IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 28 December 2022, accepted 30 January 2023, date of publication 8 February 2023, date of current version 13 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3243224

RESEARCH ARTICLE

A Design Support Tool Set for Interface Circuits Between Synchronous and Asynchronous Modules

SHOGO SEMBA, (Member, IEEE), AND HIROSHI SAITO, (Member, IEEE)
School of Computer Science and Engineering, The University of Aizu, Aizuwakamatsu 965-8580, Japan

Corresponding author: Shogo Semba (shogo-s@u-aizu.ac.jp)

This work was supported in part by the Grant-in-Aid for Scientific Research from Japan Society for the Promotion of Science under Grant 21K11812.

ABSTRACT In this paper, we propose a design support tool set for interface circuits between synchronous and asynchronous modules. To facilitate the design of interface circuits between synchronous and asynchronous modules, the proposed tool set generates interface circuits and design constraints based on a predefined communication scheme. In addition, the proposed tool set performs timing verification and delay adjustment to guarantee the operations of the generated interface circuits. In the experiment, we evaluated the latency and overhead of the generated interface circuits. The latency and handshake overhead of the interface circuits generated by the proposed tool set depend on the cycle time of the receiver module. In addition, we designed a system which consists of a synchronous RISC-V processor and an asynchronous multilayer perceptron (MLP) circuit using the proposed tool set. The energy consumption of the system was reduced by 34.0% compared with a system which uses a synchronous MLP circuit.

INDEX TERMS Interface circuits, asynchronous circuits, design automation, low power.

I. INTRODUCTION

Most digital systems are designed based on the concept of System-on-a-Chip (SoC). SoCs are composed of several circuits such as microprocessors, memories, specific circuits, and so on. When these circuits are controlled by a clock signal, the power consumption of the clock network becomes high because the clock signal is distributed to the whole area. On the other hand, these circuits are often controlled by different clock signals. In such a case, synchronizers are required to reduce the metastability problem between different timing modules.

To solve these problems, Globally Asynchronous Locally Synchronous (GALS) was proposed in [1]. GALS systems are composed of several local synchronous modules. Each local module is controlled by an independent clock signal and communicated with other local modules asynchronously. However, interface circuits such as a two-flop synchronizer [2] and [3] are required to guarantee asynchronous communications between local synchronous modules.

Asynchronous circuits will be used instead of synchronous modules to reduce power consumption. Compared with synchronous circuits, asynchronous circuits have low power consumption because circuit components are controlled by local handshake signals instead of clock signals. When asynchronous circuits are communicated with synchronous circuits, interface circuits are also required between synchronous and asynchronous modules.

To guarantee asynchronous communications between synchronous and asynchronous modules, interface circuits based on handshake protocols were proposed in [4], [5], and [6]. In the interface circuits, local clock signals to write data to the internal registers of the interface circuits are generated using handshake signals. However, to guarantee the timing for writing data to the registers, the generation timings for the local clock signals must be adjusted by referring to the timing constraints for each register. In addition, design constraints are required for the interface circuits to satisfy the required performance and to transfer data correctly.

The associate editor coordinating the review of this manuscript and approving it for publication was Ludovico Minati.

Interface circuits designed in [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], and [16] use a design flow with commercial electronic design automation (EDA) tools for synchronous circuits. As commercial EDA tools do not intend the design of asynchronous circuits, design processes which are not supported by commercial EDA tools are performed by specific tools or hands.

In this paper, we propose a design support tool set for interface circuits between synchronous and asynchronous modules. The design target is the interface circuits described in [6]. The proposed tool set generates register transfer level (RTL) models with design constraints for the interface circuits. In addition, the proposed tool set performs timing verification and delay adjustment to satisfy the timing constraints of the interface circuits. The contributions of this paper are as follows.

- Using the proposed tool set, we can design interface circuits between synchronous and asynchronous modules automatically.
- The proposed tool set can be integrated into the general design flow due to the use of commercial EDA tools.

We expect that the proposed tool set enables the integration of asynchronous circuits into SoC designs. For example, there are many SoC designs including a synchronous CPU, synchronous hard-wired logics such as accelerators, and so on. When the synchronous hard-wired logics are always running, the power consumption of the SoC designs becomes high. We expect that the power consumption of the SoC designs can be reduced by converting the synchronous hard-wired logics to asynchronous ones and connecting the synchronous CPU and the asynchronous hard-wired logics through the interface circuits generated by the proposed tool set.

The rest of this paper is organized as follows. Section II describes related work. Section III describes asynchronous circuits with bundled-data implementation. Section IV describes the target interface circuits. Section V describes the proposed tool set for the design of interface circuits between synchronous and asynchronous modules. Section VI describes the experimental results. Finally, section VII describes the conclusion and future work.

II. RELATED WORK

Several design methods for interface circuits were proposed to transfer data between modules with different timings. On the other hand, to design asynchronous circuits using commercial EDA tools, design methods based on the design flow for synchronous circuits were proposed. In this section, we describe the differences between these methods and our proposed method.

To communicate between modules with different timings, several design methods for interface circuits were proposed in [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], and [16]. We describe those design methods for interface circuits and describe the differences between those methods and our proposed method.

For the communication between synchronous circuits with different timing signals, a two-flop synchronizer is well used (e.g., [2] and [3]). The two-flop synchronizer has a simple structure and reduces the metastability problem.

References [7], [8], [9] proposed an interface circuit based on handshake signals to transfer data between local synchronous modules in GALS systems. In [7], an interface circuit is designed using a pausible clock [17]. The pausible clock is based on an on-chip ring oscillator to pause the clock signal. In [8] and [9], to reduce the power consumption of interface circuits, clock gating is applied to interface circuits. In [7], [8], [9], controllers of the interface circuits are synthesized by specific tools such as 3D Tool [18] and Petrify [19].

References [10] and [11] proposed an interface circuit based on FIFOs to transfer data between local synchronous modules in GALS systems. The interface circuits consist of FIFOs and FIFO controllers. In each FIFO controller, a clock signal is generated using a mutual exclusion element (arbiter) and C-element [20].

Compared with [2], [3], and [7], [8], [9], [10], [11] where the target is the interface circuit between synchronous modules, we focus on the interface circuit between synchronous and asynchronous modules. We are going to reduce the power consumption of a system by using asynchronous modules.

References [12], [13], [14], [15], [16] proposed FIFOs to transfer data between synchronous and asynchronous modules. For each pipeline stage in the FIFOs, data are written to register using full/empty signals, read/write signals, handshake signals, or a clock signal. In [12], [13], [14], [15], [16], the FIFOs are synthesized using commercial EDA tools. Moreover, in [15], controllers of the FIFOs are synthesized by specific tools such as Petrify and Minimalist [21]. In [16], a design framework is provided to synthesize the FIFOs. However, the control circuits of the FIFOs are complex because the decision of memory addresses and the generation of full/empty or read/write signals are required. Compared with [12], [13], [14], [15], [16], we focus on the interface circuit based on handshake protocols. The interface circuit based on handshake protocols has a simple structure compared with the interface circuit based on FIFOs.

References [4], [5], [6] proposed an interface circuit based on handshake protocols to transfer data between synchronous and asynchronous modules. In [4], registers that are controlled by clock signals and handshake signals are used to support the coherent communication of multi-bit data between synchronous and asynchronous modules. In [5], interface circuits based on the two-flop synchronizer are designed to transfer data between synchronous and asynchronous modules. In [6], interface circuits using the two-flop synchronizer and Click Element [22] are used to communicate between synchronous and asynchronous modules. Compared with [4], [5], [6], the target of this paper is to support the design of interface circuits between synchronous and asynchronous modules by the proposed tool set.

To make the design of asynchronous circuits easy, several design methods based on the design flow with commercial

EDA tools for synchronous circuits were proposed in [23], [24], [25], [26], [27], [28], [29] and [30]. We describe those design methods for asynchronous circuits and describe the differences between those methods and our proposed method.

References [23], [24], [25], [26] proposed a conversion method from synchronous circuits to asynchronous circuits with bundled-data implementation. In [23], [24], [25], DFFs in synchronous gate level (GL) netlists synthesized by a commercial synthesis tool are replaced into master-slave latches with corresponding latch controllers. In [26], for synchronous RTL models generated by a high-level synthesis (HLS) tool, the clock signal of registers is replaced to the local clock signals from asynchronous control modules.

References [27] and [28] proposed a design flow for asynchronous circuits from a high-level language. In [27], asynchronous GL netlists are generated from Communicating Sequential Processes (CSP) models using a design flow called Proteus. In, [28], asynchronous circuits are generated from a high-level language called Haste using a design flow called TiDE.

References [29] and [30] proposed a design flow for asynchronous circuits on Field Programmable Gate Arrays (FPGAs). In [29], delay constraints for asynchronous circuits are generated by a developed tool. In addition, the timing for writing data to registers is guaranteed by a developed tool. In [30], placement constraints for asynchronous circuits are generated to reduce the number of delay adjustments by fixing the placement of asynchronous control modules.

Compared with [23], [24], [25], [26], [27], [28], [29], [30], the target of this paper is the design support for interface circuits between synchronous and asynchronous modules. The proposed tool set generates RTL models with delay and placement constraints for the interface circuits. Moreover, the proposed tool set adjusts the timing for writing data to the internal registers in the generated interface circuits.

III. ASYNCHRONOUS CIRCUITS WITH BUNDLED-DATA IMPLEMENTATION USING CLICK ELEMENTS

Bundled-data implementation is one of the data encoding schemes in asynchronous circuits. In bundled-data implementation, N-bit signals are represented by $N + 2$ signals. Additional two signals correspond to local handshake signals; the request signal *req* and the acknowledgment signal *ack*. The timing for writing data to registers is guaranteed by delay elements on *req* and *ack*.

