Received January 25, 2022, accepted March 7, 2022, date of publication March 10, 2022, date of current version March 18, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3158487

RTL Conversion Method From Pipelined Synchronous RTL Models Into Asynchronous Ones

SHOGO SEMBA[1], (Member, IEEE), AND HIROSHI SAITO[2], (Member, IEEE)
[1]Graduate School of Computer Science and Engineering, The University of Aizu, Aizu-Wakamatsu 965-8580, Japan
[2]School of Computer Science and Engineering, The University of Aizu, Aizu-Wakamatsu 965-8580, Japan

Corresponding author: Shogo Semba (d8211108@u-aizu.ac.jp)

This work was supported by the Grant-in-Aid for Scientific Research from the Japan Society for the Promotion of Science under Grant 21K11812 and Grant 20J11724.

ABSTRACT In this paper, we propose a conversion method from pipelined synchronous Register Transfer Level (RTL) models into pipelined asynchronous RTL models with bundled-data implementation. To know data-path resources controlled by each pipeline stage, the proposed method generates a control data flow graph (CDFG) from synchronous RTL models. After generating the CDFG, the proposed method generates asynchronous RTL models by analyzing each pipeline stage on the CDFG, assigning asynchronous control modules, and connecting the control modules to the data-path resources. In addition, we also propose optimization methods during the conversion. In the experiment, we converted four pipelined synchronous RTL models into pipelined asynchronous ones. In addition, we performed logic synthesis for the converted asynchronous RTL models to check the quality of the asynchronous RTL models. The synthesized asynchronous circuits without the optimization methods could reduce the energy consumption by 1.47% on average compared to synchronous circuits. Moreover, the optimization methods could reduce the energy consumption by 15.12% on average compared to synchronous circuits. Furthermore, the optimization methods reduced the energy consumption by up to 34.72% compared to asynchronous circuits without the optimization methods.

INDEX TERMS Asynchronous circuits, RTL models, conversion, low power.

I. INTRODUCTION

Most of the digital integrated circuits used in computer systems are synchronous circuits. In synchronous circuits, circuit components are controlled by global clock signals. When the semiconductor miniaturization technology is advanced more and more, the power consumption of clock networks becomes high because clock signals with high frequency are distributed to a wide area.

In asynchronous circuits, circuit components are controlled by local handshake signals or self-timed signals instead of global clock signals. Therefore, asynchronous circuits are potentially low power consumption and low electromagnetic interference compared to synchronous circuits. However, the design of asynchronous circuits is more difficult than the design of synchronous circuits. According to the selection of data encoding, handshake protocol, and delay model, design methods and design constraints are different. In addition, Electronic Design Automation (EDA) tools to support the design of asynchronous circuits are insufficient.

To facilitate the design of asynchronous circuits, conversion methods from synchronous Gate-Level (GL) netlists into asynchronous ones were proposed in [1]–[5]. In the GL conversion, D flip-flops (DFFs) in synchronous GL netlists synthesized by a synthesis tool are converted into master-slave latches. The converted latches are controlled by inserting latch controllers based on local handshake signals. However, logic optimization considering the characteristics of asynchronous circuits cannot be performed, because logic synthesis is performed for synchronous Register Transfer Level (RTL) models with a clock constraint.

We proposed a conversion method from synchronous RTL models into asynchronous ones in [22]. Compared to the GL conversion methods, we can optimize asynchronous circuits.

The associate editor coordinating the review of this manuscript and approving it for publication was Gian Domenico Licciardo.

For example, operations at each cycle can be executed at their delay using local handshake signals or self-timed signals in asynchronous circuits. For the GL conversion methods, the delays of operations at each cycle are equalized because logic synthesis is performed for synchronous RTL models with a clock constraint. For the RTL conversion method, the delays of operations at each cycle can be changed because logic synthesis is performed for asynchronous RTL models with different delay constraints at each cycle.

However, the RTL conversion method in [22] cannot deal with pipelined synchronous RTL models. Actually, pipelined synchronous circuits are used to improve the performance in many applications. Hence, by converting pipelined synchronous RTL models into pipelined asynchronous ones, we can design low power circuits more than pipelined synchronous circuits.

In this paper, we propose a conversion method from pipelined synchronous RTL models into pipelined asynchronous RTL models. The proposed method is an extension of [22]. To know which data-path resources in pipelined synchronous RTL models are controlled at each pipeline stage, we extend [22] to generate a control data flow graph (CDFG) from pipelined synchronous RTL models. In addition, to deal with pipeline stalls, the proposed method generates a CDFG including conditions that pipeline stages stall the operations. After generating the CDFG, the proposed method assigns asynchronous control modules by analyzing each pipeline stage in the CDFG. Then, the proposed method generates asynchronous RTL models by connecting the control modules to the data-path resources.

On the other hand, the quality of asynchronous circuits from the RTL conversion depends on the representation style of the synchronous RTL models. To obtain the high quality of asynchronous circuits, we also propose optimization methods during the RTL conversion. The optimization methods are based on [24]. In addition, in this paper, we propose a new optimization method to convert DFFs into D latches to reduce the dynamic power consumption of registers.

The main contributions of this paper are as follows.

1) The proposed method converts pipelined synchronous RTL models to pipelined asynchronous RTL ones, enabling different input intervals and pipeline stalls.
2) The proposed method generates optimized asynchronous RTL models in terms of circuit area, dynamic power consumption, and energy consumption.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 describes asynchronous circuits with bundled-data implementation used in this work. Section 4 describes the overview of the RTL conversion method proposed in [22]. Section 5 describes the proposed RTL conversion method for pipelined synchronous RTL models. Section 6 describes the experimental results. Finally, section 7 describes the conclusion and future work.

II. RELATED WORK

To design asynchronous circuits, 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.

Conversion methods from synchronous GL netlists into asynchronous GL netlists with bundled-data implementation were proposed in [1]–[5]. Branover *et al.* [1] replaces each register in synchronous GL netlists synthesized by a commercial synthesis tool into a pair of latches with corresponding latch controllers based on Doubly-Latched Asynchronous Pipeline (DLAP). Similarly, [2]–[4] are approaches to replace DFFs in synchronous GL netlists into master-slave latches with corresponding latch controllers. These methods are called *Desynchronization*. A tool which automatically performs *Desynchronization* was developed in [5].

On the other hand, conversion methods from synchronous GL netlists into dual-rail asynchronous GL netlists were proposed in [6]–[10]. Zhou *et al.* [6] generates dual-rail asynchronous circuits by replacing gates in synchronous GL netlists with gates in an asynchronous library. A toolset called *Uncle* (Unified NULL Convention Logic Environment) which generates asynchronous circuits based on Null Convention Logic (NCL) [11], [12] was proposed in [7]. *Uncle* generates asynchronous circuits by expanding single-rail netlists to dual-rail netlists and generating acknowledgment networks. Similarly, design flows for asynchronous circuits based on NCL were proposed in [8] and [9]. In addition, [10] is a compiler to generate Quasi-Delay-Insensitive (QDI) circuits using NCL gates or Differential Cascade Voltage Swing Logic (DCVSL) gates.

Compared to [1]–[10] where the GL conversion is the target, we focus on the RTL conversion. The RTL conversion has advantages compared to the GL conversion. As an example, we can generate optimized asynchronous circuits (e.g., performance, area, or power consumption) by performing logic synthesis assigning path delay constraints for asynchronous RTL models.

In contrast, a synthesis flow called *Pulsar-F* which generates asynchronous circuits based on QDI circuits was proposed in [13]. Sartori *et al.* [13] is based on the Pulsar flow described in [14]. *Pulsar-F* accepts RTL models which are treated using commercial synthesis tools such as Cadence Genus. It can automatically generate optimized asynchronous circuits from RTL models and cycle time constraints. However, to achieve better performance and area results for QDI circuits, it uses a specific cell library. Compared to [13], [14], in this paper, the target asynchronous circuit is bundled-data implementation and the target library is a standard cell library.

In addition, as related work, design flows for asynchronous circuits were proposed in [15], [16], and [21]. In [15], a design environment called *Proteus* was proposed. *Proteus* synthesizes synchronous GL netlists from synchronous RTL models obtained from Communicating Sequential Processes (CSP) models. Then, *Proteus* translates the synchronous GL netlists to asynchronous GL netlists. In [16], a design environment called *TiDE* was proposed. *TiDE* synthesizes handshake circuits from high-level languages called *Haste*. Then, *TiDE* generates asynchronous circuits from the handshake circuits.

Furthermore, synthesis methods for asynchronous RTL models from VHDL behavior models were proposed in [17]–[19]. Garcia *et al.* [17] generates locally-clocked asynchronous circuits by generating data-path resources and asynchronous controllers based on Extended Burst-Mode (XBM) [20]. A tool called *VHDLASYN* which automatically generates an asynchronous RTL model using [17] was developed in [18]. On the other hand, a synthesis system to synthesize an asynchronous circuit called *MOODs* was proposed in [19]. For a VHDL behavior model, *MOODs* generates an asynchronous RTL model by operation scheduling and resource allocation.

In [15]–[19], the target is not the conversion from synchronous circuits to asynchronous circuits. These methods directly generate asynchronous circuits from behavioral models through operation scheduling of asynchronous circuits and synthesis of asynchronous controllers. In this paper, we focus on the conversion from synchronous circuits into asynchronous circuits.

As another research, a design flow for asynchronous circuits was proposed in [21]. The design flow accepts a design language called ACT (for asynchronous circuit toolkit). The ACT supports representing circuits at several levels such as Communicating Hardware Process (CHP), handshaking expansion (HSE), GL, and transistor-level. For example, a CHP description is transformed into a GL description. Compared to [21], we focus on the conversion from synchronous circuits into asynchronous circuits at RTL.

This paper is an extension of [22]–[24]. Semba and Saito [22] converts non-pipelined synchronous RTL models into non-pipelined asynchronous ones. To deal with various synchronous RTL models, the RTL conversion method generates intermediate representations from the synchronous RTL models. Then, the RTL conversion method generates the asynchronous RTL models from the intermediate representations. However, there is a problem that the RTL conversion method cannot deal with pipelined synchronous RTL models. To solve this problem, [23] converts pipelined synchronous RTL models into pipelined asynchronous ones. However, there is a restriction that the input interval of pipeline circuits is one cycle in [24]. In this paper, there is no such restriction. On the other hand, [24] proposed optimization methods during the RTL conversion [22] to obtain the high quality of asynchronous circuits. In this paper, we also propose a conversion from DFFs to D latches to optimize the area of registers which is not described in [24].

III. ASYNCHRONOUS CIRCUITS WITH BUNDLED-DATA IMPLEMENTATION

Bundled-data implementation is one of the data encoding schemes in asynchronous circuits. Figure 1 shows asynchronous circuits with bundled-data implementation. In the

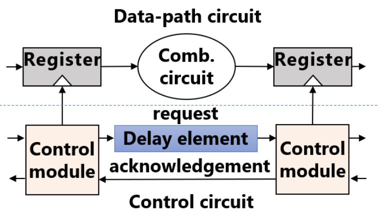


FIGURE 1. Asynchronous circuits with bundled-data implementation.

bundled-data implementation, a one-bit signal is represented by one signal. The timing for writing data to registers is guaranteed by delay elements on request signals in a control circuit. Hence, the performance of the bundled-data implementation depends on the delay of the data-path, and the control-path is delay-matched to the data-path.

A. CIRCUIT MODEL USED IN THIS WORK

Figure 2(a) shows the circuit model of bundled-data implementation used in this work. This circuit model consists of a data-path circuit and a control circuit.

