (dst)) in the KeyRing graph, which can be evaluated with STA. This delay includes the delay of the DE and the delays of the clock distribution network, 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 predecessor Keys. To determine the *effective delay* between two clocking events in the KeyRing—that is, the delay between two stages in the KeyRing as per Definition 4—this recursivity should be accounted for.

Algorithm 1: Determine the *effective* delay between a source (*src*) and a destination (*dst*) KU

```
Data: Let $G$ be the graph of Figure 5b, $A[e, s]$ be the forward adjacency lists of
      a node $(e, s) \in G$, $\mu$ and $\nu$ be tables, and $\Gamma$ be a stack
Function EFFECTIVE_DELAY(($src$), ($dst$)) is
  Initialize($\Gamma$, $\mu$, $\nu$)
  push($\Gamma$, ($src$))
  while $\Gamma \neq \phi$ do
    $(e, s) \leftarrow$ pop($\Gamma$)
    $\nu(e, s) \leftarrow \mu(e, s)$
    foreach $(i, j) \in A[e, s]$ do
      $\mu(i, j) \leftarrow \max\left(\mu(i,j),\ \mu(e,s) + \delta_{e,s}^{i,j}\right)$
      if $(i, j)$ has not been reached yet then
        push($\Gamma$, $(i, j)$)
      end
    end
  end
  return $\mu(src) - \nu(dst)$
end
```

To this end, we propose a *longest-path algorithm* (Algorithm 1) that computes the effective delay between two clocking events by finding the longest path between two nodes of the KeyRing graph (Figure 5b). It is derived from the breadth-first search (BFS) and the Dijkstra algorithms [30], which are well-known methods for solving similar problems. Each arc of the graph is weighted by δ_a^b, which has been defined as the latency between two adjacent nodes a and b. The algorithm starts from the source node to traverse 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 [30]. 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 1 can be used to determine the delay Δ_{src}^{dst} between two stages in a KeyRing system by computing the effective delay between a source, (src), and a destination, (dst), KU. Similarly, it can determine the period $T(e, s)$ of a clock by computing the effective delay between two clocking events of the same KUs.

$$\Delta_{src}^{dst} = \text{effective\_delay}((src), (dst)) \tag{3}$$

$$T(e, s) = \text{effective\_delay}((e, s), (e, s)) \tag{4}$$

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

Proof. From Equation (4), $T(e, s)$ is computed with Algorithm 1 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.

4 | TIMING-DRIVEN DESIGN FLOW

This section deals with the integration of KeyRing systems into an automated design flow. The focus is on timing-driven synthesis and STA. Timing-driven synthesis engines can optimize a netlist to meet a performance target, and STA tools can verify whether these targets are met. Trade-offs can be made between area, power, and performance, which improves the resulting design and significantly lowers design time compared with a non-timing-driven flow. However, the integration of self-timed circuits into standard timing-driven EDA flows is challenging. Given the similarities between KeyRing systems and asynchronous BD circuits, as discussed in Section 2.1, we first review existing methods for implementing BD circuits with standard EDA tools (Section 4.1). The methods retained rely on Relative Timing Constraints [16–18,20]. We propose RTCs definition of KeyRing circuits (Section 4.2), from which we derive a set of timing constraints (Section 4.3) that allows standard timing engines to correctly apprehend KeyRing circuits.

4.1 | Background

Asynchronous BD circuits work under timing assumptions similar to synchronous systems (bounded delay model): glitches are allowed in the combinational logic because data transitions are expected to happen strictly before a validity tag is analogous to a clock signal. Thus, they are more compatible with synchronous EDA flows than other asynchronous design styles [5]. Many techniques have been developed to alleviate the various limitations related to the implementation of BD circuits using standard EDA tools.

4.1.1 | Standard cells

A first limitation is related to the use of asynchronous-specific gates. Typically, the C-elements used in the *micropipeline* BD template [31] are not provided as standard cells by silicon vendors. Thus, additional time and expertise are needed to design and characterize these cells. Some asynchronous templates have addressed this issue. Most notably, the mousetrap [32] and click elements [15] templates rely solely on standard components (D-latches, flip-flops, XOR etc.). Likewise, as previously mentioned, the KeyRing template uses only standard components. Compared with the AnARM template, it favours flip-flops over latches to facilitate the definition of timing constraints.

4.1.2 | Logical synthesis

Three main approaches have been explored for the synthesis of asynchronous models [33, 34]: (i) Custom synthesis tools [15, 35] that rely on specific languages (e.g. Balsa, Haste). (ii) *Desynchronization* design flow [36], which starts with the synthesis of a synchronous specification and converts the netlist into an equivalent asynchronous BD circuit (flip-flops are replaced with latches, and the clock tree is replaced with handshake control networks). (iii) Template-based approach [37, 38], which relies on various empirical techniques applied to standard synthesis tools to synthesize hardware descriptions of BD circuits. Control structures (asynchronous templates) are designed at the gate level, and a set of *dont_touch* directives are used to prevent their automatic removal. The first two methods are research areas of their own, and their study is beyond the scope of this paper. This work uses a template-based approach in combination with a standard synthesis tool.

4.1.3 | Timing analysis

The methods presented for the logical synthesis of BD circuits do not guarantee their temporal correctness, nor do they allow timing-driven optimizations. To this end, timing constraints must be derived from the circuit and supported throughout the design flow stages. A suitable set of timing constraints should allow the synthesis tool (i) to prevent hazards due to timing loops and (ii) to meet (with timing-driven optimizations) and verify (with STA) the performance targets of the system. However, modelling the timing of asynchronous BD circuits with standard timing constraints is a major challenge [16–20].