In bundled-data implementation, there are two handshake protocols. From here, we represent a rising transition of a signal as $signal+$ and a falling transition of a signal as $signal-$. One is the two-phase handshake protocol in which only two signal transitions ($req+$ and $ack+$ or $req-$ and $ack-$) are used to transfer data. Another is the four-phase handshake protocol in which four signal transitions ($req+$, $ack+$, $req-$, and $ack+$) are used to transfer data.

In this work, we use Click Element [22] to control asynchronous modules. Click Element is one of the control templates used in the design of bundled-data implementation.

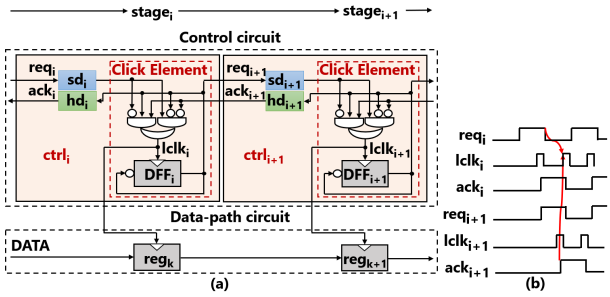


FIGURE 1. Asynchronous circuits with bundled-data implementation using Click Elements: (a) circuit model and (b) timing diagram of $ctrl_i$.

Click Elements are implemented as a data-driven two-phase handshake protocol.

A. CIRCUIT MODEL

Figure 1(a) shows the circuit model of bundled-data implementation using Click Elements. This circuit model consists of a data-path circuit and a control circuit. The data-path circuit is the same as the one used in synchronous circuits. The control circuit consists of control modules $ctrl_i$ ($0 \le i \le n-1$) assigned for each pipeline stage $stage_i$.

$ctrl_i$ consists of a Click Element, a delay element sd_i to satisfy the setup constraint of registers, and a delay element hd_i to satisfy the hold constraint of registers. The Click Element consists of a D Flip-Flop (DFF) DFF_i and a logic that generates a local clock signal $lclk_i$.

$ctrl_i$ starts its operation when req_i+ arrives at $ctrl_i$. req_i+ generates $lclk_i+$ through sd_i. Then, $lclk_i+$ controls DFF_i in $ctrl_i$ and registers reg_k in the data-path circuit. After the control of DFF_i and reg_k, DFF_i generates ack_i+ to pass the control to $ctrl_{i+1}$. In addition, ack_i+ arrives at $ctrl_{i-1}$ through hd_i to acknowledge that the operation of $ctrl_i$ is completed. Note that the behavior of $ctrl_i$ for req_i- is the same as the behavior of $ctrl_i$ for req_i+. Figure 1(b) shows the timing diagram of $ctrl_i$. Red arrows represent the generation of $lclk_i+$ from req_i- and $ack_{i+1}+$.

B. TIMING CONSTRAINTS

In bundled-data implementation, it is necessary to satisfy setup, hold, branch, and pulse width constraints to operate the circuit correctly. The detail of the timing constraints is described in [29] and [30]. In this sub-section, we describe the setup and hold constraints. We introduce p ($0 \le p \le m-1$) which represents the identifier of paths.

1) SETUP CONSTRAINT

The input data for reg_k must be stable before the setup time to write the input data to reg_k. This is called the setup constraint for reg_k. Figure 2(a) shows paths related to the setup constraint for reg_k. $sdp_{i,p}$ represents a data-path from $lclk_{i-1}$ to the destination register reg_k through the source register reg_{k-1}. In contrast, $scp_{i,p}$ represents a control-path from $lclk_{i-1}$ to reg_k through sd_i. We define the maximum

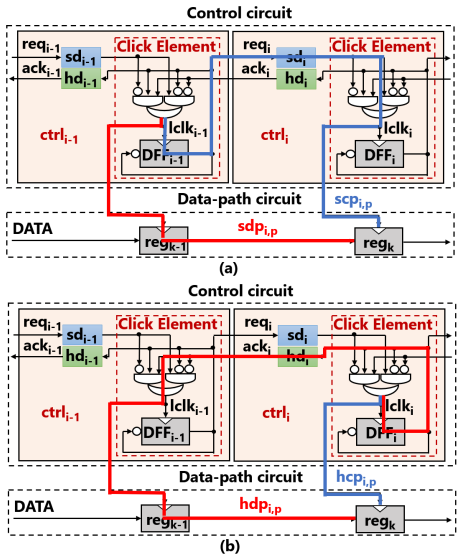


FIGURE 2. Paths related to timing constraints: (a) setup and (b) hold constraints.

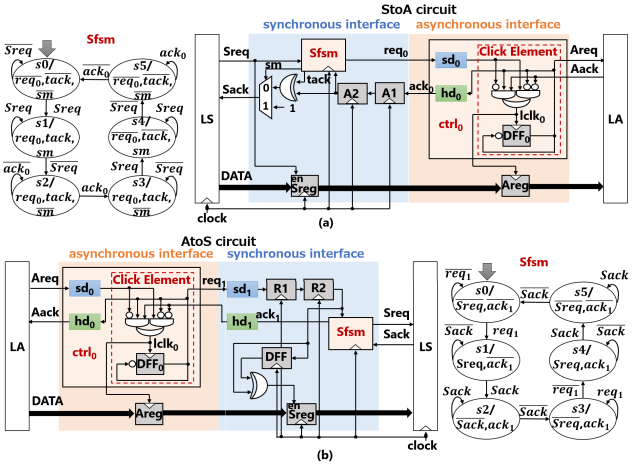


FIGURE 3. Interface circuits: (a) *StoA* circuit and (b) *AtoS* circuit.

delay of $sdp_{i,p}$ as $t_{maxsdp_{i,p}}$, the minimum delay of $scp_{i,p}$ as $t_{minscp_{i,p}}$, the margin for $t_{maxsdp_{i,p}}$ as $t_{sdpm_{i,p}}$, and the setup time of reg_k as $t_{setup_{i,p}}$. The setup constraint can be represented by the following inequality.

$$t_{minscp_{i,p}} > t_{maxsdp_{i,p}} + t_{sdpm_{i,p}} + t_{setup_{i,p}} \tag{1}$$

If this setup constraint is violated, we must adjust the number of cells in sd_i.

2) HOLD CONSTRAINT

The data must be stable for the hold time after the next input data are written to reg_k. This is called the hold constraint for reg_k. Figure 2(b) shows paths related to the hold constraint for reg_k. $hdp_{i,p}$ represents a data-path from $lclk_i$ to the destination register reg_k through hd_i. In contrast, $hcp_{i,p}$ represents a control-path from $lclk_i$ to reg_k. We define the minimum delay of $hdp_{i,p}$ as $t_{minhdp_{i,p}}$, the maximum delay of $hcp_{i,p}$ as $t_{maxhcp_{i,p}}$, the margin for $t_{maxhcp_{i,p}}$ as $t_{hcpm_{i,p}}$, and the hold time of reg_k as $t_{hold_{i,p}}$. The hold constraint can be represented by the following inequality.

$$t_{minhdp_{i,p}} > t_{maxhcp_{i,p}} + t_{hcpm_{i,p}} + t_{hold_{i,p}} \tag{2}$$

If this hold constraint is violated, we must adjust the number of cells in hd_i.

C. GLOBAL CYCLE TIME

To evaluate the performance of bundled-data implementation, a local cycle time (lct) and a global cycle time (gct) are defined in [6]. lct_i is the maximum delay of control-paths for $ctrl_i$ corresponding to $stage_i$. gct is the maximum value of lct_i.

lct_i is obtained by the following equation.

$$lct_i = max(t_{maxscp_{i,0}} - t_{maxdp_{i,0}}, \cdots, t_{maxscp_{i,m-1}} - t_{maxdp_{i,m-1}}) \tag{3}$$

$t_{maxdp_{i,p}}$ represents the maximum delay of $sdp_{i,p}$ except for the maximum path delay from the source register to the destination register.

IV. INTERFACE CIRCUITS BETWEEN SYNCHRONOUS AND ASYNCHRONOUS MODULES

In this paper, the design target of the proposed tool set is two interface circuits described in [6]. One is the *StoA* circuit to transfer data from a local synchronous module (*LS*) to a local asynchronous module (*LA*). Another is the *AtoS* circuit to transfer data from an *LA* to an *LS*.

The *StoA* and *AtoS* circuits consist of two interfaces called the synchronous interface and the asynchronous interface. The synchronous interface is controlled by the four-phase handshake protocol. In contrast, the asynchronous interface is controlled by the two-phase handshake protocol.

Figure 3 shows the *StoA* and *AtoS* circuits in [6]. The synchronous interface consists of a finite state machine (FSM) *Sfsm*, a register *Sreg*, a two-flop synchronizer (*A*1 and *A*2 or *R*1 and *R*2), and other components such as an XOR gate and a multiplexer. The two-flop synchronizer is used to receive ack_i which is an asynchronous input. In contrast, the asynchronous interface consists of $ctrl_i$ based on the Click Element and a register *Areg*. In the asynchronous interface, any synchronizer is not required for req_i because the timing for writing data to *Areg* is guaranteed by sd_i.