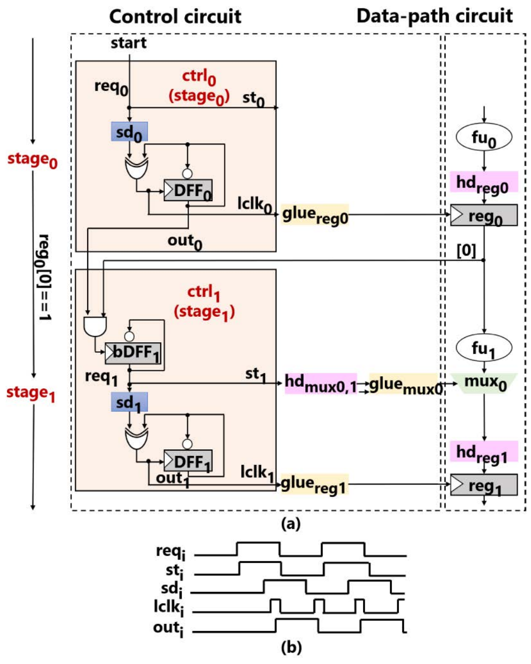


FIGURE 2. Asynchronous circuits with bundled-data implementation used in this work: (a) circuit model and (b) timing diagram of $ctrl_i$.

The data-path circuit is almost the same as the one used in synchronous circuits. It consists of registers reg_k, multiplexers mux_l, and functional units fu_h. Note that a delay element hd_{reg_k} is inserted on the input signal of reg_k if hold violations occur on reg_k. In the traditional bundled-data implementation, delay elements are not used in the data-path because the operation of the data-path is guaranteed by request and acknowledgment signals. In this work, the acknowledgment signal is not used to improve the performance of the bundled-data implementation. We assume that the bundled-data implementation can start its operation without waiting for the acknowledgment signal.

The control circuit is based on pipeline stages in synchronous circuits. It consists of control modules $ctrl_i$ ($0 \leq i \leq n-1$) assigned for each pipeline stage $stage_i$. Glue logics $glue_{reg_k}$ and $glue_{mux_l}$ are logics to control reg_k and mux_l. If hold violations occur on reg_k by a transition of the control signal for mux_l, a delay element $hd_{mux_{i,l}}$ is inserted on the control signal for mux_l.

$ctrl_i$ is obtained by modifying a Click element [25]. It consists of a DFF DFF_i, an XOR gate, and a delay element sd_i. When there are control branches, a DFF $bDFF_i$ and an AND gate are inserted before sd_i. When $ctrl_i$ requires several request signals, an XOR gate is inserted before sd_i. The acknowledgment signal used in traditional asynchronous circuits is not used in $ctrl_i$. Only the request signal req_i is used for succeeding control modules. Hence, each $ctrl_i$ is operated by self-timing using sd_i which guarantees setup constraints for reg_k. $ctrl_i$ is operated by the rising transition and falling transition of req_i. Data are written to reg_k by the rising transition of $lclk_i$. Hence, the performance of the bundled-data implementation used in this work depends on the delay of the control circuit including the delay elements.

The control circuit starts its operation when a rising transition of the input signal $start$ arrives at the control circuit. Figure 2(b) shows the timing diagram of $ctrl_i$. $ctrl_i$ starts its operation when a rising transition of out_{i-1} or $lclk_{i-1}$ from $ctrl_{i-1}$ arrives at $ctrl_i$. The signal transition generates a rising transition of req_i. Then, req_i generates a rising transition of st_i. st_i controls mux_l through $glue_{mux_l}$. req_i also generates a rising transition of $lclk_i$ through sd_i and the XOR gate. $lclk_i$ controls reg_k through $glue_{reg_k}$ and DFF_i. DFF_i generates a rising transition of out_i to pass the control to $ctrl_{i+1}$. Finally, $ctrl_i$ generates a falling transition of $lclk_i$ by using out_i. Note that the behavior of $ctrl_i$ in the case of the falling transition of req_i is the same as the case of the rising transition of req_i.

B. TIMING CONSTRAINTS USED IN THIS WORK

In asynchronous circuits with bundled-data implementation used in this work, 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 [30]. In this sub-section, we describe the setup and hold constraints for pipelined circuits.

Before the explanation for the timing constraints, we define a local cycle time (lct) and a global cycle time (gct). lct_i represents a maximum delay for operating $stage_i$ while gct represents the cycle time in asynchronous circuits.

Figure 3 represents paths related to lct_i. lct_i and gct can be represented by the following equations.

$$lct_i = max(t_{maxcp_{i,p}} - t_{maxlclktoreg_{i,p}}, \cdots, t_{maxcp_{i,q}} - t_{maxlclktoreg_{i,q}}) \quad (1)$$

$$gct = max(lst_0, \cdots, lst_{n-1}) \quad (2)$$

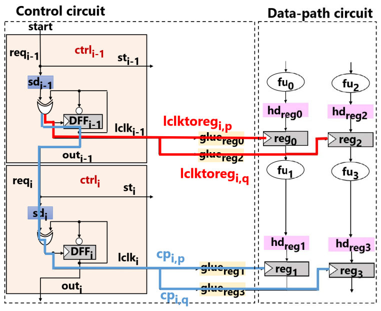


FIGURE 3. Paths related to a local cycle time.

p and q represent the identifier of paths. $t_{maxcp_{i,p}}$ represents the maximum delay of a control-path $cp_{i,p}$ from $lclk_{i-1}$ to the destination register through sd_i. $t_{maxlclktoreg_{i,p}}$ represents the maximum delay of a path from $lclk_{i-1}$ to the source register. lct_i is the largest value of $t_{maxcp_{i,p}}$ minus $t_{maxlclktoreg_{i,p}}$ in $stage_i$. gct is the maximum value of lct_i.

1) SETUP CONSTRAINT

The input data for the register 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.

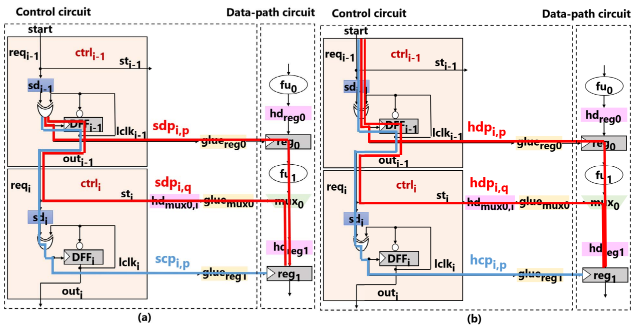


FIGURE 4. Timing constraints for registers: (a) setup constraint and (b) hold constraint.

Figure 4(a) shows data-paths $sdp_{i,p}$ and $sdp_{i,q}$ and a control-path $scp_{i,p}$ related to the setup constraint. $sdp_{i,p}$ ($sdp_{i,q}$) represents a data-path from the output of $lclk_{i-1}$ to the destination register reg_1 through reg_0 ($glue_{mux_0}$). $scp_{i,p}$ represents a control-path from the output of $lclk_{i-1}$ to the destination register reg_1 through sd_i. We define the maximum 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 the destination register as $t_{setup_{i,p}}$. Thus, the setup constraint can be represented by the following inequality.

$$t_{minscp_{i,p}} > t_{maxsdp_{i,p}} + t_{sdpm_{i,p}} + t_{setup_{i,p}} \quad (3)$$

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

2) HOLD CONSTRAINT

The data must be stable for the hold time after the next input data are written to the register reg_k. This is called the hold constraint for reg_k.

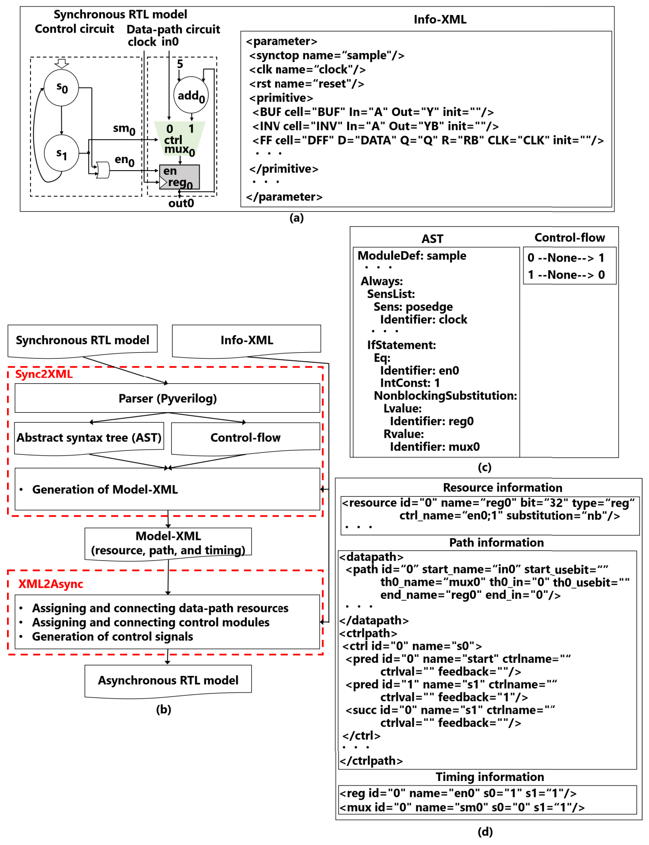
Figure 4(b) shows data-paths $hdp_{i,p}$ and $hdp_{i,q}$ and a control-path $hcp_{i,p}$ related to the hold constraint. $hdp_{i,p}$ ($hdp_{i,q}$) represents a data-path from the input signal *start* to the destination register reg_1 through reg_0 ($glue_{mux_0}$). $scp_{i,p}$ represents a control-path from the input signal *start* to the destination register reg_1 through sd_i. 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 $t_{hcpm_{i,p}}$, the hold time of the destination register as $t_{hold_{i,p}}$, and the input interval as II. Then, the hold constraint can be represented by the following inequality.

$$t_{minhdp_{i,p}} + gct \times II > t_{maxhcp_{i,p}} + t_{hcpm_{i,p}} + t_{hold_{i,p}} \tag{4}$$

If the hold constraint is violated, we need to adjust the number of cells for hd_{reg_k} or $hd_{mux_{i,l}}$.

IV. RTL CONVERSION

The RTL conversion method in [22] generates non-pipelined asynchronous RTL models from non-pipelined synchronous RTL models through *Sync2XML* and *XML2Async*. The RTL conversion method takes a parameter file called Info-eXtensible Markup Language (XML) as another input. The Info-XML consists of a top-level module name, a global clock signal name, and so on. Figure 5(a) shows the structure of a synchronous RTL model and a part of the Info-XML. Figure 5(b) shows the RTL conversion flow in [22].



```
<parameter>
 <synctop name="sample"/>
 <clk name="clock"/>
 <rst name="reset"/>
 <primitive>
  <BUF cell="BUF" In="A" Out="Y" init=""/>
  <INV cell="INV" In="A" Out="YB" init=""/>
  <FF cell="DFF" D="DATA" Q="Q" R="RB" CLK="CLK" init=""/>
  . . .
 </primitive>
 . . .
</parameter>
```

```
ModuleDef: sample
  . . .
 Always:
  SensList:
   Sens: posedge
    Identifier: clock
   . . .
  IfStatement:
   Eq:
    Identifier: en0
    IntConst: 1
   NonblockingSubstitution:
     Lvalue:
      Identifier: reg0
     Rvalue:
      Identifier: mux0
```

```
0 --None--> 1
1 --None--> 0
```

```
<resource id="0" name="reg0" bit="32" type="reg"
          ctrl_name="en0;1" substitution="nb"/>
. . .
```

```
<datapath>
 <path id="0" start_name="in0" start_usebit=""
       th0_name="mux0" th0_in="0" th0_usebit=""
       end_name="reg0" end_in="0"/>
 . . .
</datapath>
<ctrlpath>
 <ctrl id="0" name="s0">
  <pred id="0" name="start" ctrlname=""
        ctrlval="" feedback=""/>
  <pred id="1" name="s1" ctrlname=""
        ctrlval="" feedback="1"/>
  <succ id="0" name="s1" ctrlname=""
        ctrlval="" feedback=""/>
 </ctrl>
 . . .
</ctrlpath>
```

```
<reg id="0" name="en0" s0="1" s1="1"/>
<mux id="0" name="sm0" s0="0" s1="1"/>
```

FIGURE 5. RTL conversion method in [22]: (a) synchronous RTL model and Info-XML, (b) RTL conversion flow, (c) AST and control-flow, and (d) Model-XML.