The most promising avenues in this research area rely on RTCs. In a first approach reported in [17, 18], RTCs are translated into *min_delay* and *max_delay* constraints between each sequential stage and respectively applied to the control and data paths. This method allows the use of STA to verify the timing, but it does not benefit from the ability of the synthesis tool to explore the design space with timing-driven optimizations because the constraints must be updated during the flow. Other works have proposed leveraging the concept of *clock* to improve compatibility with standard EDA tools. The work reported in Ref. [39] first proposed to define a pair of non-overlapping *virtual clocks* on the handshake signals of a BD circuit. The works reported in Ref. [37, 40] explore the idea further by defining *generated clocks*, derived from *virtual clocks*, to model the propagation of events in the control path. The resulting timing paths are defined as *zero-cycle paths* to account for the non-periodic characteristics of BD operations. These clock constraints enable some optimizations of the data path and facilitate STA. However, similar to the *min/max_delay* method, they should be updated during the flow to reflect the actual delays of the control paths. The LCS methodology [19, 20] draws the best from these previous works. First, it exhaustively defines the RTCs of a BD template. Then, for each RTC, a set of clock constraints is advantageously defined in a way that preserves the timing relationship between the data path and the control path, allowing the synthesis tool to optimize them concurrently. Our proposed method adapts the LCS methodology to the case of KeyRing circuits.

4.2 | Relative timing constraints of KeyRing circuits

RTCs formally define the timing requirements a circuit must satisfy to operate correctly. They are described by the following relation [18, 20],

$$pod \mapsto poc_{early} \prec poc_{late} - \varepsilon \tag{5}$$

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. *Protocol-level* RTCs define the timing relations between adjacent stages having control path dependencies such as those typically found in BD circuits (*Req* and *Ack*) and KeyRing circuits (*Keys*). More detailed classifications of RTCs can be found in [17, 20]. The timing path between the *pod* and the data pin of a register is the *launch* path, and the timing path between the *pod* and the control pin of a register is the *capture* path. An RTC translates to a *setup* condition if the capture event reaches the *poc* after the launch event, and it translates to a *hold* condition if the capture event reaches the *poc* before the launch event.

Figure 6 shows the generic KeyRing circuit with an overlay representing the protocol-level RTC definitions spanning multiple EU stages. Without a loss of generality, these definitions are shown for a KeyRing having $E = S = 3$ and $\alpha = 1$. The *pods* serve as references for the RTC definitions. In a synchronous circuit, this is usually defined as the main clock, and thus it can be defined implicitly. In a KeyRing (or BD) circuit, by contrast, it varies with each stage and must be explicitly defined. In KeyRing circuits, finding the *pods* is further complicated by the recursive dependencies of the Keys. In practice, a *pod* can be localized as the last common point between the launch and capture paths. That is, a KU at the intersection of a launch path from dependent KUs to the data input of $R_{e,s}$, and a capture path from dependent KUs to the clock input of $R_{e,s}$ is the *pod* of a KeyRing RTC. A specificity of KeyRing circuits, compared with BD circuits, is that each stage (e, s) has two setup RTCs and two hold RTCs. The first RTC comes from the dependent stage (e, s−1), while the second RTC comes from the dependent stage (e−1, s) (see Figure 5b). We denote $C_{e,s}$ and $D_{e,s}$ the clock pin and the data pin (respectively) of register $R_{e,s}$, and $K_{e,s}$ the Key of the (e, s) KU. Moreover, note that Figure 6 denotes launch paths with dashed lines and capture paths with plain lines:

$$\text{setup (a):} \quad C_{e,s-1} \mapsto D_{e,s}^{max} \prec C_{e,s} - \varepsilon \tag{6}$$

$$\text{setup (b):} \quad C_{e-1,s} \mapsto D_{e,s}^{max} \prec C_{e,s} - \varepsilon \tag{7}$$

$$\text{hold (a):} \quad C_{e,s-1} \mapsto C_{e,s} \prec \{C_{e+1,s-1}, D_{e,s}^{min}\} - \varepsilon \tag{8}$$

$$\text{hold (b):} \quad C_{e-1,s} \mapsto C_{e,s} \prec \{C_{e-1,s+1}, D_{e,s}^{min}\} - \varepsilon \tag{9}$$

The setup RTCs are defined by Equations (6) and (7) and shown in Figure 6a with yellow and blue, respectively. The capture path starts at $C_{e,s-1}$ ($C_{e-1,s}$) and reaches $C_{e,s}$ from the DE of $K_{e,s-1}$ ($K_{e-1,s}$) through the shortest path. The launch path starts at $C_{e,s-1}$ ($C_{e-1,s}$) and reaches $R_{e,s}$ from $R_{e,s-1}$ ($R_{e-1,s}$) and the combinational logic through the longest path. The hold RTCs are defined by Equations (8) and (9) and are represented by Figure 6b in yellow and blue, respectively. The capture path starts at $C_{e,s-1}$ ($C_{e-1,s}$) and reaches $C_{e,s}$ from the DE of $K_{e,s-1}$ ($K_{e-1,s}$) through the longest path. The launch path starts at $C_{e,s-1}$ ($C_{e-1,s}$), then reaches $C_{e+1,s-1}$ ($C_{e-,s+1}$) from the DE of $K_{e,s-1}$ ($K_{e-1,s}$), and reaches $R_{e,s}$ from $R_{e+1,s-1}$ ($R_{e-1,s+1}$) and the combinational logic through the shortest path.

