Study on an RTL Conversion Method from Pipelined Synchronous RTL Models into Asynchronous RTL Models.

Shogo Semba
The University of Aizu, Japan
d8211108@u-aizu.ac.jp

Hiroshi Saito
The University of Aizu, Japan
hiroshis@u-aizu.ac.jp

Abstract— In this paper, we propose a conversion method from pipelined synchronous Register Transfer Level (RTL) models into asynchronous RTL models with bundled-data implementation. The proposed method generates a control data flow graph (CDFG) from a given synchronous RTL model. After generating the CDFG, the proposed method generates an asynchronous RTL model by analyzing each pipeline stage on the CDFG, assigning asynchronous control modules, and connecting the control modules. In the experiment, we converted four pipelined synchronous RTL models into asynchronous ones. In addition, we performed logic synthesis for the converted asynchronous RTL models. The synthesized asynchronous circuits could reduce the energy consumption by 3.6% on average.

I. Introduction

Most of the digital integrated circuits are synchronous circuits in which circuit components are controlled by global clock signals. Synchronous circuits face problems when the semiconductor miniaturization technology is advanced more and more. One is synchronization failures caused by clock skew. Another is the increase of the power consumption due to the distribution of clock signals with high frequency to wide area.

In asynchronous circuits, circuit components are controlled by local handshake signals or self-timings instead of global clock signals. Therefore, asynchronous circuits are potentially low power consumption compared to synchronous circuits. However, the design of asynchronous circuits is more difficult than the design of synchronous circuits because design methods and design constraints are different according to the selection of data encoding, handshake protocol, and delay model.

To facilitate the design of asynchronous circuits, conversion methods from synchronous Gate-Level (GL) netlists into asynchronous ones were proposed in [1–6]. However, logic optimization considering the characteristics of asynchronous circuits cannot be performed in the GL conversion methods, 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 [7]. Compared to the GL conversion methods, the RTL conversion method can optimize asynchronous circuits by assigning constraints considering the characteristics of asynchronous circuits during logic synthesis. However, the RTL conversion method cannot deal with pipelined synchronous RTL models. In many applications, pipelined circuits are adopted to increase the performance. Hence, by converting pipelined synchronous RTL models into asynchronous ones, we design low power circuits more than synchronous circuits.

In this paper, we propose a conversion method from pipelined synchronous RTL models into asynchronous RTL models. The proposed method is an extension of [7]. The proposed method generates a control data flow graph (CDFG) from a synchronous RTL model. After generating the CDFG, the proposed method assigns asynchronous control modules by analyzing each pipeline stage in the CDFG. Then, the proposed method generates an asynchronous RTL model by connecting the control modules to the data-path resources.

The proposed method generates CDFGs from pipelined synchronous RTL models before conversion which is not generated in [7]. In [7], asynchronous control modules are assigned directly by referring to the states of the finite state machine (FSM) in non-pipelined synchronous RTL models. However, it is difficult to assign asynchronous control modules directly for pipelined synchronous RTL models because they do not have an FSM or the number of states in the FSM is not equal to the number of pipeline stages. Therefore, the proposed method generates CDFGs to assign asynchronous control modules.

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

II. Asynchronous Circuits with Bundled-data Implementation

Bundled-data implementation is one of the data encoding schemes in asynchronous circuits. In the bundled-data implementation, one-bit data is represented by one signal. The timing to write data to registers is guaranteed by delay elements on request signals *req* in a control circuit. Hence, the performance of the bundled-data implementation depends on the delay of the control circuit including the delay elements.

Figure 1 shows the circuit model of asynchronous circuits with bundled-data implementation used in this

