

A dissertation
submitted in partial satisfaction of the requirements
for the degree of Doctor of Computer Science and Engineering

**Automatic Conversion from Synchronous RTL
Models to Asynchronous RTL Models**



by

d8211108 Shogo Semba

Graduate Department of Computer and Information Systems

The University of Aizu

March 2022

© Copyright by d8211108 Shogo Semba, March 2022

All Rights Reserved.

The dissertation titled

*Automatic Conversion from Synchronous RTL Models to
Asynchronous RTL Models*

by

d8211108 Shogo Semba

is reviewed and approved by:

Main referee

Senior Associate Professor

Hiroshi Saito

Hiroshi Saito 

Feb 10, 2022

Professor

Ben Abdallah Abderazek

A. Ben Abdallah 

Feb. 9, 2022

Senior Associate Professor

Yukihide Kohira

Yukihide Kohira 

Feb. 9, 2022

Senior Associate Professor


Yoichi Tomioka

Yoichi Tomioka 

Feb. 9, 2022

Professor

Tomohiro Yoneda

Tomohiro Yoneda 

Feb. 3, 2022

THE UNIVERSITY OF AIZU

March 2022

Contents

Chapter 1 Introduction	1
1.1 Research Background	1
1.2 Purpose of this Dissertation	2
1.3 Organization of this Dissertation	3
Chapter 2 Related Research	4
Chapter 3 Asynchronous Circuits with Bundled-data Implementation	8
3.1 Circuit Model	8
3.2 Timing Constraints	11
3.2.1 Setup Constraints	11
3.2.2 Hold Constraints	13
3.2.3 Branch Constraints	16
3.2.4 Pulse-Width Constraints	17
Chapter 4 Conversion Method	18
4.1 Overview	18
4.2 Inputs	22
4.2.1 Target Synchronous RTL Models	22
4.2.2 Info-XML	24
4.3 Sync2XML	25
4.3.1 Parser	26
4.3.2 Generation of a CDFG	26
4.3.3 Generation of Model-XML	32
4.4 XML2Async	37
4.4.1 Generation of Asynchronous RTL Models	39
4.4.2 Generation of Asynchronous RTL Simulation Models	46
4.4.3 Generation of a Set of Non-Optimization Constraints	48
Chapter 5 Optimization Methods	49
5.1 Modularization for Data-Path Resources	49
5.2 Use of Appropriate DFFs	51
5.3 Latch Insertion as Operand Isolation	53
5.3.1 Latch Insertion Algorithm	56
5.3.2 Latch Control Signal	59
5.3.3 Timing Constraints	60
5.4 Conversion from DFFs into D latches	62
5.4.1 Timing Constraints	62
Chapter 6 Experimental Results	67

6.1	Preparation	67
6.2	Conversion Results	70
6.3	Evaluation after Logic Synthesis	70
6.3.1	Comparison of <i>sync</i> and <i>RTLsync</i>	70
6.3.2	Evaluation of Proposed Optimization Methods	72
6.3.3	Comparison of the RTL Conversion Method and the GL Conversion Method	80
6.3.4	Discussion	85
Chapter 7 Conclusion and Future Work		87
7.1	Conclusion	87
7.2	Future Work	87

List of Figures

Figure 1.1	Desynchronization	2
Figure 3.1	Asynchronous circuits with bundled-data implementation.	9
Figure 3.2	Circuit models of asynchronous circuits with bundled-data implementation used in this dissertation.	10
Figure 3.3	Control module $ctrl_i$	11
Figure 3.4	Timing diagram of $ctrl_i$ used in this dissertation.	11
Figure 3.5	Paths related to setup constraints.	12
Figure 3.6	Paths related to local cycle time.	13
Figure 3.7	Paths related to hold constraints for non-pipelined circuits.	14
Figure 3.8	Paths related to hold constraints for pipelined circuits.	15
Figure 3.9	Paths related to branch constraint.	16
Figure 3.10	Path related to pulse-width constraint.	17
Figure 4.1	Flow of proposed RTL conversion method.	19
Figure 4.2	Conversion flow.	20
Figure 4.3	Comparison of conversions.	21
Figure 4.4	Example of non-pipelined synchronous RTL model.	23
Figure 4.5	Example of pipelined synchronous RTL model.	24
Figure 4.6	Example of Info-XML.	25
Figure 4.7	Example of AST generated using <i>Pyverilog</i> [33].	27
Figure 4.8	Example of control flow generated by <i>Pyverilog</i> [33].	28
Figure 4.9	Example of the generation of a CFG from a control flow.	29
Figure 4.10	Example of the generation of a CFG from an AST.	30
Figure 4.11	Example of the generation of a DFG from an AST.	31
Figure 4.12	Example of the generation of a CDFG.	33
Figure 4.13	Generated CDFG from Fig. 4.5.	34
Figure 4.14	Example of resource information.	35
Figure 4.15	Example of data-path information.	36
Figure 4.16	Example of control-path information.	38
Figure 4.17	Example of timing information.	39
Figure 4.18	Generated Verilog HDL model of the register.	40
Figure 4.19	Generated control module $ctrl_i$	41
Figure 4.20	Generation of glue logics $w0$	42
Figure 4.21	Instantiation of resources.	44
Figure 4.22	Connection of resources.	44
Figure 4.23	Connection from data-path resources to control modules.	45
Figure 4.24	Generation of glue logics.	45
Figure 4.25	Generated top-level module.	46
Figure 4.26	Asynchronous RTL simulation model for $ctrl_i$	47

Figure 4.27	Example of a set of non-optimization constraints for ASIC im- plementations.	48
Figure 5.1	Example of the maximum delay constraints.	50
Figure 5.2	Modularization of data-path resources.	52
Figure 5.3	Area estimation for <i>reg₂</i>	54
Figure 5.4	Example of operand isolations.	55
Figure 5.5	Example of unnecessary power consumption for functional units.	55
Figure 5.6	Delay parameter for each data-path resource.	56
Figure 5.7	Latch insertion for non-pipeined circuits.	58
Figure 5.8	Latch insertion for pipeined circuits.	59
Figure 5.9	Latch controller.	60
Figure 5.10	Timing diagram for <i>lst_i</i>	60
Figure 5.11	Paths related to setup constraint.	61
Figure 5.12	Paths related to hold constraint.	61
Figure 5.13	Examples of the structures of registers and D latches.	62
Figure 5.14	Paths related to setup constraints for D latches.	63
Figure 5.15	Paths related to hold constraints for non-pipelined circuits.	64
Figure 5.16	Paths related to hold constraints for pipelined circuits.	66
Figure 6.1	Architecture of LENET used in this research.	68
Figure 6.2	Logic design flow.	69
Figure 6.3	Functional verification flow.	72
Figure 6.4	Waveforms of EWFs for non-pipelined circuits.	73
Figure 6.5	Circuit areas of <i>sync</i> and <i>RTLAsync</i>	74
Figure 6.6	Execution times of <i>sync</i> and <i>RTLAsync</i>	74
Figure 6.7	Dynamic power consumptions of <i>sync</i> and <i>RTLAsync</i>	75
Figure 6.8	Energy consumptions of <i>sync</i> and <i>RTLAsync</i>	75
Figure 6.9	Evaluation of the modularization for data-path resources.	76
Figure 6.10	Evaluation of the use of appropriate DFFs.	77
Figure 6.11	Circuit area of operand isolation by latches.	78
Figure 6.12	Execution time of operand isolation by latches.	78
Figure 6.13	Dynamic power consumption of operand isolation by latches.	79
Figure 6.14	Energy consumption of operand isolation by latches.	79
Figure 6.15	Circuit area of the conversion from DFFs into D latches.	80
Figure 6.16	Execution time of the conversion from DFFs into D latches.	80
Figure 6.17	Dynamic power consumption of the conversion from DFFs into D latches.	81
Figure 6.18	Energy consumption of the conversion from DFFs into D latches.	81
Figure 6.19	Circuit area of the combination of the optimization methods.	82
Figure 6.20	Execution time of the combination of the optimization methods.	82
Figure 6.21	Dynamic power consumption of the combination of the opti- mization methods.	83
Figure 6.22	Energy consumption of the combination of the optimization methods.	83
Figure 6.23	Execution times of <i>GLAsync</i> and <i>RTLAsync</i>	84
Figure 6.24	Circuit areas of <i>GLAsync</i> and <i>RTLAsync</i>	84
Figure 6.25	Dynamic power consumptions of <i>GLAsync</i> and <i>RTLAsync</i>	85
Figure 6.26	Energy consumptions of <i>GLAsync</i> and <i>RTLAsync</i>	85

Figure 6.27	Example of GL netlists.	86
Figure 6.28	Logic synthesis for the RTL conversion.	86

List of Tables

Table 5.1	Applicability of optimization methods.	50
Table 6.1	RTL conversion results.	71
Table 6.2	Number of inserted D latches.	77

Acknowledgment

I would like to express my greatest appreciation to Senior Associate Professor Hiroshi Saito for guiding me during this research. Additionally, I would like to thank Professor Ben Abdallah Abderazek, Senior Associate Professor Yukihide Kohira, Senior Associate Professor Yoichi Tomioka, and Professor Tomohiro Yoneda for the helpful comments and suggestions. Finally, I would like to thank all members in my laboratory who provided useful comments and suggestions.

Abstract

Most digital integrated circuits are synchronous circuits wherein circuit components are controlled by global clock signals. Recently, the continuous advancements in semiconductor miniaturization technology have allowed more circuit components to be integrated into one chip. However, the increase in the power consumption for clock networks attributed to the distribution of high-frequency clock signals to a wide area will become a significant problem.

In contrast, in asynchronous circuits, circuit components are controlled by local handshake signals or self-timed signals instead of global clock signals. Therefore, asynchronous circuits have potentially low power and low latency compared with synchronous circuits. However, the design of asynchronous circuits is more difficult compared to the design of synchronous circuits. However, there are only a limited number of available electronic design automation (EDA) tools that can support the design of asynchronous circuits.

Design methods based on the design flow used in synchronous circuits with commercial EDA tools have been proposed to make asynchronous circuit designs easy. These methods convert synchronous gate level (GL) netlists into asynchronous GL netlists; such conversion methods are called GL conversion. Although GL conversion simplifies asynchronous circuit design, it can lose an advantage of asynchronous circuits. In asynchronous circuits, each operation can be executed by the required delay. However, the delays of operations in asynchronous GL netlists obtained by GL conversion tend to be equalized because logic synthesis is performed for synchronous register transfer level (RTL) models with clock constraints. In addition, GL conversion methods are not suitable for field programmable gate array (FPGA) designs because the standard design entry for commercial synthesis tools for FPGAs is an RTL model.

For solving these problems, we propose a method for automatic conversion from synchronous RTL models to asynchronous RTL models in this dissertation. We could obtain more optimum asynchronous circuits because we can perform logic synthesis for asynchronous RTL models with appropriate constraints. In addition, the proposed method is suitable for FPGA designs because it generates asynchronous RTL models.

The contributions of this dissertation are as follows: First, the proposed method automatically converts synchronous RTL models to asynchronous RTL ones, which enables the different representation styles of the synchronous RTL models. Second, the implemented conversion tool facilitates asynchronous circuit designs from synchronous RTL models with the generation of non-optimization constraints for logic synthesis and RTL simulation models for FPGA implementation. Third, the proposed method supports both the application specific integrated circuit (ASIC) and FPGA designs.

Chapter 1 (Introduction)

This chapter presents the research background, purpose of this dissertation, and organization of this dissertation.

Chapter 2 (Related Research)

This chapter presents the following related research.

1. Conversion methods from synchronous circuits to asynchronous circuits
2. Behavioral synthesis for asynchronous circuits
3. Synthesis methods for asynchronous circuits from asynchronous RTL models

In this chapter, we describe differences between these methods and our work.

Chapter 3 (Asynchronous Circuits with Bundled-data Implementation)

This chapter presents asynchronous circuits with bundled-data implementation used in this dissertation. In this chapter, we describe the circuit model, behavior, and timing constraints of asynchronous circuits with bundled-data implementation.

Chapter 4 (Conversion Method)

This chapter presents the proposed automatic conversion method from synchronous RTL models to asynchronous RTL models. The proposed method comprises two parts; the first generates an intermediate representation from a given synchronous RTL model; the second generates an asynchronous RTL model from the intermediate representation. The proposed method can address different representation styles of synchronous RTL models because of the use of the intermediate representation. From the generated intermediate representation, the proposed method generates asynchronous RTL models through the generation of data-path circuits and asynchronous control modules. In addition to generating asynchronous RTL models, the proposed method generates an asynchronous RTL simulation model for FPGAs. Further, the proposed method generates non-optimization constraints to prevent optimizations for primitive cells used in a control circuit.

Chapter 5 (Optimization Methods)

This chapter presents four optimization methods during the conversion from synchronous RTL models to asynchronous RTL models. The first one is the modularization for data-path resources to reduce the area of data-path circuits; the second one is the use of appropriate D flip-flops (DFFs) to reduce the area of registers; the third is inserting latches before data-path resources to reduce the dynamic power consumption of data-path circuits; and the fourth is the conversion from DFFs into D latches to reduce the dynamic power consumption of registers.

Chapter 6 (Experimental Results)

In this chapter, we describe the conversion of eight synchronous RTL models into asynchronous RTL models. We demonstrate that conversion time depends on the size and number of states or pipeline stages in the synchronous RTL model. In addition, we verified the functional correctness of converted asynchronous RTL models through logic simulation. Moreover, we performed logic synthesis for converted asynchronous RTL models to evaluate circuit area, execution time, dynamic power consumption, and energy consumption. Asynchronous circuits obtained from the proposed RTL conversion can reduce energy consumption compared with synchronous circuits. Moreover, the combination of optimization methods can reduce more energy consumption in many cases. The proposed RTL conversion can reduce energy consumption compared with GL conversion in many cases. Compared with GL conversion, RTL conversion allows designers to insert operand isolation easily, explore optimum circuits by changing constraints used for logic synthesis from strict to loose, and implement asynchronous circuits on FPGAs easily.

Chapter 7 (Conclusion and Future Work)

This chapter presents our conclusion and future work.

Chapter 1

Introduction

This chapter describes the research background, purpose of this dissertation, and organization of this dissertation in Sections 1.1, 1.2, and 1.3, respectively.

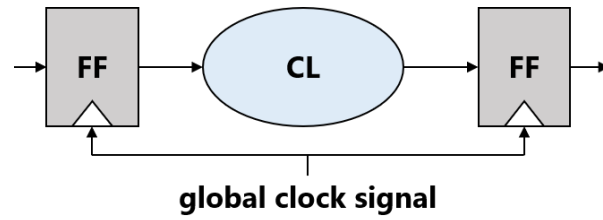
1.1 Research Background

Most digital integrated circuits used in computer systems are synchronous circuits. In synchronous circuits, circuit components are controlled using global clock signals as shown in Fig. 1.1a. For application specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs), synchronous circuits are designed using a design flow with commercial electronic design automation (EDA) tools. However, synchronous circuits encounter problems owing to the continuous advancements in the semiconductor miniaturization technology. The problem is an increase in power consumption attributed to the distribution of high-frequency clock signals to a wide area.

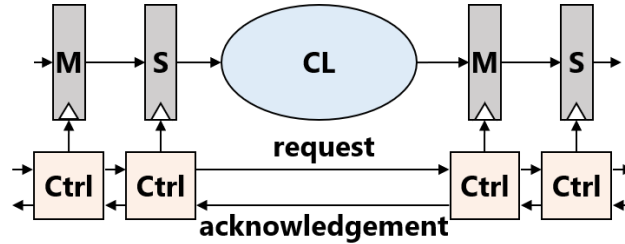
In asynchronous circuits, circuit components are controlled using local handshake signals or self-timed signals instead of global clock signals. Therefore, asynchronous circuits have potentially low-power consumption and low electromagnetic interference compared with synchronous circuits. Further, the performance of asynchronous circuits depends on the average delay because the operation timing of each circuit component in asynchronous circuits is guaranteed by local handshake signals or self-timings. Thus, asynchronous circuits have potentially low latency compared with synchronous circuits wherein the performance depends on the clock cycle time.

However, the design of asynchronous circuits is more difficult than that of synchronous circuits. Based on the applications, designers must select an appropriate data encoding scheme, delay model, and handshake protocol. Design methods and constraints differ based on the selection. In addition, operation timings must be guaranteed by local handshake signals or self-timed signals. Furthermore, circuits without unexpected signal transitions (hazards) that cause a malfunction of circuits are required. Moreover, EDA tools to support the design of asynchronous circuits are insufficient.

Conversion methods from synchronous circuits into asynchronous circuits were proposed in [1–4, 6, 7, 9–12, 14–16] to facilitate the design of asynchronous circuits. These methods convert synchronous GL netlists after logic synthesis into asynchronous GL netlists (desynchronized GL netlists). These methods are based on the design flow used in synchronous circuits with commercial EDA tools. For example, [1–4] converted flip-flops (FFs) in the synchronous GL netlist into master-slave latches. The converted



(a) Synchronous circuits.



(b) Desynchronized circuits.

Figure 1.1: Desynchronization

latches were controlled by inserted latch controllers based on local handshake signals (Fig. 1.1b).

The GL conversion methods suffer from several problems. The GL conversion methods cannot utilize logic optimization considering the characteristics of asynchronous circuits. In asynchronous circuits, operations at each cycle can be executed at their delay using local handshake signals or self-timings. However, operation delays at each cycle in asynchronous GL netlists generated by GL conversion methods are equalized because logic synthesis is performed for synchronous register transfer level (RTL) models with a clock constraint. Further, the optimization of asynchronous circuits is restricted to GL optimizations only because logic synthesis is not performed for asynchronous circuits. Moreover, GL conversion methods are not suitable for FPGA designs because the standard design entry for commercial FPGA design tools is an RTL model.

1.2 Purpose of this Dissertation

In this dissertation, we propose a method for automatic conversion from synchronous RTL models to asynchronous RTL models with bundled-data implementation. Conversion targets are synchronous RTL models designed manually or generated using commercial high-level synthesis (HLS) tools such as Cadence Stratus HLS [38].

The proposed method comprises two parts: the generation of an intermediate representation from a synchronous RTL model and the generation of an asynchronous RTL model with bundled-data implementation from the intermediate representation.

In the first part, the proposed method generates an intermediate representation from a synchronous RTL model. The proposed method can address different representation styles of synchronous RTL models because of the use of the intermediate representation. In this dissertation, we use the eXtensible Markup Language (XML) as the intermediate representation. The proposed method generates a control data flow graph (CDFG) from a synchronous RTL model. Then, the proposed method generates an intermediate

representation by analyzing the CDFG.

In the second part, the proposed method generates an asynchronous RTL model with bundled-data implementation from the intermediate representation. The proposed method assigns data-path resources and asynchronous control modules by referring to the intermediate representation. Thus, the proposed method generates an asynchronous RTL model by connecting control modules to data-path resources. In addition to generating asynchronous RTL models, the proposed method generates an asynchronous RTL simulation model for an FPGA design environment. Further, the proposed method generates a set of non-optimization constraints to prevent the optimization of asynchronous control modules.

The quality of asynchronous circuits from the RTL conversion depends on the representation style of synchronous RTL models. Further, we propose four optimization methods during the proposed RTL conversion from synchronous RTL models to asynchronous RTL models to obtain high-quality asynchronous circuits.

The main contributions of this dissertation are as follows.

1. The proposed method automatically converts synchronous RTL models to asynchronous RTL ones, which enables different representation styles of synchronous RTL models.
2. The implemented conversion tool facilitates asynchronous circuit designs from synchronous RTL models with the generation of non-optimization constraints for logic synthesis and RTL simulation models for FPGA implementation.
3. the proposed method supports both the ASIC and FPGA designs.

1.3 Organization of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 describes the design methods of asynchronous circuits based on the design flow used in synchronous circuits with commercial EDA tools and differences between these methods and this dissertation. Chapter 3 presents the circuit model, behavior, and timing constraints of asynchronous circuits with bundled-data implementation used in this dissertation. Chapter 4 describes the overview of the conversion method, inputs for the conversion, method that generates an intermediate representation from a synchronous RTL model, and method that generates an asynchronous RTL model with bundled-data implementation from the intermediate representation. Chapter 5 describes the proposed optimization methods during the conversion from synchronous RTL models into asynchronous RTL models. Chapter 6 describes conversion results demonstrating that the proposed method can generate asynchronous RTL models from various synchronous RTL models. In addition, Chapter 6 describes the evaluation results of converted asynchronous circuits in terms of circuit area, execution time, dynamic power consumption, and energy consumption. Finally, Chapter 7 describes the conclusion and future work.

Chapter 2

Related Research

Several design methods have been proposed to design asynchronous circuits easily. Even among them, design methods using the design flow for synchronous circuits with commercial EDA tools are related to our study. In this section, we describe differences between these methods and our method.

Cortadella et al. proposed an approach for converting synchronous circuits into asynchronous ones called *Desynchronization* [1, 2]. This approach replaces FFs in synchronous GL netlists synthesized using a commercial logic synthesis tool to pairs of master-slave latches. Subsequently, this approach inserts latch controllers based on handshake signals.

Andrikos et al. proposed a tool that automatically performs *Desynchronization* called *drdesync* [3]. For synchronous GL netlists synthesized by a commercial logic synthesis tool, *drdesync* automatically converts FFs into pairs of master-slave latches. Further, *drdesync* automatically inserts latch controllers and delay elements.

Srinivasan et al. proposed *Desynchronization* for verification [4]. This approach inserts Muller C-elements [5] before asynchronous controllers used in *Desynchronization*. The Muller C-elements are used to guarantee the operation of asynchronous controllers correctly; this approach confirms the equivalence before/after conversion using refinement-based verification.

For large digital systems, Branover et al. proposed an approach to convert synchronous circuits into asynchronous ones [6]. The conversion target is the pipelined synchronous GL netlists generated by a commercial logic synthesis tool; this approach replaces FFs in the synchronous GL netlists to pairs of master-slave latches with corresponding asynchronous controllers. Subsequently, this approach constructs request and acknowledgment networks.

Zhang et al. proposed an approach for generating asynchronous circuits with bundled-data implementation from synchronous RTL models [7]. This approach replaces FFs in synchronous GL netlists generated by a commercial logic synthesis tool into pairs of master-slave latches. Subsequently, this approach inserts asynchronous control modules based on Click elements [8].

Kondratyev et al. proposed a design flow for asynchronous circuits based on null convention logic (NCL) [9, 10]. The NCL is a quasi-delay-insensitive (QDI) asynchronous circuit using dual-rail implementation; they obtained NCL circuits by replacing gates of synchronous GL netlists generated using a commercial logic synthesis tool to NCL gates. In NCL circuits, completion detectors guarantee the write timing of registers.

Reese et al. proposed a design method for asynchronous circuits called *Unified NULL Convention Logic Environment (Uncle)* [11]. *Uncle* accepts synchronous RTL models; subsequently, *Uncle* synthesizes synchronous GL netlists from synchronous RTL models using a commercial logic synthesis tool. For synchronous GL netlists, *Uncle* generates dual-rail asynchronous circuits via dual-rail expanding, generating acknowledgment signals, latch balancing to optimize the performance, and cell merging to optimize the circuit area.

Sartori et al. proposed a synthesis flow for asynchronous circuits based on QDI circuits [12]. This synthesis flow is called *Pulse-F*, which generates NCL circuits by replacing gates in the synchronous GL netlists synthesized by a commercial logic synthesis tool to NCL gates. After generating NCL circuits, *Pulse-F* generates optimized NCL circuits by applying Pulser optimization flow [13] based on clock constraints.

Zhou et al. proposed a compiler for asynchronous circuits based on QDI circuits [14]. This compiler accepts Verilog hardware description language (HDL) of the synchronous GL netlists obtained from the Synopsys Design Compiler. Subsequently, this compiler converts combinational circuits of synchronous GL netlists into dual-rail asynchronous circuits. Moreover, this compiler replaces clock signals to local handshake signals for controlling registers.

Oberg et al. proposed an approach for converting synchronous circuits into asynchronous circuits using commercial logic synthesis tools [15]. This approach requires initially preparing a library for asynchronous circuits initially. For synchronous GL netlists obtained by logic synthesis, this approach replaces logics with gates in the prepared library. Registers are controlled using handshake signals generated by Muller C-elements [5].

Cortadella et al. proposed an approach for synthesizing elastic circuits based on local handshake signals from synchronous circuits [16]. This approach is based on *Desynchronization* and replaces FFs in the synchronous GL netlist to elastic buffers, which comprise data-path circuits based on pairs of master-slave latches and control circuits based on handshake signals. After replacing the elastic buffers, the elastic circuits are generated by inserting a control circuit that inputs a clock signal.

Compared with [1–4, 6, 7, 9–12, 14–16], in which the GL conversion was the objective, we focus on RTL conversion. RTL conversion has advantages over GL conversion. For example, we can generate optimized asynchronous circuits (e.g., performance, area, or power consumption) by performing logic synthesis that assigns appropriate constraints for asynchronous RTL models or optimized asynchronous RTL models. In addition, we can evaluate asynchronous circuits at the RTL. Moreover, we can select not only ASIC but also FPGA as the target because of the RTL conversion.

Wu et al. proposed an approach for converting pipelined synchronous RTL models into pipelined asynchronous RTL models [17]. This approach requires initially specifying pipeline stages in the Verilog HDL of synchronous RTL models. Subsequently, a control module is assigned for each specified pipeline stage. After assigning control modules, this approach replaces a clock signal for registers to a signal of control modules.

However, [17] is not an automatic conversion. Pipeline stages must be manually specified to the Verilog HDL of synchronous RTL models. The method of generating control signals of multiplexers was not described in [17]. In this dissertation, we propose a method for the automatic conversion from synchronous RTL models into asynchronous ones. Moreover, we describe the generation method for the control signals of

multiplexers.

Beerel et al. proposed a design environment for asynchronous circuits called *Proteus* [18]. *Proteus* generates synchronous RTL models from a high-level language based on communicating sequential processes (CSPs). Thereafter, *Proteus* generates the GL netlists of synchronous circuits using commercial logic synthesis tools. For the GL netlists of synchronous circuits, *Proteus* generates the GL netlists of asynchronous circuits based on the approach called *ClockFree*. Further, *ClockFree* performs the optimization (e.g., clustering, fixed fanouts, and slack matching [19]) and the insertion of asynchronous controllers.

Handshake Solutions proposed a design environment called *Tide* [20]. In *Tide*, the behavioral models of applications must be specified using a high-level language called *Haste*. For the *Haste* models, *Tide* synthesizes handshake circuits, and subsequently, *Tide* generates asynchronous GL netlists by mapping cells for the synthesized handshake circuits. For the generated GL netlists, the layout design is synthesized using a commercial layout synthesis tool.

Garcia et al. proposed a synthesis method for asynchronous circuits on FPGAs [21]. For behavioral models specified by VHDL, this method generates asynchronous RTL models by generating a data-path circuit, a two-phase asynchronous controller, and an interface circuit. The asynchronous controller is based on extended burst-mode (XBM) [22] machines.

Curtinhas et al. developed *VHDLASYN* [23], which automatically generates asynchronous RTL models based on the method described in [21]. For behavioral models specified by VHDL, *VHDLASYN* generates asynchronous RTL models while minimizing the latency or number of resources by scheduling operations and allocating resources.

Sacker et al. proposed a synthesis system for synthesizing asynchronous circuits [24]. This synthesis system is called *MOODs*. For behavioral models specified by VHDL, *MOODs* generates asynchronous RTL models through operation scheduling, resource allocation, and control synthesis. A data-path circuit comprises chained, parallel, and sharing units. In the control synthesis, *MOODs* synthesizes two-phase asynchronous controllers.

Josipovic et al. proposed a high-level synthesis of elastic circuits [16] from the C language [25]; this method generates a data-flow graph (DFG) and control-flow graph (CFG) from the C language. Thereafter, it assigns and connects elastic components by referring to DFG and CFG.

In [18, 20, 21, 23–25], the objective was not conversion from synchronous circuits to asynchronous circuits. These methods directly generate asynchronous circuits from behavioral models through the operation scheduling of asynchronous circuits and synthesis of asynchronous controllers. In this dissertation, we focus on the conversion from synchronous circuits to asynchronous circuits.

Yoshimi et al. proposed a design method for asynchronous circuits with bundled-data implementation [26]. This method is based on a tool [27] that supports the design of asynchronous circuits with bundled-data implementation. Logic synthesis, layout synthesis, and static timing analysis are performed using commercial EDA tools for asynchronous RTL models based on Q modules [28]. Further, the design method supports a timing constraint generation, timing verification, and delay adjustment, which are not supported by commercial EDA tools with their own tools.

Gibiluka et al. proposed a synthesis flow for asynchronous circuits with bundled-

data implementation [29]. Relative timing constraints (RTCs) [30] for asynchronous RTL models are generated in the synthesis flow. For asynchronous RTL models with the RTC, logic synthesis and layout synthesis are performed using commercial EDA tools.

The researchers in [26, 27, 29] focused on the design method from asynchronous RTL models to layout designs. The generation method for asynchronous RTL models was not described in [26, 27, 29]. In this dissertation, we focus on the generation of asynchronous RTL models.

On one hand, Chapter 4 of this dissertation is based on [31], wherein we proposed a method for the automatic conversion from synchronous RTL models into asynchronous RTL models. However, the target is only non-pipelined circuits. Asynchronous control modules are assigned directly by referring to the states of the finite state machine (FSM) in non-pipelined synchronous RTL models. Further, we propose a conversion method from pipelined synchronous RTL models into asynchronous RTL models. The proposed method generates a CDFG from pipelined synchronous RTL models before conversion, which was not generated in [31], to assign pipelined asynchronous control modules

On the other hand, Chapter 5 is based on [32], in which we proposed optimization methods during RTL conversion [31] to obtain the high-quality of asynchronous circuits. Further, we propose a conversion from DFFs to D latches to optimize the area of registers, which was not described in [32].

Chapter 3

Asynchronous Circuits with Bundled-data Implementation

In this chapter, we describe the asynchronous circuits with bundled-data implementation used in this study. Section 3.1 describes the circuit model and behavior of asynchronous circuits with bundled-data implementation. Section 3.2 describes the timing constraints for the circuit model.

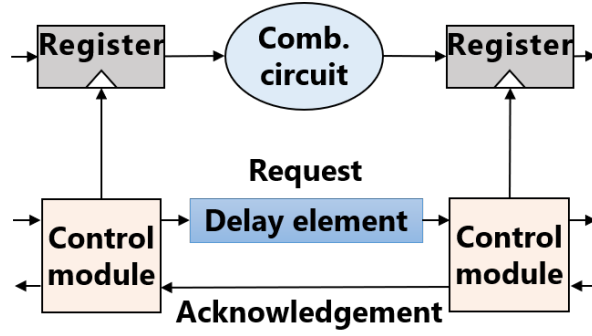
Bundled-data implementation is a data encoding scheme in asynchronous circuits. In the bundled-data implementation, a one-bit signal is represented by one signal. Therefore, the same data-path circuit used in synchronous circuits can be used. The time required to write the data to registers is guaranteed by delay elements on the request signals in a control circuit. Hence, the performance of the bundled-data implementation depends on the delay of the control circuit which includes the delay elements.

Figure 3.1 shows asynchronous circuits with bundled-data implementation. Figure 3.1a shows the bundled-data implementation based on handshake signals. In this bundled-data implementation, a request signal and an acknowledgment signal are used to guarantee the operation. Thus, the performance of the bundled-data implementation depends on the delays of the request and acknowledgment signals. Figure 3.1b shows the bundled-data implementation based on self-timing. In this bundled-data implementation, only the request signal is used to guarantee the operation. Therefore, the performance of the bundled-data implementation depends on the delay of the request signal. In this dissertation, we use the bundled-data implementation based on self-timing because the performance can be improved compared with the bundled-data implementation based on handshake signals.

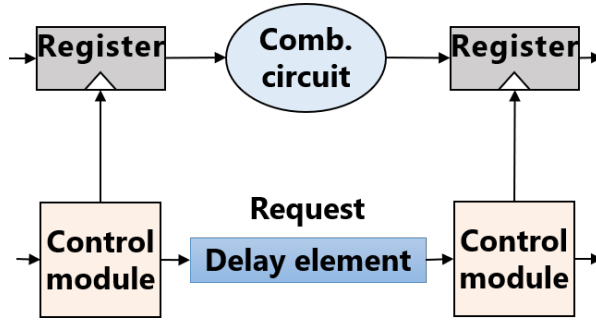
3.1 Circuit Model

Figure 3.2 shows the circuit models of asynchronous circuits with bundled-data implementation used in this study. The circuit model comprises a data-path circuit and control circuit. We assume that target circuit models are both non-pipelined and pipelined circuits. Figure 3.2a shows non-pipelined asynchronous circuits and Figure 3.2b shows pipelined asynchronous circuits.

The data-path circuit is almost the same as the one used in synchronous circuits. It comprises registers reg_k , multiplexers mux_l , and functional units fu_h . h , l , and k represent the identifier of registers, multiplexers, and functional units, respectively. If



(a) Bundled-data implementation based on handshake signals.



(b) Bundled-data implementation based on self-timing.

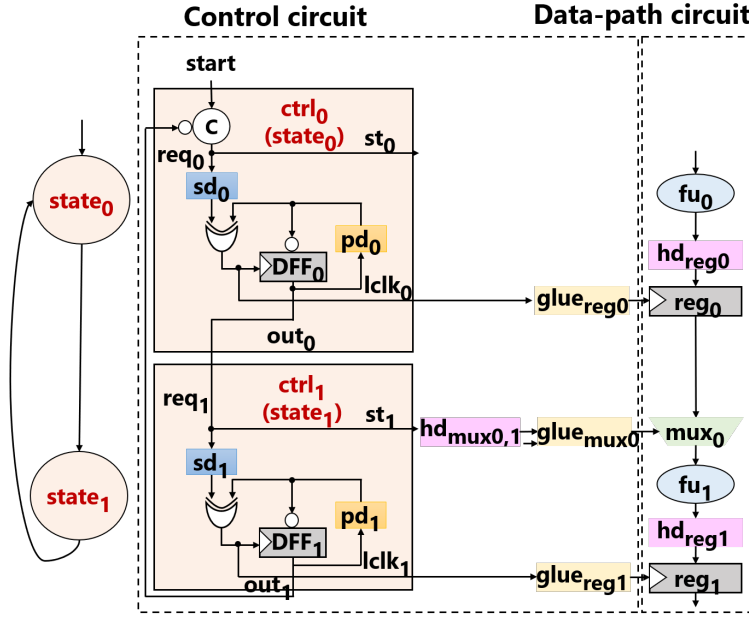
Figure 3.1: Asynchronous circuits with bundled-data implementation.

hold violations occur on reg_k , a delay element hd_{reg_k} is inserted on the input signal of reg_k .

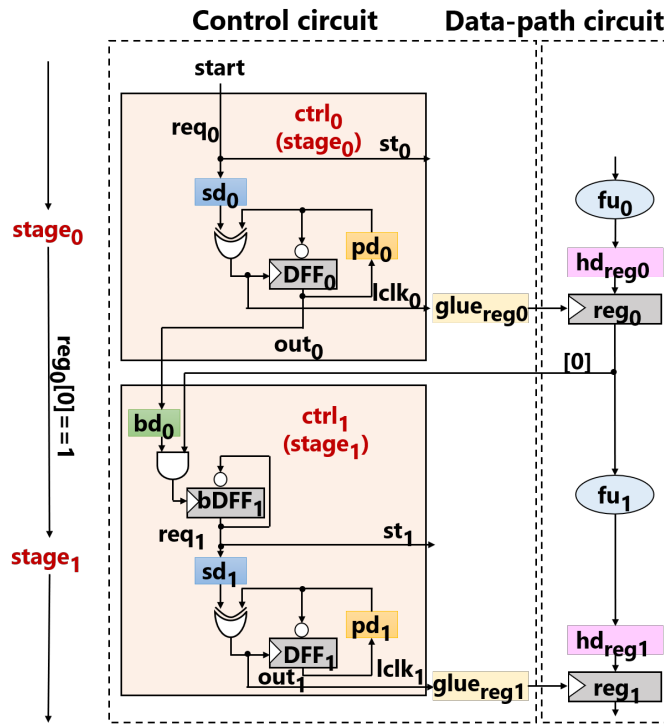
The control circuit is based on an FSM or pipeline stages in the synchronous circuits. For non-pipelined circuits, the control circuit comprises control modules $ctrl_i$ ($0 \leq i \leq n - 1$) assigned for each state $state_i$. For pipelined circuits, the control circuit comprises control modules $ctrl_i$ ($0 \leq i \leq n - 1$) assigned for each pipeline stage $stage_i$. i represents a state or a pipeline stage. The glue logics $glue_{reg_k}$ and $glue_{mux_l}$ represent logics used to generate write signals for reg_k and control signals for mux_l , respectively. If hold violations occur on reg_k through 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 .

The control module $ctrl_i$ is obtained by modifying Click elements [8]; they comprise a D flip-flop DFF_i , an XOR gate, and a delay element sd_i . A D flip-flop $bDFF_i$ and an AND gate are inserted before sd_i when there are control branches. An XOR gate is inserted before sd_i when $ctrl_i$ requires several request signals. Figure 3.3a shows the structure of $ctrl_i$ based on Click elements [8]. The acknowledgment signal ack_i and request signal req_i are used in $ctrl_i$ based on Click elements. Figure 3.3b shows the structure of $ctrl_i$ used in this study. In $ctrl_i$ used in this dissertation, ack_i used in traditional asynchronous circuits is not used. 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 the setup constraints for reg_k . Further, $ctrl_i$ is operated by the rising and falling transitions of req_i . The data are written to reg_k by the rising transition of $lclk_i$.

The control circuit begins its operation when the rising transition of the input signal $start$ arrives at the control circuit. $ctrl_i$ begins its operation when the rising transition



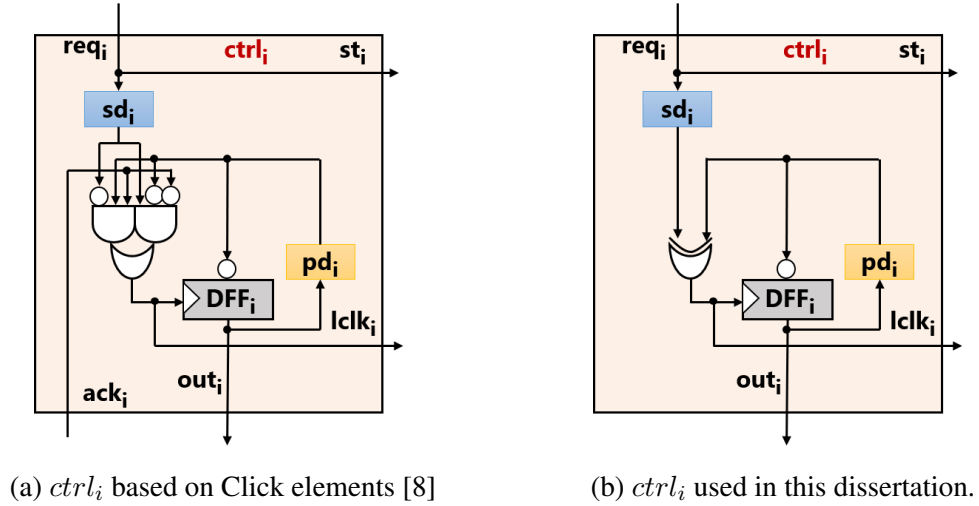
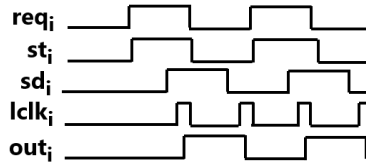
(a) Non-pipelined circuit model.



(b) Pipelined circuit model.

Figure 3.2: Circuit models of asynchronous circuits with bundled-data implementation used in this dissertation.

of out_{i-1} or $lclk_{i-1}$ from $ctrl_{i-1}$ arrives at $ctrl_i$. Figure 3.4 shows the timing diagram of $ctrl_i$; the signal transition of out_{i-1} or $lclk_{i-1}$ generates the rising transition of req_i . Subsequently, req_i generates the rising transition of st_i , which controls mux_i through $glue_{mux_i}$. In addition, req_i generates the rising transition of $lclk_i$ through sd_i and the XOR gate. Further, $lclk_i$ controls reg_k through $glue_{reg_k}$, and $lclk_i$ controls DFF_i ,

Figure 3.3: Control module $ctrl_i$.Figure 3.4: Timing diagram of $ctrl_i$ used in this dissertation.

which generates the rising transition of out_i to pass the control to $ctrl_{i+1}$. Finally, $ctrl_i$ generates the falling transition of $lclk_i$ using out_i . The behavior of $ctrl_i$ for the falling transition of req_i is the same as that of the rising transition of req_i .