Setup RTCs in KeyRing circuits are similar to those in BD circuits [19, 20], but hold RTCs are different. Indeed, the hold conditions in a typical asynchronous BD stage are associated with the feedback path of the acknowledgement signal. The lack of an acknowledgement signal path in a typical multicycle

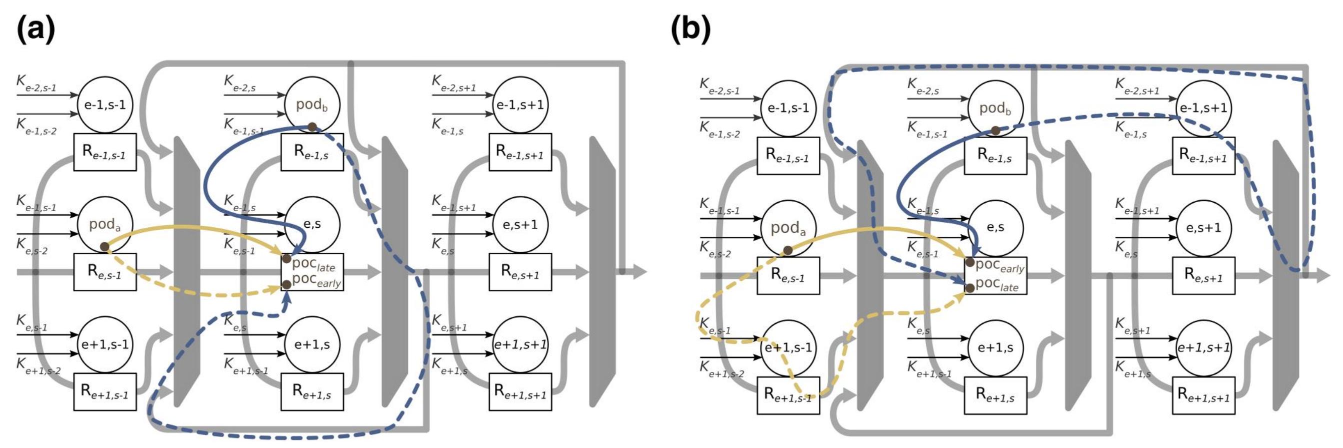

FIGURE 6 Protocol-level relative timing constraint definitions in a KeyRing circuit. Dashed lines represent launch paths, and plain lines represent capture paths

EU may suggest that KeyRing circuits are exempt from hold conditions altogether, as discussed in Section 2. However, hold conditions are revealed when looking at the interaction of multiple EU stages, as shown in Figure 6. In a KeyRing circuit, hold violations happen in 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 $C_{e,s}$.

Lastly, we must consider *multipod* RTCs. Although an RTC is by definition relative to a single *pod*, we call multipod RTCs 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 in KeyRing circuits (see Figure 6) and should thus be constrained appropriately. Multipod 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), which increases the likelihood of satisfying the setup constraint, but works against meeting the hold condition. Thus, additional constraints are only needed for multipod hold timing paths. However, when trying to define new hold RTCs from parent KUs, we found that there will always be a race between pod_a and pod_b, regardless of the stage at which the RTC starts. In practice, as long as the clock definitions used to translate KeyRing RTCs belong to the same clock group, interclock timing paths are automatically constrained by STA. The solution that we used to safely constrain multipod hold RTCs is to add hold margins between interclock definitions.

4.3 | Static timing analysis of KeyRing circuits

For the RTCs defined in the previous section to effectively constrain the design in a timing-driven design flow and to enable setup and hold timing verifications for each EU stage using standard STA reports, they must be translated into a set of timing constraints using the Synopsys Design Constraints (SDC) format that the tools can interpret appropriately. As already mentioned, this task is not trivial [16–20]. This section deals with this translation for the case of KeyRing circuits.

An important contribution of the work reported in Ref. [20] was to show that standard EDA tools already deal with RTCs. Indeed, timing engines use clock constraints defined at every register clock pin to derive setup and hold timing conditions from RTCs defined in the timing characterization file of a standard cell library (usually in *Liberty* format). Protocol-level RTCs are typically not included in Liberty models provided with standard cell libraries because they involve multiple components. In Ref. [34], Liberty models of micropipeline BD stages are proposed to make the tool interpret protocol-level RTCs of the stages as internal RTCs of components, which are exploited using *min/max_delay* timing constraints. However, this approach does not allow concurrent timing-driven optimizations of the data and control paths. Instead of hiding protocol-level RTCs in Liberty models, the LCS methodology [19, 20] defines a set of local clock constraints such that the tool interprets RTCs between adjacent stages as a function of the internal RTCs of their components. The main advantage of using clock constraints (create_clock and create_generated_clock SDC commands) is that they allow standard EDA tools to be used as they were intended, thus improving support for timing-driven optimizations.

Algorithm 2 shows the proposed set of timing constraints that makes KeyRing circuits compatible with a timing-driven design flow. Its underlying principles are as follow:

Algorithm 2: KeyRing timing constraints pseudocode

```
  # Root clocks
1 foreach (e, s) in KeyRing do
2     create clock $c_{e,s}$ on $C_{e,s}$
3     generate clock $k_{e,s}$ from $c_{e,s}$ on $K_{e,s}$
4 end
  # Setup launch & capture clocks
5 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 interclock
  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
```

4.3.1 | Root clocks

The clocks, $c_{e,s}$ and $k_{e,s}$, defined on the clock ($C_{e,s}$) and Key ($K_{e,s}$) pins of each KU are used (i) to break architectural loops that the timing engine finds by travelling along the KeyRing (we take advantage of the fact that a clock constraint definition breaks any timing path crossing its source point); and (ii) to generate the *launch* and the *capture* clocks in the design (we use the ability of generated clock constraints 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 provide support for the rest of the constraints. The architecture (Figure 4) facilitates these constraint definitions.