Sync2XML generates the abstract syntax tree (AST) and the control-flow from given synchronous RTL models through *Pyverilog* [26]. Figure 5(c) shows a part of the AST and the control-flow for the synchronous RTL model in Fig.5(a). The AST represents the structure of RTL models. "Lvalue" and "Rvalue" represent the left side and right side of an assignment statement. "IfStatement" and "CaseStatement" represent "if" statement and "case" statement. On the other hand, the control-flow represents state transitions of RTL models. Values described by decimal numbers represent state variables. Arrows represent state transitions and the conditions of the state transition. Note that *Pyverilog* generates the control-flow only when there are finite state machines in synchronous RTL models.

After generating the AST and control-flow, *Sync2XML* generates an intermediate representation called Model-XML from the AST and control-flow. The Model-XML consists of data-path resource information, path information, and timing information as shown in Fig.5(d). In the data-path resource information, $\langle resource \rangle$ represents a data-path resource in synchronous RTL models. In the path information, $\langle path \rangle$ represents a data-path while $\langle ctrl \rangle$ represents a control-path. $\langle ctrl \rangle$ includes $\langle pred \rangle$ and $\langle succ \rangle$. $\langle pred \rangle$ represents a preceding control state while $\langle succ \rangle$ represents a succeeding control state. In the timing information, $\langle reg \rangle$ represents a register write signal name and its value while $\langle mux \rangle$ represents a multiplexer control signal name and its value.

After generating the Model-XML, *XML2Async* generates asynchronous RTL models from the Model-XML. *XML2Async* assigns and connects data-path resources by referring to the resource and path information. Then, *XML2Async* assigns and connects control modules by referring to the path information. Finally, *XML2Async* connects the control modules to the data-path resources by referring to the timing information.

V. PROPOSED METHOD

To deal with pipelined synchronous RTL models, we extend the RTL conversion method described in Sec. IV. Figure 6 shows the extended RTL conversion flow. The bold types represent extensions. The extensions are a generation of a CDFG and an analysis of pipeline stages in *Sync2XML*. The other extensions are assigning and connecting control modules and a generation of register write signals and multiplexer control signals in *XML2Async*.

A. TARGET PIPELINED SYNCHRONOUS RTL MODELS

There are restrictions of pipelined synchronous RTL models. The proposed method assumes that the data-path circuit is composed of functional units, registers, and multiplexers as described in Sec. III. The proposed method assumes that the target synchronous RTL models have only one control circuit. The control circuit must be represented by a finite state machine (FSM) or registers to control pipeline stages.

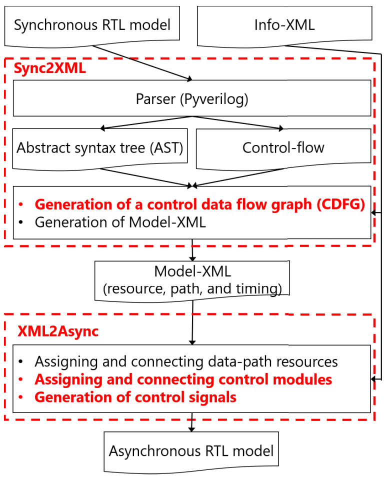

Synchronous RTL model
Info-XML
Sync2XML
Parser (Pyverilog)
Abstract syntax tree (AST)
Control-flow
• Generation of a control data flow graph (CDFG)
• Generation of Model-XML
Model-XML
(resource, path, and timing)
XML2Async
• Assigning and connecting data-path resources
• Assigning and connecting control modules
• Generation of control signals
Asynchronous RTL model

FIGURE 6. **Extended RTL conversion flow.**

The proposed method assumes that the target synchronous RTL models are described by Verilog Hardware Description Language (HDL). Syntax such as "function", "task", "for", "while", "wait", and "[sub,5'h0+:32] (concatenation)" must not be included in Verilog HDL. We are going to deal with the syntax in our future work.

On the other hand, the proposed method can deal with several synchronous RTL models. For example, it can deal with synchronous RTL models regardless of whether clock gating for registers is performed. In addition, it can deal with synchronous RTL models even if the number of cycles for the input interval (II) in pipelined circuits is changed.

Figure 7 shows examples of pipelined synchronous RTL models. If pipeline stalls are included in synchronous RTL models, pipeline stages stall the operation during the stall. When the II is two cycles or more, resources can be shared by multiplexers.

B. EXTENSION OF Sync2XML

To convert pipelined synchronous RTL models into asynchronous ones, we must know which data-path resources in pipelined synchronous RTL models are controlled at each $stage_i$. This is because data-path resources at each $stage_i$ are controlled by each $ctrl_i$ in pipelined asynchronous circuits.

To know the data-path resources controlled by each $stage_i$, we extend *Sync2XML* to generate a CDFG from the AST and control-flow generated by *Pyverilog*. By analyzing each $stage_i$ in the CDFG, we can know the data-path resources controlled by each $stage_i$. Then, *Sync2XML* generates the Model-XML by extracting pipeline stage information and timing information.

In addition, we extend *Sync2XML* to deal with different IIs which is not described in [23]. When the input interval is two or more, pipelined circuits can include multiplexers to share registers and functional units at different pipeline stages. The multiplexers and shared registers are controlled by several pipeline stages. Therefore, the analysis

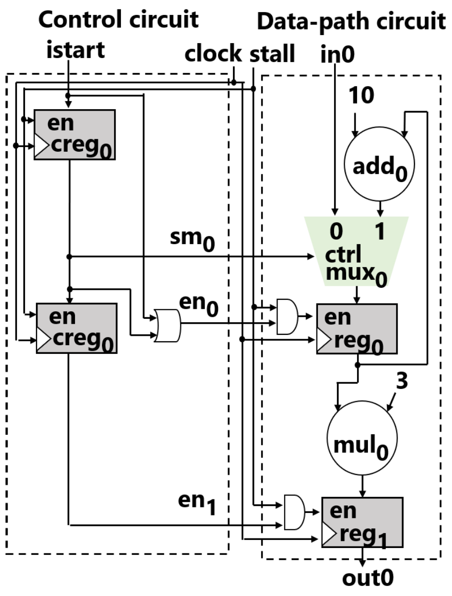


FIGURE 7. **Example of pipelined synchronous RTL models.**

of pipelined synchronous RTL models will be complex to know multiplexers and registers controlled at each pipeline stage. In this paper, to know them, the proposed method generates a CDFG including multiplexers and shared registers at different pipeline stages. The generation of such CDFG is not described in [23]. Note that we consider applications where the input interval is fixed. We do not consider applications where the input interval can vary in time. To deal with such applications is our future work.

1) GENERATION OF A CDFG

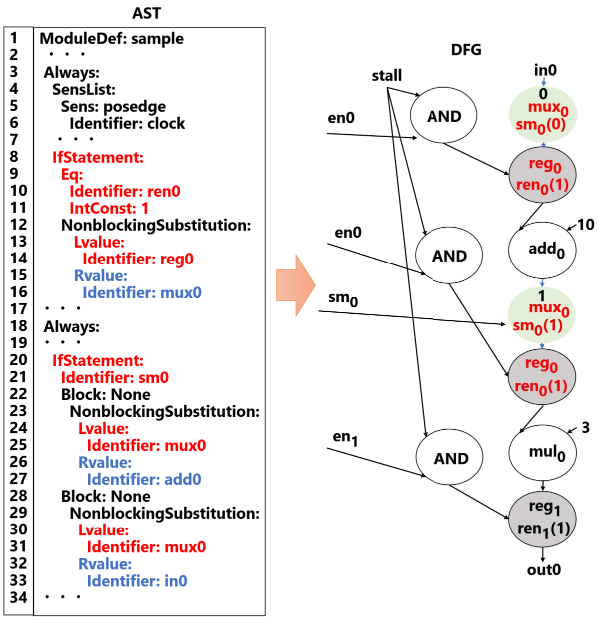
The CDFG used in this work represents the control flow and data flow in synchronous RTL models. The CDFG is a combination of the control flow graph (CFG) and data flow graph (DFG). The CFG represents a flow of the control circuit while the DFG represents a flow of the data-path circuit. The CDFG consists of nodes, edges, and $stage_i$ as shown in Fig.10. The nodes represent resources such as registers, multiplexers, functional units, and basic logic operations. The nodes except functional units and basic logic operations have a control signal name and its value (the underside of the nodes in Fig.10). The edges represent a connection between resources. Between registers represents $stage_i$. $stage_i$ has a conditional signal *cond* and its value *val* to start the operation in $stage_i$.

For a DFG, *Sync2XML* generates nodes and edges from the AST. *Sync2XML* generates nodes from "Lvalue" in the AST and edges from "Rvalue" in the AST. *Sync2XML* also extracts the label for the nodes from the variable name for "Lvalue" and the control signal with its value for the nodes from "IfStatement" or "CaseStatement" in the AST.

When the II is two cycles or more, *Sync2XML* generates nodes of multiplexers to share registers and functional units.

The same nodes are generated multiple times. On the other hand, *Sync2XML* generates appropriate edges for multiplexers from "Rvalue", "IfStatement", and "CaseStatement" in the AST.

Figure 8 shows the generated DFG from the AST for Fig.7. The red color represents the relationship between the AST and nodes for the DFG. For example, node reg_0 is generated from "Lvalue" at line 13 in the AST. In addition, control signal ren_0 is extracted in reg_0 from "IfStatement" at line 8 in the AST. The blue color represents the relationship between the AST and edges for the DFG. For example, the edge from mux_0 to reg_0 is generated from "Rvalue" at line 15 in the AST. Note that reg_0 and mux_0 are generated two times because reg_0 is shared by mux_0. The edges of mux_0 are generated from "IfStatement" at line 20, "Rvalue" at line 26, and "Rvalue" at line 33 in the AST.

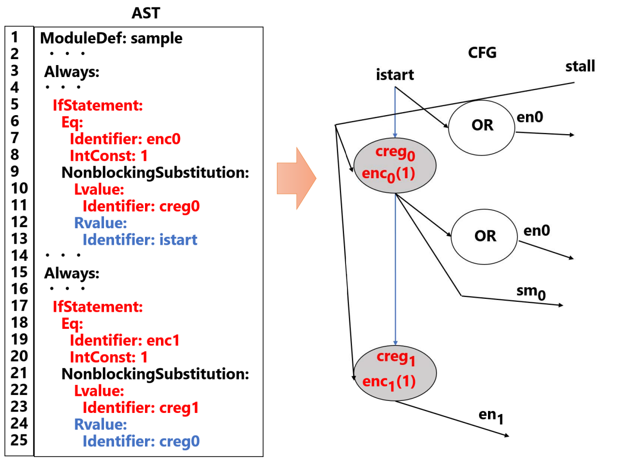

AST

```
1  ModuleDef: sample
2   • • •
3   Always:
4     SensList:
5       Sens: posedge
6         Identifier: clock
7       • • •
8     IfStatement:
9       Eq:
10        Identifier: ren0
11        IntConst: 1
12      NonblockingSubstitution:
13        Lvalue:
14          Identifier: reg0
15        Rvalue:
16          Identifier: mux0
17  • • •
18  Always:
19  • • •
20    IfStatement:
21      Identifier: sm0
22      Block: None
23        NonblockingSubstitution:
24          Lvalue:
25            Identifier: mux0
26          Rvalue:
27            Identifier: add0
28      Block: None
29        NonblockingSubstitution:
30          Lvalue:
31            Identifier: mux0
32          Rvalue:
33            Identifier: in0
34  • • •
```

FIGURE 8. DFG for the synchronous RTL model in Fig.7.

