Static Timing Analysis of Asynchronous Bundled-Data Circuits

Grégoire Gimenez*†, Abdelkarim Cherkaoui*, Guillaume Cogniard†, Laurent Fesquet*
*Univ. Grenoble Alpes, CNRS, Grenoble INP**, TIMA, F-38000 Grenoble, France
†Dolphin Integration, F-38240 Meylan, France

***Abstract*—Self-timed circuits appear today as an attractive solution for designing robust and low-power chips dedicated to smart sensing and Internet of Things (IoT) platforms. However, a massive adoption of this technology by the industry requires industrial-grade tools for the whole design flow. The gap between asynchronous bundled-data and synchronous circuits is sufficiently tight to exploit the existing commercial tools without impacting the design flow and the time-to-market. This paper especially addresses the timing analysis of asynchronous bundled-data circuits with standard EDA tools and presents a method for exhaustively defining and verifying their relative timing constraints. This new approach only uses a combination of clocks to describe every possible event propagation path, allowing the tools to fully capture the relative timing constraints. Moreover, this can be adapted to different controller implementations and fully automated. A case-study, based on a 128-bit AES implemented in UMC 55nm uLP technology, illustrates the proposed methodology and evaluates its efficiency in terms of complexity and execution time.**

***Keywords*—Asynchronous circuit, micropipeline, bundled-data circuit, static timing analysis, relative timing constraints**

INTRODUCTION

Asynchronous circuits provide interesting solutions to address design problems inherent to the clock in digital circuits [1] (power consumption, clock domain crossing, electromagnetic emission, current peaks and sensitivity to process, voltage, and temperature). Bundled-data (BD) circuits [2] are the asynchronous circuit class which resembles the most synchronous circuits. They are composed of a datapath having the same structure than synchronous circuits, and of a control part which organizes data transfers between registers. They are particularly attractive since they offer increased robustness with an area similar to synchronous circuits [3].

BD circuits benefit today from several design methods which can be classified into two groups: template-based methods and methods synthesizing the circuit from a control model (usually based on Petri nets). In template-based methods, the control circuit is built as a connection of basic blocks [4]–[6]. Then, a model is derived for verification. On the contrary, in synthesis-based methods, the controller model is specified and verified. The circuit is thereafter generated either by direct-mapping [7], [8] or by synthesizing a high-level control specification [9]. These methods specify and verify the functionality of BD circuits, but they do not guarantee their temporal correctness.

**Institute of Engineering Univ. Grenoble Alpes

Timing analysis of BD circuits is not trivial. It relies on relative timing constraints (RTC), which are difficult to handle within standard EDA tools. Different approaches have been proposed in the literature to perform the timing analysis of BD circuits. The framework presented in [10] evaluates RTC on a robustness criterion and classifies them in order to resolve conflicts between constraints within the same set. [11], [12] use timed high-level models to verify timings in asynchronous circuits. Most of the other works rely on standard EDA tools to perform static timing analysis (STA) of BD circuit [13], [14]. They use sets of set_min/max_delay to constrain both the control and the data paths. In [15], virtual clocks are used to constrain the datapath but set_min_delay are kept to constrain the control circuit.

In the min/max methods, designers have to manually identify and specify relevant timing arcs for the analysis. Moreover, timing information should be provided for all process, voltage and temperature (PVT) corners. These are mostly iterative approaches based on successive extractions and post-processing of the timing results by third party scripts. Thus, these methods lack of automation and may have time-to-market issues due to the increasing number of PVT corners: they are not well-suited to large scale integration. Moreover, they do not benefit from EDA tools ability to explore the design space and to perform optimizations.

This paper proposes a methodology for using a standard EDA environment to perform static timing analysis of BD circuits: PrimeTime®, Synopsys® Design Constraints file (SDC) and Liberty files. To define each RTC, the proposed approach uses sets of clocks which are easily interpreted by the STA tool. The designer does not need to specify timing arcs to be cut, neither startpoints nor endpoints for STA at circuit level. Instead, he only provides generic rules for the used protocol and controller template. Therefore, the method is adaptable and fully automatable. It generates a PVT-independent SDC which is reused in all design steps. The proposed method has been used to perform STA on an AES (Advanced Encryption Standard) cipher in a 55 nm technology node. The results show a 20% runtime reduction without taking into account the specification time lessening.

I. BUNDLED-DATA TIMING ASSUMPTIONS

The correctness of BD circuits relies on two types of RTC [10], [16]: 1) *Controller template-level RTC*: They guarantee that the controller template respects its quasi-delay insensitive

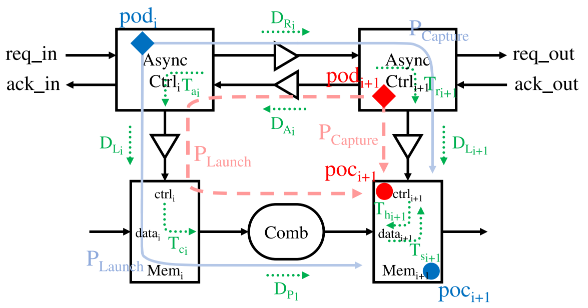


Figure 1: Hold (dashed) and setup (solid) paths in BD circuits. Delays are represented with dotted arrows.