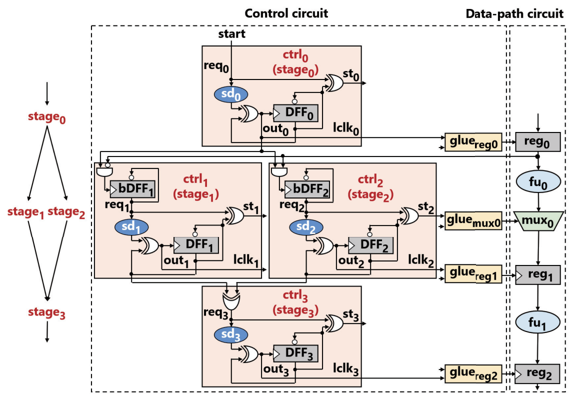
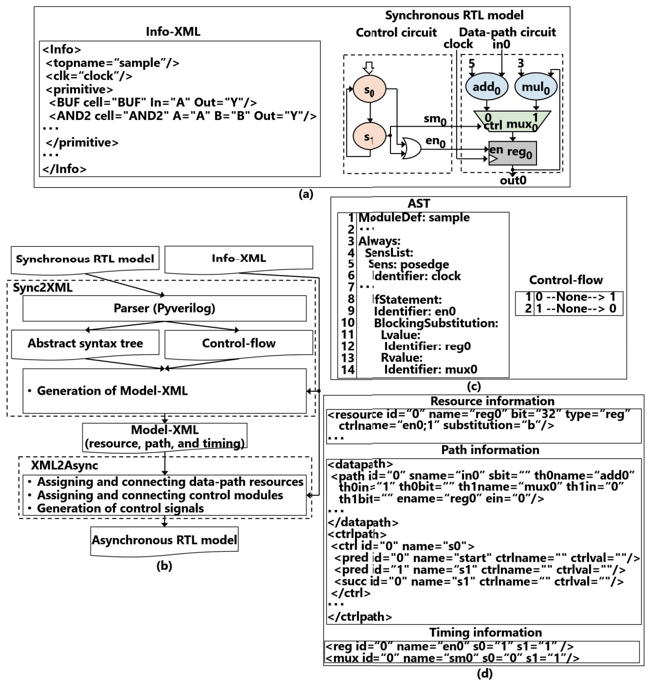


Fig. 1. Asynchronous circuits with bundled-data implementation.



```
<Info>
 <topname="sample"/>
 <clk="clock"/>
 <primitive>
  <BUF cell="BUF" In="A" Out="Y"/>
  <AND2 cell="AND2" A="A" B="B" Out="Y"/>
...
 </primitive>
...
</Info>
```

```
ModuleDef: sample
...
Always:
 SensList:
  Sens: posedge
   Identifier: clock
...
   IfStatement:
   Identifier: en0
   BlockingSubstitution:
    Lvalue:
     Identifier: reg0
    Rvalue:
     Identifier: mux0
```

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

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

```
<datapath>
 <path id="0" sname="in0" sbit="" th0name="add0"
  th0in="1" th0bit="" th1name="mux0" th1in="0"
  th1bit="" ename="reg0" ein="0"/>
...
</datapath>
<ctrlpath>
 <ctrl id="0" name="s0">
  <pred id="0" name="start" ctrlname="" ctrlval=""/>
  <pred id="1" name="s1" ctrlname="" ctrlval=""/>
  <succ id="0" name="s1" ctrlname="" ctrlval=""/>
 </ctrl>
...
</ctrlpath>
```

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

Fig. 2. RTL conversion method in [7]: (a) Info-XML and synchronous RTL model, (b) RTL conversion flow, (c) AST and control flow, and (d) Model-XML.

work. This circuit model consists of a data-path circuit and a control circuit.

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.

The control circuit consists of control modules $ctrl_i$ ($0 \leq i \leq n-1$) assigned for each pipeline stage $stage_i$. $glue_{reg_k}$ and $glue_{mux_l}$ are logics to control reg_k and mux_l.

$ctrl_i$ is an extended Click element [8]. It consists of a D flip-flop DFF_i, two XOR gates, and a delay element sd_i. If there are control branches, a D flip-flop $bDFF_i$ and an AND gate are inserted before sd_i. In $ctrl_i$, the acknowledgment signal used in traditional asynchronous circuits is not used. Only the request signal is used for succeeding control modules. Hence, each $ctrl_i$ is operated by a 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 registers by the rising transition of $lclk_i$.

The control circuit starts its operation when a rising transition of the input signal $start$ arrives at the control circuit. $ctrl_i$ 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 through one of the XOR gates. st_i controls mux_l through $glue_{mux_l}$. req_i also generates a rising transition of $lclk_i$ through sd_i and the other 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 falling transitions of $lclk_i$ and st_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.

III. RTL Conversion

The RTL conversion method in [7] generates asynchronous RTL models from synchronous RTL models described by Verilog Hardware Description Language (HDL) 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 2(a) shows a part of the Info-XML and the structure of a synchronous RTL model. Figure 2(b) shows the RTL conversion flow in [7].