The behavior of the *StoA* circuit is as follows. The *StoA* circuit starts its operation when *Sreq*+ from *LS* arrives at *Sfsm*. *Sreq*+ controls *Sreg*. Then, to acknowledge that *DATA* is written to *Sreg*, *Sfsm* controls the multiplexer to generate *Sack*+. *Sfsm* also generates req_i+ to transfer data from *Sreg* to *Areg*. After the generation of req_i+, $lclk_i$+ is generated from req_i+ through sd_i. Then, $lclk_i$+ controls DFF_i and *Areg*. After the control of DFF_i and *Areg*, DFF_i generates ack_i+ and *Areq*+ to pass the control to *LA*. ack_i+ arrives at the two-flop synchronizer (*A*1 and *A*2) through hd_i to acknowledge that the operation of the asynchronous interface

is completed. Finally, to acknowledge that the operation of the synchronous interface is completed, *Sfsm* controls the multiplexer to generate $Sack-$ using the output of $A2$ when $Sreq-$ from *LS* arrives at *Sfsm*.

In contrast, the behavior of the *AtoS* circuit is as follows. The *AtoS* circuit starts its operation when $Areq+$ from *LA* arrives at $ctrl_i$. $Areq+$ generates $lclk_i+$ through sd_i. Then, $lclk_i+$ controls DFF_i and *Areg*. After the control of DFF_i and *Areg*, DFF_i generates $Aack+$ and $req_{i+1}+$ to pass the control to *Sfsm*. $Aack+$ arrives at *LA* through hd_i to acknowledge that the operation of the asynchronous interface is completed. $req_{i+1}+$ arrives at the two-flop synchronizer ($R1$ and $R2$) through sd_{i+1} to transfer data from *Areg* to *Sreg*. The output of $R2$ controls *Sreg* and *Sfsm*. Then, *Sfsm* generates $Sreq+$ to pass the control to *LS*. When $Sack+$ from *LS* arrives at *Sfsm*, *Sfsm* generates $Sreq-$ and $ack_{i+1}+$ to acknowledge that the operation of the synchronous interface is completed.

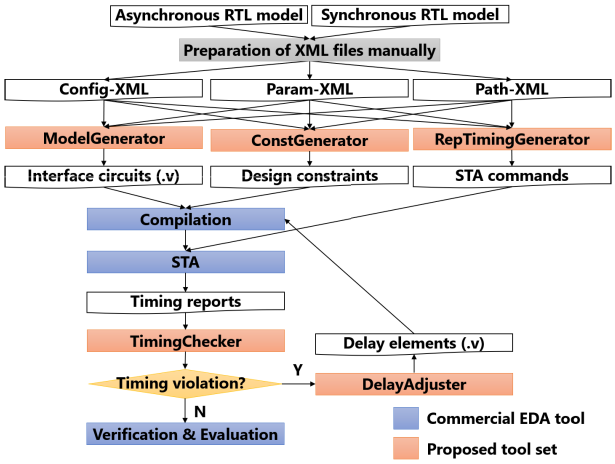


FIGURE 4. Flow of proposed tool set.

V. PROPOSED TOOL SET

In this paper, we propose a design support tool set for the interface circuits described in Sect. IV. Currently, the proposed tool set aims to implement the interface circuits on Intel FPGAs. The proposed tool set is based on the approaches for implementing asynchronous circuits on FPGAs described in [29] and [30]. Note that it is not difficult to adapt the proposed tool set to Xilinx FPGAs and Application Specific Integrated Circuits (ASICs) because we can replace RTL models and design constraints for Intel FPGAs with those for Xilinx FPGAs and ASICs. This extension is our future work.

FPGAs are reconfigurable devices whose circuit structures can be changed freely. Recently, FPGAs are used in many fields such as the Internet of Things (IoT) and deep learning because the design cost is low and the product life cycle is long compared with ASICs. In addition, since there are examples of asynchronous circuits implemented on FPGAs in [29] and [30], a design support tool set for interface circuits is also required according to the consideration of the connection between synchronous and asynchronous circuits in FPGAs. From the above, in this paper, we focus on FPGAs instead of ASICs.

A. OVERVIEW

Figure 4 shows the flow of the proposed tool set. To integrate the proposed tool set into the design flow for synchronous circuits, we use commercial EDA tools for compilation, static timing analysis (STA), verification, and evaluation. On the other hand, the processes which are not supported by commercial EDA tools are carried out by the proposed tool set. We support the generation of RTL models, design constraints, and STA commands for interface circuits. In addition, we support timing verification and delay adjustment.

As inputs of the proposed tool set, we use Extensible Markup Language (XML) files because XML can freely define tags according to the content of data. The XML files are prepared from synchronous and asynchronous RTL models. The XML files include the design parameters and the path information between synchronous and asynchronous modules. The detail of XML files is described in Sect. V-B.

After the preparation of the XML files, the proposed tool set generates RTL models and design constraints for the interface circuits from the XML files. *ModelGenerator* generates RTL models for the interface circuits. *ConstGenerator* generates delay constraints to satisfy the performance and placement constraints to fix the placement of the interface circuits on the target FPGA.

Then, we perform compilation and STA using commercial EDA tools. The compilation is performed for the RTL models of synchronous modules, asynchronous modules, and interface circuits using the generated design constraints. After the compilation, *RepTimingGenerator* generates STA commands to analyze path delays in the interface circuits. The STA is performed by a commercial EDA tool using the generated STA commands.

Finally, we perform timing verification and delay adjustment. *TimingChecker* checks whether there are timing violations in the interface circuits or not. If there are timing violations, *DelayAdjuster* adjusts the delays of delay elements to satisfy the timing constraints of the interface circuits.

B. INPUTS OF THE PROPOSED TOOL SET

Interface circuits are required to connect *LS* with *LA*. In this work, three kinds of XMLs are needed for the proposed tool set to synthesize the interface circuits automatically. The three XMLs are called Config-XML, Param-XML, and Path-XML, respectively. We prepare the Param-XML and Path-XML from synchronous and asynchronous RTL models manually. Figure 5 shows examples of Config-XML, Param-XML, and Path-XML. Arrows represent the dependencies between RTL models and XMLs.

1) CONFIG-XML

The Config-XML represents parameters for the generation of design constraints for the interface circuits as shown in

Fig. 5(a). ⟨*delayconst*⟩ represents whether delay constraints for the interface circuits are generated or not. ⟨*placeconst*⟩ represents whether placement constraints for the interface circuits are generated or not. The details of the generation of delay and placement constraints are described in Sect. V-D.

2) PARAM-XML

The Param-XML represents parameters for the generation of RTL models and design constraints for the interface circuits as shown in Fig. 5(b) and (d). Figure 5(b) is for the StoA circuit while Fig. 5(d) is for the AtoS circuit.

In the Param-XML, ⟨*StoA*⟩ and ⟨*AtoS*⟩ represent parameters for *StoA* and *AtoS* circuits. ⟨*StoA*⟩ and ⟨*AtoS*⟩ are prepared from synchronous and asynchronous RTL models. ⟨*sync*⟩ and ⟨*async*⟩ represent parameters for the synchronous interface and the asynchronous interface.

We explain the preparation of ⟨*StoA*⟩ in the Param-XML using Fig. 5(b). We assume that *LS* sends *reg0out* using the request signal *s2aSreq* and the acknowledgment signal *s2aSack* while *LA* receives *reg0out* using the request signal *a2sAreq* and the acknowledgment signal *a2sAack*. Therefore, an *StoA* circuit is expected to send *reg0out* from *LS* to *LA*. In ⟨*sync*⟩, a request signal name, an acknowledgment signal name, a clock signal name, and the clock cycle time are specified in "Sreq", "Sack", "Sclk", and "Sct". In this example, those are *s2aSreq*, *a2sSack*, *clock*1, and 18 ns, respectively. In ⟨*async*⟩, a request signal name, an acknowledgment signal name, and a global cycle time are specified in "Areq", "Aack", and "Agct". In this example, those are *s2aAreq*, *s2aAack*, and 8 ns, respectively.

⟨*const*⟩ in the Param-XML represents parameters for design constraints of *StoA* and *AtoS* circuits. ⟨*delayconst*⟩ and ⟨*placeconst*⟩ represent parameters for delay and placement constraints. In ⟨*delayconst*⟩, the target *gct* and the ratio of the *gct* are specified in "Tgct" and "crmax". In ⟨*placeconst*⟩, whether placement constraints are applied to the top module, synchronous interface, asynchronous interface, $ctrl_i$, *Sfsm*, and register are specified in "top", "sync", "async", "ctrl", "fsm", and "reg". ⟨*margin*⟩ represents margins for the timing constraints. In ⟨*margin*⟩, a control-path margin for the setup constraints, a data-path margin for the setup constraints, a control-path margin for the hold constraints, and a data-path margin for the hold constraints are specified in "scpm", "sdpm", "hcpm", and "hdpm". ⟨*ctrdelay*⟩, ⟨*pathratio*⟩, and ⟨*delement*⟩ represent parameters for control-path delays in $ctrl_i$, ratios of control-path delays in $ctrl_i$, and cell information used as delay elements, respectively.

3) PATH-XML

The Path-XML represents the path information between *LS* and *LA* as shown in Fig. 5(c) and (e). Figure 5(c) is for the StoA circuit while Fig. 5(e) is for the AtoS circuit. The Path-XML is used to generate RTL models, design constraints, and STA commands for the interface circuits.

In the Path-XML, ⟨*StoA*⟩ or ⟨*AtoS*⟩ represent the path information from *LS* to *LA* or *LA* to *LS*. ⟨*StoA*⟩ and ⟨*AtoS*⟩ are prepared from synchronous and asynchronous RTL models. ⟨*hierarchy*⟩ represents the hierarchical structures of *LS* and *LA*. ⟨*path*⟩ represents a path between *LS* and *LA*.