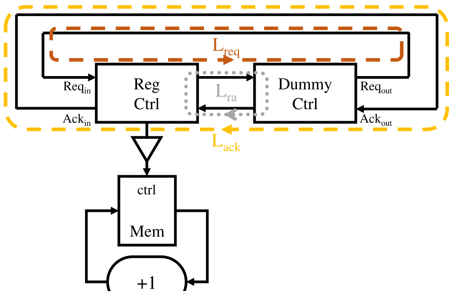


Figure 2: Architectural (L_{ack} and L_{req}) and local (L_{ra}) loops in the control circuit of a counter.

assumptions (*e.g.* isochronic forks). For example, they include RTC due to the feedback loops in some C-element implementations, or the min-pulse width constraints linked to the pulse generators often used for 2-phase protocols. [17] proposes a method to implement most of them using standard EDA tools. 2) *Protocol-level RTC*: They guarantee that data is sampled and transmitted while it is valid (setup and hold conditions).

This paper addresses protocol-level RTC and proposes a method to handle them in such way that they are transparent to the designer.

A. Protocol-level RTC in BD circuits

BD timing constraints can be expressed with the following RTC [10], [16]:

$$pod_i \mapsto data_{i+1} \prec ctrl_{i+1} \tag{1}$$

$$pod_{i+1} \mapsto ctrl_{i+1} \prec data_{i+1} \tag{2}$$

where pod_i is a point-of-divergence, and $data_i$ and $ctrl_i$ form a point-of-convergence (poc) as defined in timing characterization files (*i.e.* Liberty file).

Equation (1) presents the setup declination of the BD constraints while Eq. (2) shows its hold counterpart. Figure 1 exhaustively describes the relevant propagation paths in two adjacent BD circuit stages. pod_i, also called *last common point* by STA tools, is located in the asynchronous controller. It is the startpoint of both the launch and the capture timing paths. $data_i$ is the data pin of the memory element. It is the timing endpoint. In Fig. 1, RTC 1 and 2 can be expressed with the following inequalities:

$$D_{R_i} + T_{r_{i+1}} + D_{L_{i+1}} > D_{L_i} + T_{c_i} + D_{P_i} + T_{s_{i+1}} \tag{3}$$

$$D_{A_i} + T_{a_i} + D_{L_i} + T_{c_i} + D_{P_i} > D_{L_{i+1}} + T_{h_{i+1}} \tag{4}$$

B. Implementation considerations

A worst case scenario for hold timings happens when there is no combinational cells between two successive stages of the pipeline: $D_{P_i} = 0$. In this case, a delayed acknowledge signal may be required to compensate the virtual skew S_i that exists at the control pin of the two registers:

$$S_i = D_{L_{i+1}} - D_{L_i} \tag{5}$$

$$D_{A_i} > S_i + T_{h_{i+1}} - T_{c_i} - T_{a_i} \tag{6}$$

D_{A_i} should be as small as possible to avoid the degradation of the circuit performances (area, power, throughput). It results the following constraint on the skew:

$$S_i < T_{c_i} + T_{a_i} - T_{h_{i+1}} \tag{7}$$

This can be further approximated with $S_i < T_{a_i}$ if we consider that hold and capture times of the sequential element are equal. On the other hand, there is no lower bound for the virtual skew. However, any negative value of S_i should be compensated by the request delay D_{R_i}. This would consequently decrease the throughput of the circuit. Additionally, most of the BD designs contain architectural loops. It implies that $-S_i$ is also bounded by T_{a_i}. Finally, Eq. (8) gives a practical target for the virtual skew during physical synthesis. Insuring such limits guarantees a minimal request delay and that no acknowledge delay is required.

$$|S_i| < T_{a_i} \tag{8}$$

C. Synchronous EDA tools limitations

Several difficulties complicate the specification of BD circuits RTC in commercial EDA tools. They are mainly linked to the synchronous paradigm in which these tools have been developed and are used:

1) **clock vs event:** STA tools expect that a clock reaches every control pin of the sequential elements to activate the timing checks defined in their characterization file. However, the irregular propagation of events cannot be efficiently captured with the fixed-period clock definition.
2) **Local loops:** In BD control circuits, each request/acknowledge channel intrinsically forms a combinational loop between adjacent stages. Such loops are not supported by the timing analysis engine as they cannot be statically analyzed. If no precaution is taken, the tools automatically and arbitrarily cut these loops. They might disable relevant timing paths. An example of such local loops is illustrated by Fig. 2 with the dotted ellipse L_{ra}.
3) **Architectural loops:** The architectural loops in the controller are the consequence of loops in the datapath. These loops cannot be broken without disabling a timing path that should imperatively be held. As a basic example, the counter presented in Fig. 2 has two architectural loops L_{req} and L_{ack} (dashed arrows).

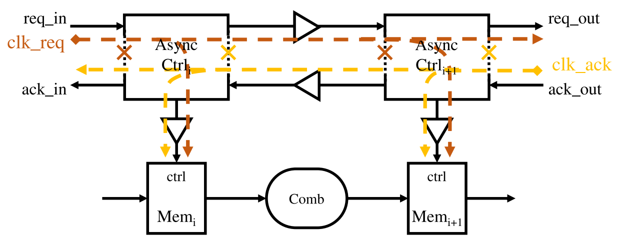


Figure 3: Request and acknowledge clocks propagation in a BD circuit.

4) **Point-of-divergence localization:** *pod*, which serve as references for the RTC, cannot be guessed by the synchronous EDA tools. They need to be manually described in an understandable manner for the timing engine without breaking any meaningful timing path.
5) **Datapath timing constraint:** The datapath should be constrained to optimize the circuit performances. Most importantly, the timing relationship between the datapath and the control path should be preserved - the RTC straddle for these two parts of the circuit. Ensuring this dependency should allow timing-driven tools to concurrently optimize control and data paths.