Sync2XML generates the abstract syntax tree (AST) and the control flow from given synchronous RTL models through *Pyverilog* [9]. Figure 2(c) shows a part of the AST and the control flow for the synchronous RTL model in Fig.2(a). The AST represents the structure of RTL models while the control flow represents state transitions.

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 including data-paths and control-paths, and timing information including register write signals and multiplexer control signals as shown in Fig.2(d).

After generating the Model-XML, *XML2Async* generates an asynchronous RTL model with bundled-data implementation from the Model-XML. *XML2Async* assigns data-path resources and connects the data-path resources from the resource and path information. *XML2Async* also assigns control modules and connects the control modules from the path information. Finally, to connect the control modules to the data-path resources, *XML2Async* generates register write signals and multiplexer control signals from the timing information.

IV. Proposed Method

We extend the RTL conversion method described in Sec.III. Figure 3 shows the extended RTL conversion flow.

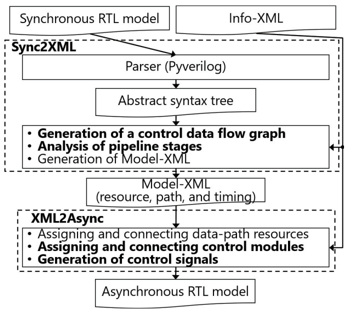

Synchronous RTL model
Info-XML
Sync2XML
Parser (Pyverilog)
Abstract syntax tree
• Generation of a control data flow graph
• Analysis of pipeline stages
• Generation of Model-XML
Model-XML
(resource, path, and timing)
XML2Async
• Assigning and connecting data-path resources
• Assigning and connecting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

Fig. 3. Extended RTL conversion flow.

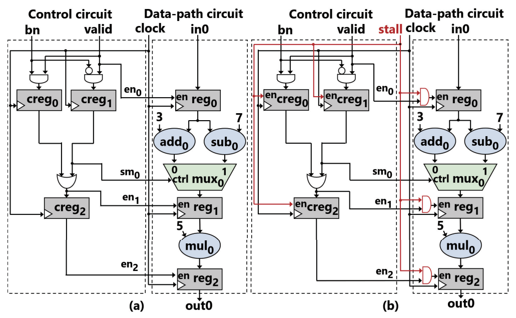


Fig. 4. Pipelined synchronous RTL models: (a) without pipeline stalls and (b) with pipeline stalls.

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 data-path circuit must be composed of registers, functional units, and multiplexers as shown in Fig.1. The synchronous RTL models are assumed to have only one control circuit. The input interval for the synchronous RTL models is only one cycle. The proposed method does not care whether forwarding operation is included or not. Syntax such as "function", "task", "for", "while", "wait", and "[sub,5'h0+:32] (concatenation)" must not be included in Verilog HDL. To deal with the syntax is our future work.

Figure 4 shows examples of pipelined synchronous RTL models. The structure of RTL models may be changed by depending on whether the RTL models include pipeline stalls or not. Figures 4(a) and (b) show RTL models without pipeline stalls and with pipeline stalls. If pipeline stalls are included in synchronous RTL models, pipeline stages stall the operation during the stall. Note that $creg_0$, $creg_1$, and $creg_2$ represent registers in the control circuit.

B. *Extension of Sync2XML*

In asynchronous circuits, data-path resources in each $stage_i$ are controlled by each $ctrl_i$. To know the data-path resources controlled by each $stage_i$, *Sync2XML* generates a CDFG from the AST generated by *Pyverilog*. After generating the CDFG, *Sync2XML* analyzes each $stage_i$ in the CDFG. Then, *Sync2XML* extracts the pipeline stage information and timing information, and generates them into the Model-XML.

B.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 and data flow graph. The CDFG consists of nodes, edges, and $stage_i$ as shown in Fig.5. The nodes represent resources such as registers and functional units in the data-path circuit and registers and basic logic operations in the control circuit. The nodes except functional units and basic logic operations have a control signal name (the left side of the nodes in Fig.5) and its value (the upper side of the nodes in Fig.5). 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 $stage_i$.

Sync2XML generates nodes from "Lvalue" or "Instance" in the AST and edges from "Rvalue" or "PortArg" in the AST. *Sync2XML* also extracts the label for the nodes from the variable name and the control signal with its value for the nodes from "IfStatement" or "CaseStatement" in the AST which represents branches.