3.2 Timing Constraints

In the asynchronous circuits with bundled-data implementation used in this dissertation, it is necessary to satisfy the setup, hold, branch, and pulse-width constraints to operate the circuit correctly. In this section, we describe the timing constraints.

3.2.1 Setup Constraints

The input data for reg_k must be stable before the setup time to write the input data to reg_k ; this is called the setup constraint for reg_k . Figure 3.5 shows the data-path $sdp_{i,p}$ and control-path $scp_{i,p}$ related to the setup constraint; p represents the identifier of paths. There are two data-paths $sdp_{i,p}$ (red line): a path from the output of $lclk_{i-1}$ to the destination register reg_1 through the source register reg_0 (Fig. 3.5a), and a path from the output of $lclk_{i-1}$ to the destination register reg_1 through the glue logic $glue_{mux_0}$ (Fig. 3.5b). Further, $scp_{i,p}$ (blue line) represents a path from the output of $lclk_{i-1}$ to the destination register reg_1 through the delay element sd_i . We define the maximum delay of $sdp_{i,p}$ as $t_{maxsdp_{i,p}}$, minimum delay of $scp_{i,p}$ as $t_{minscp_{i,p}}$, margin for $t_{maxsdp_{i,p}}$ as $t_{sdpm_{i,p}}$, and setup time of the destination register as $t_{setup_{i,p}}$. Thus, the setup constraint

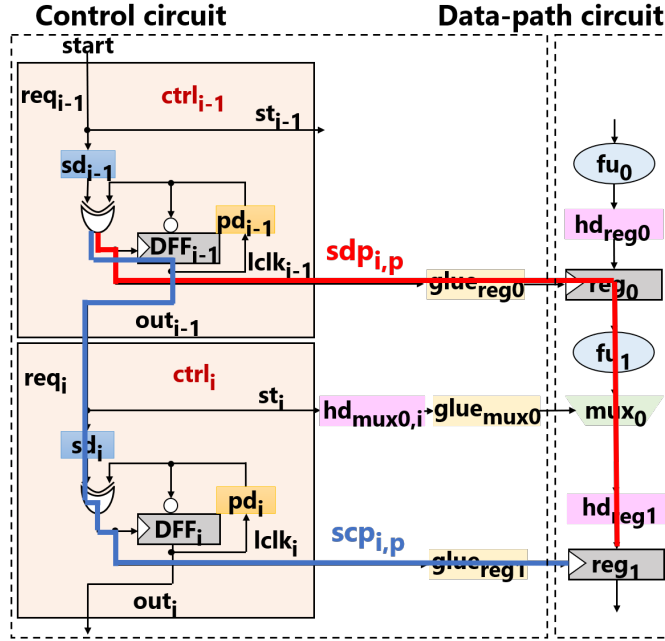
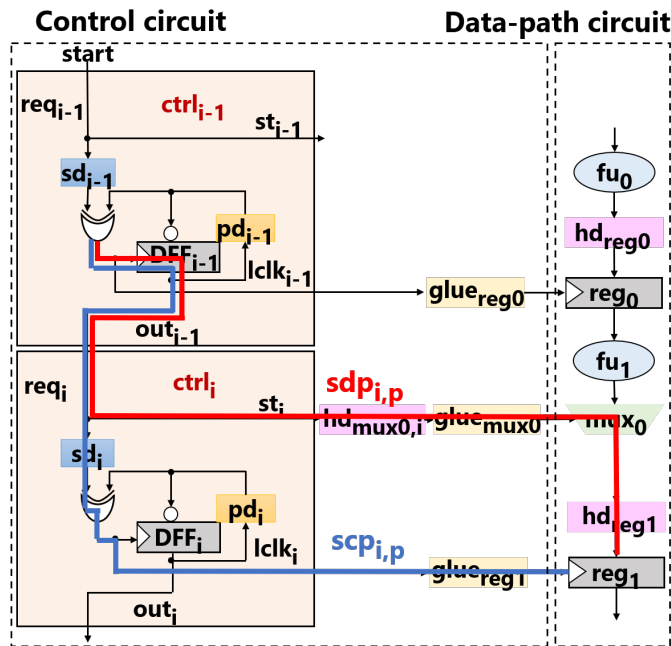

 (a) Data-path $sdp_{i,p}$ through a source register and control-path $scp_{i,p}$.

 (b) Data-path $sdp_{i,p}$ through $glue_{mux1}$ and control-path $scp_{i,p}$.

Figure 3.5: Paths related to setup constraints.

can be represented by

$$t_{min scp_{i,p}} > t_{max sdp_{i,p}} + t_{sdpm_{i,p}} + t_{setup_{i,p}} \quad (3.1)$$

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

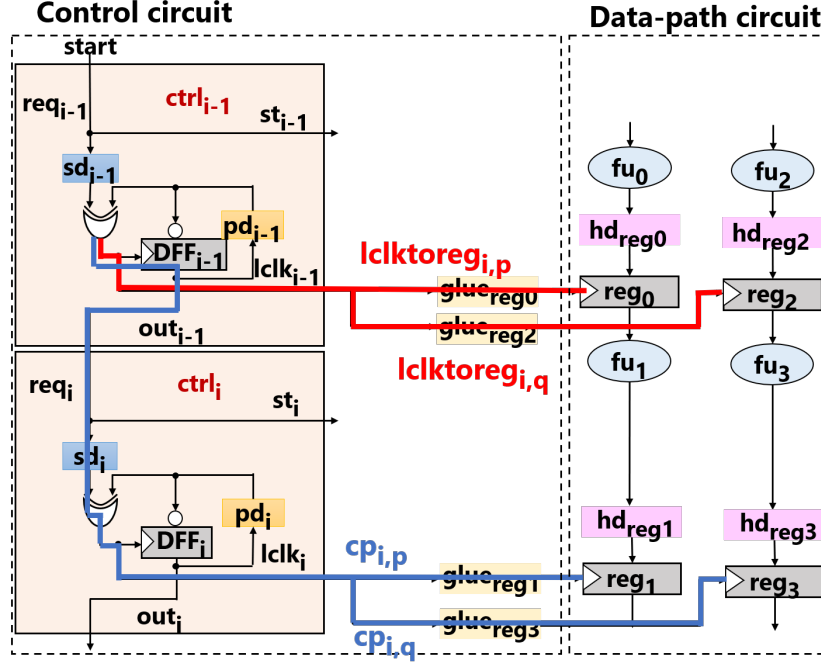


Figure 3.6: Paths related to local cycle time.

3.2.2 Hold Constraints

The data should be stable for the hold time after input data are written to reg_k ; this is called the hold constraint for reg_k . The hold constraints used in this dissertation differ between non-pipelined and pipelined circuits.

We define a local cycle time (lct_i) and a global cycle time (gct) before the explanation for the hold constraints. Figure 3.6 shows the paths related to lct_i , which represents a maximum delay for the operating $stage_i$. Further, gct represents the cycle time in asynchronous circuits.

lct_i and gct can be represented by

$$lct_i = \max(t_{maxcp_{i,p}} - t_{maxlclktoreg_{i,p}}, \dots, t_{maxcp_{i,q}} - t_{maxlclktoreg_{i,q}}) \quad (3.2)$$

$$gct = \max(lst_0, \dots, lst_{n-1}) \quad (3.3)$$

Further, $t_{maxcp_{i,p}}$ 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, and lct_i represents the largest value of $t_{maxcp_{i,p}}$ minus $t_{maxlclktoreg_{i,p}}$ in $stage_i$. Finally, gct represents the maximum value of lct_i .

Figure 3.7 shows the data-path $hdp_{i,p}$ and control-path $hcp_{i,p}$ related to the hold constraint for non-pipelined circuits. There are two data-paths $hdp_{i,p}$ (red line): a path from the output of $lclk_i$ to the destination register reg_1 through the source register reg_1 (Fig. 3.7a), and a path from the output of $lclk_i$ to the destination register reg_1 through the glue logic $glue_{mux_0}$ (Fig. 3.7b). Further, $hcp_{i,p}$ (blue line) represents a path from the output of $lclk_i$ to the destination register reg_1 . We define the minimum delay of $hdp_{i,p}$ as $t_{minhdp_{i,p}}$, maximum delay of $hcp_{i,p}$ as $t_{maxhcp_{i,p}}$, margin for $t_{maxhcp_{i,p}}$ as $t_{hcpm_{i,p}}$, and hold time of the destination register as $t_{hold_{i,p}}$. Thus, the hold constraint can be

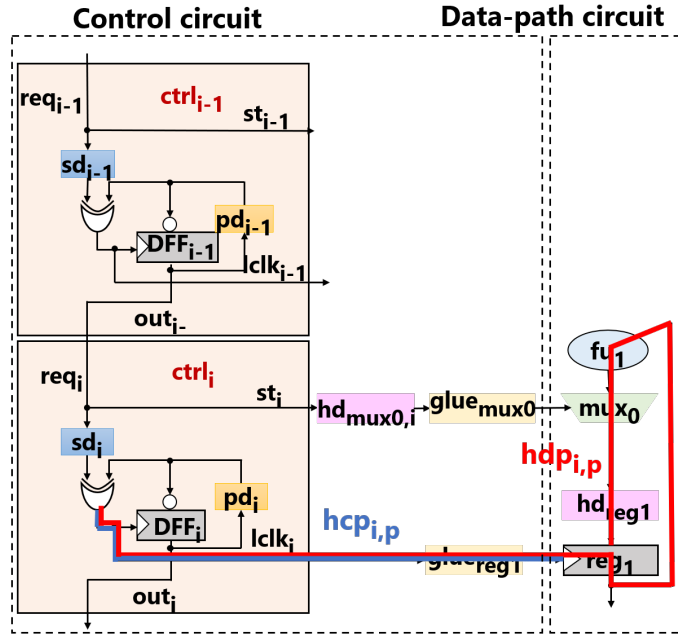
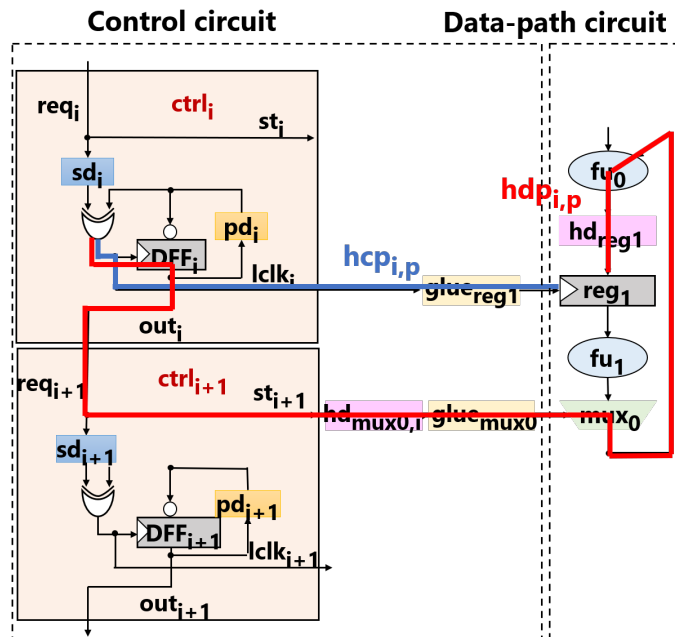

 (a) Data-path $hdp_{i,p}$ through a source register and control-path $hcp_{i,p}$.

 (b) Data-path $hdp_{i,p}$ through $glue_{mux1}$ and control-path $hcp_{i,p}$.

Figure 3.7: Paths related to hold constraints for non-pipelined circuits.

represented by

$$t_{minhdp_{i,p}} > t_{maxhcp_{i,p}} + t_{hcp_{i,p}} + t_{hold_{i,p}} \quad (3.4)$$

If the hold constraint is violated, we must adjust the number of cells for $hdreg_k$ or $hdmux_{i,l}$.

Figure 3.8 shows the data-path $hdp_{i,p}$ and control-path $hcp_{i,p}$ related to the hold constraint for pipelined circuits. There are two data-paths $hdp_{i,p}$ (red line): a path from the input signal $start$ to the destination register reg_1 through the source register

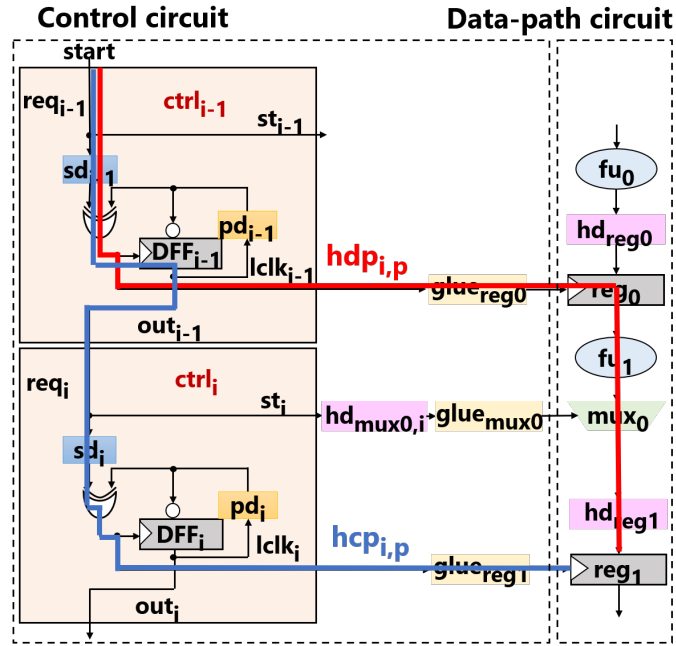
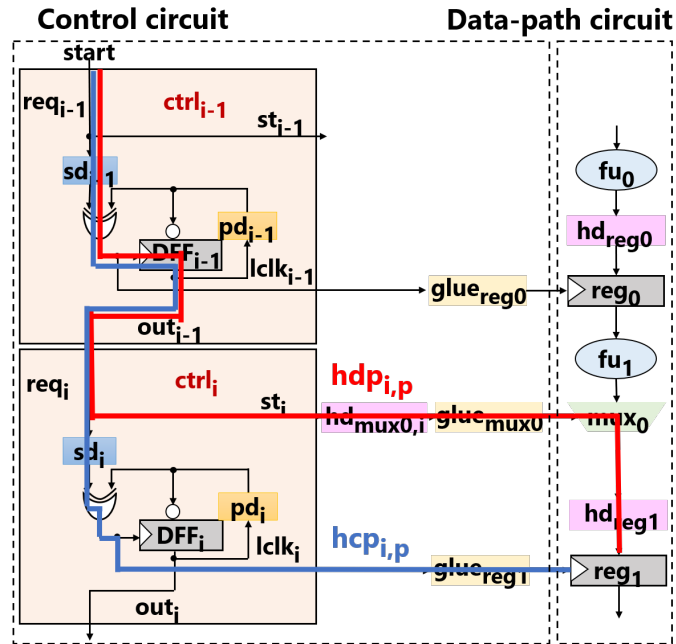

 (a) Data-path $hdp_{i,p}$ through a source register and control-path $hcp_{i,p}$.

 (b) Data-path $hdp_{i,p}$ through $glue_{mux_1}$ and control-path $hcp_{i,p}$.

Figure 3.8: Paths related to hold constraints for pipelined circuits.

reg_1 (Fig. 3.8a), and a path from the input signal $start$ to the destination register reg_1 through the glue logic $glue_{mux_0}$ (Fig. 3.8b). Further, $hcp_{i,p}$ (blue line) represents a path from the input signal $start$ to the destination register reg_1 . We define the minimum delay of $hdp_{i,p}$ as $t_{minhdp_{i,p}}$, maximum delay of $hcp_{i,p}$ as $t_{maxhcp_{i,p}}$, margin for $t_{maxhcp_{i,p}}$ as $t_{hcp_{i,p}}$, hold time of the destination register as $t_{hold_{i,p}}$, and input interval as II . Thus,

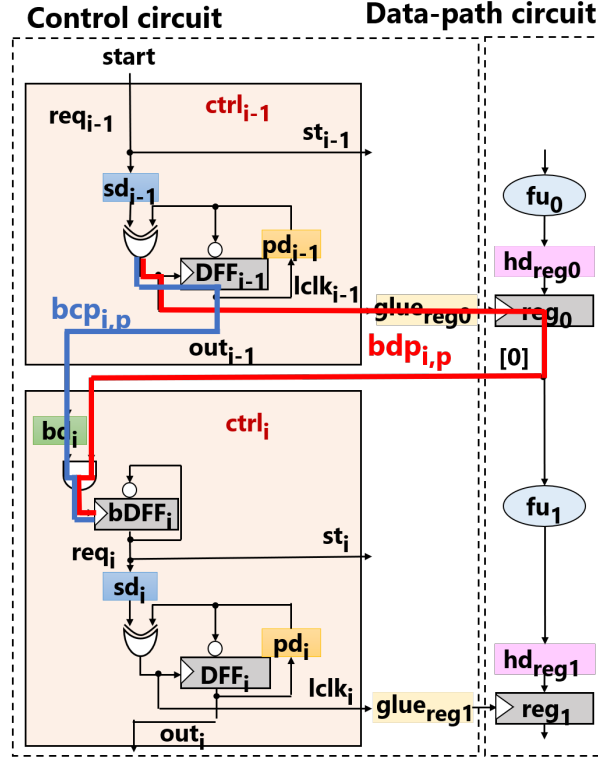


Figure 3.9: Paths related to branch constraint.

the hold constraint can be represented by

$$t_{minhd_{i,p}} + gct \times II > t_{maxh_{i,p}} + t_{h_{i,p}} + t_{hold_{i,p}} \quad (3.5)$$

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

3.2.3 Branch Constraints

The conditional signal for the branch must be arrived at a branch evaluation logic in $ctrl_i$ before the arrival of the control signal from the previous control module if a control branch occurs; this is called the branch constraint for $ctrl_i$. Figure 3.9 shows the data-path $bdp_{i,p}$ and control-path $bcp_{i,p}$ related to the branch constraint. $bdp_{i,p}$ (red line) represents a path from the output of $lclk_{i-1}$ to the clock pin of $bDFF_i$ through the destination register reg_1 . Further, $bcp_{i,p}$ (blue line) represents a path from the output of $lclk_{i-1}$ to the clock pin of $bDFF_i$ through DFF_{i-1} . We define the maximum delay of $bdp_{i,p}$ as $t_{maxbd_{i,p}}$, minimum delay of $bcp_{i,p}$ as $t_{minbcp_{i,p}}$, and margin for $t_{maxbd_{i,p}}$ as $t_{bdpm_{i,p}}$. Thus, the branch constraint can be represented by

$$t_{minbcp_{i,p}} > t_{maxbd_{i,p}} + t_{bdpm_{i,p}} \quad (3.6)$$

We must adjust the number of cells for bd_i if the branch constraint is violated.

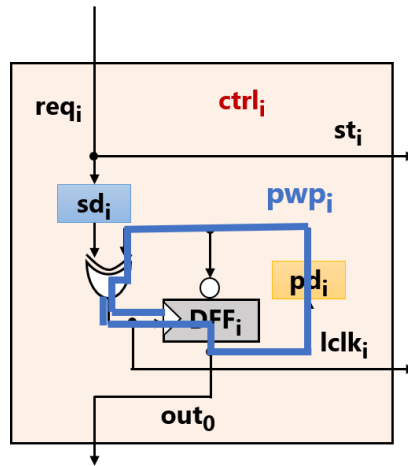


Figure 3.10: Path related to pulse-width constraint.

3.2.4 Pulse-Width Constrains

The delay from the rising transition of $lclk_i$ to the falling transition of $lclk_i$ must be larger than the pulse-width of the target register; this is called the pulse-width constraint for the target register. Figure 3.9 shows the path related to the pulse-width constraint. A pulse-width path pwp_i (blue line) represents a path from the output of $lclk_i$ to the clock pin of DFF_i through the input of $lclk_i$. We define the minimum delay of pwp_i as t_{minpwp_i} and pulse-width of DFF_i as t_{pluse} . Therefore, the pulse-width constraint can be represented by

$$t_{minpwp_i} > t_{pluse} \quad (3.7)$$

We must adjust the number of cells for pd_i if the pulse-width constraint is violated.

Chapter 4

Conversion Method

In this chapter, we describe the proposed method for the automatic conversion from synchronous RTL models to asynchronous RTL models with bundled-data implementation. Section 4.1 describes the overview of the proposed method. Section 4.2 describes the inputs of the proposed method. Section 4.3 describes the method that generates an intermediate representation represented by XML from synchronous RTL models. Finally, Section 4.4 describes the method that generates asynchronous RTL models from the intermediate representation.

4.1 Overview

Figure 4.1 shows the flow of the proposed method. The proposed method comprises two parts: the first part generates an XML as the intermediate representation from given synchronous RTL models (*Sync2XML*), and the second part generates asynchronous RTL models from the XML (*XML2Async*).

The proposed method represents synchronous RTL models in XML, which a markup languages used to represent structures. In XML, we can freely define the data structure of each element using tags. The proposed method can manage different representation styles of synchronous RTL models because they are represented using XML. Further, we call the XML that represents synchronous RTL models as the Model-XML.

The proposed method accepts synchronous RTL models as input; the proposed method assumes that synchronous RTL models are described by Verilog HDL. The conversion targets are synchronous RTL models designed manually as described in Section 4.2.1 or generated using commercial HLS tools such as Cadence Stratus HLS.

Moreover, we represent all information required for conversion in a different XML file, which includes the circuit model, implementation target, etc. We call this XML as Info-XML and describe the detail of Info-XML in Section 4.2.2.

The proposed method generates an abstract syntax tree (AST) and a control flow from given synchronous RTL models using *Pyverilog* [33]. The AST represents the structure of RTL models, whereas the control flow represents the state transitions of RTL models.

The proposed method generates a Model-XML from the AST and control flow after generating the AST and control flow. The proposed method generates a CDFG from the AST and control flow. For the CDFG, the proposed method generates a Model-XML by analyzing the states of an FSM or pipeline stages. The Model-XML comprises the data-path resource information, path information including data-paths and control-paths, and

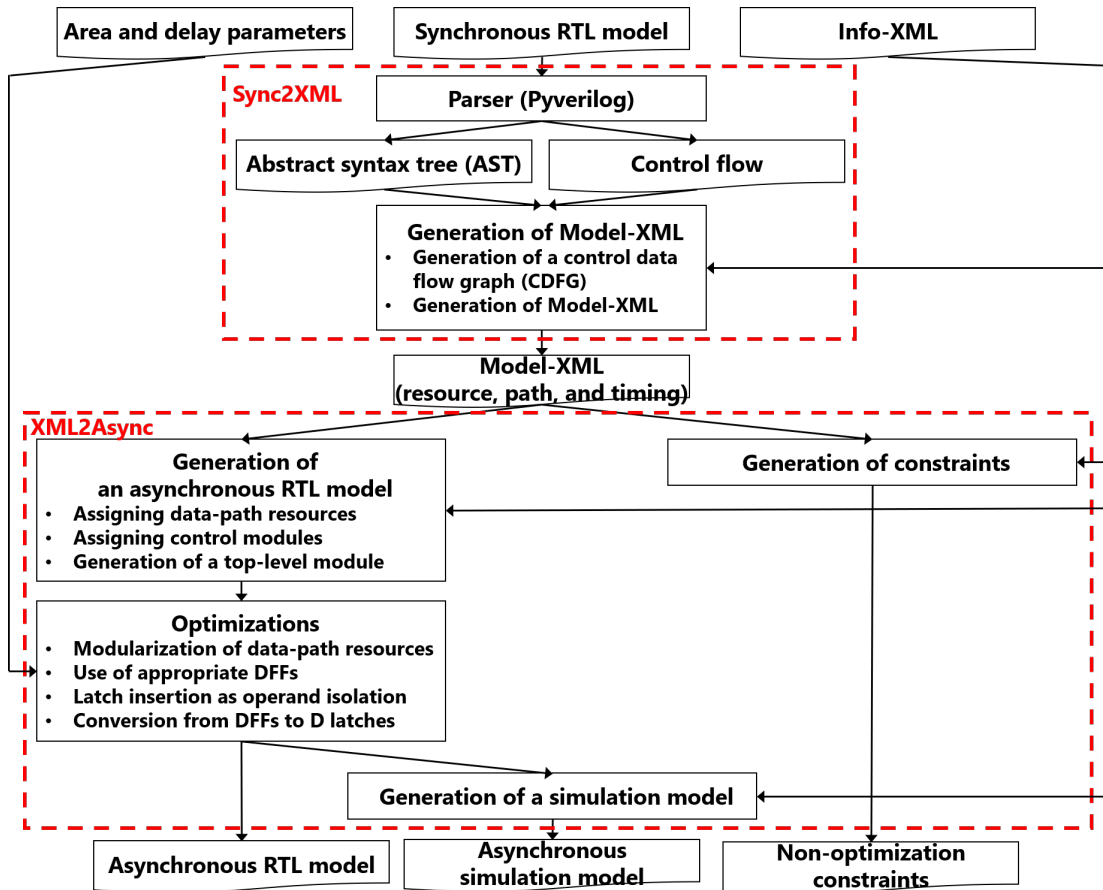


Figure 4.1: Flow of proposed RTL conversion method.

timing information including register write and multiplexer control signals.

The proposed method generates an asynchronous RTL model with bundled-data implementation from the Model-XML after generating the Model-XML. The proposed method assigns and connects data-path resources by referring to the resource and path information. Further, the control modules are assigned and connected by referring to the path information. The proposed method connects the data-path resources to control modules by referring to the path information to generate a top-level module. Finally, the proposed method connects the control modules to data-path resources by referring to the timing information.

The proposed method generates asynchronous RTL simulation models of bundled-data implementation and a set of non-optimization constraints to support FPGA designs and logic synthesis. Primitive cells are used to prevent hazardous behaviors in control modules and preserve the correct timing required for bundled-data implementation. In FPGA implementations, designers cannot perform RTL simulations using asynchronous RTL models with primitive cells; therefore, the proposed method generates asynchronous RTL simulation models without primitive cells when the target is an FPGA. In addition to generating asynchronous RTL models, the proposed method generates non-optimization constraints for control modules and delay elements to prevent the optimization of primitive cells during the logic synthesis or layout synthesis.

Moreover, the proposed method performs four optimization methods to obtain the high quality of asynchronous RTL models. The proposed method takes other input files

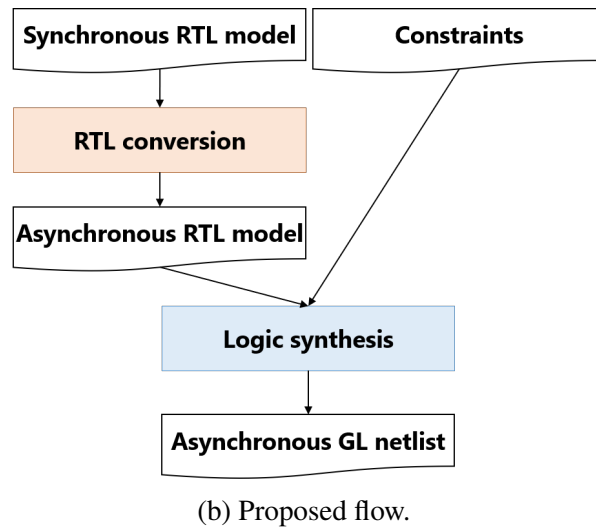
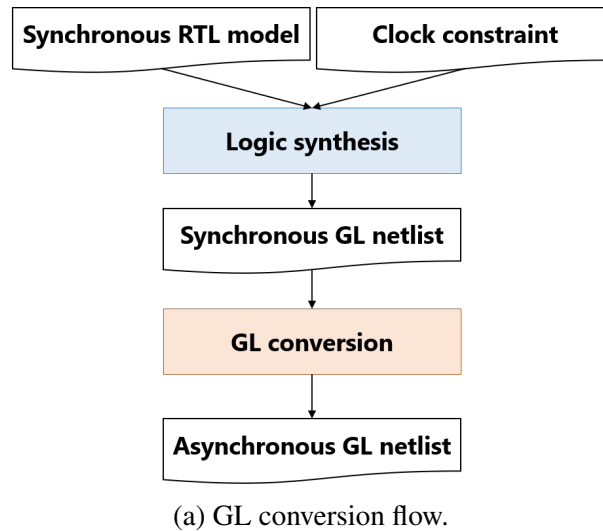


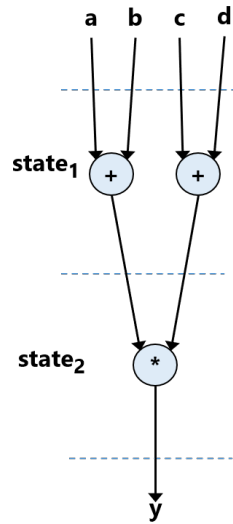
Figure 4.2: Conversion flow.

such as area and delay parameters to perform the optimization methods. We describe the detail of each optimization method and parameter in Chapter 5.

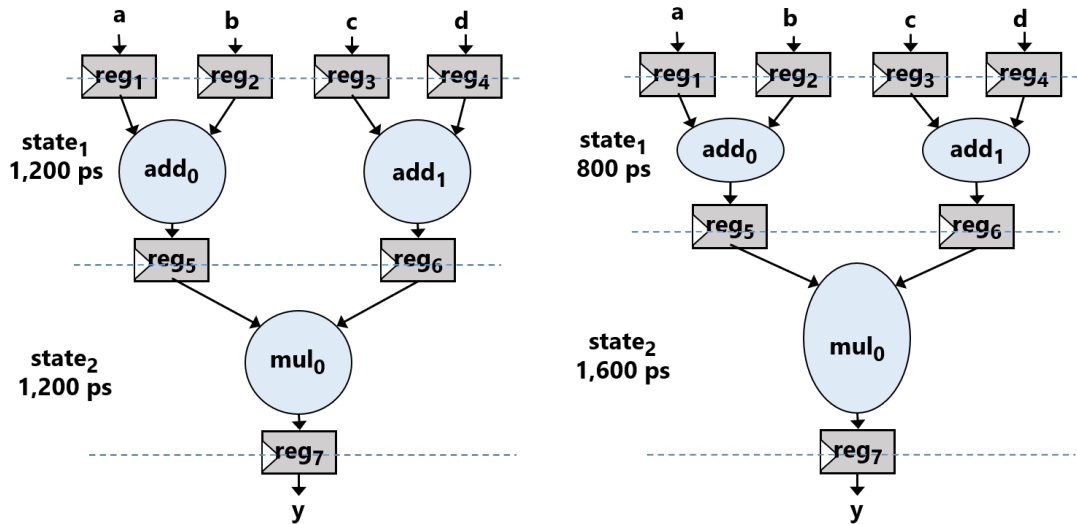
Compared with the GL conversion methods [1–4, 6, 7, 9–12, 14–16], the proposed method has the following advantages:

1. We can evaluate asynchronous circuits at RTL.
2. The proposed method is suitable for FPGA designs.
3. We can generate optimized asynchronous circuits by performing logic synthesis for asynchronous RTL models with appropriate constraints.
4. We can obtain optimized asynchronous RTL models by applying optimization methods during the RTL conversion.

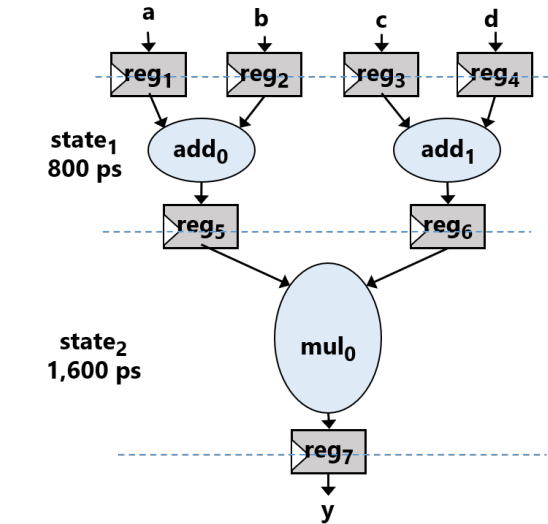
The proposed method can evaluate asynchronous circuits at the RTL. For example, GL conversion methods generate asynchronous GL netlists from synchronous GL netlists (Fig. 4.2a). However, functional verification cannot be performed immediately



(a) Data flow.



(b) Data-path circuit obtained using GL conversion.



(c) Data-path circuit obtained using RTL conversion.

Figure 4.3: Comparison of conversions.

because a delay adjustment for satisfying the timing constraints of the generated asynchronous GL netlists is not performed. However, in the proposed method, functional verification can be performed easily using the RTL simulation for asynchronous RTL models (Fig. 4.2b).

The proposed method generates asynchronous RTL models, and therefore, it is suitable for FPGA designs. The standard design entry for commercial FPGA design tools is an RTL model; however, the GL conversion methods are not suitable for FPGA designs because they generate asynchronous GL netlists from synchronous ones.

In the RTL conversion, we can generate optimized asynchronous circuits by performing logic synthesis for asynchronous RTL models with appropriate constraints. We describe this advantage more clearly in Fig. 4.3. Figure 4.3a depicts data flow in a data-path circuit, where nodes represent operations and edges represent data dependency. In GL conversion, the delay of each cycle in the synthesized data-path circuit is equalized

(1, 200 ps in $state_1$ and $state_2$) because logic synthesis is performed for synchronous RTL models with a clock constraint (Fig. 4.3b). In contrast, in the RTL conversion, we perform logic synthesis with different delay constraints for each cycle to fully utilize asynchronous circuits (Fig. 4.3c). For example, we assign a loose delay constraint for operations (e.g., 1, 600 ps for mul_0 in $state_2$) whose resources consume more power, whereas we assign a strict delay constraint for operations (e.g., 800 ps for add_0 and add_1 in $state_1$) whose resources consume less power under a latency constraint. This can result in a more optimum data-path circuit than the data-path circuit used in GL conversion.

Further, in the RTL conversion, we can obtain optimized asynchronous RTL models by applying optimization methods during the RTL conversion. For example, in the asynchronous GL netlists obtained from GL conversion, wire names were changed and resources were replaced with gates, which leads to the difficulty performing optimization such as the operand isolation. In contrast, in the RTL conversion, the optimization can be applied to resources in asynchronous RTL models. Therefore, optimized asynchronous RTL models can be obtained easily.

4.2 Inputs

4.2.1 Target Synchronous RTL Models

The proposed method assumes that the target synchronous RTL models are designed manually or generated using commercial HLS tools. Target synchronous RTL models are both non-pipelined and pipelined synchronous RTL models.

The proposed method assumes that the data-path circuit of target synchronous RTL models is composed of functional units, registers, and multiplexers as described in Chapter 3. Macros such as memories are considered functional units or registers; further, the proposed method assumes that DFFs are used for registers.

The proposed method assumes that target synchronous RTL models have only one control circuit. The proposed method assumes that the control circuit is represented by an FSM or registers to control the pipeline stages. Control circuits must be modified such that they are unified in one FSM if the control circuit has several FSMs.

The proposed method assumes that the target synchronous RTL models are specified by Verilog HDL. Syntaxes such as "function," "task," "for," "while," "wait," and "[sub,5'h0+:32] (concatenation)" must not be included in Verilog HDL. We will address syntax in our future research.

Further, the proposed method does not concern the following: whether clock gating for registers is performed and the number of cycles for input intervals in the pipelined circuits.

Figure 4.4 shows an example of the target non-pipelined synchronous RTL model. Figure 4.4a-d show the top-level module using Verilog HDL, register using Verilog HDL, control circuit using Verilog HDL, and RTL structure, respectively. The control circuit comprises an FSM.

Figure 4.5 shows an example of the target pipelined synchronous RTL model. Figure 4.5a-c show the top-level module using Verilog HDL, control circuit using Verilog HDL, and RTL structure. The structure of RTL models may change depending on whether RTL models include pipeline stalls. Pipeline stages stall the operation during

```

module sample(clock, reset, start, in0, in1, cond, out0);
  input clock, reset, start, cond;
  input [31:0] in0, in1;
  output [31:0] out0;
  wire [31:0] reg0_out, reg1_out, reg2_out, reg3_out,
    reg4_out, mux0_out, add0_out, mul0_out;
  wire en0, en1, en2, en3, en4, sm0;
  dff32e reg0(.in(cond), .reset(reset), .clock(clock),
    .en(en0), .out(reg0_out));
  dff32e reg1(.in(in0), .reset(reset), .clock(clock),
    .en(en1), .out(reg1_out));
  dff32e reg2(.in(in1), .reset(reset), .clock(clock),
    .en(en2), .out(reg2_out));
  dff32e reg3(.in(add0_out), .reset(reset), .clock(clock),
    .en(en3), .out(reg3_out));
  dff32e reg4(.in(mul0_out), .reset(reset), .clock(clock),
    .en(en4), .out(reg4_out));
  mux2 mux0(.in0(reg1_out), .in1(reg2_out),
    .ctrl(sm0), .out(mux0_out));
  add32 add0(.a(reg1_out), .b(reg2_out),
    .out(add0_out));
  mul32 mul0(.a(mux0_out), .b(reg3_out),
    .out(mul0_out));
  assign out0 = reg4_out;
  ctrl ctrl(.clock(clock), .reset(reset), .start(start),
    .cond(reg0_out[0]), .sm0(sm0), .en0(en0), .en1(en1),
    .en2(en2), .en3(en3), .en4(en4));
endmodule
    
```

(a) Verilog HDL of the top-level module.

```

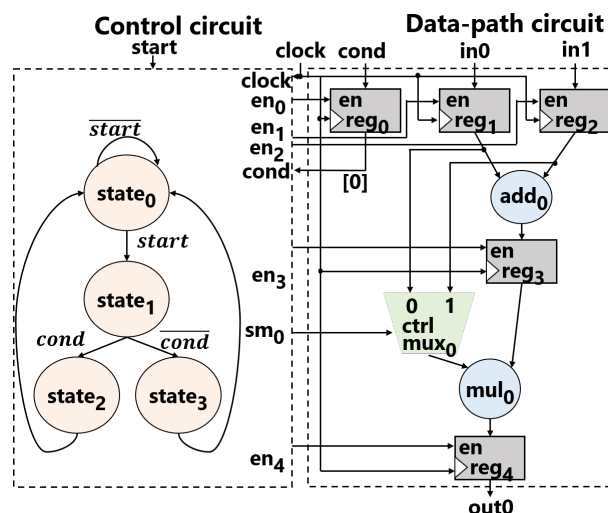
module dff32e (in, reset, clock, en, out);
  input [31:0] in;
  input reset, clock, en;
  output [31:0] out;
  reg [31:0] out;
  always @(posedge clock or posedge reset)
    if(reset == 1) out <= 0;
    else if(en == 1) out <= in;
endmodule
    
```

(b) Verilog HDL of the registers.

```

module ctrl(clock, reset, start, cond, sm0, en0, en1, en2,
  en3, en4);
  input clock, reset, start, cond;
  output reg en0, en1, en2, en3, en4, sm0;
  reg [2:0] state, next_state;
  always @(posedge clock or posedge reset)begin
    if(reset == 1)
      state<= 0;
    else
      state<=next_state;
  end
  always @(state or start or cond) begin
    case(state)
      0: if(start) next_state<=1; else next_state<=state;
      1: if(cond==1) next_state<=2; else next_state<=3;
      2: next_state<=0;
      3: next_state<=0;
      default: if(start)next_state<=1; else next_state<=state;
    endcase
  end
  always @(state) begin
    case(state)
      0: begin
          en0<=1; en1<=1; en2<=1; en3<=0; en4<=0; sm0<=0;
        end
      1: begin
          en0<=0; en1<=0; en2<=0; en3<=1; en4<=0; sm0<=0;
        end
      2: begin
          en0<=0; en1<=0; en2<=0; en3<=0; en4<=1; sm0<=0;
        end
      3: begin
          en0<=0; en1<=0; en2<=0; en3<=0; en4<=1; sm0<=1;
        end
      default: begin
          en0<=0; en1<=0; en2<=0; en3<=0; en4<=0; sm0<=0;
        end
    endcase
  end
endmodule
    
```

(c) Verilog HDL of the control module.



(d) RTL structure.