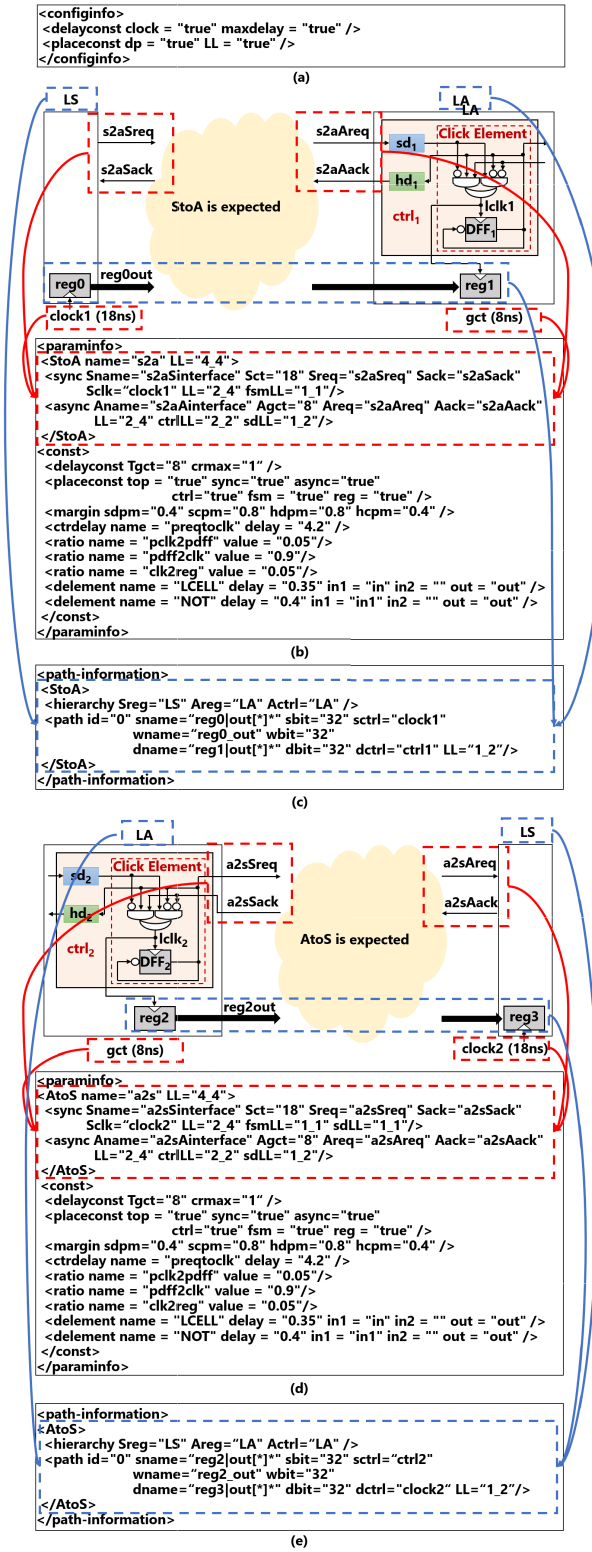


FIGURE 5. Inputs of proposed tool set: (a) Config-XML, (b) Param-XML for *StoA* circuit, (c) Path-XML for *StoA* circuit, (d) Param-XML for *AtoS* circuit, and (e) Path-XML for *AtoS* circuit.

We explain the preparation of ⟨*StoA*⟩ in the Path-XML using Fig. 5(c). *LS* sends *reg*0*out* from *reg*0 and *LA* receives *reg*0*out* to reg1. We assume that the bit-width of *reg*0*out* is 32. *reg*0 is controlled by *clock*1 while *reg*1 is controlled by $lclk_1$ from $ctrl_1$. In ⟨*hierarchy*⟩, parent hierarchies for the register of *LS*, the register of *LA*, and the control module of *LA* are specified in "Sreg", "Areg", and "Actrl". In this example, those are *LS*, *LS*, and *LA*, respectively. In ⟨*path*⟩, a source register, its bit width, its controller, a wire name, its bit width, a destination register, its bit width, and its controller are specified in "sname", "sbit", "sctrl", "wname", "wbit", "dname", "dbit", and "dctrl". In this example, those are *reg*0, 32, *clock*1, *reg*0*out*, 32, *reg*1, 32, and $ctrl_1$, respectively.

C. MODELGENERATOR

ModelGenerator generates RTL models for *StoA* and *AtoS* circuits by referring to the Param-XML and Path-XML. The RTL models are specified by Verilog Hardware Description Language (HDL).

First, *ModelGenerator* generates an *StoA* or *AtoS* circuit. *ModelGenerator* obtains a module name from "name" of ⟨*StoA*⟩ or ⟨*AtoS*⟩ in the Param-XML. *ModelGenerator* generates registers by obtaining the number of transfer data from the Path-XML. *ModelGenerator* obtains the number of transfer data by counting the number of paths with different "sname" of ⟨*path*⟩ in the Path-XML. *ModelGenerator* obtains the bit-width for "sname" by referring to "sbit" of ⟨*path*⟩ in the Path-XML. In addition, *ModelGenerator* generates controllers (*Sfsm* and $ctrl_i$) and other components. The controllers are predesigned as a template.

Next, *ModelGenerator* decides the number of cells in delay elements (sd_i) initially using the following equation.

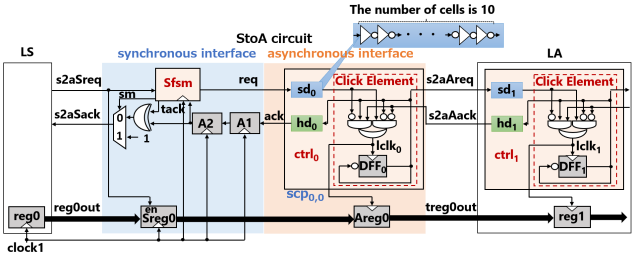
$$num_{sd_i} = \lceil (gct - t_{preqlclk})/t_{cell} \rceil \tag{4}$$

where num_{sd_i} represents the number of cells in delay elements, *gct* represents the global cycle time, $t_{preqlclk}$ represents the delay from DFF_{i-1} to $lclk_i$, and t_{cell} represents the delay of the cell used as the delay element. *gct*, $t_{preqlclk}$, and t_{cell} are obtained by referring to "Agct" of ⟨*async*⟩, "value" of ⟨*ctrdelay*⟩, and "delay" of ⟨*delement*⟩ in the Param-XML.

In addition, *ModelGenerator* generates hd_i using "assign" statements initially. This is because cells in hd_i are not required if there are no hold violations.

To implement delay elements, *ModelGenerator* generates primitive cells by referring to "name", "in", and "out" of ⟨*delement*⟩ in the Param-XML. Basically, *ModelGenerator* uses inverters for the delay elements to reduce the difference between the delays of the rising and falling transitions of a signal. If *num* is an odd number, a buffer is used as the last one of the delay elements. In addition, *ModelGenerator* uses the "synthesis keep" attribute for Intel FPGAs to prevent the optimization of the generated primitive cells.

Finally, *ModelGenerator* instantiates the generated *StoA* or *AtoS* circuit. Then, *ModelGenerator* connects *LS* with *LA* through the generated *StoA* or *AtoS* circuit using wire names




```
module s2a(clock, reset, Sreq, Sack, Sin0, Areq, Aack, Aout0);
    input clock, reset, Sreq, Aack;
    input [31:0] Sin0;
    output Sack, Areq;
    output [31:0] Aout0;
    wire req, ack;
    wire [31:0] Sout0;
    wire [31:0] Sout1;

    s2aSinterface s2aSinterface(.clock(clock), .reset(reset), .Sreq(Sreq), .Sack(Sack),
                                .req(req), .ack(ack), .Sin0(Sin0), .Sout0(Sout0));
    s2aAinterface s2aAinterface(.reset(reset), .Areq(Areq), .Aack(Aack),
                                .req(req), .ack(ack), .Ain0(Sout0), .Aout0(Aout0));
endmodule
module s2aSinterface(clock, reset, Sreq, Sack, req, ack, Sin0, Sout0);
    input reset, clock, Sreq, ack;
    input [31:0] Sin0;
    output Sack, req;
    output [31:0] Sout0;
    wire A1out, A2out;
    wire tSack, sm, tmpSack;

    s2adffa1 A1(.in(ack), .reset(reset), .clock(clock), .out(A1out));
    s2adffa1 A2(.in(A1out), .reset(reset), .clock(clock), .out(A2out));
    s2aSfsm Sfsm(.clock(clock), .reset(reset), .Sreq(Sreq), .ack(A2out), .tSack(tSack), .req(req), .sm(sm));
    s2aedffa32 Sreg0(.in(Sin0), .reset(reset), .clock(clock), .en(Sreq), .out(Sout0));
    assign tmpSack = tSack ^ A2out;
    s2amux2 mux2(.in0(tmpSack), .in1(1'b1), .sm(sm), .out(Sack));
endmodule
module s2aAinterface(reset, Areq, Aack, req, ack, Ain0, Aout0);
    input reset, req, Aack;
    input [31:0] Ain0;
    output ack, Areq;
    output [31:0] Aout0;
    wire lclk;

    s2actrl ctrl0(.reset(reset), .preq(req), .nack(Aack), .lclk(lclk), .pack(ack), .nreq(Areq));
    s2adffa32 Areg0(.in(Ain0), .reset(reset), .clock(lclk), .out(Aout0));
endmodule
module s2asd(in, out);
    input in;
    output out;
    wire w0 /* synthesis keep */;
    wire w1 /* synthesis keep */;
    wire w2 /* synthesis keep */;
    wire w3 /* synthesis keep */;
    wire w4 /* synthesis keep */;
    wire w5 /* synthesis keep */;
    wire w6 /* synthesis keep */;
    wire w7 /* synthesis keep */;
    wire w8 /* synthesis keep */;
    NOT d0 (.in1(in), .out(w0));
    NOT d1 (.in1(w0), .out(w1));
    NOT d2 (.in1(w1), .out(w2));
    NOT d3 (.in1(w2), .out(w3));
    NOT d4 (.in1(w3), .out(w4));
    NOT d5 (.in1(w4), .out(w5));
    NOT d6 (.in1(w5), .out(w6));
    NOT d7 (.in1(w6), .out(w7));
    NOT d8 (.in1(w7), .out(w8));
    NOT d9 (.in1(w8), .out(out));
endmodule
```

FIGURE 6. Example of RTL model for *StoA* circuit.