II. Proposed Methodology

The methodologies using commercial EDA tools fail to overcome all the difficulties introduced in Section I-C. All min/max delay methods split RTC into two disjoint timing constraints which compare timing paths to fixed time values. These fixed values need to be characterized at each PVT corner, leaving the timing engine ignorant of the real timing assumptions that it should implement. In the sequel, we proposed two methods to overcome these limitations. Both methods aim at instructing the tool on how to find the RTC without explicitly specifying them. The first approach, based on decoupling request and acknowledge paths, allows to perform the STA of linear BD circuits. The second approach, called *Local Clock Sets* (LCS) methodology, is generic to any BD circuit. Its main specificity is the possibility to define generic rules for an implementation style (protocol and controller template) instead of defining RTC specifically for the circuit itself. Proposed methodologies are illustrated with Weak Condition Half-Buffer (WCHB) controller templates [18] driving standard flip-flops. It implements a 4-phase protocol. Section II-D discusses the validity of the method and its extension to other protocols and controller templates.

A. Decoupling request and acknowledge paths

An asynchronous control circuit consists of two different combinational paths as shown in Fig. 3: a request path, flowing from left to right, and an acknowledge path propagating in the opposite direction. Making abstraction of the protocol relationship between these two signal paths, they are independent and respectively match the two RTC of Eqs. (3) and (4). Such a decoupled description of the controller breaks the local loops

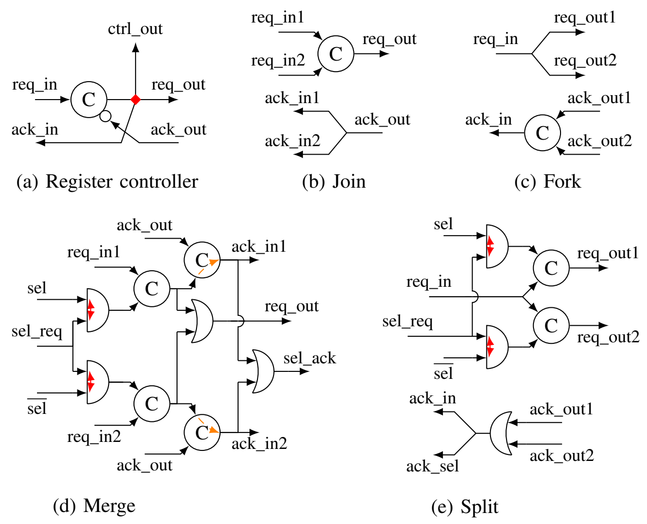


Figure 4: 4-phase WCHB control circuits

and enables to separately verify setup and hold timings of the BD circuit.

In standard EDA tools, splitting the control signal path into two distinct paths can easily be done using *set_disable_timing* commands on the appropriate timing arcs. For example, when a WCHB controller (Fig. 4a) is used, disabling the timing arcs $ack\_out \rightarrow req\_out$ of each C-element is sufficient. It breaks the local loops and simultaneously leaves a unique combinational path from the input request to every control pin of the memory elements. A similar approach can be done by disabling the timing arcs $req\_in \rightarrow req\_out$ and, thus, creating a unique combinational path from the acknowledge port to each register. A single clock can then be declared either on the request input or on the acknowledge input of the controller to respectively enable setup or hold checks.

Using clocks addresses the limitations 4 and 5 pointed in Section I-C: 1) EDA tools can now guess every *pod*. They are the last common points on the clock path, between the launch and the capture paths. For WCHB controllers, they are located at the output of each C-element (represented by a square in Fig. 4a). 2) Data and control pins of a register form a *poc*. They are automatically recognized by the timing engine once a clock reaches a memory control pin. Timing relationships between these two points can then be exploited by the STA engine. This information is included in the timing characterization files.

The usual behavior of the synchronous timing engine is to check setup between two successive active edges of the clock (Fig. 5a). But for BD circuits, the same signal edge generates the launch and the capture events (see arrows in Fig. 5b). Fortunately, EDA tools provide specific commands to modify the cycle number at which checks must be done with the *set_multicycle_path* command. This latter transforms a periodic clock into a single non-periodic event when specifying 0-

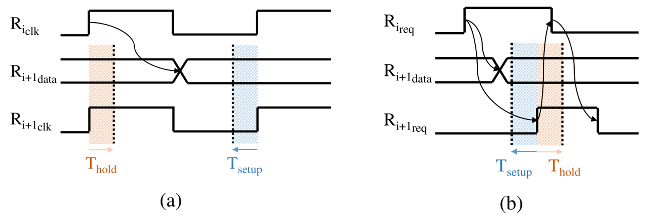


Figure 5: Timing waveform for a 2-stage (a) synchronous and (b) asynchronous pipeline.