Figure 4.4: Example of non-pipelined synchronous RTL model.


```

module sample(clock, reset, istart, stall, in0, in1, in2,
             out0);
    input clock, reset, istart, stall;
    input [31:0] in0, in1, in2;
    output [31:0] out0;
    wire [31:0] reg0_out, reg1_out, reg2_out;
    wire [31:0] add0_out, mul0_out;
    wire en0, ten0, en1, ten1, en2, ten2;

    assign ten0 = en0 & stall;
    dff32 reg0(.in(in0), .reset(reset), .clock(clock),
              .en(ten0), .out(reg0_out));
    assign ten1 = en1 & stall;
    dff32 reg1(.in(add0_out), .reset(reset), .clock(clock),
              .en(ten1), .out(reg1_out));
    assign ten2 = en2 & stall;
    dff32 reg2(.in(mul0_out), .reset(reset), .clock(clock),
              .en(ten2), .out(reg2_out));
    sub32 sub0(.a(reg0_out), .b(10), .out(sub0_out));
    mul32 mul0(.a(reg1_out), .b(3), .out(mul0_out));
    assign out = reg2_out;
    ctrl ctrl(.clock(clock), .reset(reset), .istart(istart),
             .stall(stall), .en0(en0), .en1(en1), .en2(en2));
endmodule

module ctrl(clock, reset, istart, stall, en0, en1, en2,
           en3, en4, en5);
    input clock, reset, istart, stall;
    output en0, en1, en2, en3, en4, en5;
    wire creg0_out, creg1_out;

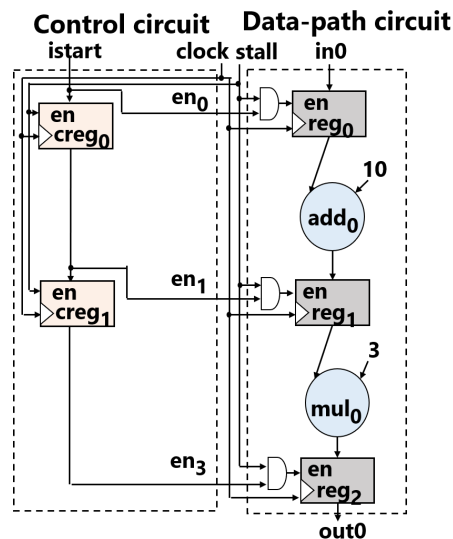
    dff1 creg0(.in(istart), .reset(reset), .clock(clock),
              .en(stall), .out(creg0_out));
    dff1 creg1(.in(creg0_out), .reset(reset), .clock(clock),
              .en(stall), .out(creg1_out));

    assign en0 = start;
    assign en1 = creg0;
    assign en2 = creg1;
endmodule

```

(a) Verilog HDL of the top-level module.

(b) Verilog HDL of the control module.



(c) RTL structure.

Figure 4.5: Example of pipelined synchronous RTL model.

the stall if pipeline stalls are included in pipelined synchronous RTL models. Note that `creg0` and `creg1` represent registers in the control circuit.

4.2.2 Info-XML

In the Info-XML, we describe the information required for the conversion except for the Model-XML. Figure 4.6 shows an example of the Info-XML. The Info-XML begins with `<parameter>`.

The Info-XML includes information required to analyze synchronous RTL models. `<synctop>` represents the top-level module name of a particular synchronous RTL

```

<parameter>
  <synctop name="sample"/>
  <ctrlmodule name=""/>
  <fsm name="global_state" reset="0"/>
  <fsmnext name="global_state_next"/>
  <spipeline mode="N" stage="creg0, creg1" interval="1" control="stage" stall="Y"/>
  <clk name="clock"/>
  <rst name="reset"/>
  <timeunit unit="ps"/>
  <cycle clock="1000"/>
  <tool target="ASIC"/>
  <asynctop name="samplea"/>
  <apipeline mode="N" interval="1"/>
  <primitive>
    <BUF cell="BUF" In="A" Out="Y" init=""/>
    <INV cell="INV" In="A" Out="YB" init=""/>
    <DLATCH cell="DLATCH" D="DATA" Q="Q" CLR=""
      ICLR="RB" G="GT" GE="" init=""/>
    <AND cell="AND2" A="A" B="B" Out="Y" init=""/>
    <NAND cell="NAND2" A="A" B="B" Out="YB" init=""/>
    <OR cell="OR2" A="A" B="B" Out="Y" init=""/>
    <NOR cell="NOR2" A="A" B="B" Out="YB" init=""/>
    <XOR cell="XOR2" A="A" B="B" Out="Y" init=""/>
    <XNOR cell="XNOR2" A="A" B="B" Out="YB" init=""/>
    <FF cell="DFF" D="DATA" Q="Q" R="RB" CLK="CLK" init=""/>
  </primitive>
</parameter>

```

Figure 4.6: Example of Info-XML.

model; $\langle ctrlmodule \rangle$ represents the control circuit name; $\langle fsm \rangle$ represents the state variable name; $\langle fsmnext \rangle$ represents the next state variable name; $\langle spipeline \rangle$ represents the information of pipeline stages; $\langle clk \rangle$ represents the global clock signal name; $\langle rst \rangle$ represents the global reset signal name; $\langle timeunit \rangle$ represents the time unit for delays; and $\langle cycle \rangle$ represents the target clock cycle time. In $\langle spipeline \rangle$, "mode" represents whether the circuit is a pipelined circuit, "stage" represents register names to control pipeline stages, "interval" represents the number of the input interval, "control" represents whether the control circuit has an FSM, and "stall" represents whether the number of stall signals is one.

In addition, the Info-XML includes information required to generate asynchronous RTL models. $\langle tool \rangle$ represents the synthesis target name. In $\langle tool \rangle$, we can describe "Quartus" for Intel FPGAs, "Vivado" for Xilinx FPGAs, or "ASIC". $\langle asynctop \rangle$ represents the top-level module name of the asynchronous RTL model, $\langle apipeline \rangle$ represents the information of pipeline stages, , and $\langle primitive \rangle$ represents the primitive cells used in the control modules. In $\langle apipeline \rangle$, "mode" represents whether the circuit is a pipelined circuit and "interval" represents the number of the input interval. In $\langle primitive \rangle$, we describe the cell name $cell$, I/O pin name such as In or Out , and initial value $init$ for Xilinx FPGAs. Primitive cells are used to prevent hazardous behaviors in control modules and preserve the correct timing required for bundled-data implementation.

4.3 Sync2XML

Sync2XML generates a Model-XML from a given synchronous RTL model and Info-XML through an analysis of the synchronous RTL model and a generation of a CDFG.

4.3.1 Parser

Pyverilog is a tool kit that uses Python to analyze Verilog HDL models. *Pyverilog* consists of a syntax analyzer, data-flow analyzer, control flow analyzer, and code generator; the parser generates an AST from a given Verilog HDL. The data-flow analyzer generates graphs to define each signal from the AST. The control flow analyzer generates an FSM from results in the data-flow analyzer. *Pyverilog* assumes that the control circuit is represented by one FSM. The code generator generates Verilog HDL from the AST. We use an AST and a control flow for a given synchronous RTL model to generate the Model-XML.

The AST represents the structure of RTL models. Figure 4.7 shows one part of the AST for the synchronous RTL model. In the AST, "Portlist" represents I/O ports, "Always" represents "always" statements, "Assign" represents "assign" statements, and "Instance" represents instantiated resources. The proposed method recognizes a signal name, bit width, resource type, etc. from the AST.

The control flow represents state transitions in the FSM of a synchronous RTL model. Figure 4.8 shows the control flow for the synchronous RTL model. Values described using decimal numbers represent state variables. The arrows represent the state transition and "(" represents the condition of the state transition. The proposed method recognizes the states and state transitions from the control flow.

4.3.2 Generation of a CDFG

In asynchronous circuits, data-path resources in each $stage_i$ ($state_i$) are controlled by each $ctrl_i$. *Sync2XML* generates a CDFG from the AST and control flow generated by *Pyverilog* to determine the data-path resources controlled by each $stage_i$.

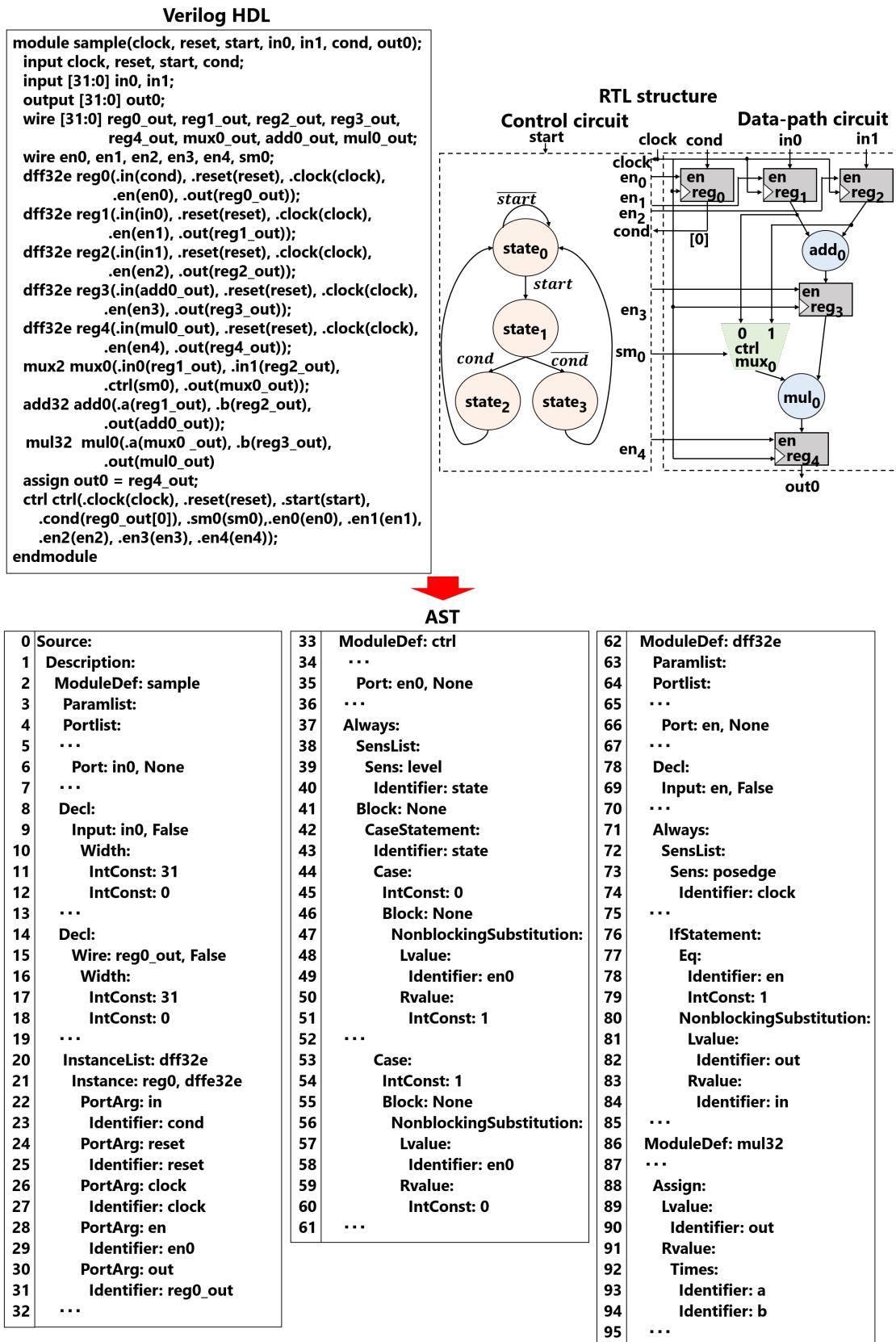
The CDFG used in this dissertation represents the control flow and data flow in synchronous RTL models. *Sync2XML* generates a CFG and DFG from the AST and control flow. Subsequently, *Sync2XML* generates a CDFG by combining the CFG and DFG.

Generation of a CFG

Sync2XML generates a CFG from the AST and control flow. The CFG comprises nodes and edges. The nodes represent states of the FSM or registers to control the pipeline stages. The nodes have a control signal name with its value. The edges represent a dependence between nodes.

Sync2XML generates nodes and edges from the AST and control flow. The method for generating nodes and edges depends 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 nodes from values described using decimal numbers. *Sync2XML* extracts the control signal name with its value from "("; moreover, *Sync2XML* generates edges from arrows. After generating the edges, *Sync2XML* eliminates nodes without incoming edges and nodes with only the self-loop. Further, *Sync2XML* eliminates incoming or outgoing edges that correspond to these nodes. Subsequently, for the remaining nodes, *Sync2XML* reassigns node labels from the smallest node label. Finally, *Sync2XML* adds the primary input signal *start* that triggers the FSM from the outside. As the proposed method assumes only one FSM, we generate only one CFG from the given synchronous RTL models.

Figure 4.7: Example of AST generated using *Pyverilog* [33].

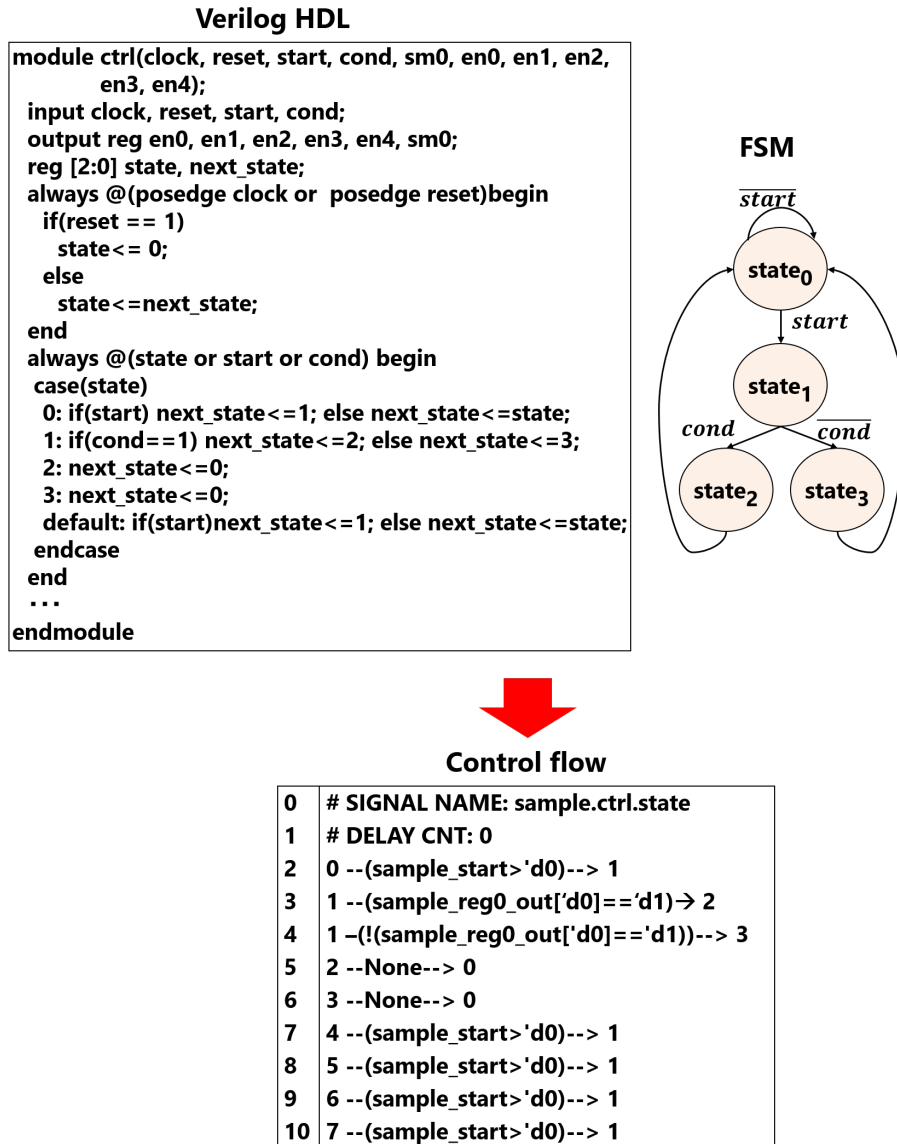
Figure 4.8: Example of control flow generated by *Pyverilog* [33].

Figure 4.9 shows an example of the generation of the CFG from the control flow. *Sync2XML* generates the eight nodes from the control flow. For example, node 1 is generated from line 3 in the control flow. As an example on edge generation, the edge from nodes 1 to 2 is generated from line 3 in the control flow. After generating the edges, *Sync2XML* eliminates nodes 4, 5, 6, and 7 without incoming edges. Finally, *Sync2XML* moves the input signal *start* to trigger the CFG.

In the absence of control flow, *Sync2XML* generates nodes and edges from the AST. *Sync2XML* generates nodes from the "Lvalue" or "Instance" for the control circuit in the AST. *Sync2XML* extracts the label for the nodes from the variable name for the "Lvalue" or "Instance" in the AST. Further, *Sync2XML* extracts the control signal with its value for the nodes from the "IfStatement" or "CaseStatement" in the AST. In addition, *Sync2XML* generates edges from the "Rvalue" or "PortArg" in the AST.

Figure 4.10 shows an example of the generation of the CFG from the AST. Red represents an example of node generations and blue represents an example of edge

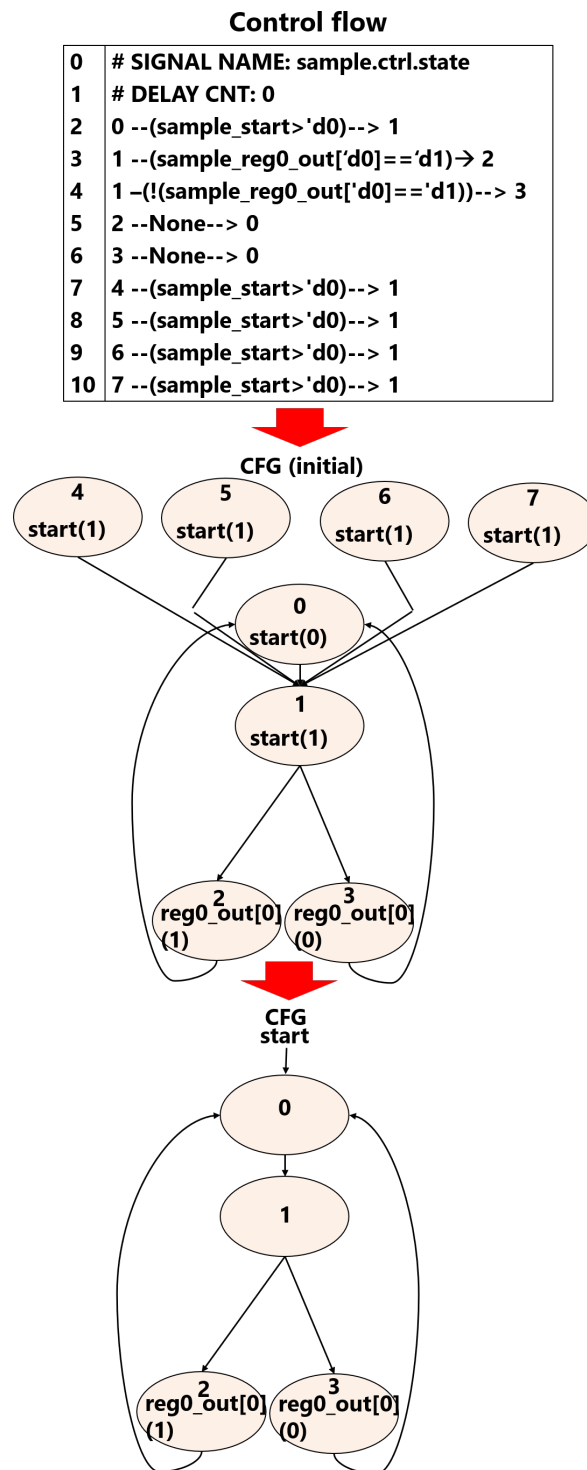


Figure 4.9: Example of the generation of a CFG from a control flow.

generations. As an example on node generation, node $creg_1$ is generated from the "Instance" at line 3 in the AST. As an example on edge generation, the edge from $istart$ to $creg_0$ is generated from the "PortArg" at line 4 in the AST.

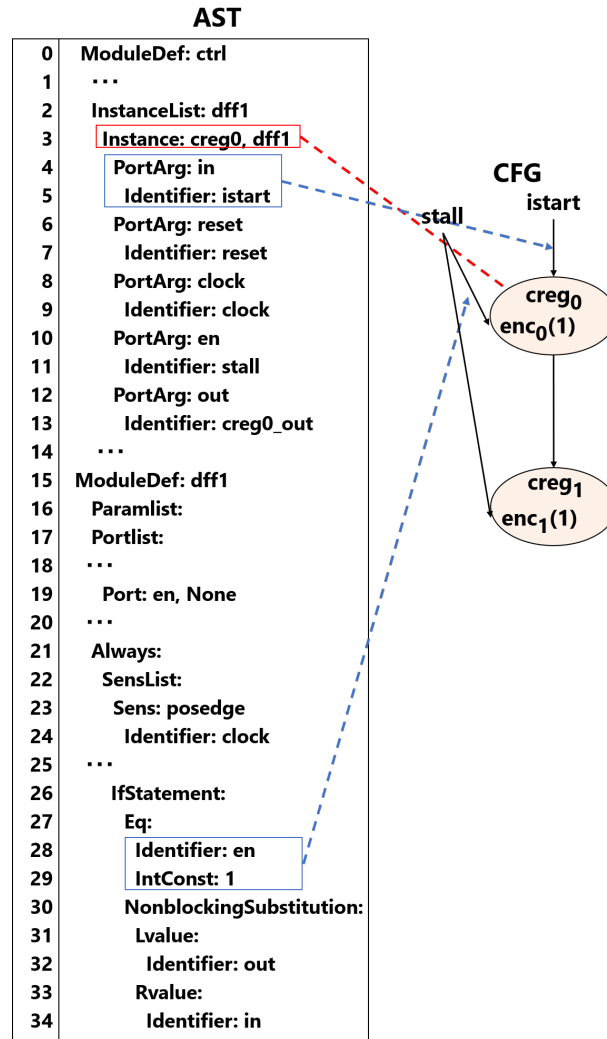


Figure 4.10: Example of the generation of a CFG from an AST.

Generation of a DFG

Sync2XML generates a DFG from the AST. The DFG comprises nodes and edges. The nodes represent resources such as registers and functional units in the data-path circuit. The nodes have a control signal name and a value except for functional units and basic logic operations. The edges represent a dependence between nodes.

Sync2XML generates nodes from the "Lvalue" or "Instance" in the AST. *Sync2XML* extracts the label for the nodes from the variable name. Further, *Sync2XML* extracts the control signal with its value for the nodes from the "IfStatement" or "CaseStatement" in the AST.

After generating the nodes, *Sync2XML* generates edges from the "Rvalue" or "PortArg" in the AST. Further, *Sync2XML* extracts the bit width for the edge from the "Pointer" or "Partselect" in the AST.

Figure 4.11 shows an example of the generation of the DFG from the AST. Red represents an example of node generations and blue represents an example of edge generations. For example, node reg_1 is generated from the "Instance" at line 37 in the AST, and the edge from reg_1 to add_0 is generated from the "PortArg" at line 44 in the

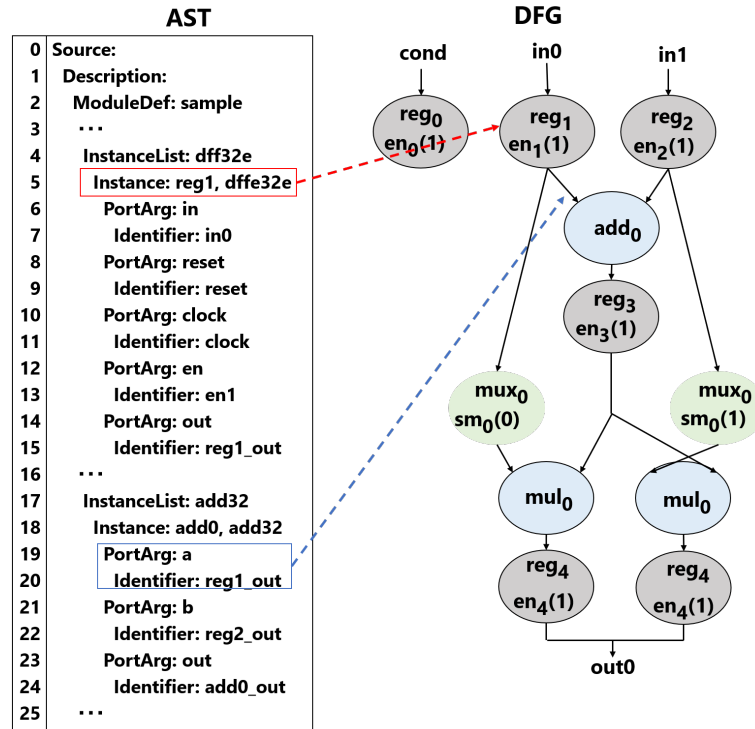


Figure 4.11: Example of the generation of a DFG from an AST.

AST.

Combination of CFG and DFG

Sync2XML generates a CDFG by combining the CFG and DFG. The CDFG comprises nodes, edges, and $state_i$ ($stage_i$). Between registers represent $state_i$ ($stage_i$). Further, $state_i$ ($stage_i$) has a conditional signal $cond$ and its value val to begin the operation in $state_i$ ($stage_i$).

Sync2XML combines the CFG and DFG by generating edges from the AST and control flow. In addition, *Sync2XML* generates edges from the CFG to DFG by referring to the "IfStatement," "CaseStatement," and "PortArg" in the AST. Further, *Sync2XML* generates edges from the DFG to CFG by referring to "()" in the control flow.

Sync2XML considers between registers as $state_i$ ($stage_i$). The extraction method for $cond$ and val for $state_i$ ($stage_i$) differs based on whether there is the control flow. Without the control flow, *Sync2XML* extracts $cond$ and val for $state_i$ ($stage_i$) from the "IfStatement" and "CaseStatement" for the control circuit in the AST. With the control flow, *Sync2XML* extracts $cond$ and val for $state_i$ ($stage_i$) from "()" in the control flow.

Moreover, the extraction method for $cond$ and val for $state_i$ ($stage_i$) differs based on whether there are stall signals. *Sync2XML* extracts $cond$ and val from the AST and control flow without a stall signal. With stall signals, the extraction method differs based 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 simultaneously resumed by one stall signal. In contrast, *Sync2XML* extracts $cond$ and val from the AST and control flow in the presence of multiple stall

signals. If the II is different, *Sync2XML* generates a CDFG in the same manner.

Figure 4.12 shows an example of the generation of a CDFG. Red represents an example of edges from the CFG to the DFG and blue represents an example of edges from the DFG to the CFG. The edge from node 0 to node reg_0 is generated from the "CaseStatement" at line 10 in the AST as an example of edges from the CFG to the DFG. The edge from node reg_0 to node 2 is generated from "()" at line 3 in the control flow as an example of edges from the DFG to the CFG. We consider between reg_3 and reg_4 as $stage_2$ as an example on statei generations. The conditional signal $reg_0-out[0]$ and its value 1 are given to $stage_2$ from "()" at line 3 in the control flow. Figure 4.13 shows the generated CDFG for the pipelined synchronous RTL model of Fig. 4.5.

4.3.3 Generation of Model-XML

Generation of resource information

Sync2XML generates the resource information $\langle resource_info \rangle$ in the Model-XML from the AST and Info-XML. Figure 4.14 represents the generated resource information in the Model-XML from the AST.

In $\langle resource_info \rangle$, *Sync2XML* represents all resources and input/output pins in the data-path circuit in the synchronous RTL model using $\langle resource \rangle$. *Sync2XML* generates each $\langle resource \rangle$ from "Instance," "Assign," and "Port" in the AST. First, *Sync2XML* obtains the resource or input/output name $name$ from "Instance," "Assign," and "Port." Further, *Sync2XML* recognizes the resource type $type$ from "ModuleDef," "Rvalue" in "Assign," and "Decl" using the label of "Instance," "Assign," and "Port." Then, *Sync2XML* obtains the control signal name $ctrl_name$ with its bit width from "PortArg" in "Instance," "IfStatement" in "ModuleDef," and "Decl." Moreover, *Sync2XML* obtains the assignment type $substitution$ from "NoblockingSubstitution," "BlockingSubstitution," and "Assign." For "BlockingSubstitution" ("Assign"), *Sync2XML* assigns the number to "BlockingSubstitution" ("Assign") in the order in which they appear. Finally, *Sync2XML* obtains the bit width bit from "Width" in "Decl."

Figure 4.14 shows an example of the generation of resource information from the AST. $\langle resource\ id = "5" \rangle$ in the resource information represents the 32-bit register. *Sync2XML* obtains the resource name from the "Instance" at line 21 and assigns reg_0 to $name$. Thereafter, *Sync2XML* obtains the bit width from the "Width" in the "Decl" at line 14 and assigns 32 to bit . Further, *Sync2XML* obtains the resource type from the "ModuleDef" at line 33 and assigns reg to $type$. Moreover, *Sync2XML* obtains the control signal name from the "PortArg" at line 28 and "IfStatement" at line 47 and assigns $en0;1$ to $ctrl_name$. Finally, *Sync2XML* obtains the assignment type from the "NoblockingSubstitution" at line 51 and assigns nb to $substitution$.

Generation of path information

Sync2XML generates the path information $\langle path_info \rangle$ from the CDFG. The path information comprises data-path information $\langle datapath \rangle$, state information $\langle ctrlpath \rangle$, and state transition information $\langle loop \rangle$ in the control circuit.

In $\langle datapath \rangle$, *Sync2XML* represents a data-path using $\langle path \rangle$. For the CDFG, *Sync2XML* generates each $\langle path \rangle$ by checking the path from the destination register or output pin to the source register or input pin. First, *Sync2XML* obtains the start point $start_name$, through point th_name , and end point end_name from the nodes in

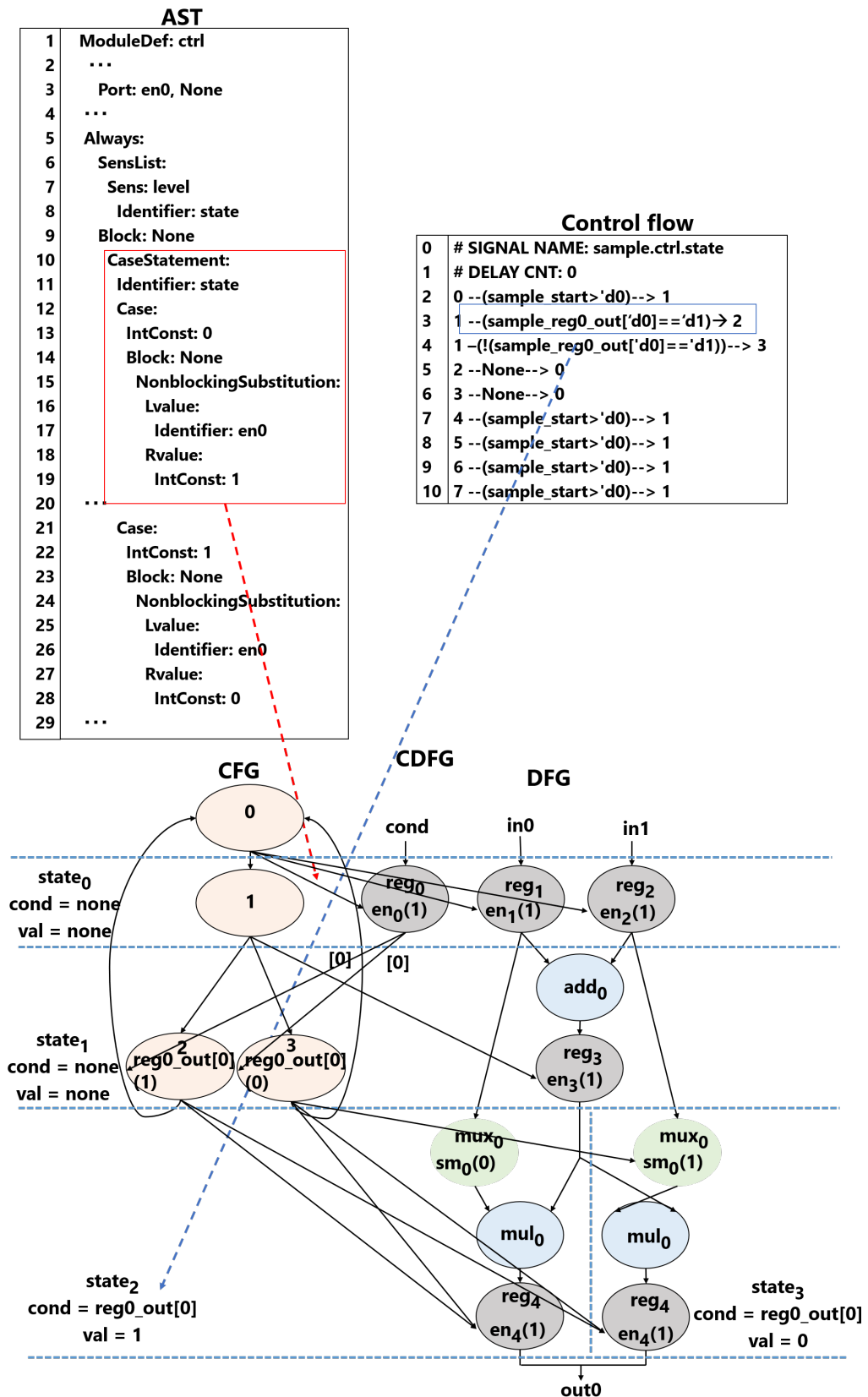


Figure 4.12: Example of the generation of a CFG.

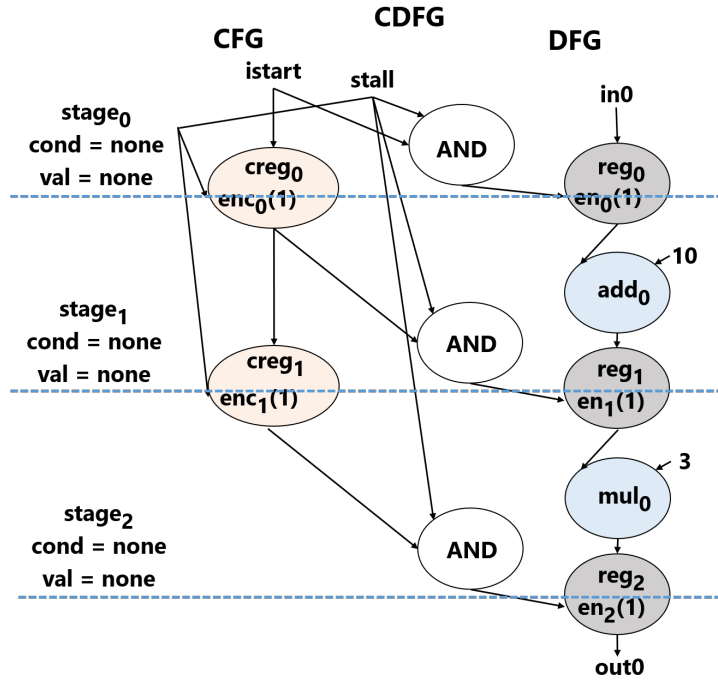


Figure 4.13: Generated CDFG from Fig. 4.5.

the CDFG. Subsequently, *Sync2XML* obtains the bit width for the start point $start_bit$ and bit width for the through point th_bit from the edges in the CDFG. If edges do not have the bit-width information, $start_bit$ or th_bit is empty. Finally, *Sync2XML* obtains the input operand of the the through point th_in and input operand of the the end point end_in from the edges in the CDFG. If the path contains several through points, *Sync2XML* represents all of them adding the identifier to th_name (i.e., $th0_name$ and $th1_name$).

Figure 4.15 shows an example of the generation of data-path information from the CDFG. $\langle path\ id = "3" \rangle$ in the data-path information represents the path from reg_1 to reg_3 through add_0 . *Sync2XML* obtains the start point name from the label in reg_1 and assigns reg_1 to $start_name$. Thereafter, *Sync2XML* obtains the through point name from the label in add_0 and assigns add_0 to $th0_name$. In addition, *Sync2XML* obtains the input operand of add_0 from the edge from reg_1 to add_0 and assigns 0 to $th0_in$. Further, *Sync2XML* obtains the end point name from the label in reg_3 and assigns reg_3 to end_name . In addition, *Sync2XML* obtains the input operand of reg_3 from the edge from add_0 to reg_3 and assigns 0 to end_in .

Sync2XML analyzes preceding and succeeding states (pipeline stages) for each $state_i$ ($stage_i$) before $\langle ctrlpath \rangle$ is generated. In the CDFG, $state_j$ ($stage_j$) ($j \neq i$) represents a succeeding state (pipeline stage) for $state_i$ ($stage_i$) when the resources of $state_i$ ($stage_i$) are connected to the resources of $state_j$ ($stage_j$). In contrast, $state_j$ ($stage_j$) represents a preceding pipeline stage for $state_i$ ($stage_i$) when the resources of $state_i$ ($stage_i$) are connected to the resources of $state_j$ ($stage_j$). Further, *Sync2XML* extracts a conditional signal and its value for the transition between states (pipeline stages) from the $cond$ and val of $state_j$ ($stage_j$).

Sync2XML generates the control-path information after analyzing $state_i$ ($stage_i$). *Sync2XML* generates the control-path information for each $state_i$ ($stage_i$) using $\langle ctrl \rangle$.

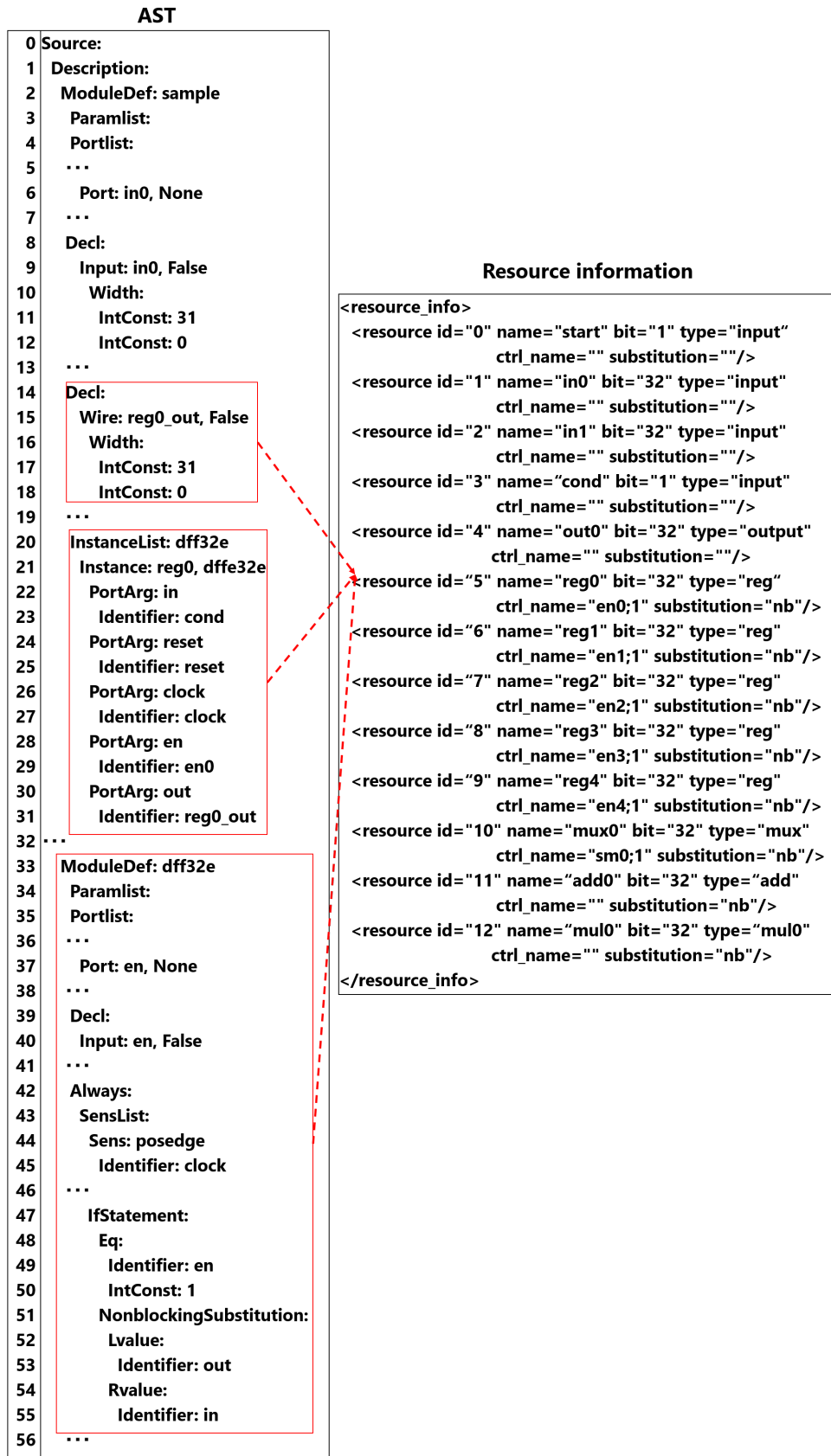
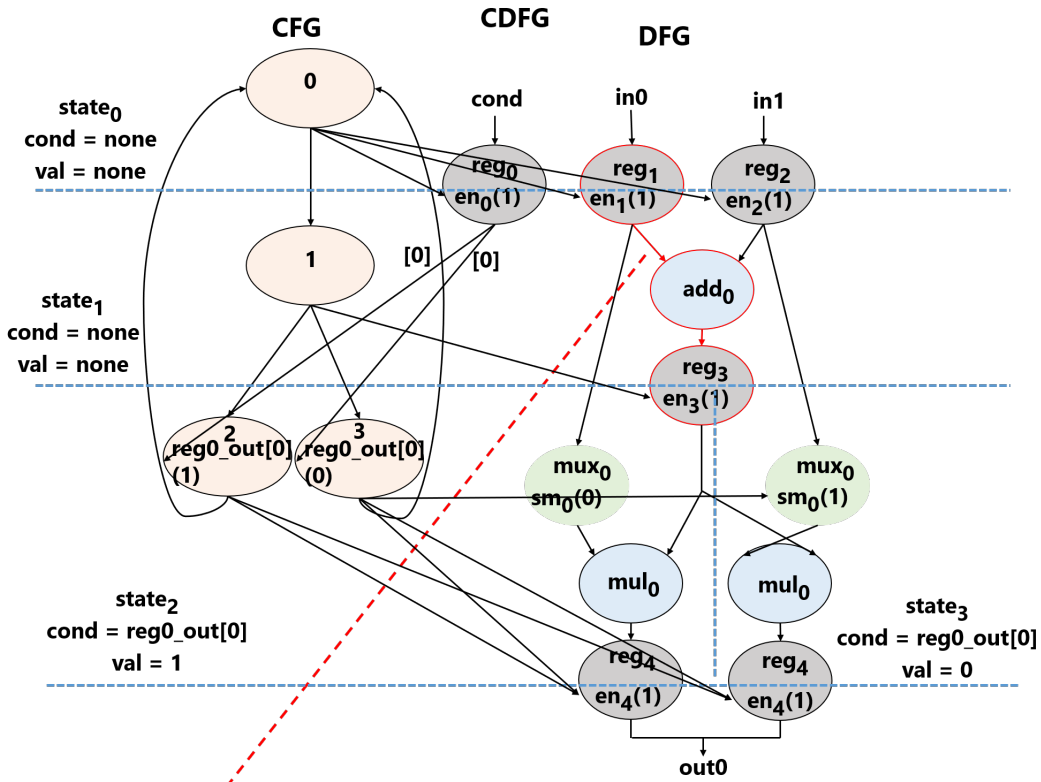


Figure 4.14: Example of resource information.