4.3.2 | 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 data paths. The *setup* clocks translate RTCs (6) and (7), while the *hold* clocks translate RTCs (8) and (9). For each RTC, a *launch* and a *capture* clock are 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 poc_{early}. The *pods* are automatically determined as the last common point between the launch clock and the capture clock paths corresponding to one of the root clocks $c_{e,s}$.

4.3.3 | 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). In addition, we increase the margin between crossed pairs of hold-launch and capture clocks to safely constrain multipod timing paths (see Section 4.2).

4.3.4 | Exceptions

First, root clocks are excluded from the timing analysis with false path definitions (using the set_false_path SDC commands), as they are not intended to constrain the data paths. Then, interclock timing interactions between selected groups of clocks are disabled (using the set_clock_groups asynchronous SDC commands). Every group of four setup (hold) clocks generated for each stage (e, s) is added to the $setup_{e,s}$ ($hold_{e,s}$) group after being defined. Only the interclock interactions between clocks of the 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 (reaching) capture clocks (launch clocks) are defined as false paths because capture clocks (launch clocks) do not launch (capture) data.

4.3.5 | Miscellaneous

The default behaviour of synchronous tools in dealing with clocking events must be altered to be suitable for KeyRing circuits. First, clocks are propagated (using the set_propagated_clock SDC command), which 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 DE delays, KU logic delays 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 must 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 is achieved by defining a *zero-cycle path* between launch and capture clocks (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 adequate use of the control path (including the DE) delay as a constraint for the associated data path.

5 | CASE STUDY: *KeyV* PROCESSORS

This final section presents a use case for the KeyRing microarchitecture and its associated timing-driven design flow. We propose the *KeyV* processors, implementing the RV32IM variant of the RISC-V ISA: KeyV$_{362}$, using an ($E = 3$, $S = 6$, $\alpha = 2$) KeyRing, and KeyV$_{661}$, using an ($E = 6$, $S = 6$, $\alpha = 1$) KeyRing, thereby achieving twice the level of parallelism of KeyV$_{362}$. We chose not to compare KeyV with other RISC-V processors in the literature, as we reckon with the limitations of such comparisons in this context. For instance, memories in KeyV are simply modelled in the test bench, which limits the relevance of performance and energy comparisons. Instead, we have developed *SynV*—our own synchronous alternative to KeyV—to explore design trade-offs. SynV is based on an in-order six-stage pipeline. It reuses most of the modules used in KeyV, implements the same variant of the ISA, and was synthesized using the same technology and EDA tools, with and without clock-gating. We emphasize making a *fair* comparison between the processors to uncover, among the characteristics of the KeyV processors, those that are attributable to the KeyRing microarchitecture. The KeyV and SynV processors are synthesized with a 65 nm ASIC design kit from TSMC, using Synopsys EDA tools (DC and PrimeTime). Their comparisons are based on postsynthesis timing simulations using the *CoreMark* benchmark [22], combined with activity-aware power analysis.

We first discuss the microarchitectures of KeyV (Section 5.1). We then present our experimental protocol along with the design flow adapted to KeyV (Section 5.2). Finally, we validate our EDA methodology using postsynthesis results and compare KeyV and SynV in synthesis time, area, performance, and power consumption (Section 5.3).

5.1 | Microarchitecture

Figure 7 shows the block diagram of the KeyV and the SynV processors. Both the SynV pipeline and the KeyV EUs are composed of six stages—*Fetch* (F), *Decode* (D), *Register Read* (R), *Execute* (E), *Memory* (M), *Register Write* (W)—and rely on the same sequential modules: the PC controls the address bus of the instruction memory (IMEM); the Decode module decodes RV32IM instructions; the RF is a 32×32 bits array of registers; the ALU contains an adder, a shifter, and logical submodules, as well as a multiplier and a divider (*mul/div*) submodule; the LSU is interfaced with the data memory (DMEM); and finally the system (SYS) module is responsible for handling system tasks, including Control Status Register instructions that provide access to the performance counters. As already mentioned, memories are not part of the processors. For simplicity, we used behavioural models with an ideal latency of one cycle.

In KeyV, each resource uses a different clock coming from the KeyRing, while in SynV, the resources use the global clock. In addition to generating self-timed clocks, and orchestrating the stages concurrency across EUs, the KeyRing also generates control signals—*sel* in Figure 7a—to arbitrate the access to shared resources in the XBS. Resources are triggered by different clocks depending on the stage at which they are operated. Clocks coming from the same stage of different EUs are ORed together before going to a resource, as illustrated in Figure 7a. For example, the F clock triggering the IMEM is driven by F_0 or F_1 or … or F_e. Following this reasoning, the PC is clocked by the W clock, the Decode module is clocked by the D clock, the ALU is clocked by the E clock, and the LSU, DMEM, and SYS modules are clocked by the M clock. Note that KeyV also uses a synchronous clock for the performance counters that is synchronized with the M stages.

Figure 8 illustrates the main differences between the KeyV$_{362}$ and the KeyV$_{661}$ microarchitectures. Following the design principles developed in Section 3, these two microarchitectures correspond to two possible KeyRing configurations for a fixed number of stages $S = 6$: three and six EUs. KeyV$_{362}$ comprises three EUs of six stages with a dependency shift between stages $\alpha = 2$, which provides an ILP of 3. Although EUs have six stages, only three of them can be exploited concurrently because of the value of α—M, R, F, or W, E, D—, as shown in Figure 8a. KeyV$_{661}$ comprises six EUs of six 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 six-stage pipeline, six EU stages can be exploited concurrently, as shown in Figure 8b. The reason for considering the KeyV$_{362}$ microarchitecture is its resilience against hold violations. Figure 8 illustrates protocol-level hold RTCs (see Section 4) between D and R stages in KeyV processors, with dashed and plain arrows (respectively launch and capture paths). 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 reaches the RF through the XBS. In KeyV$_{661}$, since R and D stages are concurrent, this hold condition can only be enforced through timing constraints. In KeyV$_{362}$, by contrast, this hold condition is architecturally enforced through KUs dependencies in the KeyRing; thus, it is always met.