"Sreq", "Sack", "Areq", and "Aack" of ⟨*StoA*⟩ or ⟨*AtoS*⟩ in the Param-XML.

Figure 6 shows the generated RTL model for the *StoA* circuit from the Param-XML and Path-XML in Fig. 5. The 32-bit registers *Sreg*0 and *Areg*0 are generated by referring to ⟨*path*⟩ in ⟨*StoA*⟩. Then, the controllers *Sfsm* and $ctrl_0$ and other components are generated. Moreover, 10 inverters (*NOT*) in sd_0 are generated by the equation (4) ($\lceil (gct - t_{preqlclk})/t_{cell} \rceil = \lceil (8.0 - 4.2)/0.4 \rceil = 10$). Finally, the generated *StoA* circuit is connected with local modules using wire names such as *s2aSreq*, *s2aSack*, *s2aAreq*, and *s2aAack*. Note that the generation of the *AtoS* circuit is the same as the generation of the *StoA* circuit.

D. CONSTGENERATOR

To achieve the target performance and reduce the number of delay adjustments, *ConstGenerator* generates delay and

```
### create_clock ###
create_clock -name s2alclk -period 8.0 -waveform {0 4.0} ¥
[get_pins {s2a|s2aAinterface|ctrl0|lclk*|combout }]

### set_max_delay ###
set_max_delay -from [get_ports {clock1}] -to [get_registers {s2a|s2aSinterface|Sfsm|req*}] 0.4
set_max_delay -from [get_registers {s2a|s2aSinterface|Sfsm|req*}] ¥
-to [get_pins {s2a|s2aAinterface|ctrl0|lclk*|combout}] 7.2
set_max_delay -from [get_pins {s2a|s2aAinterface|ctrl0|lclk*|combout}] ¥
-to [get_registers {s2a|s2aAinterface|Areg0|out[*]*}] 0.4
```

FIGURE 7. Example of delay constraints.

placement constraints. The delay constraints are specified in a Synopsys Design Constraints (SDC) file. The placement constraints are specified in a Tool Command Language (TCL) file.

1) GENERATION OF DELAY CONSTRAINTS

Since there is no global clock signal in asynchronous circuits, delay constraints are used to satisfy the performance of asynchronous circuits.

ConstGenerator generates delay constraints for the asynchronous interface when "true" is assigned to ⟨*delayconst*⟩ in the Config-XML. For the asynchronous interface, *ConstGenerator* generates local clock constraints for lck_i and maximum delay constraints for scp_i by referring to ⟨*delayconst*⟩ and ⟨*pathratio*⟩ in the Param-XML.

ConstGenerator decides the local clock cycle time $const_{lck_i}$ by the following equation.

$$const_{lck_i} = Tgct * crmax \tag{5}$$

Tgct and *crmax* are obtained from ⟨*delayconst*⟩ in the Param-XML. Note that the pulse width of $lclk_i$ is assumed to the half of $const_{lck_i}$.

ConstGenerator generates maximum delay constraints for scp_i. Maximum delay constraints are usually assigned to ports and registers. Therefore, scp_i is divided into a path from $lclk_{i-1}$ to DFF_{i-1}, a path from DFF_{i-1} to $lclk_i$, and a path from $lclk_i$ to the destination register. As an example in Fig. 6, the scp_i in the asynchronous interface is divided into the path from *clock*1 to *Sfsm*, the path from *Sfsm* to $lclk_0$, and the path from $lclk_0$ to *Areg*0.

ConstGenerator decides the value $const_{scp_{i,j}}$ for each maximum delay constraint by the following equation.

$$const_{scp_{i,j}} = Tgct * crmax * R_j \tag{6}$$

where R_j ($0 \leq j \leq 2$) represents the ratio of the delay for each divided path. R_j from $lclk_{i-1}$ to DFF_{i-1}, from DFF_{i-1} to $lclk_i$, and from $lclk_i$ to the destination register are obtained from "pclk2pdff", "pdff2lck", and "lck2dff" of ⟨*pathratio*⟩ in the Param-XML.

Figure 7 shows the generated delay constraints for the *StoA* circuit by referring to the Param-XML in Fig. 5. In $lclk_i$ of the asynchronous interface, $const_{scp_i}$ is 8.0 ($Tgct * crmax = 8 * 1.0$). In the path from *clock*1 to *Sfsm*, $const_{scp_{i,j}}$ for *pclk*2*pdff* is 0.4 ($Tgct * crmax * R_j = 8 * 1.0 * 0.05$). Similar to $const_{scp_{i,j}}$ for *pclk*2*pdff*, $const_{scp_{i,j}}$ for *pdff* 2*lck* and *lck*2*dff* are 7.2 and 0.4.

```
set_global_assignment -name PARTITION_NETLIST_TYPE POST_SYNTH -section_id Top
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL ¥
PLACEMENT_AND_ROUTING -section_id Top
set_instance_assignment -name PARTITION_HIERARCHY ctrl0 -to s2a|s2aAinterface|ctrl0 ¥
-section_id ctrl0
set_global_assignment -name PARTITION_NETLIST_TYPE POST_SYNTH -section_id ctrl0
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL ¥
PLACEMENT_AND_ROUTING -section_id ctrl0
set_instance_assignment -name PARTITION_HIERARCHY sd0 -to s2a|s2aAinterface|ctrl0|sd0 ¥
-section_id sd0
set_global_assignment -name PARTITION_NETLIST_TYPE POST_SYNTH -section_id sd0
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL ¥
PLACEMENT_AND_ROUTING -section_id sd0
```

(a)

```
set_instance_assignment -name LL_MEMBER_OF s2a -to top|s2a -section_id s2a
set_global_assignment -name LL_AUTO_SIZE OFF -section_id s2a
set_global_assignment -name LL_WIDTH 4 -section_id s2a
set_global_assignment -name LL_HEIGHT 4 -section_id s2a
set_global_assignment -name LL_RESERVED ON -section_id s2a
set_global_assignment -name LL_PARENT rvcoremp_top -section_id s2a
set_instance_assignment -name LL_MEMBER_OF s2aAinterface ¥
-to s2a|s2aAinterface -section_id s2aAinterface
set_global_assignment -name LL_AUTO_SIZE OFF -section_id s2aAinterface
set_global_assignment -name LL_WIDTH 2 -section_id s2aAinterface
set_global_assignment -name LL_HEIGHT 4 -section_id s2aAinterface
set_global_assignment -name LL_RESERVED ON -section_id s2aAinterface
set_global_assignment -name LL_PARENT s2a -section_id s2aAinterface
```

(b)

FIGURE 8. Example of placement constraints: (a) Design Partitions and (b) LogicLocks.

2) GENERATION OF PLACEMENT CONSTRAINTS

For Intel FPGAs, when asynchronous circuits are re-synthesized by delay adjustment, the placements of control modules and delay elements are slightly changed. As a result, the number of delay adjustments is increased because the delays of control modules and delay elements are changed. In addition, when the distance between the synchronous interface and the asynchronous interface in *StoA* and *AtoS* circuits becomes long, the transfer latency between the synchronous interface and the asynchronous interface may be increased.

To reduce the number of delay adjustments and reduce the transfer delay, *ConstGenerator* generates two kinds of placement constraints when "true" is assigned to ⟨*placeconst*⟩ in the Config-XML. One is Design Partitions to specify which stage of synthesis is kept for each resource. Another is LogicLocks to specify the placement region for each resource.

ConstGenerator generates Design Partitions for *StoA* and *AtoS* circuits as shown in Fig. 8(a). To keep the synthesis result of *StoA* and *AtoS* circuits, *ConstGenerator* generates Design Partitions for the top-level modules of *StoA* and *AtoS* circuits. Since the hazard-free implementation for $ctrl_i$ is required, *ConstGenerator* also generates Design Partitions for the $ctrl_i$ in *StoA* and *AtoS* circuits to prevent hazards caused by optimizations. In addition, to prevent significant changes in delays of sd_i and hd_i by re-synthesis, *ConstGenerator* also generates Design Partitions for the sd_i and hd_i in *StoA* and *AtoS* circuits.