Data-path information

```

<path_info>
<datapath>
<path id="0" start_name="cond" start_usebit="" end_name="reg0" end_in="0"/>
<path id="1" start_name="in0" start_usebit="" end_name="reg1" end_in="0"/>
<path id="2" start_name="in1" start_usebit="" end_name="reg2" end_in="0"/>
<path id="3" start_name="reg1" start_usebit=""
  th0_name="add0" th0_in="0" th0_usebit=""
  end_name="reg3" end_in="0"/>
<path id="4" start_name="reg2" start_usebit=""
  th0_name="add0" th0_in="1" th0_usebit=""
  end_name="reg3" end_in="0"/>
<path id="5" start_name="reg1" start_usebit=""
  th0_name="mux0" th0_in="0" th0_usebit=""
  th1_name="mul0" th1_in="0" th1_usebit=""
  end_name="reg4" end_in="0"/>
<path id="6" start_name="reg2" start_usebit=""
  th0_name="mux0" th0_in="1" th0_usebit=""
  th1_name="mul0" th1_in="0" th1_usebit=""
  end_name="reg4" end_in="0"/>
<path id="7" start_name="reg3" start_usebit=""
  th0_name="mul0" th0_in="1" th0_usebit=""
  end_name="reg4" end_in="0"/>
<path id="8" start_name="reg4" start_usebit="" end_name="out0" end_in="0"/>
</datapath>
...
</path_info>

```

Figure 4.15: Example of data-path information.

Further, $\langle ctrl \rangle$ represents $stage_i$ ($stage_i$) and *Sync2XML* reassigns labels by converting $state_i$ to s_i . Subsequently, *Sync2XML* generates preceding state (pipeline stage) information $\langle pred \rangle$ and succeeding state (pipeline stage) information $\langle succ \rangle$ into $\langle ctrl \rangle$. Here, $\langle pred \rangle$ and $\langle succ \rangle$ represent preceding and succeeding states (pipeline stages) for each $stage_i$ ($stage_i$), respectively. Moreover, *Sync2XML* assigns a conditional signal $ctrlname$ and its value $ctrlval$ to operate preceding or succeeding states (pipeline stages) to $\langle pred \rangle$ or $\langle succ \rangle$. If the previous states are traversed by a feedback state transition, *Sync2XML* represents *feedback* as 1.

Figure 4.16 shows an example of the generation of control-path information from the CDFG. $\langle path\ id = "2" \rangle$ in the control-path information represents the control-path information that corresponds $state_2$. *Sync2XML* obtains the state name from the label in $state_2$ and assigns s_2 to *name*. Thereafter, *Sync2XML* obtains the preceding state name from $state_1$ and assigns s_1 to *name* in $\langle predecessor \rangle$. Further, *Sync2XML* obtains the control signal name and its value from *cond* and *val* in $state_1$ and assigns $reg_{out}[0]$ to $ctrl_name$ and 1 to $ctrl_val$. In addition, *Sync2XML* obtains the succeeding state name from $state_0$ and assigns s_0 to *name* in $\langle successor \rangle$.

After generating $\langle ctrl \rangle$, *Sync2XML* generates $\langle loop \rangle$ as follows: The state group $\langle group \rangle$ of the traversed states are defined until the outgoing state transition becomes feedback. For each state in $\langle group \rangle$, *Sync2XML* generates the initial state *start*, intermediate state *th*, and end state *end* in the state transition.

Generation of timing information

Sync2XML generates the timing information $\langle timing \rangle$ from the CDFG. $\langle timing \rangle$ comprises the register write signal information $\langle reg \rangle$ and the multiplexer control signal information $\langle mux \rangle$.

Sync2XML analyzes the values of the register write signals and multiplexer control signals before generating $\langle timing \rangle$. The value of the register write signal for reg_k is the control value held by reg_k for $state_i$ if reg_k exists in $state_i$ on the CDFG. The value of the multiplexer control signal for mux_l is the control value held by mux_l for $state_i$ if mux_l exists in $state_i$ on the CDFG.

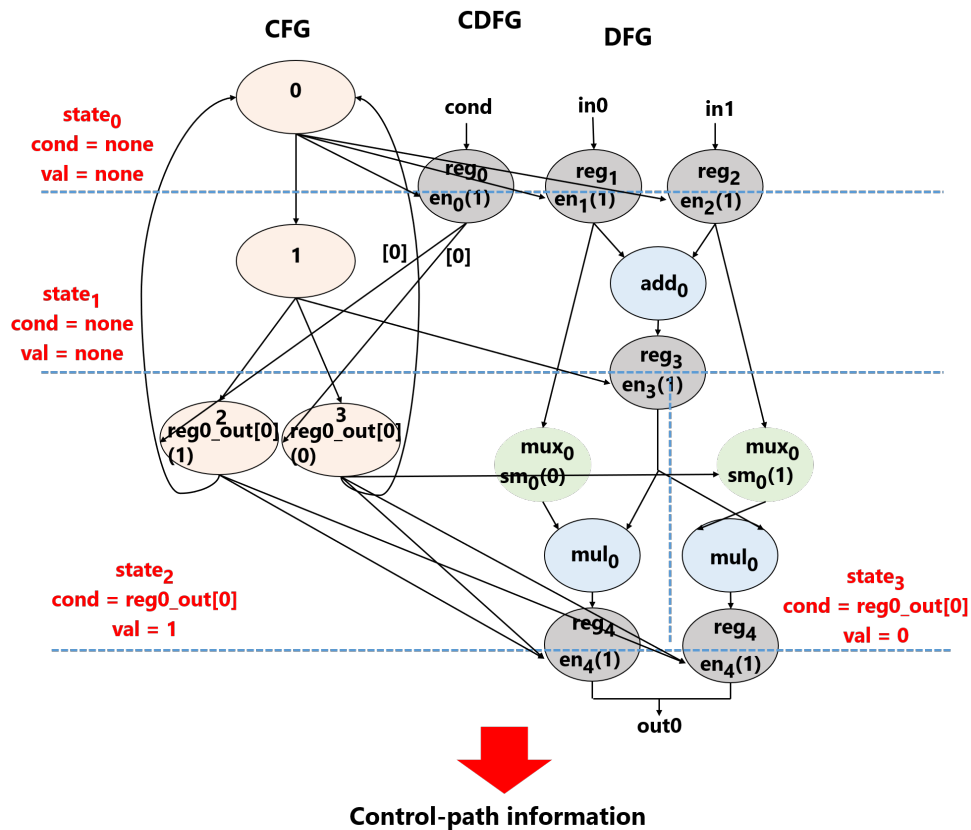
Sync2XML generates the register write signal information and the multiplexer control signal information for each control signal of registers and multiplexers using $\langle reg \rangle$ and $\langle mux \rangle$ after analyzing $state_i$ ($stage_i$). Finally, *Sync2XML* assigns the control values to $\langle reg \rangle$ and $\langle mux \rangle$ from the analyzed values of control signals.

The generation method differs based on whether there is a pipeline stall. *Sync2XML* generates the register write signal information for the components to use DFFs for stall signals.

Figure 4.17 shows an example of the generation of timing information from the CDFG. $\langle reg\ id = "0" \rangle$ in the timing information represents the register write signal of reg_0 . *Sync2XML* obtains the register write signal from the control name in reg_0 and assigns en_0 to *name*. Thereafter, it obtains the value of the register write signal from the control value in reg_0 and assigns 1 to s_0 .

4.4 XML2Async

XML2Async generates an asynchronous RTL model with bundled-data implementation from the Model-XML and Info-XML. Moreover, *XML2Async* generates an asyn-



```

<path_info>
...
<ctrlpath>
  <ctrl id="0" name="s0">
    <predecessor id="0" name="start" ctrl_name="" ctrl_value="" feedback=""/>
    <predecessor id="1" name="s2" ctrl_name="" ctrl_value="" feedback="1"/>
    <predecessor id="2" name="s3" ctrl_name="" ctrl_value="" feedback="1"/>
    <successor id="0" name="s1" ctrl_name="" ctrl_value="" feedback=""/>
  </ctrl>
  <ctrl id="1" name="s1">
    <predecessor id="0" name="s0" ctrl_name="" ctrl_value="" feedback=""/>
    <successor id="0" name="s2" ctrl_name="reg_out[0]" ctrl_value="1" feedback=""/>
    <successor id="1" name="s3" ctrl_name="reg_out[0]" ctrl_value="0" feedback=""/>
  </ctrl>
  <ctrl id="2" name="s2">
    <predecessor id="0" name="s1" ctrl_name="reg_out[0]" ctrl_value="1" feedback=""/>
    <successor id="0" name="s0" ctrl_name="" ctrl_value="" feedback="1"/>
  </ctrl>
  <ctrl id="3" name="s3">
    <predecessor id="0" name="s1" ctrl_name="reg_out[0]" ctrl_value="0" feedback=""/>
    <successor id="0" name="s0" ctrl_name="" ctrl_value="" feedback="1"/>
  </ctrl>
</ctrlpath>
<loop>
  <group id="0" start="s0" th0="s1" end="s2"/>
  <group id="1" start="s0" th0="s1" end="s3"/>
</loop>
</path_info>

```

Figure 4.16: Example of control-path information.

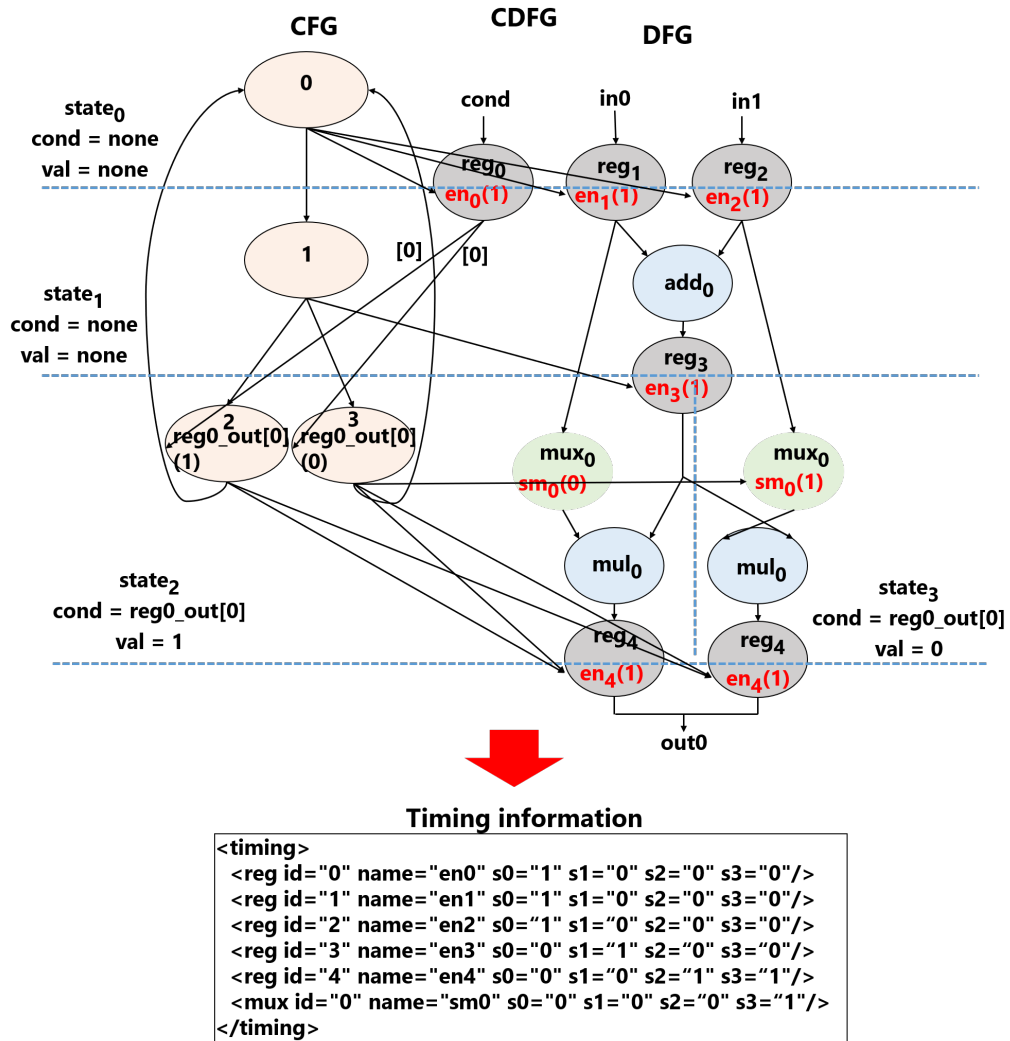


Figure 4.17: Example of timing information.

chronous RTL simulation model when the target implementation is an FPGA and a set of non-optimization constraints for preventing the optimization of primitive cells in the control circuit.

4.4.1 Generation of Asynchronous RTL Models

XML2Async generates an asynchronous RTL model by assigning data-path resources, assigning control modules, and generating a top-level module.

Assigning Data-Path Resources

XML2Async assigns data-path resources by referring to $\langle resource \rangle$ in the resource information $\langle resource_info \rangle$ of the Model-XML. For each $\langle resource \rangle$, *XML2Async* obtains the resource name from *name*, bit width from *bit*, resource type *type*, control signal name with its bit width from *ctrl_name*, the assignment type from *substitution*. In addition, *XML2Async* obtains the input signal names, bit width, and input operand numbers by referring to $\langle datapath \rangle$ in $\langle path_info \rangle$.

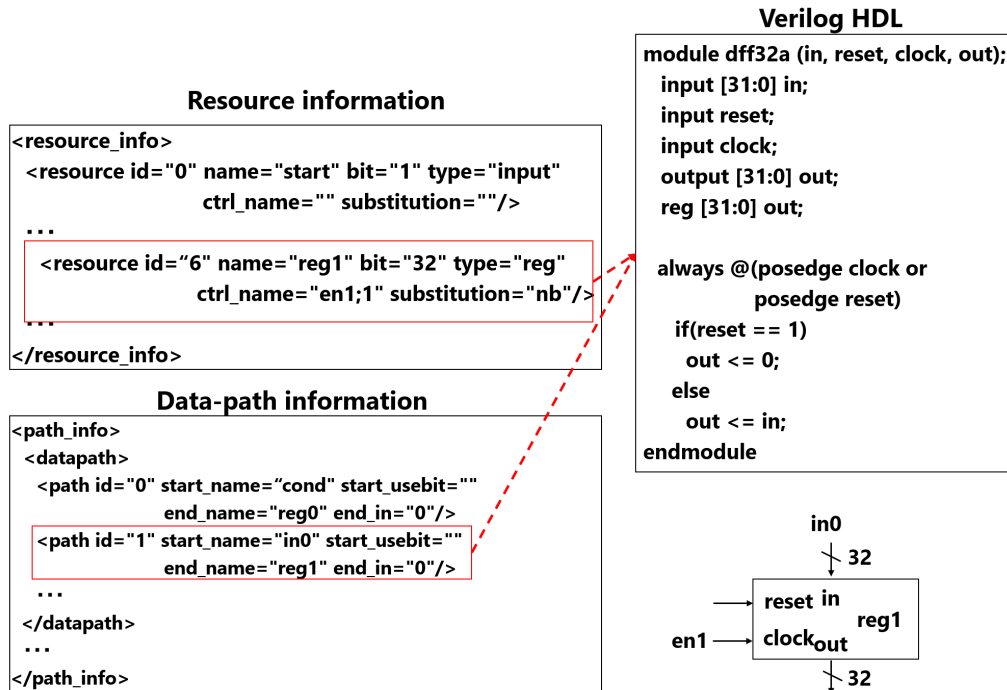


Figure 4.18: Generated Verilog HDL model of the register.

Figure 4.18 represents generated Verilog HDL models from $\langle resource\ id = "6" \rangle$ in the resource information of Fig. 4.18. First, *XML2Async* obtains the resource name as reg_1 from *name*. Thereafter, it recognizes the resource type as the register because *type* is *reg*. In addition, it recognizes the output bit width as 32 from *bit* and the assignment type as the non-blocking substitution from *substitution*. Further, *XML2Async* recognizes the control signal name as *en1* and the control signal bit width as 1 from *en1;1* in *ctrl_name*. Next, it extracts $\langle path\ id = "1" \rangle$, which includes the resource name reg_1 in the through points *th_name* or the end point *end_name*. From this $\langle path \rangle$, *XML2Async* obtains the input signal name as *in0* from *start_name*. Further, *XML2Async* recognizes the input signal bit width as 32, which is the same as the output bit width of reg_1 because *start_usebit* is empty. *XML2Async* obtains the input signal bit width as 16 if a description such as [15 : 0] exists in *start_usebit* or *th_usebit*. Finally, *XML2Async* obtains the input operand as 0 from *end_in*. *XML2Async* generates the Verilog HDL for registers with such information (Fig. 4.18).

XML2Async decides the module name and I/O pin names beforehand; this helps prevent multiple resources of the same structure.

Assigning Control Modules

XML2Async assigns control modules by referring to $\langle ctrlpath \rangle$ in the path information $\langle path.info \rangle$ of the Model-XML and $\langle primitive \rangle$ in Info-XML. For one $\langle ctrl \rangle$, *XML2Async* assigns one control module. If there are pipeline stalls or if II is different, *XML2Async* assigns and connects $ctrl_i$ in the same manner. Figures 4.19 depicts the Verilog HDL model and circuit structure of the generated control module $ctrl_i$.

XML2Async generates a logic w_0 from out_{i-1} or $lclk_{i-1}$ of the previous control modules $ctrl_{i-1}$ and the conditional signals from the data-path circuit for the control

```

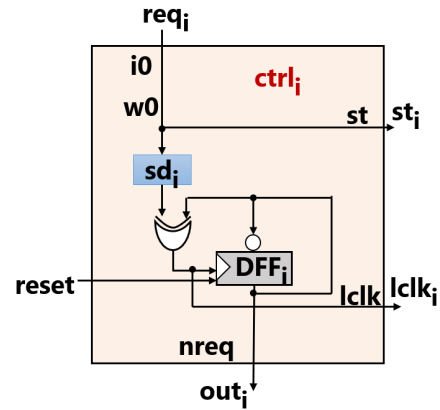
module ctrl (reset, i0, st, lclk, nreq);
  input reset, i0;
  output st, lclk, nreq;
  wire dreq, w0;
  wire ack;

  //logic
  assign w0 = i0;
  //sd
  sd sdi(.in(w0), .out(dreq));
  //lclk
  XOR2 lclk0(.A(dreq), .B(ack), .Y(lclk));
  //DFF
  DFF dff(.DATA(~ack), .RB(~reset), .CLK(lclk), .Q(ack));
  //st
  assign st = w0;

  assign nreq = ack;
endmodule

```

(a) Verilog HDL model.



(b) Circuit structure.

Figure 4.19: Generated control module $ctrl_i$.

branch. *XML2Async* searches $ctrl_{i-1}$ using the *name* of $\langle predecessor \rangle$ in $\langle ctrl \rangle$ and obtains out_{i-1} or $lclk_{i-1}$. Moreover, *XML2Async* obtains conditional signal names and their values from $ctrl_name$ and $ctrl_value$ in $\langle predecessor \rangle$.

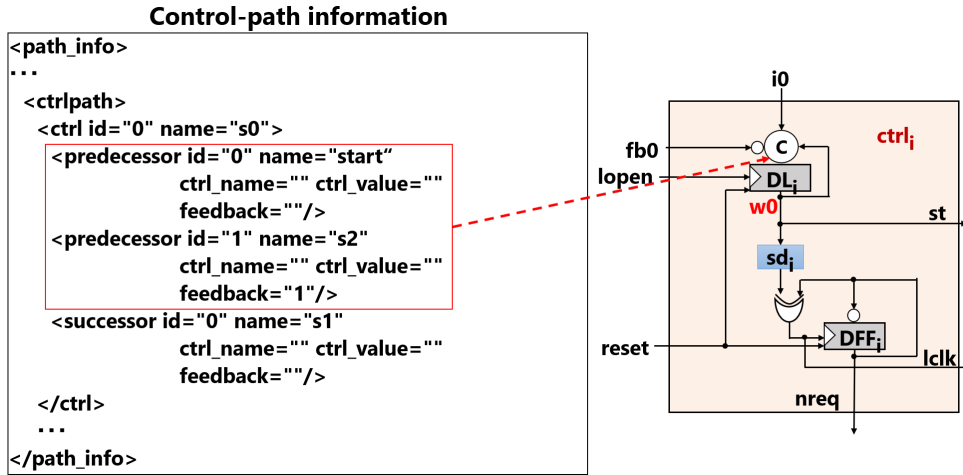
The generation method for $w0$ of $ctrl_i$ differs depending on the following conditions.

- (Condition 1) Input signals of the control module are primary input signals and feedback signals.
- (Condition 2) There are branches for the control module.
- (Condition 3) There are several previous control modules.

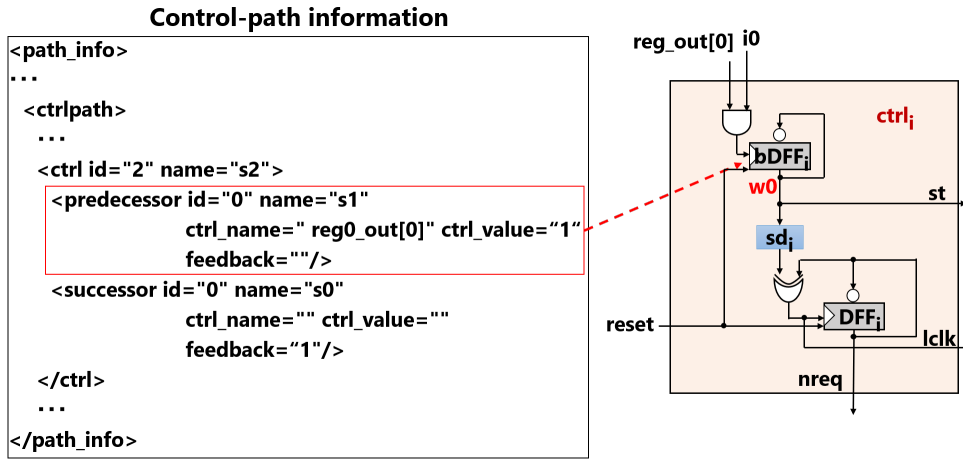
In condition 1, *XML2Async* decides whether the input signals of the control module are primary input signals and feedback signals by referring to $\langle predecessor \rangle$. *XML2Async* recognizes this *name* as a primary input signal when *name* in $\langle predecessor \rangle$ is the different from *name* of each $\langle ctrl \rangle$. *XML2Async* recognizes this *name* as a feedback signal when *feedback* in $\langle predecessor \rangle$ is 1. Here, *XML2Async* generates $w0$ using a C-element. Figure 4.20a depicts $w0$ using a C-element.

In condition 2, *XML2Async* decides whether there are branches for the control module by referring to $ctrl_name$ in $\langle predecessor \rangle$. *XML2Async* recognizes that there is a branch for the control module from the previous control module based on a conditional signal. Here, *XML2Async* generates $w0$ on the input signal using an AND gate and a DFF ($bDFF_i$). The inputs of the AND gate are $lclk_{i-1}$ from $ctrl_{i-1}$ and a conditional signal. Figure 4.20b depicts $w0$ using an AND gate.

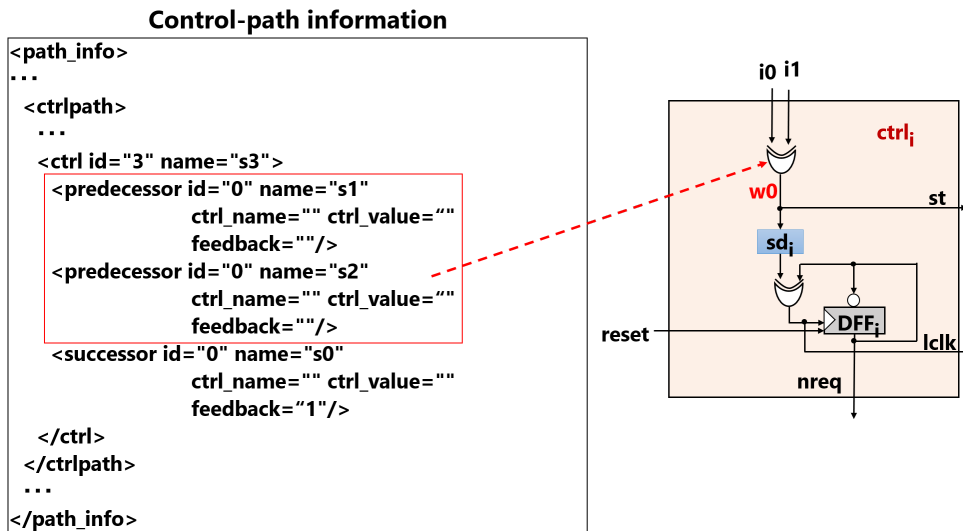
In condition 3, *XML2Async* decides whether there are several previous control modules by referring to the number of $\langle predecessor \rangle$. *XML2Async* recognizes that there are several previous modules when there are several $\langle predecessor \rangle$. Here, *XML2Async*



(a) w_0 using the C-element.



(b) w_0 using the AND gate and DFF.



(c) w_0 using the OR gate for the input signal and feedback signal.

Figure 4.20: Generation of glue logics w_0 .

generates $w0$ on the input signal using an XOR gate. Figure 4.20c depicts $w0$ using an XOR gate.

After generating $w0$, *XML2Async* generates a delay element sd_i by referring to *INV* in the $\langle primitive \rangle$ in the Info-XML. $ctrl_i$ is operated by the rising and falling transitions of req_i . Hence, sd_i comprises two inverters in that the delay does not change between the rising and falling transitions.

Finally, *XML2Async* generates $lclk_i$ and $df f_i$ by referring to $\langle primitive \rangle$ in the Info-XML. First, *XML2Async* generates $lclk_i$ from $\langle XOR2 \rangle$; *XML2Async* obtains the cell name as "XOR2" from *cell* and obtains the port names from *IN*. Thereafter, it decides the instance name as $lclk_i$. Similarly, *XML2Async* generates $df f_i$ from $\langle DFF \rangle$.

Generation of a Top-level Module

XML2Async generates a top-level module from Model-XML. *XML2Async* generates the top-level module by obtaining the top-level module and I/O signal names, instantiating data-path resources and control modules, connecting the data-path resources, connecting the control modules, and connecting between data-path resources and control modules.

First, *XML2Async* obtains the top-level module from $\langle asynctop \rangle$ in the Info-XML. In addition, *XML2Async* obtains I/O signals for the top-level module by referring to the *type* in $\langle resource \rangle$. If the *type* is *input* or *output*, it is an input or output signal, respectively.

Next, *XML2Async* instantiates all generated data-path resources and control modules. *XML2Async* obtains the instance name for the data-path resource from *name* in $\langle resource \rangle$ of the Model-XML. Further, *XML2Async* obtains the instance name for the control module from *name* in the $\langle ctrl \rangle$ of the Model-XML. *XML2Async* replaces "s" into "ctrl" in the instance name for the control module. Figure 4.21 shows the instantiated resources by referring to the resource information in Fig. 4.14 and the control-path information in Fig. 4.16.

After the instantiation of data-path resources and control modules, *XML2Async* connects between the data-path resources and control modules. For the data-path resources, *XML2Async* connects data-path resources by referring to $\langle path \rangle$ in $\langle datapath \rangle$. *XML2Async* performs the connection by providing the same signal name. *XML2Async* obtains the signal name from *start_name*, *th_name*, and *end_name*, the input operand number from *th_in* and *end_in*, and the bit width for the connection from *start_usebit* and *th_usebit*. For the control modules, *XML2Async* connects the control modules by referring to $\langle predecessor \rangle$ ($\langle successor \rangle$) in $\langle ctrl \rangle$. Further, it obtains the previous control modules names from *name* in $\langle predecessor \rangle$ and obtains the next control modules from *name* in $\langle successor \rangle$. Figure 4.22 shows the connection of data-path resources and the connection of control modules by referring to the data-path information in Fig. 4.15 and control-path information in Fig. 4.16.

XML2Async connects the data-path resources to the control modules by referring to $\langle path \rangle$ in $\langle datapath \rangle$ after the connection of the data-path resources and control modules. *XML2Async* performs the connection by providing the same signal name. *XML2Async* obtains the signal name from *start_name*, *th_name*, and *end_name*. Thereafter, *XML2Async* obtains the input operand number from *th_in* and *end_in*. Finally, *XML2Async* obtains the bit width for the connection from *start_usebit* and *th_usebit*. Figure 4.23 shows the connection from the data-path resources to the control modules

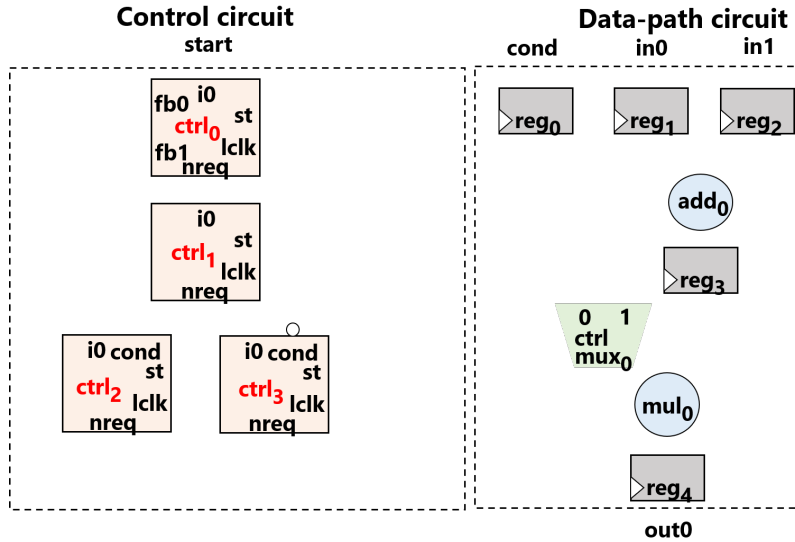


Figure 4.21: Instantiation of resources.

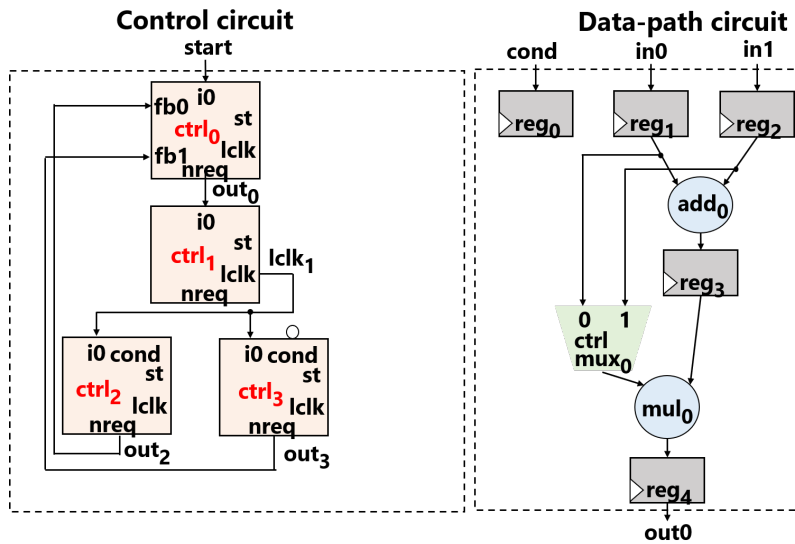


Figure 4.22: Connection of resources.

by referring to the data-path information presented in Fig. 4.15.

Finally, *XML2Async* connects control modules to data-path resources by referring to $\langle reg \rangle$ and $\langle mux \rangle$ in $\langle timing \rangle$. Further, it connects the control modules to the data-path resources through $glue_{reg_k}$ and $glue_{mux_l}$.

XML2Async generates register write signals ($glue_{reg_k}$) by referring to $\langle reg \rangle$. *XML2Async* obtains the signal name from $name$. The register write signal assignment comprises the logical OR of $lclk_i$, where s_i in $\langle reg \rangle$ is equal to 1. The connection method through the generation of control signals differs depending on whether there is a pipeline stall. *XML2Async* connects $ctrl_i$ to the DFFs to use DFFs for stall signals.

In addition, *XML2Async* generates multiplexer control signals ($glue_{mux_l}$) by referring to $\langle mux \rangle$; it obtains the signal name from $name$. The multiplexer control assign-

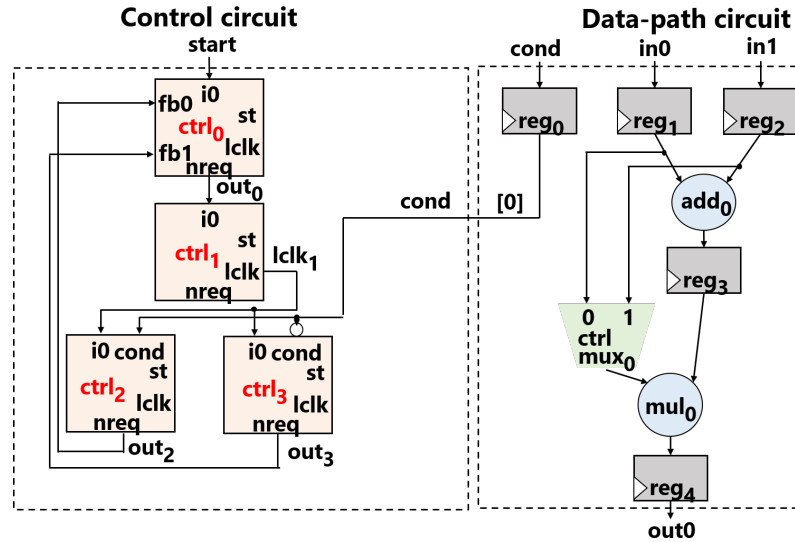


Figure 4.23: Connection from data-path resources to control modules.

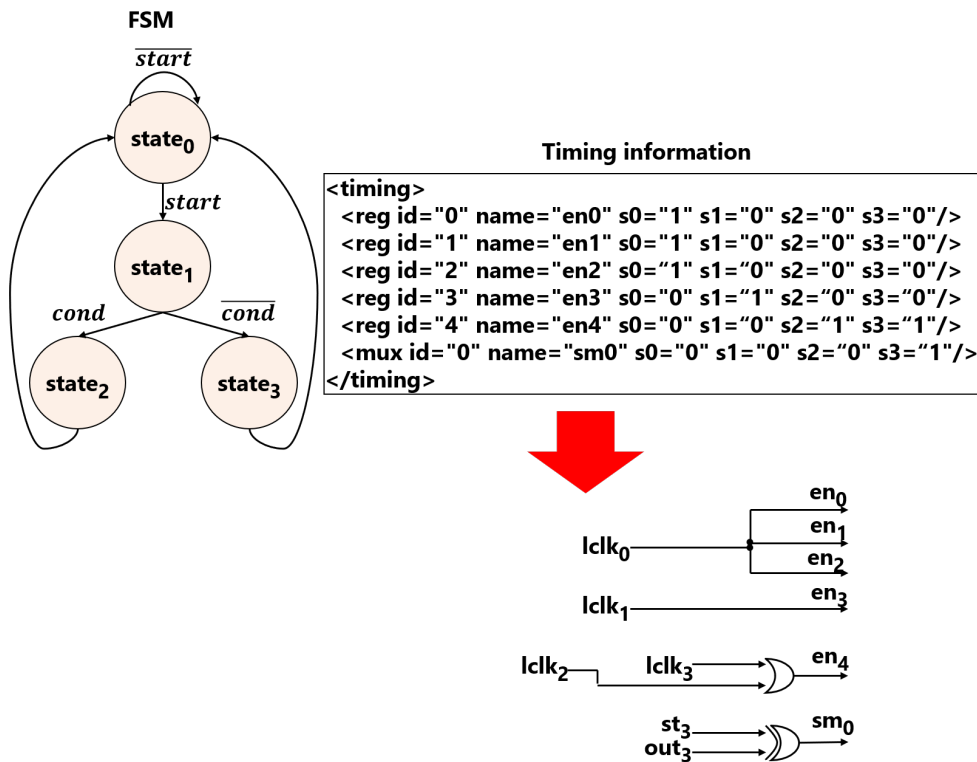
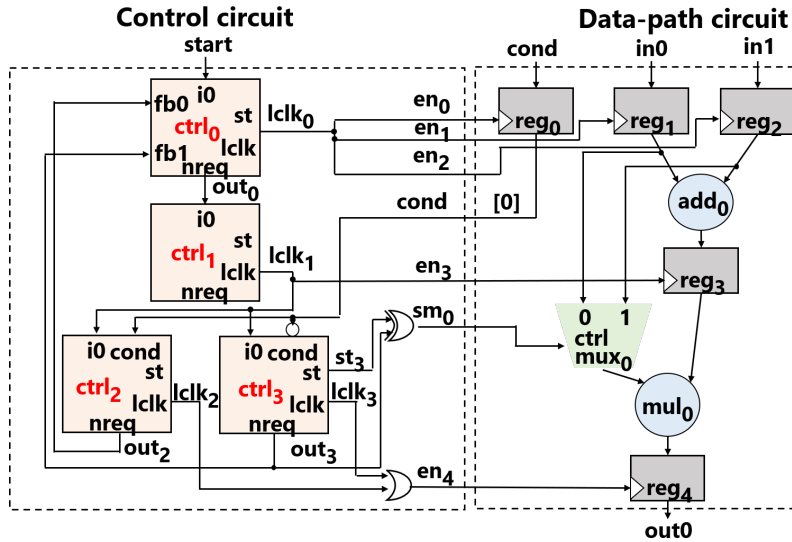


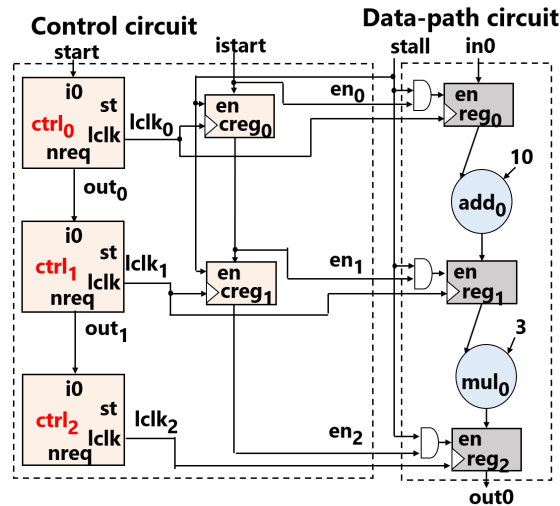
Figure 4.24: Generation of glue logics.

ment comprises the logical XOR of st_i , where s_i in $\langle mux \rangle$ is different from the value of the previous state s_{i-1} . If there are several XOR gates, *XML2Async* generates multiplexer control signals using the logical OR of these XOR gates. *XML2Async* initializes multiplexer control signals using the logical XOR of st_i and out_i when $ctrl_i$ generates feedback signals and s_i is 1.