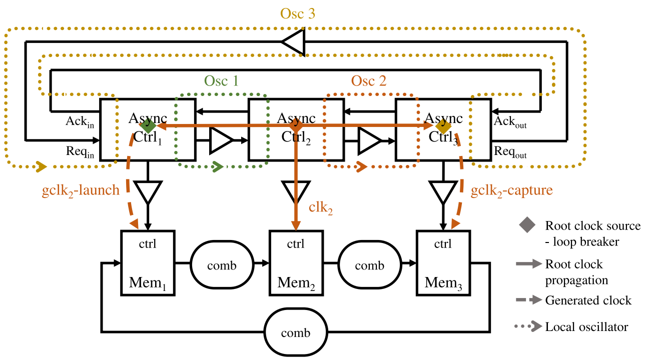


Figure 6: 3-stage looped BD circuit with local oscillators. LCS of oscillator 2 are represented.

cycle path for the setup analysis. Moreover, the clocks should be propagated using the *set_propagated_clocks* command. This way, the timing engine includes the control circuit delays in its computation. The hold timings are also checked at 0-cycle as in synchronous circuits. Finally, every setup timing can be verified when propagating a request event, and all the hold checks are analyzed when propagating an acknowledge event.

This first approach presents two limitations. First, setup and hold timings cannot be simultaneously analyzed since two conflicting sets of *set_disable_timing* commands are required: chip designers have to run the analysis twice. This is very limiting in terms of runtime and scalability. More importantly, this methodology fails to handle the architectural loops. Thus, it does not suit all types of designs.

B. Defining local clocks

To address these limitations, we developed a more accurate methodology. It also uses clocks to model events in the control circuit. But, instead of decoupling the request and acknowledge paths, it models the asynchronous controller as a series of local oscillators (Osc 1, 2 and 3 in Fig. 6), each generating a local clock. Two subtleties of the synchronous EDA tools are used. Firstly, local loops can be broken by defining root clocks (*create_clock* SDC command) instead of using the conventional *set_disable_timing* commands. These clocks actually define the startpoints for the timing analysis. They break any timing path crossing their source point. Thus, they can be used to break any timing cycle for the STA engine. By carefully selecting the breakpoints, both local and architectural loops are broken using this technique. Secondly, as clock definitions do not disable any timing arc, it is still possible to propagate an event through these breakpoints. Generated clocks can be used for this purpose (*create_generated_clock* SDC command). They allow to create launch and capture clocks beyond a root clock.

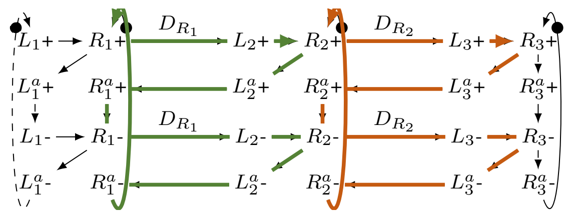


Figure 7: STG of a 3-stage pipeline using WCHB controllers. Two different local loops are highlighted (resp. green and brown arrows).

Arcs to be cut and *pod* are defined at the controller template level. The signal transition graph (STG) can be leveraged for specifying these generic rules. Figure 7 shows the STG of the 3-stage pipeline of Fig. 6 using WCHB controllers (for clarity, the channel between stage 3 and 1 has not been drawn). Such a representation of the asynchronous control circuit enables to identify the local timing loops and the right location for the clock sources to break these loops. We can see that the two timing cycles (highlighted in green and brown in Figs. 6 and 7) intersect in a single point: R_2. This latter is the best clock source candidate for breaking the local loops. It is the *pod* of the setup RTC between the stages 2 and 3 in the pipeline (and the hold RTC between stages 1 and 2):

$$pod_2 \mapsto data_3 \prec ctrl_3$$
$$pod_2 \mapsto ctrl_2 \prec data_2$$

Fig. 6 also shows the three clocks required to implement these two RTC. They are grouped in two local clock sets (LCS) composed of a root clock and a generated capture or launch clock: $\{clk_2; gclk_2\text{-}capture\}$ and $\{clk_2; gclk_2\text{-}launch\}$. They respectively enable to verify setup and hold timings.

For some controller templates, "dummy" memory controllers (DMC) should be added in architectural loops [6] to avoid deadlocks. These DMC generate local loops that should be broken even if no local clock is required (DMC do not drive any memory element). Dummy root clocks are created for that purpose. Notice that such clocks are not *pod*. Thus, they should not be propagated through the adjacent register controllers. On the opposite, any clock reaching the inputs of the DMC should cross them by creating a generated clock. These clocks are called event propagation clocks (EPC). They propagate the appropriate events through the structure (*i.e.* capture clocks for setup through request paths and launch clocks for hold through acknowledge paths).

Most BD designs require more complex controller structures. Merge and split controllers are used in combination of fork and join controllers to finely manage the data flow in the

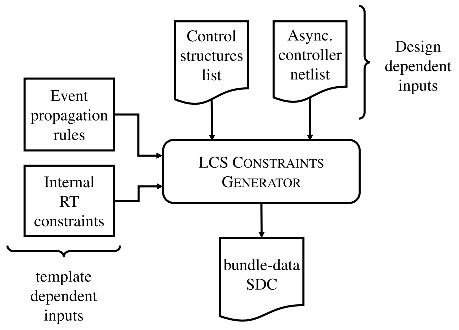


Figure 8: SDC file generation flow for the LCS methodology.

pipeline [19]. They are called flow control structures (FCS). These structures must be specifically treated to accurately propagate event. At the template level, arcs from requests to acknowledges generate local loops. They should be cut (specifying *set_disable_timing* or defining a dummy root clock) if they exist. In our case, these arcs are only present in the merge structure (dashed orange arrows in Fig. 4d). We used a *set_disable_timing* command since these arcs are not critical for the timing analysis. Indeed, they belong to a hold capture path and are unlikely longer than the environment response ($req\_out \rightarrow ack\_out$) with which they are synchronized. In the worst case, ignoring them will simply over-constrain this timing path. No additional clock is required but, to explicitly define all the possible event propagation paths, EPC should be defined.