For a CFG, *Sync2XML* generates nodes and edges from the AST or control-flow. The generation method for nodes and edges differs depending on whether there is a control-flow. For a control-flow, *Sync2XML* generates nodes and edges from the control-flow. *Sync2XML* extracts the label for the nodes from values described by decimal numbers and the control signal with its value from the arrows in the control-flow. *Sync2XML* also generates edges from arrows in the control-flow. In the absence of a control-flow, *Sync2XML* generates nodes from "Lvalue" for the control circuit in the AST. *Sync2XML* extracts the label for the nodes from the variable name for "Lvalue" and the control signal with its value for the nodes from "IfStatement" or "CaseStatement" in the AST. *Sync2XML* also generates edges from "Rvalue" in the AST.

Figure 9 shows the generated CFG from the AST for the synchronous RTL model in Fig.7. In this example, the CFG


AST

```
1  ModuleDef: sample
2   • • •
3   Always:
4   • • •
5    IfStatement:
6      Eq:
7        Identifier: enc0
8        IntConst: 1
9      NonblockingSubstitution:
10       Lvalue:
11         Identifier: creg0
12       Rvalue:
13         Identifier: istart
14  • • •
15  Always:
16   • • •
17   IfStatement:
18     Eq:
19       Identifier: enc1
20       IntConst: 1
21     NonblockingSubstitution:
22       Lvalue:
23         Identifier: creg1
24       Rvalue:
25         Identifier: creg0
```

FIGURE 9. CFG for the synchronous RTL model in Fig.7.

is generated from the AST. A control-flow is not generated by *Pyverilog* because the synchronous RTL model does not have FSMs. The red color represents the relationship between the AST and nodes for the CFG. For example, node $creg_0$ is generated from "Lvalue" at line 10 in the AST. In addition, control signal enc_0 is extracted in $creg_0$ from "IfStatement" at line 5 in the AST. The blue color represents the relationship between the AST and edges for the CFG. For example, the edge from *istart* to $creg_0$ is generated from "Rvalue" at line 12 in the AST.

Then, *Sync2XML* regards between registers as $stage_i$. The extraction method for *cond* and *val* for $stage_i$ differs depending on whether there is a control-flow. With a control-flow, *Sync2XML* extracts *cond* and *val* for $stage_i$ from arrows in the control flow. Without a control-flow,, *Sync2XML* extracts *cond* and *val* for $stage_i$ from "IfStatement" and "CaseStatement" for the control circuit in the AST.

In addition, the extraction method for *cond* and *val* for $stage_i$ differs depending on whether there are stall signals. Without a stall signal, *Sync2XML* extracts *cond* and *val* from the AST and control-flow. With stall signals, the extraction method differs depending on whether there are multiple stall signals or one stall signal. *Sync2XML* does not extract *cond* and *val* if there is one stall signal, because the operations of $ctrl_i$ and $ctrl_{i-1}$ cannot be resumed by one stall signal at the same time. In contrast, *Sync2XML* extracts *cond* and *val* from the AST and control-flow if there are multiple stall signals.

Figure 10 shows the generated CDFG for the synchronous RTL model in Fig.7. For example, we regard between *in*0 and reg_0 as $stage_0$. The conditional signal *cond* and its value *val* of $stage_0$ are empty because there is one stall signal in the synchronous RTL model.

2) ANALYSIS OF PIPELINE STAGES

For the generated CDFG, *Sync2XML* analyzes preceding and succeeding pipeline stages for each $stage_i$. *Sync2XML* also analyzes register write signals and multiplexer control signals from the CDFG. After analyzing the CDFG, *Sync2XML*

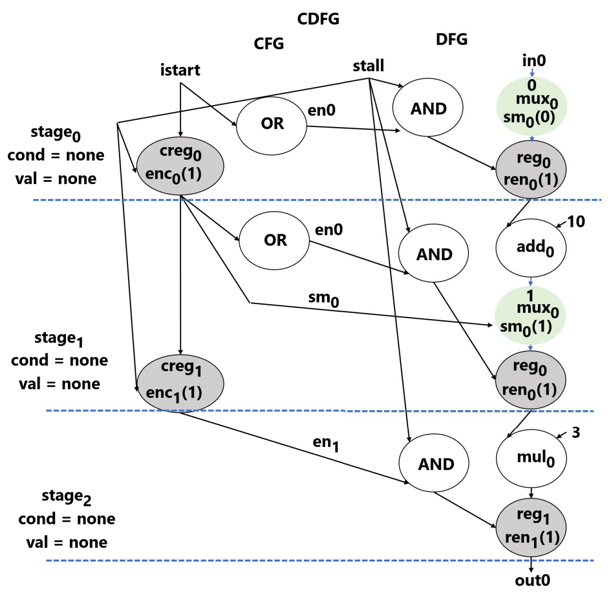


FIGURE 10. CDFG for the synchronous RTL model in Fig.7.

generates pipeline stage information and timing information into the Model-XML.

a: ANALYSIS OF PRECEDING AND SUCCEEDING PIPELINE STAGES

Sync2XML analyzes preceding and succeeding pipeline stages for each $stage_i$. In the CDFG, $stage_j$ ($j \neq i$) is a succeeding pipeline stage for $stage_i$ when resources of $stage_i$ are connected to resources of $stage_j$. In contrast, $stage_j$ is a preceding pipeline stage for $stage_i$ when resources of $stage_j$ are connected to resources of $stage_i$. *Sync2XML* also extracts a conditional signal and its value for the transition between pipeline stages from *cond* and *val* of $stage_j$.

After analyzing $stage_i$, *Sync2XML* generates pipeline stage information into the Model-XML. In the pipeline stage information, *Sync2XML* generates the pipeline stage information for each $stage_i$ using $\langle ctrl \rangle$. $\langle ctrl \rangle$ represents $stage_i$. *Sync2XML* also generates preceding pipeline stage information $\langle pred \rangle$ and succeeding pipeline stage information $\langle succ \rangle$ into $\langle ctrl \rangle$. $\langle pred \rangle$ and $\langle succ \rangle$ represent preceding and succeeding pipeline stages for each $stage_i$. If there is no preceding pipeline stage, *Sync2XML* assigns the external input signal *start* to $\langle pred \rangle$. Moreover, *Sync2XML* assigns a conditional signal *ctrlname* and its value *ctrlval* to operate preceding or succeeding pipeline stages to $\langle pred \rangle$ or $\langle succ \rangle$.

Figure 11(a) shows the generated pipeline stage information from the CDFG in Fig.10. The blue color represents the relationship between the CDFG and pipeline stage information. *Sync2XML* generates three $\langle ctrl \rangle$. For $\langle ctrl \rangle$ corresponding to $stage_1$, *Sync2XML* assigns $stage_0$ to $\langle pred \rangle$ because the preceding pipeline stage for $stage_1$ is $stage_0$. *ctrlname* and *ctrlval* are empty because $stage_1$ does not have a conditional signal. *Sync2XML* also assigns $stage_2$ to $\langle succ \rangle$ because the succeeding pipeline stage for $stage_1$ is $stage_2$. *ctrlname* and *ctrlval* are empty because $stage_2$ does not have a conditional signal. *Sync2XML* generates all $\langle ctrl \rangle$ in the same way.

b: ANALYSIS OF REGISTER WRITE SIGNALS AND MULTIPLEXER CONTROL SIGNALS

Sync2XML analyzes values of register write signals and multiplexer control signals. If there is reg_k in $stage_i$ on the CDFG, the value of the register write signal for reg_k is 1 for $stage_i$. If there is mux_l in $stage_i$ on the CDFG, the value of the multiplexer control signal for mux_l is the control value held by mux_l for $stage_i$.

After analyzing $stage_i$, *Sync2XML* generates the register write signal information and the multiplexer control signal information for each control signal of registers and multiplexers using $\langle reg \rangle$ and $\langle mux \rangle$. In the timing information, the control signal name of registers and multiplexers and the control value at each pipeline stage are described. $\langle reg \rangle$ represents a register write signal name and its value while $\langle mux \rangle$ represents a multiplexer control signal name and its value. *Sync2XML* assigns the control values to $\langle reg \rangle$ and $\langle mux \rangle$ from the analyzed values of the control signals.

Figure 11(b) shows the generated timing information from the CDFG in Fig.10. The red color represents the relationship between the CDFG and timing information. Four $\langle reg \rangle$ and one $\langle mux \rangle$ are generated. For example, for reg_1, the register write signal ren_1 whose values of $stage_2$ is 1 to $\langle reg \rangle$ because there is reg_1 in $stage_1$ on the CDFG.

C. EXTENSION OF XML2Async

XML2Async generates pipelined asynchronous RTL models from the Model-XML by assigning control modules, connecting the control modules, and generating control signals. Generated asynchronous RTL models are represented by Verilog HDL.

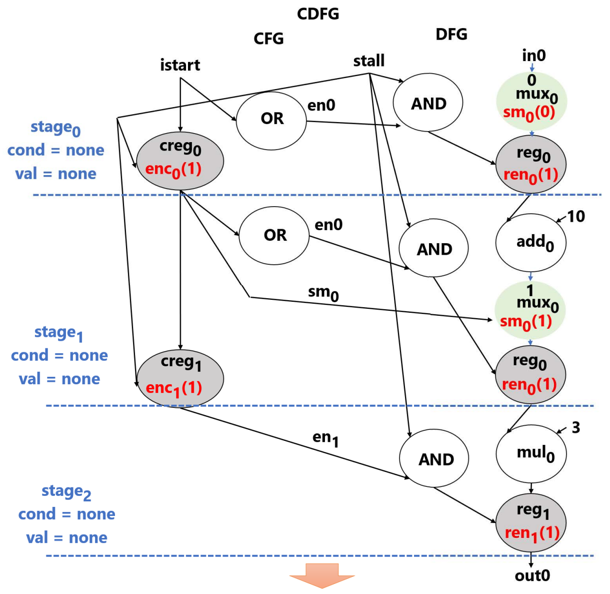
1) ASSIGNING AND CONNECTING CONTROL MODULES

XML2Async assigns $ctrl_i$ for each $\langle ctrl \rangle$ in the Model-XML. *XML2Async* also connects $ctrl_i$ by referring to $\langle pred \rangle$ and $\langle succ \rangle$ in $\langle ctrl \rangle$. If there are pipeline stalls or if the II is different, *XML2Async* assigns and connects $ctrl_i$ in the same way.

$ctrl_i$ in Fig.12 represents assigned $ctrl_i$ by referring to $\langle ctrl \rangle$ in Fig.11(a). *XML2Async* assigns three $ctrl_i$ for three $\langle ctrl \rangle$. For $ctrl_1$, *XML2Async* connects $ctrl_0$ to $ctrl_1$ by referring to $\langle pred \rangle$ in $\langle ctrl \rangle$. *XML2Async* also connects $ctrl_1$ to $ctrl_2$ by referring to $\langle succ \rangle$ in $\langle ctrl \rangle$. *XML2Async* connects all control modules in the same way.