Figure 4.24 shows the generation of register write signals and multiplexer control



(a) Asynchronous RTL model for Fig. 4.4.



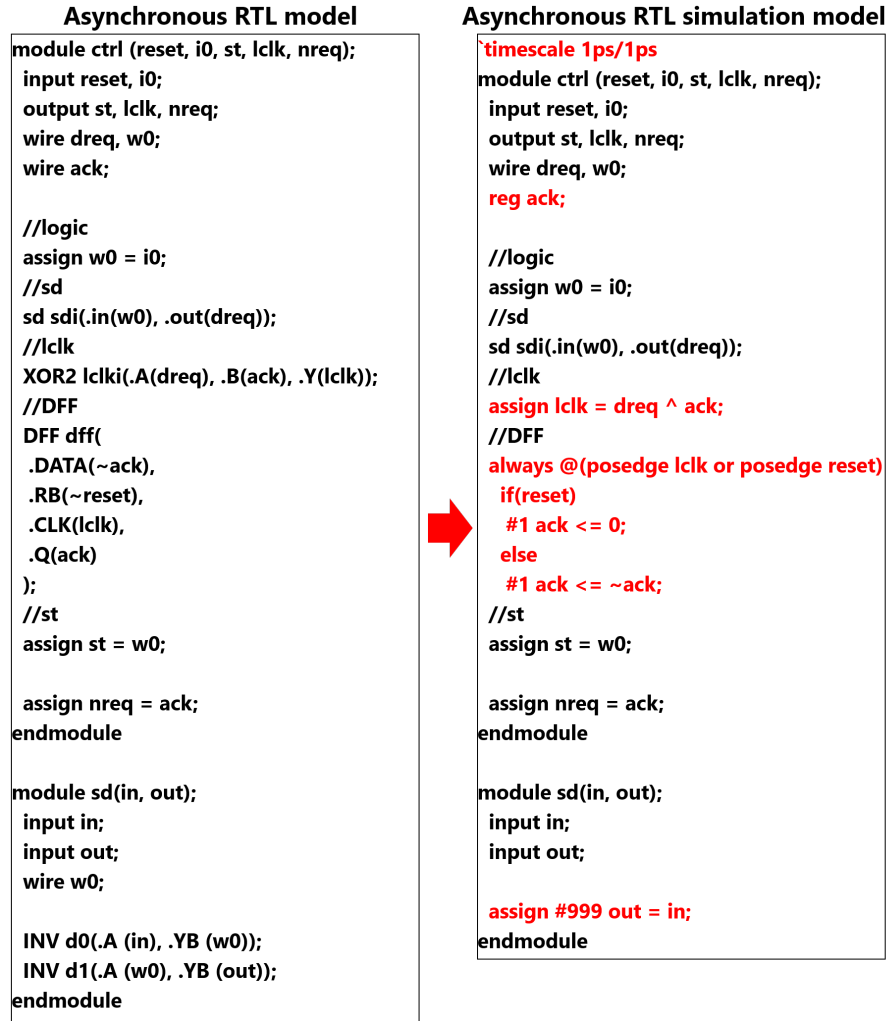
(b) Asynchronous RTL model for Fig. 4.5.

Figure 4.25: Generated top-level module.

signals. For generating the register write signals, the assignment of en_4 comprises a logical OR of $lclk_2$ and $lclk_3$ because the values of s_2 and s_3 in $\langle reg \rangle$ are 1. For generating the multiplexer control signals, the assignment of sm_0 comprises a logical XOR of st_3 and out_3 because the value of s_3 in $\langle mux \rangle$ is 1, and $ctrl_3$ generates the feedback signal. Figure 4.25a shows the converted asynchronous RTL model from the synchronous RTL model in Fig. 4.4. Figure 4.25b shows the converted asynchronous RTL model from the synchronous RTL model in Fig. 4.5.

4.4.2 Generation of Asynchronous RTL Simulation Models

XML2Async generates an asynchronous RTL simulation model when the target device is an FPGA. An asynchronous RTL simulation model is generated because designers cannot perform RTL simulation in an FPGA design environment using RTL

Figure 4.26: Asynchronous RTL simulation model for $ctrl_i$.

models with primitive cells. Therefore, *XML2Async* generates control modules that do not include primitive cells.

XML2Async replaces the primitive cells in the control modules with "always" or "assign" statements to generate the asynchronous RTL simulation model. In this dissertation, *XML2Async* implements DFFs using the "always" statement and inserts one time unit delay (i.e., #1). Further, *XML2Async* implements the logics except for the delay elements sd_i using the "assign" statement with the zero time unit. *XML2Async* obtains the time unit from $\langle timeunit \rangle$ in the Info-XML.

XML2Async implements delay elements sd_i for the simulation model using the "assign" statement. Moreover, *XML2Async* generates the delay $time_{sd_i}$ of sd_i using

$$time_{sd_i} = (CT - 1) \quad (4.1)$$

where CT represents the target cycle time. *XML2Async* obtains the value from $\langle cycle \rangle$ of the Info-XML.

Figure 6.4a shows the generated Verilog HDL of $ctrl_i$ for asynchronous RTL simulation models. *XML2Async* obtains the delay unit from the $\langle timeunit \rangle$ of the Info-XML in Fig. 4.6. Further, *XML2Async* obtains 1,000 ps as CT from $\langle cycle \rangle$ of the Info-XML


```

set_dont_touch [list ¥
[get_cells {ctrl/ctrl*/lclk*}] ¥
[get_cells {ctrl/ctrl*/dff*}] ¥
[get_cells {ctrl/ctrl*/sd*}] ¥
] true

```

Figure 4.27: Example of a set of non-optimization constraints for ASIC implementations.

in Fig. 4.6. Subsequently, *XML2Async* implements DFF_i using the "always" statement with 1 ps because DFF_i is a DFF. Finally, *XML2Async* implements sd_i using the "assign" statement with 999 by subtracting the delay of DFF_i (1 ps) from CT (1,000).

4.4.3 Generation of a Set of Non-Optimization Constraints

XML2Async generates non-optimization constraints for the control modules to prevent the optimization of logics in the control modules. In addition, *XML2Async* generates non-optimization constraints for delay elements to guarantee the correct timing to write the data into registers. Therefore, *XML2Async* generates non-optimization constraints for DFF_i ($bDFF_i$), $lclk_i$, and delay elements.

XML2Async generates `set_dont_touch` commands for ASICs. Further, *XML2Async* generates `DONT_TOUCH` commands for Xilinx FPGAs. Figure 4.27 is an example of the non-optimization constraints when the target implementation is an ASIC. For Intel FPGAs, *XML2Async* does not generate non-optimization constraints, because the control modules and delay elements are not optimized when we use Intel Quartus Prime 19.1.

Chapter 5

Optimization Methods

In this chapter, we describe optimization methods during RTL conversion. Section 5.1 describes the modularization for data-path resources for reducing the area of data-path circuits. Section 5.2 describes the use of appropriate DFFs to reduce the area of registers. Section 5.3 describes inserting latches before data-path resources to reduce the dynamic power consumption of data-path circuits. Section 5.4 describes the conversion from DFFs to D latches for reducing the dynamic power consumption of registers.

The quality of asynchronous circuits after RTL conversion depends on representation styles before the conversion because the proposed RTL conversion method does not perform optimization during the conversion. We propose optimization methods that can be applied during RTL conversion to obtain high-quality asynchronous circuits

In this dissertation, We propose four optimization methods during RTL conversion from synchronous RTL models to asynchronous RTL models with bundled-data implementation: (1) modularization for data-path resources to reduce the area of data-path circuits; (2) use of appropriate DFFs to reduce the area of registers; (3) inserting latches before data-path resources to reduce the dynamic power consumption of data-path circuits; and (4) conversion from DFFs to D latches to reduce the dynamic power consumption of registers.

Table 5.1 indicates the applicability of optimization methods. The modularization for data-path resources is suitable for non-pipelined circuits because we use different maximum delay constraints for each state based on latency constraints. In contrast, the modularization for data-path resources is not suitable for pipelined circuits because we use the same value for each local clock constraint based on clock constraints. The use of appropriate DFFs is suitable for all circuits if circuits have DFFs with an enable signal. The latch insertion is suitable for non-pipelined circuits and pipelined circuits whose Π is two cycles. However, the latch insertion is not suitable for pipelined circuits whose Π is one cycle because all pipeline stages in pipelined circuits whose Π is one cycle are operated simultaneously. The conversion of DFFs into D latches is suitable for all circuits.

5.1 Modularization for Data-Path Resources

Data-path resources are modularized to optimize the area of data-path circuits. Modularization implies that the proposed method converts operational RTL descriptions into structural RTL descriptions. The proposed method converts operations into functional

Table 5.1: Applicability of optimization methods.

Optimization methods	Non-pipelined circuits	Pipelined circuits where $\Pi=1$	Pipelined circuits where $\Pi=2$ or more
Modularization for data-path resources	Suitable	Not suitable	Not suitable
Use of appropriate DFFs	Suitable	Suitable	Suitable
Latch insertion as operand isolation	Suitable	Not suitable	Suitable
Conversion from DFFs to D latches	Suitable	Suitable	Suitable

```

set_max_delay ¥
-from <names> ¥
-through <names> ¥
-to <names> ¥
<value>

```

Figure 5.1: Example of the maximum delay constraints.

units and variables changed by conditions except the register variables into multiplexers.

In asynchronous circuits with bundled-data implementation, each state can have a different delay attributed to the use of the local handshake signals or self-timing. We assign maximum delay constraints for data-paths in asynchronous circuits with bundled-data implementation to obtain the same performance as synchronous circuits. We can reduce the area of data-path circuits by assigning a different delay for each state. For example, we assign a loose delay constraint for operations whose resource is a large area, whereas we assign a strict delay constraint for operations whose resource is a small area under a latency constraint. This can result in an optimum data-path circuit.

Figure 5.1 shows an example of the maximum delay constraints. In the maximum delay constraints, we can specify a start point, a through point, and an end point. The start point is an input pin or source register in the data-path. The through point is an input or output of functional units or multiplexers in the data-path. The end point is an output pin or destination register in the data-path.

In operational RTL models, we cannot assign appropriate maximum delay constraints because we cannot specify through points in the maximum delay constraints. There are data-paths whose source and destination are the same but the passed resources are different. Thus, we cannot distinguish the assignment of the maximum delay constraints for such data-paths.

Figure 5.2a shows a part of the Verilog HDL of an asynchronous RTL model without modularization, a DFG, a structure, and maximum delay constraints. The DFG shows that the operations in $state_0$ and $state_1$ are different. An adder is used in $state_0$, while a multiplier is used in $state_1$. Frequently, the delay of the adder is shorter than the delay of the multiplier. Here, if the data-path delay in $state_0$ is 600 ps, the latency is unchanged even if we assign 1,400 ps to $state_1$. However, we cannot distinguish the maximum delay constraints because the source and destination registers are the same (i.e., reg_0).

We can assign data-path resources as through points in the maximum delay constraints by modularizing data-path resources during RTL conversion. This results in the

assignment of a different delay for data-paths that are activated in different states even if the source and destination registers are the same.

The proposed method modularizes the data-path resources during *XML2Async* to specify the through points in the maximum delay constraints. The proposed method generates modules for data-path resources by referring to the resource information. Subsequently, it instantiates data-path resources by using the corresponding module.

Figure 5.2b shows a part of the Verilog HDL of an asynchronous RTL model with modularization, a DFG, a structure, and maximum delay constraints. Instead of operations, actual resources are instantiated by the modularization. The operational RTL descriptions for *add_out*, *mul_out*, and *mux₁_out* in Fig. 5.2a(a) are converted into structural RTL descriptions using modules *add₀*, *mul₀*, and *mux₁* in Fig. 5.2b, which results in the assignment of a different delay for the data-paths.

5.2 Use of Appropriate DFFs

Appropriate DFFs are used to optimize the area of registers. In asynchronous circuits with bundled-data implementation, data are written to registers using a *lclk_i* signal from *ctrl_i*. If data are not written to registers in *state_i*, there is no connection from *lclk_i* to the registers, which enables the registers to use DFFs without an enable signal. Therefore, we can reduce the area of registers because a DFF without an enable signal is generally smaller than a DFF with an enable signal.

However, in some scenarios, enable signals are generated by data-path resources. Here, we cannot remove the enable signals, which is particularly remarkable in the synchronous RTL models synthesized by an HLS tool with a clock gating option.

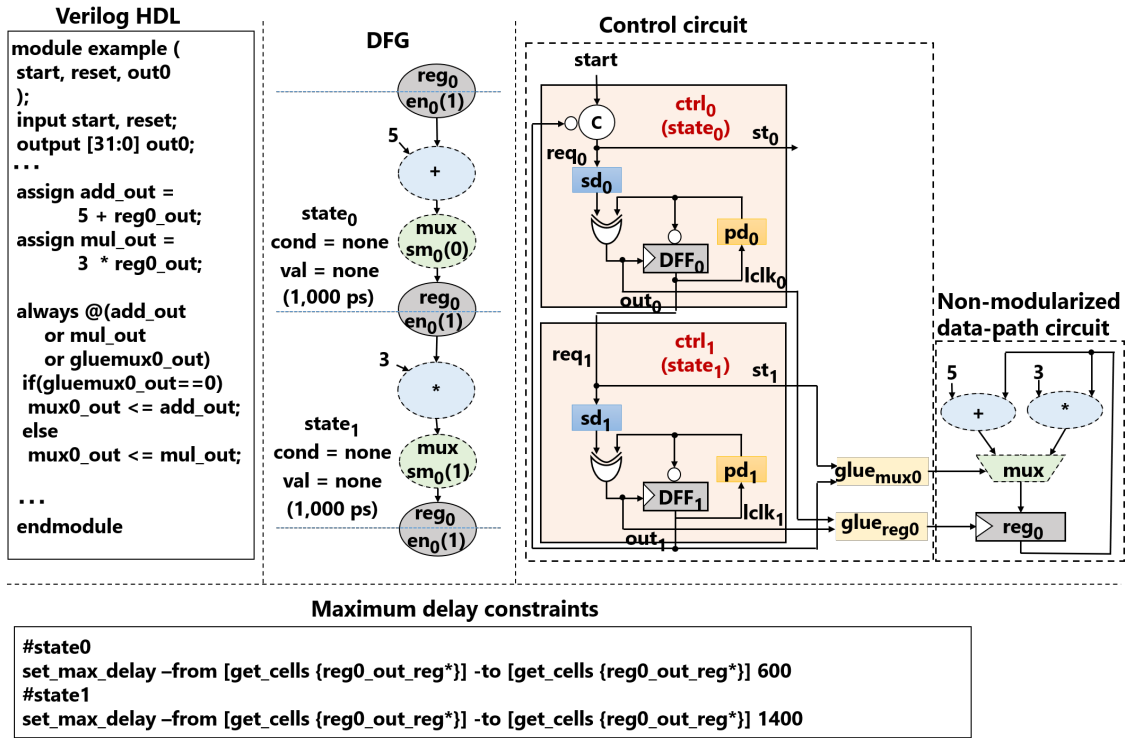
During *XML2Async*, the proposed method moves the assignment for the enable signal to the outside of the register descriptions to avoid the use of DFFs with an enable signal. This results in the insertion of a glue logic (e.g., AND operation) for the registers that consists of *lclk_i* and the enable signal. However, this optimization is valid if the area of DFFs without an enable signal and the glue logics is smaller than the area of DFFs with an enable signal.

This optimization is similar to clock gating in synchronous circuits. The proposed method uses this optimization approach to minimize reduce the circuit area of the asynchronous circuits. A register with an enable signal is used during logic synthesis for converted asynchronous RTL models when register descriptions in synchronous circuits include a condition for clock gating. This is because we do not assign a clock gating option for the synthesis of the converted asynchronous RTL models. Therefore, the proposed method moves the condition to the outside of the register descriptions and generates a register write signal using the condition signal with *lclk_i* or the output of the glue logic for the register; this results in the use of DFFs without an enable signal.

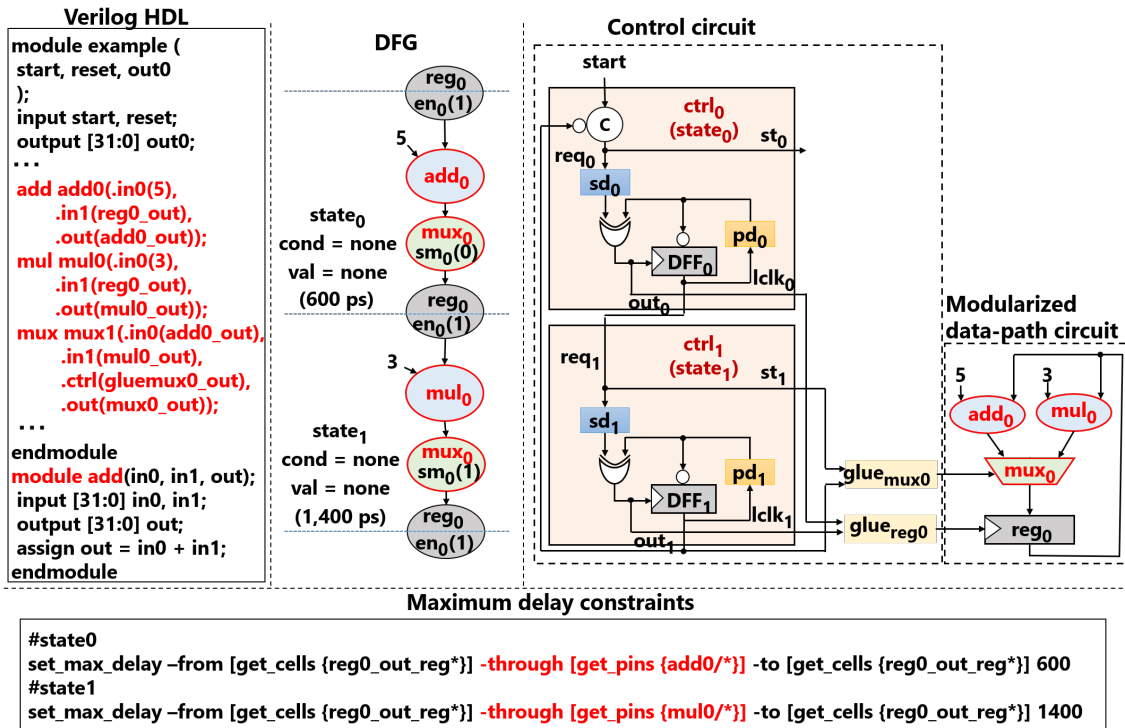
The proposed method moves assignments to the outside of the registers when the following inequality is satisfied.

$$area_{enableDFF} - area_{DFF} > threshold \quad (5.1)$$

where $area_{enableDFF}$, $area_{DFF}$, and $threshold$ represent the circuit area for DFFs with an enable signal, circuit area for DFFs without an enable signal, and a threshold value, respectively. The glue logics for generating the enable signals are included in $area_{DFF}$. Since we do not know the exact area of the glue logics after logic synthesis, we estimate



(a) Non-modularized data-path resources.



(b) Modularized data-path resources.

Figure 5.2: Modularization of data-path resources.

$area_{enableDFF}$ and $area_{DFF}$ by referring to the used technology library, number of literals, and logic operations in the assignments for the enable signals. We estimate the area of the register that has the smallest bit width using DFFs with an enable signal and

DFFs without an enable signal in the used technology library to decide the threshold value. Next, we calculate the difference between the area using DFFs with an enable signal and the area using DFFs without an enable signal. Finally, we set a threshold value that is smaller than the obtained difference because in RTL, we do not know used DFFs and inserted gate cells.

The proposed method estimates $area_{enableDFF}$ and $area_{DFF}$ by referring to area parameters and resource information in the Model-XML. The proposed method calculates $area_{enableDFF}$ by multiplying the area of DFFs with an enable signal in the area parameters and the bit width for the register in the resource information. In addition, the proposed method multiplies the area of DFFs without an enable signal in the area parameters and the bit width for the register in the resource information. Thereafter, the proposed method calculates $area_{DFF}$ by adding the multiplied value and the area of the glue logic.

Figure 5.3 shows an example of the optimization. Figure 5.3a shows the given area parameters and the resource information. We specify the required values in the parameters to use appropriate DFFs. The proposed method calculates $area_{enableDFF}$ and $area_{DFF}$ by referring to the parameters in Fig. 5.3a, bit width of the registers in the resource information, and number of logics and literals in the assignments of the enable signals. In this example, DFFs without an enable signal are used because the difference between $area_{enableDFF}$ and $area_{DFF}$ is more than that of the threshold value (Fig. 5.3b and Fig. 5.3c). The proposed method moves the condition signal ($reg0_out[0]$) for the assignment of reg_2 to outside of reg_2 with the "assign" statement. Thereafter, the proposed method inserts a new logic that comprises the logical AND of $reg0_out[0]$ and $glureg_2_out$ for generating a register write signal (en_2) for reg_2 .

5.3 Latch Insertion as Operand Isolation

The aim of using latch insertion as an operand isolation is to reduce the dynamic power consumption of data-path circuits. Figure 5.4 shows the examples of operand isolation.

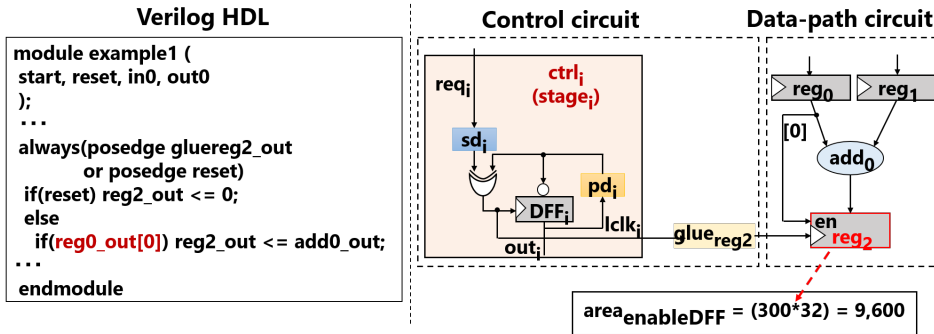
In general operand isolation, the logic synthesis tools insert AND gates before functional units for isolating the inputs when the operations are not required at the functional units (Fig. 5.4a). However, data-path circuits waste power because a change in the AND gates to 0 is propagated to the functional units.

The proposed method prevents dynamic power consumption by using latches (Fig. 5.4b). Compared with AND gates, the latches do not propagate unnecessary signal transitions to the functional units. An operand isolation method [34] using AND gates, latches, etc., is available for synchronous circuits. For such synchronous circuits, their latency may be degraded because of the increase in the cycle time caused by the latch insertion that affects all cycles. In addition, the number of cycles may be increased when the inserted latches are controlled by clock signals. In the proposed method, the number of cycles is not changed by inserting latches because of the use of the signals in $ctrl_i$. Even if the latches are inserted and the critical path delay is increased, the effect for the performance is localized to the inserted cycle.

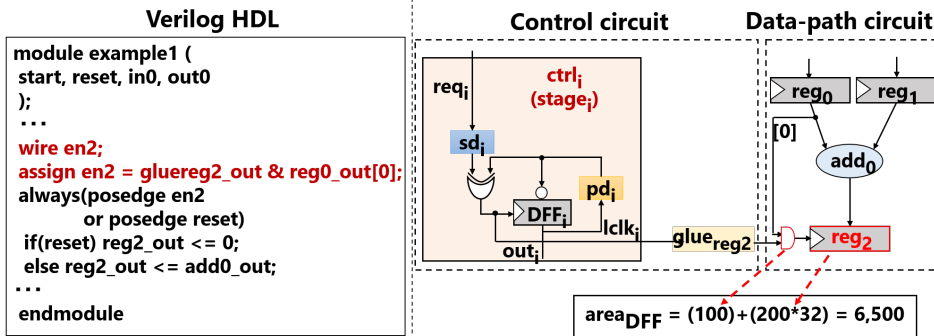
The registers are separated into master-slave latches (Fig. 5.4c). The slave latch is used as an isolator as in the proposed method. However, the timing verification will be complicated because registers with the role of an isolator are based on the master-slave

Area parameter	Resource information
<pre> <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> </pre>	<pre> <resource_info> <resource id="0" name="reg0" bit="32" type="reg" ctrl="en0;1" substitution="nb"/> <resource id="1" name="reg1" bit="32" type="reg" ctrl="en1;1" substitution="nb"/> <resource id="2" name="reg2" bit="32" type="reg" ctrl="en2;1" substitution="nb"/> <resource id="3" name="add0" bit="32" type="add" ctrl="" substitution="a"/> <resource id="4" name="en2" bit="1" type="and" ctrl="" substitution="a"/> </resource_info> </pre>

(a) Area parameter and resource information.



(b) Registers with an enable signal.



(c) Registers without an enable signal.

Figure 5.3: Area estimation for reg_2 .

latches and other registers are based on DFFs. However, we can distinguish between the registers and isolators easily because the proposed method simply inserts latches when the isolations are required.

An additional register is inserted based on DFFs as an isolator (Fig. 5.4d). Latches are considered better than DFFs in terms of circuit area and dynamic power consumption to prevent unnecessary signal transitions. Therefore, the proposed method uses latches as isolators.

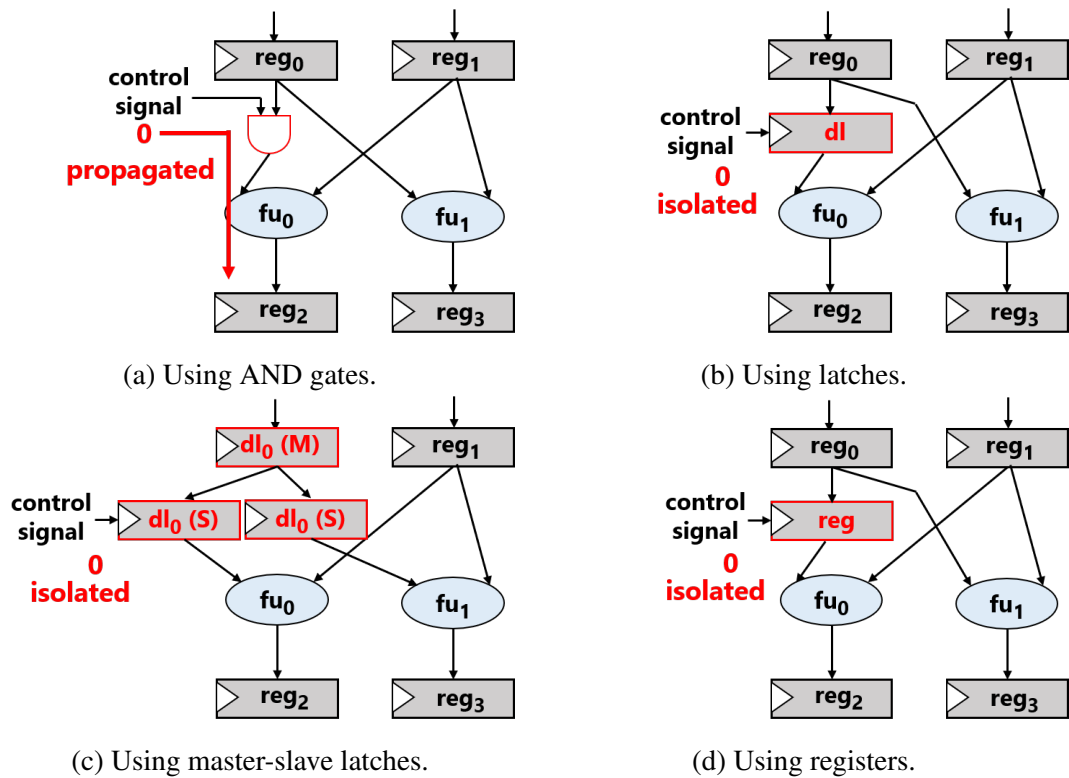


Figure 5.4: Example of operand isolations.

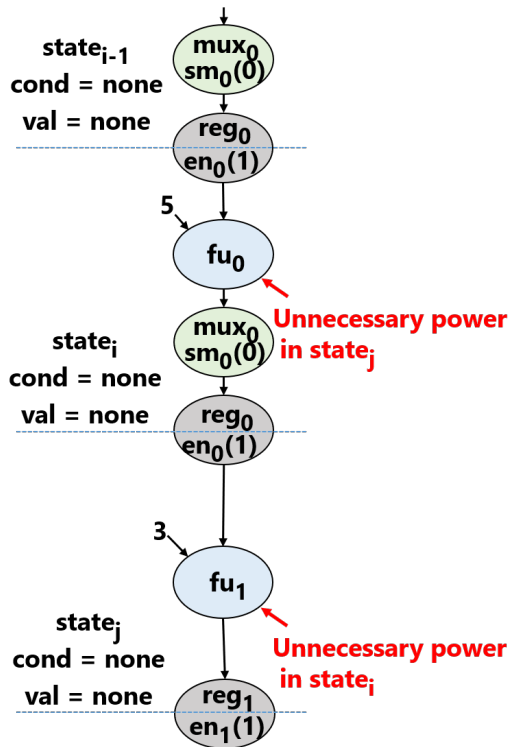


Figure 5.5: Example of unnecessary power consumption for functional units.

The power of the functional units may be wasted when the value of a register is used by several functional units in different states. Figure 5.5 shows an example of the

add
max : 500.0
sub
max : 500.0
mul
max : 1100.0
reg
max : 150.0
mux
max : 100.0
latch
max : 120.0
.
.
.

Figure 5.6: Delay parameter for each data-path resource.

unnecessary power consumption of functional units. We assume that fu_1 operates in $state_i$ and fu_2 operates in $state_j$. Here, both fu_1 and fu_2 operate because the value of the register is propagated to fu_1 and fu_2 when the value of the register that is the source of fu_1 and fu_2 is changed. In each state, one operation is valid while the other operation is invalid, which means that the latter scenario only wastes power. The proposed method recognizes such a relationship by referring to the data-paths in a DFG.

The performance may be degraded by the delay of the inserted latches when the proposed method inserts latches in critical paths. The proposed method insert latches for data-paths that are not critical paths to preserve the critical path delays. Further, the proposed method estimates the delay of data-paths to determine whether the paths are critical paths. For a DFG, the proposed method can expect which data-path resources are used in each data-path. Therefore, we estimate the delays of the data-paths by summing the delays of the expected data-path resources.

Moreover, the reduction effect of dynamic power consumption will be small because the inserted latches are few when the proposed method inserts latches in only non-critical paths. The proposed method increases the number of data-paths that latches can be inserted into by summing the critical path delay and a margin for reducing more power.

5.3.1 Latch Insertion Algorithm

Algorithm 1 represents the proposed latch insertion algorithm for non-pipelined circuits. This algorithm accepts a DFG extracted from a CDFG.

First, we define terminologies used in the proposed latch insertion algorithm: $dp_{i,p}$ represents the p -th data-path in a state $state_i$ or a pipeline stage $stage_i$. A source $src_{dp_{i,p}}$ in $dp_{i,p}$ represents an input signal or a register. Moreover, a sink in $dp_{i,p}$ represents an output signal or a register. $dp_{i,p}$ also includes multiplexers $mux_{dp_{i,p}}$ and functional units $fu_{dp_{i,p}}$ between the source and the sink. $t_{dp_{i,p}}$ represents the path delay of $dp_{i,p}$, t_{state_i} represents the critical path delay in $state_i$, t_{cp} represents the critical path delay in the pipelined circuit, $margin_{state_i}$ represents a margin of the critical path delay in $state_i$, and t_{dl_d} represents the delay of the D latch of the delay parameters. In the delay parameters shown in Fig. 5.6, max represents the maximum delay of the data-path

Algorithm 1 Latch Insertion Algorithm for Non-Pipelined Circuits**Input:** DFG and delay parameters**Output:** DFG including dl_d

```

1: foreach  $state_i$  do
2:   foreach  $dp_{i,p}$  do
3:     estimate  $t_{dp_{i,p}}$ 
4:   end foreach
5:   estimate  $t_{state_i}$ 
6: end foreach
7: foreach  $state_i$  do
8:   foreach  $dp_{i,p}$  do
9:     foreach  $state_j$  do
10:      foreach  $dp_{j,m}$  do
11:        if ( $src_{dp_{i,p}} == src_{dp_{j,m}}$ ) then
12:          if ( $(t_{dp_{j,m}} + t_{dl_d})$  is smaller than  $(t_{state_j} + margin_{state_j})$ ) then
13:            insert  $dl_d$  between  $src_{dp_{j,m}}$  and  $fu_{dp_{j,m}}$  ( $mux_{dp_{j,m}}$ )
14:          end if
15:        end if
16:      end foreach
17:    end foreach
18:  end foreach
19: end foreach

```

resources.

We prepare delay parameters as shown in Fig. 5.6 to estimate path delays. In Fig. 5.6, max represents the maximum delay of the data-path resources. In this method, delay parameters are prepared as follows: (1) we perform logic synthesis for RTL models including registers, functional units, and multiplexers using a clock constraint, and we explore the fastest clock cycle time without timing violations. (2) We obtain the delays of the registers, functional units, and multiplexers after logic synthesis. (3) Finally, we define the delays as max for the data-path resources.

For each $dp_{i,p}$, the proposed latch insertion algorithm estimates $t_{dp_{i,p}}$ by adding the max of each data-path resource in $dp_{i,p}$. Then, the proposed latch insertion algorithm estimates t_{state_i} in $state_i$; t_{state_i} represents the maximum delay in $t_{dp_{i,p}}$. After the estimation of all critical path delays, the algorithm explores all $dp_{j,m}$ in $state_j$. For each $dp_{j,m}$ in $state_j$, the algorithm inserts a D latch dl_d between $src_{dp_{j,m}}$ and $fu_{dp_{j,m}}$ if $src_{dp_{j,m}}$ is the same as $src_{dp_{i,p}}$ and the delay of $t_{dp_{j,m}}$ and t_{dl_d} is smaller than the total delay of t_{state_j} and $margin_{state_j}$. If there is a multiplexer $mux_{dp_{j,m}}$ before $fu_{dp_{j,m}}$, dl_d is inserted before $mux_{dp_{j,m}}$.

Figure 5.7 shows an example of latch insertion for non-pipelined circuits. The number represented by "()" is the estimated delay of the corresponding data-path. In this example, the margin is 0. The critical path delay t_{state_1} in $state_1$ is 1,350 and the critical path delay t_{state_2} in $state_2$ is 650. Although $src_{dp_{2,0}}$ is the same as $src_{dp_{1,0}}$ (i.e., reg_0), the algorithm does not insert dl_d between reg_0 and sub_0 in $state_2$ because the insertion of the latch results in an increase in the critical path delay in $state_2$. Similarly, the algorithm does not insert dl_d between reg_2 and mul_0 , and dl_d between reg_2 and sub_0 ; only dl_0 is inserted between reg_0 and add_0 in this case.

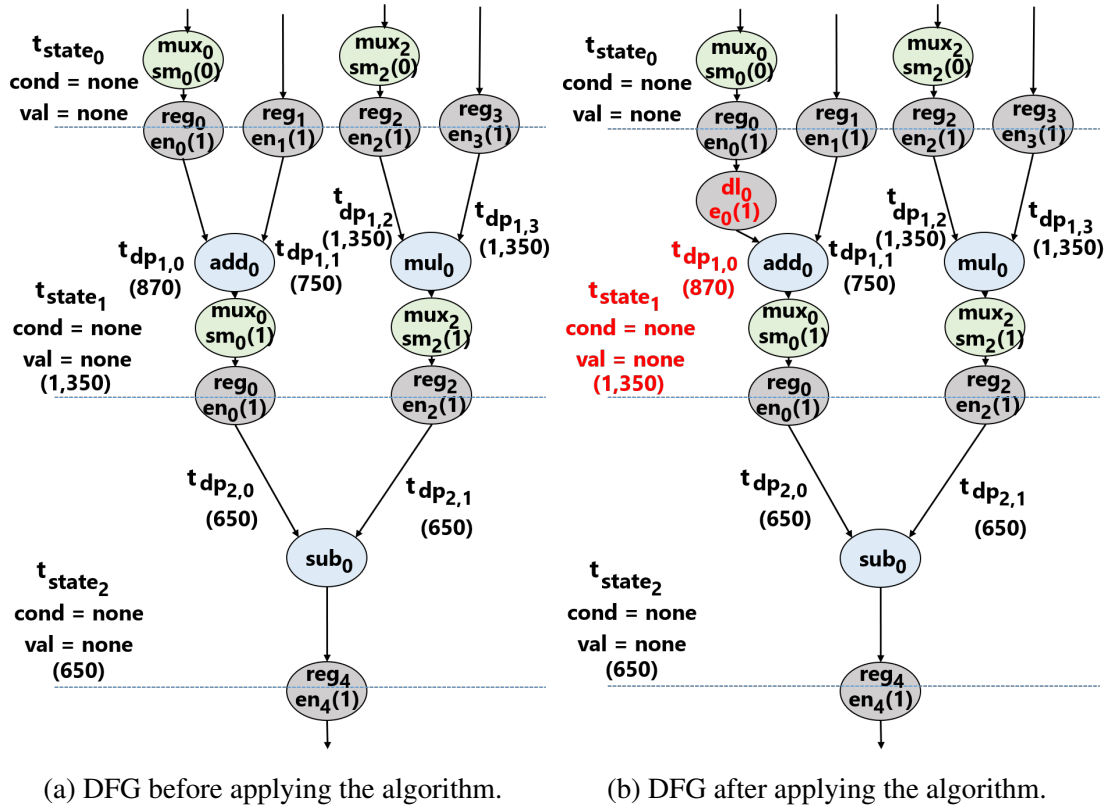


Figure 5.7: Latch insertion for non-pipeined circuits.