Each generated EPC is ultimately related to a single root clock (*i.e.* an event). A set of EPC explicitly defines the propagation of a single event across the control circuit until it reaches another local clock (*i.e.* a new pipeline stage) and generates a capture or launch clock. A LCS is eventually a list of clocks that defines a possible event propagation path in the control circuit and maps to a singular BD RTC. It can be defined with the following generic form:

$$LCS = \{root\_clk, epc_1, ..., epc_i, capture/launch\_clk\}$$

By systematizing the definition of LCS, all the RTC of a BD circuit can be implemented. This approach makes the timing engine aware of all the real timing paths and violations. Moreover, it does not presume, at any time, the location of the matched delay. Thereby, this approach is seen as a complete translation, and not only a simple interpretation of the RTC. Finally, it gives to the tools all the latitude to perform any complex timing optimization.

C. Local clock set SDC generation

The list of LCS can be automatically built from a limited number of inputs. Fig. 8 presents the method to generate this LCS list. It generates a BD constraints file (SDC format) from the following inputs:

- *Asynchronous controller netlist*. It is a gate-level netlist of the controller. In our current design flow, this netlist is manually derived from the datapath specification using the flow presented in [6]. However, there is no constraint on the way this netlist is created.
- *List of control structures*. It gives the hierarchical names list of all the basic blocks which compose the controller (*i.e.* register controllers, dummy controllers and FCS). The non-controlling elements should be sorted according to the request (resp. acknowledge) propagation direction. It enables to generate series of EPC that reflect the setup (resp. hold) events propagation in the control circuit.
- *Event propagation rules*. They define the location of *pod* and the acknowledge/request paths for each type of register controller. They also list the required EPC and arcs to cut for each kind of FCS.
- *Internal RT constraints*. They include the additional RTC that may be required depending on the used template. These constraints are internal to each control structure and they do not affect communications between adjacent controllers. Translation of these RTC into SDC commands is detailed in [17].

```
# I) root clocks definition
FOR each register controller
  DO create clock on point of divergence
  DO add it to local clock set list
END
# II) dummy clocks definition
FOR each dummy controller
  DO create dummy clock on loop-breaker point
END
# events propagation
DO propagate clocks (update_timing)
# III) event propagation clocks definition
FOR each req/ack input of FCS & dummy controllers
  FOR each clock propagating to current input
    IF current clock is not dummy
      DO create generated clock on controller output \
                                 from req or ack input
      DO add it to the appropriate local clock set
# events propagation
      DO propagate clocks (update_timing)
    ENDIF
  END
END
# IV) capture/launch clocks definition
FOR each req/ack input of register controllers
  FOR each clock propagating to current input
    IF current clock is not dummy
      DO create generated clock on controller output \
                                 from req or ack input
      DO add it to the appropriate local clock set
    ENDIF
  END
END
# V) timing exceptions definition
DO define hold false path to/from capture clocks
DO define setup false path to/from launch clocks
DO define asynchronous group foreach local clock set
# Transform clock into event
DO define 0-cycle path from all clocks to all clocks
DO define all clocks as propagated
```

Figure 9: LCS SDC generator pseudo code.

Figure 9 gives the pseudo-code used within PrimeTime® tool to generate the timing constraint file. It goes through the following steps:

1) *Root clocks definition:* a root clock is created for each register controller. They define all the *pod* of the design. They also break the architectural loops and most of the

local request/acknowledge loops.

2) *Dummy clocks definition:* dummy clocks are created for every dummy loop controller. They break any remaining local loop.
3) *EPC definition:* a first pass of generated clocks definition is done to propagate root clocks in the FCS. All FCS elements are processed, one after the other, with respect to the actual event propagation in the controller. The ordered list of control structures prevents any discontinuity in the EPC declarations that may break timing relationships between the EPC and their root clock. The launch and capture propagation paths are differentiated according to the event propagation rules.
4) *Capture/launch clocks definition:* a second pass of clock generation is done to propagate events through the break-points (root clocks). Once again, launch and capture paths are separately processed.
5) *Timing exception definition:* finally, each clock is transformed into an event. The timing interactions between different local clock sets are disabled (set_false_path or set_clock_group commands), and irrelevant setup and hold timing paths are deactivated.

Several $update\_timing$ commands are required to generate all the LCS. They renew the timing database of the tool and ensure that each newly defined clock is propagated to adjacent controllers. Thus, they enable the tool to retrieve the connections between each control component and the appropriate event propagation path through the controller. All these possible propagation paths are successively captured in the list of LCS, creating a relative timing aware description of the controller: the *Local Clock Set Model*.

D. Validity and extension to other protocols and controller templates

Equation (4) does not completely model the early 4-phase protocol implemented by our WCHB controllers. It rather models the 2-phase protocol. Indeed, the return-to-zero (RTZ) phase is not taken into account in the crossed delay elements. The proposed launch LCS (*i.e.* which propagate an acknowledge event) inherit this limitation: they do not include EPC to model the RTZ phase of the protocol. This causes the design to be over-constrained since the missing delays would normally help in fixing hold timing. However, as previously mentioned in Section I-B, the hold timings are seldom an issue in BD circuits. Most of the time, there is no need to add any delay to fix them. In practice, this approximation rarely leads to the insertion of additional delays.