In KeyV, microarchitectural hazards are caused by the overlapping of instructions across EUs. First, *structural hazards* that would arise from concurrent access to shared resources by multiple EUs are addressed by the KeyRing. In addition, concurrent access to the RF at the R and W stages by different EUs would cause a structural conflict between their clocks. To solve it, 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 $i + 1$. Then, *data hazards* that would occur when at

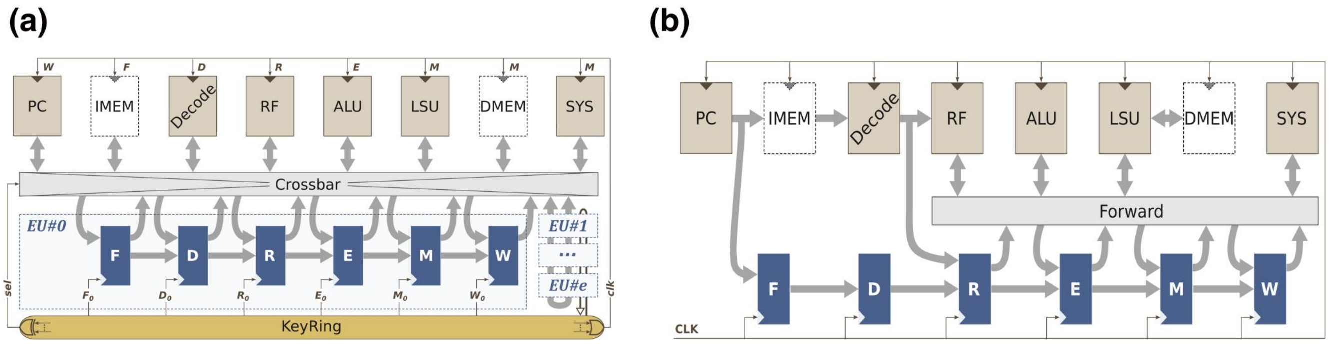

FIGURE 7 Block diagram of the RISC-V (RV32IM) processors compared in this work. *KeyV* uses in-order KeyRing microarchitectures, and *SynV* uses an in-order six-stage pipeline. They both use the same modules and are synthesized using the same technology and EDA tools with a timing-driven design flow

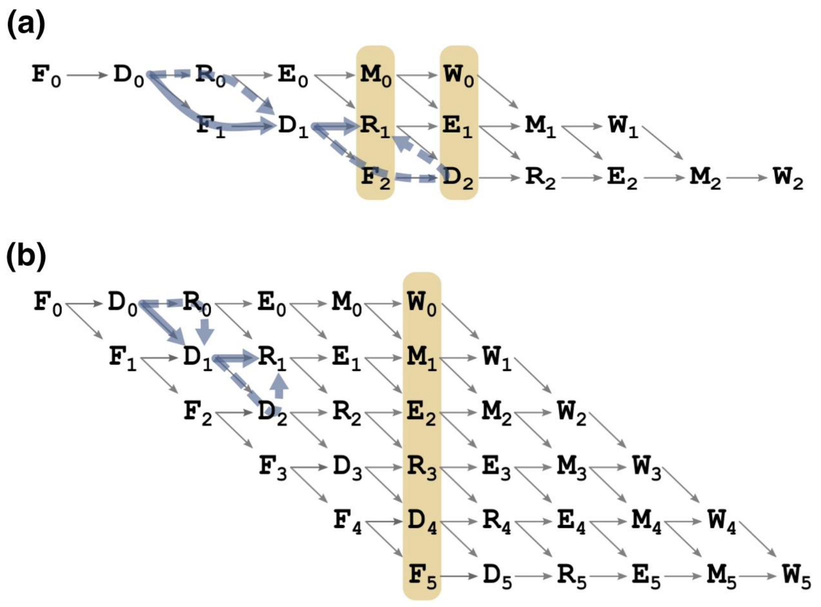

FIGURE 8 Comparison of instruction-level parallelism in KeyV processors

least one operand used by an EU instruction is used as a destination register by another EU's preceding instruction are addressed by the XBS. Finally, *control hazards* that would arise because of branching instructions are handled by flushing out instructions from EUs. Both SynV and KeyV use a simple *predicted-not-taken* branching strategy, where the branch outcome is computed during the E stage and exposed from the M register. In SynV, the four preceding stages are flushed, while in KeyV all other EUs are flushed, and the EU responsible for the branch is also the EU in which the destination instruction is processed.

The multiplication and division operations (part of the ALU) require 32 cycles to complete. In SynV, preceding stages are stalled during the computation of a mul/div instruction, while in KeyV, the E stage is stalled—the E clock is not released—until the operation completes. By means of successive dependencies in the KeyRing, all the clocks are halted until the operation finishes. In KeyV, the mul/div modules are clocked by a dedicated KeyRing that we call the *inner-KeyRing*. It has the same architecture as the main KeyRing, but it is composed of only one KU ($E = 1$, $S = 1$, $\alpha = 1$). It is started after the R stage of a mul/div instruction, stopped after having released 32 clocks, and synchronized with the E stage. The details of the inner-KeyRing are not covered further here.