Sync2XML regards between registers as $stage_i$. Then, *Sync2XML* extracts *cond* and *val* for $stage_i$ from "Rvalue", "IfStatement", and "CaseStatement" for the control circuit in the AST. *Sync2XML* also labels the generated pipeline stages in the order from $stage_0$.

On the other hand, the extraction method for *cond* and *val* for $stage_i$ is different depending on whether there are stall signals or not. When there is no stall signal, *Sync2XML* extracts *cond* and *val* from the AST. When there are stall signals, the extraction method is different depending on whether there are multiple stall signals or one stall signal. *Sync2XML* does not extracts *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 if there are multiple stall signals.

Figure 5 shows the generated CDFG for Fig.4(a). For example of node generations, the node of reg_1 is generated from "Lvalue" for reg_1 in the AST. The control signal en_1 is given to the node from "IfStatement". For example of edge generations, the edge from mux_0 to reg_1 is generated from "Rvalue" for reg_1 in the AST. For example of $stage_i$ generations, we regard between reg_0 and reg_1 through add_0 as $stage_1$. The conditional signal *bn* and its value 1 are given to $stage_1$ from "Rvalue" for $creg_0$ in the AST.

B.2. Analysis of pipeline stages

For the generated CDFG, *Sync2XML* analyzes preceding and succeeding pipeline stages for each $stage_i$.

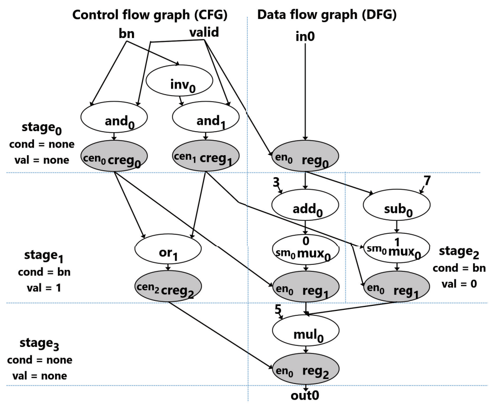
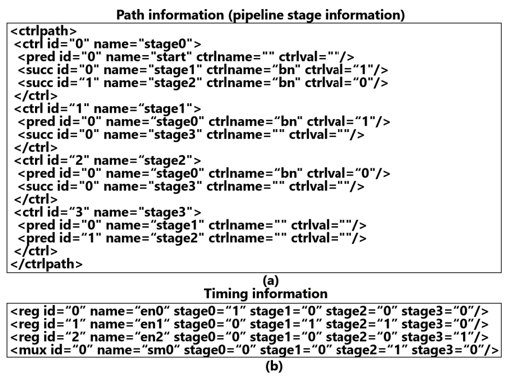


Fig. 5. CDFG for the synchronous RTL model in Fig.4(a).


Path information (pipeline stage information)

```
<ctrlpath>
 <ctrl id="0" name="stage0">
  <pred id="0" name="start" ctrlname="" ctrlval=""/>
  <succ id="0" name="stage1" ctrlname=“bn" ctrlval=“1"/>
  <succ id=“1" name="stage2" ctrlname=“bn" ctrlval=“0"/>
 </ctrl>
 <ctrl id=“1" name=“stage1">
  <pred id="0" name=“stage0" ctrlname=“bn" ctrlval=“1"/>
  <succ id="0" name="stage3" ctrlname="" ctrlval=""/>
 </ctrl>
 <ctrl id=“2" name=“stage2">
  <pred id="0" name=“stage0" ctrlname=“bn" ctrlval=“0"/>
  <succ id="0" name="stage3" ctrlname="" ctrlval=""/>
 </ctrl>
 <ctrl id=“3" name="stage3">
  <pred id="0" name=“stage1" ctrlname="" ctrlval=""/>
  <pred id=“1" name=“stage2" ctrlname="" ctrlval=""/>
 </ctrl>
</ctrlpath>
```

(a)

Timing information

```
<reg id=“0” name=“en0“ stage0=“1” stage1=“0” stage2=“0” stage3=“0”/>
<reg id=“1” name=“en1“ stage0=“0” stage1=“1” stage2=“1” stage3=“0”/>
<reg id=“2” name=“en2“ stage0=“0” stage1=“0” stage2=“0” stage3=“1”/>
<mux id=“0” name=“sm0” stage0=“0” stage1=“0” stage2=“1” stage3=“0”/>
```

(b)

Fig. 6. Model-XML generated from CDFG in Fig.5: (a) pipeline stage information and (b) timing information.