Yet, a more accurate LCS could be built. It requires two additional EPC. They model the RTZ phase of the protocol by propagating the launch event along a request-acknowledge loop between two adjacent stages of the pipeline. Similar techniques can be used to translate any other 4-phase protocol [20] into its LCS form.

At this point, the LCS model handles the protocol-level RTC but does not express the template internal RTC. The controllers we used to illustrate the method do not have any internal RTC

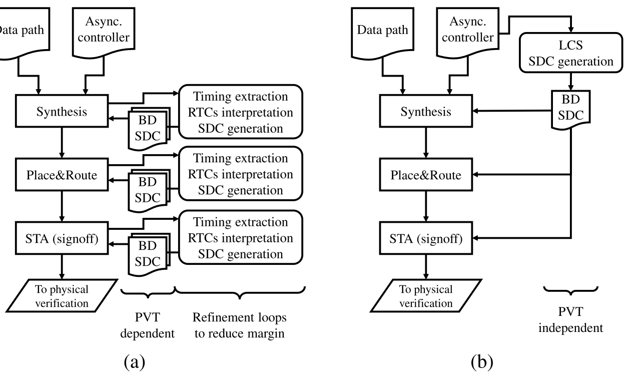


Figure 10: BD implementation flows for a) min/max delay and for b) LCS methodologies.

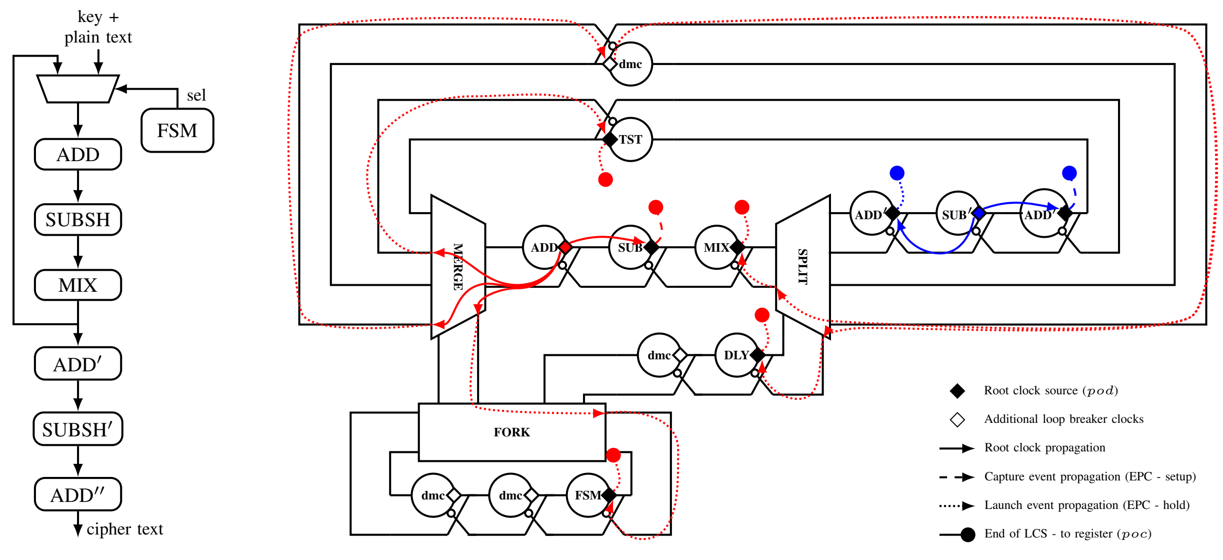
since they are delay insensitive [21]. However, many other templates have been proposed to enhance either performances, energy or area of BD circuit [18], [20], [22]. Most of them are at least quasi-delay insensitive templates. Thus, specific controller-related RTC should be defined.

[17] presents a method to transform most of these RTC into SDC constraints. It uses set_data_check commands to define the different poc. However, since incorrect behavior of the timing engine can appear when such commands are applied on clock paths, we recommend to use a dedicated timing mode to implement and verify these RTC. Other template related RTC, like minimum pulse width constraints, are already described in the Liberty files of the sequential elements but should be activated by defining generated pulse clocks. It can be done with the -$waveform$ option of the $create\_generated\_clock$ command. The set_sense command also helps in specifying the expected type of pulse. These commands are perfectly compatible with the LCS methodology.

E. Bundled-data circuits implementation

Figure 10 contrasts the min/max delay and the LCS based implementation flows. While the first one is an iterative approach that requires several PVT-dependent timing constraints files, the LCS methodology creates a single SDC constraint file that defines all the required RTC. This file is generated once. It can be then indistinctly used in logical synthesis and physical implementation tools to define RTC or in a STA tool to check the BD timing assumptions at any step of the design flow.

The LCS model can be used in several different ways during synthesis. A first approach aims at shortening each stage of the datapath. It uses the LCS model as previously presented and removes the propagated attribute of each clock. Thus, it instructs the tool to target a null delay for the datapath. The synthesis tool will attempt to reduce the datapath length to its minimum possible. This is a brute force approach which leads to high runtimes and drastically increases the datapath area. A second method enable to specify a timing budget for each stage of the pipeline. It uses the LCS model as the former method and also removes the multicycle attribute to



(a) datapath

(b) control circuit

Figure 11: 128-bit AES bundle-data circuit.

transform the events back into periodic clocks. A period can then be specified for each root clock to individually constrain each stage of the pipeline. This approach enables to finely tune the BD circuit at the expense of additional iterations. Finally, a more straightforward approach consists in using the same period for each LCS of the circuit. It will constrain the datapath similarly to synchronous designs.