2) GENERATION OF CONTROL SIGNALS

XML2Async generates register write signals and multiplexer control signals by referring to $\langle reg \rangle$ and $\langle mux \rangle$ in the Model-XML. The assignment of register write signals consists of the logical OR of $lclk_i$ where $stage_i$ in $\langle reg \rangle$ is equal to 1. The assignment of multiplexer control signals consists of the logical XOR of st_i where $stage_i$ in $\langle mux \rangle$ is different



Pipeline stage information

```
<ctrlpath>
  <ctrl id="0" name="stage0">
    <pred id="0" name="start" ctrlname="" ctrlval=""/>
    <succ id="0" name="stage1" ctrlname="" ctrlval=""/>
  </ctrl>
  <ctrl id="1" name="stage1">
    <pred id="0" name="stage0" ctrlname="" ctrlval=""/>
    <succ id="0" name="stage2" ctrlname="" ctrlval=""/>
  </ctrl>
  <ctrl id="2" name="stage2">
    <pred id="0" name="stage1" ctrlname="" ctrlval=""/>
  </ctrl>
</ctrlpath>
```

(a)

Timing information

```
<timing>
  <reg id="0" name="ren0" stage0="1" stage1="1" stage2="0"/>
  <reg id="1" name="ren1" stage0="0" stage1="0" stage2="1"/>
  <reg id="2" name="enc0" stage0="1" stage1="0" stage2="0"/>
  <reg id="3" name="enc1" stage0="0" stage1="1" stage2="0"/>
  <mux id="0" name="sm0" stage0="0" stage1="1" stage2="0"/>
</timing>
```

(b)

FIGURE 11. Model-XML generated from the CDFG in Fig.10: (a) pipeline stage information and (b) timing information.

from the value of previous stage $stage_{i-1}$. If there are several XOR gates, *XML2Async* generates multiplexer control signals using the logical OR of these XOR gates.

On the other hand, the connection method through the generation of control signals differs depending on whether there is a pipeline stall. The proposed method uses the components in the control circuit of the synchronous circuits if there is a pipeline stall. Therefore, *XML2Async* connects $ctrl_i$ to that components and does not generate multiplexer control signals. In contrast, *XML2Async* generates multiplexer control signals and does not connects $ctrl_i$ to that components.

The control signals in Fig.12 represent generated control signals by referring to $\langle reg \rangle$ in Fig.11(b). As an example of the generation of the register write signal ren_1 for reg_1, the assignment of ren_1 is just an assignment statement because the values of $stage_2$ in $\langle reg \rangle$ is 1. Figure 12 shows converted asynchronous RTL models from the synchronous RTL models in Fig.7.

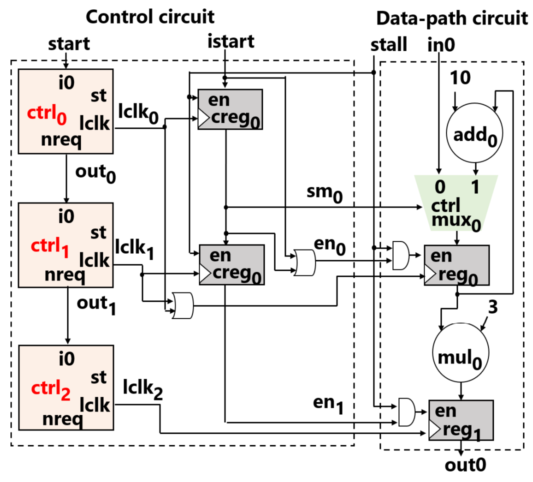


FIGURE 12. Pipelined asynchronous RTL models for the pipelined synchronous RTL model in Fig.7.

D. OPTIMIZATION METHODS

The quality of asynchronous circuits after the RTL conversion depends on the representation styles before the conversion. To obtain the high quality of asynchronous circuits, we used three optimization methods that can be applied during RTL conversion. One is the conversion from DFFs into D latches to reduce the dynamic power consumption of registers which is not described in [24]. Second is the use of appropriate DFFs to reduce the area of registers. Third is inserting latches before data-path resources to reduce the dynamic power consumption of data-path circuits. The second and third optimization methods are based on [24].

The optimization methods depend on the II. The first and second optimization methods can be used independently of the II. On the other hand, the third optimization method can not be used when the II is one cycle. This is because all pipeline stages operate every cycle, thus unnecessary operations do not occur.

1) CONVERSION FROM DFFs INTO D LATCHES

The purpose of the conversion from DFFs into D latches is to optimize the dynamic power consumption of registers. In general, D latches are low power and small area compared to DFFs. Therefore, the proposed method converts DFFs into D latches during *XML2Async*.

a: TIMING CONSTRAINTS

It is necessary to satisfy the setup and hold constraints for the converted D latches to operate the circuit correctly.

The input data for the D latch dl_k must be arrived at dl_k until dl_k is closed. This is called the setup constraint for dl_k. Figure 13(a) shows a data-path $sdp_{i,p}$ and a control-path $scp_{i,p}$ related to the setup constraint. $sdp_{i,p}$ represents a data-path from the output of $lclk_{i-1}$ to the destination D latch dl_1 through the source D latch dl_0. $scp_{i,p}$ represents a control-path from the output of $lclk_{i-1}$ to the destination D latch dl_1 through $ctrl_i$. We define the maximum delay of $sdp_{i,p}$ as $t_{maxsdp_{i,p}}$, the minimum delay of $scp_{i,p}$ as $t_{minscp_{i,p}}$, and the margin for $t_{maxsdp_{i,p}}$ as $t_{sdpm_{i,p}}$. Then, the setup constraint can be represented by the following inequality.

$$t_{minscp_{i,p}} > t_{maxsdp_{i,p}} + t_{sdpm_{i,p}} \tag{5}$$

If the setup constraint is violated, we need to adjust the number of cells for sd_i.

The D latch dl_k must be closed until the next input data arrives at dl_k after the input data are written to dl_k. This is called the hold constraint for dl_k. Figure 13(b) shows a data-path $hdp_{i,p}$ and a control-path $hcp_{i,p}$ related to the hold constraint. $hdp_{i,p}$ represents a data-path from the input signal *start* to the destination D latch dl_1 through the source D latch dl_0. $hcp_{i,p}$ represents a control-path from the input signal *start* to the destination D latch dl_1 through $ctrl_i$. 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 $t_{hcpm_{i,p}}$, the hold time of the destination register as $t_{hold_{i,p}}$, and the input interval as II. Then, the hold constraint can be represented by the following inequality.

$$t_{minhdp_{i,p}} + gct \times II > t_{maxhcp_{i,p}} + t_{hcpm_{i,p}} + t_{hold_{i,p}} \tag{6}$$

If the hold constraint is violated, we need to adjust the number of cells for $hddl_k$.

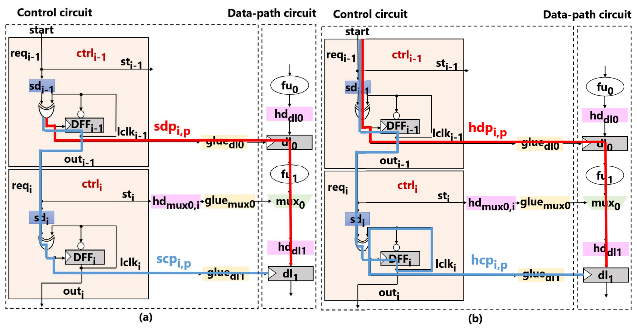


FIGURE 13. Timing constraints: (a) setup constraint and (b) hold constraint.

2) USE OF APPROPRIATE DFFs

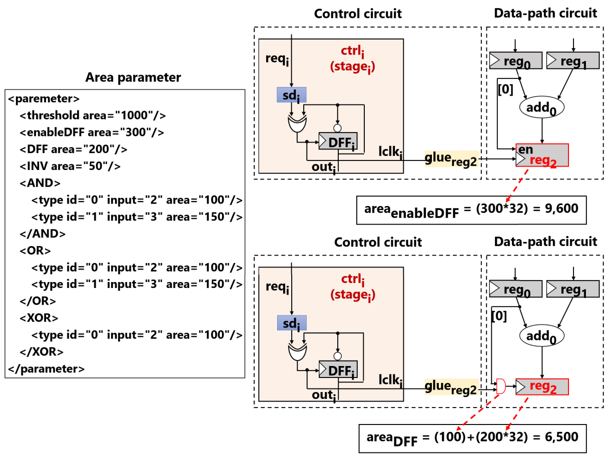
The purpose of the use of appropriate DFFs is to optimize the area of registers. In general, DFFs without an enable signal are low power and small area compared to DFFs with an enable signal. During *XML2Async*, to avoid the use of DFFs with an enable signal, the optimization method moves the assignment for the enable signal to the outside of the register descriptions. This results in the insertion of a glue logic (e.g., AND operation) for the registers in which consists of $lclk_i$ and the enable signal.

The optimization method moves the assignments to the outside of the registers when the following inequality is satisfied.

$$area_{enableDFF} - area_{DFF} > threshold \tag{7}$$

where $area_{enableDFF}$, $area_{DFF}$, and *threshold* represent the circuit area for DFFs with an enable signal, the circuit area for DFFs without an enable signal, and a threshold value. The detail of the use of appropriate DFFs is described in [24].

Figure 14 shows an example. The optimization method calculates $area_{enableDFF}$ and $area_{DFF}$ by referring to the area parameters, the bit-width of the registers, and the number of logics and literals in the assignments of the enable signals. In this example, the DFFs without an enable signal are used because the difference between $area_{enableDFF}$ and $area_{DFF}$ is more than the threshold value. The optimization method moves the condition signal ($reg_0\_out[0]$) for the assignment of reg_2 to outside of reg_2 with "assign" statement. Then, the optimization method inserts a new logic which consists of logical AND of $reg_0\_out[0]$ and $gluereg_2\_out$ to generate a register write signal (en_2) for reg_2.


Area parameter

```
<parameter>
  <threshold area="1000"/>
  <enableDFF area="300"/>
  <DFF area="200"/>
  <INV area="50"/>
  <AND>
    <type id="0" input="2" area="100"/>
    <type id="1" input="3" area="150"/>
  </AND>
  <OR>
    <type id="0" input="2" area="100"/>
    <type id="1" input="3" area="150"/>
  </OR>
  <XOR>
    <type id="0" input="2" area="100"/>
  </XOR>
</parameter>
```

$area_{enableDFF}$ = (300*32) = 9,600

$area_{DFF}$ = (100)+(200*32) = 6,500

FIGURE 14. Area estimation for reg_2.

3) LATCH INSERTION AS OPERAND ISOLATION

The purpose of latch insertion as operand isolation is to reduce the dynamic power consumption of data-path circuits. The optimization method prevents the dynamic power consumption by using D latches during *XML2Async*. Compared to AND gates, D latches do not propagate unnecessary signal transitions to the functional units. However, when the optimization method inserts D latches in critical paths, the performance may be degraded by the delay of the inserted D latches. To preserve critical path delays, we only insert D latches for data-paths which are not critical paths in each pipeline stage. The detail of the latch insertion algorithm is described in [24].

To estimate path delays, we prepare delay parameters as shown in Fig.15(a). In Fig.15(a), *max* represents the maximum delay of data-path resources. In this method, the delay parameters are prepared by the following way. First, we perform logic synthesis for RTL models including registers, functional units, and multiplexers using a clock constraint. We also explore the fastest clock cycle time without timing violations. Then, we obtain the delays of the registers, functional units, and multiplexers after logic synthesis. Finally, we define the delays as *max* for data-path resources.

The optimization method inserts D latches for data-paths which are not critical paths. By referring to the delay parameters, the optimization method calculates the path delays $t_{dp_{i,l}}$ by summing *max* of each data-path resource in $dp_{i,l}$. $dp_{i,l}$ represents the l-th data-path in $stage_i$. The critical path delay t_{stage_i} in $stage_i$ is the maximum delay in $t_{dp_{i,l}}$. Then, optimization method inserts D latches dl_d for data-paths which $t_{dp_{i,l}}$ is not t_{stage_i}.