5.2 | Design flow

The KeyV and SynV processors have been synthesized with Synopsys DC, using the *TSMC65GP* 65 nm ASIC design kit from TSMC. We set the operating conditions to the typical case at TT/1 V/25°C. The timing-driven synthesis is performed in two incremental steps: A first optimization pass is geared towards fixing setup violations, and a second optimization pass tries to fix hold violations. In synchronous designs, hold optimizations are generally performed during the place and route flow to address issues due to the clock tree delays. In KeyRing designs, by contrast, hold optimizations must be performed during synthesis, as DEs and KUs are the backbone of the clock trees. The only differences between the KeyV and the SynV synthesis flows are the timing constraints. The SynV timing constraints are standards targeting a global clock of 500 MHz base frequency, and the KeyV timing constraints are based on Algorithm 2. We have adjusted the size of DEs using incremental synthesis with the objective of maximizing performance. To support the specification of the KeyRing timing constraints, we developed a generic *tcl* library that facilitates the modelling of KeyRing circuits within standard EDA tools. We used this library to create the timing constraints of KeyV_{362} and KeyV_{661}. As opposed to most EDA methods proposed in the literature that need to reload the timing constraints after each implementation step, the proposed KeyRing EDA flow relies on a single definition of the timing constraints, and KeyV achieves similar levels of optimization as SynV.

5.3 | Results and discussion

Table 1 provides a summary of the KeyV (KeyV_{362} and KeyV_{661}) and SynV (SynV and its clock-gated version, SynV_{cg}) synthesis results. First, it shows that the number of clock definitions used to constrain KeyV is much larger than those used for SynV, which is expected (see Section 4). Similarly, the synthesis run time is also much more important for KeyV than for SynV, which is likely due to the complexity of the timing constraints. Then, the area reported for each core shows a reduced size of SynV processors compared with KeyV processors. Because modules are the same, this area overhead is due to replicated data paths in EUs, and the added area of the KeyRing and the XBS.

Figure 9 shows one of the 334 timing reports in KeyV_{661} as a practical example illustrating how the proposed timing constraints are interpreted by the STA tool. The proposed timing constraints are based on the KeyRing microarchitecture and its associated timing model, which is the main contribution of this work. The report is a minimum delay (hold) analysis of the paths between the R_5 launch clock and the R_0 capture clock. Consistent with the RTC definitions represented in Figure 6b, the launch clock (K_main_02_hold_right_launch, corresponding to $hl^{e,s}_{e-1,s}$ in Algorithm 2) activates the path from R_5 ((5, 2) KU), through the clock path reaching E_5 ((5, 3) KU), and then through the data path reaching a register data pin at the R stage clocked by R_0 ((0, 2) KU). Alongside, the capture clock (K_main_02_hold_right_capture, corresponding to $hc^{e,s}_{e-1,s}$ in Algorithm 2) activates the path from R_5 through the clock path reaching the registers at the R stage triggered by R_0. Notice that the synthesis engine has automatically sized the rf/rf_reg register to meet the timing constraints: it uses a flip-flop cell with a drive of 1 (EDFCNQD1), which is faster than the default (EDFCNQD0). Having the setup and hold KeyRing RTCs accurately translated in timing reports, combined with this automated gate sizing, contributes to demonstrating the effectiveness of the proposed timing-driven EDA flow. The other contribution to this demonstration is the successful execution of postsynthesis timing simulations of KeyV processors.

TABLE 1 Summary of KeyV and SynV synthesis results

Core	#clock definitions	Synthesis time	Area (mm^2)
SynV	2	~5 min	0.44
SynV_{cg}	2	~10 min	0.40
KeyV_{362}	212	~100 h	0.69
KeyV_{661}	534	~150 h	0.98

Figure 10 shows postsynthesis results obtained from the performance and the power analysis of the processors based on the execution of the CoreMark benchmark in timing simulations. Figure 10a compares the performance and power efficiency of the KeyV and SynV processors. It shows that KeyV_{362} performance is 4× lower than SynV, while KeyV_{661} has performance only 1.3× lower than that of SynV. The reduced performance of KeyV_{362} is mainly attributable to its level of parallelism, which as discussed in Section 5.1, is half that of KeyV_{661} and SynV. The performance of KeyV_{661} is mainly limited by the DEs sizes. Very tight hold constraints—including artificial hold margins to address interclock hold violations—and extremely long synthesis runs were the two main factors that prevented us from further improving the performance of KeyV_{661}. Figure 10b shows the power consumption of the KeyV and SynV processors broken down by modules. The low power consumption of KeyV_{362} is mainly attributable to the reduced activity due to its poor performance. In terms of power efficiency, KeyV_{362} performs 1.3× better than SynV, and KeyV_{661} surpasses SynV by a factor of 1.5×. SynV_{cg} is 2.68× more power-efficient than SynV, and 1.77× more power-efficient than KeyV_{661}. When looking at the power breakdown, it appears that the RF, the pipeline/EUs, and the ALU are the main sources of power consumption. In particular, the RF is the primary source of power reduction in KeyV and SynV_{cg}. In KeyV, this reduced power consumption (9.05× for KeyV_{362} and 3.18× for KeyV_{661}) is explained by reduced activity of the RF due to (i) a lower average frequency of operation due to reduced performance, and (ii) a more controlled use of the clocks. In SynV_{cg}, there is a 9.46× reduction in the RF power consumption, which is due to the control of each register by a dedicated clock-gating cell. Despite their area overhead, EUs 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 performance of KeyV, and part can be attributed to a more controlled activity of the clocks by the KeyRing. This analysis also applies to the ALU, in which the registers of the mul/div submodules in KeyV are only activated when the inner-KeyRing is in use. Finally, additional power is consumed by the XBS 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 power efficiency is improved over SynV, indicating that this work is a step in the right direction.

```
clock C_1_2_hold_down_launch
keyring/click_1_1/clk                          (CKBD0)              0.00
keyring/de_1_1                                                      5.62
keyring/click_2_1/clk                          (CKBD0)              5.83
xu_2/rs1_addr_reg[0]/CP                        (DFCNQD1)            6.16
[...]
rf/o_ra_reg[31]/D                              (EDFCNQD1)           6.92
data arrival time                                                   6.90
```

```
clock C_1_2_hold_down_capture
keyring/click_1_1/clk                          (CKBD0)              0.00
keyring/de_1_1                                                      5.79
keyring/click_1_2/clk                          (CKBD0)              6.00
rf/o_ra_reg[31]/CP                             (EDFCNQD1)           6.77
inter-clock uncertainty                                             6.87
library hold time                                                   6.91
data required time                                                  6.91
```

```
slack (MET) data required time - data arrival time                  0.01
```