ConstGenerator generates LogicLocks for *StoA* and *AtoS* circuits by referring to a Logic Lock region "LL" in the Param-XML and Path-XML. To reduce routing delays between the synchronous interface and the asynchronous interface, *ConstGenerator* generates LogicLocks for the top-level modules of *StoA* and *AtoS* circuits. When logics in *StoA* and *AtoS* circuits are placed in different regions, the delays of the logics are varied by routing delays. Therefore, *ConstGenerator* generates LogicLocks for the *Sfsm* and $ctrl_i$, sd_i, and registers of *StoA* and *AtoS* circuits to fix their delays

```
### report_timing ###
report_timing -file s2aTimingReport/s_s0_p0_max.txt ¥
-from_clock { CLK1 } -to_clock { s2alclk } ¥
-from    [get_registers {s2a|s2aSinterface|Sreg0|out[*]*}] ¥
-to      [get_registers {s2a|s2aAinterface|Areg0|out[*]*}] -show_routing -nworst 1

### report_path ###
report_path -file s2aTimingReport/s2alclk_s2aAreg0_max.txt ¥
-rise_from   [get_pins {s2a|s2aAinterface|ctrl0|lclk*|combout}] ¥
-to      [get_registers {s2a|s2aAinterface|Areg0|out[*]*}] -show_routing -nworst 1
```

FIGURE 9. Example of STA commands.

of them. Note that *ConstGenerator* does not generate LogicLocks for hd_i because cells are not inserted in hd_i if there are no hold violations.

Figure 8(b) shows the generated LogicLocks for the *StoA* circuit by referring to the Param-XML and Path-XML in Fig. 5. The LogicLock commands for the top-level module for the *StoA* circuit are generated by referring to "LL" of ⟨*StoA*⟩ in the Param-XML. Similarly, the LogicLock commands for the *Sfsm*, $ctrl_i$, sd_i, *Sreg*, and *Areg* are generated.

E. REPTIMINGGENERATOR

To analyze the timing constraints, we must analyze delays of all paths in the *StoA* and *AtoS* circuits. Therefore, *RepTimingGenerator* generates STA commands to analyze path delays. STA commands are specified in a TCL file.

RepTimingGenerator generates STA commands for *StoA* and *AtoS* circuits as shown in Fig. 9. For paths between registers in *StoA* and *AtoS* circuits, *RepTimingGenerator* generates *report_timing* commands. For other paths in *StoA* and *AtoS* circuits, *RepTimingGenerator* generates *report_path* commands. In addition, for paths between the *StoA* or *AtoS* circuit and *LS* or *LA*, *RepTimingGenerator* generates STA commands by referring to ⟨*path*⟩ in the Param-XML.

F. TIMINGCHECKER

In *StoA* and *AtoS* circuits, it is necessary to satisfy setup and hold constraints for the internal registers to operate the circuits correctly.

To verify the setup and hold constraints, *TimingChecker* calculates the path delays from the timing report files generated by STA using *report_timing* and *report_path* commands. To analyze the timing constraints, *TimingChecker* obtains margins $t_{sdpm_{i,p}}$ and $t_{hcpm_{i,p}}$ from "sdpm" and "scpm" of ⟨*margin*⟩ in the Param-XML.

The timing constraints of *StoA* and *AtoS* circuits are similar to the inequalities (1) and (2). Since the synchronous interfaces in *StoA* and *AtoS* circuits are controlled by clock signals, we consider the number of cycles of paths for scp_i, sdp_i, hcp_i, and hdp_i through the synchronous interfaces. Therefore, we extend the inequalities (1) and (2) to the following inequalities (7) and (8).

$$t_{minscp_{i,p}} + Sct * num > t_{maxsdp_{i,p}} + t_{sdpm_{i,p}} + t_{setup_{i,p}} \quad (7)$$

$$t_{minhdp_{i,p}} + Sct * num > t_{maxhcp_{i,p}} + t_{hcpm_{i,p}} + t_{hold_{i,p}} \quad (8)$$

where $scp_{i,p}$ represents a control-path except for the path through the synchronous interface and $hdp_{i,p}$ represents a data-path except for the path through the synchronous interface. *num* represents the number of cycles of the path in the synchronous interface.

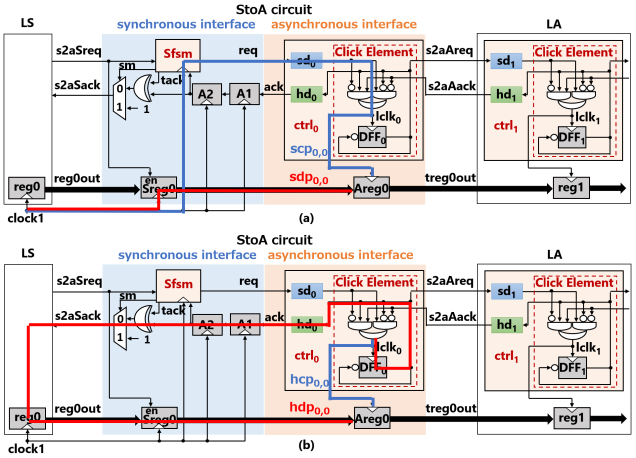


FIGURE 10. Paths related to timing constraints for *StoA* circuit: (a) setup and (b) hold constraints.

1) TIMING VERIFICATION OF THE *StoA* CIRCUIT

TimingChecker verifies the setup and hold constraints for *Areg*. In contrast, it does not verify the timing constraints for *Sreg* because the timing constraints for *Sreg* are verified by STA for *LS*.

To verify the setup constraint for *Areg*, *TimingChecker* checks whether the inequality (1) is satisfied or not. Figure 10(a) shows scp_i and sdp_i related to the setup constraint for *Areg* of the *StoA* circuit.

To verify the hold constraint for *Areg*, *TimingChecker* checks whether the inequality (8) is satisfied or not. Figure 10(b) shows hcp_i and hdp_i related to the hold constraint for *Areg* of the *StoA* circuit.

2) TIMING VERIFICATION OF THE *AtoS* CIRCUIT

TimingChecker verifies the setup and hold constraints for *Sreg* and *Areg*.

To verify the setup constraint for *Areg*, *TimingChecker* checks whether the inequality (1) is satisfied or not. Similarly, to verify the hold constraint for *Areg*, *TimingChecker* checks whether the inequality (2) is satisfied or not.

To verify the setup constraint for *Sreg*, *TimingChecker* checks whether the inequality (7) is satisfied or not. Figure 11(a) shows scp_i and sdp_i related to the setup constraint for *Sreg* of the *AtoS* circuit.

To verify the hold constraint for *Sreg*, *TimingChecker* checks whether the inequality (8) is satisfied or not. Figure 11(b) shows hcp_i and hdp_i related to the hold constraint for *Sreg* of the *AtoS* circuit.

G. DELAYADJUSTER

DelayAdjuster increases the number of cells in sd_i and hd_i if timing violations are identified by *TimingChecker*. In

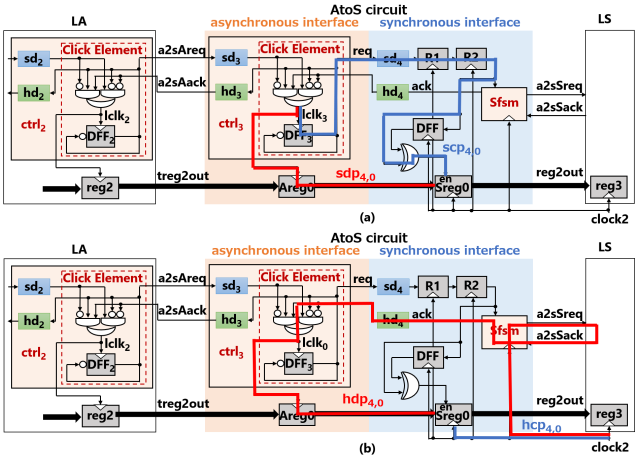


FIGURE 11. Paths related to timing constraints for *AtoS* circuit: (a) setup and (b) hold constraints.

contrast, *DelayAdjuster* decreases the number of cells in sd_i and hd_i if timing violations are not identified by *TimingChecker* and timing violations are not happened by deleting some of the cells in sd_i and hd_i.

DelayAdjuster adjusts the number of cells in sd_i and hd_i. When the setup constraint for the register which is controlled by $ctrl_i$ is not satisfied, *DelayAdjuster* adjusts the number of cells in sd_i. When the hold constraint for the register which is controlled by $ctrl_i$ is not satisfied, *DelayAdjuster* adjusts the number of cells in hd_i.

If there are timing violations for the registers, to satisfy the timing constraints, *DelayAdjuster* inserts primitive cells in sd_i and hd_i by the following equation.

$$num_{i,insert} = \lceil (-t_{i,dff})/t_{cell} \rceil \tag{9}$$

where $num_{i,insert}$ represents the number of inserted cells, and $t_{i,dff}$ represents the difference between the left and right sides of the inequalities (1), (2), (7), and (8).

If timing violations are not happened by deleting some of the cells in sd_i and hd_i, to reduce the transfer delay, *DelayAdjuster* deletes primitive cells in sd_i and hd_i by the following equation.

$$num_{i,delete} = \lceil (t_{i,dff} - t_{margin_{i,p}})/t_{cell} \rceil \tag{10}$$

where $num_{i,delete}$ represents the number of deleted cells, and $t_{margin_{i,p}}$ represents the margins for $scp_{i,p}$ and $hdp_{i,p}$. $t_{margin_{i,p}}$ are obtained by "scpm" and "hdpm" of $\langle margin \rangle$ in the Param-XML.

VI. EXPERIMENTAL RESULT

In the experiment, to clarify the quality of the *StoA* and *AtoS* circuits generated by the proposed tool set, we evaluate the latency and overhead of the generated *StoA* and *AtoS* circuits by changing cycle times. Then, to clarify that the proposed tool set can be applicable for realistic designs, we design the *StoA* and *AtoS* circuits to connect a synchronous RISC-V processor and a Multilayer Perceptron (MLP) circuit. For the experiment, we implemented the proposed tool set using Python 3.7 and Eclipse 2021-12. The tools were performed on a Windows 10 machine (Intel Core i9-10900@2.8 GHz CPU and 64 GB memory).