Algorithm 2 Latch Insertion Algorithm for Pipelined Circuits**Input:** DFG and delay parameters**Output:** DFG including dl_d

```

1: foreach  $stage_i$  do
2:   foreach  $dp_{i,p}$  do
3:     estimate  $t_{dp_{i,p}}$ 
4:   end foreach
5:   estimate  $t_{stage_i}$ 
6: end foreach
7: estimate  $t_{cp}$ 
8: foreach  $stage_i$  do
9:   foreach  $dp_{i,p}$  do
10:    foreach  $stage_j$  do
11:     foreach  $dp_{j,m}$  do
12:      if ( $src_{dp_{i,p}} == src_{dp_{j,m}}$ ) then
13:        if ( $(t_{dp_{j,m}} + t_{dl_d})$  is smaller than  $(t_{cp} + margin_{stage_j})$ ) then
14:          insert  $dl_d$  between  $src_{dp_{j,m}}$  and  $fu_{dp_{j,m}}(mux_{dp_{j,m}})$ 
15:        end if
16:      end if
17:    end foreach
18:  end foreach
19: end foreach
20: end foreach

```

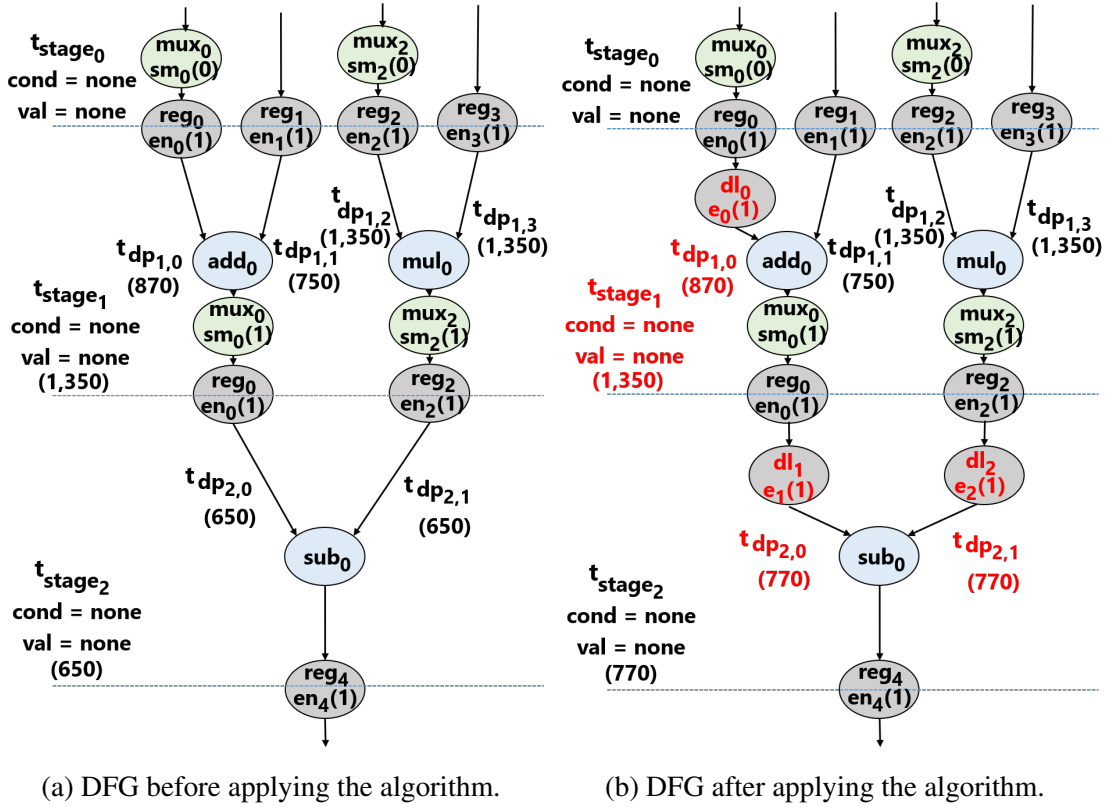


Figure 5.8: Latch insertion for pipelined circuits.

In contrast, algorithm 2 presents the proposed latch insertion algorithm for the pipelined circuits. In pipelined circuits, the performance of asynchronous circuits depends on the cycle time. Therefore, the proposed method defines the maximum delay in the critical path delays of all stages as the critical path delays t_{cp} .

For each $dp_{i,p}$, the algorithm for pipelined circuits estimates $t_{dp_{i,p}}$ by adding the max of each data-path resource in $dp_{i,p}$. Thereafter, it estimates t_{stage_i} in $stage_i$. t_{stage_i} represents the maximum delay in $t_{dp_{i,p}}$. After the estimation of all critical path delays, the algorithm defines the critical path delays t_{cp} ; then, it explores all $dp_{j,m}$ in $stage_j$. For each $dp_{j,m}$ in $stage_j$, we insert a D latch dl_d between $src_{dp_{j,m}}$ and $fu_{dp_{j,m}}$ if $src_{dp_{j,m}}$ is the same as $src_{dp_{i,p}}$ and the delay of $t_{dp_{j,m}}$ and t_{dl_d} is smaller than the total delay of t_{cp} and $margin_{stage_j}$.

Figure 5.8 shows an example of the latch insertion for pipelined circuits. In this example, the margin is 0, and the critical path delay t_{cp} is 1,350. Although $src_{dp_{1,2}}$ is the same as $src_{dp_{2,1}}$ (i.e., reg_2), the algorithm does not insert dl_d between reg_2 and mul_0 in $stage_1$ because the insertion of the latch results in an increase in the critical path delay t_{cp} . Therefore, dl_0 , dl_1 , and dl_2 are inserted in this scenario.

5.3.2 Latch Control Signal

The inserted D latches of $stage_i$ are controlled by a local state signal lst_i in $ctrl_i$ (Fig. 5.9). lst_i is implemented by an XOR operation using st_i and out_i ; if the inserted latches are controlled by several $ctrl_i$, they can be controlled by lst_i through an OR operation ($glue_{dl_d}$).

Figure 5.10 shows the timing diagram of $ctrl_i$. A local state signal lst_i opens the

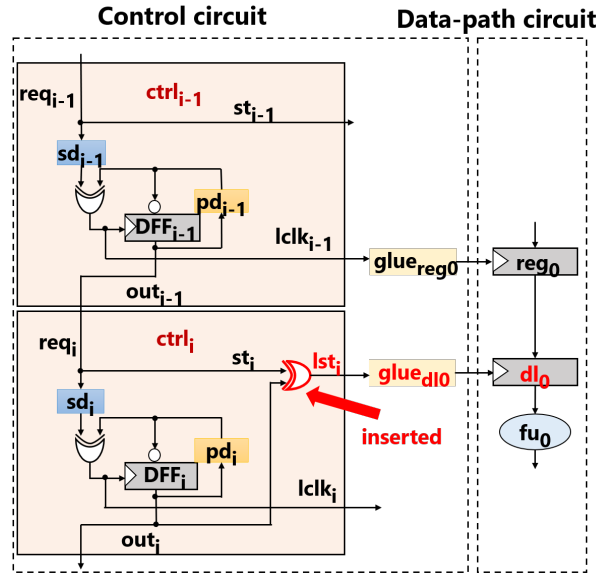
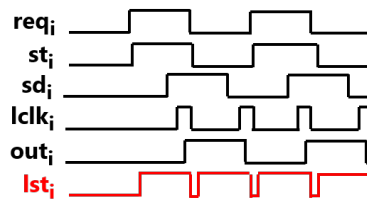


Figure 5.9: Latch controller.


 Figure 5.10: Timing diagram for l_{st_i} .

inserted latches after the rising transition of st_i arrives at the XOR gate. Subsequently, l_{st_i} closes the inserted latches after the rising transition of out_i arrives at the XOR gate. The behavior of l_{st_i} for the falling transition of st_i and out_i is the same as that for the rising transition of st_i and out_i .

5.3.3 Timing Constraints

In latch insertion, it is necessary to satisfy the setup and hold constraints to operate the circuit correctly. If the timing constraints are violated, unnecessary signal transitions are propagated to the functional units through the inserted D latches.

Setup Constraints

A setup constraint implies that the input data for the D latch must be arrived at the D latch until the D latch is closed. Figure 5.11 shows a data-path $sdp_{i,p}$ and control-path $scp_{i,p}$ related to the setup constraint. $sdp_{i,p}$ (red line) represents a path from the output of $lclk_{i-1}$ to the destination D latch dl_0 through the source register reg_0 . $scp_{i,p}$ (blue line) represents a path from the output of $lclk_{i-1}$ to the destination D latch dl_0 through l_{st_i} . We define the maximum delay of $sdp_{i,p}$ as $t_{maxsdp_{i,p}}$, minimum delay of $scp_{i,p}$ as $t_{minscp_{i,p}}$, and margin for $t_{maxsdp_{i,p}}$ as $t_{sdpm_{i,p}}$. Thus, the setup constraint can

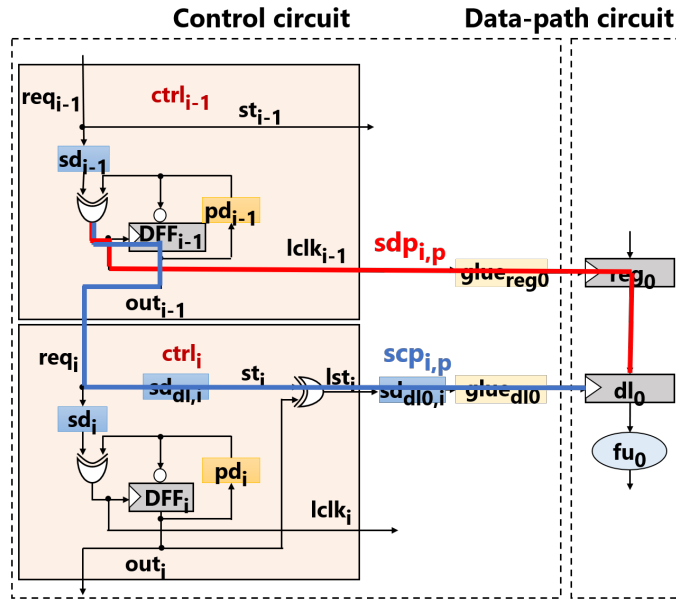


Figure 5.11: Paths related to setup constraint.

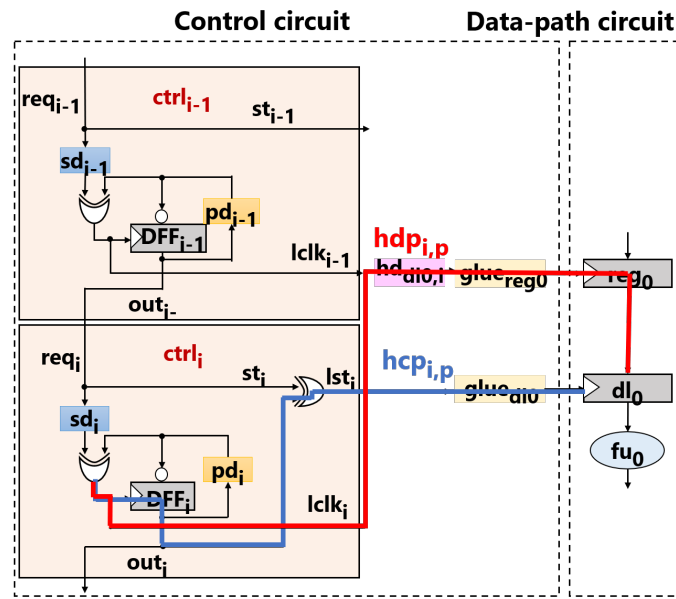


Figure 5.12: Paths related to hold constraint.

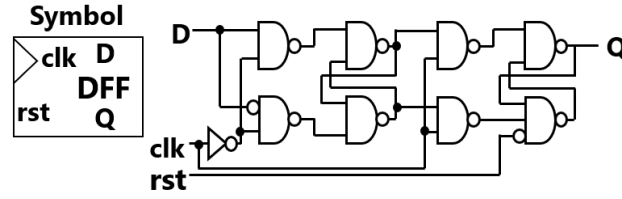
be represented by

$$t_{minsc_{i,p}} > t_{maxsd_{i,p}} + t_{sdpm_{i,p}} \quad (5.2)$$

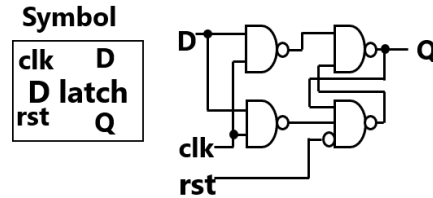
If the setup constraint is violated, we can adjust the number of cells for $sd_{dl,i}$ or $sd_{dl,d,i}$.

Hold Constraints

A hold constraint implies that D latches must be closed until the next input data arrives at the D latches after input data are written to the D latches. Figure 5.12 shows a data-path $hdp_{i,p}$ and control-path $hcp_{i,p}$ related to the hold constraint. $hdp_{i,p}$ (red line)



(a) Register.



(b) D latch.

Figure 5.13: Examples of the structures of registers and D latches.

represents a path from the output of $lclk_i$ to the destination D latch dl_0 through the source register reg_0 . $hcp_{i,p}$ (blue line) represents a path from the output of $lclk_i$ to the destination D latch dl_0 through lst_i . We define the minimum delay of $hdp_{i,p}$ as $t_{minhdp_{i,p}}$, maximum delay of $hcp_{i,p}$ as $t_{maxhcp_{i,p}}$, margin for $t_{maxhcp_{i,p}}$ as $t_{hcpm_{i,p}}$, and hold time of the destination D latch as $t_{hold_{i,p}}$. Thus, the hold constraint can be represented by

$$t_{minhdp_{i,p}} > t_{maxhcp_{i,p}} + t_{hcpm_{i,p}} + t_{hold_{i,p}} \quad (5.3)$$

If the hold constraint is violated, we can adjust the number of cells for $hd_{dl_d,i}$.

5.4 Conversion from DFFs into D latches

The aim of the conversion from DFFs into D latches is to optimize the dynamic power consumption of registers. Figure 5.13 shows the examples of the structures of registers and D latches. In general, D latches have low power and a small area compared with DFFs. Therefore, the proposed method converts DFFs into D latches during *XML2Async*.

5.4.1 Timing Constraints

It is necessary to satisfy the setup and hold constraints to operate the circuit correctly. We describe the timing constraints.

Setup Constraints

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 5.14 shows a data-path $sdp_{i,p}$ and control-path $scp_{i,p}$ related to the setup constraint. There are two types of data-paths $sdp_{i,p}$ (red line): from the output of $lclk_{i-1}$ to the destination D latch dl_1 through the source D latch dl_0

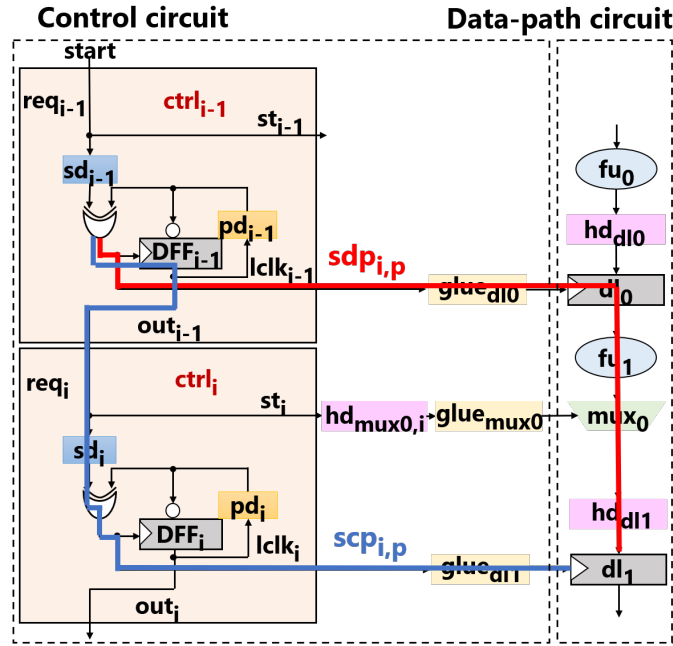
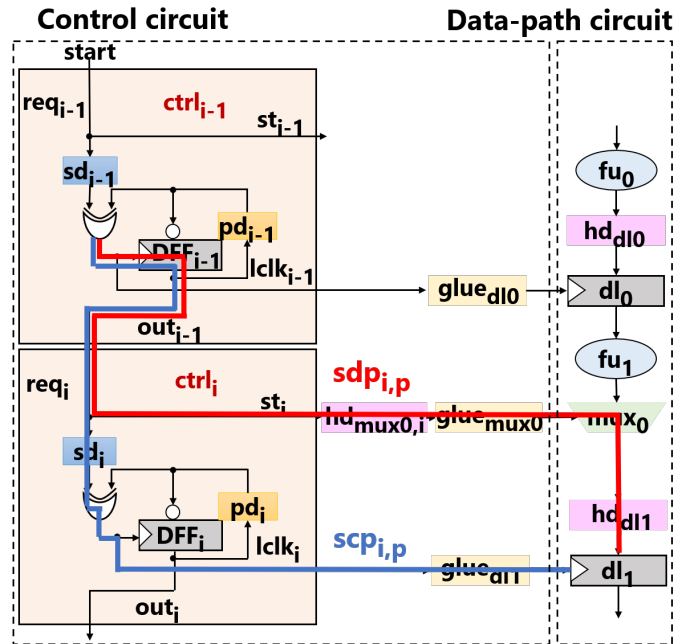

 (a) Data-path $sd_{i,p}$ through a source register and control-path $scp_{i,p}$.

 (b) Data-path $sd_{i,p}$ through $glue_{mux_1}$ and control-path $scp_{i,p}$.

Figure 5.14: Paths related to setup constraints for D latches.

(Fig. 5.14a) and from the output of $lclk_{i-1}$ to the destination D latch dl_1 through the glue logic $glue_{mux_0}$ (Fig. 5.14b). $scp_{i,p}$ (blue line) represents a path from the output of $lclk_{i-1}$ to the D latch dl_1 through the delay element sd_i . We define the maximum delay of $sd_{i,p}$ as $t_{maxsd_{i,p}}$, minimum delay of $scp_{i,p}$ as $t_{minscp_{i,p}}$, and margin for $t_{maxsd_{i,p}}$ as $t_{sdpm_{i,p}}$. Thus, the setup constraint can be represented by

$$t_{minscp_{i,p}} > t_{maxsd_{i,p}} + t_{sdpm_{i,p}} \quad (5.4)$$

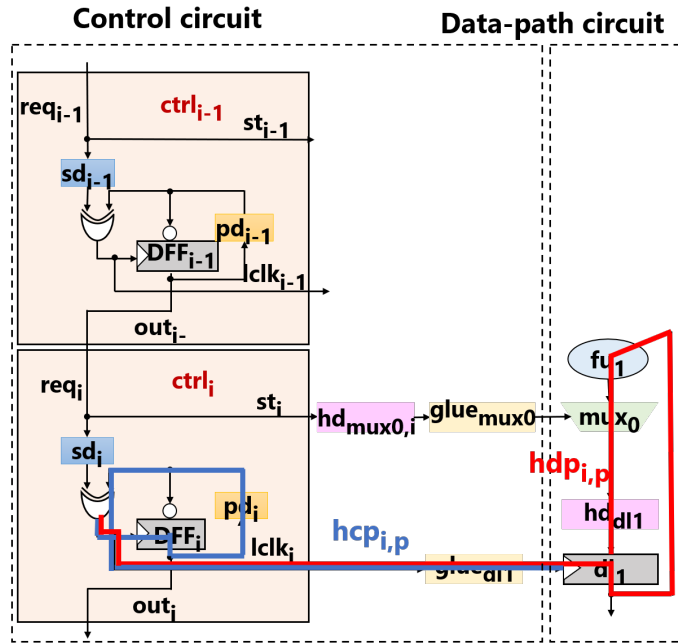
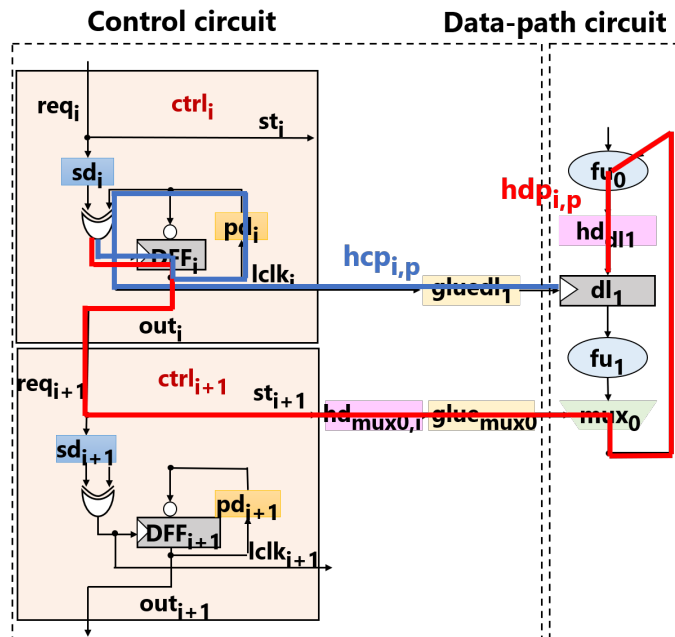

 (a) Data-path $hdp_{i,p}$ through a source register and control-path $hcp_{i,p}$.

 (b) Data-path $hdp_{i,p}$ through $glue_{mux1}$ and control-path $hcp_{i,p}$.

Figure 5.15: Paths related to hold constraints for non-pipelined circuits.

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

Hold Constraints

D latches 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 . The hold constraint used in this dissertation differs between the non-pipelined circuits and the pipelined circuits.

Figure 5.15 shows a data-path $hdp_{i,p}$ and control-path $hcp_{i,p}$ related to the hold

constraint for the non-pipelined circuits. There are two types of data-paths $hdp_{i,p}$ (red line): from the output of $lclk_i$ to the destination D latch dl_1 through the source D latch dl_1 (Fig. 5.15a) and from the output of $lclk_i$ to the destination D latch dl_1 through the glue logic $glue_{mux_0}$ (Fig. 5.15b). Further, $hcp_{i,p}$ (blue line) represents a path from the output of $lclk_i$ to the destination D latch dl_1 through $DFFi$. We define the minimum delay of $hdp_{i,p}$ as $t_{minhdp_{i,p}}$, maximum delay of $hcp_{i,p}$ as $t_{maxhcp_{i,p}}$, margin for $t_{maxhcp_{i,p}}$ as $t_{hcpm_{i,p}}$, and hold time of the destination D latch as $t_{hold_{i,p}}$. Thus, the hold constraint can be represented by

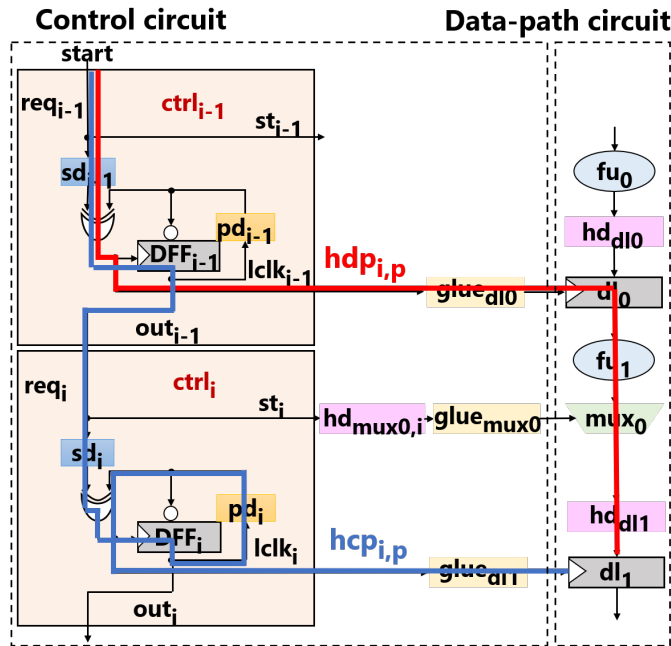
$$t_{mincdp_{i,p}} > t_{maxccp_{i,p}} + t_{hcpm_{i,p}} + t_{hold_{i,p}} \quad (5.5)$$

If the hold constraint is violated, we should adjust the number of cells for $hddl_k$ of $hdmux_{i,l}$.

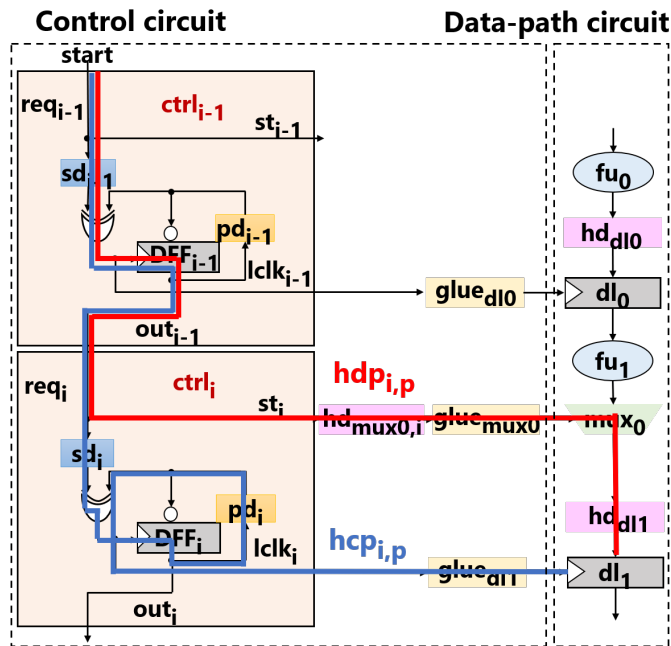
Figure 5.16 shows a data-path $hdp_{i,p}$ and control-path $hcp_{i,p}$ related to the hold constraint for the pipelined circuits. There are two types of data-paths $hdp_{i,p}$ (red line): from the input signal $start$ to the destination D latch dl_1 through the source D latch dl_1 (Fig. 5.16a) and from the input signal $start$ to the destination D latch dl_1 through the glue logic $glue_{mux_0}$ (Fig. 5.16b). Here, $hcp_{i,p}$ (blue line) is a path from the input signal $start$ to the destination D latch dl_1 through $DFFi$. We define the minimum delay of $hdp_{i,p}$ as $t_{minhdp_{i,p}}$, maximum delay of $hcp_{i,p}$ as $t_{maxhcp_{i,p}}$, margin for $t_{maxhcp_{i,p}}$ as $t_{hcpm_{i,p}}$, hold time of the destination D latch as $t_{hold_{i,p}}$, global cycle time as gct , and input interval as II . Thus, the hold constraint can be represented by

$$t_{minhdp_{i,p}} + gct \times II > t_{maxhcp_{i,p}} + t_{hcpm_{i,p}} + t_{hold_{i,p}} \quad (5.6)$$

If the hold constraint is violated, we should adjust the number of cells for $hddl_k$ or $hdmux_{i,l}$ or increase gct .



(a) Data-path $hdp_{i,p}$ through a source register and control-path $hcp_{i,p}$.



(b) Data-path $hdp_{i,p}$ through $glue_{mux1}$ and control-path $hcp_{i,p}$.

Figure 5.16: Paths related to hold constraints for pipelined circuits.

Chapter 6

Experimental Results

In this chapter, we describe the experimental results. Section 6.1 describes the preparation for the experiments. Section 6.2 describes the conversion results which indicate that the proposed method can generate asynchronous RTL models from various synchronous RTL models. Section 6.3 presents the evaluation of the converted asynchronous circuits in terms of the circuit area, execution time, dynamic power consumption, and energy consumption.

6.1 Preparation

For the experiments, we implemented a conversion tool for the proposed method using Java. The conversion tool was executed on a Windows 10 machine (Intel Core i7-8700 @3.2 GHz CPU and 16 GB memory).

We prepared synchronous RTL models to demonstrate that the proposed method can generate asynchronous RTL models from synchronous RTL models. "Manual" represents synchronous RTL models designed manually. "HLS" represent synchronous RTL models obtained by synthesizing SystemC models using Cadence Stratus HLS 18.1. The synchronous RTL models for non-pipelined circuits were the DIFFerential EQUation solver (DIFFEQ), Elliptic Wave Filter (EWF), and Inverse Discrete Cosine Transform (IDCT). The synchronous RTL models for pipelined circuits were DIFFEQ, EWF, MultiLayer Perceptron (MLP) [35], for which the number of neurons was 32, Advanced Encryption Standard (AES) [36], and LENET [37], which includes only the second convolutional and pooling layers. Figure 6.1 shows the architecture of LENET synthesized using Stratus HLS. In this research, LENET can receive image data in parallel. Further, we prepared synchronous RTL models whose II was one cycle and two cycles to demonstrate that the proposed method can use synchronous RTL models with different IIs. Moreover, we prepared synchronous RTL models with a hard stall (Hard) and a soft stall (Soft) by applying directives [38] in Stratus HLS. Hard indicates that the operations of all pipeline stages stall, whereas Soft implies that the operations of the specified pipeline stages stall. In addition, we applied the clock gating option to HLS for synchronous RTL models. The library was the eShuttle 65 nm process technology.

As a reference, we synthesized the synchronous circuits (*sync*) from synchronous RTL models using Cadence Genus 18.1 with the eShuttle 65 nm technology library. We explored the fastest synchronous circuits without timing violations. In non-pipelined circuits, the clock cycle times of DIFFEQ, EWF, and IDCT were 1,600, 1,600, and 1,700 ps. In the pipelined circuits, the clock cycle times of DIFFEQ, EWF, MLP, AES,

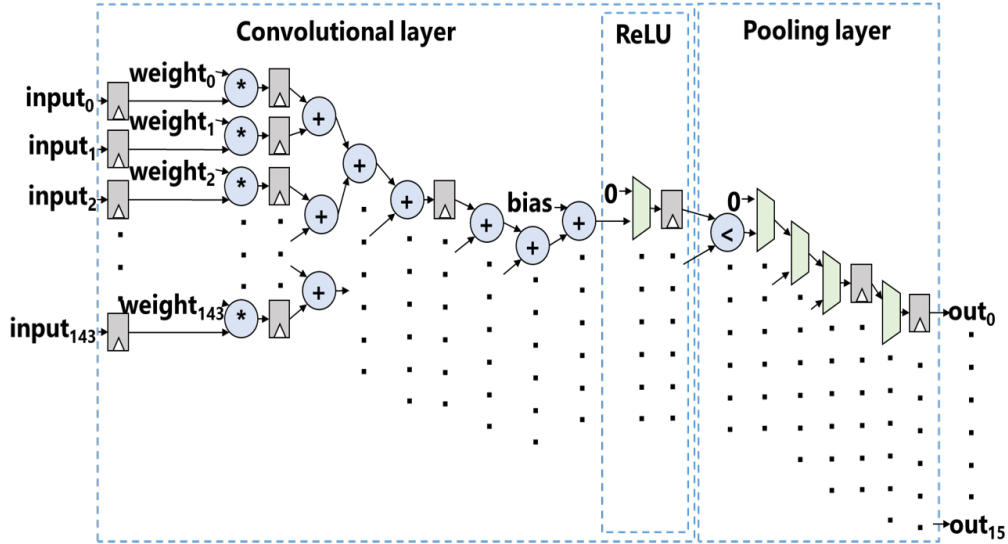


Figure 6.1: Architecture of LENET used in this research.

and LENET were 1,400, 1,500, 600, 900, and 2,000 ps, respectively, when the II was one cycle. The clock cycle times of DIFFEQ, EWF, MLP, and AES were 1,400, 1,500, 700, and 900 ps, respectively, when the II was two cycles. Further, we applied the clock gating option to logic synthesis for synchronous circuits.

We generated GL netlists from the asynchronous RTL models based on the design method in [26] to evaluate the quality of the converted asynchronous RTL models. Figure 6.3a shows the logic design flow after RTL conversion. We synthesized the asynchronous RTL models with maximum delay constraints for each path and local clock constraints for $lclk_i$ to obtain the same performance as the synchronous circuits. We referred to [26] to generate these constraints.

We obtained eight asynchronous RTL models from a synchronous RTL model to evaluate each optimization method.

- RTL_{async} - asynchronous circuits without optimization
- RTL_{async_m} - with modularization of data-path resources
- RTL_{async_r} - with appropriate DFFs
- $RTL_{async_{op}}$ and $RTL_{async_{op10}}$ - with latch insertion (0% margin and 10% margins)
- RTL_{async_l} - with D latches instead of DFFs
 - $RTL_{async_{l2(l4)}}$ - with strict clock constraints (200 ps (400 ps) smaller than the clock cycle time of the synchronous circuits)
- RTL_{async_a} - with combination of RTL_{async_m} , $RTL_{async_{op10}}$, and RTL_{async_l}

RTL_{async_a} does not include $async_r$ because the used library does not include a D latch with an enable signal. Since pipelined circuits assume that all pipeline stages operate at the same delay, we did not design RTL_{async_m} . We did not design $RTL_{async_{op}}$

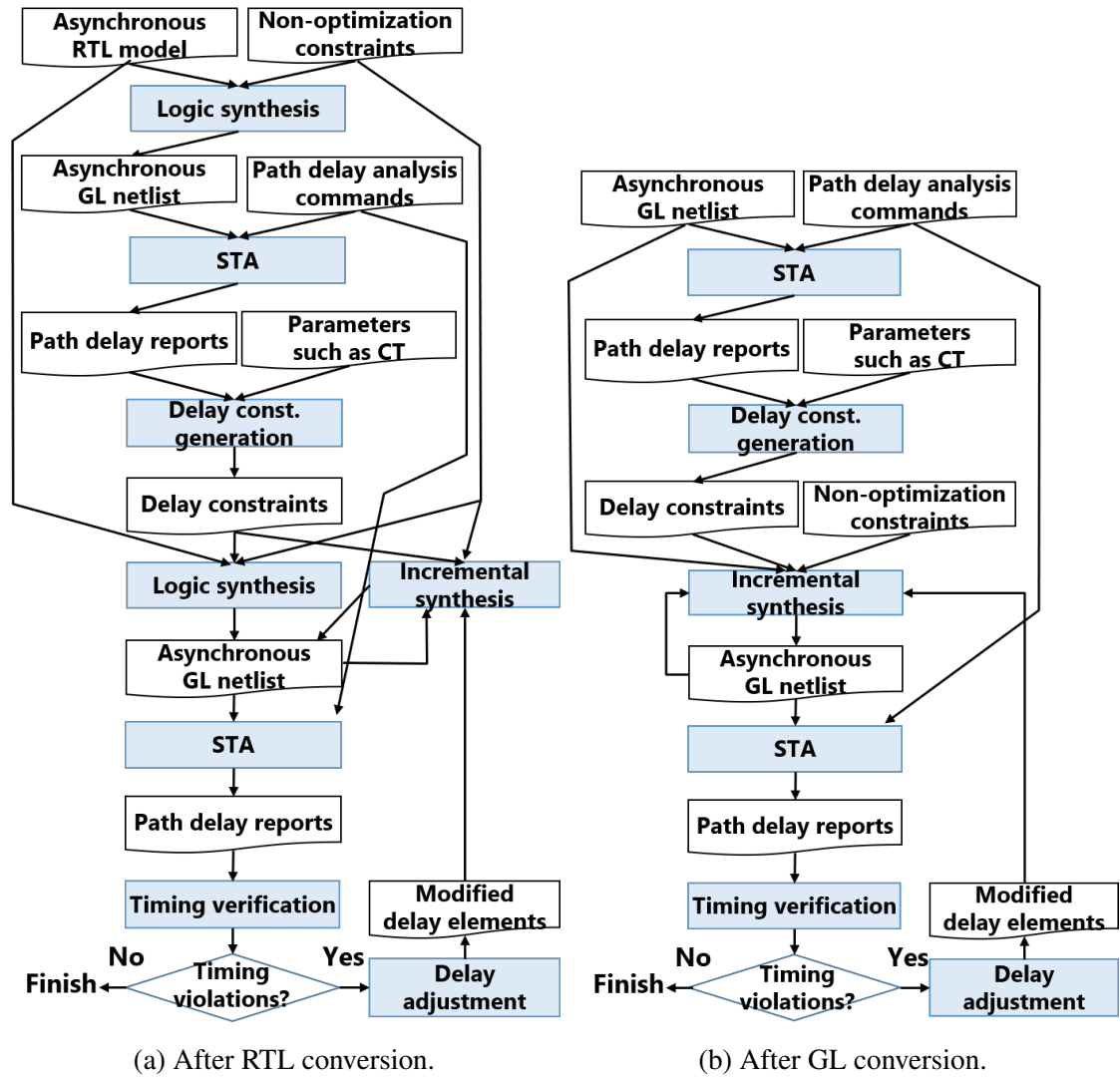


Figure 6.2: Logic design flow.

because all pipeline stages in pipelined circuits whose II is one cycle are operated simultaneously.

In addition to RTL conversion, we manually performed GL conversion for the comparison between RTL and GL conversion. We inserted control modules $ctrl_i$ for the synchronous GL netlist synthesized with a clock constraint. Thereafter, we replaced the clock signal of the registers to the output signal of the glue logics $glue_{reg_k}$. We converted DFFs to D latches. Figure 6.3b shows the logic design flow after GL conversion. We assigned the maximum delay constraints and local clock constraints generated by the same generation method used in the RTL conversion. The synthesized asynchronous GL netlists GL_{async_l} satisfied timing constraints for bundled-data implementations. We could not design GL_{async_m} and $GL_{async_{op}}$ because it was difficult to apply optimization methods to the asynchronous GL netlists. In the asynchronous GL netlists, the wire names were changed and resources were replaced with gates.

The used commercial EDA tools were Cadence Stratus HLS 18.1 for high-level synthesis, Cadence Genus 18.1 for logic synthesis, Synopsys VCS Q-2020.03-SP1 for logic simulation, and Synopsys PrimeTime Q-2019.12-SP3 for static timing analysis (STA) and power analysis.

6.2 Conversion Results

Table 6.1 summarizes the conversion results obtained using the proposed method. *Type*, *CT*, *State*, *Stage*, and *Sverilog* represent the type of stalls or IIs, clock cycle time, number of states, number of pipeline stages, and number of lines in Verilog HDL of synchronous RTL models, respectively. *AST*, *Model-XML*, *Averilog*, and *Time* represent the number of lines in the AST, number of lines in the Model-XML, number of lines in Verilog HDL of asynchronous RTL models, and conversion time, respectively.

Table 6.1 indicates that 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 state or pipeline stage. In addition, the conversion time was increased when the pipeline stalls were included because the number of lines in the AST with the pipeline stalls was more than the number of lines in the AST without the pipeline stalls. When II is changed from one cycle to two cycles, the conversion time was increased because the shared data-path resources were analyzed multiple times. Moreover, the conversion time was increased when optimization is performed during RTL conversion.

The conversion time increased when circuits have more Verilog HDL lines and states/stages. For example, the conversion time is large in the case of control intensive circuits such as AES. In contrast, the conversion time of data intensive circuits such as LENET is short even if the circuit area is large.

For all benchmark circuits, we performed a logic simulation to verify the functional correctness of the converted asynchronous RTL models. Figure 6.3 shows the flows for the functional verification after RTL conversion and after logic synthesis. For the simulation, we generated a standard delay format (SDF) file by synthesizing the asynchronous RTL models. We prepared test patterns for the simulation by using arbitrary values. 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. Figure 6.4 shows the waveform of EWF for the non-pipelined circuits. The waveforms indicate that all output values of the asynchronous RTL models (asynchronous GL netlists) were the same as the output values of the synchronous RTL models (synchronous GL netlists).

6.3 Evaluation after Logic Synthesis

6.3.1 Comparison of *sync* and *RTLasync*

Figure 6.5 shows the circuit areas of *sync* and *RTLasync*. The circuit areas were obtained from the report file generated by Genus. *RTLasync* reduced the circuit area in the half of circuits. The proposed method is useful for larger circuits such as non-pipelined IDCT and pipelined AES and LENET because it uses the loose maximum delay constraints (non-pipeilned IDCT) or local clock constraints (pipelined AES and LENET). Even for increased circuits, the area overhead was very small between 0.1% and 2.3%.

Figure 6.6 shows the execution times of *sync* and *RTLasync*. The execution times were obtained by simulating the designed circuits with an arbitrary test sequence using VCS. The difference between the execution times of *sync* and *RTLasync* was very

Table 6.1: RTL conversion results.