Sync2XML also analyzes register write signals and multiplexer control signals. After analyzing the CDFG, *Sync2XML* generates pipeline stage information and timing information into the Model-XML.

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 $stage_j$.

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 pipeline stage information and timing information into the Model-XML. In the pipeline stage information, *Sync2XML* generates the pipeline stage information for each $stage_i$ using $\langle ctrl \rangle$. *Sync2XML* also generates preceding pipeline stage information $\langle pred \rangle$ and succeeding pipeline stage information $\langle succ \rangle$ into $\langle ctrl \rangle$. 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 $\langle succ \rangle$. On the other hand, in the timing information, *Sync2XML* generates the register write signal information or the multiplexer control signal information for each control signal of registers and multiplexers using $\langle mux \rangle$ or $\langle reg \rangle$. Finally, *Sync2XML* assigns the control values to $\langle reg \rangle$ and $\langle mux \rangle$ from the analyzed values of the control signals.

Figure 6 shows the generated Model-XML from the CDFG in Fig.5. In the pipeline stage information, *Sync2XML* generates four $\langle ctrl \rangle$. For $\langle ctrl \rangle$ corresponding to $stege_1$, *Sync2XML* assigns $stege_0$ to $\langle pred \rangle$, *bn* to *ctrlname*, and 1 to *ctrlval* because the preceding pipeline stage for $stege_1$ is $stege_0$. *Sync2XML* also assigns $stege_3$ to $\langle succ \rangle$ because the succeeding pipeline stage for $stege_1$ is $stege_3$. In the timing information, *Sync2XML* assigns the register write signal en_1 whose values of $stage_1$ and $stage_2$ are 1 to $\langle reg \rangle$ because there is reg_1 in $stage_1$ and $stage_2$ 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. All resources are represented by Verilog HDL.

C.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 $\langle ctrl \rangle$.

$ctrl_i$ in Fig.7(a) represents assigned $ctrl_i$ by referring to $\langle ctrl \rangle$ in Fig.6(a). *XML2Async* assigns four $ctrl_i$ for four $\langle ctrl \rangle$. 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 $ctrl_3$ by referring to $\langle succ \rangle$ in $\langle ctrl \rangle$. *XML2Async* connects all control modules in the same way.

C.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. Similarly, the assignment of multiplexer control signals consists of the logical OR of st_i where $stage_i$ in $\langle mux \rangle$ is equal to 1.

The control signals in Fig.7(a) represent generated control signals by referring to $\langle reg \rangle$ and $\langle mux \rangle$ in Fig.6(b). For example of the generation of the register write signal en_1 for reg_1, the assignment of en_1 consists of the logical OR of $lclk_1$ and $lclk_2$ because the values of $stage_1$ and $stage_2$ in $\langle reg \rangle$ are 1. Figure 7 shows converted asynchronous RTL models from the synchronous RTL models in Fig.4.

TABLE I
RTL conversion results.

Name	*Stall*	*CT* [ps]	*Stage*	*Sverilog* [lines]	*AST* [lines]	*Model-XML* [lines]	*Averilog* [lines]	*Time* [s]
DIFFEQ	None	1,200	4	164	765	110	359	1.9
	Hard	1,200	4	209	859	188	437	2.1
	Soft	1,200	4	208	922	177	457	2.0
EWF	None	1,200	9	668	3,202	429	1,345	2.5
	Hard	1,200	9	859	3,588	746	1,624	2.8
	Soft	1,200	9	825	3,703	715	1,674	2.7
MLP	None	400	20	16,925	94,130	22,276	36,668	283.1
	Hard	400	20	20,734	101,164	28,039	40,497	341.9
	Soft	400	20	18,863	99,661	27,974	40,613	338.1
AES	None	600	41	130,036	94,6154	192,782	138,675	2992.3
	Hard	600	41	133,499	95,5070	198,594	142,578	3130.8
	Soft	600	41	132,152	95,4490	198,474	142,820	3172.3

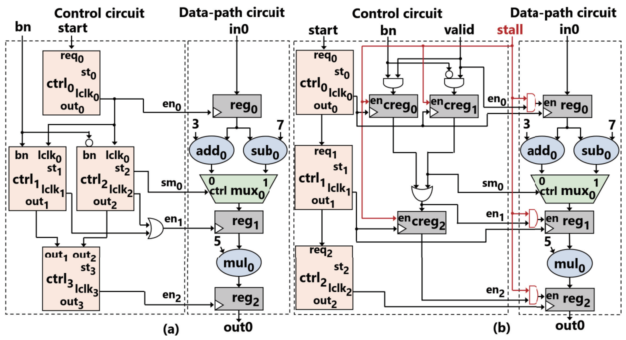


Fig. 7. Pipelined asynchronous RTL models: (a) for Fig.4(a) and (b) for Fig.4(b).