TABLE 1. Number of lines in all XML files and the number of delay and placement constraints generated by the proposed tool set.

Name	*Path*	*XML* [lines]	*Delay*	*DP*	*LL*
StoA	1	46	5	10	8
	2	47	6	10	10
	3	48	7	10	12
AtoS	1	46	8	14	9
	2	47	10	14	11
	3	48	12	14	13

TABLE 2. Latency and handshake overhead of the generated interface circuits.

SSct (*SAgct*)	*RSct* (*RAgct*)	*Name*	*Latency* [ns]	*Overhead* [ns]
10	10	*StoS*	50	120
		StoA	30	50
		AtoS	50	50
	20	*StoS*	90	160
		StoA	50	60
		AtoS	90	100
20	10	*StoS*	60	160
		StoA	40	80
		AtoS	60	50
	20	*StoS*	100	240
		StoA	60	100
		AtoS	100	100

First, to clarify the quality of the *StoA* and *AtoS* circuits, we generated *StoA* and *AtoS* circuits by the proposed tool set and performed RTL simulation using ModelSim-Intel FPGA Edition 2020.1. To check that the generated *StoA* and *AtoS* circuits can transfer data at different cycle times, we changed the values of "Sct" and "Agct" in the Param-XML. "Sct" and "Agct" in the Param-XML were set to 10 ns and 20 ns. For the RTL simulation, we replaced the primitive cells in sd_i to "assign" statements with delays corresponding to the values of "Sct" and "Agct". In the RTL simulation, we assume that a synchronous module sends arbitrary data to an asynchronous module through the generated *StoA* circuits. Then, the asynchronous module sends the data to the synchronous module through the generated *AtoS* circuits (i.e., round-trip data transfers). After the RTL simulation, we confirmed that all data are correctly transferred.

Table 1 represents the number of lines in all XML files and the number of constraints generated by the proposed tool set. *Name*, *Path*, *XML*, *Delay*, *DP*, and *LL* represent the interface circuit name, the number of data-paths between synchronous and asynchronous modules, the number of lines in the XMLs, the number of delay constraints, the number of Design Partitions, and the number of LogicLocks for the *StoA* or *AtoS* circuit, respectively. Table 1 represents that *XML*, *Delay*, and *LL* depend on the number of data-paths between synchronous and asynchronous modules.

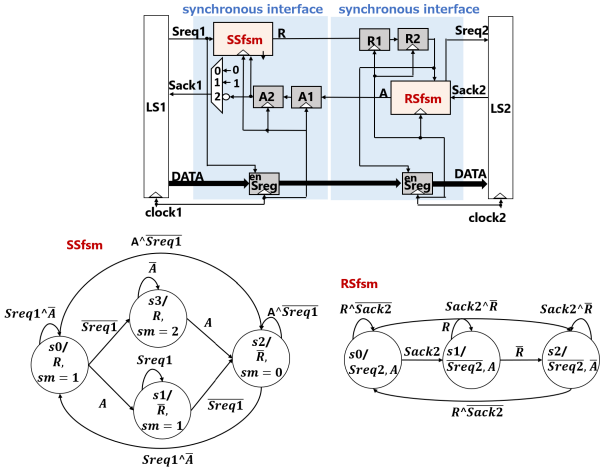


FIGURE 12. Interface circuit based on [2].

Table 2 represents the latency and handshake overhead obtained from the RTL simulation for the *StoA* and *AtoS* circuits generated by the proposed tool set when the number of data-paths between synchronous and asynchronous modules was one. The latency represents the delay until the receiver module receives the data after the sender module sends the data to the receiver module. The handshake overhead represents the delay until the sender module sends the next data to the receiver module after the sender module sends the data. *SSct* (*SAgct*), *RSct* (*RAgct*), *Name*, *Latency*, and *Overhead* represent *Sct* (*Agct*) of the sender module, *Sct* (*Agct*) of the receiver module, interface circuit name, latency, and handshake overhead, respectively. *StoS* represents a reference interface circuit based on the two-flop synchronizer as shown in Fig. 12 to evaluate the latency and handshake overhead.

If the cycle time of the receiver module was increased, the latencies of all interface circuits were increased. This is because the two-flop synchronizer was inserted on the signals for the receiver module in the cases of the *StoS* and *AtoS* circuits or the delay of sd_i was implemented based on the cycle time of the receiver module in the cases of the *StoA* circuits. Similarly, if the cycle time of the receiver module was increased, the handshake overheads of all interface circuits were increased because the sender module waits for the next data transfer until the acknowledgment signal from the receiver module is received.

Next, we connected a synchronous RISC-V processor and an asynchronous MLP circuit through the generated *StoA* and *AtoS* circuits using the proposed tool set. The MLP circuit consists of three layers. The number of neurons is 32 for each layer to classify given handwritten numbers from 0 to 9. In this system, the RISC-V processor sends two 32-bit data to the MLP circuit 13 times through the *StoA* circuit. After the inference by the MLP circuit, the MLP circuit sends the inference result (32-bit data) to the RISC-V processor through the *AtoS* circuit. The connected system is called RISC-Vs_MLPa.

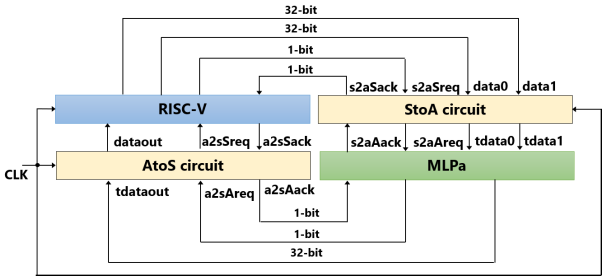


FIGURE 13. Structure of RISC-Vs_MLPa.

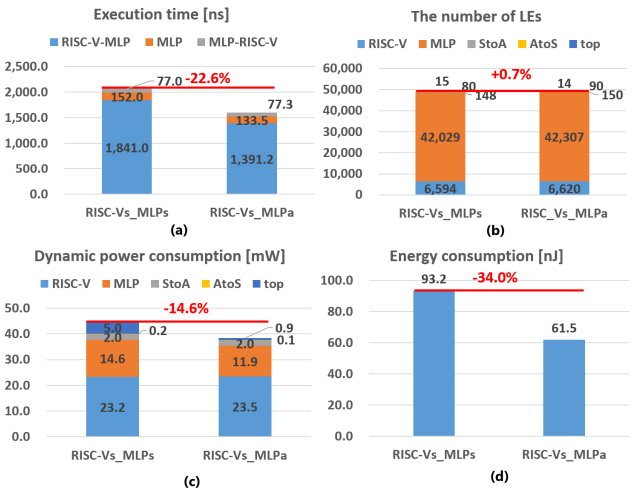


FIGURE 14. Evaluation results: (a) execution time, (b) number of LEs, (c) dynamic power consumption, and (d) energy consumption.

Figure 13 shows the structure of RISC-Vs_MLPa. The synchronous RISC-V processor was designed by referring to [31]. The asynchronous MLP circuit was designed by referring to [26] and [29]. We synthesized the RISC-Vs_MLPa using Quartus Prime 21.1. The target device was EP4CE115F29C7 (Cyclone IV E). The voltage and the temperature were set to 1.2 V and 85°. *Sct* of the synchronous RISC-V processor was set to 18 ns while *Agct* of the asynchronous MLP circuit was set to 8 ns. The number of delay adjustments by the proposed tool set was 2. For the *StoA* circuit, the number of lines in the XMLs, the number of delay constraints, the number of Design Partitions, and the number of LogicLocks were 71, 6, 10, and 10, respectively. For the *AtoS* circuit, the number of lines in the XMLs, the number of delay constraints, the number of Design Partitions, and the number of LogicLocks were 46, 8, 14, and 9, respectively.

To check the quality of the RISC-Vs_MLPa, we compared the RISC-Vs_MLPa and a RISC-Vs_MLPs which uses synchronous MLP circuit [32] and *StoS* circuits in terms of the execution time, the number of Logic Elements (LEs), the dynamic power consumption, and the energy consumption. Figure 14 shows the evaluation results.

Figure 14(a) shows the execution time of the RISC-Vs_MLPa. The execution time was obtained from the GL simulation using Modelsim. *RISC-V-MLP*, *MLP*, and

MLP-RISC-V represent the transfer time from the RISC-V processor to the MLP circuit, the inference time by the MLP circuit, and the transfer time from the MLP circuit to the RISC-V processor. Compared with the RISC-Vs_MLPs, the execution time of the RISC-Vs_MLPa was reduced by 22.6%. This reduction comes from that the handshake overhead of the *StoA* circuit was reduced compared with the *StoS* circuit because synchronizers were not used in the asynchronous interface and the asynchronous interface operated with the two-phase handshake protocol. Note that *RISC-V-MLP* was dominant in the execution time because the wait time caused by the handshake overhead of the *StoA* circuit occurred 12 times to transfer a single handwritten number.