Name	Type	Opt	CT [ps]	State or Stage	Sverilog [lines]	AST [lines]	Model-XML [lines]	Averilog [lines]	Time [s]
DIFFEQ (non-pipeline) (HLS)	None	None	1,600	6	347	945	180	804	2.4
		m	1,600	6	347	945	180	837	2.5
		r	1,600	6	347	945	180	810	2.4
		op	1,600	6	347	945	180	804	2.5
		op10	1,600	6	347	945	180	833	2.5
		l	1,600	6	347	945	180	873	2.4
EWF (non-pipeline) (HLS)	None	a	1,600	6	347	945	180	470	2.5
		None	1,600	10	620	1,721	453	1,435	2.9
		m	1,600	10	620	1,721	453	1,485	3.3
		r	1,600	10	620	1,721	453	1,451	3.0
		op	1,600	10	620	1,721	453	1,495	3.1
		op10	1,600	10	620	1,721	453	1,610	3.1
IDCT (non-pipeline) (Manual)	None	l	1,600	10	620	1,721	453	1,435	3.0
		a	1,600	10	620	1,721	453	1,660	3.4
		None	1,700	22	968	15,965	1,510	3,597	1,226.0
		m	1,700	22	968	15,965	1,510	3,238	1,226.4
		r	1,700	22	968	15,965	1,510	3,631	1,225.9
		op	1,700	22	968	15,965	1,510	3,626	1,226.4
DIFFEQ (pipeline) (HLS)	II=1	op10	1,700	22	968	15,965	1,510	3,626	1,226.3
		l	1,700	22	968	15,965	1,510	3,597	1,225.8
		a	1,700	22	968	15,965	1,510	3,267	1,226.7
	II=2	None	1,200	4	254	765	110	348	2.0
		l	1,200	4	254	765	110	348	2.0
		None	1,200	4	232	662	99	474	2.0
op		1,200	4	232	662	99	562	2.0	
op10		1,200	4	232	662	99	576	2.1	
l		1,200	4	232	662	99	474	2.1	
EWF (pipeline) (HLS)	Hard	a	1,200	4	232	662	99	576	2.1
		None	1,200	4	341	859	188	384	2.1
		None	1,200	4	331	922	177	410	2.3
	Soft	None	1,200	9	1,060	3,202	429	1,337	2.7
		l	1,200	9	1,060	3,202	429	1,337	2.7
		None	1,200	9	1,075	2,738	372	1,691	2.8
II=2	op	1,200	9	1,075	2,738	372	1,898	3.1	
	op10	1,200	9	1,075	2,738	372	2,013	3.1	
	l	1,200	9	1,075	2,738	372	1,691	2.9	
	a	1,200	9	1,075	2,738	372	2,013	3.1	
	Hard	None	1,200	9	1,439	3,588	746	1,428	2.8
	Soft	None	1,200	9	1,333	3,703	715	1,557	2.8
MLP (pipeline) (HLS)	II=1	None	400	20	24,977	94,130	22,276	36,609	330.4
		l	400	20	24,977	94,130	22,276	36,609	332.9
	II=2	None	400	20	25,325	93,363	22,856	57,992	627.8
		op	400	20	25,325	93,363	22,856	60,533	1,143.3
		op10	400	20	25,325	93,363	22,856	63,066	1,166.4
		l	400	20	25,325	93,363	22,856	57,992	623.6
		a	400	20	25,325	93,363	22,856	63,066	1,161.9
	Hard	None	400	20	32,676	101,164	28,039	36,464	366.0
	Soft	None	400	20	28,894	99,661	27,974	36,618	379.5
AES (pipeline) (HLS)	II=1	None	600	41	321,434	946,154	192,782	138,913	3,379.9
		l	600	41	321,434	946,154	192,782	138,913	3,546.8
	II=2	None	600	41	320,402	936,948	191,462	142,123	22,479.0
		op	600	41	320,402	936,948	191,462	142,630	30,513.8
		op10	600	41	320,402	936,948	191,462	143,242	29,943.6
		l	600	41	320,402	936,948	191,462	142,123	22,460.0
		a	600	41	320,402	936,948	191,462	143,242	29,848.4
	Hard	None	600	41	328,075	955,070	198,594	139,638	3,687.6
	Soft	None	600	41	325,299	954,490	198,474	139,960	3,823.8
LENET (pipeline) (HLS)	II=1	None	2,000	6	28,956	132,009	42,584	52,152	821.1
		l	2,000	6	28,956	132,009	42,584	52,152	823.7

small between -0.3% and $+1.4\%$ because asynchronous circuits were designed with latency or cycle time constraints obtained by clock constraints for synchronous circuits.

Figure 6.7 shows the dynamic power consumptions of *sync* and *RTLasync*. The

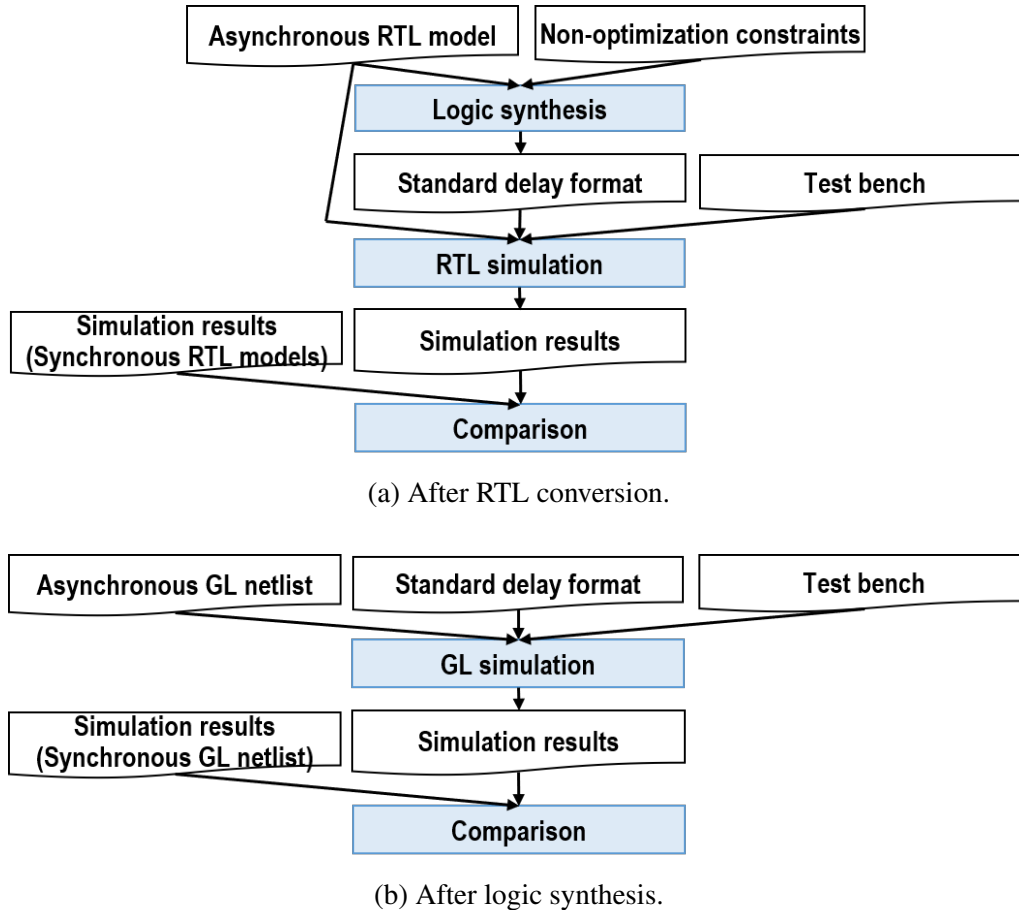


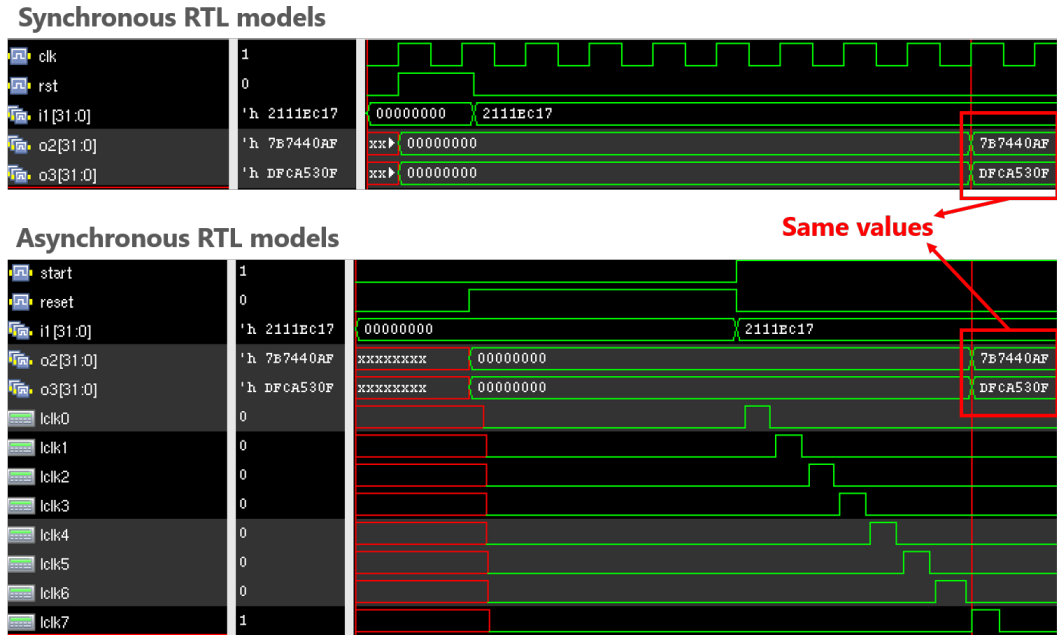
Figure 6.3: Functional verification flow.

dynamic power consumption was obtained by PrimeTime with the VCD file generated by VCS. *RTL_{async}* reduced the dynamic power consumption in all circuits, except for DIFFEQ where II was one cycle because of the use of the loose maximum delay constraints for non-pipelined circuits and local clock constraints for the pipelined circuits. In particular, *RTL_{async}* demonstrated a large reduction where combinational circuits are large, such as pipelined MLP, AES, and LENET.

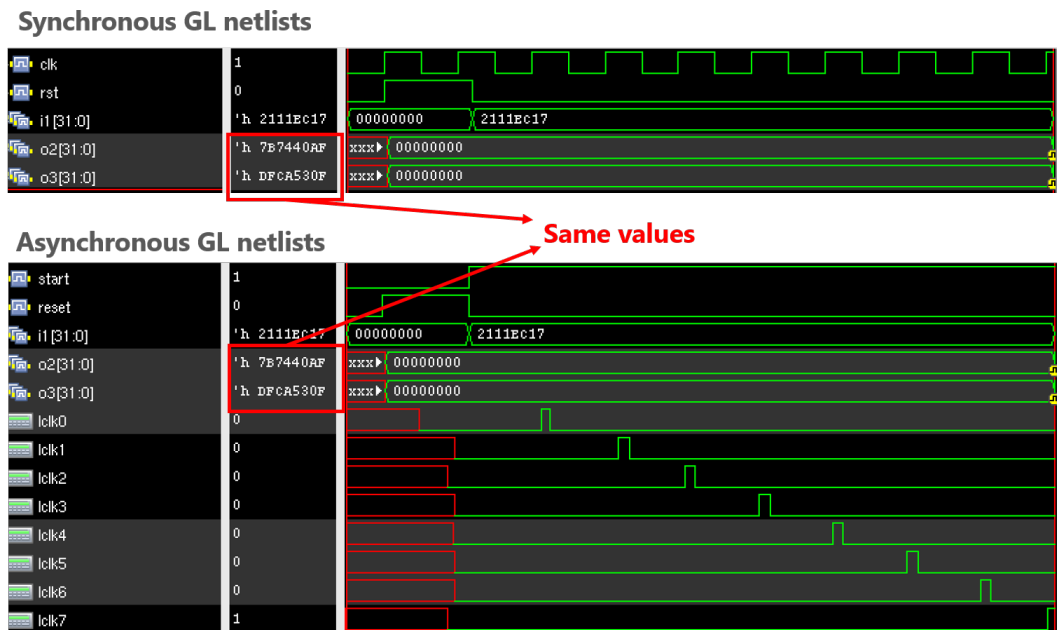
Figure 6.8 shows the energy consumptions of *sync* and *RTL_{async}*. The energy consumption was obtained by multiplying the execution time and dynamic power consumption. *RTL_{async}* reduced the energy consumption in all circuits except for DIFFEQ where II was one cycle; this result emanated from the direct effect of the reduction in the dynamic power consumption because the difference in the execution time was very small.

6.3.2 Evaluation of Proposed Optimization Methods

Figure 6.9 shows the evaluation results of the modularization for the data-path resources. Figures 6.9a-d show the circuit area, execution time, dynamic power consumption, and energy consumption, respectively. Compared with *RTL_{async}*, the circuit area and dynamic power consumption of *RTL_{async_m}* were reduced because of the use of the loose maximum delay constraints with a through point for operations that use high-power resources (e.g., multipliers). The difference between the execution



(a) Waveforms of synchronous RTL model and asynchronous RTL model.

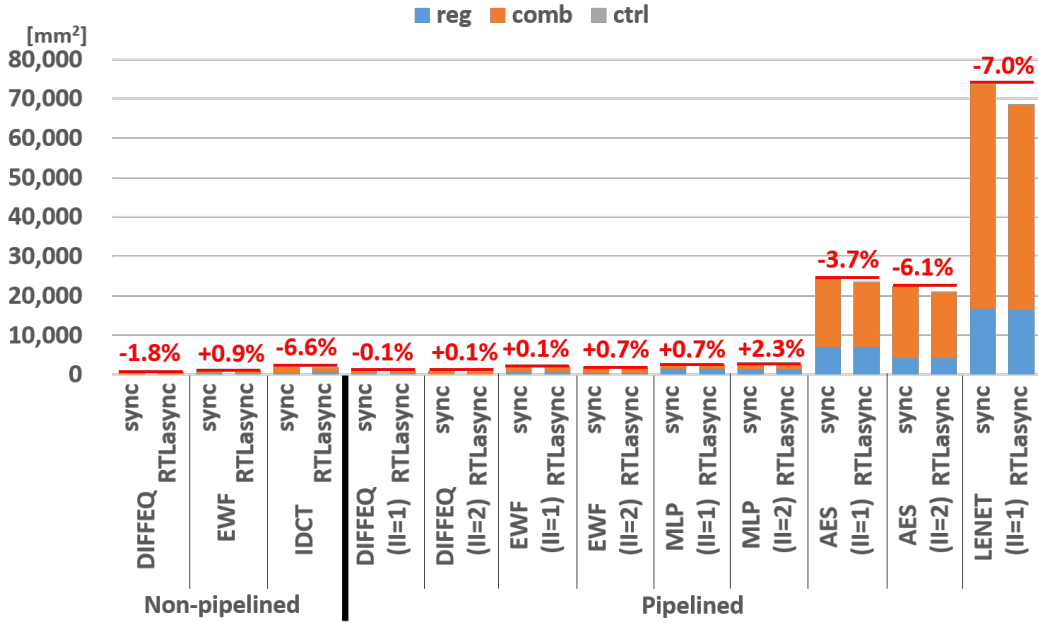
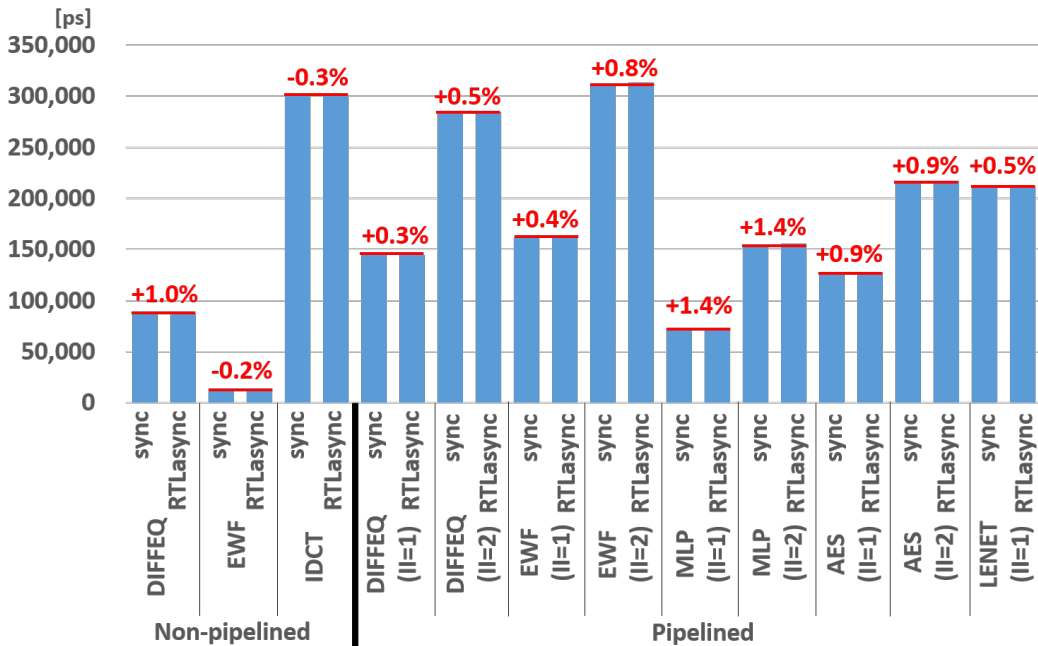


(b) Waveforms of synchronous GL netlist and asynchronous GL netlist.

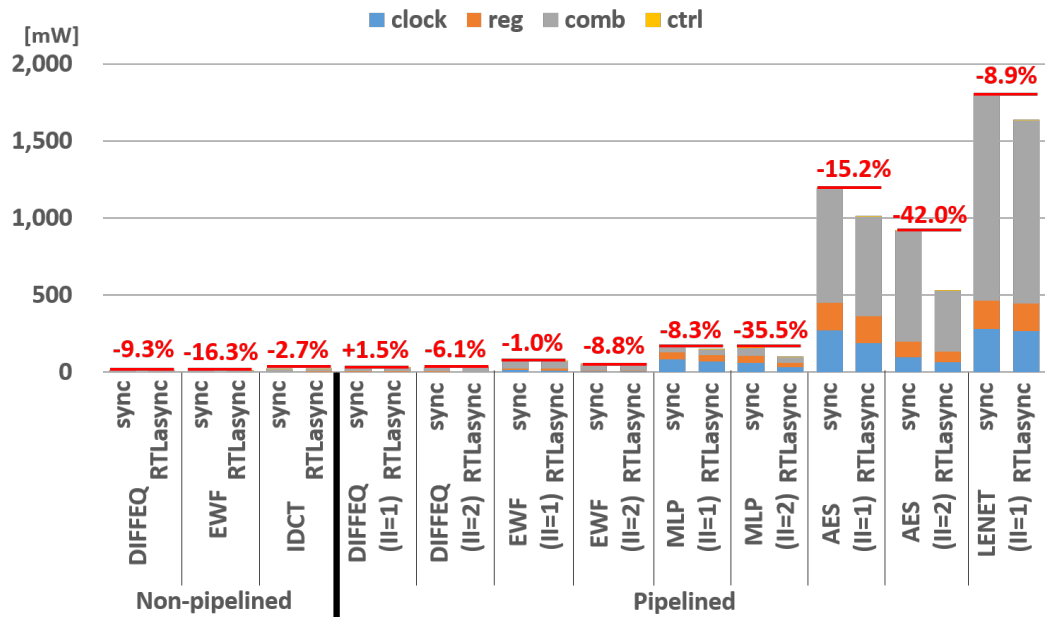
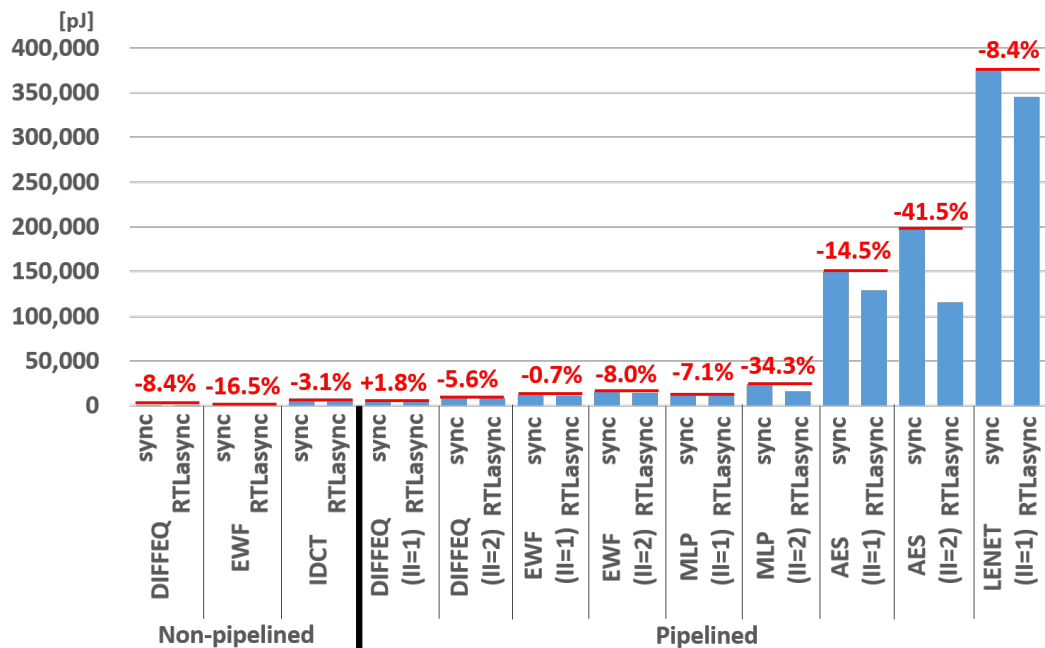
Figure 6.4: Waveforms of EWFs for non-pipelined circuits.

times of RTL_{async} and RTL_{async}_m was very small between -0.6% and $+0.3\%$ because we assigned the same values for the maximum delay constraints. Compared with RTL_{async} , the energy consumption of RTL_{async}_m was reduced because of the direct effect of the reduction of dynamic power consumption.

Figure 6.10 shows the evaluation results regarding the use of appropriate DFFs. Figures 6.10a-d show the circuit area, execution time, dynamic power consumption, and energy consumption, respectively. Compared with RTL_{async} , the circuit area and dynamic power consumption of RTL_{async}_r were reduced because of the use of

Figure 6.5: Circuit areas of *sync* and *RTLAsync*.Figure 6.6: Execution times of *sync* and *RTLAsync*.

DFFs without an enable signal instead of DFFs with an enable signal. $RTLAsync_r$ is useful for circuits including many DFFs with an enable signal such as IDCT. The difference between the execution times of $RTLAsync$ and $RTLAsync_r$ was very small between -0.2% and $+0.1\%$ because we assigned the same values for the maximum delay constraints. The energy consumption of $RTLAsync_r$ was reduced compared with that of $RTLAsync$. This result is attributed to the direct effect of the reduction in

Figure 6.7: Dynamic power consumptions of *sync* and *RTLAsync*.Figure 6.8: Energy consumptions of *sync* and *RTLAsync*.

dynamic power consumption.

Figures 6.11 and 6.12 show the circuit area and execution time of operand isolation by latches. Table 6.2 lists the number of inserted D latches. Compared with *RTLAsync*, the circuit areas of *RTLAsync_{op}* and *RTLAsync_{op10}* were increased because of the insertion of D latches. Since the number of the inserted D latches increased by adding the margin, the circuit area of *RTLAsync_{op10}* was increased compared to that *RTLAsync_{op}*. Moreover, the difference between the execution times of *RTLAsync* and *RTLAsync_{op}*

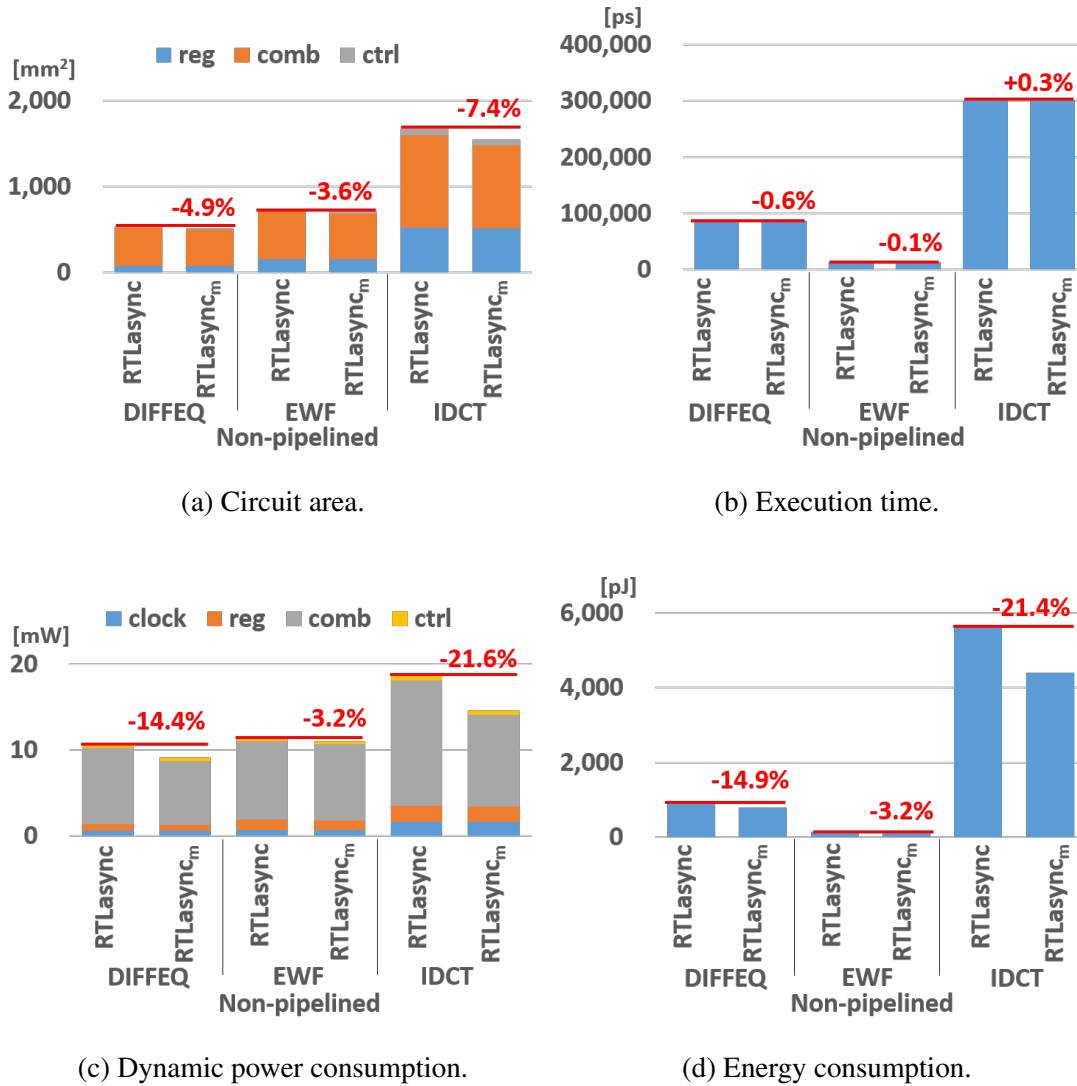


Figure 6.9: Evaluation of the modularization for data-path resources.

was very small between -0.1% and $+1.2\%$ because the proposed method did not insert D latches in the critical paths. In contrast, the execution time of $RTLasync_{op10}$ was increased by adding the margin.

Figures 6.13 and 6.14 show the dynamic power consumption and energy consumption of operand isolation by latches. Compared with $RTLasync$, except for MLP, the dynamic power consumption of $RTLasync_{op}$ and $RTLasync_{op10}$ was reduced because of the insertion of D latches to prevent unnecessary operations. Therefore, operand isolation by latches is useful for data-path resources that have a sufficient bit width. In our future research, we plan to modify the latch insertion algorithm by considering the bit width and power consumption of resources. Compared with $RTLasync$, except for MLP, the energy consumption of $RTLasync_{op}$ and $RTLasync_{op10}$ was reduced because of the decrease in the dynamic power consumption.

Figures 6.15, 6.16, 6.17, and 6.18 show the circuit area, execution time, dynamic power consumption, and energy consumption of the conversion from DFFs into D latches. Compared with $RTLasync_l$, the dynamic power consumption of $RTLasync_l$ was reduced in all circuits because of the use of D latches instead of DFFs. Similarly, the circuit area of $RTLasync_l$, except for AES, was reduced. The difference

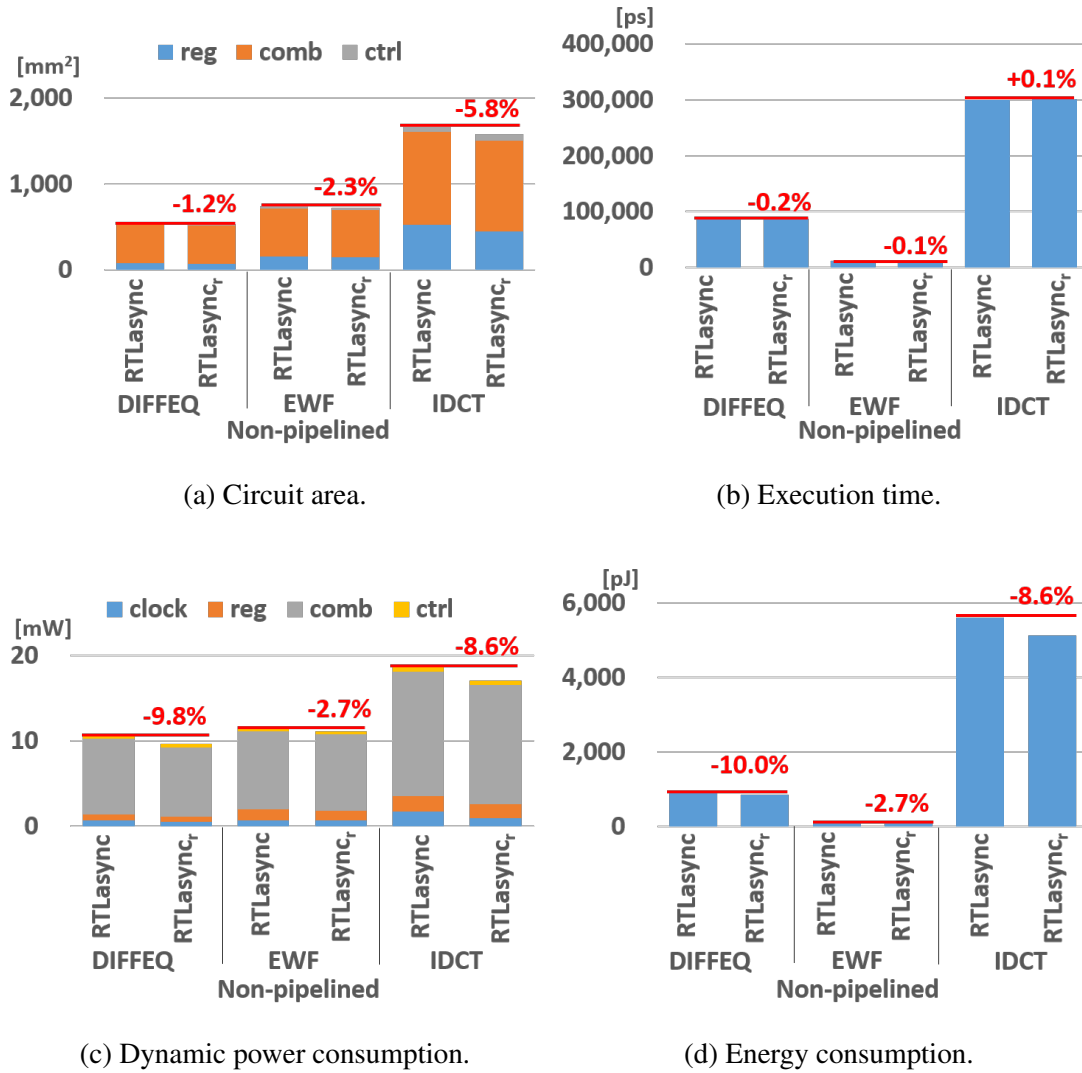


Figure 6.10: Evaluation of the use of appropriate DFFs.

Table 6.2: Number of inserted D latches.

Name	Type	Number of inserted D latches
DIFFEQ (non-pipelined circuits)	<i>RTLasync_{op}</i>	0
	<i>RTLasync_{op10}</i>	1
EWF (non-pipelined circuits)	<i>RTLasync_{op}</i>	3
	<i>RTLasync_{op10}</i>	7
IDCT (non-pipelined circuits)	<i>RTLasync_{op}</i>	1
	<i>RTLasync_{op10}</i>	1
DIFFEQ (pipelined circuits) (II=2)	<i>RTLasync_{op}</i>	5
	<i>RTLasync_{op10}</i>	6
EWF (non-pipelined circuits) (II=2)	<i>RTLasync_{op}</i>	12
	<i>RTLasync_{op10}</i>	20
MLP (non-pipelined circuits) (II=2)	<i>RTLasync_{op}</i>	177
	<i>RTLasync_{op10}</i>	356
AES (non-pipelined circuits) (II=2)	<i>RTLasync_{op}</i>	33
	<i>RTLasync_{op10}</i>	75

between the execution times of *RTLasync* and *RTLasync_i* was small between -2.4%

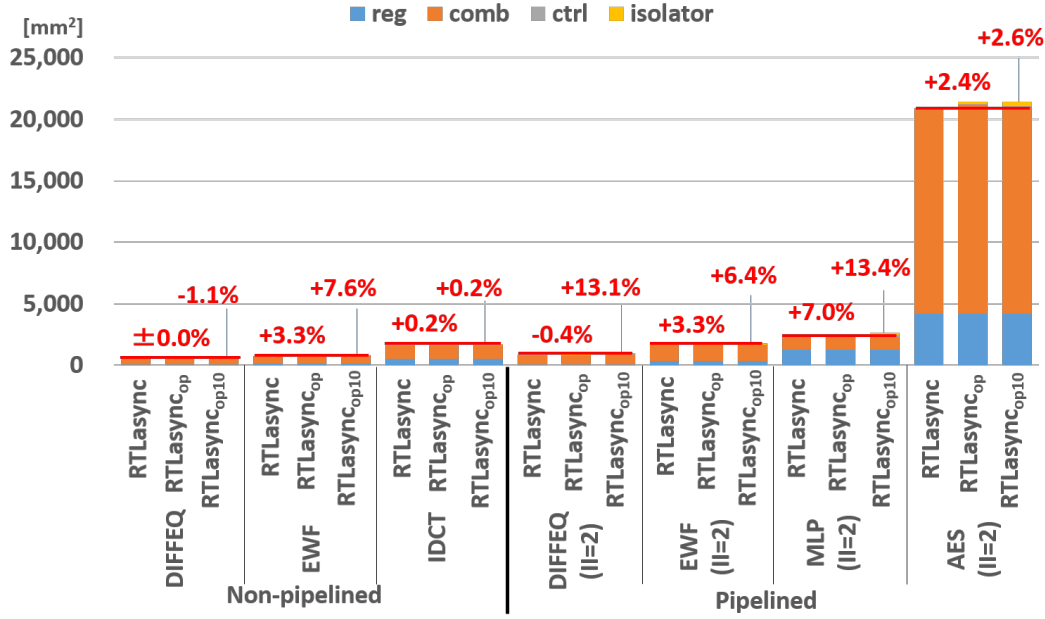


Figure 6.11: Circuit area of operand isolation by latches.

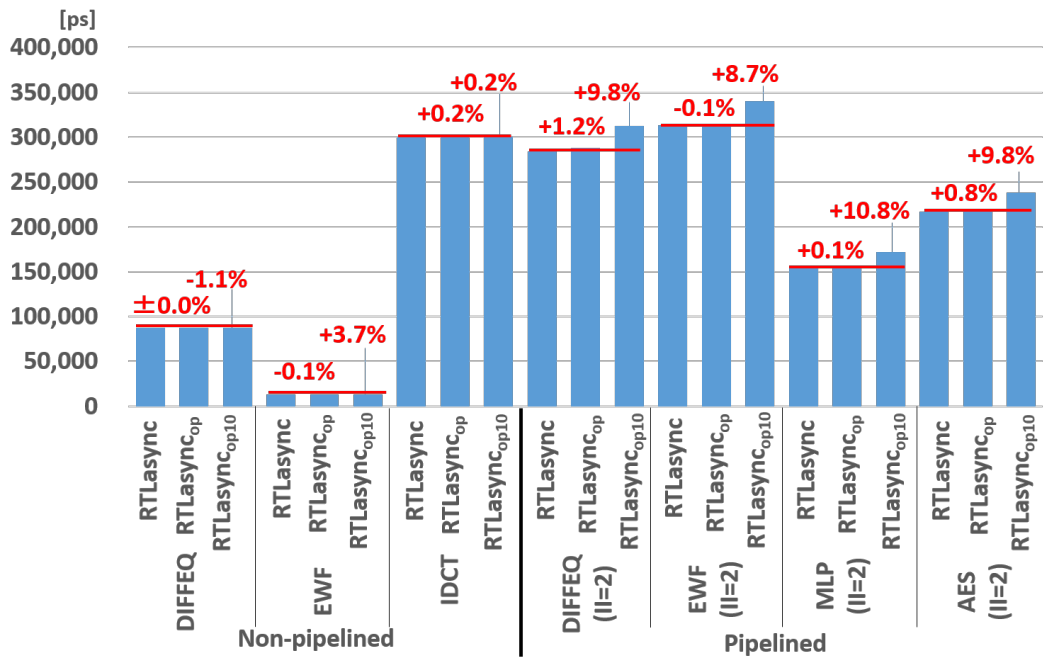


Figure 6.12: Execution time of operand isolation by latches.

and +0.2% because we assigned the same values for the maximum delay constraints and local clock constraints. Compared with *RTLasync*, the energy consumption of *RTLasync_i* was reduced. This result was attributed to the direct effect of the reduction in dynamic power consumption. Therefore, the conversion from DFFs into D latches is useful for circuits with many registers such as non-pipelined IDCT and pipelined MLP, AES, and LENET.

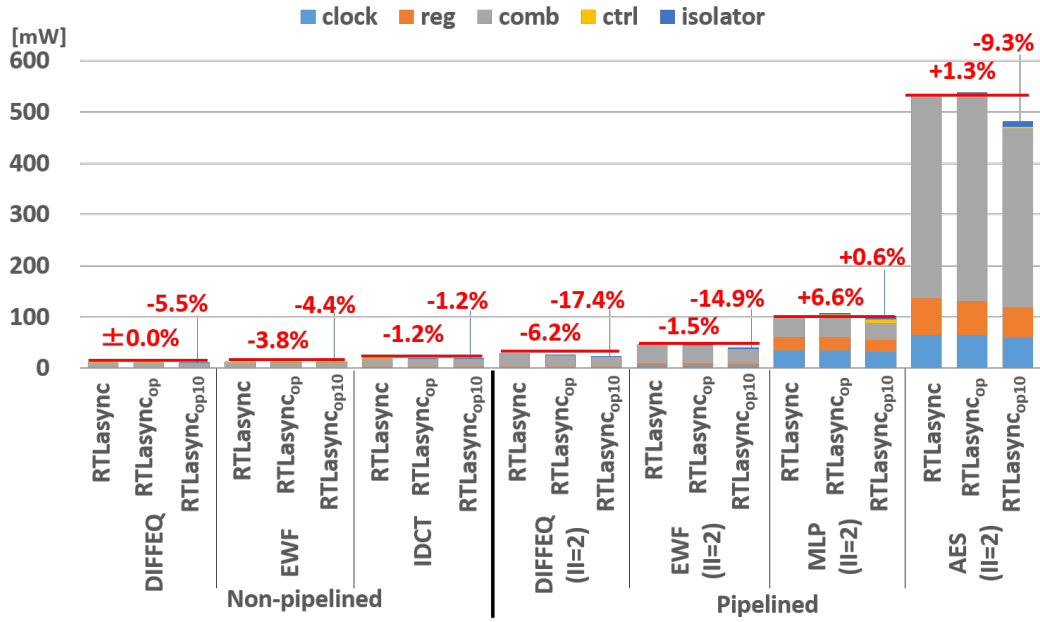


Figure 6.13: Dynamic power consumption of operand isolation by latches.

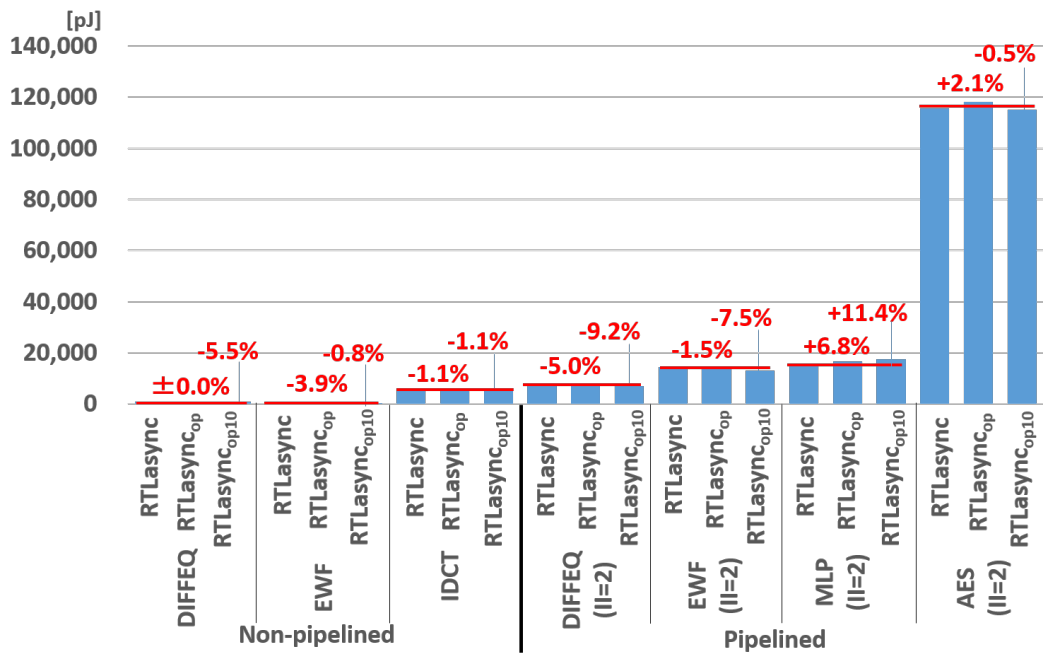


Figure 6.14: Energy consumption of operand isolation by latches.