V. Experimental Results

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

For the experiment, 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 are a differential equation solver (DIFFEQ), an elliptic wave filter (EWF), a multilayer perceptron (MLP) [10] whose the number of neuron is 32, and the advanced encryption standard (AES) [11]. In addition, we prepared synchronous RTL models with a hard stall (Hard) and a soft stall (Soft) by applying directives [12] in Stratus HLS. Hard means that the operations of all pipeline stages stall while Soft means that the operations of specified pipeline stages stall. We also applied clock gating option to HLS for synchronous RTL models. The library was eShuttle 65nm process technology.

Table I shows the conversion results using the proposed method. *Stall*, *CT*, *Stage*, and *Sverilog* represent the type of stalls, 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. Note that the clock cycle time of the synchronous RTL models represents the shortest one without timing violations during HLS.

From Table I, the conversion time depends on the number of pipeline stages and the number of lines in the AST. Compared to the conversion time for the RTL models without stalls, the conversion time for the RTL models with stalls was increased because the number of lines in the AST with stalls is more than the number of lines in the AST without stalls.

To verify the functional correctness of the converted asynchronous RTL models, we performed logic simulation using Synopsys VCS Q-2020.03-SP1. To perform the simulation, we prepared a test bench with 100 arbitrary test patterns. For the simulation, we generated a Standard Delay Format file by synthesizing the asynchronous RTL models using Cadence Genus 18.1. 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.

To check the quality of the converted asynchronous RTL models (*async*), we performed logic synthesis based on the design flow in [13]. To obtain the same performance as the synchronous circuits (*sync*), we generated the maximum delay constraints for the control-paths and the local clock constraints for each pipeline stage in the asynchronous RTL models from the clock cycle time of the synchronous circuits. The clock cycle time of the synchronous circuits for DIFFEQ, EWF, MLP, and AES were $1,400$ ps, $1,500$ ps, 600 ps, and 900 ps, respectively. The clock cycle time of the synchronous RTL models represents the shortest one without timing violations during logic synthesis.

Figure 8(a) shows the circuit area obtained by Genus. The circuit area of *async* was reduced by 0.9% on average. This is because the structure of the data-path circuit was changed by assigning the constraints which are different from the constraints of the synchronous circuit. However, the area of the control circuit of *async* was increased by inserting the control modules. Whether there are stalls or not did not have a significant impact on the circuit area.

Figure 8(b) shows the execution time obtained by logic simulation. In Hard and Soft, 20 cycles were stalled. The execution time of *async* was increased by 2.0% on average. This is because the cycle time of *async* was slightly