Figure 14(b) shows the number of LEs of the RISC-Vs_MLPa. The number of LEs was obtained from the synthesis report generated by Quartus Prime. Compared with the RISC-Vs_MLPs, the number of LEs of the RISC-Vs_MLPa was increased by 0.7%. The number of LEs of the *StoA* and *AtoS* circuits was negligibly small because the *StoA* and *AtoS* circuits were composed of few components. The number of LEs of the RISC-V processor in RISC-Vs_MLPa was changed due to the different resource sharing results between the RISC-Vs_MLPs and RISC-Vs_MLPs. In addition, the number of LEs of the asynchronous MLP circuit in RISC-Vs_MLPa was increased compared to the synchronous MLP circuit in RISC-Vs_MLPs due to the insertion of asynchronous control modules with delay elements. Moreover, the number of LEs of the *StoA* and *AtoS* circuits was increased slightly compared with the *StoS* circuit because the number of LEs used in $ctrl_i$ was larger than the number of LEs used in *Sfsm* and the two-flop synchronizer.

Figure 14(c) shows the dynamic power consumption of the RISC-Vs_MLPa. The dynamic power consumption was obtained by PowerPlay Power Analyzer with the value change dump (VCD) file generated by Modelsim during GL simulation. The dynamic power consumption of the RISC-Vs_MLPa was reduced by 14.6% compared with the RISC-Vs_MLPs. This is because the dynamic power consumptions of *MLP* and *top* (the routing power of the clock network for the top-level module) were reduced because the clock power was reduced by converting the synchronous MLP circuit to the asynchronous MLP circuit. On the other hand, the *StoA* and *AtoS* circuits did not have a significant impact on the dynamic power consumption due to the small number of LEs and short latencies.

Figure 14(d) shows the energy consumption of the RISC-Vs_MLPa. The energy consumption was the product of the execution time and the dynamic power consumption. Compared with the RISC-Vs_MLPs, the energy consumption of the RISC-Vs_MLPa was reduced by 34.0% because the execution time and dynamic power consumption were reduced.

VII. CONCLUSION

In this paper, we proposed a design support tool set for interface circuits between synchronous and asynchronous modules. The proposed tool set generates RTL models and design constraints for the interface circuits. In addition, the proposed tool set performs timing verification and delay adjustment to guarantee the operations of the generated interface circuits.

In the experiment, to clarify the quality of the interface circuits generated by the proposed tool set, we evaluated the latency and overhead of the interface circuits. The latency and handshake overhead of the interface circuits generated by the proposed tool set depend on the cycle time of the receiver module. In addition, to clarify that the proposed tool set can be applicable for realistic designs, we designed a system which consists of a synchronous RISC-V processor and an asynchronous MLP circuit using the proposed tool set. The energy consumption of the system was reduced by 34.0% compared with a system which uses a synchronous MLP circuit.

As our future work, we are going to extend the interface circuits to deal with a burst transfer to reduce the handshake overhead. In addition, we are going to extend the interface circuits to deal with standard interfaces. Moreover, we are going to design the interface circuits with a clock gating to reduce the power consumption of the interface circuits. Furthermore, we are going to compare the interface circuits generated by the proposed tool set and other GALS interfaces. We will also extend the proposed tool set to deal with ASIC designs. As the device technology is different from FPGAs, the extension may clarify the usefulness of the proposed tool set more.

REFERENCES

[1] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, Tech. Rep. STAN-CS-84-1026, 1984.

[2] R. Ginosar, "Fourteen ways to fool your synchronizer," in *Proc. 9th Int. Symp. Asynchronous Circuits Syst.*, 2003, pp. 89–96.

[3] R. Ginosar, "Metastability and synchronizers: A tutorial," *IEEE Des. Test Comput.*, vol. 28, no. 5, pp. 23–35, Sep. 2011.

[4] J. Kessels, "Register-communication between mutually asynchronous domains," in *Proc. 11th IEEE Int. Symp. Asynchronous Circuits Syst.*, Mar. 2005, pp. 66–75.

[5] D. L. Oliveira, K. Garcia, L. A. Faria, J. L. V. Oliveira, and L. Romano, "A synchronous wrapper for high-speed heterogeneous systems on FPGAs," in *Proc. IEEE ANDESCON*, Oct. 2016, pp. 1–4.

[6] S. Semba and H. Saito, "A study on the design of interface circuits between synchronous-asynchronous modules using click elements," in *Proc. SASIMI*, Oct. 2022, pp. 139–144.

[7] D. L. Oliveira, T. Curtinhas, L. A. Faria, and L. Romano, "A novel asynchronous interface with pausible clock for partitioned synchronous modules," in *Proc. IEEE 6th Latin Amer. Symp. Circuits Syst. (LASCAS)*, Feb. 2015, pp. 1–4.

[8] E. Amini, M. Najibi, and H. Pedram, "Globally asynchronous locally synchronous wrapper circuit based on clock gating," in *Proc. IEEE Comput. Soc. Annu. Symp. Emerg. VLSI Technol. Archit.*, 2006, pp. 193–199.

[9] E. Amini, M. Najibi, Z. Jeddi, and H. Pedram, "FPGA implementation of gated clock based globally asynchronous locally synchronous wrapper circuits," in *Proc. Int. Symp. Signals, Circuits Syst.*, Jul. 2007, pp. 1–4.

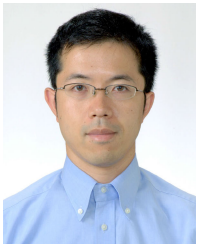
[10] S. Moore, G. Taylor, R. Mullins, and P. Robinson, "Point to point GALS interconnect," in *Proc. 8th Int. Symp. Asynchronous Circuits Syst.*, 2002, pp. 69–75.

[11] B. Keller, M. Fojtik, and B. Khailany, "A pausible bisynchronous FIFO for GALS systems," in *Proc. 21st IEEE Int. Symp. Asynchronous Circuits Syst.*, May 2015, pp. 1–8.

[12] E. Beigne and P. Vivet, "Design of on-chip and off-chip interfaces for a GALS NoC architecture," in *Proc. 12th IEEE Int. Symp. Asynchronous Circuits Syst.*, Mar. 2006, pp. 172–183.

[13] T. Ono and M. Greenstreet, "A modular synchronizing FIFO for NoCs," in *Proc. 3rd ACM/IEEE Int. Symp. Networks-Chip*, May 2009, pp. 224–233.
[14] F. Huemer and A. Steininger, "Timing domain crossing using Müller pipelines," in *Proc. 26th IEEE Int. Symp. Asynchronous Circuits Syst. (ASYNC)*, May 2020, pp. 44–53.
[15] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 8, pp. 857–873, Aug. 2004.
[16] A. M. S. Abdelhadi, "Synthesizable synchronization FIFOs utilizing the asynchronous pulse-based handshake protocol," in *Proc. IEEE Nordic Circuits Syst. Conf. (NorCAS)*, Oct. 2020, pp. 1–7.
[17] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Proc. 6th Int. Symp. Adv. Res. Asynchronous Circuits Syst.*, 2000, pp. 52–59.
[18] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits: Part I (specification and hazard-free implementation) and Part II (automatic synthesis)," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 2, pp. 101–132, Feb. 1999.
[19] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. Inf. Syst.*, vol. 80, no. 3, pp. 315–325, 1997.
[20] D. E. Müller and W. S. Bartky, "A theory of asynchronous circuits," in *Proc. Int. Symp. Theory Switching*, 1959, pp. 204–243.
[21] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana, "MINIMALIST: An environment for synthesis, verification and testability of burst-mode asynchronous machines," Dept. Comput. Sci., Columbia Univ., New York, NY, USA, Tech. Rep., CUCS-020-99, 1999.
[22] A. Peeters, F. T. Beest, M. de Wit, and W. Mallon, "Click elements: An implementation style for data-driven compilation," in *Proc. IEEE Symp. Asynchronous Circuits Syst.*, May 2010, pp. 3–14.
[23] A. Branover, R. Kol, and R. Ginosar, "Asynchronous design by conversion: Converting synchronous circuits into asynchronous ones," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 2004, pp. 870–875.
[24] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 25, no. 10, pp. 1904–1921, Oct. 2006.
[25] N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, "A fully-automated desynchronization flow for synchronous circuits," in *Proc. 44th ACM/IEEE Design Autom. Conf.*, Jun. 2007, pp. 982–985.
[26] S. Semba and H. Saito, "RTL conversion method from pipelined synchronous RTL models into asynchronous ones," *IEEE Access*, vol. 10, pp. 28949–28964, 2022.
[27] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC flow for GHz asynchronous designs," *IEEE Des. Comput.*, vol. 28, no. 5, pp. 36–51, Sep./Oct. 2011.
[28] A. Taubin, *Design Automation of Real-Life Asynchronous Devices and Systems*. Norwell, MA, USA: Now, 2007.
[29] K. Takizawa, S. Hosaka, and H. Saito, "A design support tool set for asynchronous circuits with bundled-data implementation on FPGAs," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 232–235.
[30] T. Otake and H. Saito, "A design method for designing asynchronous circuits on commercial FPGAs using placement constraints," *IEICE Trans. Fundamentals Electron., Commun. Comput. Sci.*, vol. E103.A, no. 12, pp. 1427–1436, 2020.
[31] J. Miura, H. Miyazaki, and K. Kise, "A portable and Linux capable RISC-V computer system in Verilog HDL," 2020, *arXiv:2002.03576*.
[32] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 65–74.

SHOGO SEMBA (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from The University of Aizu, in 2017, 2019, and 2022, respectively. He is currently a Postdoctoral Researcher with The University of Aizu. His research interests include asynchronous circuit design and the Internet of Things (IoT) systems.



HIROSHI SAITO (Member, IEEE) received the B.S. and M.S. degrees in computer science and engineering from The University of Aizu, in 1998 and 2000, respectively, and the Ph.D. degree in electronic engineering from The University of Tokyo, in 2003. He is a Senior Associate Professor with The University of Aizu. His research interests include asynchronous circuit design, multi-core system design, and application of sensor networks.