Figures 6.19, 6.20, 6.21, and 6.22 show the circuit area, execution time, dynamic power consumption, and energy consumption of the combination of the optimization methods. Compared to each optimization method, RTL_{async}_a was found to achieve the best energy reduction for non-pipelined DIFFEQ and IDCT and pipelined DIFFEQ, EWF, and AES. In contrast, the energy consumption of pipelined MLP was increased because the bit-level operand isolation could not reduce the dynamic power consump-

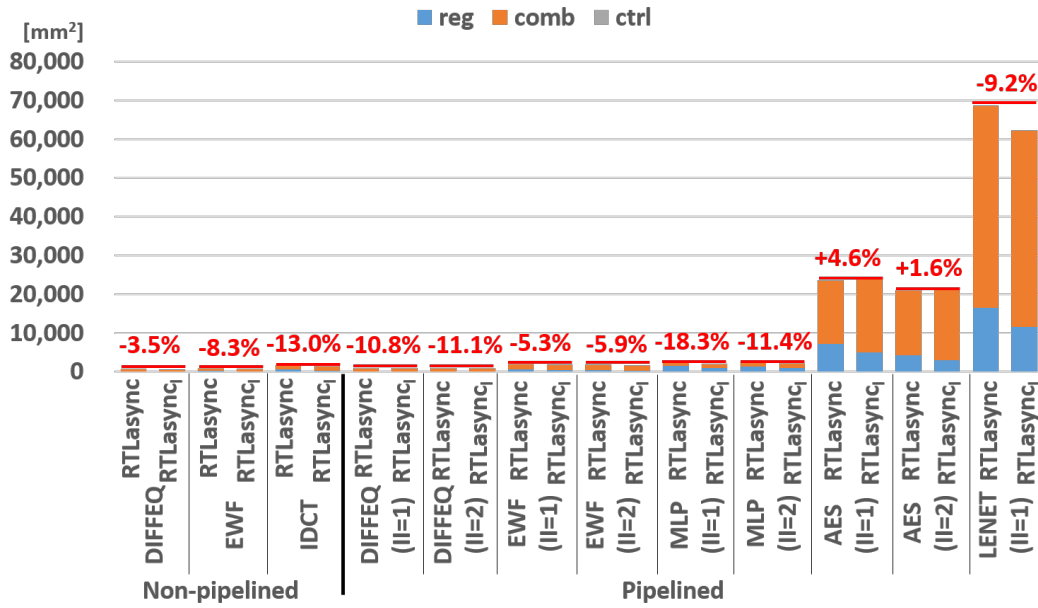


Figure 6.15: Circuit area of the conversion from DFFs into D latches.

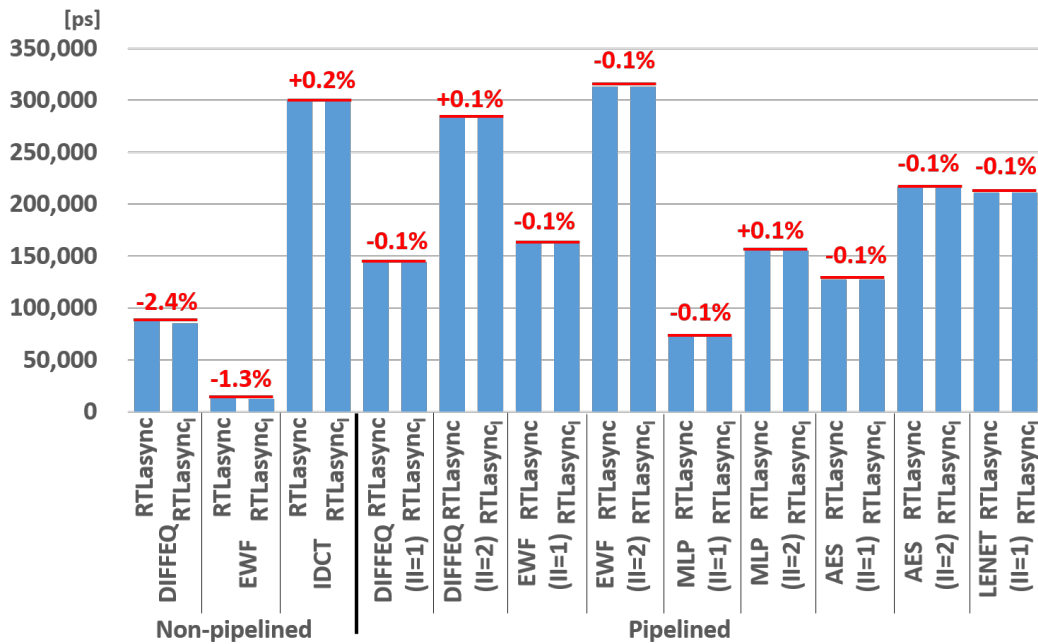


Figure 6.16: Execution time of the conversion from DFFs into D latches.

tion.

6.3.3 Comparison of the RTL Conversion Method and the GL Conversion Method

Figure 6.23 shows the comparison between *GLasync* and *RTLasync* in terms of the execution times. *RTLasync* reduced the execution time in non-pipelined circuits

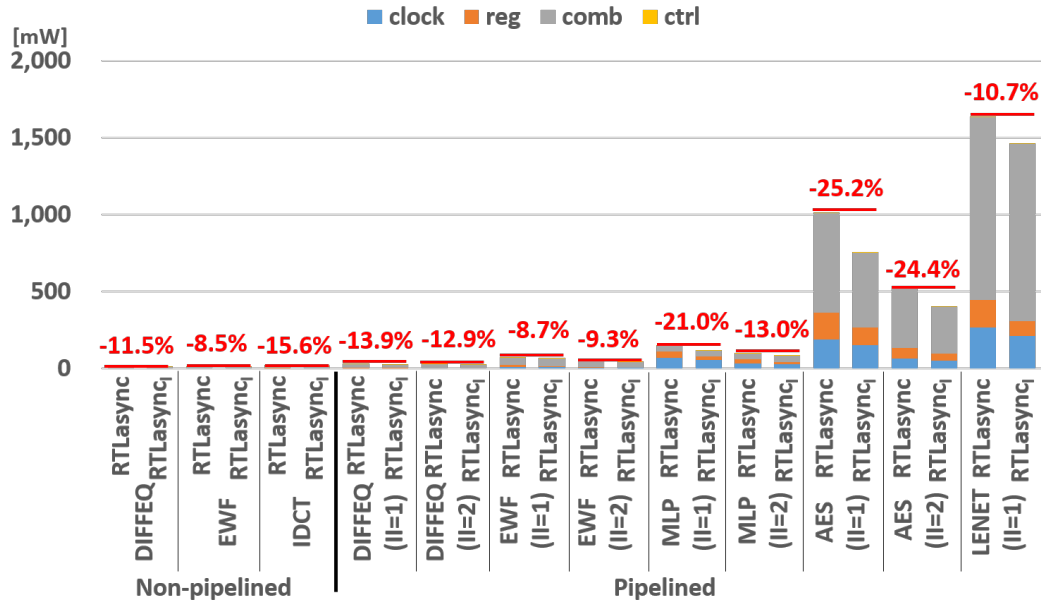


Figure 6.17: Dynamic power consumption of the conversion from DFFs into D latches.

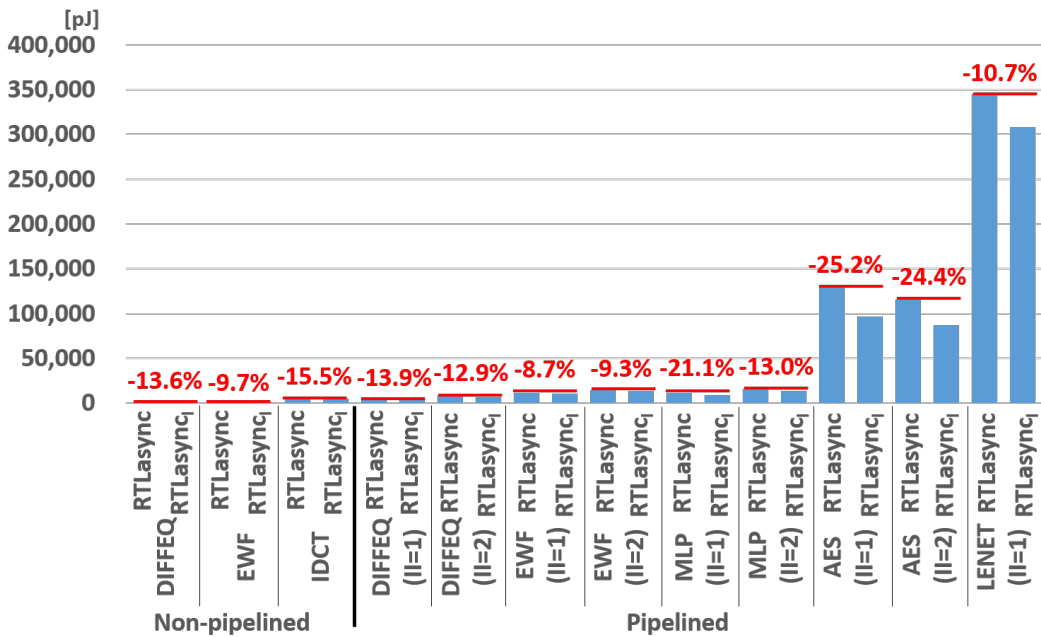


Figure 6.18: Energy consumption of the conversion from DFFs into D latches.

and pipelined circuits where II was one cycle. Further, in non-pipelined circuits, the critical path delays from control modules to registers through multiplexers were reduced. In pipelined circuits where II was one cycle, the critical path delays were reduced by assigning strict local clock constraints. In pipelined circuits where II was two cycles, the execution time of *RTLAsync* was increased because the critical path delays were increased via operand isolation.

Figure 6.24 shows the comparison between *GLAsync* and *RTLAsync* terms of the

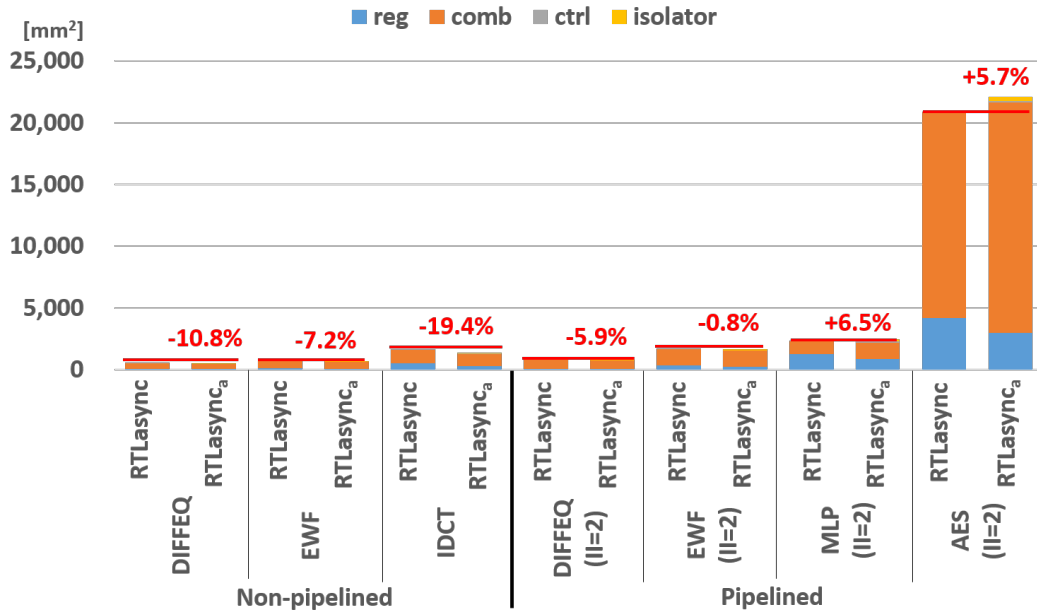


Figure 6.19: Circuit area of the combination of the optimization methods.

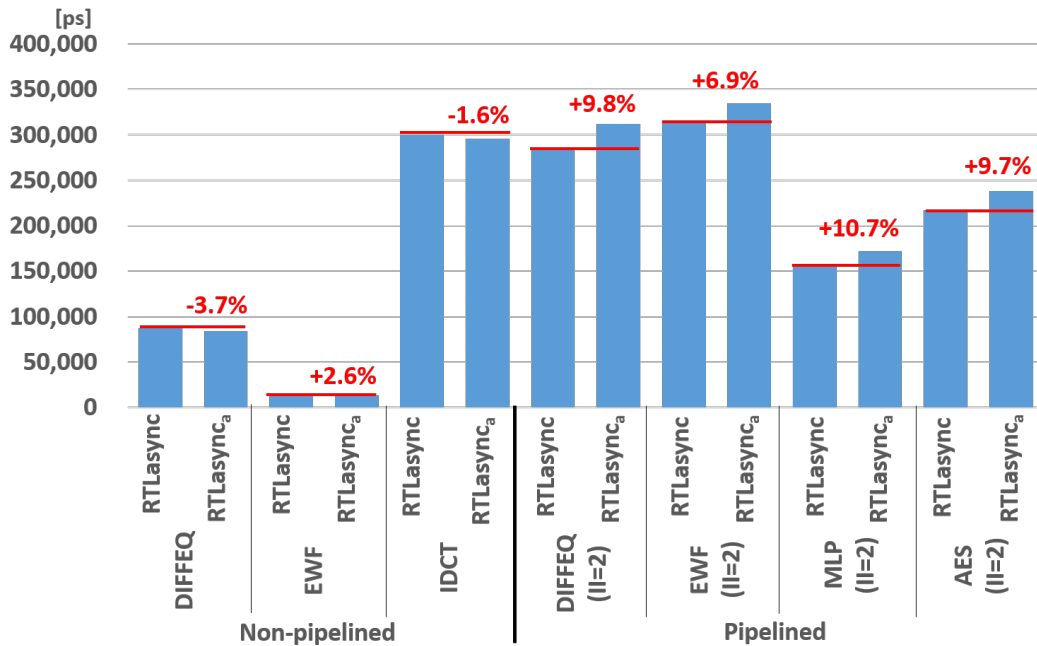


Figure 6.20: Execution time of the combination of the optimization methods.

circuit areas. *RTLasync* reduced the circuit area in non-pipelined circuits because the area of the combinational circuits was reduced by assigning loose maximum delay constraints for operations that use high-power resources (e.g., multipliers). In pipelined circuits besides MLP and LENET where II was one cycle, the circuit area of *RTLasync* was increased. In pipelined circuits where II was one cycle, the area of the combinational circuits was increased by assigning strict local clock constraints. In pipelined circuits where II was two cycles, the circuit area was increased via operand isolation.

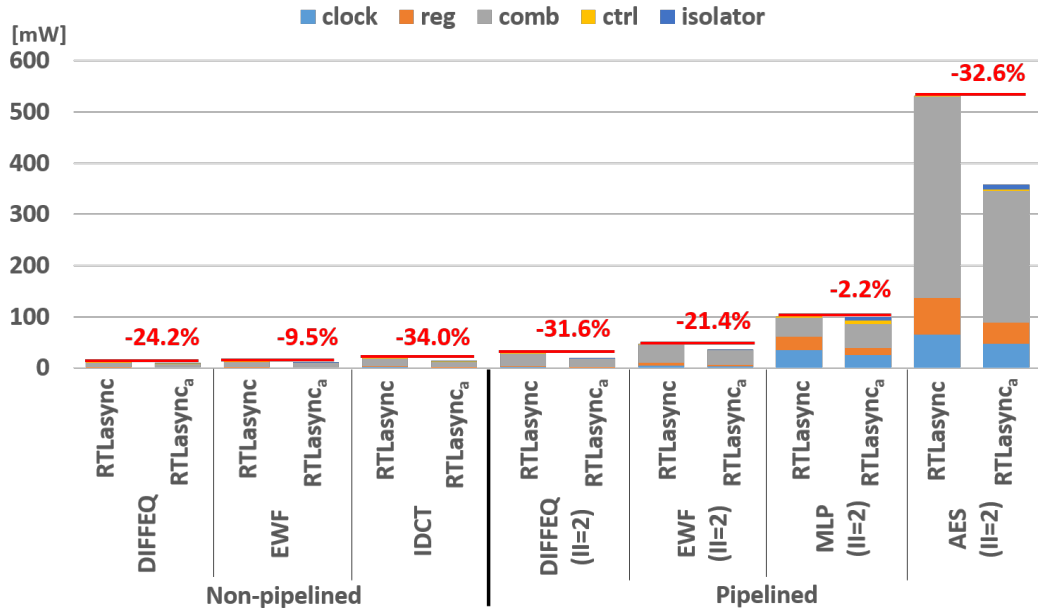


Figure 6.21: Dynamic power consumption of the combination of the optimization methods.

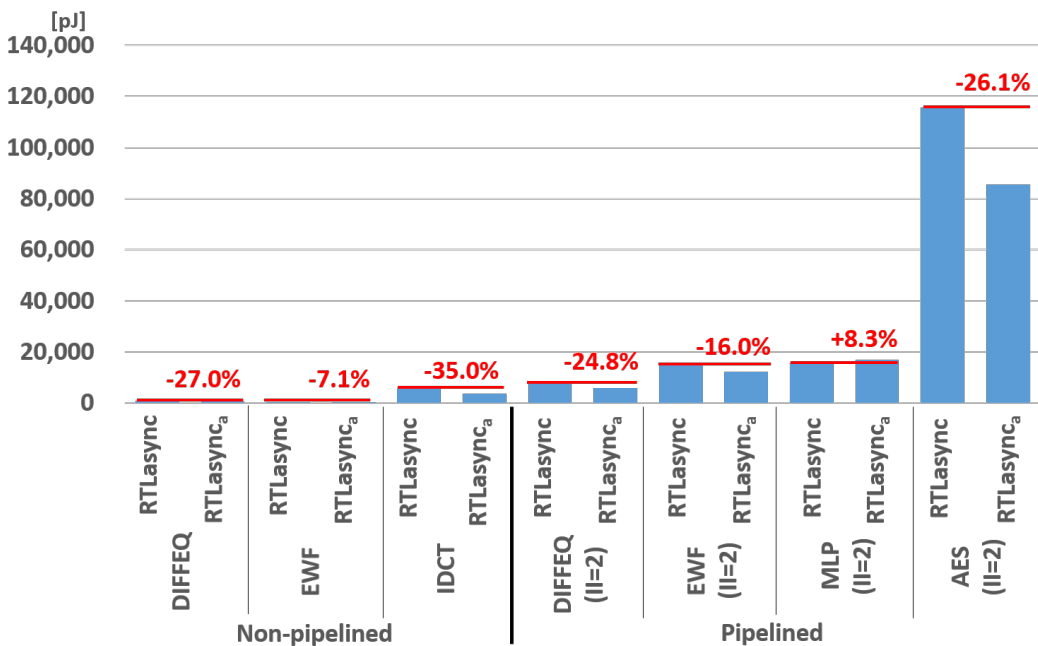


Figure 6.22: Energy consumption of the combination of the optimization methods.

Figure 6.25 shows the comparison between *GLasync* and *RTLasync* in terms of the dynamic power consumption. *RTLasync* reduced the dynamic power consumption in non-pipelined circuits and pipelined circuits where II was two cycles, except in MLP. In non-pipelined circuits, the dynamic power consumption of the combinational circuits was reduced by assigning loose maximum delay constraints for operations that use high power resources. In pipelined circuits where II was two cycles, except in MLP, the

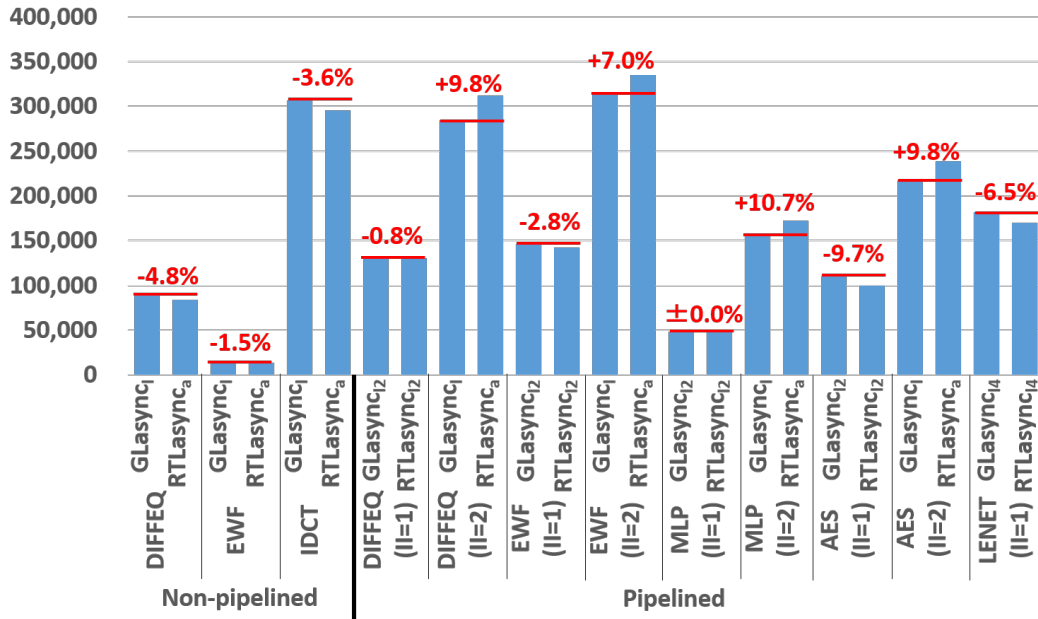


Figure 6.23: Execution times of *GLasync* and *RTLasync*.

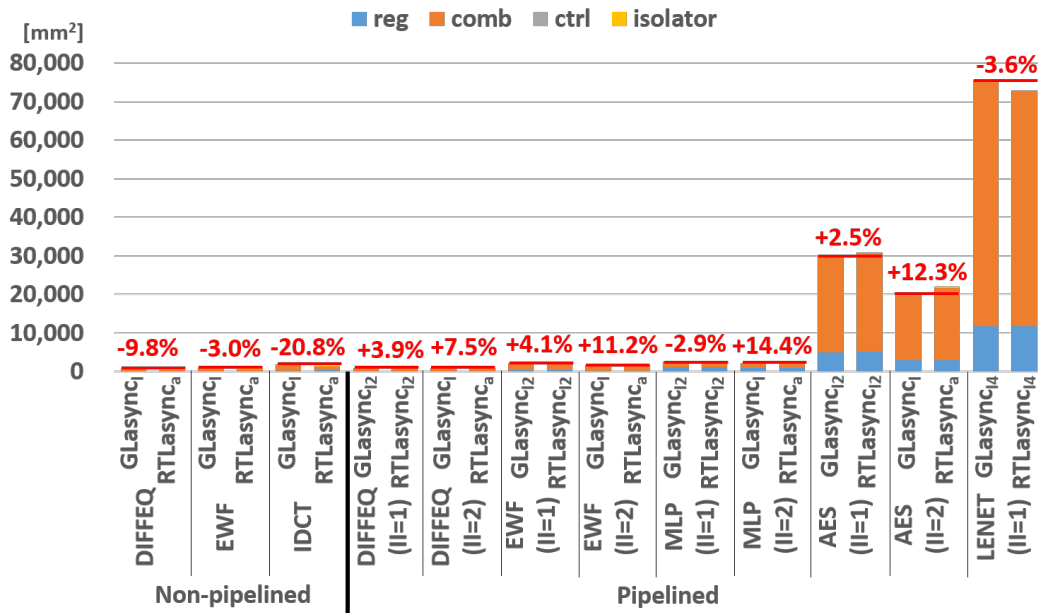
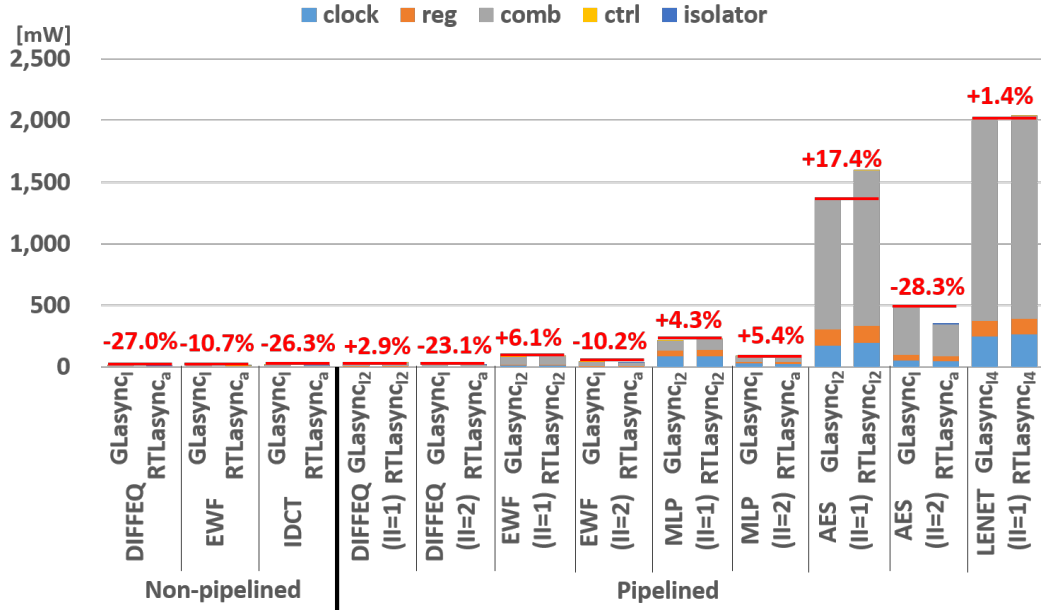
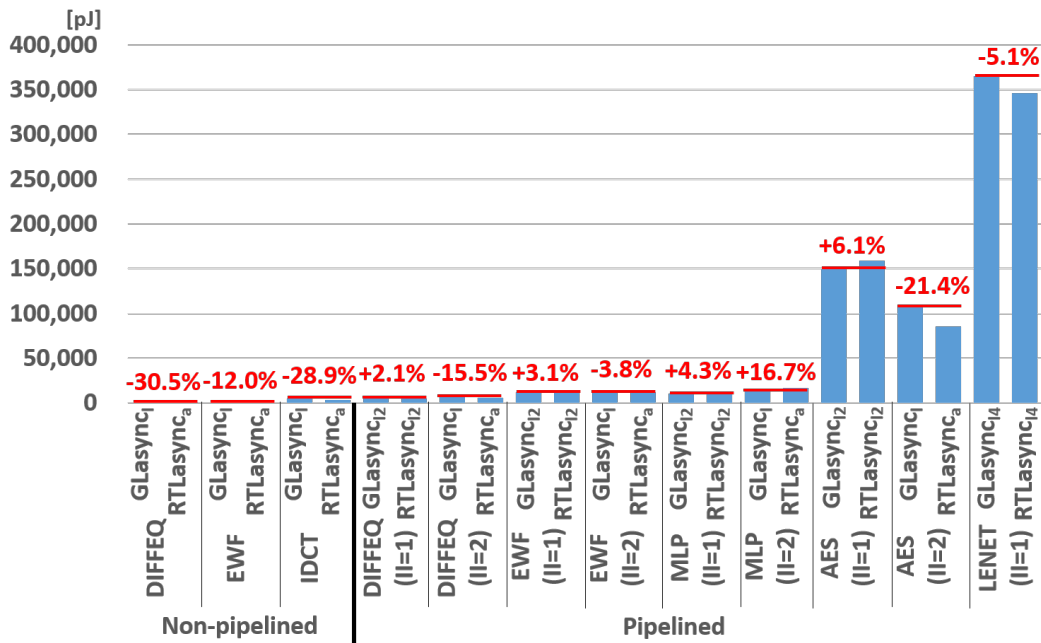


Figure 6.24: Circuit areas of *GLasync* and *RTLasync*.

dynamic power consumption of the combinational circuits was reduced by operand isolation. In pipelined circuits where II was one cycle, the dynamic power consumption of *RTLasync* was increased because the execution time was reduced.

Figure 6.26 shows the comparison between *GLasync* and *RTLasync* in terms of the energy consumption. *RTLasync* reduced the dynamic power consumption in non-pipelined circuits and pipelined circuits where II was two cycles, except for MLP. This result is ascribed to the direct influence of the reduction in dynamic power consumption. Moreover, in pipelined LENET, *RTLasync* reduced the dynamic power consumption.

Figure 6.25: Dynamic power consumptions of *GLasync* and *RTLasync*.Figure 6.26: Energy consumptions of *GLasync* and *RTLasync*.

This result is ascribed to the direct influence of the reduction in execution time.

6.3.4 Discussion

Energy consumption is reduced from synchronous RTL models to achieve RTL conversion without optimization methods. This result is attributed to the direct effect of the reduction in the dynamic power consumption caused by the use of loose maximum de-

```

DFF ¥s_reg_53_reg[0] (.RB (n_5), .CLK (rc_gclk_8039),
.DATA (cc[0]), .Q (n_539));
BUF g27396(.A (n_539), .Y (n_7390));
NOR2 mul_811_65_g27397(.A (n_7392),
.BB (mul_811_65_n_916), .YB (n_7393));
BUF g27398(.A (n_7585), .Y (n_7392));
XNOR2 mul_811_65_g27400(.A (n_7585),
.B (mul_811_65_n_916), .YB (n_7394));
NAND3 g27421(.A (n_7582), .B (n_7414), .C (n_7758),
.YB (n_7416));
OA21 g27423(.A0 (n_7282), .A1 (n_3103), .B (n_7290),
.Y (n_7414));

```

To insert operand isolators difficulty

Figure 6.27: Example of GL netlists.

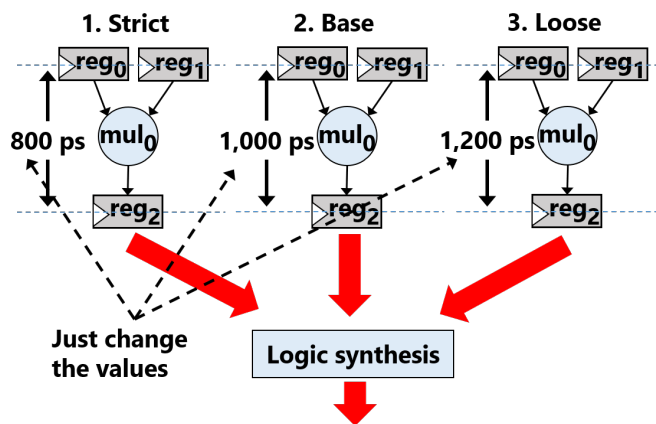


Figure 6.28: Logic synthesis for the RTL conversion.

lay constraints for non-pipelined circuits and the local clock constraints for pipelined circuits.

Optimized asynchronous RTL models are obtained from RTL conversion with optimization methods. The modularization and use of appropriate DFFs optimize circuit area and dynamic power consumption. The operand isolation is useful for reducing dynamic power consumption if the operations have a sufficient bit width. The use of D latches is the best effect for optimization. The combination of all optimization methods results in the best optimization effect in many cases.

The comparison between the GL conversion and RTL conversion that the RTL conversion reduces energy consumption in many cases. In addition, compared with the GL conversion, the RTL conversion allows designers to insert operand isolation easily. For example, it is difficult to insert isolators in the GL conversion because the wire names are changed and resources are replaced with gates (Fig. 6.27). Further, the RTL conversion allows designers to explore optimum circuits by changing constraints used for logic synthesis from strict to loose (Fig. 6.28). Moreover, the RTL conversion allows designers to implement asynchronous circuits on FPGAs easily.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation, we proposed a method for automatic conversion method from synchronous RTL models into asynchronous RTL models with bundled-data implementation. The proposed method generates an intermediate representation from a given synchronous RTL model. Subsequently, the proposed method generates an asynchronous RTL model with bundled-data implementation from the intermediate representation. In addition to generating asynchronous RTL models, the proposed method generates an asynchronous RTL simulation model for FPGA implementations and non-optimization constraints to prevent optimizations for primitive cells used in the control circuit.

We proposed four optimization methods during the RTL conversion to obtain high-quality asynchronous circuits: (1) the modularization for data-path resources to reduce the area of data-path circuits; (2) the use of appropriate DFFs to reduce the area of registers; (3) inserting latches before data-path resources to reduce the dynamic power consumption of data-path circuits; and (4) conversion from DFFs into D latches to reduce the dynamic power consumption of registers.

In the experiment, we converted synchronous RTL models into asynchronous RTL models with bundled-data implementation. We demonstrated that the conversion time depends on the size and the number of states (pipeline stages) in the synchronous RTL model. Thereafter, we performed logic synthesis for the converted asynchronous RTL models to compare them with the synchronous circuits and asynchronous circuits obtained from GL conversion. The asynchronous circuits obtained from the proposed RTL conversion reduced the energy consumption compared with that when using synchronous circuits. Moreover, the combination of the optimization methods could reduce more energy consumption in many cases. The proposed RTL conversion reduced the energy consumption compared with the GL conversion in many cases. Compared with the GL conversion, the RTL conversion allows designers to insert operand isolation easily, explore optimum circuits by changing constraints used for logic synthesis from strict to loose, and implement asynchronous circuits on FPGAs easily.

7.2 Future Work

As future research, we plan to extend the proposed method to incorporate various HLS tools. We will convert asynchronous circuits with low energy consumption from

various synchronous RTL models by increasing the Verilog HDL syntax that can be solved. In addition, we will modify the latch insertion algorithm by considering the bit width and the power consumption of resources. Moreover, we will propose optimization methods to reduce the number of switching for the combinational circuits.

References

- [1] J. Cortadella et al., "Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications," *IEEE TCAD*, vol. 25, pp. 1904–1921, 2006.
- [2] I. Blunno et al., "Handshake protocols for de-synchronization," *Proc. ASYNC*, pp. 149–158, 2004.
- [3] N. Andrikos et al., "A Fully-Automated Desynchronization Flow for Synchronous Circuits," *Proc. DAC*, pp. 982–985, 2007.
- [4] S. K. Srinivasan et al., "Desynchronization: Design For Verification," *Proc. FMCAD*, pp. 215–222, 2011.
- [5] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," *Proc. International Symposium on the Theory of Switching*, pp. 204–243, 1959.
- [6] A. Branover et al., "Asynchronous Design By Conversion: Converting Synchronous Circuits into Asynchronous Ones," *Proc. DATE*, pp. 870–875, 2004.
- [7] Y. Zhang, "Challenges in Building An Open-source Flow from RTL to Bundled-Data Design," *Proc. ASYNC*, 2018.
- [8] A. Peeters et al., "Click Elements: An Implementation Style for Data-Driven Compilation," *Proc. ASYNC*, pp. 3–14, 2010.
- [9] M. Lighthart et al., "Asynchronous Design Using Commercial HDL Synthesis Tools," *Proc. ASYNC*, pp. 114–125, 2000.
- [10] A. Kondratyev and K. Lwin, "Design of Asynchronous Circuits by Synchronous CAD Tools," *Proc. DAC*, pp.411–414, 2002.
- [11] R. B. Reese et al., "Uncle - An RTL Approach to Asynchronous Design," *Proc. ASYNC*, pp. 65–72, 2012.
- [12] M. L. L. Sartori et al., "A Frontend using Traditional EDA Tools for the Pulsar QDI Design Flow," *Proc. ASYNC*, pp. 3–10, 2020.
- [13] M. L. L. Sartori et al., "Pulsar: Constraining QDI Circuits Cycle Time Using Traditional EDA Tools," *Proc. ASYNC*, pp. 114–123, 2019.
- [14] R. Zhou et al., "Quasi-Delay-Insensitive Compiler: Automatic Synthesis of Asynchronous Circuits from Verilog Specifications," *Proc. NWSCAS*, pp. 1–4, 2011.
- [15] J. Oberg et al., "Automatic Synthesis of Asynchronous Circuits from Synchronous RTL Description," *Proc. NORCHIP*, pp. 200–205, 2005.

-
- [16] J. Cortadella et al., "Synthesis of Synchronous Elastic Architectures," Proc. DAC, pp. 657–662, 2006.
- [17] H Wu, "A method to transform synchronous pipeline circuits to bundled-data asynchronous circuits using commercial EDA tools," IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC), 2019.
- [18] P. A. Beerel et al., "Proteus: An ASIC Flow for GHz Asynchronous Designs," IEEE Design & Test of Computers, vol. 28, pp. 36–51, 2011.
- [19] P. A. Beerel et al., "Slack matching asynchronous designs," Proc. ASYNC, pp. 184–194, 2006.
- [20] A. Taubin et al., "Design Automation of Real-Life Asynchronous Devices and Systems, Chapter 2," pp. 19–39, 2007.
- [21] K. Garcia et al., "Synthesis of locally-clocked asynchronous systems with bundled-data implementation on FPGAs," Proc. SPL, pp.1–6, 2014.
- [22] K. Y. Yun and D. L. Dill, "Automatic Synthesis of Extended Burst-Mode Circuits: Part I (Specification and Hazard-Free Implementations)," IEEE TCAD, vol.18, no.2, pp.101–117, 1999.
- [23] T. Curtinhas et al., "VHDLASYN: A tool for synthesis of asynchronous systems from of VHDL behavioral specifications," Proc PRIME-LA, pp.1–4, 2018.
- [24] M. Sacker et al., "A General Purpose Behavioural Asynchronous Synthesis System," Proc. ASYNC, pp.125–134, 2004.
- [25] L. Josipovic, "Dynamically Scheduled High-level Synthesis," Proc. FPGA, 127–136, 2018.
- [26] K. Yoshimi and H. Saito, "A delay Adjustment Method for Asynchronous Circuits with Bundled-data Implementation Considering a Latency Constraint," Proc. SASIMI, pp.219–224, 2016.
- [27] J. Takizawa et al., "A Design Support Tool Set for Asynchronous Circuits with Bundled-data Implementation on FPGAs," Proc. FPL, pp.232–235, 2014.
- [28] E. U. Rosenberger et al., "Q-Modules: internally clocked delay-insensitive modules," IEEE TC, vol. C-37, no. 9, pp. 1005–1018, 1988.
- [29] M. Gibiluka, "A Bundled-Data Asynchronous Circuit Synthesis Flow Using a Commercial EDA Framework," Proc. Euromicro Conference on Digital System Design, pp.79–86, 2015.
- [30] K. Stevens et al., "Relative Timing," Proc. ASYNC, pp.208–218, 1999.
- [31] 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.

- [32] S. Semba et al., "Optimization Methods during RTL Conversion from Synchronous RTL Models to Asynchronous RTL Models," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E103-A, No. 12, pp. 1417–1426, 2020.
- [33] 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.
- [34] M. Munch, B. Wurth, R. Mehra, J. Sproch, and N. Wehn, "Automating RT-Level Operand Isolation to Minimize Power Consumption in Datapaths," *Proc. DATE*, pp.624–631, 2000.
- [35] Y. Umuroglu et al., "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, pp. 65–74, 2017.
- [36] 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.
- [37] Y. LeCun et al., "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [38] Cadence, "Stratus High-Level Synthesis User Guide," product version 18.1, 2018.

List of Publications

Academic Journals (Referred)

Major Journals

1. Shogo Semba and Hiroshi 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, July, 2019.
2. Shogo Semba, Hiroshi Saito, Masato Tatsuoka, and Katsuya Fujimura, "Optimization Methods during RTL Conversion from Synchronous RTL Models to Asynchronous RTL Models," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E103-A, No. 12, pp. 1417–1426, December, 2020.

Proceedings at International Conferences (Referred)

Major Conferences

1. Shogo Semba and Hiroshi Saito, "Comparison of RTL Conversion and GL Conversion from Synchronous Circuits to Asynchronous Circuits," *Proc. 2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May, 2019.

Non-Major Conferences

1. Shogo Semba and Hiroshi Saito, "A Study on the Optimization of Asynchronous Circuits During RTL Conversion from Synchronous Circuits," *Proc. The 22nd Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2019)*, pp.274–279, October, 2019.
2. Shogo Semba and Hiroshi Saito, "Study on an RTL Conversion Method from Pipelined Synchronous RTL Models into Asynchronous RTL Models," *Proc. The 23rd Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2021)*, pp.229–234, March, 2021.