Figure 15 shows an example of the latch insertion. The value represented by "()" is the estimated delay of the corresponding data-path. t_{stage_1} is 1, 350 and t_{stage_2} is 650. add_0 operates in $stage_1$ and sub_0 operates in $stage_2$. In such a case, when the value of reg_0 which is the source of add_0 and sub_0 is changed, both add_0 and sub_0 operate because the value of reg_0 is propagated to add_0 and sub_0. In each $stage_i$, one of the operations is valid while the other operation is invalid. Therefore, dl_0 is inserted between reg_0 and add_0 to prevent the propagation of the value of reg_0 to add_0 in $stage_2$. On the other hand, dl_d does not inserted between reg_0 and sub_0 because the insertion of a latch results in the increase of the critical path delay in $stage_2$.

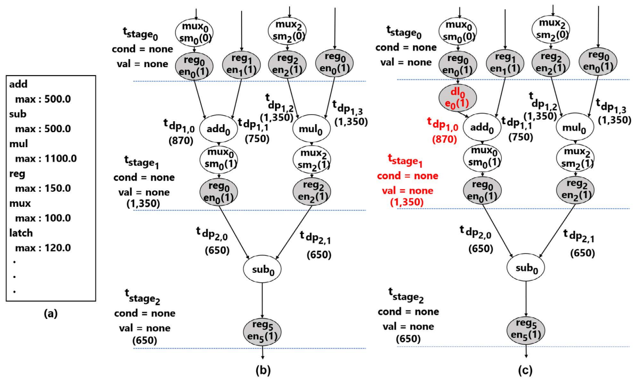


FIGURE 15. Latch insertion: (a) delay parameters, (b) DFG before applying the latch insertion, and (c) DFG after applying the latch insertion.

VI. EXPERIMENTAL RESULTS

In the experiment, we converted pipelined synchronous RTL models into asynchronous RTL models using the proposed method. For the experiments, we implemented a conversion tool for the proposed method using Java. The conversion tool was performed on a Windows 10 machine (Intel Core i7-8700 @3.2 GHz CPU and 16 GB memory).

We prepared four synchronous RTL models synthesized by high-level synthesis (HLS) from SystemC models using Cadence Stratus HLS 18.1. The prepared synchronous RTL models were a differential equation solver (DIFFEQ), an elliptic wave filter (EWF), the multilayer perceptron (MLP) [27] whose number of neurons is 32, and the advanced encryption standard (AES) [28]. To show that the proposed method can deal with synchronous RTL models with different IIs, we also prepared synchronous RTL models whose II was one cycle and two cycles. In addition, we prepared synchronous RTL models with a hard stall (Hard) and a soft stall (Soft) by applying directives [29] in Stratus HLS. Hard means that the operations of all pipeline stages stall, whereas Soft means that the operations of specified pipeline stages stall. We also applied the clock gating option to HLS for synchronous RTL models. The library was eShuttle 65 nm process technology.

As a reference, we synthesized the synchronous circuits (*sync*) from the RTL models for DIFFEQ, EWF, MLP, and AES using Cadence Genus 18.1 with an eShuttle 65 nm technology library. To compare the quality of the asynchronous circuits generated by the proposed method with the quality of the synchronous circuits, we explored the fastest synchronous circuits without timing violations. When the II was one cycle, the clock cycle times of DIFFEQ, EWF, MLP, and AES were 1,400 ps, 1,500 ps, 600 ps, 900 ps. When the II was two cycles, the clock cycle times of DIFFEQ, EWF, MLP, and AES were 1,400 ps, 1,500 ps, 700 ps, 900 ps. We also applied the clock gating option to logic synthesis for synchronous circuits.

To evaluate the quality of the converted asynchronous RTL models (*async*), we performed logic synthesis based on the design flow in [30]. To obtain the same performance as the synchronous circuits, we synthesized the asynchronous RTL models with maximum delay constraints for all control-paths and local clock constraints for $lclk_i$. To generate these constraints, we referred to [30]. We obtained five asynchronous RTL models from the synchronous RTL model.

- *async* - no optimization
- $async_l$ - with D latches instead of DFFs
- $async_r$ - with appropriate DFFs
- $async_{op}$ - with latch insertion as operand isolation
- $async_{opl}$ - with combination of $async_l$ and $async_{op}$

Note that $async_{opl}$ does not include $async_r$ because the used library does not include a D latch with an enable signal. We also could not design $async_{op}$ for MLP and AES because there was no non-critical path in each pipeline stage.

A. CONVERSION RESULTS

Table 1 shows the conversion results using the proposed method. *Type*, *CT*, *Stage*, and *Sverilog* represent the type of synchronous RTL models, the clock cycle time, the number of pipeline stages, and the number of lines in Verilog HDL of synchronous RTL models. *AST*, *Model-XML*, *Averilog*, and *Time* represent the number of lines in the AST, the number of lines in the Model-XML, the number of lines in Verilog HDL of asynchronous RTL models, and the conversion time.

From table 1, the conversion time depends on the number of pipeline stages and the number of lines in the AST. This is because the proposed method generates Model-XML from the AST and analyzes data-paths and control-paths in synchronous RTL models for each pipeline stage. Compared to the conversion time for the RTL models without stalls, the conversion time for the RTL models with stalls was increased

TABLE 1. RTL conversion results. This table represents the type of synchronous RTL models, the clock cycle time, the number of pipeline stages, the number of lines in Verilog HDL of synchronous RTL models, the number of lines in the AST, the number of lines in the Model-XML, the number of lines in Verilog HDL of asynchronous RTL models, and the conversion time. Red colors represent the worst results in each class and blue colors represent the best results in each class.

$Name$	$Type$	CT [ps]	$Stage$	$Sverilog$ [lines]	AST [lines]	$Model\text{-}XML$ [lines]	$Averilog$ [lines]	$Time$ [s]
DIFFEQ	II=1	1,200	4	164	765	110	348	1.9
	II=2	1,200	4	149	662	99	337	2.0
	Hard	1,200	4	209	859	188	384	2.1
	Soft	1,200	4	208	922	177	410	2.0
EWF	II=1	1,200	9	668	3,202	429	1,337	2.5
	II=2	1,200	9	648	2,738	372	1,190	3.9
	Hard	1,200	9	859	3,588	746	1,428	2.8
	Soft	1,200	9	825	3,703	715	1,557	2.7
MLP	II=1	400	20	16,925	94,130	22,276	36,609	283.1
	II=2	400	20	17,074	93,363	22,856	43,227	613.9
	Hard	400	20	20,734	101,164	28,039	36,464	341.9
	Soft	400	20	18,863	99,661	27,974	36,618	338.1
AES	II=1	600	41	130,036	946,154	192,782	138,553	2,992.3
	II=2	600	41	129,062	936,948	191,462	133,713	22,177.8
	Hard	600	41	133,499	955,070	198,594	139,278	3,130.8
	Soft	600	41	132,152	954,490	198,474	139,600	3,172.3
Total		13,600.0	296.0	602,075.0	4,197,419.0	885,293.0	711,053.0	33,070.1
Average		850.0	18.5	37,629.7	262,338.7	55,330.8	44,440.8	2,066.9

TABLE 2. Evaluation results after logic synthesis. This table represents the circuit area, execution time, dynamic power consumption, and energy consumption of asynchronous circuits. Red colors represent the worst results in each class and blue colors represent the best results in each class. The average ratio represents the average value for the increase/decrease ratio of each circuit.

Name	Type		Circuit area [μm^2]				Execution time [ps]	Dynamic power [mW]					Energy [pJ]
			register	comb.	ctrl	total		clock	register	comb.	ctrl	total	
DIFFEQ	II=1	$sync$	124,434	680,346	846	805,626	144,357	2.97	3.05	23.90	0.01	29.93	4,320.61
		$async$	124,686	672,147	8,748	805,581	144,717	2.88	3.04	24.03	0.44	30.39	4,397.95
	II=2	$sync$	91,008	736,389	1,125	828,522	282,957	1.66	2.13	25.40	0.03	29.22	8,268.00
		$async$	90,918	739,332	8,712	838,962	283,739	1.23	1.74	24.78	0.26	28.01	7,947.53
EWF	II=1	$sync$	567,063	1,280,421	1,881	1,849,365	162,159	12.90	11.30	47.80	0.01	72.01	11,677.07
		$async$	567,045	1,265,004	19,260	1,851,309	162,751	11.90	11.36	47.30	0.84	71.40	11,620.42
	II=2	$sync$	378,747	1,295,442	2,628	1,676,817	310,659	5.78	6.33	39.00	0.07	51.18	15,899.53
		$async$	379,008	1,301,067	23,112	1,703,187	313,284	4.42	5.76	36.09	0.79	47.06	14,743.15
MLP	II=1	$sync$	1,473,579	937,350	4,158	2,415,087	71,560	85.40	41.70	38.70	0.02	165.82	11,866.08
		$async$	1,473,795	937,458	20,178	2,431,431	72,541	70.70	41.20	37.71	2.43	152.04	11,029.13
	II=2	$sync$	1,277,478	1,007,622	3,888	2,288,988	152,728	62.70	42.50	51.00	0.18	156.38	23,883.60
		$async$	1,276,677	1,004,994	60,993	2,342,664	155,502	35.00	26.66	36.42	3.22	101.30	15,752.35
AES	II=1	$sync$	7,119,378	17,149,941	8,505	24,277,824	126,167	270.90	181.40	741.40	0.02	1,193.72	150,608.07
		$async$	7,105,842	16,220,745	59,652	23,386,239	127,331	191.20	171.70	645.10	3.65	1,011.65	128,814.41
	II=2	$sync$	4,209,957	18,018,693	9,090	22,237,740	215,234	96.10	100.20	721.00	0.32	917.62	197,503.02
		$async$	4,188,654	16,637,049	59,643	20,885,346	217,226	65.30	70.80	393.60	2.15	531.85	115,900.93
Total		$sync$	15,241,644	41,106,204	32,121	56,379,969	1,465,821	538.41	388.61	1,688.20	0.66	2,615.88	424,025.98
		$async$	15,206,625	38,777,796	260,298	54,244,719	1,477,091	382.63	332.26	1,245.03	13.78	1,973.70	309,836.59
Average		$sync$	1,905,205.50	5,138,275.50	4,015.13	7,047,496.13	183,227.63	67.30	48.58	211.03	0.08	326.99	53,003.25
		$async$	1,900,828.13	4,847,224.50	32,537.25	6,780,589.88	184,636.38	47.83	41.53	155.63	1.72	246.71	38,729.57
Average ratio		$async$	-0.07%	-1.86%	790.42%	-0.48%	0.85%	-22.88%	-12.53%	-12.49%	5,857.00%	-14.04%	-13.36%

because the number of lines in the AST with stalls is more than the number of lines in the AST without stalls. When the II was two cycles, the conversion time was increased because shared data-path resources were analyzed multiple times.

We also performed logic simulation to verify the functional correctness of the converted asynchronous RTL models. For the simulation, we generated a Standard Delay Format (SDF) file by synthesizing the asynchronous RTL models. Then, we prepared a test bench with 100 arbitrary test patterns. After the simulation, we confirmed that all output values of the asynchronous RTL models were the same as the output values of the synchronous RTL models.