FIGURE 9 Simplified *hold* timing report between R_5 and R_0 in KeyV_{661}

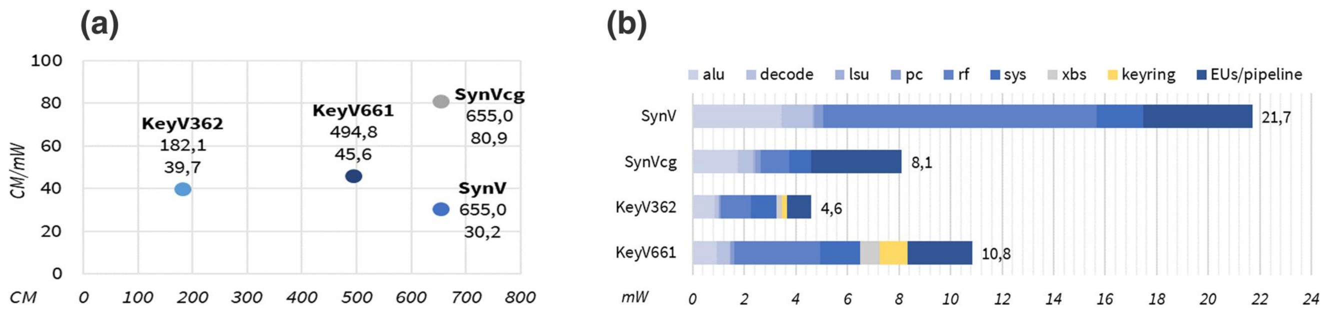

FIGURE 10 KeyV and SynV performance comparison based on postsynthesis timing simulations of the CoreMark benchmark with activity-aware power analysis

6 | CONCLUSION

This paper presented the KeyRing self-timed microarchitecture and a framework for the timing-driven synthesis of KeyRing circuits using standard EDA flows. The KeyRing microarchitecture was derived from the microarchitecture of the AnARM processor as a generic adaptation of its underlying ad hoc asynchronous design style to the case of in-order sequential systems. The main modifications were contributed to the self-timed clock management system, which was shown to be more robust than the original and to provide better compatibility with standard timing engines. An abstract model of KeyRing circuits was proposed to derive relations linking performance and ILP with microarchitectural parameters and support a dedicated timing model. The RTC formalism was used to rigorously define setup and hold timing conditions in KeyRing circuits. These timing definitions were advantageously translated into practical timing constraints fully compatible with a standard synthesis flow (i.e. timing-driven synthesis and STA). The KeyRing microarchitecture and its associated EDA flow were showcased through the design of the RV32IM KeyV processors (KeyV_{362} and KeyV_{661}). KeyV_{362} is more robust to timing violations at the cost of important performance degradations, while KeyV_{661} provides better performance but relies on more complex timing constraints. Two synchronous alternatives to KeyV based on a six-stage pipeline—SynV (without clock-gating) and SynV_{cg} (with clock-gating)—were used as a baseline for performance and power consumption comparisons. The combination of timing reports analysis with the successful execution of the CoreMark benchmark in postsynthesis timing simulations demonstrated the validity of our proposed design methodology. Postsynthesis results first showed that the synthesis time of KeyV processors is extremely long because of the complexity of the timing constraints. 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}.

ACKNOWLEDGEMENTS

The authors would like to thank CMC for providing the necessary CAD tools, and NSERC (Canada) for providing partial funding. This work was funded by the Canadian Network for Research and Innovation in Machining Technology, Natural Sciences and Engineering Research Council of Canada (NSERC).