The aim of the LCS method is not to insert the matched delays during physical synthesis. However, reporting the timing slack on each LCS directly gives the delay values to be inserted in the control circuit. They can be then implemented using $set\_minimum\_delay$ commands or using the tool clock tree synthesis capabilities. Ultimately, the matched delays will be automatically inserted using advanced concurrent clock and data optimization capabilities of the place and route tools. In synchronous designs, these techniques consist in degrading the clock tree skew to transfer timing slack between two adjacent stages. Applied to BD circuits, it will insert delays in the control circuit to meet the defined RTC.

Finally, LCS constraints are used during the timing signoff step of the flow to verify the BD RTC in every PVT corner.

III. Case Study

We used a 128-bit AES core as a test case for the LCS methodology. This block has been implemented in three different ways. We created timing constraints for each of them and performed STA. Then, we compared SDC files complexity and tool execution times. A synchronous version of the cryptographic block is used as reference. We developed a BD version of the design and generated the SDC file in two manners: firstly, using the common set_min/max_delay method, and secondly, using the LCS methodology. The datapath, which is common to the synchronous and the BD implementations, is presented in Fig. 11a. For the synchronous design, a global clock is distributed to each register stage (ADD, $SUBSH$, MIX...). This clock is replaced by an asynchronous control circuit in the BD design. We used the 4-phase WCHB controllers previously introduced. The control circuit is presented in Fig. 11b. For the sake of simplicity, flip-flops, delays elements and details on the FCS are not given.

Each pod is represented with a small colored square. They are the sources of the root clocks. In the case of this AES block, some events can generate up to 5 LCS (ADD controller). Each LCS is composed of a root clock and of one or several generated clocks. For instance, the LCS between ADD and MIX stages brings out 1 root clock and 4 generated clocks. In Fig. 11b, we only represented seven LCS, associated with pod of stages ADD and SUB'. In total, 9 root clocks, 4 dummy clocks and 54 generated clocks are required to build the 26 LCS that fully describe the BD RTC of the test-case (half are used for setup verification, the other half for hold verification).

Table I shows that LCS methodology has an important overhead in terms of timing constraint file complexity comparatively to the synchronous circuit and the min/max delay flow. Although the SDC is complex, it is generated in an automated fashion using high-level information. Moreover, analyzing the execution times of the $update\_timing$ command reported in Table II slightly mitigates this conclusion. This table presents the runtime variations for a single execution of the command and when taking into account the number of executions required by the flow (between parenthesis). Thereby the runtime of the min/max_delay method is composed of two values corresponding to the two successive calls to the command (one to extract the propagation times and another one once min/max delays have been adjusted). At the end, it takes only

1.5 more times than for the synchronous design to calculate all the timings for the LCS methodology. The larger number of clocks increases the size of the timing graph the tool has to process. It results in a larger memory footprint. However, the LCS methodology adds commands to deactivate every path that is shared between two different LCS. Ultimately, the effective number of timing paths to be analyzed remains the same. On the other hand, the competing min/max delay flow almost doubles the runtime of the synchronous analysis since it needs to call the $update\_timing$ command twice. Finally, comparing the two asynchronous methodologies shows a runtime reduction larger than 20% for the LCS approach.

IV. Discussion

At this point, the LCS method requires the designer to partially describe the structure of the asynchronous controller. In particular, any error in the order of the FCS may mask some RTC and skew the timing verification. Nonetheless, generating this list is potentially automatable. For example, the high-level synthesis methods and the direct-mapping methods use a high-level model of the controller (*e.g.* a Petri net) either to map the circuit or to verify it. Therefore, the model is able to directly provide this list.

Using the LCS timing constraints to analyze the AES block implemented with min/max delay revealed that a singular RTC located in the merge and split structures has been missed (red arrows in Figs. 4d and 4e). Like sequential elements, these structures are the intersection points between the datapath and the control circuit: BD constraints apply. In particular, the selector data of the (de)multiplexers, which comes from the datapath, should reach the block before the selector request arrives. This RTC and the associated min and max delays were not defined. In our case, this issue was not captured by the simulations. Fortunately, the LCS methodology recovers such RTC. Indeed, STA engine automatically infers a $clock\_gating\_check$ on every cell having on its inputs both a clock and a data signal. This specific behavior of the EDA tool is mainly due to the necessity of preventing any glitches on the clock tree that may otherwise generate faults in the circuit.

Finally, min/max methods do not guarantee that all RTC are exhaustively described and solved. Most of the time, RTC of Eqs. (1) and (2) are reduced to $S_i = 0$ (null virtual skew) and $D_P < D_R$ (setup constraint as shown in Fig. 1) to avoid conflicts between RTC of adjacent stages. They ignore parts of the RTC (*i.e.* D_L, T_a, T_r, T_c, T_s and T_h). Consequently, they are susceptible to any modification in these delays during the implementation. Moreover, these methods require as many SDC files as PVT corners. In practice, they split the setup RTC ($D_P < D_R$) into two absolute constraints: $D_P < \delta$ and $D_R > \delta$, where δ is a PVT dependent value.

On the contrary, the LCS method makes the timing engine aware of all the possible RTC without explicitly specifying delay values: it is a complete translation of the RTC instead of a simple interpretation. It uses a single SDC file, PVT-independent, all along the design flow (as shown in Fig. 10b).