B. EVALUATION AFTER LOGIC SYNTHESIS

We evaluated the designed circuits after logic synthesis in terms of circuit area, execution time, dynamic power consumption, and energy consumption. The circuit area was obtained from the report file generated by Genus. Note that the circuit area does not include the wiring area. The execution time was obtained by simulating the designed circuits

TABLE 3. Evaluation results for asynchronous circuits applying the optimization methods. This table represents the circuit area, execution time, dynamic power consumption, and energy consumption of asynchronous circuits applying each optimization method. Red colors represent the worst results in each class and blue colors represent the best results in each class. The average ratio represents the average value for the increase/decrease ratio of each circuit.

Name	Type		Circuit area [μm^2]				Execution time [ps]	Dynamic power [mW]					Energy [pJ]
			register	comb.	ctrl	total		clock	register	comb.	ctrl	total	
DIFFEQ	II=1	$async$	124,686	672,147	8,748	805,581	144,717	2.88	3.04	24.03	0.44	30.39	4,397.95
		$asyncl$	91,413	618,093	8,748	718,254	144,733	2.35	2.03	21.33	0.44	26.15	3,784.77
	II=2	$async$	90,918	739,332	8,712	838,962	283,739	1.23	1.74	24.78	0.26	28.01	7,947.53
		$async_l$	66,636	661,059	8,694	736,389	284,328	1.02	1.12	21.95	0.29	24.38	6,931.92
		$async_{op}$	114,300	715,005	10,773	840,078	286,346	1.22	2.33	22.91	0.43	26.89	7,699.84
		$async_{opl}$	89,748	642,429	10,845	743,022	286,333	1.01	1.65	20.57	0.44	23.67	6,777.50
EWF	II=1	$async$	567,045	1,265,004	19,260	1,851,309	162,751	11.90	11.36	47.30	0.84	71.40	11,620.42
		$asyncl$	402,147	1,331,469	19,260	1,752,876	162,741	9.44	7.01	47.70	0.84	64.99	10,576.54
	II=2	$async$	379,008	1,301,067	23,112	1,703,187	313,284	4.42	5.76	36.09	0.79	47.06	14,743.15
		$async_l$	271,989	1,283,742	22,671	1,578,402	313,271	3.54	3.34	34.95	0.72	42.55	13,329.68
		$async_{op}$	402,822	1,300,662	24,714	1,728,198	312,680	4.43	6.22	35.73	0.94	47.32	14,796.02
		$async_{opl}$	296,514	1,303,731	24,597	1,624,842	312,866	3.55	3.17	35.15	0.93	42.80	13,390.66
MLP	II=1	$async$	1,473,795	937,458	20,178	2,431,431	72,541	70.70	41.20	37.71	2.43	152.04	11,029.13
		$asyncl$	1,028,916	937,197	20,664	1,986,777	72,531	55.00	25.60	37.02	2.43	120.05	8,707.35
	II=2	$async$	1,276,677	1,004,994	60,993	2,342,664	155,502	35.00	26.66	36.42	3.22	101.30	15,752.35
		$async_l$	891,486	1,123,218	61,362	2,076,066	155,520	27.40	16.23	47.20	3.33	94.16	14,643.76
		$async_{op}$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		$async_{opl}$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
AES	II=1	$async$	7,105,842	16,220,745	59,652	23,386,239	127,331	191.20	171.70	645.10	3.65	1,011.65	128,814.41
		$asyncl$	5,022,477	19,387,251	59,661	24,469,389	127,323	151.00	113.90	488.00	3.65	756.55	96,326.22
	II=2	$async$	4,188,654	16,637,049	59,643	20,885,346	217,226	65.30	70.80	393.60	2.15	531.85	114,838.10
		$async_l$	2,993,463	18,156,240	59,643	21,209,346	217,218	52.20	47.10	300.70	2.15	402.15	87,357.44
		$async_{op}$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		$async_{opl}$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Total		$async$	15,206,625	38,777,796	260,298	54,244,719	1,477,091	382.63	332.26	1,245.03	13.78	1,973.70	309,836.59
		$async_l$	10,768,527	43,498,269	260,703	54,527,499	1,477,665	301.95	216.33	998.85	13.85	1,530.98	241,654.45
		$async_{op}$	517,122	2,015,667	35,487	2,568,276	599,026	5.65	8.55	58.64	1.37	74.21	22,495.86
		$async_{opl}$	386,262	1,946,160	35,442	2,367,864	599,199	4.56	4.82	55.72	1.37	66.47	20,168.17
Average		$async$	1,900,828.13	4,847,224.50	32,537.25	6,780,589.88	184,636.38	47.83	41.53	155.63	1.72	246.71	38,729.57
		$async_l$	1,346,065.88	5,437,283.63	32,587.88	6,815,937.38	184,708.13	37.74	27.04	124.86	1.73	191.37	30,206.81
		$async_{op}$	258,561.00	1,007,833.50	17,743.50	1,284,138.00	299,513.00	2.83	4.28	29.32	0.69	37.11	11,247.93
		$async_{opl}$	193,131.00	973,080.00	17,721.00	1,183,932.00	299,599.50	2.28	2.41	27.86	0.69	33.24	10,084.08
Average ratio		$async_l$	-28.62%	3.21%	0.11%	-7.40%	0.02%	-20.13%	-36.66%	-5.64%	0.76%	-15.40%	-15.37%
		$async_{op}$	16.00%	-1.66%	15.29%	0.80%	0.36%	-0.29%	20.95%	-4.27%	42.19%	-1.72%	-1.38%
		$async_{opl}$	-11.53%	-6.45%	15.45%	-8.02%	0.39%	-18.78%	-25.07%	-9.80%	43.48%	-12.27%	-11.95%

with an arbitrary test sequence using VCS Q-2020.03-SP1. Note that the used delay data were data after post-synthesis. The dynamic power consumption was obtained by PrimeTime Q-2019.12-SP3 with the VCD file generated by VCS. The energy consumption was the product of the execution time and the dynamic power consumption.

Table 2 shows the evaluation results after logic synthesis. *async* could reduce the circuit area by 0.48% on average due to the insertion of the control modules and the change of the structure of the data-path circuit by assigning the constraints. Compared to *async* where II was one cycle, the area of the control circuit of *async* where II was two cycles was increased because the control circuit includes the control signals for multiplexers. The execution time of *async* was increased by 0.85% on average because we adjusted the delay element in control modules to satisfy the setup constraint. The delays of the control-paths were longer than the critical path delays of the data-paths. *async* could reduce the dynamic power consumption by 14.04% on average because the only required circuit components are operated due to the use of local signals instead of global clock signals. Compared to *async* where II was one cycle, the dynamic power consumption of *async* where II was two cycles was reduced because all data-path resources are not operated every cycle. *async* could reduce the energy consumption by 13.36% on average because the reduction of the dynamic power consumption was higher than the increase of the execution time.

Table 3 shows the evaluation results for the asynchronous circuits applying the optimization methods. Compared to *async*, $async_l$ could reduce the circuit area by 7.40% on average due to the use of D latches instead of DFFs. On the other hand, $async_l$ did not have a significant impact on the execution because we assigned the same values for the maximum delay constraints and local clock constraints. $async_l$ could reduce the dynamic power consumption by 15.40% on average because the dynamic power consumption of the register and clock was reduced due to the use of D latches instead of DFFs. Compared to $async_l$ where II was two cycles, the reduction amount of the dynamic power consumption of $async_l$ where II was one cycle was high because the area of registers is large. Similarly, $async_l$ could reduce the energy consumption by 15.37% on average.

On the other hand, compared to *async*, the circuit area of $async_{op}$ was increased by 0.80% on average because we inserted D latches as isolators. $async_{op}$ did not have a significant impact on the execution time because D latches

TABLE 4. Evaluation results for asynchronous circuits with Hard and Soft. This table represents the circuit area, execution time, dynamic power consumption, and energy consumption of asynchronous circuits with Hard and Soft. Red colors represent the worst results in each class and blue colors represent the best results in each class. The average ratio represents the average value for the increase/decrease ratio of each circuit.

Name	Type		Circuit area [μm^2]				Execution time [ps]	Dynamic power [mW]					Energy [pJ]
			register	comb.	ctrl	total		clock	register	comb.	ctrl	total	
DIFFEQ	Hard	$sync$	124,398	679,752	1,071	805,221	144,357	2.69	2.85	21.30	0.02	26.86	3,877.43
		$async$	146,556	789,381	9,495	945,432	144,772	3.06	3.21	25.13	0.44	31.84	4,609.54
		$async_r$	124,128	676,008	9,360	809,496	144,744	2.60	2.81	21.03	0.44	26.88	3,890.72
		$async_l$	91,782	634,428	9,945	736,155	145,249	2.13	1.81	19.90	0.47	24.31	3,531.00
	Soft	$sync$	125,019	679,743	1,521	806,283	144,357	2.67	2.85	21.30	0.04	26.86	3,877.43
		$async$	147,807	787,140	9,486	944,433	144,980	2.82	3.16	24.76	0.44	31.18	4,520.48
		$async_r$	124,677	678,798	9,522	812,997	144,948	2.60	2.79	20.96	0.45	26.80	3,884.61
		$async_l$	92,214	633,384	10,026	735,624	144,937	2.13	1.81	19.89	0.48	24.31	3,523.42
EWF	Hard	$sync$	565,173	1,234,206	2,106	1,801,485	162,159	11.60	10.26	40.10	0.02	61.98	10,050.61
		$async$	696,843	1,379,943	21,357	2,098,143	162,779	12.90	12.37	45.31	0.83	71.41	11,624.05
		$async_r$	565,281	1,241,991	20,862	1,828,134	162,760	10.70	10.34	40.71	0.83	62.58	10,185.52
		$async_l$	402,930	1,354,446	22,581	1,779,957	162,873	8.40	6.21	42.98	0.90	58.49	9,526.44
	Soft	$sync$	567,225	1,241,478	3,681	1,812,384	162,159	11.10	10.27	40.50	0.08	61.95	10,045.75
		$async$	699,975	1,385,451	22,932	2,108,358	162,811	11.60	11.68	45.46	0.93	69.67	11,343.04
		$async_r$	568,593	1,240,569	22,878	1,832,040	162,792	10.60	10.34	40.76	0.93	62.63	10,195.66
		$async_l$	403,911	1,349,577	24,012	1,777,500	162,782	8.43	6.20	42.71	0.99	58.33	9,495.07
MLP	Hard	$sync$	1,465,866	929,889	4,158	2,399,913	71,587	76.50	37.20	34.40	0.03	148.13	10,604.18
		$async$	1,748,646	1,275,615	25,245	3,049,506	72,626	74.20	50.20	49.04	2.30	175.74	12,763.29
		$async_r$	1,459,368	931,491	24,030	2,414,889	72,598	61.70	36.50	33.84	2.30	134.34	9,752.82
		$async_l$	1,024,839	1,087,128	27,900	2,139,867	72,569	48.00	22.70	41.46	2.65	114.81	8,331.65
	Soft	$sync$	1,463,589	928,314	8,433	2,400,336	71,560	71.50	36.80	34.20	0.40	142.90	10,225.92
		$async$	1,762,614	1,236,636	23,598	3,022,848	72,626	66.60	47.40	45.45	2.16	161.61	11,737.09
		$async_r$	1,467,360	926,901	23,517	2,417,778	72,598	61.70	36.50	33.95	2.15	134.30	9,749.91
		$async_l$	1,021,545	980,667	26,253	2,028,465	72,569	48.10	22.50	35.07	2.47	108.14	7,847.61
AES	Hard	$sync$	7,126,956	17,771,436	8,505	24,906,897	126,194	244.10	157.60	644.60	0.02	1,046.32	132,039.31
		$async$	8,668,296	23,861,466	70,389	32,600,151	127,360	205.00	184.30	554.10	3.65	947.05	120,616.29
		$async_r$	7,106,868	16,476,984	67,878	23,651,730	128,123	163.60	146.30	557.33	3.62	870.85	111,575.91
		$async_l$	5,020,452	20,498,958	75,843	25,595,253	128,814	128.40	99.20	448.37	4.10	680.07	87,602.54
	Soft	$sync$	7,123,626	17,847,189	17,505	24,988,320	126,167	205.70	147.60	553.50	0.53	907.33	114,475.10
		$async$	8,651,610	24,458,337	69,372	33,179,319	127,002	177.00	167.90	488.39	3.45	836.74	106,267.65
		$async_r$	7,121,196	16,846,452	70,380	24,038,028	127,951	163.90	147.60	541.82	3.72	857.04	109,659.13
		$async_l$	5,025,672	19,307,844	74,466	24,407,982	128,437	128.90	98.60	406.53	3.94	637.97	81,938.95
Total		$sync$	18,561,852	41,312,007	46,980	59,920,839	1,008,540	625.86	405.43	1,389.90	1.14	2,422.33	295,195.74
		$async$	22,522,347	55,173,969	251,874	77,948,190	1,014,956	553.18	480.22	1,277.64	14.20	2,325.24	283,481.43
		$async_r$	18,537,471	39,019,194	248,427	57,805,092	1,016,514	477.40	393.18	1,290.40	14.44	2,175.42	268,894.28
		$async_l$	13,083,345	45,846,432	271,026	59,200,803	1,018,230	374.49	259.03	1,056.91	16.00	1,706.43	211,796.69
Average		$sync$	2,320,231.50	5,164,000.88	5,872.50	7,490,104.88	126,067.50	78.23	50.68	173.74	0.14	302.79	36,899.47
		$async$	2,815,293.38	6,896,746.13	31,484.25	9,743,523.75	126,869.50	69.15	60.03	159.71	1.78	290.66	35,435.18
		$async_r$	2,317,183.88	4,877,399.25	31,053.38	7,225,636.50	127,064.25	59.68	49.15	161.30	1.81	271.93	33,611.78
		$async_l$	1,635,418.13	5,730,804.00	33,878.25	7,400,100.38	127,278.75	46.81	32.38	132.11	2.00	213.30	26,474.59
Average ratio		$async$	20.69%	24.63%	557.27%	23.00%	0.75%	-0.59%	19.03%	13.64%	4,365.01%	9.60%	10.42%
		$async_r$	-17.20%	-20.33%	-1.36%	-18.76%	0.15%	-12.54%	-16.88%	-12.12%	1.10%	-12.27%	-12.12%
		$async_l$	-40.86%	-14.28%	7.22%	-23.22%	0.31%	-30.59%	-47.23%	-15.73%	10.86%	-24.97%	-24.74%