ORCID

Mickael Fiorentino https://orcid.org/0000-0002-7998-9004

REFERENCES

1. Horowitz, M.: 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), 10–14 (2014)
2. Nowick, S.M., Singh, M.: Asynchronous design—Part 1: Overview and recent advances. IEEE Design Test. 32(3), 5–18 (2015)
3. Nowick, S.M.: Asynchronous design—Part 2: systems and methodologies. IEEE Design Test. 32(3), 19–28 (2015)
4. Yakovlev, A., Vivet, P., Renaudin, M.: 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), 1715–1724 (2013)
5. Carmona, J., et al.: Elastic circuits. IEEE Trans. Comput. Aided Des Integrated Circ. Syst. 28(10), 1437–1455 (2009)
6. Laurence, M.: Introduction to octasic asynchronous processor technology. In: 2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems, 113–117 (2012)
7. Awad, T., et al.: Clock signal propagation method for integrated circuits (ICs) and integrated circuit making use of same. US Patent US8 130 019B1, Mar 2012. https://patents.google.com/patent/US8130019B1
8. Awad, T.: Method for sharing a resource and circuit making use of same. US Patent US8 689 218B1, Apr 2014. https://patents.google.com/patent/US8689218B1
9. Fiorentino, M., et al.: AnARM: a 28nm energy efficient ARM processor based on Octasic asynchronous technology. In: 2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 58–59 (May 2019)
10. Trudeau, L.C., et al.: A low-latency, energy-efficient L1 cache based on a self-timed pipeline. In: 2015 21st IEEE International Symposium on Asynchronous Circuits and Systems, 17–18 (May 2015)
11. Benyoussef, M., Thibeault, C., Savaria, Y.: A prediction model for implementing DVS in single-rail bundled-data handshake-Free asynchronous circuits. 2019 IEEE International Symposium on Circuits and Systems (ISCAS), 1–5 (May 2019)
12. Hasib, O.A.-T., Savaria, Y., Thibeault, C.: Optimization of small-delay defects test quality by clock speed selection and proper masking based on the weighted slack percentage. IEEE Trans. Very Large Scale Integr. Syst, 28, 1–13 (2020)
13. Zhang, Q., et al.: Method and apparatus for asynchronous processor with a token ring based parallel processor scheduler. US Patent US20 150 074 680A1, Mar 2015. https://patents.google.com/patent/US20150074680A1
14. Huang, T., et al.: Method and apparatus for asynchronous processor pipeline and bypass passing. US Patent US9 846 581B2, Dec 2017. https://patents.google.com/patent/US9846581B2
15. Peeters, A., et al.: Click elements: an implementation style for data-driven compilation. In: 2010 IEEE Symposium on Asynchronous Circuits and Systems, 3–14 (May 2010)
16. Stevens, K., Ginosar, R., Rotem, S.: Relative timing (asynchronous design). IEEE Trans. Very Large Scale Integr. Syst. 11(1), 129–140 (Feb. 2003)
17. Stevens, K., Xu, Y., Vij, V.: Characterization of asynchronous templates for integration into clocked CAD flows. In: 2009 15th IEEE Symposium on Asynchronous Circuits and Systems, 151–161 (May 2009)
18. Manoranjan, J.V., Stevens, K.: Qualifying relative timing constraints for asynchronous circuits. In: 2016 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 91–98 (May 2016)
19. Gimenez, G., et al.: Static timing analysis of asynchronous bundled-data circuits. In: 2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 110–118 (May 2018)
20. Gimenez, G., Simatic, J., Fesquet, L.: 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, 86–95 (May 2019)
21. Waterman, A., Asanovic, K.: The RISC-V instruction set manual volume I: User-level ISA. RISC-V, Tech. Rep. (May 2017)
22. "CPU Benchmark – MCU Benchmark – CoreMark – EEMBC Embedded Microprocessor Benchmark Consortium. https://www.eembc.org/coremark/
23. Mead, C., Conway, L.: Introduction to VLSI systems. Addison-Wesley (1980)
24. Beerel, P.A., Ozdag, R.O., Ferretti, M.: A designer's guide to asynchronous VLSI. Cambridge University Press (2010)
25. Roncken, M., et al.: Naturalized communication and testing,. In: 2015 21st IEEE International Symposium on Asynchronous Circuits and Systems, 77–84 (May 2015)
26. Hennessy, J.L., Patterson, D.A.: Computer architecture: a quantitative approach. Elsevier (Oct. 2011)
27. Traylor, R.L.: Self-timed data pipeline apparatus using asynchronous stages having toggle flip-flops. US Patent US5 386 585A, Jan 1995. https://patents.google.com/patent/US5386585A/en
28. Quinton, B.R., Greenstreet, M.R., Wilton, S.J.E.: Practical asynchronous interconnect network design. IEEE Trans. Very Large Scale Integr. Syst. 16(5), 579–588 (2008)
29. Tang, K., Padubidri, S.: Diagonal and toroidal mesh networks. IEEE Trans. Comput. 43(7), 815–826 (1994)
30. Cormen, T.H., et al.: Introduction to algorithms. MIT Press (2001)
31. Sutherland, I.E.: Micropipelines. Commun. ACM. 32(6), 720–738 (1989)
32. Singh, M., Nowick, S.M.: MOUSETRAP: high-speed transition-signalling asynchronous pipelines. IEEE Trans. Very Large Scale Integr Syst. 15(6), 684–698 (2007)
33. Sparsø, J.: Current trends in high-level synthesis of asynchronous circuits. In: 2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009), 347–350 (Dec. 2009)
34. Smirnov, A., Taubin, A.: Synthesizing asynchronous micropipelines with design compiler. SNUG, 35 (2006)
35. Edwards, D., Bardsley, A.: Balsa: an asynchronous hardware synthesis language. Comput. J. 45(1), 12–18 (2002)

36. Cortadella, J., et al.: Desynchronization: synthesis of asynchronous circuits from synchronous specifications. IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 25(10), 1904–1921 (2006)
37. Fiorentino, M., et al.: A practical design method for prototyping self-timed processors using FPGAs. In: 2016 IEEE International Symposium on Circuits and Systems (ISCAS), 1754–1757 (May 2016)
38. Mardari, A., Jelčicová, Z., Sparsø, J.: Design and FPGA-implementation of asynchronous circuits using two-phase handshaking. In: 2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 9–18 (May 2019)
39. Andrikos, N., et al.: A fully-automated desynchronization flow for synchronous circuits. In: 2007 44th ACM/IEEE Design Automation Conference, 982–985 (Jun. 2007)
40. Fiorentino, M., Savaria, Y., Thibeault, C.: FPGA implementation of Token-based Self-timed processors: a case study. In: 2017 15th IEEE International New Circuits and Systems Conference (NEWCAS, 313–316 (Jun. 2017)

How to cite this article: Fiorentino, M., Thibeault, C., Savaria, Y.: Introducing *KeyRing* self-timed microarchitecture and timing-driven design flow. IET Comput. Digit. Tech. 15(6), 409–426 (2021). https://doi.org/10.1049/cdt2.12032