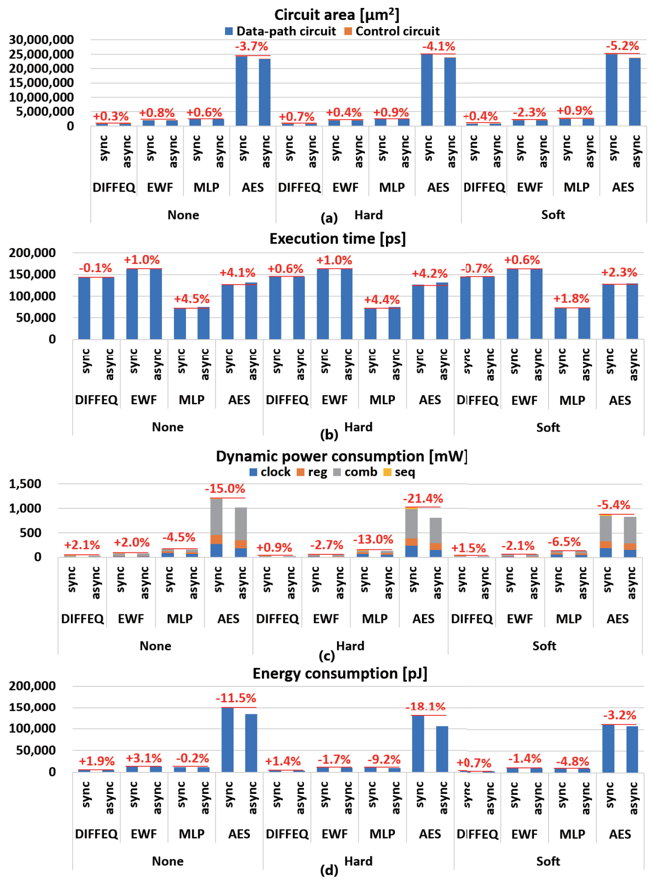


Fig. 8. Evaluation results: (a) circuit area, (b) execution time, (c) dynamic power consumption, and (d) energy consumption.

slower than the cycle time of *sync* due to the insertion of the delay element in which the delay was longer than the critical path delays of the data-paths. Whether there are stalls or not did not have a significant impact on the execution time.

Figure 8(c) shows the dynamic power consumption obtained by Synopsys PrimeTime Q-2019.12-SP3 with the Value Change Dump file generated by VCS. *async* without stalls could reduce the dynamic power consumption by 3.9% on average. This is because the dynamic power consumption of the clock networks was reduced due to the use of the asynchronous control modules instead of the global clock signals. Moreover, the dynamic power consumption of the registers and combinational circuits was reduced because the only required circuit components are operated in asynchronous circuits. Also, *async* with stalls could reduce the dynamic power consumption by 6.1% on average because the dynamic power consumption of logics for stalls in *async* was reduced.

Figure 8(d) shows the energy consumption obtained by multiplying the execution time and the dynamic power consumption. *async* could reduce the energy consumption by 3.6% on average because the reduction effect of the dynamic power consumption was higher than the increase of the execution time.

VI. Conclusion

In this paper, we proposed a conversion method from pipelined synchronous RTL models into asynchronous RTL models with bundled-data implementation. In the experiment, we converted four pipelined synchronous RTL models into asynchronous ones. Compared to synchronous circuits, the energy consumption of asynchronous circuits using the proposed method was reduced by 3.6% on average.

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.

Acknowledgements

This work is partially supported by Grant-in-Aid for Scientific Research from Japan Society for the promotion of science (#18K11221 and #20J11724).

References

[1] J. Cortadella et al., "Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications", IEEE TCAD, vol. 25, pp. 1904–1921, 2006.

[2] N. Andrikos et al., "A Fully-Automated Desynchronization Flow for Synchronous Circuits", Proc. DAC, pp. 982–985, 2007.

[3] A. Kondratyev and K. Lwin, "Design of Asynchronous Circuits by Synchronous CAD Tools", Proc. DAC, pp.411–414, 2002.

[4] R. Zhou et al., "Quasi-Delay-Insensitive Compiler: Automatic Synthesis of Asynchronous Circuits from Verilog Specifications", Proc. NWSCAS, pp. 1–4, 2011.

[5] A. Branover et al., "Asynchronous Design By Conversion: Converting Synchronous Circuits into Asynchronous Ones", Proc. DATE, pp. 870–875, 2004.

[6] R. B. Reese et al., "Uncle - An RTL Approach to Asynchronous Design", Proc. ASYNC, pp. 65–72, 2012.

[7] S. Semba and H. Saito, "Conversion from Synchronous RTL Models to Asynchronous RTL Models", IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, vol. E102-A, No. 7, pp. 904–913, 2019.

[8] Ad Peeters et al., "Click Elements: An Implementation Style for Data-Driven Compilation", Proc. ASYNC, pp. 3–14, 2010.

[9] S. Takamaeda-Yamazaki, "Pyverilog: A Python-based Hardware Design Processing Toolkit for Verilog HDL", Proc. ARC, Lecture Notes in Computer Science, Vol.9040/2015, pp.451–460, 2015.

[10] Y. Umuroglu et al., "FINN: A Framework for Fast, Scalable Binarizrd Neural Network Inference", Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, pp. 65–74, 2017.

[11] Yuko Hara et al., "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis", Journal of Information Processing, vol. 17, pp. 242–254, 2009.

[12] Cadence, "Stratus High-Level Synthesis User Guide", product version 18.1, 2018.

[13] S. Semba and H. Saito, "Comparison of RTL Conversion and GL Conversion from Synchronous Circuits to Asynchronous Circuits", Proc. ISCAS, pp. 1–4, 2019.