were inserted considering the critical path delays. $async_{op}$ could reduce the dynamic power consumption by 1.72% on average. This result comes from the insertion of D latches to prevent unnecessary operations. Similarly, $async_{op}$ could reduce the energy consumption by 1.38% on average. Also, $async_{opl}$ could reduce the energy consumption by 11.95% on average compared to *async*.

Table 4 shows the evaluation results for the asynchronous circuits with Hard and Soft. Compared to *sync*, the circuit area of *async* was increased by 23.00% on average because DFFs with an enable signal were used. The execution time of *async* was increased by 0.75% on average because the control-path delays were longer than the critical path delays of the data-paths. The dynamic power consumption of *async* was increased by 9.60% on average due to the use of DFFs with an enable signal. Compared to *async* with Hard, the dynamic power consumption of *async* with Soft was low because the number of operations of data-path resources was small by multiple stall signals in the case of Soft. The energy consumption of *async* was increased by 10.42% on average.

On the other hand, compared to *async*, $async_r$ and $async_l$ could reduce the circuit area by 18.76% and 23.22% on average due to the use of DFFs without an enable signal in $async_r$ and the use of D latches instead of DFFs in $async_l$. $async_r$ and $async_l$ did not have a significant impact on the execution time because we assigned the same values for the maximum delay constraints and local clock constraints. Moreover, $async_r$ and $async_l$ could reduce the dynamic power consumption by 12.27% and 24.97% on average. Compared to $async_r$ with Hard and $async_l$ with Hard, the dynamic power consumption of $async_r$ with Soft and $async_l$ with Soft was low because the number of operations of data-path resources was small by multiple stall signals in the case of Soft. $async_r$ and $async_l$ could reduce the energy consumption by 12.12% and 24.74% on average. This result comes from the reduction of the dynamic power consumption.

From the experimental results, the proposed method converted pipelined synchronous RTL models into pipelined asynchronous ones regardless of the difference in the IIs and the type of stalls. In addition, compared to synchronous circuits, the energy consumption of asynchronous circuits without the optimization methods was reduced by 1.47% on average. Moreover, the optimization methods could reduce the energy consumption by 15.12% on average compared to synchronous circuits. Furthermore, the energy consumption of asynchronous circuits with the optimization methods was reduced by up to 34.72% compared to the asynchronous circuits without the optimization methods.

VII. CONCLUSION

In this paper, we proposed a conversion method from pipelined synchronous RTL models into pipelined asynchronous RTL models with bundled-data implementation. In addition, we also proposed optimization methods during the proposed RTL conversion to obtain the high quality of asynchronous circuits.

In the experiment, the proposed method converted pipelined synchronous RTL models into pipelined asynchronous ones regardless of the difference in the II and the type of stalls. Moreover, we confirmed the quality of generated asynchronous RTL models. Compared to synchronous circuits, the energy consumption of asynchronous circuits using the proposed method was reduced by 1.47% on average. On the other hand, the optimization methods could reduce the energy consumption by 15.12% on average compared to synchronous circuits. In addition, the optimization methods reduced the energy consumption by up to 34.72% compared to asynchronous circuits without the optimization methods.

As our future work, we extend the proposed method to deal with pipelined synchronous RTL models including multiple control circuits. In addition, we are going to propose low energy optimization methods during the RTL conversion for pipelined asynchronous circuits. Moreover, we are going to reduce the conversion time.

REFERENCES

[1] 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.

[2] 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.

[3] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "Handshake protocols for de-synchronization," in *Proc. 10th Int. Symp. Asynchronous Circuits Syst.*, 2004, pp. 149–158.

[4] S. K. Srinivasan and R. S. Katti, "Desynchronization: Design for verification," in *Proc. FMCAD*, Oct. 2011, pp. 215–222.

[5] 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.

[6] R. Zhou, K.-S. Chong, B.-H. Gwee, and J. S. Chang, "Quasi-delay-insensitive compiler: Automatic synthesis of asynchronous circuits from verilog specifications," in *Proc. IEEE 54th Int. Midwest Symp. Circuits Syst. (MWSCAS)*, Aug. 2011, pp. 1–4.

[7] R. B. Reese, S. C. Smith, and M. A. Thornton, "Uncle—An RTL approach to asynchronous design," in *Proc. IEEE 18th Int. Symp. Asynchronous Circuits Syst.*, May 2012, pp. 65–72.

[8] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," in *Proc. 6th Int. Symp. Adv. Res. Asynchronous Circuits Syst. (ASYNC)*, 2000, pp. 114–125.

[9] A. Kondratyev and K. Lwin, "Design of asynchronous circuits by synchronous CAD tools," in *Proc. Design Autom. Conf.*, Aug. 2002, pp. 411–414.

[10] J. Oberg, J. Plosila, and P. Ellervee, "Automatic synthesis of asynchronous circuits from synchronous RTL descriptions," in *Proc. NORCHIP*, 2005, pp. 200–205.

[11] M. K. Fant and A. S. Brandt, "NULL conventional logic: A complete and consistent logic for asynchronous digital circuit synthesis," *Proc. Int. Conf. Appl. Specific Syst., Archit. Processors*, 1996, pp. 261–273.

[12] G. E. Sobelman and K. Fant, "CMOS circuit design of threshold gates with hysteresis," in *Proc. ISCAS IEEE Int. Symp. Circuits Syst.*, May 1998, pp. 61–64.

[13] M. L. L. Sartori, M. T. Moreira, and N. L. V. Calazans, "A frontend using traditional EDA tools for the pulsar QDI design flow," in *Proc. 26th IEEE Int. Symp. Asynchronous Circuits Syst. (ASYNC)*, May 2020, pp. 3–10.

[14] M. L. L. Sartori, R. N. Wuerdig, M. T. Moreira, and N. L. V. Calazans, "Pulsar: Constraining QDI circuits cycle time using traditional EDA tools," in *Proc. 25th IEEE Int. Symp. Asynchronous Circuits Syst. (ASYNC)*, May 2019, pp. 114–123.

[15] 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.

[16] A. Taubin, J. Cortadella, L. Lavagno, A. Kondratyev, and A. Peeters, "Design automation of real-life asynchronous devices and systems," *Found. Trends Electron. Des. Automat.*, vol. 2, no. 1, pp. 1–133, 2007.

[17] K. Garcia, D. L. Oliveira, T. Curtinhas, and R. d'Amore, "Synthesis of locally-clocked asynchronous systems with bundled-data implementation on FPGAs," in *Proc. 9th Southern Conf. Program. Log. (SPL)*, Nov. 2014, pp. 1–6.

[18] T. Curtinhas, D. L. Oliveira, and O. Saotome, "VHDLASYN: A tool for synthesis of asynchronous systems from of VHDL behavioral specifications," in *Proc. 2nd Conf. PhD Res. Microelectron. Electron. Latin Amer. (PRIME-LA)*, Feb. 2018, pp. 1–4.

[19] M. Sacker, A. D. Brown, P. R. Wilson, and A. J. Rushton, "A general purpose behavioural asynchronous synthesis system," in *Proc. 10th Int. Symp. Asynchronous Circuits Syst.*, Apr. 2004, pp. 125–134.

[20] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits: Part I (specification and hazard-free implementations)," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 2, pp. 101–117, Feb. 1999.

[21] R. Manohar, "An open-source design flow for asynchronous circuits," in *Proc. Government Microcircuit Appl. Crit. Technol. Conf. (GOMACTech)*, 2019, pp. 1–5.

[22] S. Semba and H. Saito, "Conversion from synchronous RTL models to asynchronous RTL models," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. E102.A, no. 7, pp. 904–913, 2019.

[23] S. Semba and H. Saito, "Study on an RTL conversion method from pipelined synchronous RTL models into asynchronous RTL models," in *Proc. SASIMI*, 2021, pp. 229–234.

[24] S. Semba, H. Saito, M. Tatsuoka, and K. Fujimura, "Optimization methods during RTL conversion from synchronous RTL models to asynchronous RTL models," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. E103.A, no. 12, pp. 1417–1426, 2020.

[25] A. Peeters, F. te 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.

[26] S. Takamaeda-Yamazaki, "Pyverilog: A Python-based hardware design processing toolkit for verilog HDL," in *Applied Reconfigurable Computing* (Lecture Notes in Computer Science), vol. 9040. Springer, 2015, pp. 451–460.

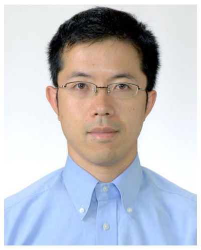
[27] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 65–74.

[28] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *J. Inf. Process.*, vol. 17, pp. 242–254, Oct. 2009.

[29] *Stratus High-Level Synthesis User Guide, Product Version 18.1*, Cadence, San Jose, CA, USA, 2018.

[30] K. Yoshimi and H. Saito, "A delay adjustment method for asynchronous circuits with bundled-data implementation considering a latency constraint," in *Proc. SASIMI*, 2016, pp. 219–224.

SHOGO SEMBA (Member, IEEE) received the B.S. and M.S. degrees in computer science and engineering from The University of Aizu, in 2017 and 2019, respectively, where he is currently pursuing the Ph.D. degree. His research interest includes asynchronous circuit design.



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 currently 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.