Table I: Comparison of timing constraint file contents for synchronous, min/max delay and LCS versions of the AES circuit.

# commands	synchronous	asynchronous min/max delay	asynchronous LCS
$create\_clocks$	1	0	13
$create\_generated\_clocks$	0	0	54
$set\_disable\_timing$	0	15	2
set_min/max_delay	0	27	0
other	3	0	83
TOTAL	4	42	152
Ratio	1	10	38

Table II: Execution time and memory footprint of the $update\_timing$ command for synchronous, min/max delay and LCS versions of 10 AES blocks.

$update\_timing$	synchronous	asynchronous min/max delay	asynchronous LCS
Runtime (s)	9.84	8.96/9.65	14.95
Min. # of calls	1	2	1
Variation	*ref*	-2% (+89%)	+52% (+52%)
Memory (MB)	850.04	840.14	1032.62
Variation	*ref*	-1.2%	+21%

Giving the EDA tools the knowledge of the RTC that should be met is a key element for the optimization of any BD circuit. Combined with the LCS method, advanced capabilities of EDA tools (*e.g.* concurrent clock and data optimization) can be leveraged to efficiently match delays, to solve timing violations and to optimize the performances.

Conclusion

This work presents an automatable method based on standard EDA tools to perform STA of BD circuits, called local clock sets (LCS) methodology. Instead of defining min/max delay constraints, LCS models the circuit as interacting local oscillators, and propagates clock events in the appropriate paths to verify hold and setup timing constraints. Thus, it enables the STA tool to fully apprehend RTC instead of partially interpreting them. It also enables the tools to automatically solve violations. LCS generates a PVT-independent constraint file in an automated fashion. The rules for generating LCS are defined for an implementation style (protocol and controller template) and are not circuit dependent. Moreover, this constraint file is generated once and, then, used during the whole design flow. It does not need to be updated at each design step. Therefore, the proposed methodology makes the timing specification of BD circuits much easier as it only requires a few circuit dependent information. Moreover, the approach guides EDA tools to perform STA in an automated fashion making the operations transparent to the designer.

Acknowledgment

This work has been supported by Dolphin Integration.

References

[1] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design*. Springer, 2002.

[2] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[3] 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.

[4] M. Ferretti and P. A. Beerel, "High performance asynchronous design using single-track full-buffer standard cells," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 6, pp. 1444–1454, 2006.

[5] D. Hand, M. T. Moreira, H. H. Huang, D. Chen, F. Butzke, Z. Li, M. Gibiluka, M. Breuer, N. L. V. Calazans, and P. A. Beerel, "Blade – a timing violation resilient asynchronous template," in *21st IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2015, pp. 21–28.

[6] J. Simatic, A. Cherkaoui, B. François, R. Possamai Bastos, and L. Fesquet, "A practical framework for specification, verification and design of self-timed pipelines," in *23rd IEEE International Symposium on Asynchronous Circuits and Systems (Async 2017)*. San Diego, CA, United States: IEEE, May 2017.

[7] L. A. Hollaar, "Direct implementation of asynchronous control units," *IEEE Transactions on Computers*, vol. C-31, no. 12, pp. 1133–1141, 1982.

[8] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky, *Concurrent Hardware: The Theory and Practice of Self-timed Design*. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[9] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev, "Petrify: a tool for synthesis of petri nets and asynchronous circuits," Software, http://www.cs.upc.edu/ jordicf/petrify/ [accessed 2016-04-11].

[10] J. V. Manoranjan and K. S. Stevens, "Qualifying relative timing constraints for asynchronous circuits," in *22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2016, pp. 91–98.

[11] J. B. Møller, H. Hulgaard, and H. R. Andersen, "Timed verification of asynchronous circuits," *Concurrency and Hardware Design*, vol. 2549, pp. 274–312, 2002.

[12] S. Chakraborty, K. Y. Yun, and D. L. Dill, "Practical timing analysis of asynchronous circuits using time separation of events," in *Custom Integrated Circuits Conference, 1998. Proceedings of the IEEE 1998*. IEEE, 1998, pp. 455–458.

[13] K. S. 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.

[14] 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.

[15] 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*, June 2007, pp. 982–985.

[16] K. Stevens, R. Ginosar, and S. Rotem, "Relative timing," in *Proceedings. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1999, pp. 208–218.

[17] M. Prakash and P. Beerel, "Static timing analysis of template-based asynchronous circuits," Mar. 3 2015, uS Patent 8,972,915.

[18] S. B. Furber and P. Day, "Four-phase micropipeline latch control circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 2, pp. 247–253, June 1996.

[19] E. Yahya, "Performace modeling, analysis and optimization of multi-protocol asynchronous circuits," Ph.D. dissertation, Institut National Polytechnique de Grenoble - INPG, Dec 2009.

[20] J. Simatic, A. Cherkaoui, R. P. Bastos, and L. Fesquet, "New asynchronous protocols for enhancing area and throughput in bundled-data pipelines," in *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, Aug 2016, pp. 1–6.

[21] J. A. Brzozowski and K. Raahemifar, "Testing c-elements is not elementary," in *Proceedings Second Working Conference on Asynchronous Design Methodologies*, May 1995, pp. 150–159.

[22] K. Y. Yun, P. A. Beerel, and J. Arceo, "High-performance asynchronous pipeline circuits," in *Proceedings Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Mar 1996, pp. 17–28.